HeatWave User Guide

Abstract

This document describes how to use HeatWave. It covers how to load data, run queries, optimize analytics workloads, and use HeatWave machine learning capabilities.

For information about creating and managing a HeatWave Cluster on Oracle Cloud Infrastructure (OCI), see *HeatWave on OCI Service Guide*.

For information about creating and managing a HeatWave Cluster on Amazon Web Services (AWS), see *HeatWave on AWS Service Guide*.

For information about creating and managing a HeatWave Cluster on Oracle Database Service for Azure (ODSA), see *HeatWave for Azure Service Guide*.

For MySQL Server documentation, refer to the *MySQL Reference Manual*.

For information about the latest HeatWave features and updates, refer to the HeatWave Release Notes.

For legal information, see the Legal Notices.

For help with using MySQL, please visit the MySQL Forums, where you can discuss your issues with other MySQL users.

Document generated on: 2024-10-04 (revision: 79912)

Table of Contents

Preface and Legal Notices

This is the user manual for HeatWave.

Legal Notices

License Restrictions

Warranty Disclaimer

Restricted Rights Notice

Hazardous Applications Notice

Trademark Notice

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at
http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc.

Access to Oracle Support for Accessibility

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit
http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info or visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs` if you are hearing impaired.

Chapter 1 Overview

Table of Contents

HeatWave is a massively parallel, high performance, in-memory query accelerator that accelerates MySQL performance by orders of magnitude for analytics workloads, mixed workloads, and machine learning. HeatWave can be accessed through Oracle Cloud Infrastructure (OCI), Amazon Web Services (AWS), and Oracle Database Service for Azure (ODSA).

HeatWave consists of a MySQL DB System and HeatWave nodes. Analytics queries that meet certain prerequisites are automatically offloaded from the MySQL DB System to the HeatWave Cluster for accelerated processing. With a HeatWave Cluster, you can run online transaction processing (OLTP), online analytical processing (OLAP), and mixed workloads from the same MySQL database without requiring extract, transfer, and load (ETL), and without modifying your applications. For more information about the analytical capabilities of HeatWave, see Chapter 2, *HeatWave MySQL*.

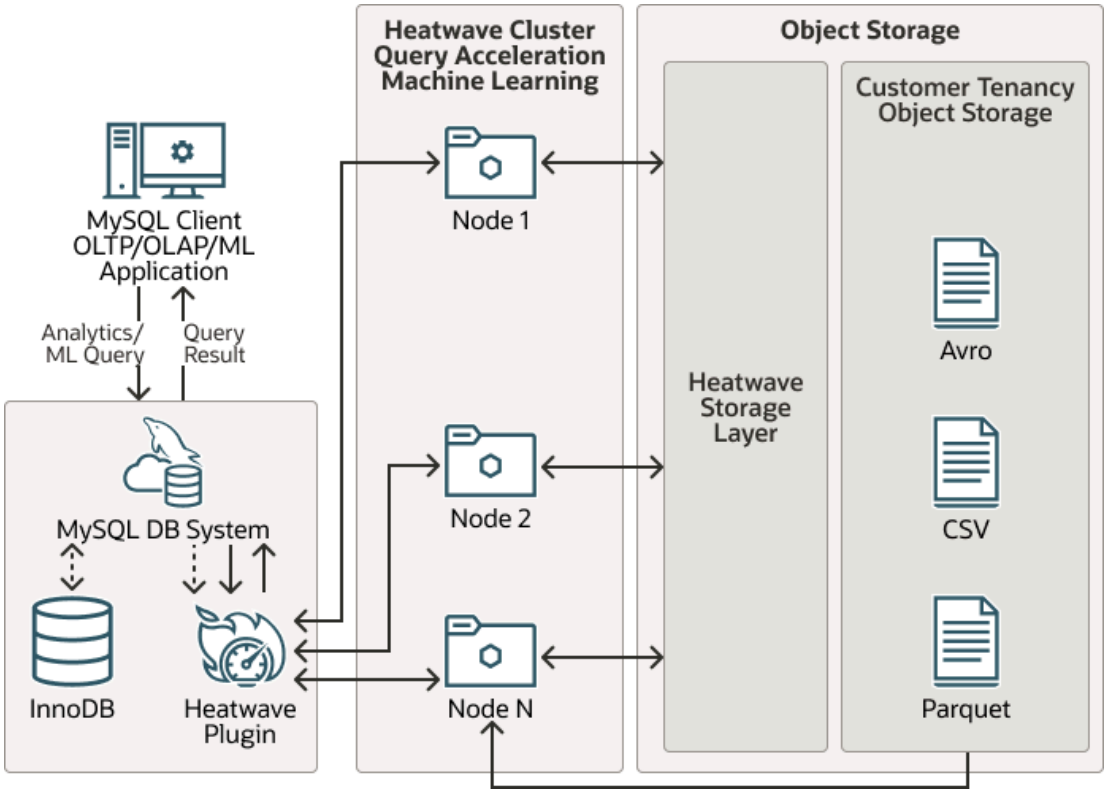
The MySQL DB System includes a HeatWave plugin that is responsible for cluster management, query scheduling, and returning query results to the MySQL DB System. The HeatWave nodes store data in memory and process analytics and machine learning queries. Each HeatWave node hosts an instance of the HeatWave query processing engine (`RAPID`).

Enabling a HeatWave Cluster also provides access to HeatWave AutoML, which is a fully managed, highly scalable, cost-efficient, machine learning solution for data stored in MySQL. HeatWave AutoML provides a simple SQL interface for training and using predictive machine learning models, which can be used by novice and experienced ML practitioners alike. Machine learning expertise, specialized tools, and algorithms are not required. With HeatWave AutoML, you can train a model with a single call to an SQL routine. Similarly, you can generate predictions with a single `CALL` or `SELECT` statement which can be easily integrated with your applications.

1.1 HeatWave Architectural Features

The HeatWave architecture supports OLTP, OLAP and machine learning.

Figure 1.1 HeatWave Architecture



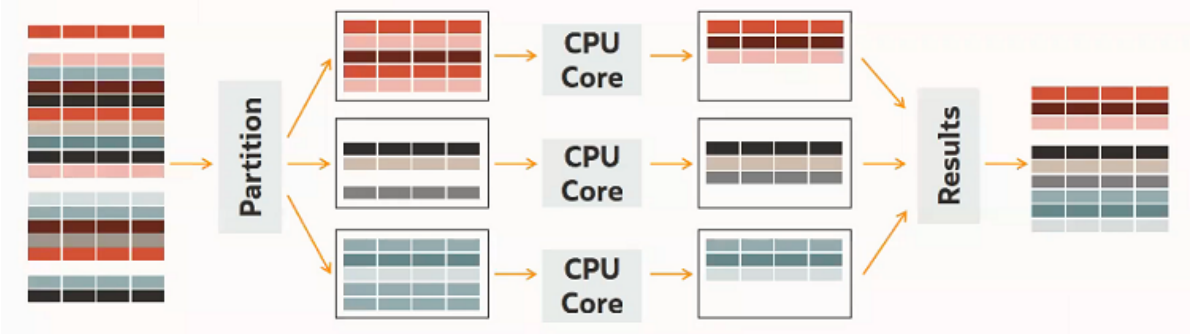
In-Memory Hybrid-Columnar Format

HeatWave stores data in main memory in a hybrid columnar format. The HeatWave hybrid approach achieves the benefits of columnar format for query processing, while avoiding the materialization and update costs associated with pure columnar format. Hybrid columnar format enables the use of efficient query processing algorithms designed to operate on fixed-width data, and permits vectorized query processing.

Massively Parallel Architecture

The HeatWave massively parallel architecture uses internode and intranode partitioning of data. Each node within a HeatWave Cluster, and each CPU core within a node, processes the partitioned data in parallel. HeatWave is capable of scaling to thousands of cores. This massively parallel architecture, combined with high-fanout, workload-aware partitioning, accelerates query processing.

Figure 1.2 HeatWave Massively Parallel Architecture



Push-Based Vectorized Query Processing

HeatWave processes queries by pushing vector blocks (slices of columnar data) through the query execution plan from one operator to another. A push-based execution model avoids deep call stacks and saves valuable resources compared to tuple-based processing models.

Scale-Out Data Management

When analytics data is loaded into HeatWave, the HeatWave Storage Layer automatically persists the data for pause and resume of the HeatWave Cluster and for fast recovery in case of a HeatWave node or cluster failure. Data is automatically restored by the HeatWave Storage Layer when the HeatWave Cluster resumes after a pause or recovers a failed node or cluster. This automated, self-managing storage layer scales to the size required for your HeatWave Cluster and operates independently in the background. HeatWave on OCI persists the data to OCI Object Storage. HeatWave on AWS persists the data to AWS S3.

Native MySQL Integration

Native integration with MySQL provides a single data management platform for OLTP, OLAP, mixed workloads, and machine learning. HeatWave is designed as a pluggable MySQL storage engine, which enables management of both the MySQL and HeatWave using the same interfaces.

Changes to analytics data on the MySQL DB System are automatically propagated to HeatWave nodes in real time, which means that queries always have access to the latest data. Change propagation is performed automatically by a light-weight algorithm.

Users and applications interact with HeatWave through the MySQL DB System using standard tools and standard-based ODBC/JDBC connectors. HeatWave supports the same ANSI SQL standard and ACID properties as MySQL and the most commonly used data types. This support enables existing applications to use HeatWave without modification, allowing for quick and easy integration.

1.2 HeatWave MySQL

Analytics and machine learning queries are issued from a MySQL client or application that interacts with the HeatWave Cluster by connecting to the MySQL DB System. Results are returned to the MySQL DB System and to the MySQL client or application that issued the query.

The number of HeatWave nodes required depends on data size and the amount of compression that is achieved when loading data into the HeatWave Cluster. A HeatWave Cluster in Oracle Cloud Infrastructure (OCI) or Oracle Database Service for Azure (ODSA) supports up to 64 nodes. On Amazon Web Services (AWS), a HeatWave Cluster supports up to 128 nodes.

On Oracle Cloud Infrastructure (OCI), data that is loaded into HeatWave is automatically persisted to OCI Object Storage, which allows data to be reloaded quickly when the HeatWave Cluster resumes after a pause or when the HeatWave Cluster recovers from a cluster or node failure.

HeatWave network traffic is fully encrypted.

1.3 HeatWave AutoML

With HeatWave AutoML, data and models never leave HeatWave, saving you time and effort while keeping your data and models secure. HeatWave AutoML is optimized for HeatWave shapes and scaling, and all HeatWave AutoML processing is performed on the HeatWave Cluster. HeatWave distributes ML computation among HeatWave nodes, to take advantage of the scalability and massively parallel

processing capabilities of HeatWave. For more information about the machine learning capabilities of HeatWave, see Chapter 3, *HeatWave AutoML*.

1.4 HeatWave GenAI

The HeatWave GenAI feature of HeatWave lets you communicate with unstructured data in HeatWave using natural language queries. It uses large language models (LLMs) to enable natural language communication and provides an inbuilt vector store that you can use to store enterprise-specific proprietary content to perform vector searches. HeatWave GenAI also includes HeatWave Chat which is a chatbot that extends the generative AI and vector search functionalities to let you ask multiple follow-up questions about a topic in a single session. HeatWave Chat can even draw its knowledge from documents ingested by the vector store.

See: Chapter 4, *HeatWave GenAI*.

1.5 HeatWave Lakehouse

The Lakehouse feature of HeatWave enables query processing on data resident in Object Storage. The source data is read from Object Storage, transformed to the memory optimized HeatWave format, stored in the HeatWave persistence storage layer in Object Storage, and then loaded to HeatWave cluster memory.

- Provides in-memory query processing on data resident in Object Storage.
- Data is not loaded into the MySQL InnoDB storage layer.
- Supports structured and semi-structured relational data in the following file formats:
  - Avro.
  - CSV.
  - JSON.

    As of MySQL 8.4.0, Lakehouse supports Newline Delimited JSON.
  - Parquet.
- With this feature, users can now analyse data in both InnoDB and an object store using familiar SQL syntax in the same query.

See: Chapter 5, *HeatWave Lakehouse*. To use Lakehouse with HeatWave AutoML, see: Section 3.12, “HeatWave AutoML and Lakehouse”.

1.6 HeatWave Autopilot

HeatWave Autopilot automates many of the most important and often challenging aspects of achieving exceptional query performance at scale, including cluster provisioning, loading data, query processing, and failure handling. It uses advanced techniques to sample data, collect statistics on data and queries, and build machine learning models to model memory usage, network load, and execution time. The machine learning models are used by HeatWave Autopilot to execute its core capabilities. HeatWave Autopilot makes the HeatWave query optimizer increasingly intelligent as more queries are executed, resulting in continually improving system performance.

System Setup

- *Auto Provisioning*

Estimates the number of HeatWave nodes required by sampling the data, which means that manual cluster size estimations are not necessary.

- For HeatWave on OCI, see Generating a Node Count Estimate in the *HeatWave on OCI Service Guide*.
- For HeatWave on AWS, see Estimating Cluster Size with HeatWave Autopilot in the *HeatWave on AWS Service Guide*.
- For HeatWave for Azure, see Provisioning HeatWave Nodes in the *HeatWave for Azure Service Guide*.

- *Auto Shape Prediction*

  For HeatWave on AWS, the Auto Shape Prediction feature in HeatWave Autopilot uses MySQL statistics for the workload to assess the suitability of the current shape. Auto Shape Prediction provides prompts to upsize the shape and improve system performance, or to downsize the shape if the system is under-utilized. See: Autopilot Shape Advisor in the *HeatWave on AWS Service Guide*.

Data Load

- *Auto Parallel Load*

  Optimizes load time and memory usage by predicting the optimal degree of parallelism for each table loaded into HeatWave. See: Section 2.2.3, “Loading Data Using Auto Parallel Load”.

- *Auto Schema Inference*

  Lakehouse Auto Parallel Load extends Auto Parallel Load with Auto Schema Inference that can analyze the data, infer the external table structure, and create the database and all tables. It can also use header information from the external files to define the column names. See: Section 5.2.4.1, “Lakehouse Auto Parallel Load Schema Inference”.

- *Autopilot Indexing*

  Autopilot Indexing can make secondary index suggestions to improve workload performance. See: Section 2.8.1, “Autopilot Indexing”

- *Auto Encoding*

  Determines the optimal encoding for string column data, which minimizes the required cluster size and improves query performance. See: Section 2.7.4, “Auto Encoding”.

- *Auto Data Placement*

  Recommends how tables should be partitioned in memory to achieve the best query performance, and estimates the expected performance improvement. See: Section 2.7.5, “Auto Data Placement”.

- *Auto Data Compression*

  HeatWave and HeatWave Lakehouse can compress data stored in memory using different compression algorithms. To minimize memory usage while providing the best query performance, auto compression dynamically determines the compression algorithm to use for each column based on its data characteristics. Auto compression employs an adaptive sampling technique during the data loading process, and automatically selects the optimal compression algorithm without user intervention. Algorithm selection is based on the compression ratio and the compression and decompression rates,

which balance the memory needed to store the data in HeatWave with query execution time. See: Section 2.2.6, “Data Compression”.

- *Unload Advisor*

  Unload Advisor can recommend tables to unload to reduce HeatWave memory usage. See: Section 2.7.7, “Unload Advisor”.

- *Auto Unload*

  Auto Unload can automate the process of unloading data from HeatWave. See: Section 2.5.3, “Unloading Data Using Auto Unload”.

Query Execution

- *Auto Query Plan Improvement*

  Collects previously executed queries and uses them to improve future query execution plans. See: Section 2.3.4, “Auto Query Plan Improvement”.

- *Adaptive Query Execution*

  Adaptive query optimization automatically improves query performance and memory consumption, and mitigates skew-related performance issues as well as out of memory errors. It uses various statistics to adjust data structures and system resources after query execution has started. It independently optimizes query execution for each HeatWave node based on actual data distribution at runtime. This helps improve the performance of ad hoc queries by up to 25%. The HeatWave optimizer generates a physical query plan based on statistics collected by Autopilot. During query execution, each HeatWave node executes the same query plan. With adaptive query execution, each individual HeatWave node adjusts the local query plan based on statistics such as cardinality and distinct value counts of intermediate relations collected locally in real-time. This allows each HeatWave node to tailor the data structures that it needs, resulting in better query execution time, lower memory usage, and improved data skew-related performance.

- *Auto Query Time Estimation*

  Estimates query execution time to determine how a query might perform without having to run the query. See: Section 2.7.6, “Auto Query Time Estimation”.

- *Auto Change Propagation*

  Auto Change Propagation intelligently determines the optimal time when changes to data on the MySQL DB System should be propagated to the HeatWave Storage Layer. Only available for HeatWave on OCI.

- *Auto Scheduling*

  Prioritizes queries in an intelligent way to reduce overall query execution wait times. See: Section 2.3.3, “Auto Scheduling”.

- *Auto Thread Pooling*

  Provides sustained throughput during high transaction concurrency. Where multiple clients are running queries concurrently, Auto Thread Pooling applies workload-aware admission control to eliminate resource contention caused by too many waiting transactions. Auto Thread Pooling automatically manages the settings for the thread pool control variables `thread_pool_size`, `thread_pool_max_transactions_limit`, and `thread_pool_query_threads_per_group`. For details of how the thread pool works, see Thread Pool Operation.

Failure Handling

- *Auto Error Recovery*

  For HeatWave on OCI, Auto Error Recovery recovers a failed node or provisions a new one and reloads data from the HeatWave storage layer when a HeatWave node becomes unresponsive due to a software or hardware failure. See: HeatWave Cluster Failure and Recovery in the *HeatWave on OCI Service Guide*.

  For HeatWave on AWS, Auto Error Recovery recovers a failed node and reloads data from the MySQL DB System when a HeatWave node becomes unresponsive due to a software failure.

1.7 MySQL Functionality for HeatWave

The following items provide additional functionality for MySQL that are only available with HeatWave MySQL:

- See Section 2.17, “Bulk Ingest Data to MySQL Server” for bulk ingest.
- See Section 2.12.1, “Aggregate Functions” for the `HLL()` function.
- See Section 2.12.1.1, “GROUP BY Modifiers” for the `CUBE` modifier.
- See Section 2.13, “SELECT Statement” for the `QUALIFY` and `TABLESAMPLE` clauses.
- See Section 2.12.13, “Vector Functions” for the `DISTANCE()` function.

Chapter 2 HeatWave MySQL

Table of Contents

When a HeatWave Cluster is enabled, queries that meet certain prerequisites are automatically offloaded from the MySQL DB System to the HeatWave Cluster for accelerated processing.

Queries are issued from a MySQL client or application that interacts with the HeatWave Cluster by connecting to the MySQL DB System. Results are returned to the MySQL DB System and to the MySQL client or application that issued the query.

Manually loading data into HeatWave involves preparing tables on the MySQL DB System and executing load statements. See Section 2.2.2, “Loading Data Manually”. The Auto Parallel Load utility facilitates the process of loading data by automating required steps and optimizing the number of parallel load threads. See Section 2.2.3, “Loading Data Using Auto Parallel Load”.

For HeatWave on AWS, load data into HeatWave using the HeatWave Console. See Manage Data in HeatWave with Workspaces in the *HeatWave on AWS Service Guide*.

For HeatWave for Azure, see Importing Data to HeatWave in the *HeatWave for Azure Service Guide*.

When HeatWave loads a table, the data is sharded and distributed among HeatWave nodes. After a table is loaded, DML operations on the tables are automatically propagated to the HeatWave nodes. No user action is required to synchronize data. For more information, see Section 2.2.7, “Change Propagation”.

On Oracle Cloud Infrastructure, OCI, data loaded into HeatWave, including propagated changes, automatically persists in the HeatWave Storage Layer to OCI Object Storage for fast recovery in case of a HeatWave node or cluster failure. For HeatWave on AWS, data is recovered from the MySQL DB System.

After running a number of queries, use the HeatWave Autopilot Advisor to optimize the workload. Advisor analyzes the data and query history to provide string column encoding and data placement recommendations. See Section 2.7, “Workload Optimization for OLAP”.

2.1 Before You Begin

Before you begin using HeatWave, the following is assumed:

- An operational MySQL DB System, and able to connect to it using a MySQL client. If not, refer to the following procedures:
  - For HeatWave on OCI, see Creating a DB System, and Connecting to a DB System in the *HeatWave on OCI Service Guide*.
  - For HeatWave on AWS, see Creating a DB System, and Connecting from a Client in the *HeatWave on AWS Service Guide*.
  - For HeatWave for Azure, see Provisioning HeatWave and Connecting to HeatWave in the *HeatWave for Azure Service Guide*.
- The MySQL DB System has an operational HeatWave Cluster. If not, refer to the following procedures:
  - For HeatWave on OCI, see Adding a HeatWave Cluster in the *HeatWave on OCI Service Guide*.
  - For HeatWave on AWS, see Creating a HeatWave Cluster in the *HeatWave on AWS Service Guide*.
  - For HeatWave for Azure, see Provisioning HeatWave in the *HeatWave for Azure Service Guide*.

2.2 Loading Data to HeatWave MySQL

This section describes how to load data into HeatWave. The following methods are supported:

- Loading data manually. This method loads one table at a time and involves executing multiple statements for each table. See Section 2.2.2, “Loading Data Manually”.
- Loading data using Auto Parallel Load. This HeatWave Autopilot enabled method loads one or more schemas at a time and facilitates loading by automating manual steps and optimizing the number of parallel load threads for a faster load. See Section 2.2.3, “Loading Data Using Auto Parallel Load”.
- For users of HeatWave on AWS, load data using the HeatWave Console. This GUI-based and HeatWave Autopilot enabled method loads selected schemas and tables using an optimized number of parallel load threads for a faster load. See Manage Data in HeatWave with Workspaces in the *HeatWave on AWS Service Guide*.

HeatWave loads data with batched, multi-threaded reads from `InnoDB`. HeatWave then converts the data into columnar format and sends it over the network to distribute it among HeatWave nodes in horizontal slices. HeatWave partitions data by the table primary key, unless the table definition includes data placement keys. See Section 2.7.2, “Defining Data Placement Keys”.

Concurrent DML operations and queries on the MySQL node are supported while a data load operation is in progress; however, concurrent operations on the MySQL node can affect load performance and vice versa.

After tables are loaded, changes to table data on the MySQL DB System node are automatically propagated to HeatWave. For more information, see Section 2.2.7, “Change Propagation”.

For each table that is loaded in HeatWave, the default heap segment size is 64KB per table, and this is allocated from the root heap. The root heap available to HeatWave is approximately 400GB.

HeatWave compresses data as it is loaded, which permits HeatWave nodes to store more data at a minor cost to performance. If you do not want to compress data as it is loaded in HeatWave, you must disable compression before loading data. See Section 2.2.6, “Data Compression”.

For related best practices, see Section 2.9, “Best Practices”.

2.2.1 Prerequisites

Before loading data, ensure that you have met the following prerequisites:

- The data you want to load must be available on the MySQL DB System. For information about importing data into a MySQL DB System, refer to the following instructions:
  - For HeatWave on OCI, see Importing and Exporting Databases in the *HeatWave on OCI Service Guide*.
  - For HeatWave on AWS, see Importing Data in the *HeatWave on AWS Service Guide*.
  - For HeatWave for Azure, see Importing Data to HeatWave in the *HeatWave for Azure Service Guide*.

    > **Tip**
    >
    > For an OLTP workload that resides in an on-premise instance of MySQL Server, inbound replication is recommended for replicating data to the MySQL DB System for offload to the HeatWave Cluster. See Section 2.9, “Best Practices” and Replication in the *HeatWave on OCI Service Guide*.

- The tables you intend to load must be `InnoDB` tables. You can manually convert tables to InnoDB using the following `ALTER TABLE` statement:

```
mysql> ALTER TABLE tbl_name ENGINE=InnoDB;
```

- The tables you intend to load must be defined with a primary key. You can add a primary key using the following syntax:

```
mysql> ALTER TABLE tbl_name ADD PRIMARY KEY (column);
```

  Adding a primary key is a table-rebuilding operation. For more information, see Primary Key Operations.

  Primary key columns defined with column prefixes are not supported.

  Load time is affected if the primary key contains more than one column, or if the primary key column is not an `INTEGER` column. The impact on MySQL performance during load, change propagation, and

query processing depends on factors such as data properties, available resources (compute, memory, and network), and the rate of transaction processing on the MySQL DB System.

- Identify all of the tables that your queries access to ensure that you load all of them into HeatWave. If a query accesses a table that is not loaded into HeatWave, it will not be offloaded to HeatWave for processing.
- Column width cannot exceed 65532 bytes.
- The number of columns per table cannot exceed 1017.

2.2.2 Loading Data Manually

As of MySQL 8.4.0, HeatWave supports InnoDB partitions. Selectively loading and unloading partitions will reduce memory requirements. The use of partition information can allow data-skipping and accelerate query processing.

HeatWave Guided Load uses HeatWave Autopilot to exclude schemas, tables, and columns that cannot be loaded, and define `RAPID` as the secondary engine. To load data manually, follow these steps:

1. Optionally, applying string column encoding and data placement workload optimizations. For more information, see Section 2.7, “Workload Optimization for OLAP”.
2. Loading tables or partitions using `ALTER TABLE ... SECONDARY_LOAD` statements. See Section 2.2.2.3, “Loading Tables” and Section 2.2.2.4, “Loading Partitions”.

2.2.2.1 Excluding Table Columns

Before loading a table into HeatWave, columns with unsupported data types must be excluded; otherwise, the table cannot be loaded. For a list of data types that HeatWave supports, see Section 2.10, “Supported Data Types”.

Optionally, exclude columns that are not relevant to the intended queries. Excluding irrelevant columns is not required but doing so reduces load time and the amount of memory required to store table data.

To exclude a column, specify the `NOT SECONDARY` column attribute in an `ALTER TABLE` or `CREATE TABLE` statement, as shown below. The `NOT SECONDARY` column attribute prevents a column from being loaded into HeatWave when executing a table load operation.

```
mysql> ALTER TABLE tbl_name MODIFY description BLOB NOT SECONDARY;
```

```
mysql> CREATE TABLE orders (id INT, description BLOB NOT SECONDARY);
```

> **Note**
>
> If a query accesses a column defined with the `NOT SECONDARY` attribute, the query is executed on the MySQL DB System by default.

To include a column that was previously excluded, refer to the procedure described in Section 2.4, “Modifying Tables”.

2.2.2.2 Defining the Secondary Engine

For each table that you want to load into HeatWave, you must define the HeatWave query processing engine (`RAPID`) as the secondary engine for the table. To define `RAPID` as the secondary engine, specify the `SECONDARY_ENGINE` table option in an `ALTER TABLE` or `CREATE TABLE` statement:

```
mysql> ALTER TABLE tbl_name SECONDARY_ENGINE = RAPID;
```

```
mysql> CREATE TABLE orders (id INT) SECONDARY_ENGINE = RAPID;
```

2.2.2.3 Loading Tables

To load a table into HeatWave, specify the `SECONDARY_LOAD` clause in an `ALTER TABLE` statement.

```
mysql> ALTER TABLE tbl_name SECONDARY_LOAD;
```

The `SECONDARY_LOAD` clause has these properties:

- Data is read using the `READ COMMITTED` isolation level.

2.2.2.4 Loading Partitions

As of MySQL 8.4.0, HeatWave supports InnoDB partitions, see Partition Selection. To load partitions into HeatWave, specify the `SECONDARY_LOAD` clause in an `ALTER TABLE` statement.

```
mysql> ALTER TABLE tbl_name SECONDARY_LOAD PARTITION (p0, p1, ..., pn);
```

The `SECONDARY_LOAD PARTITION` clause is valid if the table has been loaded to HeatWave or not. For example:

```
mysql> ALTER TABLE t1 SECONDARY_LOAD;
mysql> ALTER TABLE t1 ADD PARTITION (PARTITION p4 VALUES LESS THAN (2002));
mysql> ALTER TABLE t1 SECONDARY_LOAD PARTITION (p4);
```

The `SECONDARY_LOAD` clause has these properties:

- Data is read using the `READ COMMITTED` isolation level.

See: Section 2.18.6, “Partition Selection Limitations”.

2.2.3 Loading Data Using Auto Parallel Load

Auto Parallel Load facilitates the process of loading data into HeatWave by automating many of the steps involved, including:

- Excluding schemas, tables, and columns that cannot be loaded.
- Defining `RAPID` as the secondary engine for tables that are to be loaded.
- Verifying that there is sufficient memory available for the data.
- Optimizing load parallelism based on machine-learning models.
- Loading data into HeatWave.

Auto Parallel Load, which can be run from any MySQL client or connector, is implemented as a stored procedure named `heatwave_load`, which resides in the MySQL `sys` schema. Running Auto Parallel Load involves issuing a `CALL` statement for the stored procedure, which takes `schemas` and *`options`* as arguments; for example, this statement loads the `tpch` schema:

```
mysql> CALL sys.heatwave_load(JSON_ARRAY("tpch"),NULL);
```

2.2.3.1 Auto Parallel Load Requirements

- The user must have the following MySQL privileges:

- The `PROCESS` privilege.
- The `EXECUTE` privilege on the `sys` schema.
- The `SELECT` privilege on the Performance Schema.

- To run Auto Parallel Load in `normal` mode, the HeatWave Cluster must be active.

2.2.3.2 Auto Parallel Load Syntax

MySQL 9.0.0 adds support for Lakehouse Incremental Load with the `refresh_external_tables` option, see: Section 5.2.6, “Lakehouse Incremental Load”.

```
mysql> CALL sys.heatwave_load (input_list,[options]);

input_list: {
     JSON_ARRAY(input [,input] ...)
}

options: {
     JSON_OBJECT("key","value"["key","value"] ...)
          "key","value": {
          ["mode",{"normal"|"dryrun"|"validation"}]
          ["output",{"normal"|"compact"|"silent"|"help"}]
          ["sql_mode","sql_mode"]
          ["policy",{"disable_unsupported_columns"|"not_disable_unsupported_columns"}]
          ["set_load_parallelism",{true|false}]
          ["auto_enc",JSON_OBJECT("mode",{"off"|"check"})]
          ["refresh_external_tables",{true|false}]
          }
}

input: {
     'db_name' | db_object
}

db_object: {
     JSON_OBJECT("key","value"["key","value"] ...)
          "key","value": {
          "db_name": "db_name",
          ["tables": JSON_ARRAY(table [, table] ...)]
          ["exclude_tables": JSON_ARRAY(table [, table] ...)]
          }
}

table: {
     'table_name' | table_object
}

table_object: {
     JSON_OBJECT("key","value"["key","value"] ...)
          "key","value": {
          "table_name": "table_name",
          ['engine_attribute': engine_attribute_object],
          ['columns': JSON_ARRAY('column_name' [, 'column_name', ...]}],
          ['exclude_columns': JSON_ARRAY('column_name' [, 'column_name', ...]}]
          }
}

engine_attribute_object: {
     JSON_OBJECT("key","value"["key","value"] ...)
          "key","value": {
          "sampling": true|false,
          "dialect": {dialect_section},
```

```
          "file": JSON_ARRAY(file_section [, file_section]...),
          }
}
```

MySQL 8.4.0 adds support for the following:

- An `input_list` JSON array replaces the `db_list` JSON array. This adds an include list to exactly specify the tables and columns to load for a set of queries. It is no longer necessary to include a complete schema, and exclude unnecessary tables and columns.

  `input_list` is backwards compatible with `db_list`.

- A `validation` mode for external files.

As of MySQL 8.4.0, do not use the `external_tables` option. It will be deprecated in a future release. Use `db_object` with `tables` or `exclude_tables` instead.

```
mysql> CALL sys.heatwave_load (input_list,[options]);

input_list: {
     JSON_ARRAY(input [,input] ...)
}

options: {
     JSON_OBJECT("key","value"["key","value"] ...)
          "key","value": {
          ["mode",{"normal"|"dryrun"|"validation"}]
          ["output",{"normal"|"compact"|"silent"|"help"}]
          ["sql_mode","sql_mode"]
          ["policy",{"disable_unsupported_columns"|"not_disable_unsupported_columns"}]
          ["set_load_parallelism",{true|false}]
          ["auto_enc",JSON_OBJECT("mode",{"off"|"check"})]
          }
}

input: {
     'db_name' | db_object
}

db_object: {
     JSON_OBJECT("key","value"["key","value"] ...)
          "key","value": {
          "db_name": "db_name",
          ["tables": JSON_ARRAY(table [, table] ...)]
          ["exclude_tables": JSON_ARRAY(table [, table] ...)]
          }
}

table: {
     'table_name' | table_object
}

table_object: {
     JSON_OBJECT("key","value"["key","value"] ...)
          "key","value": {
          "table_name": "table_name",
          ['engine_attribute': engine_attribute_object],
          ['columns': JSON_ARRAY('column_name' [, 'column_name', ...]}],
          ['exclude_columns': JSON_ARRAY('column_name' [, 'column_name', ...]}]
          }
}

engine_attribute_object: {
     JSON_OBJECT("key","value"["key","value"] ...)
          "key","value": {
          "sampling": true|false,
```

```
          "dialect": {dialect_section},
          "file": JSON_ARRAY(file_section [, file_section]...),
          }
}
```

As of MySQL 8.4.0 use `input_list` to define what to load. `input_list` is a JSON array and requires one or more valid `input` which can be either a valid schema name or a `db_object`. An empty array is permitted to view the Auto Parallel Load command-line help, see Section 2.2.3.5, “Auto Parallel Load Command-Line Help”. This is backwards compatible with `db_list`.

Before MySQL 8.4.0, `db_list` specifies the schemas to load. The list is a JSON array and requires one or more valid schema names. An empty array is permitted to view the Auto Parallel Load command-line help.

Use key-value pairs in `JSON` format to specify parameters. HeatWave uses the default setting if there is no option setting. Use `NULL` to specify no arguments.

For syntax examples, see Section 2.2.3.6, “Auto Parallel Load Examples”.

Auto Parallel Load `options` is a JSON object literal that includes:

- `mode`: Defines the Auto Parallel Load operational mode. Permitted values are:
  - `normal`: The default. Generates and executes the load script.
  - `dryrun`: Generates a load script only. Auto Parallel Load executes in `dryrun` mode automatically if the HeatWave Cluster is not active.
  - `validation`: Only use with Lakehouse. `validation` performs the same checks as `dryrun` and also validates external files before loading. It follows all the `options` and the load configuration, for example column information, `sql_mode`, `is_strict_mode` and `allow_missing_files`, but does not load any tables. It uses schema inference and might modify the schema, see: Section 5.2.4.1, “Lakehouse Auto Parallel Load Schema Inference”. `validation` is faster than a full load, particularly for large tables. The memory requirement is similar to running a full load.

    > **Note**
    >
    > `validation` requires created tables.

- `output`: Defines how Auto Parallel Load produces output. Permitted values are:
  - `normal`: The default. Produces summarized output and sends it to `stdout` and to the `heatwave_autopilot_report` table. See Section 2.7.9, “Autopilot Report Table”.
  - `silent`: Sends output to the `heatwave_autopilot_report` table only. See Section 2.7.9, “Autopilot Report Table”. The `silent` output type is useful if human-readable output is not required; when the output is consumed by a script, for example. For an example of a stored procedure with an Auto Parallel Load call that uses the `silent` output type, see Section 2.2.3.6, “Auto Parallel Load Examples”.
  - `compact`: Produces compact output.
  - `help`: Displays Auto Parallel Load command-line help. See Section 2.2.3.5, “Auto Parallel Load Command-Line Help”.
- `sql_mode`: Defines the SQL mode used while loading tables. Auto Parallel Load does not support the MySQL global or session `sql_mode` variable. To run Auto Parallel Load with a non-oci-default SQL mode configuration, specify the configuration using the Auto Parallel Load `sql_mode` option as a string value. If no SQL modes are specified, the default OCI SQL mode configuration is used.

For information about SQL modes, see Server SQL Modes.

- `policy`: Defines the policy for handling of tables containing columns with unsupported data types. Permitted values are:

  - `disable_unsupported_columns`: The default. Disable columns with unsupported data types and include the table in the load script. Columns that are explicitly pre-defined as `NOT SECONDARY` are ignored (they are neither disabled or enabled).

    Auto Parallel Load does not generate statements to disable columns that are explicitly defined as `NOT SECONDARY`.

  - `not_disable_unsupported_columns`: Exclude the table from the load script if the table contains a column with an unsupported data type.

    A column with an unsupported data type that is explicitly defined as a `NOT SECONDARY` column does not cause the table to be excluded. For information about defining columns as `NOT SECONDARY`, see Section 2.2.2.1, “Excluding Table Columns”.

- `exclude_list`: Defines a list of schemas, tables, and columns to exclude from the load script. Names must be fully qualified without backticks.

  Do not use as of MySQL 8.4.0. Use `db_object` with `tables`, `exclude_tables`, `columns` or `exclude_columns` instead. `exclude_list` will be deprecated in a future release.

  Auto Parallel Load automatically excludes database objects that cannot be offloaded, according to the default `policy` setting. These objects need not be specified explicitly in the exclude list. System schemas, non-`InnoDB` tables, tables that are already loaded in HeatWave, and columns explicitly defined as `NOT SECONDARY` are automatically excluded.

- `set_load_parallelism`: Enabled by default. Optimizes load parallelism based on machine-learning models by optimizing the `innodb_parallel_read_threads` variable setting before loading each table.

- `auto_enc`: Checks if there is enough memory for string column encoding. Settings include:

  - `mode`: Defines the `auto_enc` operational mode. Permitted values are:

    - `off`: Disables the `auto_enc` option. No memory checks are performed.

    - `check`: The default. Checks if there is enough memory on the MySQL node for dictionary-encoded columns and if there is enough root heap memory for variable-length column encoding overhead. Dictionary-encoded columns require memory on the MySQL node for dictionaries. For each loaded table, 64KB of memory, the default heap segment size, must be allocated from the root heap for variable-length column encoding overhead. If there is not enough memory, Auto Parallel Load executes in `dryrun` mode and prints a warning about insufficient memory. The `auto_enc` option runs `check` mode if it is not specified explicitly and set to `off`. For more information, see Section 2.2.3.4, “Memory Estimation for String Column Encoding”.

- `refresh_external_tables`: Only use with Lakehouse. Set to `true` to refresh external tables. See: Section 5.2.6, “Lakehouse Incremental Load”.

  Set to `false`, the default setting, to only load new tables, unloaded tables, and not refresh external tables.

  This option only works with `mode` set to `normal`. All other `mode` settings ignore this option.

- `external_tables`: non-InnoDB tables which do not store any data, but refer to data stored externally. For the `external_tables` syntax, see: Section 5.2.4.2, “Lakehouse Auto Parallel Load with the external_tables Option”.

  Do not use as of MySQL 8.4.0. Use `db_object` with `tables` or `exclude_tables` instead. It will be deprecated in a future release.

The `db_object` is a JSON object literal that includes:

- `db_name`: The name of the database to load.
- Use one or other of the following, but not both. The use of both parameters will throw an error.
  - `tables`: An optional JSON array of `table` to include in the load.
  - `exclude_tables`: As of MySQL 8.4.0, an optional JSON array of `table` to exclude from the load.
- `table`: Either a valid table name or a `table_object`.
- As of MySQL 8.4.0, `table_object` is a JSON object literal that includes:
  - `table_name`: The name of the table to load.
  - `engine_attribute`: A JSON object literal that includes:
    - `sampling`: Only use with Lakehouse. If set to `true`, the default setting, Lakehouse Auto Parallel Load samples the data to infer the schema and collect statistics.

      If set to `false`, Lakehouse Auto Parallel Load performs a full scan to infer the schema and collect statistics. Depending on the size of the data, this can take a long time.

      Auto Parallel Load uses the inferred schema to generate `CREATE TABLE` statements. The statistics are used to estimate storage requirements and load times. See: Section 5.2.4.1, “Lakehouse Auto Parallel Load Schema Inference”.
    - For `dialect` and `file`, see: Section 5.2.2, “Lakehouse External Table Syntax”.
  - Use one or other of the following, but not both. The use of both parameters will throw an error.
    - `columns`: An optional JSON array of `column_name` to include in the load.
    - `exclude_columns`: An optional JSON array of `column_name` to exclude from the load.

2.2.3.3 Running Auto Parallel Load

Run Auto Parallel Load in `dryrun` mode first to check for errors and warnings and to inspect the generated load script. To load a single schema in `dryrun` mode:

```
mysql> CALL sys.heatwave_load(JSON_ARRAY("tpch"), JSON_OBJECT("mode","dryrun"));
```

In `dryrun` mode, Auto Parallel Load sends the load script to the `heatwave_autopilot_report` table only. See Section 2.7.9, “Autopilot Report Table”. It does not load data into HeatWave.

If Auto Parallel Load fails with an error, inspect the errors with a query to the `heatwave_autopilot_report` table.

```
mysql> SELECT log FROM sys.heatwave_autopilot_report
         WHERE type="error";
```

When Auto Parallel Load finishes running, query the `heatwave_autopilot_report` table to check for warnings.

```
mysql> SELECT log FROM sys.heatwave_autopilot_report
          WHERE type="warn";
```

Issue the following query to inspect the load script that was generated.

```
mysql> SELECT log->>"$.sql" AS "Load Script"
          FROM sys.heatwave_autopilot_report
          WHERE type = "sql" ORDER BY id;
```

Once you are satisfied with the Auto Parallel Load `CALL` statement and the generated load script, reissue the `CALL` statement in `normal` mode to load the data into HeatWave. For example:

```
mysql> CALL sys.heatwave_load(JSON_ARRAY("tpch"), JSON_OBJECT("mode","normal"));
```

> **Note**
>
> Retrieve DDL statements in a table or use the following statements to produce a list of DDL statements to easily copy and paste.
>
> ```
> mysql> SET SESSION group_concat_max_len = 1000000;
> mysql> SELECT GROUP_CONCAT(log->>"$.sql" SEPARATOR ' ')
>           FROM sys.heatwave_autopilot_report
>           WHERE type = "sql" ORDER BY id;
> ```

The time required to load data depends on the data size. Auto Parallel Load provides an estimate of the time required to complete the load operation.

Tables are loaded in sequence, ordered by schema and table name. Load-time errors are reported as they are encountered. If an error is encountered while loading a table, the operation is not terminated. Auto Parallel Load continues running, moving on to the next table.

When Auto Parallel Load finishes running, it checks if tables are loaded and shows a summary with the number of tables that were loaded and the number of tables that failed to load.

2.2.3.4 Memory Estimation for String Column Encoding

The `auto_enc` option is run in `check` mode by default to ensure that there is enough memory for string column encoding.

The following example uses the `auto_enc` option in `check` mode, if you want to ensure that there is sufficient memory for string column encoding before attempting a load operation. Insufficient memory can cause a load failure.

```
mysql> CALL sys.heatwave_load(JSON_ARRAY("tpch"),
          JSON_OBJECT("mode","dryrun","auto_enc",JSON_OBJECT("mode","check")));
```

> **Note**
>
> The `auto_enc` option runs in `check` mode regardless of whether it is specified explicitly in the Auto Parallel Load call statement.

Look for capacity estimation data in the Auto Parallel Load output. The results indicate whether there is sufficient memory to load all tables.

2.2.3.5 Auto Parallel Load Command-Line Help

To view Auto Parallel Load command-line help, issue the following statement:

```
mysql> CALL sys.heatwave_load(JSON_ARRAY(),JSON_OBJECT("output","help"));
```

The command-line help provides usage documentation for the Auto Parallel Load utility.

2.2.3.6 Auto Parallel Load Examples

- Load a single schema with default options.

```
mysql> CALL sys.heatwave_load(JSON_ARRAY("tpch"),NULL);
```

- Load multiple schemas with default options.

```
mysql> CALL sys.heatwave_load(JSON_ARRAY("tpch","airportdb","employees","sakila"),NULL);
```

- Load multiple schemas with the `not_disable_unsupported_columns` policy, which causes tables with unsupported columns to be excluded from the load operation. Unsupported columns are those with unsupported data types.

```
mysql> CALL sys.heatwave_load(JSON_ARRAY("tpch","airportdb","employees","sakila"),
         JSON_OBJECT("policy","not_disable_unsupported_columns"));
```

- Load multiple schemas, excluding specified tables and a particular column:

```
mysql> CALL sys.heatwave_load(JSON_ARRAY("tpch","airportdb"),
         JSON_OBJECT("exclude_list",JSON_ARRAY("tpch.orders","airportdb.employee.salary")));
```

- Load tables that begin with an “hw” prefix from a schema named `schema_customer_1`.

```
mysql> SET @exc_list = (SELECT JSON_OBJECT('exclude_list',
                JSON_ARRAYAGG(CONCAT(table_schema,'.',table_name)))
                FROM information_schema.tables
                WHERE table_schema = 'schema_customer_1'
                AND table_name NOT LIKE 'hw%');
mysql> CALL sys.heatwave_load(JSON_ARRAY('schema_customer_1'), @exc_list);
```

- Load all schemas with tables that start with an “hw” prefix.

```
mysql> SET @db_list = (SELECT json_arrayagg(schema_name) FROM information_schema.schemata);
mysql> SET @exc_list = (SELECT JSON_OBJECT('exclude_list',
         JSON_ARRAYAGG(CONCAT(table_schema,'.',table_name)))
         FROM information_schema.tables
         WHERE table_schema NOT IN
         ('mysql','information_schema', 'performance_schema','sys')
         AND table_name NOT LIKE 'hw%');
mysql> CALL sys.heatwave_load(@db_list, @exc_list);
```

  You can check `db_list` and `exc_list` using `SELECT JSON_PRETTY(@db_list);` and `SELECT JSON_PRETTY(@exc_list);`

- Call Auto Parallel Load from a stored procedure:

```
DROP PROCEDURE IF EXISTS auto_load_wrapper;
DELIMITER //
CREATE PROCEDURE auto_load_wrapper()
BEGIN
  -- AUTOMATED INPUT
  SET @db_list = (SELECT JSON_ARRAYAGG(schema_name) FROM information_schema.schemata);
  SET @exc_list = (SELECT JSON_ARRAYAGG(CONCAT(table_schema,'.',table_name))
  FROM information_schema.tables WHERE table_schema = "db0");

  CALL sys.heatwave_load(@db_list, JSON_OBJECT("output","silent","exclude_list",
  CAST(@exc_list AS JSON)));

  -- CUSTOM OUTPUT
```

```
  SELECT log as 'Unsupported objects' FROM sys.heatwave_autopilot_report WHERE type="warn"
  AND stage="VERIFICATION" and log like "%Unsupported%";
  SELECT Count(*) AS "Total Load commands Generated"
  FROM sys.heatwave_autopilot_report WHERE type = "sql" ORDER BY id;

END //
DELIMITER ;

CALL auto_load_wrapper();
```

2.2.3.7 The Auto Parallel Load Report Table

MySQL 8.0.32 deprecates the `heatwave_load_report` table, and replaces it with the `heatwave_autopilot_report` table in the `sys` schema. A future release will remove it. See Section 2.7.9, “Autopilot Report Table”.

When MySQL runs Auto Parallel Load, it sends output including execution logs and a generated load script to the `heatwave_load_report` table in the `sys` schema.

The `heatwave_load_report` table is a temporary table. It contains data from the last execution of Auto Parallel Load. Data is only available for the current session and is lost when the session terminates or when the server is shut down.

Auto Parallel Load Report Table Query Examples

Query the `heatwave_load_report` table after MySQL runs Auto Parallel Load, as in the following examples:

- View error information in case Auto Parallel Load stops unexpectedly:

```
mysql> SELECT log FROM sys.heatwave_load_report WHERE type="error";
```

- View warnings to find out why tables cannot be loaded:

```
mysql> SELECT log FROM sys.heatwave_load_report WHERE type="warn";
```

- View the generated load script to see commands that would be executed by Auto Parallel Load in `normal` mode:

```
mysql> SELECT log->>"$.sql" AS "Load Script"
          FROM sys.heatwave_load_report
          WHERE type = "sql" ORDER BY id;
```

- View the number of load commands generated:

```
mysql> SELECT Count(*) AS "Total Load Commands Generated"
          FROM sys.heatwave_load_report
          WHERE type = "sql" ORDER BY id;
```

- View load script data for a particular table:

```
mysql> SELECT log->>"$.sql"
          FROM sys.heatwave_load_report
          WHERE type="sql" AND log->>"$.schema_name" = "db0" AND log->>"$.table_name" = "tbl"
          ORDER BY id;
```

- Concatenate Auto Parallel Load generated DDL statements into a single string to copy and paste for execution. The `group_concat_max_len` variable sets the result length in bytes for the `GROUP_CONCAT()` function to accommodate a potentially long string. (The default `group_concat_max_len` setting is 1024 bytes.)

```
mysql> SET SESSION group_concat_max_len = 1000000;
```

```
mysql> SELECT GROUP_CONCAT(log->>"$.sql" SEPARATOR ' ')
         FROM sys.heatwave_load_report
         WHERE type = "sql" ORDER BY id;
```

2.2.4 Monitoring Load Progress

The time required to load a table depends on data size. You can monitor load progress by issuing the following query, which returns a percentage value indicating load progress.

```
mysql> SELECT VARIABLE_VALUE
          FROM performance_schema.global_status
          WHERE VARIABLE_NAME = 'rapid_load_progress';
+----------------+
| VARIABLE_VALUE |
+----------------+
| 100.000000     |
+----------------+
```

> **Note**
>
> If necessary, you can halt a load operation using `Ctrl-C`.

2.2.5 Checking Load Status

You can check if tables are loaded by querying the `LOAD_STATUS` data from HeatWave Performance Schema tables. For example:

```
mysql> USE performance_schema;
mysql> SELECT NAME, LOAD_STATUS
          FROM rpd_tables,rpd_table_id
          WHERE rpd_tables.ID = rpd_table_id.ID;
+-----------------------------+---------------------+
| NAME                        | LOAD_STATUS         |
+-----------------------------+---------------------+
| tpch.supplier               | AVAIL_RPDGSTABSTATE |
| tpch.partsupp               | AVAIL_RPDGSTABSTATE |
| tpch.orders                 | AVAIL_RPDGSTABSTATE |
| tpch.lineitem               | AVAIL_RPDGSTABSTATE |
| tpch.customer               | AVAIL_RPDGSTABSTATE |
| tpch.nation                 | AVAIL_RPDGSTABSTATE |
| tpch.region                 | AVAIL_RPDGSTABSTATE |
| tpch.part                   | AVAIL_RPDGSTABSTATE |
+-----------------------------+---------------------+
```

The `AVAIL_RPDGSTABSTATE` status indicates that the table is loaded. For information about load statuses, see Section 7.3.9, “The rpd_tables Table”.

2.2.6 Data Compression

HeatWave compresses data as it is loaded, which permits HeatWave nodes to store more data. More data per node reduces costs by minimizing the size of the HeatWave Cluster required to store the data.

While data compression results in a smaller HeatWave Cluster, decompression operations that occur as data is accessed affect performance to a small degree. Specifically, decompression operations have a minor effect on query runtimes, on the rate at which queries are offloaded to HeatWave during change propagation, and on recovery time from Object Storage.

If data storage size is not a concern, disable data compression by setting the `rapid_compression` session variable to `OFF` before loading data:

```
mysql> SET SESSION rapid_compression=OFF;
```

The default option is AUTO which automatically chooses the best compression algorithm for each column.

2.2.7 Change Propagation

After tables are loaded into HeatWave, data changes are automatically propagated from InnoDB tables on the MySQL DB System to their counterpart tables in the HeatWave Cluster.

DML operations, INSERT, UPDATE, and DELETE, on the MySQL DB System do not wait for changes to be propagated to the HeatWave Cluster; that is, DML operations on the MySQL DB System are not delayed by HeatWave change propagation.

Data changes on the MySQL DB System node are propagated to HeatWave in batch transactions. Change propagation is initiated as follows:

- Every 200ms.
- When the change propagation buffer reaches its 64MB capacity.
- When data updated by DML operations on the MySQL DB System are read by a subsequent HeatWave query.

A change propagation failure can cause tables in HeatWave to become stale, and queries that access stale tables are not offloaded to HeatWave for processing.

Tables that have become stale due to change propagation failures resulting from out-of-code errors are automatically reloaded. A check for stale tables is performed periodically when the HeatWave Cluster is idle.

If change propagation failure has occurred for some other reason causing a table to become stale, you must unload and reload the table manually to restart change propagation for the table. See Section 2.5.1, “Unloading Tables”, and Section 2.2, “Loading Data to HeatWave MySQL”.

To check if change propagation is enabled globally, query the rapid_change_propagation_status variable:

```
mysql> SELECT VARIABLE_VALUE
          FROM performance_schema.global_status
          WHERE VARIABLE_NAME = 'rapid_change_propagation_status';
+----------------+
| VARIABLE_VALUE |
+----------------+
| ON             |
+----------------+
```

To check if change propagation is enabled for individual tables, query the POOL_TYPE data in HeatWave Performance Schema tables. RAPID_LOAD_POOL_TRANSACTIONAL indicates that change propagation is enabled for the table. RAPID_LOAD_POOL_SNAPSHOT indicates that change propagation is disabled.

```
mysql> SELECT NAME, POOL_TYPE
          FROM rpd_tables,rpd_table_id
          WHERE rpd_tables.ID = rpd_table_id.ID AND SCHEMA_NAME LIKE 'tpch';
+---------------+-------------------------------+
| NAME          | POOL_TYPE                     |
+---------------+-------------------------------+
| tpch.orders   | RAPID_LOAD_POOL_TRANSACTIONAL |
| tpch.region   | RAPID_LOAD_POOL_TRANSACTIONAL |
| tpch.lineitem | RAPID_LOAD_POOL_TRANSACTIONAL |
| tpch.supplier | RAPID_LOAD_POOL_TRANSACTIONAL |
| tpch.partsupp | RAPID_LOAD_POOL_TRANSACTIONAL |
| tpch.part     | RAPID_LOAD_POOL_TRANSACTIONAL |
```

```
| tpch.customer | RAPID_LOAD_POOL_TRANSACTIONAL |
+---------------+-------------------------------+
```

See Section 2.18.1, “Change Propagation Limitations”.

2.2.8 Reload Tables

HeatWave MySQL can reload all tables.

```
mysql> CALL sys.heatwave_reload ([options]);

options: {
     JSON_OBJECT("key","value"[,"key","value"] ...)
          "key","value": {
          ["only_user_loaded_tables",{true|false}]
          ["output",{"normal"|"silent"}]
          }
}
```

Use key-value pairs in `JSON` format to specify *`options`*. HeatWave uses the default setting if there is no defined option. Use `NULL` to specify no arguments.

The *`options`* include:

- `only_user_loaded_tables`: Enabled by default. The procedure only reloads user tables. If `false`, the procedure reloads user and system tables.
- `output`: Defines where to send the output. Permitted values are:
  - `normal`: The default. Produces summarized output and sends it to `stdout` and to the `heatwave_autopilot_report` table. See Section 2.7.9, “Autopilot Report Table”.
  - `silent`: Sends output to the `heatwave_autopilot_report` table only. See Section 2.7.9, “Autopilot Report Table”. The `silent` output type is useful if human-readable output is not required; when the output is consumed by a script, for example.

Syntax Examples

- Reload all tables with default options:

  ```
  mysql> CALL sys.heatwave_reload (NULL);
  ```

  ```
  mysql> CALL sys.heatwave_reload (JSON_OBJECT());
  ```

- Reload all user and system tables:

  ```
  mysql> CALL sys.heatwave_reload (JSON_OBJECT("only_user_loaded_tables",false));
  ```

- Reload user tables with the `silent` option:

  ```
  mysql> CALL sys.heatwave_reload (JSON_OBJECT("output","silent"));
  ```

- Reload all user and system tables with the `silent` option:

  ```
  mysql> CALL sys.heatwave_reload (JSON_OBJECT("only_user_loaded_tables",false,"output","silent"));
  ```

2.3 Running Queries

When HeatWave is enabled and the data you want to query is loaded, queries that qualify are automatically offloaded from the MySQL DB System to HeatWave for accelerated processing. No special

action is required. Simply run the query from a client, application, or interface that is connected to the MySQL DB System associated with the HeatWave Cluster. After HeatWave processes a query, results are sent back to the MySQL DB System and to the client, application, or interface that issued the query.

As of MySQL 8.4.0, HeatWave supports InnoDB partitions. Query processing in HeatWave can access partitions with standard syntax.

For information about connecting to a MySQL DB System on HeatWave Service:

- For HeatWave on OCI see Connecting to a DB System in the *HeatWave on OCI Service Guide*.
- For HeatWave on AWS, see Connecting from a Client in the *HeatWave on AWS Service Guide*.

For related best practices, see Section 2.9, “Best Practices”.

2.3.1 Query Prerequisites

The following prerequisites apply for offloading queries to HeatWave:

- The query must be a `SELECT` statement. `INSERT ... SELECT` and `CREATE TABLE ... SELECT` statements are supported, but only the `SELECT` portion of the statement is offloaded to HeatWave. See Section 2.3.8, “CREATE TABLE ... SELECT Statements”, and Section 2.3.9, “INSERT ... SELECT Statements”.
- All tables accessed by the query must be defined with `RAPID` as the secondary engine. See Section 2.2.2.2, “Defining the Secondary Engine”.
- All tables accessed by the query must be loaded in HeatWave. See Section 2.2, “Loading Data to HeatWave MySQL”.
- `autocommit` must be enabled. If `autocommit` is disabled, queries are not offloaded and execution is performed on the MySQL DB System. To check the `autocommit` setting:

```
mysql> SHOW VARIABLES LIKE 'autocommit';
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| autocommit    | ON    |
+---------------+-------+
```

- Queries must only use supported functions and operators. See Section 2.12, “Supported Functions and Operators”.
- Queries must avoid known limitations. See Section 2.18, “HeatWave MySQL Limitations”.

If any prerequisite is not satisfied, the query is not offloaded and falls back to the MySQL DB System for processing by default.

2.3.2 Running Queries

Before running a query, you can use `EXPLAIN` to determine if the query will be offloaded to HeatWave for processing. If so, the `Extra` column of `EXPLAIN` output shows: “`Using secondary engine RAPID`”.

```
mysql> EXPLAIN SELECT O_ORDERPRIORITY, COUNT(*) AS ORDER_COUNT
          FROM orders
          WHERE O_ORDERDATE >= DATE '1994-03-01'
          GROUP BY O_ORDERPRIORITY
```

```
          ORDER BY O_ORDERPRIORITY;
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: orders
   partitions: NULL
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 14862970
     filtered: 33.33
        Extra: Using where; Using temporary; Using filesort; Using secondary
        engine RAPID
```

If `Using secondary engine RAPID` does not appear in the `Extra` column, the query will not be offloaded to HeatWave. To determine why a query will not offload, refer to Section 2.15, “Troubleshooting”, or try debugging the query using the procedure described in Section 2.3.6, “Debugging Queries”.

After using `EXPLAIN` to verify that the query can be offloaded, run the query and note the execution time.

```
mysql> SELECT O_ORDERPRIORITY, COUNT(*) AS ORDER_COUNT
          FROM orders
          WHERE O_ORDERDATE >= DATE '1994-03-01'
          GROUP BY O_ORDERPRIORITY
          ORDER BY O_ORDERPRIORITY;
+-----------------+-------------+
| O_ORDERPRIORITY | ORDER_COUNT |
+-----------------+-------------+
| 1-URGENT        |     2017573 |
| 2-HIGH          |     2015859 |
| 3-MEDIUM        |     2013174 |
| 4-NOT SPECIFIED |     2014476 |
| 5-LOW           |     2013674 |
+-----------------+-------------+
5 rows in set (0.04 sec)
```

To compare HeatWave query execution time with MySQL DB System execution time, disable the `use_secondary_engine` variable and run the query again to see how long it takes to run on the MySQL DB System.

```
mysql> SET SESSION use_secondary_engine=OFF;

mysql> SELECT O_ORDERPRIORITY, COUNT(*) AS ORDER_COUNT
           FROM orders
           WHERE O_ORDERDATE >= DATE '1994-03-01'
           GROUP BY O_ORDERPRIORITY
           ORDER BY O_ORDERPRIORITY;
+-----------------+-------------+
| O_ORDERPRIORITY | ORDER_COUNT |
+-----------------+-------------+
| 1-URGENT        |     2017573 |
| 2-HIGH          |     2015859 |
| 3-MEDIUM        |     2013174 |
| 4-NOT SPECIFIED |     2014476 |
| 5-LOW           |     2013674 |
+-----------------+-------------+
5 rows in set (8.91 sec)
```

> **Note**
>
> Concurrently issued queries are prioritized for execution. For information about query prioritization, see Section 2.3.3, “Auto Scheduling”.

2.3.3 Auto Scheduling

HeatWave MySQL uses a workload-aware, priority-based, automated scheduling mechanism to schedule concurrently issued queries for execution. The scheduling mechanism prioritizes short-running queries but considers wait time in the queue so that costlier queries are eventually scheduled for execution. This scheduling approach reduces query execution wait times overall.

Concurrency queues queries within a HeatWave Cluster, and is not the same as replication.

When a HeatWave Cluster is idle, an arriving query is immediately scheduled for execution. It is not queued. A query is queued only if the HeatWave Cluster is running a preceding query.

A light-weight cost estimate is performed for each query at query compilation time.

Queries cancelled via `Ctrl-C` are removed from the scheduling queue.

To view the HeatWave query history including query start time, end time, and wait time in the scheduling queue, see Section 7.1, “HeatWave MySQL Monitoring”.

MySQL 9.0.0 introduces a novel scheduler that a HeatWave Cluster uses to run multiple concurrent queries before placing further queries on the queue. The scheduler can interleave queries inside each HeatWave node within a HeatWave Cluster to achieve the following scheduling improvements:

- Maximise query processing throughput.
- Make the query wait time proportional to the query execution time.
- Ensure that queries only receive an out of memory error if they would also receive an out of memory error when they run in isolation. If a query does raise an out of memory error, the scheduler will run the query a second time in isolation.

The scheduler can run up to 12 concurrent queries within a HeatWave Cluster. The actual number of concurrent queries depends upon the predicted memory requirements, and available memory.

The scheduler runs HeatWave AutoML, HeatWave GenAI, HeatWave Lakehouse and data load queries one at a time.

- The scheduler can run multiple analytical queries if there is no more than one data load query running.
- The scheduler can run multiple data load queries if there is no more than one analytical query running.

2.3.4 Auto Query Plan Improvement

The *Auto Query Plan Improvement* feature collects and stores query plan statistics in a statistics cache when a query is executed in HeatWave. When a new query shares query execution plan nodes with previously executed queries, the statistics collected from previously executed queries are used instead of estimated statistics, which improves query execution plans, cost estimations, execution times, and memory efficiency.

Each entry in the cache corresponds to a query execution plan node. A query execution plan may have nodes for table scans, `JOIN`, `GROUP BY` and other operations.

The statistics cache is an LRU structure. When cache capacity is reached, the least recently used entries are evicted from the cache as new entries are added. The number of entries permitted in the statistics cache is 65536, which is enough to store statistics for 4000 to 5000 unique queries of medium

complexity. The maximum number of statistics cache entries is defined by the MySQL-managed `rapid_stats_cache_max_entries` setting.

2.3.5 Dynamic Query Offload

Before MySQL 9.0.0, HeatWave MySQL uses a query cost threshold to choose the execution engine for the query.

As of MySQL 9.0.0, HeatWave MySQL uses dynamic query offload to choose the optimal execution engine for queries based on query and execution engine characteristics, which achieves improved performance.

Dynamic query offload analyses the query characteristics and execution engine static and dynamic characteristics to choose the best engine for the query, given the current system state. This includes the following:

- Secondary engine change propagation.
- Secondary engine query queue.
- The presence of indexes on the primary engine.

Dynamic query offload introduces a minimal overhead for fast queries. The overhead for more complex queries is also minimal, compared to the time required for query optimisation and subsequent execution.

2.3.6 Debugging Queries

This section describes how to debug queries that fail to offload to HeatWave for execution. Query debugging is performed by enabling MySQL optimizer trace and querying the `INFORMATION_SCHEMA.OPTIMIZER_TRACE` table for the failure reason.

1. To enable MySQL optimizer trace, set the `optimizer_trace` and `optimizer_trace_offset` variables as shown:

```
mysql> SET SESSION optimizer_trace="enabled=on";
mysql> SET optimizer_trace_offset=-2;
```

2. Issue the problematic query using `EXPLAIN`. If the query is supported by HeatWave, the `Extra` column in the `EXPLAIN` output shows the following text: “`Using secondary engine RAPID`”; otherwise, that text does not appear. The following query example uses the `TIMEDIFF()` function, which is currently not supported by HeatWave:

```
mysql> EXPLAIN SELECT TIMEDIFF(O_ORDERDATE,'2000:01:01 00:00:00.000001')
           FROM orders;
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: ORDERS
   partitions: NULL
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 1488248
     filtered: 100
        Extra: NULL
1 row in set, 1 warning (0.0011 sec)
```

3. Query the `INFORMATION_SCHEMA.OPTIMIZER_TRACE` table for a failure reason. There are two trace markers for queries that fail to offload:

- `Rapid_Offload_Fails`
- `secondary_engine_not_used`

To query for the `Rapid_Offload_Fails` trace marker, issue this query:

```
mysql> SELECT QUERY, TRACE->'$**.Rapid_Offload_Fails'
         FROM INFORMATION_SCHEMA.OPTIMIZER_TRACE;
```

To query for the `secondary_engine_not_used` trace marker, issue this query:

```
mysql> SELECT QUERY, TRACE->'$**.secondary_engine_not_used'
         FROM INFORMATION_SCHEMA.OPTIMIZER_TRACE;
```

> **Note**
>
> If the optimizer trace does not return all of the trace information, increase the optimizer trace buffer size. For more information, see Section 2.9.7, “Running Queries”.

For the `TIMEDIFF()` query example used above, querying the `Rapid_Offload_Fails` marker returns the reason for the failure:

```
mysql> SELECT QUERY, TRACE->'$**.Rapid_Offload_Fails'
          FROM INFORMATION_SCHEMA.OPTIMIZER_TRACE;
+--------------------------------------------------+--------------------------------------+
| QUERY                                            | TRACE->'$**.Rapid_Offload_Fails'     |
+--------------------------------------------------+--------------------------------------+
| EXPLAIN SELECT                                   |[{"Reason": "Function timediff is not |
| TIMEDIFF(O_ORDERDATE,'2000:01:01 00:00:00.000001')|             yet supported"}]         |
| FROM ORDERS                                      |                                      |
+--------------------------------------------------+--------------------------------------+
```

The reason reported for a query offload failure depends on the issue or limitation encountered. For common issues, such as unsupported clauses or functions, a specific reason is reported. For undefined issues or unsupported query transformations performed by the optimizer, the following generic reason is reported:

```
mysql> [{"Reason": "Currently unsupported RAPID query compilation scenario"}]
```

For a query that does not meet the query cost threshold for HeatWave, the following reason is reported:

```
mysql> [{"Reason": "The estimated query cost does not exceed secondary_engine_cost_threshold."}]
```

The query cost threshold prevents queries of little cost from being offloaded to HeatWave. For information about the query cost threshold, see Section 2.15, “Troubleshooting”.

For a query that attempts to access a column defined as `NOT SECONDARY`, the following reason is reported:

```
mysql> [{"Reason": "Column risk_assessment is marked as NOT SECONDARY."}]
```

Columns defined as `NOT SECONDARY` are excluded when a table is loaded into HeatWave. See Section 2.2.2.1, “Excluding Table Columns”.

2.3.7 Query Runtimes and Estimates

To view HeatWave query runtimes and runtime estimates use HeatWave Autopilot Advisor Auto Query Time Estimation, see: Section 2.7.6, “Auto Query Time Estimation”, or query the

`performance_schema.rpd_query_stats` table. Runtime data is useful for query optimization, troubleshooting, and estimating the cost of running a particular query or workload.

HeatWave query runtime data includes:

- Runtimes for successfully executed queries
- Runtime estimates for `EXPLAIN` queries
- Runtime estimates for queries cancelled using `Ctrl+C`
- Runtime estimates for queries that fail due to an out-of-memory error

Runtime data is available for queries in the HeatWave query history, which is a non-persistent store of information about the last 1000 executed queries.

To use Auto Query Time Estimation:

- To view runtime data for all queries in the HeatWave history:

```
mysql> CALL sys.heatwave_advisor(JSON_OBJECT("query_insights", true));
```

- To view runtime data for queries executed by the current session only:

```
mysql> CALL sys.heatwave_advisor(JSON_OBJECT("query_insights", true,
         "query_session_id", JSON_ARRAY(connection_id())));
```

See: Section 2.7.6, “Auto Query Time Estimation”.

To use the rpd_query_stats table:

- To view runtime data for all queries in the HeatWave query history:

```
mysql> SELECT query_id,
          JSON_EXTRACT(JSON_UNQUOTE(qkrn_text->'$**.sessionId'),'$[0]') AS session_id,
          JSON_EXTRACT(JSON_UNQUOTE(qkrn_text->'$**.accumulatedRapidCost'),'$[0]') AS time_in_ns,
          JSON_EXTRACT(JSON_UNQUOTE(qexec_text->'$**.error'),'$[0]') AS error_message
          FROM performance_schema.rpd_query_stats;
```

- To view runtime data for a particular HeatWave query, filtered by query ID:

```
mysql> SELECT query_id,
          JSON_EXTRACT(JSON_UNQUOTE(qkrn_text->'$**.sessionId'),'$[0]') AS session_id,
          JSON_EXTRACT(JSON_UNQUOTE(qkrn_text->'$**.accumulatedRapidCost'),'$[0]') AS time_in_ns,
          JSON_EXTRACT(JSON_UNQUOTE(qexec_text->'$**.error'),'$[0]') AS error_message
          FROM performance_schema.rpd_query_stats
          WHERE query_id = 1;
```

- `EXPLAIN` output includes the query ID. You can also query the `performance_schema.rpd_query_stats` table for query IDs:

```
mysql> SELECT query_id, LEFT(query_text,160)
          FROM performance_schema.rpd_query_stats;
```

2.3.8 CREATE TABLE ... SELECT Statements

The `SELECT` query of a `CREATE TABLE ... SELECT` statement is offloaded to HeatWave for execution, and the table is created on the MySQL DB System. Offloading the `SELECT` query to HeatWave reduces `CREATE TABLE ... SELECT` execution time in cases where the `SELECT` query is long running and complex. `SELECT` queries that produce large result sets do not benefit from this feature due to the large number of DML operations performed on the MySQL DB System instance.

The SELECT table must be loaded in HeatWave. For example, the following statement selects data from the orders table on HeatWave and inserts the result set into the orders2 table created on the MySQL DB System:

```
mysql> CREATE TABLE orders2 SELECT * FROM orders;
```

The SELECT portion of the CREATE TABLE ... SELECT statement is subject to the same HeatWave requirements and limitations as regular SELECT queries.

2.3.9 INSERT ... SELECT Statements

The SELECT query of an INSERT ... SELECT statement is offloaded to HeatWave for execution, and the result set is inserted into the specified table on the MySQL DB System. Offloading the SELECT query to HeatWave reduces INSERT ... SELECT execution time in cases where the SELECT query is long running and complex. SELECT queries that produce large result sets do not benefit from this feature due to the large number of DML operations performed on the MySQL DB System instance.

The SELECT table must be loaded in HeatWave, and the INSERT table must be present on the MySQL DB System. For example, the following statement selects data from the orders table on tHeatWave and inserts the result set into the orders2 table on the MySQL DB System:

```
mysql> INSERT INTO orders2 SELECT * FROM orders;
```

Usage notes:

- The SELECT portion of the INSERT ... SELECT statement is subject to the same HeatWave requirements and limitations as regular SELECT queries.
- Functions, operators, and attributes deprecated by MySQL Server are not supported in the SELECT query.
- The ON DUPLICATE KEY UPDATE clause is not supported.
- INSERT INTO *some_view* SELECT statements are not offloaded. Setting use_secondary_engine=FORCED does not cause the statement to fail with an error in this case. The statement is executed on the MySQL DB System regardless of the use_secondary_engine setting.
- See Section 2.3.1, “Query Prerequisites” and Section 2.18.9, “Other Limitations”.

2.3.10 Using Views

HeatWave supports querying views. The table or tables upon which a view is created must be loaded in HeatWave. Queries executed on views are subject to the same offload prerequisites and limitations as queries executed on tables.

In the following example, a view is created on the orders table, described in Section 2.6, “Table Load and Query Example”. The example assumes the orders table is loaded in HeatWave.

```
mysql> CREATE VIEW v1 AS SELECT O_ORDERPRIORITY, O_ORDERDATE
         FROM orders;
```

To determine if a query executed on a view can be offloaded to HeatWave for execution, use EXPLAIN. If offload is supported, the Extra column of EXPLAIN output shows “Using secondary engine RAPID”, as in the following example:

```
mysql> EXPLAIN SELECT O_ORDERPRIORITY, COUNT(*) AS ORDER_COUNT
```

```
          FROM v1
          WHERE O_ORDERDATE >= DATE '1994-03-01'
          GROUP BY O_ORDERPRIORITY
          ORDER BY O_ORDERPRIORITY;
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: ORDERS
   partitions: NULL
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 1488248
     filtered: 33.32999801635742
        Extra: Using where; Using temporary; Using filesort; Using secondary engine RAPID
```

2.4 Modifying Tables

HeatWave MySQL can alter tables with DDL operations without unloading them.

2.5 Unloading Data from HeatWave MySQL

2.5.1 Unloading Tables

Unloading a table from HeatWave may be necessary to replace an existing table, to reload a table, to free up memory, or simply to remove a table that is no longer used.

To unload a table from HeatWave, specify the `SECONDARY_UNLOAD` clause in an `ALTER TABLE` statement:

```
mysql> ALTER TABLE tbl_name SECONDARY_UNLOAD;
```

Data is removed from HeatWave only. The table contents on the MySQL DB System are not affected.

2.5.2 Unloading Partitions

As of MySQL 8.4.0, HeatWave supports InnoDB partitions. To unload partitions from HeatWave, specify the `SECONDARY_UNLOAD` clause in an `ALTER TABLE` statement.

```
mysql> ALTER TABLE tbl_name SECONDARY_UNLOAD PARTITION (p0, p1, ..., pn);
```

If a partition unload fails, the table becomes stale.

See: Section 2.18.6, “Partition Selection Limitations”.

2.5.3 Unloading Data Using Auto Unload

Auto Unload facilitates the process of unloading data from HeatWave by automating many of the steps involved, including:

- Excluding schemas and tables that cannot be unloaded.
- Removes the secondary engine flag for tables that are to be unloaded.
- Unloading data from HeatWave.

Auto Unload, which can be run from any MySQL client or connector, is implemented as a stored procedure named `heatwave_unload`, which resides in the MySQL `sys` schema. Running Auto Unload involves issuing a `CALL` statement for the stored procedure, which takes `schemas` and *`options`* as arguments; for example, this statement unloads the `tpch` schema:

```
mysql> CALL sys.heatwave_unload(JSON_ARRAY("tpch"),NULL);
```

2.5.3.1 Auto Unload Syntax

MySQL 8.4.0 adds an `input_list` JSON array that replaces the `db_list` JSON array. This adds an include list to exactly specify the tables to unload. It is no longer necessary to include a complete schema, and exclude unnecessary tables. `input_list` is backwards compatible with `db_list`.

```
mysql> CALL sys.heatwave_unload (input_list,[options]);

input_list: {
     JSON_ARRAY(input [,input] ...)
}

options: {
     JSON_OBJECT("key","value"["key","value"] ...)
          "key","value": {
          ["mode",{"normal"|"dryrun"}]
          ["output",{"normal"|"silent"|"help"}]
          }
}

input: {
     'db_name' | db_object
}

db_object: {
     JSON_OBJECT("key","value"["key","value"] ...)
          "key","value": {
          "db_name": "db_name",
          ["tables": JSON_ARRAY(table [, table] ...)]
          ["exclude_tables": JSON_ARRAY(table [, table] ...)]
          }
}

table: {
     'table_name'
}
```

Before MySQL 8.4.0:

```
mysql> CALL sys.heatwave_unload (db_list,[options]);

db_list: {
     JSON_ARRAY(["schema_name","schema_name"] ...)

options: {
     JSON_OBJECT("key","value"["key","value"] ...)
          "key","value": {
          ["mode",{"normal"|"dryrun"}]
          ["output",{"normal"|"silent"|"help"}]
          ["exclude_list",JSON_ARRAY(schema_name_1, schema_name_2.table_name_1, ...)]
}
```

As of MySQL 8.4.0 use `input_list` to define what to unload. `input_list` is a JSON array and requires one or more valid `input` which can be either a valid schema name or a `db_object`. An empty array is permitted to view the Auto Unload command-line help, see Section 2.5.3.3, “Auto Unload Command-Line Help”. This is backwards compatible with `db_list`.

Before MySQL 8.4.0, `db_list` specifies the schemas to unload. The list is a JSON array and requires one or more valid schema names. An empty array is permitted to view the Auto Unload command-line help.

Use key-value pairs in `JSON` format to specify parameters. HeatWave uses the default setting if there is no option setting. Use `NULL` to specify no arguments.

For syntax examples, see Section 2.5.3.4, “Auto Unload Examples”.

Auto Unload *`options`* include:

- `mode`: Defines the Auto Unload operational mode. Permitted values are:
  - `normal`: The default. Generates and executes the unload script.
  - `dryrun`: Generates an unload script only. Auto Unload executes in `dryrun` mode automatically if the HeatWave Cluster is not active.
- `output`: Defines how Auto Unload produces output. Permitted values are:
  - `normal`: The default. Produces summarized output and sends it to `stdout` and to the `heatwave_autopilot_report` table. See Section 2.7.9, “Autopilot Report Table”.
  - `silent`: Sends output to the `heatwave_autopilot_report` table only. See Section 2.7.9, “Autopilot Report Table”. The `silent` output type is useful if human-readable output is not required; when the output is consumed by a script, for example. For an example of a stored procedure with an Auto Unload call that uses the `silent` output type, see Section 2.5.3.4, “Auto Unload Examples”.
  - `help`: Displays Auto Unload command-line help. See Section 2.5.3.3, “Auto Unload Command-Line Help”.
- `exclude_list`: Defines a list of schemas and tables to exclude from the unload script. Names must be fully qualified without backticks.

  Do not use as of MySQL 8.4.0. Use `db_object` with `tables` or `exclude_tables` instead. `exclude_list` will be deprecated in a future release.

  Auto Unload automatically excludes tables that are loading, unloading or in recovery. This is when the `load_status` is one of the following: `NOLOAD_RPDGSTABSTATE`, `LOADING_RPDGSTABSTATE`, `UNLOADING_RPDGSTABSTATE` or `INRECOVERY_RPDGSTABSTATE`.

The `db_object` is a JSON object literal that includes:

- `db_name`: The name of the database to load.
- Use one or other of the following, but not both. The use of both parameters will throw an error.
  - `tables`: An optional JSON array of `table` to include in the load.
  - `exclude_tables`: As of MySQL 8.4.0, an optional JSON array of `table` to exclude from the load.
- `table`: A valid table name.

2.5.3.2 Running Auto Unload

Run Auto Unload in `dryrun` mode first to check for errors and warnings and to inspect the generated unload script. To unload a single schema in `dryrun` mode:

```
mysql> CALL sys.heatwave_unload(JSON_ARRAY("tpch"), JSON_OBJECT("mode","dryrun"));
```

In `dryrun` mode, Auto Unload sends the unload script to the `heatwave_autopilot_report` table only. See Section 2.7.9, “Autopilot Report Table”.

If Auto Unload fails with an error, inspect the errors with a query to the `heatwave_autopilot_report` table.

```
mysql> SELECT log FROM sys.heatwave_autopilot_report
          WHERE type="error";
```

When Auto Unload finishes running, query the `heatwave_autopilot_report` table to check for warnings.

```
mysql> SELECT log FROM sys.heatwave_autopilot_report
          WHERE type="warn";
```

Issue the following query to inspect the unload script that was generated.

```
mysql> SELECT log->>"$.sql" AS "Unload Script"
          FROM sys.heatwave_autopilot_report
          WHERE type = "sql" ORDER BY id;
```

Once you are satisfied with the Auto Unload `CALL` statement and the generated unload script, reissue the `CALL` statement in `normal` mode to unload the data into HeatWave. For example:

```
mysql> CALL sys.heatwave_unload(JSON_ARRAY("tpch"), JSON_OBJECT("mode","normal"));
```

> **Note**
>
> Retrieve DDL statements in a table or use the following statements to produce a list of DDL statements to easily copy and paste.
>
> ```
> mysql> SET SESSION group_concat_max_len = 1000000;
> mysql> SELECT GROUP_CONCAT(log->>"$.sql" SEPARATOR ' ')
>           FROM sys.heatwave_autopilot_report
>           WHERE type = "sql" ORDER BY id;
> ```

The time required to unload data depends on the data size.

Tables are unloaded in sequence, ordered by schema and table name. Unload-time errors are reported as they are encountered. If an error is encountered while unloading a table, the operation is not terminated. Auto Unload continues running, moving on to the next table.

When Auto Unload finishes running, it checks if tables are unloaded and shows a summary with the number of tables that were unloaded and the number of tables that failed to unload.

2.5.3.3 Auto Unload Command-Line Help

To view Auto Unload command-line help, issue the following statement:

```
mysql> CALL sys.heatwave_unload(JSON_ARRAY(),JSON_OBJECT("output","help"));
```

The command-line help provides usage documentation for the Auto Unload utility.

2.5.3.4 Auto Unload Examples

- Unload a single schema with default options.

  ```
  mysql> CALL sys.heatwave_unload(JSON_ARRAY("tpch"),NULL);
  ```

- Unload multiple schemas with default options.

```
mysql> CALL sys.heatwave_unload(JSON_ARRAY("tpch","airportdb","employees","sakila"),NULL);
```

- Unload multiple schemas, excluding specified tables:

```
mysql> CALL sys.heatwave_unload(JSON_ARRAY("tpch","airportdb"),
         JSON_OBJECT("exclude_list",JSON_ARRAY("tpch.orders")));
```

- Unload tables that begin with an “hw” prefix from a schema named `schema_customer_1`.

```
mysql> SET @exc_list = (SELECT JSON_OBJECT('exclude_list',
         JSON_ARRAYAGG(CONCAT(table_schema,'.',table_name)))
         FROM performance_schema.rpd_table_id
         WHERE schema_name = 'schema_customer_1'
         AND table_name NOT LIKE 'hw%');
mysql> CALL sys.heatwave_unload(JSON_ARRAY('schema_customer_1'), @exc_list);
```

- Unload all schemas with tables that start with an “hw” prefix.

```
mysql> SET @db_list = (SELECT JSON_ARRAYAGG(unique_schemas)
         FROM (SELECT DISTINCT(schema_name) as unique_schemas
         FROM performance_schema.rpd_table_id)
         AS loaded_schemas);
mysql> SET @exc_list = (SELECT JSON_OBJECT('exclude_list',
         JSON_ARRAYAGG(CONCAT(table_schema,'.',table_name)))
         FROM performance_schema.rpd_table_id
         WHERE table_name NOT LIKE 'hw%');
mysql> CALL sys.heatwave_unload(@db_list, @exc_list);
```

  Check `db_list` and `exc_list` with `SELECT JSON_PRETTY(@db_list);` and `SELECT JSON_PRETTY(@exc_list);`

- Call Auto Unload from a stored procedure:

```
DROP PROCEDURE IF EXISTS auto_unload_wrapper;
DELIMITER //
CREATE PROCEDURE auto_unload_wrapper()
BEGIN
  -- AUTOMATED INPUT
  SET @db_list = (SELECT JSON_ARRAYAGG(unique_schemas) FROM (SELECT DISTINCT(schema_name) as unique_sche
  SET @exc_list = (SELECT JSON_ARRAYAGG(CONCAT(schema_name,'.',table_name))
                  FROM performance_schema.rpd_table_id
                  WHERE table_name NOT LIKE 'hw%');
  CALL sys.heatwave_unload(@db_list, JSON_OBJECT("output","silent","exclude_list",
  CAST(@exc_list AS JSON)));

  -- CUSTOM OUTPUT
  SELECT log as 'Warnings' FROM sys.heatwave_autopilot_report WHERE type="warn";
  SELECT Count(*) AS "Total Unload commands Generated"
  FROM sys.heatwave_autopilot_report WHERE type = "sql" ORDER BY id;

END //
DELIMITER ;

CALL auto_unload_wrapper();
```

2.5.4 Unload All Tables

HeatWave MySQL can unload all tables.

```
mysql> CALL sys.heatwave_unload_all ([options]);

options: {
     JSON_OBJECT("key","value"["key","value"] ...)
          "key","value": {
```

```
          ["only_user_loaded_tables",{true|false}]
          ["output",{"normal"|"silent"}]
          }
}
```

Use key-value pairs in `JSON` format to specify *`options`*. HeatWave uses the default setting if there is no defined option. Use `NULL` to specify no arguments.

The *`options`* include:

- `only_user_loaded_tables`: Enabled by default. The procedure only unloads user tables. If `false`, the procedure unloads user and system tables.
- `output`: Defines where to send the output. Permitted values are:
  - `normal`: The default. Produces summarized output and sends it to `stdout` and to the `heatwave_autopilot_report` table. See Section 2.7.9, “Autopilot Report Table”.
  - `silent`: Sends output to the `heatwave_autopilot_report` table only. See Section 2.7.9, “Autopilot Report Table”. The `silent` output type is useful if human-readable output is not required; when the output is consumed by a script, for example.

Syntax Examples

- Unload all tables with default options:

```
mysql> CALL sys.heatwave_unload_all (NULL);
```

```
mysql> CALL sys.heatwave_unload_all (JSON_OBJECT());
```

- Unload all user and system tables:

```
mysql> CALL sys.heatwave_unload_all (JSON_OBJECT("only_user_loaded_tables",false));
```

- Unload user tables with the `silent` option:

```
mysql> CALL sys.heatwave_unload_all (JSON_OBJECT("output","silent"));
```

- Unload all user and system tables with the `silent` option:

```
mysql> CALL sys.heatwave_unload_all (JSON_OBJECT("only_user_loaded_tables",false,"output","silent"));
```

2.6 Table Load and Query Example

The following example demonstrates preparing and loading a table into HeatWave manually and executing a query.

It is assumed that HeatWave is enabled and the MySQL DB System has a schema named `tpch` with a table named `orders`. The example shows how to exclude a table column, encode string columns, define `RAPID` as the secondary engine, and load the table. The example also shows how to use `EXPLAIN` to verify that the query can be offloaded, and how to force query execution on the MySQL DB System to compare MySQL DB System and HeatWave query execution times.

1. The table used in this example:

```
mysql> USE tpch;
mysql> SHOW CREATE TABLE orders;
*************************** 1. row ***************************
       Table: orders
Create Table: CREATE TABLE `orders` (
  `O_ORDERKEY` int NOT NULL,
```

```
  `O_CUSTKEY` int NOT NULL,
  `O_ORDERSTATUS` char(1) COLLATE utf8mb4_bin NOT NULL,
  `O_TOTALPRICE` decimal(15,2) NOT NULL,
  `O_ORDERDATE` date NOT NULL,
  `O_ORDERPRIORITY` char(15) COLLATE utf8mb4_bin NOT NULL,
  `O_CLERK` char(15) COLLATE utf8mb4_bin NOT NULL,
  `O_SHIPPRIORITY` int NOT NULL,
  `O_COMMENT` varchar(79) COLLATE utf8mb4_bin NOT NULL,
  PRIMARY KEY (`O_ORDERKEY`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin
```

2. Exclude columns that you do not want to load, such as columns with unsupported data types:

```
mysql> ALTER TABLE orders MODIFY `O_COMMENT` varchar(79) NOT NULL NOT SECONDARY;
```

3. Encode individual string columns as necessary. For example, apply dictionary encoding to string columns with a low number of distinct values. Variable-length encoding is the default if no encoding is specified.

```
mysql> ALTER TABLE orders MODIFY `O_ORDERSTATUS` char(1) NOT NULL
          COMMENT 'RAPID_COLUMN=ENCODING=SORTED';

mysql> ALTER TABLE orders MODIFY `O_ORDERPRIORITY` char(15) NOT NULL
          COMMENT 'RAPID_COLUMN=ENCODING=SORTED';

mysql> ALTER TABLE orders MODIFY `O_CLERK` char(15) NOT NULL
          COMMENT 'RAPID_COLUMN=ENCODING=SORTED';
```

4. Define `RAPID` as the secondary engine for the table.

```
mysql> ALTER TABLE orders SECONDARY_ENGINE RAPID;
```

5. Verify the table definition changes:

```
mysql> SHOW CREATE TABLE orders;
*************************** 1. row ***************************
       Table: orders
Create Table: CREATE TABLE `orders` (
  `O_ORDERKEY` int NOT NULL,
  `O_CUSTKEY` int NOT NULL,
  `O_ORDERSTATUS` char(1) COLLATE utf8mb4_bin NOT NULL COMMENT
    'RAPID_COLUMN=ENCODING=SORTED',
  `O_TOTALPRICE` decimal(15,2) NOT NULL,
  `O_ORDERDATE` date NOT NULL,
  `O_ORDERPRIORITY` char(15) COLLATE utf8mb4_bin NOT NULL COMMENT
    'RAPID_COLUMN=ENCODING=SORTED',
  `O_CLERK` char(15) COLLATE utf8mb4_bin NOT NULL COMMENT 'RAPID_COLUMN=ENCODING=SORTED',
  `O_SHIPPRIORITY` int NOT NULL,
  `O_COMMENT` varchar(79) COLLATE utf8mb4_bin NOT NULL NOT SECONDARY,
  PRIMARY KEY (`O_ORDERKEY`)
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin SECONDARY_ENGINE=RAPID
```

6. Load the table into HeatWave.

```
mysql> ALTER TABLE orders SECONDARY_LOAD;
```

7. Use `EXPLAIN` to determine if a query on the orders table can be offloaded. `Using secondary engine RAPID` in the `Extra` column indicates that the query can be offloaded.

```
mysql> EXPLAIN SELECT O_ORDERPRIORITY, COUNT(*) AS ORDER_COUNT
          FROM orders
          WHERE O_ORDERDATE >= DATE '1994-03-01'
          GROUP BY O_ORDERPRIORITY
          ORDER BY O_ORDERPRIORITY;
*************************** 1. row ***************************
```

```
           id: 1
  select_type: SIMPLE
        table: orders
   partitions: NULL
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 14862970
     filtered: 33.33
        Extra: Using where; Using temporary; Using filesort; Using secondary
        engine RAPID
1 row in set, 1 warning (0.00 sec)
```

8. Execute the query and note the execution time:

```
mysql> SELECT O_ORDERPRIORITY, COUNT(*) AS ORDER_COUNT
          FROM orders
          WHERE O_ORDERDATE >= DATE '1994-03-01'
          GROUP BY O_ORDERPRIORITY
          ORDER BY O_ORDERPRIORITY;
+-----------------+-------------+
| O_ORDERPRIORITY | ORDER_COUNT |
+-----------------+-------------+
| 1-URGENT        |     2017573 |
| 2-HIGH          |     2015859 |
| 3-MEDIUM        |     2013174 |
| 4-NOT SPECIFIED |     2014476 |
| 5-LOW           |     2013674 |
+-----------------+-------------+
5 rows in set (0.04 sec)
```

9. To compare HeatWave query execution time with MySQL DB System execution time, disable `use_secondary_engine` and run the query again to see how long it takes to run on the MySQL DB System

```
mysql> SET SESSION use_secondary_engine=OFF;

mysql> SELECT O_ORDERPRIORITY, COUNT(*) AS ORDER_COUNT
          FROM orders
          WHERE O_ORDERDATE >= DATE '1994-03-01'
          GROUP BY O_ORDERPRIORITY
          ORDER BY O_ORDERPRIORITY;
+-----------------+-------------+
| O_ORDERPRIORITY | ORDER_COUNT |
+-----------------+-------------+
| 1-URGENT        |     2017573 |
| 2-HIGH          |     2015859 |
| 3-MEDIUM        |     2013174 |
| 4-NOT SPECIFIED |     2014476 |
| 5-LOW           |     2013674 |
+-----------------+-------------+
5 rows in set (8.91 sec)
```

2.7 Workload Optimization for OLAP

Workload optimization for online analytical processing, OLAP, includes using dictionary encoding for certain string columns and defining data placement keys to optimize for `JOIN` and `GROUP BY` query performance. Apply these optimizations manually, or use HeatWave Autopilot Advisor, which includes the following:

- Auto Encoding recommends string column encodings that minimize the required cluster size and improve query performance.

- Auto Data Placement recommends data placement keys that optimize `JOIN` and `GROUP BY` query performance.
- Auto Query Time Estimation provides runtime information for successfully executed queries and runtime estimates for `EXPLAIN` queries, queries cancelled with `Ctrl+C`, and queries that fail due to out of memory errors. Runtime data is useful for query optimization, troubleshooting, and estimating the cost of running a particular query or workload.
- Unload Advisor

  Recommends tables to unload, that will reduce HeatWave memory usage. The recommendations are based upon when the tables were last queried.

Advisor is workload-aware and provides recommendations based on machine learning models, data analysis, and HeatWave query history. Advisor analyzes the last 1000 successfully executed HeatWave queries.

Advisor is implemented as a stored procedure named `heatwave_advisor`, which resides in the MySQL `sys` schema.

To run Advisor for OLAP workloads, the HeatWave Cluster must be active, and the user must have the following MySQL privileges:

- The `PROCESS` privilege.
- The `EXECUTE` privilege on the `sys` schema.
- The `SELECT` privilege on the Performance Schema.

2.7.1 Encoding String Columns

Encoding string columns helps accelerate the processing of queries that access those columns. HeatWave supports two string column encoding types:

- Variable-length encoding (`VARLEN`)
- Dictionary encoding (`SORTED`)

When tables are loaded into HeatWave, variable-length encoding is applied to `CHAR`, `VARCHAR`, and `TEXT` type columns by default. To use dictionary encoding, you must define the `RAPID_COLUMN=ENCODING=SORTED` keyword string in a column comment before loading the table. The keyword string must be uppercase; otherwise, it is ignored.

You can define the keyword string in a `CREATE TABLE` or `ALTER TABLE` statement, as shown:

```
mysql> CREATE TABLE orders (name VARCHAR(100)
         COMMENT 'RAPID_COLUMN=ENCODING=SORTED');
```

```
mysql> ALTER TABLE orders MODIFY name VARCHAR(100)
         COMMENT 'RAPID_COLUMN=ENCODING=SORTED';
```

If necessary, you can specify variable-length encoding explicitly using the `RAPID_COLUMN=ENCODING=VARLEN` keyword string.

> **Note**
>
> Other information is permitted in column comments. For example, it is permitted for a column description to be specified alongside a column encoding keyword string:

```
COMMENT 'column_description RAPID_COLUMN=ENCODING=SORTED'
```

Tip

For string column encoding recommendations, use HeatWave Autopilot Advisor after loading tables into HeatWave and running queries. For more information, see Section 2.7, “Workload Optimization for OLAP”.

To modify or remove a string column encoding, refer to the procedure described in Section 2.4, “Modifying Tables”.

Encoding Type Selection

If you intend to run `JOIN` operations involving string columns or use string functions and operators, variable-length encoding is recommended. Variable-length encoding provides more expression, filter, function, and operator support than dictionary encoding. Otherwise, select the encoding type based on the number of distinct values in the string column relative to the cardinality of the table.

- Variable-length encoding (`VARLEN`) is best suited to columns with a high number of distinct values, such as “comment” columns.
- Dictionary encoding (`SORTED`) is best suited to columns with a low number of distinct values, such as “country” columns.

Variable-length encoding requires space for column values on the HeatWave nodes. Dictionary encoding requires space on the MySQL DB System node for dictionaries.

The following table provides an overview of encoding type characteristics:

Table 2.1 Column Encoding Type Characteristics

Encoding Type	Expression, Filter, Function, and Operator Support	Best Suited To	Space Required On
`Variable-length (VARLEN)`	Supports `JOIN` operations, string functions and operators, and `LIKE` predicates. See Section 2.14.1, “Variable-length Encoding”.	Columns with a high number of distinct values	HeatWave nodes
`Dictionary (SORTED)`	Does not support `JOIN` operations, string functions and operators, or `LIKE` predicates.	Columns with a low number of distinct values	MySQL DB System node

For additional information about string column encoding, see Section 2.14, “String Column Encoding Reference”.

2.7.2 Defining Data Placement Keys

When data is loaded into HeatWave, it is partitioned by the table primary key and sliced horizontally for distribution among HeatWave nodes by default. The data placement key feature permits partitioning data

by `JOIN` or `GROUP BY` key columns instead, which can improve `JOIN` or `GROUP BY` query performance by avoiding costs associated with redistributing data among HeatWave nodes at query execution time.

Generally, use data placement keys only if partitioning by the primary key does not provide adequate performance. Also, reserve data placement keys for the most time-consuming queries. In such cases, define data placement keys on the most frequently used `JOIN` keys and the keys of the longest running queries.

> **Tip**
>
> For data placement key recommendations, use HeatWave Autopilot Advisor after loading tables into HeatWave and running queries. For more information, see Section 2.7, “Workload Optimization for OLAP”.

Defining a data placement key requires adding a column comment with the data placement keyword string:

```
$> RAPID_COLUMN=DATA_PLACEMENT_KEY=N
```

where *`N`* is an index value that defines the priority order of data placement keys.

- The index must start with 1.
- Permitted index values range from 1 to 16, inclusive.
- An index value cannot be repeated in the same table. For example, you cannot assign an index value of 2 to more than one column in the same table.
- Gaps in index values are not permitted. For example, if you define a data placement key column with an index value of 3, there must also be two other data placement key columns with index values of 1 and 2, respectively.

You can define the data placement keyword string in a `CREATE TABLE` or `ALTER TABLE` statement:

```
mysql> CREATE TABLE orders (date DATE
          COMMENT 'RAPID_COLUMN=DATA_PLACEMENT_KEY=1');
```

```
mysql> ALTER TABLE orders MODIFY date DATE
          COMMENT 'RAPID_COLUMN=DATA_PLACEMENT_KEY=1';
```

The following example shows multiple columns defined as data placement keys. Although a primary key is defined, data is partitioned by the data placement keys, which are prioritized over the primary key.

```
mysql> CREATE TABLE orders (
          id INT PRIMARY KEY,
          date DATE COMMENT 'RAPID_COLUMN=DATA_PLACEMENT_KEY=1',
          price FLOAT COMMENT 'RAPID_COLUMN=DATA_PLACEMENT_KEY=2');
```

When defining multiple columns as data placement keys, prioritize the keys according to query cost. For example, assign `DATA_PLACEMENT_KEY=1` to the key of the costliest query, and `DATA_PLACEMENT_KEY=2` to the key of the next costliest query, and so on.

> **Note**
>
> Other information is permitted in column comments. For example, it is permitted to specify a column description alongside a data placement keyword string:
>
> ```
> mysql> COMMENT 'column_description RAPID_COLUMN=DATA_PLACEMENT_KEY=1'
> ```
>
> To modify or remove a data placement key, refer to the procedure described in Section 2.4, “Modifying Tables”.

Usage notes:

- `JOIN` and `GROUP BY` query optimizations are only applied if at least one of the `JOIN` or `GROUP BY` relations has a key that matches the defined data placement key.
- If a `JOIN` operation can be executed with or without the `JOIN` and `GROUP BY` query optimization, a compilation-time cost model determines how the query is executed. The cost model uses estimated statistics.
- A data placement key cannot be defined on a dictionary-encoded string column but are permitted on variable-length encoded columns. HeatWave applies variable-length encoding to string columns by default. See Section 2.7.1, “Encoding String Columns”.
- A data placement key can only be defined on a column with a supported data type. See Section 2.10, “Supported Data Types”.
- A data placement key column cannot be defined as a `NOT SECONDARY` column. See Section 2.2.2.1, “Excluding Table Columns”.
- For related metadata queries, see Section 2.16, “Metadata Queries”.

2.7.3 HeatWave Autopilot Advisor Syntax

```
mysql> CALL sys.heatwave_advisor ([options]);

 options: {
     JSON_OBJECT('key','value'[,'key','value'] ...)
        'key','value':
        ['output',{'normal'|'silent'|'help'}]
        ['target_schema',JSON_ARRAY({'schema_name'[,'schema_name']}]
        ['exclude_query',JSON_ARRAY('query_id'[,'query_id'] ...)]
        ['query_session_id',JSON_ARRAY('query_session_id'[,'query_session_id'] ...)]
        ['query_insights',{true|false}]
        ['auto_enc',JSON_OBJECT(auto_enc_option)]
        ['auto_dp',JSON_OBJECT(auto_dp_option)]
        ['auto_unload',JSON_OBJECT(auto_unload_option)]
}

    auto_enc_option: {
                    ['mode',{'off'|'recommend'}]
                    ['fixed_enc',JSON_OBJECT('schema.tbl.col',{'varlen'|'dictionary'}
                    [,'schema.tbl.col',{'varlen'|'dictionary'}] ...]
}

     auto_dp_option: {
                    ['benefit_threshold',N]
                    ['max_combinations',N]
}

 auto_unload_option: {
                    ['mode',{'off'|'recommend'}]
                    ['exclude_list',JSON_ARRAY(schema_name_1, schema_name_2.table_name_1, ...)]
                    ['last_queried_hours',N]
                    ['memory_gain_ascending',{true|false}]
                    ['limit_tables',N]
}
```

Advisor *`options`* are specified as key-value pairs in `JSON` format. Options include:

- `output`: Defines how Advisor produces output. Permitted values are:
  - `normal`: The default. Produces summarized output and sends it to `stdout` and to the `heatwave_autopilot_report` table. See Section 2.7.9, “Autopilot Report Table”.

    - `silent`: Sends output to the `heatwave_autopilot_report` table only. See Section 2.7.9, “Autopilot Report Table”. The `silent` output type is useful if human-readable output is not required; when the output is consumed by a script, for example.
    - `help`: Displays Advisor command-line help. See Section 2.7.8, “Advisor Command-line Help”.
- `target_schema`: Defines one or more schemas for Advisor to analyze. The list is specified as a `JSON` array. If a target schema is not specified, Advisor analyzes all schemas in the HeatWave Cluster. When a target schema is specified, Advisor generates recommendations for tables belonging to the target schema. For the most accurate recommendations, specify one schema at a time. Only run Advisor on multiple schemas if the queries access tables in multiple schemas.
- `exclude_query`: Defines the IDs of queries to exclude when Advisor analyzes query statistics. To identify query IDs, query the `performance_schema.rpd_query_stats` table. For a query example, see Section 2.7.5.2, “Auto Data Placement Examples”.
- `query_session_id`: Defines session IDs for filtering queries by session ID. To identify session IDs, query the `performance_schema.rpd_query_stats` table. For a query example, see Section 2.7.6.3, “Auto Query Time Estimation Examples”.
- `query_insights`: Provides runtime information for successfully executed queries and runtime estimates for `EXPLAIN` queries, queries cancelled using `Ctrl+C`, and queries that fail due to an out-of-memory error. See: Section 2.7.6, “Auto Query Time Estimation”. The default setting is `false`.

`auto_enc`: Defines settings for Auto Encoding, see Section 2.7.4, “Auto Encoding”. Options include:

- `mode`: Defines the operational mode. Permitted values are:
    - `off`: The default. Disables Auto Encoding.
    - `recommend`: Enables Auto Encoding.
- `fixed_enc`: Defines an encoding type for specified columns. Use this option if you know the encoding you want for a specific column and you are not interested in an encoding recommendation for that column. Only applicable in `recommend` mode. Columns with a fixed encoding type are excluded from encoding recommendations. The `fixed_enc` key is a fully qualified column name without backticks in the following format: *`schema_name.tbl_name.col_name`*. The value is the encoding type; either `varlen` or `dictionary`. Multiple key-value pairs can be specified in a comma-separated list.

`auto_dp`: Defines settings for Data Placement, which recommends data placement keys. See: Section 2.7.5, “Auto Data Placement”. Options include:

- `benefit_threshold`: The minimum query performance improvement expressed as a percentage value. Advisor only suggests data placement keys estimated to meet or exceed the `benefit_threshold`. The default value is 0.01 (1%). Query performance is a combined measure of all analyzed queries.
- `max_combinations`: The maximum number of data placement key combinations Advisor considers before making recommendations. The default is 10000. The supported range is 1 to 100000. Specifying fewer combinations generates recommendations more quickly but recommendations may not be optimal.

`auto_unload`: Defines settings for Unload Advisor, which recommends tables to unload. See: Section 2.7.7, “Unload Advisor”. Options include:

- `mode`: Defines the operational mode. Permitted values are:
    - `off`: The default. Disables Unload Advisor.

  - `recommend`: Enables Unload Advisor.

- `exclude_list`: Defines a list of schemas and tables to exclude from Unload Advisor. Names must be fully qualified without backticks.
- `last_queried_hours`: Recommend unloading tables that were not queried in the past `last_queried_hours` hours. Minimum: `1`, maximum `744`, default: `24`.
- `memory_gain_ascending`: Whether to rank the unload table suggestions in ascending or descending order based on the table size. Default: `false`.
- `limit_tables`: A limit to the number of unload table suggestions, based on the order imposed by `memory_gain_ascending`. The default is `10`.

2.7.4 Auto Encoding

Auto Encoding recommends string column encodings. Using the right string column encodings can reduce the amount of memory required on HeatWave nodes and improve query performance. HeatWave supports two string column encoding types: variable-length and dictionary. HeatWave applies variable-length encoding to string columns by default when data is loaded. Auto Encoding generates string column encoding recommendations by analyzing column data, HeatWave query history, query performance data, and available memory on the MySQL node. For more information about string column encoding, see Section 2.7.1, “Encoding String Columns”.

2.7.4.1 Running Auto Encoding

To enable Auto Encoding, specify the `auto_enc` option in `recommend` mode. See Section 2.7.3, “HeatWave Autopilot Advisor Syntax”.

> **Note**
>
> To run Advisor for both encoding and data placement recommendations, run Auto Encoding first, apply the recommended encodings, rerun the queries, and then run Auto Data Placement. This sequence allows data placement performance benefits to be calculated with string column encodings in place, which provides for greater accuracy from Advisor internal models.

For Advisor to provide string column encoding recommendations, tables must be loaded in HeatWave and a query history must be available. Run the queries that you intend to use or run a representative set of queries. Failing to do so can affect query offload after Auto Encoding recommendations are implemented due to query constraints associated with dictionary encoding. For dictionary encoding limitations, see Section 2.14.2, “Dictionary Encoding”.

In the following example, Auto Encoding is run in `recommend` mode, which analyzes column data, checks the amount of memory on the MySQL node, and provides encoding recommendations intended to reduce the amount of space required on HeatWave nodes and optimize query performance. There is no target schema specified, so Auto Encoding runs on all schemas loaded in HeatWave

```
mysql> CALL sys.heatwave_advisor(JSON_OBJECT('auto_enc',JSON_OBJECT('mode','recommend')));
```

The `fixed_enc` option can be used in `recommend` mode to specify an encoding for specific columns. These columns are excluded from consideration when Auto Encoding generates recommendations. Manually encoded columns are also excluded from consideration. (For manual encoding instructions, see Section 2.7.1, “Encoding String Columns”.)

```
mysql> CALL sys.heatwave_advisor(JSON_OBJECT('auto_enc',JSON_OBJECT('mode','recommend','fixed_enc',
```

```
          JSON_OBJECT('tpch.CUSTOMER.C_ADDRESS','varlen'))));
```

Advisor output provides information about each stage of Advisor execution, including recommended column encodings and estimated HeatWave Cluster memory savings.

```
mysql> CALL sys.heatwave_advisor(JSON_OBJECT('target_schema',JSON_ARRAY('tpch_1024'),
          'auto_enc',JSON_OBJECT('mode','recommend')));
+------------------------------+
| INITIALIZING HEATWAVE ADVISOR |
+------------------------------+
| Version: 1.12                |
|                              |
| Output Mode: normal          |
| Excluded Queries: 0          |
| Target Schemas: 1            |
|                              |
+------------------------------+
6 rows in set (0.01 sec)

+-----------------------------------------------------------+
| ANALYZING LOADED DATA                                     |
+-----------------------------------------------------------+
| Total 8 tables loaded in HeatWave for 1 schemas           |
| Tables excluded by user: 0 (within target schemas)        |
|                                                           |
| SCHEMA                              TABLES        COLUMNS |
| NAME                                LOADED         LOADED |
| ------                              ------         ------ |
| `tpch_1024`                              8             61 |
|                                                           |
+-----------------------------------------------------------+
8 rows in set (0.15 sec)

+-----------------------------------------------------------------------------------------------+
| ENCODING SUGGESTIONS                                                                          |
+-----------------------------------------------------------------------------------------------+
| Total Auto Encoding suggestions produced for 21 columns                                       |
| Queries executed: 200                                                                         |
|   Total query execution time: 28.82 min                                                       |
|   Most recent query executed on: Tuesday 8th June 2021 14:42:13                               |
|   Oldest query executed on: Tuesday 8th June 2021 14:11:45                                    |
|                                                                                               |
|                                                        CURRENT           SUGGESTED          |
| COLUMN                                                  COLUMN              COLUMN          |
| NAME                                                  ENCODING            ENCODING          |
| ------                                                --------            ---------         |
| `tpch_1024`.`CUSTOMER`.`C_ADDRESS`                      VARLEN          DICTIONARY          |
| `tpch_1024`.`CUSTOMER`.`C_COMMENT`                      VARLEN          DICTIONARY          |
| `tpch_1024`.`CUSTOMER`.`C_MKTSEGMENT`                   VARLEN          DICTIONARY          |
| `tpch_1024`.`CUSTOMER`.`C_NAME`                         VARLEN          DICTIONARY          |
| `tpch_1024`.`LINEITEM`.`L_COMMENT`                      VARLEN          DICTIONARY          |
| `tpch_1024`.`LINEITEM`.`L_SHIPINSTRUCT`                 VARLEN          DICTIONARY          |
| `tpch_1024`.`LINEITEM`.`L_SHIPMODE`                     VARLEN          DICTIONARY          |
| `tpch_1024`.`NATION`.`N_COMMENT`                        VARLEN          DICTIONARY          |
| `tpch_1024`.`NATION`.`N_NAME`                           VARLEN          DICTIONARY          |
| `tpch_1024`.`ORDERS`.`O_CLERK`                          VARLEN          DICTIONARY          |
| `tpch_1024`.`ORDERS`.`O_ORDERPRIORITY`                  VARLEN          DICTIONARY          |
| `tpch_1024`.`PART`.`P_BRAND`                            VARLEN          DICTIONARY          |
| `tpch_1024`.`PART`.`P_COMMENT`                          VARLEN          DICTIONARY          |
| `tpch_1024`.`PART`.`P_CONTAINER`                        VARLEN          DICTIONARY          |
| `tpch_1024`.`PART`.`P_MFGR`                             VARLEN          DICTIONARY          |
| `tpch_1024`.`PARTSUPP`.`PS_COMMENT`                     VARLEN          DICTIONARY          |
| `tpch_1024`.`REGION`.`R_COMMENT`                        VARLEN          DICTIONARY          |
| `tpch_1024`.`REGION`.`R_NAME`                           VARLEN          DICTIONARY          |
| `tpch_1024`.`SUPPLIER`.`S_ADDRESS`                      VARLEN          DICTIONARY          |
| `tpch_1024`.`SUPPLIER`.`S_NAME`                         VARLEN          DICTIONARY          |
| `tpch_1024`.`SUPPLIER`.`S_PHONE`                        VARLEN          DICTIONARY          |
```

```
|                                                                                                  |
| Applying the suggested encodings might improve query performance and cluster memory usage.|
|   Estimated HeatWave Cluster memory savings: 355.60 GiB                                          |
|                                                                                                  |
+--------------------------------------------------------------------------------------------------+
35 rows in set (18.18 sec)

+--------------------------------------------------------------------------------------------------+
| SCRIPT GENERATION                                                                                |
+--------------------------------------------------------------------------------------------------+
| Script generated for applying suggestions for 8 loaded tables                                    |
|                                                                                                  |
| Applying changes will take approximately 1.64 h                                                  |
|                                                                                                  |
| Retrieve script containing 61 generated DDL commands using the query below:                      |
|   SELECT log->>"$.sql" AS "SQL Script" FROM sys.heatwave_advisor_report WHERE type = "sql"|
|   ORDER BY id;                                                                                   |
|                                                                                                  |
| Caution: Executing the generated script will alter the column comment and secondary engine|
| flags in the schema                                                                              |
|                                                                                                  |
+--------------------------------------------------------------------------------------------------+
9 rows in set (18.20 sec)
```

To inspect the load script, which includes the DDL statements required to implement the recommended encodings, query the `heatwave_autopilot_report` table.

```
mysql> SELECT log->>"$.sql" AS "SQL Script"
          FROM sys.heatwave_autopilot_report
          WHERE type = "sql"
          ORDER BY id;
```

To concatenate generated DDL statements into a single string that can be copied and pasted for execution, issue the statements that follow. The `group_concat_max_len` variable sets the result length in bytes for the `GROUP_CONCAT()` function to accommodate a potentially long string. (The default `group_concat_max_len` setting is 1024 bytes.)

```
mysql> SET SESSION group_concat_max_len = 1000000;
mysql> SELECT GROUP_CONCAT(log->>"$.sql" SEPARATOR ' ')
          FROM sys.heatwave_advisor_report
          WHERE type = "sql"
          ORDER BY id;
```

Usage Notes:

- Auto Encoding analyzes string columns (`CHAR`, `VARCHAR`, and `TEXT` type columns) of tables that are loaded in HeatWave. Automatically or manually excluded columns, columns greater than 65532 bytes, and columns with manually defined encodings are excluded from consideration. Auto Encoding also analyzes HeatWave query history to identify query constraints that preclude the use of dictionary encoding. Dictionary-encoded columns are not supported in `JOIN` operations, with string functions and operators, or in `LIKE` predicates. For dictionary encoding limitations, see Section 2.14.2, “Dictionary Encoding”.

- The time required to generate encoding recommendations depends on the number of queries to be analyzed, the number of operators, and the complexity of each query.

- Encoding recommendations for the same table may differ after changes to data or data statistics. For example, changes to table cardinality or the number of distinct values in a column can affect recommendations.

- Auto Encoding does not generate recommendations for a given table if existing encodings do not require modification.

- Auto Encoding only recommends dictionary encoding if it is expected to reduce the amount of memory required on HeatWave nodes.
- If there is not enough MySQL node memory for the dictionaries of all columns that would benefit from dictionary encoding, the columns estimated to save the most memory are recommended for dictionary encoding.
- Auto Encoding uses the current state of tables loaded in HeatWave when generating recommendations. Concurrent change propagation activity is not considered.
- Encoding recommendations are based on estimates and are therefore not guaranteed to reduce the memory required on HeatWave nodes or improve query performance.

2.7.4.2 Auto Encoding Examples

- Running Auto Encoding to generate string column encoding recommendations for the `tpch` schema:

```
mysql> CALL sys.heatwave_advisor(JSON_OBJECT('target_schema',JSON_ARRAY('tpch'),
          'auto_enc',JSON_OBJECT('mode','recommend')));
```

- Running Auto Encoding with the `fixed_enc` option to force variable-length encoding for the `tpch.CUSTOMER.C_ADDRESS` column. Columns specified by the `fixed_enc` option are excluded from consideration by the Auto Encoding feature.

```
mysql> CALL sys.heatwave_advisor(JSON_OBJECT('target_schema',JSON_ARRAY('tpch'),
          'auto_enc',JSON_OBJECT('mode','recommend','fixed_enc',
           JSON_OBJECT('tpch.CUSTOMER.C_ADDRESS','varlen'))));
```

2.7.5 Auto Data Placement

Auto Data Placement generates data placement key recommendations. Data placement keys are used to partition table data among HeatWave nodes when loading tables. Partitioning table data by `JOIN` and `GROUP BY` key columns can improve query performance by avoiding costs associated with redistributing data among HeatWave nodes at query execution time. The Data Placement Advisor generates data placement key recommendations by analyzing table statistics and HeatWave query history. For more information about data placement keys, see Section 2.7.2, “Defining Data Placement Keys”.

2.7.5.1 Running Auto Data Placement

> **Note**
>
> To run Advisor for both encoding and data placement recommendations, run Auto Encoding first, apply the recommended encodings, rerun the queries, and then run Auto Data Placement. This sequence allows data placement performance benefits to be calculated with string column encodings in place, which provides for greater accuracy from Advisor internal models.

For Advisor to generate data placement recommendations:

- Tables must be loaded in HeatWave.
- There must be a query history with at least 5 queries. A query is counted if it includes a `JOIN` on tables loaded in the HeatWave Cluster or `GROUP BY` keys. A query executed on a table that is no longer loaded or that was reloaded since the query was run is not counted.

  To view the query history, query the `performance_schema.rpd_query_stats` table. For example:

```
mysql> SELECT query_id, LEFT(query_text,160)
```

```
          FROM performance_schema.rpd_query_stats;
```

For the most accurate data placement recommendations, run Advisor on one schema at a time. In the following example, Advisor is run on the `tpch_1024` schema using the `target_schema` option. No other *`options`* are specified, which means that the default option settings are used.

```
mysql> CALL sys.heatwave_advisor(JSON_OBJECT('target_schema',JSON_ARRAY('tpch_1024')));
```

Advisor output provides information about each stage of Advisor execution. The data placement suggestion output shows suggested data placement keys and the estimated performance benefit of applying the keys.

The script generation output provides a query for retrieving the generated DDL statements for implementing the suggested data placement keys. Data placement keys cannot be added to a table or modified without reloading the table. Therefore, Advisor generates DDL statements for unloading the table, adding the keys, and reloading the table.

```
mysql> CALL sys.heatwave_advisor(JSON_OBJECT('target_schema',JSON_ARRAY('tpch_1024')));
+------------------------------+
| INITIALIZING HEATWAVE ADVISOR |
+------------------------------+
| Version: 1.12                |
|                              |
| Output Mode: normal          |
| Excluded Queries: 0          |
| Target Schemas: 1            |
|                              |
+------------------------------+
6 rows in set (0.01 sec)

+-----------------------------------------------------------+
| ANALYZING LOADED DATA                                     |
+-----------------------------------------------------------+
| Total 8 tables loaded in HeatWave for 1 schemas           |
| Tables excluded by user: 0 (within target schemas)        |
|                                                           |
| SCHEMA                            TABLES        COLUMNS |
| NAME                              LOADED         LOADED |
| ------                            ------         ------ |
| `tpch_1024`                            8             61 |
|                                                           |
+-----------------------------------------------------------+
8 rows in set (0.02 sec)

+-------------------------------------------------------------------------+
| AUTO DATA PLACEMENT                                                     |
+-------------------------------------------------------------------------+
| Auto Data Placement Configuration:                                      |
|   Minimum benefit threshold: 1%                                         |
| Producing Data Placement suggestions for current setup:                 |
|   Tables Loaded: 8                                                      |
|   Queries used: 189                                                     |
|     Total query execution time: 22.75 min                               |
|     Most recent query executed on: Tuesday 8th June 2021 16:29:02       |
|     Oldest query executed on: Tuesday 8th June 2021 16:05:43            |
|   HeatWave cluster size: 5 nodes                                        |
|                                                                         |
| All possible Data Placement combinations based on query history: 120    |
| Explored Data Placement combinations after pruning: 90                  |
|                                                                         |
+-------------------------------------------------------------------------+
16 rows in set (12.38 sec)

+-----------------------------------------------------------------------------------------+
| DATA PLACEMENT SUGGESTIONS                                                              |
```

```
+-------------------------------------------------------------------------------------------+
| Total Data Placement suggestions produced for 2 tables                                    |
|                                                                                           |
| TABLE                                          DATA PLACEMENT           DATA PLACEMENT |
| NAME                                              CURRENT KEY            SUGGESTED KEY |
| ------                                         --------------           -------------- |
| `tpch_1024`.`LINEITEM`              L_ORDERKEY, L_LINENUMBER               L_ORDERKEY |
| `tpch_1024`.`SUPPLIER`                             S_SUPPKEY              S_NATIONKEY |
|                                                                                           |
| Expected benefit after applying Data Placement suggestions                                |
|   Runtime saving: 6.17 min                                                                |
|   Performance benefit: 27%                                                                |
|                                                                                           |
+-------------------------------------------------------------------------------------------+
12 rows in set (16.42 sec)

+-------------------------------------------------------------------------------------------+
| SCRIPT GENERATION                                                                         |
+-------------------------------------------------------------------------------------------+
| Script generated for applying suggestions for 2 loaded tables                             |
|                                                                                           |
| Applying changes will take approximately 1.18 h                                           |
|                                                                                           |
| Retrieve script containing 12 generated DDL commands using the query below:               |
|   SELECT log->>"$.sql" AS "SQL Script" FROM sys.heatwave_advisor_report WHERE type = "sql"|
|   ORDER BY id;                                                                            |
|                                                                                           |
| Caution: Executing the generated script will alter the column comment and secondary engine|
| flags in the schema                                                                       |
|                                                                                           |
+-------------------------------------------------------------------------------------------+
9 rows in set (16.43 sec)


SELECT log->>"$.sql" AS "SQL Script" FROM sys.heatwave_advisor_report WHERE type = "sql"
 ORDER BY id;
+-------------------------------------------------------------------------------------------+
| SQL Script                                                                                |
+-------------------------------------------------------------------------------------------+
| SET SESSION innodb_parallel_read_threads = 48;                                            |
| ALTER TABLE `tpch_1024`.`LINEITEM` SECONDARY_UNLOAD;                                      |
| ALTER TABLE `tpch_1024`.`LINEITEM` SECONDARY_ENGINE=NULL;                                 |
| ALTER TABLE `tpch_1024`.`LINEITEM` MODIFY `L_ORDERKEY` bigint NOT NULL COMMENT            |
|  'RAPID_COLUMN=DATA_PLACEMENT_KEY=1';                                                     |
| ALTER TABLE `tpch_1024`.`LINEITEM` SECONDARY_ENGINE=RAPID;                                |
| ALTER TABLE `tpch_1024`.`LINEITEM` SECONDARY_LOAD;                                        |
| SET SESSION innodb_parallel_read_threads = 48;                                            |
| ALTER TABLE `tpch_1024`.`SUPPLIER` SECONDARY_UNLOAD;                                      |
| ALTER TABLE `tpch_1024`.`SUPPLIER` SECONDARY_ENGINE=NULL;                                 |
| ALTER TABLE `tpch_1024`.`SUPPLIER` MODIFY `S_NATIONKEY` int NOT NULL COMMENT              |
|  'RAPID_COLUMN=DATA_PLACEMENT_KEY=1';                                                     |
| ALTER TABLE `tpch_1024`.`SUPPLIER` SECONDARY_ENGINE=RAPID;                                |
| ALTER TABLE `tpch_1024`.`SUPPLIER` SECONDARY_LOAD;                                        |
+-------------------------------------------------------------------------------------------+
12 rows in set (0.00 sec)
```

Usage Notes:

- If a table already has data placement keys or columns are customized before running Advisor, Advisor may generate DDL statements for removing previously defined data placement keys.
- Advisor provides recommendations only if data placement keys are estimated to improve query performance. If not, an information message is returned and no recommendations are provided.
- Advisor provides data placement key recommendations based on approximate models. Recommendations are therefore not guaranteed to improve query performance.

2.7.5.2 Auto Data Placement Examples

- Invoking Advisor without any *`options`* runs the Data Placement Advisor with the default option settings.

```
mysql> CALL sys.heatwave_advisor(NULL);
```

- Running Advisor with only the `target_schema` option runs the Data Placement Advisor on the specified schemas with the default option settings.

```
mysql> CALL sys.heatwave_advisor(JSON_OBJECT("target_schema",JSON_ARRAY("tpch","employees")));
```

- Run Advisor with the data placement `max_combinations` and `benefit_threshold` parameters. For information about these *`options`*, see Section 2.7.3, “HeatWave Autopilot Advisor Syntax”.

```
mysql> CALL sys.heatwave_advisor(JSON_OBJECT("auto_dp",
         JSON_OBJECT("max_combinations",100,"benefit_threshold",20)));
```

- The following example shows how to view the HeatWave query history by querying the `performance_schema.rpd_query_stats` table, and how to exclude specific queries from Data Placement Advisor analysis using the `exclude_query` option:

```
mysql> SELECT query_id, LEFT(query_text,160)
         FROM performance_schema.rpd_query_stats;

mysql> CALL sys.heatwave_advisor(JSON_OBJECT("target_schema",
         JSON_ARRAY("tpch"),"exclude_query",JSON_ARRAY(1,11,12,14)));
```

- This example demonstrates how to invoke the Data Placement Advisor with *`options`* specified in a variable:

```
mysql> SET @options = JSON_OBJECT(
           "target_schema", JSON_ARRAY("analytics45","sample_schema"),
           "exclude_query", JSON_ARRAY(12,24),
           "auto_dp", JSON_OBJECT(
                  "benefit_threshold", 12.5,
                  "max_combinations", 100
));

mysql> CALL sys.heatwave_advisor( @options );
```

- This example demonstrates how to invoke Advisor in silent output mode, which is useful if the output is consumed by a script, for example. Auto Data Placement is run by default if no option such as `auto_enc` or `query_insights` is specified.

```
mysql> CALL sys.heatwave_advisor(JSON_OBJECT("output","silent"));
```

2.7.6 Auto Query Time Estimation

Auto Query Time Estimation provides:

- Runtimes for successfully executed queries
- Runtime estimates for `EXPLAIN` queries.
- Runtime estimates for queries cancelled using `Ctrl+C`.
- Runtime estimates for queries that fail due to an out-of-memory error.

Runtime data can be used for query optimization, troubleshooting, or to estimate the cost of running a particular query or workload on HeatWave.

2.7.6.1 Run Auto Query Time Estimation

For Auto Query Time Estimation to provide runtime data, a query history must be available. Auto Query Time Estimation can provide runtime data for up to 1000 queries, which is the HeatWave query history limit. To view the current HeatWave query history, query the `performance_schema.rpd_query_stats` table:

```
mysql> SELECT query_id, LEFT(query_text,160)
         FROM performance_schema.rpd_query_stats;
```

The following example shows how to retrieve runtime data for the entire query history using Auto Query Time Estimation. In this example, there are three queries in the query history: a successfully executed query, a query that failed due to an out of memory error, and a query that was cancelled using `Ctrl+C`. For an explanation of Auto Query Time Estimation data, see Section 2.7.6.2, “Auto Query Time Estimation Data”.

```
mysql> CALL sys.heatwave_advisor(JSON_OBJECT('query_insights', true));
+------------------------------+
| INITIALIZING HEATWAVE ADVISOR |
+------------------------------+
| Version: 1.12                |
|                              |
| Output Mode: normal          |
| Excluded Queries: 0          |
| Target Schemas: All          |
|                              |
+------------------------------+
6 rows in set (0.01 sec)

+-----------------------------------------------------------+
| ANALYZING LOADED DATA                                     |
+-----------------------------------------------------------+
| Total 8 tables loaded in HeatWave for 1 schemas           |
| Tables excluded by user: 0 (within target schemas)        |
|                                                           |
| SCHEMA                            TABLES        COLUMNS |
| NAME                              LOADED         LOADED |
| ------                            ------         ------ |
| `tpch128`                              8             61 |
|                                                           |
+-----------------------------------------------------------+
8 rows in set (0.02 sec)

+-------------------------------------------------------------------------------------------+
| QUERY INSIGHTS                                                                            |
+-------------------------------------------------------------------------------------------+
| Queries executed on Heatwave: 4                                                           |
| Session IDs (as filter): None                                                             |
|                                                                                           |
| QUERY-ID  SESSION-ID  QUERY-STRING             EXEC-RUNTIME  COMMENT                      |
| --------  ----------  ------------             ------------  -------                      |
|        1          32  SELECT COUNT(*)                                                     |
|                       FROM tpch128.LINEITEM    0.628                                      |
|        2          32  SELECT COUNT(*)                                                     |
|                       FROM tpch128.ORDERS      0.114 (est.)  Explain.                     |
|        3          32  SELECT COUNT(*)                                                     |
|                       FROM tpch128.ORDERS,                                                |
|                       tpch128.LINEITEM         5.207 (est.)  Out of memory               |
|                                                              error during                 |
|                                                              query execution              |
|                                                              in RAPID.                    |
|        4          32  SELECT COUNT(*)                                                     |
|                       FROM tpch128.SUPPLIER,                                              |
|                       tpch128.LINEITEM         3.478 (est.)  Operation was               |
|                                                              interrupted by               |
```

```
|                                                                                |
| TOTAL ESTIMATED:   3   EXEC-RUNTIME:       8.798 sec                           |
| TOTAL EXECUTED:    1   EXEC-RUNTIME:       0.628 sec                           |
|                                                                                |
|                                                                                |
| Retrieve detailed query statistics using the query below:                      |
|     SELECT log FROM sys.heatwave_advisor_report WHERE stage = "QUERY_INSIGHTS" AND  |
|     type = "info";                                                             |
|                                                                                |
+--------------------------------------------------------------------------------+

mysql> SELECT log FROM sys.heatwave_advisor_report
          WHERE stage = "QUERY_INSIGHTS"
          AND type = "info";
+--------------------------------------------------------------------------------+
| log                                                                            |
+--------------------------------------------------------------------------------+
| {"comment": "", "query_id": 1, "query_text": "SELECT COUNT(*) FROM tpch128.LINEITEM",|
|  "session_id": 32, "runtime_executed_ms": 627.6099681854248,                   |
|  "runtime_estimated_ms": 454.398817}                                           |
|                                                                                |
| {"comment": "Explain.", "query_id": 2, "query_text": "SELECT COUNT(*)          |
|   FROM tpch128.ORDERS", "session_id": 32, "runtime_executed_ms": null,         |
|  "runtime_estimated_ms": 113.592768}                                           |
|                                                                                |
| {"comment": "Out of memory error during query execution in RAPID.", "query_id": 3,   |
|  "query_text": "SELECT COUNT(*) FROM tpch128.ORDERS, tpch128.LINEITEM",        |
|  "session_id": 32, "runtime_executed_ms": null, "runtime_estimated_ms": 5206.80822}  |
|                                                                                |
| {"comment": "Operation was interrupted by the user.", "query_id": 4,           |
|  "query_text": "SELECT COUNT(*) FROM tpch128.SUPPLIER, tpch128.LINEITEM",      |
|  "session_id": 32, "runtime_executed_ms": null, "runtime_estimated_ms": 3477.720953} |
+--------------------------------------------------------------------------------+
4 rows in set (0.00 sec)
```

2.7.6.2 Auto Query Time Estimation Data

Auto Query Time Estimation provides the following data:

- `QUERY-ID`

  The query ID.

- `SESSION-ID`

  The session ID that issued the query.

- `QUERY-STRING`

  The query string. `EXPLAIN`, if specified, is not displayed in the query string.

- `EXEC-RUNTIME`

  The query execution runtime in seconds. Runtime estimates are differentiated from actual runtimes by the appearance of the following text adjacent to the runtime: `(est.)`. Actual runtimes are shown for successfully executed queries. Runtime estimates are shown for `EXPLAIN` queries, queries cancelled by `Ctrl+C`, and queries that fail with an out-of-memory error.

- `COMMENT`

  Comments associated with the query. Comments may include:

  - `Explain`: The query was run with `EXPLAIN`.

- `Operation was interrupted by the user`: The query was successfully offloaded to HeatWave but was interrupted by a Ctrl+C key combination.
- `Out of memory error during query execution in RAPID`: The query was successfully offloaded to HeatWave but failed due to an out-of-memory error.

- `TOTAL-ESTIMATED` and `EXEC-RUNTIME`

  The total number of queries with runtime estimates and total execution runtime (estimated).

- `TOTAL-EXECUTED` and `EXEC-RUNTIME`

  The total number of successfully executed queries and total execution runtime (actual).

- `Retrieve detailed statistics using the query below`

  The query retrieves detailed statistics from the `heatwave_autopilot_report` table. For an example of the detailed statistics, see Section 2.7.6.1, “Run Auto Query Time Estimation”.

Auto Query Time Estimation data is available in machine readable format for use in scripts. Auto Query Time Estimation data is also available in `JSON` format or SQL table format with queries to the `heatwave_autopilot_report` table. See Section 2.7.9, “Autopilot Report Table”.

2.7.6.3 Auto Query Time Estimation Examples

- To view runtime data for all queries in the HeatWave query history for a particular schema:

```
mysql> CALL sys.heatwave_advisor(JSON_OBJECT('target_schema',JSON_ARRAY('tpch'),
          'query_insights',true));
```

- To view runtime data for queries issued by the current session:

```
mysql> CALL sys.heatwave_advisor(JSON_OBJECT('query_insights',true,
          'query_session_id', JSON_ARRAY(connection_id())));
```

- To view runtime data for queries issued by a particular session:

```
mysql> CALL sys.heatwave_advisor(JSON_OBJECT('query_insights', true,
          'query_session_id', JSON_ARRAY(8)));
```

- This example demonstrates how to invoke the Auto Query Time Estimation in silent output mode, which is useful if the output is consumed by a script, for example.

```
mysql> CALL sys.heatwave_advisor(JSON_OBJECT('query_insights',true,'output','silent'));
```

2.7.7 Unload Advisor

Unload Advisor recommends tables to unload, that will reduce HeatWave memory usage. The recommendations are based upon when the tables were last queried.

2.7.7.1 Running Unload Advisor

For Advisor to recommend tables to unload, the tables must be loaded in HeatWave and a query history must be available.

To enable Unload Advisor, specify the `auto_unload` option in `recommend` mode. See Section 2.7.3, “HeatWave Autopilot Advisor Syntax”.

Use the `exclude_list` option to define a list of schemas and tables to exclude from Unload Advisor.

Use the `last_queried_hours` option to only recommend unloading tables that were not queried during this past number of hours. The default is `24` hours.

Set `memory_gain_ascending` to `true` to rank the unload table suggestions in ascending order based on the table size. The default is `false`.

Use the `limit_tables` option to limit the number of unload table suggestions, based on the order imposed by `memory_gain_ascending`. The default is `10`.

2.7.7.2 Unload Advisor Examples

- To view recommendations for tables to unload, for a particular schema:

```
mysql> CALL sys.heatwave_advisor(JSON_OBJECT('auto_unload',JSON_OBJECT('mode','recommend')));
+-------------------------------------------------------------------------------------------------------------------------------
| HEATWAVE UNLOAD ADVISOR
+-------------------------------------------------------------------------------------------------------------------------------
| This feature is in Preview Mode.
| Using:
|  "last_queried_hours": 24
|  "memory_gain_ascending": false
|  "limit_tables": 10
| Number of excluded tables: 0
|
| SCHEMA                   TABLE
| NAME                     NAME                                                  REASON
| ------                   -----------      ---------------------------------------------     --------------
| `tpch128`                `LINEITEM`         LAST QUERIED ON '2022-09-12 15:30:40.538585'
| `tpch128`                `CUSTOMER`         LAST QUERIED ON '2022-09-12 15:30:40.538585'
|
|
| Storage consumed by base relations after unload: 100 GiB
+-------------------------------------------------------------------------------------------------------------------------------
```

2.7.8 Advisor Command-line Help

The command-line help provides usage documentation for the Advisor. To view Advisor command-line help:

```
mysql> CALL sys.heatwave_advisor(JSON_OBJECT('output','help'));
```

2.7.9 Autopilot Report Table

MySQL 8.0.32 deprecates the `heatwave_advisor_report` and `heatwave_load_report` tables, and replaces them with the `heatwave_autopilot_report` table in the `sys` schema. A future release will remove them.

When Advisor or Auto Parallel Load run, they send detailed output to the `heatwave_autopilot_report` table in the `sys` schema. This includes execution logs and generated load scripts.

The `heatwave_autopilot_report` table is a temporary table. It contains data from the last execution of Advisor or Auto Parallel Load. Data is only available for the current session and is lost when the session terminates or when the server shuts down.

Autopilot Report Table Query Examples

Query the `heatwave_autopilot_report` table after calls to Advisor or Auto Parallel Load, as in the following examples:

- View warning information:

```
mysql> SELECT log FROM sys.heatwave_autopilot_report WHERE type="warn";
```

- View error information if Advisor or Auto Parallel Load stop unexpectedly:

```
mysql> SELECT log FROM sys.heatwave_autopilot_report WHERE type="error";
```

- View the generated DDL statements for Advisor recommendations, or to see the commands that would be executed by Auto Parallel Load in `normal` mode:

```
mysql> SELECT log->>"$.sql" AS "SQL Script"
          FROM sys.heatwave_autopilot_report
          WHERE type = "sql"
          ORDER BY id;
```

- Concatenate Advisor or Auto Parallel Load generated DDL statements into a single string to copy and paste for execution. The `group_concat_max_len` variable sets the result length in bytes for the `GROUP_CONCAT()` function to accommodate a potentially long string. The default `group_concat_max_len` setting is 1024 bytes.

```
mysql> SET SESSION group_concat_max_len = 1000000;
mysql> SELECT GROUP_CONCAT(log->>"$.sql" SEPARATOR ' ')
          FROM sys.heatwave_autopilot_report
          WHERE type = "sql"
          ORDER BY id;
```

- For Advisor, retrieve Auto Query Time Estimation data in `JSON` format:

```
mysql> SELECT log
          FROM sys.heatwave_autopilot_report
          WHERE stage = "QUERY_INSIGHTS" AND type = "info";
```

- For Advisor, retrieve Auto Query Time Estimation data in SQL table format:

```
mysql> SELECT log->>"$.query_id" AS query_id,
          log->>"$.session_id" AS session_id,
          log->>"$.query_text" AS query_text,
          log->>"$.runtime_estimated_ms" AS runtime_estimated_ms,
          log->>"$.runtime_executed_ms" AS runtime_executed_ms,
          log->>"$.comment" AS comment
          FROM sys.heatwave_autopilot_report
          WHERE stage = "QUERY_INSIGHTS" AND type = "info"
          ORDER BY id;
```

- For Auto Parallel Load, view the number of load commands generated:

```
mysql> SELECT Count(*) AS "Total Load Commands Generated"
          FROM sys.heatwave_autopilot_report
          WHERE type = "sql" ORDER BY id;
```

- For Auto Parallel Load, view load script data for a particular table:

```
mysql> SELECT log->>"$.sql"
          FROM sys.heatwave_autopilot_report
          WHERE type="sql" AND log->>"$.schema_name" = "db0" AND log->>"$.table_name" = "tbl"
          ORDER BY id;
```

2.7.10 Advisor Report Table

MySQL 8.0.32 deprecates the `heatwave_advisor_report` table, and replaces it with the `heatwave_autopilot_report` table in the `sys` schema. A future release will remove it. See Section 2.7.9, “Autopilot Report Table”.

When MySQL runs Advisor, it sends detailed output to the `heatwave_advisor_report` table in the `sys` schema.

The `heatwave_advisor_report` table is a temporary table. It contains data from the last execution of Advisor. Data is only available for the current session and is lost when the session terminates or when the server is shut down.

Advisor Report Table Query Examples

Query the `heatwave_advisor_report` table after MySQL runs Advisor, as in the following examples:

- View Advisor warning information:

```
mysql> SELECT log FROM sys.heatwave_advisor_report WHERE type="warn";
```

- View error information if Advisor stops unexpectedly:

```
mysql> SELECT log FROM sys.heatwave_advisor_report WHERE type="error";
```

- View the generated DDL statements for Advisor recommendations:

```
mysql> SELECT log->>"$.sql" AS "SQL Script"
          FROM sys.heatwave_advisor_report
          WHERE type = "sql"
          ORDER BY id;
```

- Concatenate Advisor generated DDL statements into a single string to copy and paste for execution. The `group_concat_max_len` variable sets the result length in bytes for the `GROUP_CONCAT()` function to accommodate a potentially long string. The default `group_concat_max_len` setting is 1024 bytes.

```
mysql> SET SESSION group_concat_max_len = 1000000;
mysql> SELECT GROUP_CONCAT(log->>"$.sql" SEPARATOR ' ')
          FROM sys.heatwave_advisor_report
          WHERE type = "sql"
          ORDER BY id;
```

- Retrieve Auto Query Time Estimation data in `JSON` format:

```
mysql> SELECT log
          FROM sys.heatwave_advisor_report
          WHERE stage = "QUERY_INSIGHTS" AND type = "info";
```

- Retrieve Auto Query Time Estimation data in SQL table format:

```
mysql> SELECT log->>"$.query_id" AS query_id,
          log->>"$.session_id" AS session_id,
          log->>"$.query_text" AS query_text,
          log->>"$.runtime_estimated_ms" AS runtime_estimated_ms,
          log->>"$.runtime_executed_ms" AS runtime_executed_ms,
          log->>"$.comment" AS comment
          FROM sys.heatwave_advisor_report
          WHERE stage = "QUERY_INSIGHTS" AND type = "info"
          ORDER BY id;
```

2.8 Workload Optimization for OLTP

Online transaction processing, OLTP, does not require a HeatWave Cluster, nor the secondary engine. OLTP only requires the MySQL InnoDB primary engine.

2.8.1 Autopilot Indexing

As of MySQL 9.0.0, HeatWave Autopilot Advisor includes Autopilot Indexing that can make secondary index suggestions to improve workload performance.

Autopilot Indexing obtains workloads from the statement digest history stored in the Performance Schema, evaluates them, and identifies potential indexes that can help improve workload performance.

Indexes can improve the query performance of `SELECT` statements, but index management can increase the overhead of DML operations, `INSERT`, `UPDATE`, and `DELETE`. Autopilot Indexing can generate secondary index recommendations that optimize the overall workload performance.

Autopilot Indexing generates recommendations to add and drop indexes and also includes:

- An estimate of the overall workload performance benefit,
- An estimate of the overall storage footprint impact.
- A DDL statement, an explanation, and an estimated storage impact for each index recommendation.
- An estimate of the index creation time and the estimated performance gain of the top 5 queries of each create index recommendation.

To run Autopilot Indexing for OLTP workloads, the user must have the following MySQL privileges:

- The `PROCESS` privilege.
- The `EXECUTE` privilege on the `sys` schema.
- The `SELECT` privilege on the Performance Schema.

2.8.1.1 Autopilot Indexing Syntax

Autopilot Indexing uses the following syntax:

```
mysql> CALL sys.autopilot_index_advisor ([options]);

 options: {
     JSON_OBJECT('key','value'[,'key','value'] ...)
        'key','value':
        ['output',{'normal'|'silent'|'help'}]
        ['target_schema',JSON_ARRAY({'schema_name'['schema_name']}]
}
```

Autopilot Indexing *`options`* are specified as key-value pairs in `JSON` format. Options include:

- `output`: Defines how Autopilot Indexing produces output. Permitted values are:
  - `normal`: The default. Produces summarized output and sends it to `stdout` and to the `autopilot_index_advisor_report` table. See Section 2.8.1.5, “Autopilot Index Advisor Report Table”.
  - `silent`: Sends output to the `autopilot_index_advisor_report` table only. See Section 2.8.1.5, “Autopilot Index Advisor Report Table”. The `silent` output type is useful if human-readable output is not required; when the output is consumed by a script, for example.
  - `help`: Displays Autopilot Indexing command-line help. See Section 2.8.1.4, “Autopilot Indexing Command-line Help”.
- `target_schema`: Defines one or more schemas for Advisor to analyze. The list is specified as a `JSON` array. If a target schema is not specified, Advisor analyzes all user defined schemas. When a target schema is specified, Advisor generates recommendations for tables belonging to the target schema. For the most accurate recommendations, specify one schema at a time. Only run Advisor on multiple schemas if the queries access tables in multiple schemas.

2.8.1.2 Running Autopilot Indexing

Autopilot Indexing provides recommendations for schemas with a representative workload that has at least five queries in the SQL statement digest history in the Performance Schema.

Autopilot Indexing only evaluates SQL statements in the SQL statement digest history that access existing tables. Autopilot Indexing does evaluate past SQL statements that access a table that has been dropped and recreated.

When the workload changes, invoke Autopilot Indexing again to update index recommendations.

Autopilot Indexing recommends indexes to create and drop. Rather than drop the suggested indexes, make them invisible first, and then drop them later after confirmation that there is no impact to any user queries. For instance, if a query uses an index hint, and that index is dropped, then the query will fail to execute. This includes `FORCE INDEX`, `USE INDEX`, and `IGNORE INDEX`, see Index Hints.

Autopilot Indexing does not evaluate the following:

- System schemas.
- Small InnoDB tables, with `innodb_page_size` less than 16KB, or less than 1,000 rows.
- Statements executed from stored routines and prepared statements.
- Columns with the following data types:
  - `BINARY`
  - `VARBINARY`
  - `TINYBLOB`
  - `BLOB`
  - `MEDIUMBLOB`
  - `LONGBLOB`
  - `JSON`
  - `TEXT`
  - `TINYTEXT`
  - `MEDIUMTEXT`
  - `LONGTEXT`

Autopilot Indexing does not make recommendations for the following:

- The creation of primary keys.
- Indexes that enforce foreign key constraints.
- Functional indexes.
- Indexes on partitioned tables.
- Indexes that speed up `LIKE` predicates.

2.8.1.3 Autopilot Indexing Examples

To evaluate all user defined schemas:

```
mysql> CALL sys.autopilot_index_advisor(NULL);
```

To evaluate selected schemas:

```
mysql> CALL  sys.autopilot_index_advisor(JSON_OBJECT('target_schema',JSON_ARRAY('schema1','schema2', ...))
```

An example of the output that includes the following information:

- New indexes to create.
- Existing indexes to drop.
- The reason, expected storage footprint, and performance impact of each suggestion.
- The overall expected performance benefit and expected change to the storage footprint.
- An estimate of the time to create an index.

```
+-----------------------------------------------------------------------------------------------------------------
| INDEX SUGGESTIONS
+-----------------------------------------------------------------------------------------------------------------
| Total Index suggestions: 5
| Statements analyzed:     519798
|
| SUGGEST           TABLE                                  INDEXED                                  REASO
| ACTION            NAME                                   COLUMNS                                  SUGGE
| -------           -----                                  -------                                  ----
| CREATE            `world`.`city`                         `Population`                             Missi
| CREATE            `world`.`state`                        `Area`, `Population`                     Missi
| CREATE            `world`.`country`                      `Area`                                   Missi
| CREATE            `world`.`continent`                    `ContinentCode`                          Missi
| DROP              `world`.`countrylanguage`              `CountryCode`                            Unuse
|
| Expected performance benefit after applying all Index suggestions: 97.3%
| Expected storage footprint after applying all Index suggestions:  + 236.98 MiB
|        64.00 KiB freed up by dropping indexes.
|       237.05 MiB required for creating indexes.
|       NOTE: Indexes will be stored efficiently at time of creation.
|             To accommodate efficient future inserts, size may double.
| Expected time for applying all Index creation suggestions:         25 s
|
+-----------------------------------------------------------------------------------------------------------------
```

To clear the statement digest in the Performance Schema and start collecting statements for a new workload:

```
mysql> CALL sys.ps_truncate_all_tables(FALSE);
```

See Section 2.8.1.5, “Autopilot Index Advisor Report Table”.

2.8.1.4 Autopilot Indexing Command-line Help

The command-line help provides usage documentation for Autopilot Indexing. To view Autopilot Indexing command-line help:

```
mysql> CALL sys.autopilot_index_advisor(JSON_OBJECT('output','help'));
```

2.8.1.5 Autopilot Index Advisor Report Table

When Autopilot Indexing runs, it sends detailed output to the `autopilot_index_advisor_report` table in the `sys` schema. This includes execution logs and generated load scripts.

The `autopilot_index_advisor_report` table is a temporary table that contains data from the last execution of Autopilot Indexing. Data is only available for the current session and is lost when the session terminates or when the server shuts down.

Index Advisor Report Table Query Examples

Query the `autopilot_index_advisor_report` table after calls to Autopilot Indexing, as in the following examples:

- To view the DDL statements for the index suggestions:

```
mysql> SELECT log->>"$.sql" AS "SQL Script"
        FROM sys.autopilot_index_advisor_report
        WHERE type = "sql"
        ORDER BY id;
+-------------------------------------------------------------------------------------------+
| SQL Script                                                                                |
+-------------------------------------------------------------------------------------------+
| CREATE INDEX `autoidx_tab1108_col2` ON `world`.`city` ( `Population` );                   |
| CREATE INDEX `autoidx_tab1110_col3_col2` ON `world`.`state` ( `Area`, `Population` );     |
| CREATE INDEX `autoidx_tab1104_col1` ON `world`.`country` ( `Area` );                      |
| CREATE INDEX `autoidx_tab1109_col5` ON `world`.`continent` ( `ContinentCode` );           |
| DROP INDEX `countrylanguage_idx4` ON `world`.`countrylanguage` ( `CountryCode` );         |
+-------------------------------------------------------------------------------------------+
5 rows in set (0.00 sec)
```

  Then execute the script to implement the suggestions.

- To view explanations for Autopilot Indexing recommendations:

```
mysql> SELECT JSON_PRETTY(log) AS "Explanations"
        FROM sys.autopilot_index_advisor_report
        WHERE type = "explain"
        ORDER BY id;
+-----------------------------------------------------------------------------------------------------------+
| Explanations                                                                                              |
+-----------------------------------------------------------------------------------------------------------+
| {
  "SQL": "CREATE INDEX `autoidx_tab1108_col2` ON `world`.`city` ( `Population` );",
  "explanation": [
    {
      "reason": "Covering Index",
      "query_text": "SELECT `Name` FROM `city` WHERE `Population` = ?",
      "estimated_gain": "700.0x"
    }
  ],
  "est_create_time": "5.79 s"
} |
+-----------------------------------------------------------------------------------------------------------+
1 row in set (0.01 sec)
```

  The explanation includes the top 5 queries that benefit from the index, together with the reason and estimated gain of each query. It also includes the estimated time to create the index.

2.9 Best Practices

2.9.1 Preparing Data

The following practices are recommended when preparing data for loading into HeatWave:

- Instead of preparing and loading tables into HeatWave manually, consider using the Auto Parallel Load utility. See Section 2.2.3, “Loading Data Using Auto Parallel Load”.

- To minimize the number of HeatWave nodes required for your data, exclude table columns that are not accessed by your queries. For information about excluding columns, see Section 2.2.2.1, “Excluding Table Columns”.

- To save space in memory, set `CHAR`, `VARCHAR`, and `TEXT` type column lengths to the minimum length required for the longest string value.

- Where appropriate, apply dictionary encoding to `CHAR`, `VARCHAR`, and `TEXT` type columns. Dictionary encoding reduces memory consumption on the HeatWave Cluster nodes. Use the following criteria when selecting string columns for dictionary encoding:

  1. The column is not used as a key in `JOIN` queries.

  2. Your queries do not perform operations such as `LIKE`, `SUBSTR`, `CONCAT`, etc., on the column. Variable-length encoding supports string functions and operators and `LIKE` predicates; dictionary encoding does not.

  3. The column has a limited number of distinct values. Dictionary encoding is best suited to columns with a limited number of distinct values, such as “country” columns.

  4. The column is expected to have few new values added during change propagation. Avoid dictionary encoding for columns with a high number of inserts and updates. Adding a significant number of a new, unique values to a dictionary encoded column can cause a change propagation failure.

  The following columns from the TPC Benchmark™ H (TPC-H) provide examples of string columns that are suitable and unsuitable for dictionary encoding:

  - `ORDERS.O_ORDERPRIORITY`

    This column is used only in range queries. The values associated with column are limited. During updates, it is unlikely for a significant number of new, unique values to be added. These characteristics make the column suitable for dictionary encoding.

  - `LINEITEM.L_COMMENT`

    This column is not used in joins or other complex expressions, but as a comment field, values are expected to be unique, making the column unsuitable for dictionary encoding.

  When in doubt about choosing an encoding type, use variable-length encoding, which is applied by default when tables are loaded into HeatWave, or use Auto Encoding to obtain encoding recommendations. See Section 2.7.4, “Auto Encoding”.

- Data is partitioned by the table primary key when no data placement keys are defined. Only consider defining data placement keys if partitioning data by the primary key does not provide suitable performance.

  Reserve the use of data placement keys for the most time-consuming queries. In such cases, define data placement keys on:

  - The most frequently used `JOIN` keys.

  - The keys of the longest running queries.

  Consider using Auto Data Placement for data placement recommendations. See Section 2.7.5, “Auto Data Placement”.

2.9.2 Provisioning

To determine the appropriate HeatWave Cluster size for a workload, you can estimate the required cluster size. Cluster size estimates are generated by the HeatWave Auto Provisioning feature, which uses machine learning models to predict the number of required nodes based on node shape and data sampling. For instructions:

- For HeatWave on OCI, see Generating a Node Count Estimate in the *HeatWave on OCI Service Guide*.
- For HeatWave on AWS, see Estimating Cluster Size with HeatWave Autopilot in the *HeatWave on AWS Service Guide*.
- For HeatWave for Azure, see Provisioning HeatWave Nodes in the *HeatWave for Azure Service Guide*.

Perform a cluster size estimate:

- When adding a HeatWave Cluster to a MySQL DB System, to determine the number of nodes required for the data you intend to load.
- Periodically, to ensure that you have an appropriate number of HeatWave nodes for your data. Over time, data size may increase or decrease, so it is important to monitor the size of your data by performing cluster size estimates.
- When encountering out-of-memory errors while running queries. In this case, the HeatWave Cluster may not have sufficient memory capacity.
- When the data growth rate is high.
- When the transaction rate (the rate of updates and inserts) is high.

2.9.3 Importing Data into the MySQL DB System

MySQL Shell is the recommended utility for importing data into the MySQL DB System. MySQL Shell dump and load utilities are purpose-built for use with MySQL DB Systems; useful for all types of exports and imports. MySQL Shell supports export to, and import from, Object Storage. The minimum supported source version of MySQL is 5.7.9.

- For HeatWave on OCI, see Importing and Exporting Databases in the *HeatWave on OCI Service Guide*.
- For HeatWave on AWS, see Importing Data in the *HeatWave on AWS Service Guide*.
- For HeatWave for Azure, see Importing Data to HeatWave in the *HeatWave for Azure Service Guide*.

2.9.4 Inbound Replication

Replicating DDL operations: Before a table is loaded into HeatWave, `RAPID` must be defined as the table secondary engine; for example:

```
mysql> ALTER TABLE orders SECONDARY_ENGINE = RAPID;
```

2.9.5 Loading Data

Instead of preparing and loading tables into HeatWave manually, consider using the Auto Parallel Load utility. See Section 2.2.3, “Loading Data Using Auto Parallel Load”. Users of HeatWave on AWS also have the option of loading data from the HeatWave Console. See Manage Data in HeatWave with Workspaces in the *HeatWave on AWS Service Guide*.

The loading of data into HeatWave can be classified into three types: *Initial Bulk Load*, *Incremental Bulk Load*, and *Change Propagation*.

- *Initial Bulk Load*: Performed when loading data into HeatWave for the first time, or when reloading data. The best time to perform an initial bulk load is during off-peak hours, as bulk load operations can affect OLTP performance on the MySQL DB System.
- *Incremental Bulk Load*: Performed when there is a substantial amount of data to load into tables that are already loaded in HeatWave. An incremental bulk load involves these steps:

  1. Performing a `SECONDARY_UNLOAD` operation to unload a table from HeatWave. See Section 2.5.1, “Unloading Tables”.
  2. Importing data into the table on the MySQL DB System node.
  3. Performing a `SECONDARY_LOAD` operation to reload the table into HeatWave. See Section 2.2, “Loading Data to HeatWave MySQL”.

  Depending on the amount of data, an incremental bulk load may be a faster method of loading new data than waiting for change propagation to occur. It also provides greater control over when new data is loaded. As with initial build loads, the best time to perform an incremental bulk load is during off-peak hours, as bulk load operations can affect OLTP performance on the MySQL DB System.
- *Change Propagation*: After tables are loaded into HeatWave, data changes are automatically propagated from `InnoDB` tables on the MySQL DB System to their counterpart tables in HeatWave. See Section 2.2.7, “Change Propagation”.

Use the following strategies to improve load performance:

- *Increase the number of read threads*

  For medium to large tables, increase the number of read threads to 32 by setting the `innodb_parallel_read_threads` variable on the MySQL DB System.

  ```
  mysql> SET SESSION innodb_parallel_read_threads = 32;
  ```

  If the MySQL DB System is not busy, you can increase the value to 64.

  > **Tip**
  >
  > The Auto Parallel Load utility automatically optimizes the number of parallel read threads for each table. See Section 2.2.3, “Loading Data Using Auto Parallel Load”. For users of HeatWave on AWS, the number of parallel read threads is also optimized when loading data from the HeatWave Console. See Manage Data in HeatWave with Workspaces in the *HeatWave on AWS Service Guide*.

- *Load tables concurrently*

  If you have many small and medium tables (less than 20GB in size), load tables from multiple sessions:

  ```
  Session 1:
  mysql> ALTER TABLE supplier SECONDARY_LOAD;

  Session 2:
  mysql> ALTER TABLE parts SECONDARY_LOAD;

  Session 3:
  mysql> ALTER TABLE region SECONDARY_LOAD;
  ```

```
Session 4:
mysql> ALTER TABLE partsupp SECONDARY_LOAD;
```

- *Avoid or reduce conflicting operations*

  Data load operations share resources with other OLTP DML and DDL operations on the MySQL DB System. To improve load performance, avoid or reduce conflicting DDL and DML operations. For example, avoid running DDL and large DML operations on the `LINEITEM` table while executing an `ALTER TABLE LINEITEM SECONDARY_LOAD` operation.

2.9.6 Auto Encoding and Auto Data Placement

Advisor analyzes the data and HeatWave query history to provide string column encoding and data placement key recommendations. Run Advisor again for updated recommendations when queries change, when data changes significantly, and after reloading modified tables.

In all cases, run the queries again before running Advisor. See Section 2.7, “Workload Optimization for OLAP”.

2.9.7 Running Queries

The following practices are recommended when running queries:

- If a query fails to offload and you cannot identify the reason, enable tracing and query the `INFORMATION_SCHEMA.OPTIMIZER_TRACE` table to debug the query. See Section 2.3.6, “Debugging Queries”.

  If the optimizer trace does not return all of the trace information, increase the optimizer trace buffer size. The `MISSING_BYTES_BEYOND_MAX_MEM_SIZE` column of the `INFORMATION_SCHEMA.OPTIMIZER_TRACE` table shows how many bytes are missing from a trace. If the column shows a non-zero value, increase the `optimizer_trace_max_mem_size` setting accordingly. For example:

```
SET optimizer_trace_max_mem_size=1000000;
```

- If an `INFORMATION_SCHEMA.OPTIMIZER_TRACE` query trace indicates that a subquery is not yet supported, try unnesting the subquery. For example, the following query contains a subquery and is not offloaded as indicated by the `EXPLAIN` output, which does not show “Using secondary engine”.

```
mysql> EXPLAIN SELECT COUNT(*)
          FROM orders o
          WHERE o_totalprice> (SELECT AVG(o_totalprice)
          FROM orders
          WHERE o_custkey=o.o_custkey);
*************************** 1. row ***************************
           id: 1
  select_type: PRIMARY
        table: o
   partitions: NULL
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 14862970
     filtered: 100.00
        Extra: Using where
*************************** 2. row ***************************
           id: 2
```

```
  select_type: DEPENDENT SUBQUERY
        table: orders
   partitions: NULL
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 14862970
     filtered: 10.00
        Extra: Using where
2 rows in set, 2 warnings (0.00 sec)
```

This query can be rewritten as follows to unnest the subquery so that it can be offloaded.

```
mysql> EXPLAIN SELECT COUNT(*)
          FROM orders o, (SELECT o_custkey, AVG(o_totalprice) a_totalprice
          FROM orders
          GROUP BY o_custkey)a
          WHERE o.o_custkey=a.o_custkey AND o.o_totalprice>a.a_totalprice;
```

- By default, `SELECT` queries are offloaded to HeatWave for execution and fall back to the MySQL DB System if that is not possible. To force a query to execute on HeatWave or fail if that is not possible, set the `use_secondary_engine variable` to `FORCED`. In this mode, a `SELECT` statement returns an error if it cannot be offloaded. The `use_secondary_engine` variable can be set as shown:

  - Using a `SET` statement before running queries:

    ```
    mysql> SET SESSION use_secondary_engine = FORCED;
    ```

  - Using a `SET_VAR` optimizer hint when issuing a query:

    ```
    mysql> SELECT /*+ SET_VAR(use_secondary_engine = FORCED) */ ... FROM ...
    ```

- If you encounter out-of-memory errors when running queries:

  1. Avoid or rewrite queries that produce a Cartesian product. In the following query, a `JOIN` predicated is not defined between the `supplier` and `nation` tables, which causes the query to select all rows from both tables:

     ```
     mysql> SELECT s_nationkey, s_suppkey, l_comment FROM lineitem, supplier, nation
               WHERE s_suppkey = l_suppkey LIMIT 10;
     ERROR 3015 (HY000): Out of memory in storage engine 'Failure detected in RAPID; query
     execution cannot proceed'.
     ```

     To avoid the Cartesian product, add a relevant predicate between the `supplier` and `nation` tables to filter out rows:

     ```
     mysql> SELECT s_nationkey, s_suppkey, l_comment
               FROM lineitem, supplier, nation
               WHERE s_nationkey = n_nationkey AND s_suppkey = l_suppkey LIMIT 10;
     ```

  2. Avoid or rewrite queries that produce a Cartesian product introduced by the MySQL optimizer. Due to lack of quality statistics or non-optimal cost decisions, MySQL optimizer may introduce one or more Cartesian products in a query even if a query has predicates defined among all participating tables. For example:

     ```
     mysql> SELECT o_orderkey, c_custkey, l_shipdate, s_nationkey, s_suppkey, l_comment
               FROM lineitem, supplier, nation, customer, orders
               WHERE c_custkey = o_custkey AND o_orderkey = l_orderkey
               AND c_nationkey = s_nationkey AND c_nationkey = n_nationkey AND c_custkey < 3000000
               LIMIT 10;
     ERROR 3015 (HY000): Out of memory in storage engine 'Failure detected in RAPID;
     ```

```
query execution cannot proceed'.
```

The EXPLAIN plan output shows that there is no common predicate between the first two table entries (NATION and SUPPLIER).

```
mysql> EXPLAIN SELECT o_orderkey, c_custkey, l_shipdate, s_nationkey, s_suppkey, l_comment
          FROM lineitem, supplier, nation, customer, orders
          WHERE c_custkey = o_custkey AND o_orderkey = l_orderkey AND c_nationkey = s_nationkey
          AND c_nationkey = n_nationkey AND c_custkey < 3000000 LIMIT 10;
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: supplier
   partitions: NULL
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 99626
     filtered: 100.00
        Extra: Using secondary engine RAPID
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: nation
   partitions: NULL
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 25
     filtered: 10.00
        Extra: Using where; Using join buffer (hash join); Using secondary engine RAPID
*************************** 3. row ***************************
           id: 1
  select_type: SIMPLE
        table: customer
   partitions: NULL
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 1382274
     filtered: 5.00
        Extra: Using where; Using join buffer (hash join); Using secondary engine RAPID
*************************** 4. row ***************************
           id: 1
  select_type: SIMPLE
        table: orders
   partitions: NULL
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 14862970
     filtered: 10.00
        Extra: Using where; Using join buffer (hash join); Using secondary engine RAPID
*************************** 5. row ***************************
           id: 1
  select_type: SIMPLE
        table: lineitem
   partitions: NULL
```

```
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 56834662
     filtered: 10.00
        Extra: Using where; Using join buffer (hash join); Using secondary engine RAPID
```

To force a join order so that there are predicates associated with each pair of tables, add a `STRAIGHT_JOIN` hint. For example:

```
mysql> EXPLAIN SELECT o_orderkey, c_custkey, l_shipdate, s_nationkey, s_suppkey, l_comment
          FROM SUPPLIER STRAIGHT_JOIN CUSTOMER STRAIGHT_JOIN NATION STRAIGHT_JOIN ORDERS
          STRAIGHT_JOIN LINEITEM WHERE c_custkey = o_custkey and o_orderkey = l_orderkey
          AND c_nationkey = s_nationkey AND c_nationkey = n_nationkey AND c_custkey < 3000000
          LIMIT 10;
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: supplier
   partitions: NULL
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 99626
     filtered: 100.00
        Extra: Using secondary engine RAPID
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: customer
   partitions: NULL
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 1382274
     filtered: 5.00
        Extra: Using where; Using join buffer (hash join); Using secondary engine RAPID
*************************** 3. row ***************************
           id: 1
  select_type: SIMPLE
        table: nation
   partitions: NULL
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 25
     filtered: 10.00
        Extra: Using where; Using join buffer (hash join); Using secondary engine RAPID
*************************** 4. row ***************************
           id: 1
  select_type: SIMPLE
        table: orders
   partitions: NULL
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
```

```
         rows: 14862970
     filtered: 10.00
        Extra: Using where; Using join buffer (hash join); Using secondary engine RAPID
*************************** 5. row ***************************
           id: 1
  select_type: SIMPLE
        table: lineitem
   partitions: NULL
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 56834662
     filtered: 10.00
        Extra: Using where; Using join buffer (hash join); Using secondary engine RAPID
```

3. Avoid or rewrite queries that produce a very large result set. This is a common cause of out of memory errors during query processing. Use aggregation functions, a `GROUP BY` clause, or a `LIMIT` clause to reduce the result set size.

4. Avoid or rewrite queries that produce a very large intermediate result set. In certain cases, large result sets can be avoided by adding a `STRAIGHT_JOIN` hint, which enforces a join order in a decreasing of selectiveness.

5. Check the size of your data by performing a cluster size estimate. If your data has grown substantially, the HeatWave Cluster may require additional nodes.

   - For HeatWave on OCI, see Generating a Node Count Estimate in the *HeatWave on OCI Service Guide*.
   - For HeatWave on AWS, see Estimating Cluster Size with HeatWave Autopilot in the *HeatWave on AWS Service Guide*.
   - For HeatWave for Azure, see Provisioning HeatWave Nodes in the *HeatWave for Azure Service Guide*.

6. HeatWave optimizes for network usage by default. Try running the query with the `MIN_MEM_CONSUMPTION` strategy by setting by setting `rapid_execution_strategy` to `MIN_MEM_CONSUMPTION`. The `rapid_execution_strategy` variable can be set as shown:

   - Using a `SET` statement before running queries:

     ```
     mysql> SET SESSION rapid_execution_strategy = MIN_MEM_CONSUMPTION;
     ```

   - Using a `SET_VAR` optimizer hint when issuing a query:

     ```
     mysql> SELECT /*+ SET_VAR(rapid_execution_strategy = MIN_MEM_CONSUMPTION) */ ... FROM ...
     ```

   - Unloading tables that are not used. These tables consume memory on HeatWave nodes unnecessarily. See Section 2.5.1, “Unloading Tables”.
   - Excluding table columns that are not accessed by your queries. These columns consume memory on HeatWave nodes unnecessarily. This strategy requires reloading data. See Section 2.2.2.1, “Excluding Table Columns”.

7. After running queries, consider using HeatWave Autopilot Advisor for encoding and data placement recommendations. See Section 2.7, “Workload Optimization for OLAP”.

2.9.8 Monitoring

The following monitoring practices are recommended:

- For HeatWave on OCI, monitor operating system memory usage by setting an alarm to notify you when memory usage on HeatWave nodes remains above 450GB for an extended period of time. If memory usage exceeds this threshold, either reduce the size of your data or add nodes to the HeatWave Cluster. For information about using metrics, alarms, and notifications, refer to Metrics in the *HeatWave on OCI Service Guide*.
- For HeatWave on AWS, you can monitor memory usage on the **Performance** tab of the HeatWave Console. See Performance Monitoring in the *HeatWave on AWS Service Guide*.
- For HeatWave for Azure, select **Metrics** on the details page for the HeatWave Cluster to access Microsoft Azure Application Insights. See About Oracle Database Service for Azure.
- Monitor change propagation status. If change propagation is interrupted and tables are not automatically reloaded for some reason, table data becomes stale. Queries that access tables with stale data are not offloaded to HeatWave for processing. For instructions, see Section 2.2.7, “Change Propagation”.

2.9.9 Reloading Data

Reloading data is recommended in the following cases:

- After resizing the cluster by adding or removing nodes. Reloading data distributes the data among all nodes of the resized cluster.
- After a maintenance window. Maintenance involves a DB System restart, which requires that you reload data into HeatWave. On OCI, consider setting up a HeatWave Service event notification or Service Connector Hub notification to let you know when an update has occurred. For information about MySQL DB System maintenance:
  - For HeatWave on OCI, see Maintenance in the *HeatWave on OCI Service Guide*.

    For HeatWave on AWS, see Maintenance in the *HeatWave on AWS Service Guide*.

    For HeatWave for Azure, see the Oracle Database Service for Azure documentation.
  - For information about HeatWave Service events, see Managing a DB System in the *HeatWave on OCI Service Guide*.
  - For information about Service Connector Hub, see Service Connector Hub.
  - For table load instructions, see Section 2.2, “Loading Data to HeatWave MySQL”.

    > **Tip**
    >
    > Instead of loading data into HeatWave manually, consider using the Auto Parallel Load utility, which prepares and loads data for you using an optimized number of parallel load threads. See Section 2.2.3, “Loading Data Using Auto Parallel Load”.
- When the HeatWave Cluster is restarted due to a DB System restart. Data in the HeatWave Cluster is lost in this case, requiring reload.

2.10 Supported Data Types

HeatWave supports the following data types. Columns with unsupported data types must be excluded, and defined as `NOT SECONDARY` before loading a table. See Section 2.2.2.1, “Excluding Table Columns”.

- Numeric data types:
  - `BIGINT`
  - `BOOL`
  - `DECIMAL`
  - `DOUBLE`
  - `FLOAT`
  - `INT`
  - `INTEGER`
  - `MEDIUMINT`
  - `SMALLINT`
  - `TINYINT`
- Temporal data types:
  - `DATE`
  - `DATETIME`
  - `TIME`
  - `TIMESTAMP`
  - `YEAR`

  Temporal types are only supported with strict SQL mode. See Strict SQL Mode.

  For limitations related to `TIMESTAMP` data type support, see Section 2.18.2, “Data Type Limitations”.
- String and text data types:
  - `CHAR`
  - `VARCHAR`
  - `TEXT`
  - `TINYTEXT`
  - `MEDIUMTEXT`
  - `LONGTEXT`
  - `VECTOR`

For string and text data type limitations, see Section 2.18.2, “Data Type Limitations”.

- `ENUM`

  For `ENUM` limitations, see Section 2.18.2, “Data Type Limitations”.

- JSON

  As of MySQL 9.0.0, the `JSON` data type supports the following functions and operators:

  - Cast-functions and operators, see: Section 2.12.3, “Cast Functions and Operators”.
  - `COALESCE()` and `IN()`, see: Section 2.12.4, “Comparison Functions and Operators”.
  - Flow control functions and operators, see: Section 2.12.5, “Control Flow Functions and Operators”.
  - Mathematical functions, see: Section 2.12.10, “Mathematical Functions”.
  - String functions and operators, see: Section 2.12.11, “String Functions and Operators”.

2.11 Supported SQL Modes

Default MySQL DB System SQL modes are supported, which include `ONLY_FULL_GROUP_BY`, `STRICT_TRANS_TABLES`, `NO_ZERO_IN_DATE`, `NO_ZERO_DATE`, `ERROR_FOR_DIVISION_BY_ZERO`, and `NO_ENGINE_SUBSTITUTION`. See Server SQL Modes.

In addition, the following SQL modes are supported:

- `ANSI_QUOTES`
- `HIGH_NOT_PRECEDENCE`
- `IGNORE_SPACE`
- `NO_BACKSLASH_ESCAPES`
- `REAL_AS_FLOAT`
- `TIME_TRUNCATE_FRACTIONAL`

Temporal types are supported only with strict SQL mode. See Strict SQL Mode.

2.12 Supported Functions and Operators

This section describes functions and operators supported by HeatWave MySQL.

2.12.1 Aggregate Functions

The following table shows supported aggregate functions. The *VARLEN Support* column identifies functions that support variable-length encoded string columns. See Section 2.7.1, “Encoding String Columns”.

Table 2.2 Aggregate (GROUP BY) Functions

Name	VARLEN Support	Description
`AVG()`		Return the average value of the argument

Name	VARLEN Support	Description
COUNT()	Yes	Return a count of the number of rows returned
COUNT(DISTINCT)		Return the count of a number of different values
GROUP_CONCAT()	Yes	Return a concatenated string. See Section 2.18.3, “Functions and Operator Limitations”.
HLL()	Yes	Only available in HeatWave MySQL. Similar to COUNT(DISTINCT), but faster and with user defined precision.
JSON_ARRAYAGG()		Return result set as a single JSON array. Supported as of MySQL 9.0.0.
JSON_OBJECTAGG()		Return result set as a single JSON object literal. Supported as of MySQL 9.0.0.
MAX()	Yes	Return the maximum value
MIN()	Yes	Return the minimum value
STD()		Return the population standard deviation
STDDEV()		Return the population standard deviation
STDDEV_POP()		Return the population standard deviation
STDDEV_SAMP()		Return the sample standard deviation
SUM()		Return the sum
VAR_POP()		Return the population standard variance
VAR_SAMP()		Return the sample variance
VARIANCE()		Return the population standard variance

- `HLL(expr,[expr...], precision)`

  Returns an approximate count of the number of rows with different non-`NULL` *`expr`* values. `4` ≤ *`precision`* ≤ `15`. The default value is `10`.

  If there are no matching rows, `HLL()` returns `0`.

  As of MySQL 8.4.0, HyperLogLog, `HLL()`, is available in the HeatWave primary and secondary engines. Before MySQL 8.4.0, HyperLogLog, `HLL()`, is only available in the HeatWave secondary engine. It is similar to `COUNT(DISTINCT)`, but with user defined precision.

  `HLL()` is faster than `COUNT(DISTINCT)`. The greater the *`precision`*, the greater the accuracy, and the greater the amount of memory required.

  For each `GROUP`, `HLL()` requires $2^{\text{precision}}$ bytes of memory. The default value of `10` requires `1KiB` of memory per `GROUP`.

  An example with precision set to `8` that uses 256 bytes of memory per `GROUP`:

```
mysql> SELECT HLL(results, 8) FROM student;
```

2.12.1.1 GROUP BY Modifiers

Only available in the HeatWave secondary engine, the `GROUP BY` clause permits the following:

- A `CUBE` modifier in addition to the `WITH ROLLUP` modifier.
- A `ROLLUP` modifier as a preferred alternative to the `WITH ROLLUP` modifier.

See: GROUP BY Modifiers

The `ROLLUP` modifier generates aggregated results that follow the hierarchy for the selected columns. The `CUBE` modifier generates aggregated results for all possible combinations of the selected columns. For a single column the results are the same.

A `ROLLUP` modifier example:

```
mysql> SELECT
          IF(GROUPING(year), 'All years', year) AS year,
          IF(GROUPING(country), 'All countries', country) AS country,
          IF(GROUPING(product), 'All products', product) AS product,
        SUM(profit) AS profit
        FROM sales
        GROUP BY ROLLUP (year, country, product);
+-----------+---------------+--------------+--------+
| year      | country       | product      | profit |
+-----------+---------------+--------------+--------+
| 2000      | Finland       | Computer     |   1500 |
| 2000      | Finland       | Phone        |    100 |
| 2000      | Finland       | All products |   1600 |
| 2000      | India         | Calculator   |    150 |
| 2000      | India         | Computer     |   1200 |
| 2000      | India         | All products |   1350 |
| 2000      | USA           | Calculator   |     75 |
| 2000      | USA           | Computer     |   1500 |
| 2000      | USA           | All products |   1575 |
| 2000      | All countries | All products |   4525 |
| 2001      | Finland       | Phone        |     10 |
| 2001      | Finland       | All products |     10 |
| 2001      | USA           | Calculator   |     50 |
| 2001      | USA           | Computer     |   2700 |
```

```
| 2001      | USA           | TV           |    250 |
| 2001      | USA           | All products |   3000 |
| 2001      | All countries | All products |   3010 |
| All years | All countries | All products |   7535 |
+-----------+---------------+--------------+--------+
```

A CUBE modifier example that uses the same data:

```
mysql> SELECT
         IF(GROUPING(year), 'All years', year) AS year,
         IF(GROUPING(country), 'All countries', country) AS country,
         IF(GROUPING(product), 'All products', product) AS product,
       SUM(profit) AS profit
       FROM sales
       GROUP BY CUBE (year, country, product);
+-----------+---------------+--------------+--------+
| year      | country       | product      | profit |
+-----------+---------------+--------------+--------+
| 2001      | USA           | Computer     |   2700 |
| 2000      | USA           | Computer     |   1500 |
| 2000      | India         | Calculator   |    150 |
| 2001      | USA           | TV           |    250 |
| 2000      | USA           | Calculator   |     75 |
| 2000      | Finland       | Phone        |    100 |
| 2001      | Finland       | Phone        |     10 |
| 2000      | Finland       | Computer     |   1500 |
| 2001      | USA           | Calculator   |     50 |
| 2000      | India         | Computer     |   1200 |
| All years | All countries | All products |   7535 |
| 2001      | All countries | All products |   3010 |
| 2000      | All countries | All products |   4525 |
| All years | India         | All products |   1350 |
| All years | Finland       | All products |   1610 |
| All years | USA           | All products |   4575 |
| 2001      | USA           | All products |   3000 |
| 2000      | India         | All products |   1350 |
| 2000      | Finland       | All products |   1600 |
| 2000      | USA           | All products |   1575 |
| 2001      | Finland       | All products |     10 |
| All years | All countries | TV           |    250 |
| All years | All countries | Computer     |   6900 |
| All years | All countries | Phone        |    110 |
| All years | All countries | Calculator   |    275 |
| 2001      | All countries | Computer     |   2700 |
| 2000      | All countries | Phone        |    100 |
| 2000      | All countries | Calculator   |    225 |
| 2001      | All countries | Phone        |     10 |
| 2001      | All countries | TV           |    250 |
| 2001      | All countries | Calculator   |     50 |
| 2000      | All countries | Computer     |   4200 |
| All years | Finland       | Computer     |   1500 |
| All years | USA           | Calculator   |    125 |
| All years | USA           | TV           |    250 |
| All years | USA           | Computer     |   4200 |
| All years | India         | Calculator   |    150 |
| All years | Finland       | Phone        |    110 |
| All years | India         | Computer     |   1200 |
+-----------+---------------+--------------+--------+
```

2.12.2 Arithmetic Operators

The following table shows supported arithmetic operators. All arithmetic operators are supported with variable-length encoded string columns, see Section 2.7.1, “Encoding String Columns”.

Table 2.3 Arithmetic Operators

Name	Description
`DIV`	Integer division
`/`	Division operator
`-`	Minus operator
`%`, `MOD`	Modulo operator
`+`	Addition operator
`*`	Multiplication operator
`-`	Change the sign of the argument

2.12.3 Cast Functions and Operators

The following operations are supported with the `CAST()` function.

- `CAST()` from and to all the HeatWave supported numeric, temporal, string and text data types. See Section 2.10, “Supported Data Types”.
- `CAST()` of `ENUM` columns to `CHAR`, `DECIMAL`, `FLOAT`, and to `SIGNED` and `UNSIGNED` numeric values. `CAST()` operates on the `ENUM` index rather than the `ENUM` values.

See Section 2.18.3, “Functions and Operator Limitations”.

2.12.4 Comparison Functions and Operators

The following table shows supported comparison functions and operators. The *VARLEN Support* column identifies functions and operators that support variable-length encoded string columns. See Section 2.7.1, “Encoding String Columns”.

Table 2.4 Comparison Functions and Operators

Name	VARLEN Support	Description
`BETWEEN ... AND ...`	Yes	Check whether a value is within a range of values
`COALESCE()`	Yes	Return the first non-NULL argument. Not supported as a `JOIN` predicate.
`=`	Yes	Equal operator
`<=>`		NULL-safe equal to operator
`>`	Yes	Greater than operator
`>=`	Yes	Greater than or equal operator
`GREATEST()`	Yes	Return the largest argument
`IN()`	Yes	Check whether a value is within a set of values. *`expr` `IN (`value`,...)`* comparisons where the expression is a single value and compared values are constants of

Name	VARLEN Support	Description
		the same data type and encoding are optimized for performance. For example, the following `IN()` comparison is optimized: `SELECT * FROM Customers WHERE Country IN ('Germany', 'France', 'Spain');`
`IS`		Test a value against a boolean
`IS NOT`		Test a value against a boolean
`IS NOT NULL`	Yes	NOT NULL value test
`IS NULL`	Yes	NULL value test
`ISNULL()`		Test whether the argument is NULL
`LEAST()`	Yes	Return the smallest argument
`<`	Yes	Less than operator
`<=`	Yes	Less than or equal operator
`LIKE`	Yes	Simple pattern matching
`NOT BETWEEN ... AND ...`	Yes	Check whether a value is not within a range of values
`!=`, `<>`	Yes	Not equal operator
`NOT IN()`	Yes	Check whether a value is not within a set of values
`NOT LIKE`	Yes	Negation of simple pattern matching
`STRCMP()`	Yes	Compare two strings.

2.12.5 Control Flow Functions and Operators

The following table shows supported control flow operators. The *VARLEN Support* column identifies functions and operators that support variable-length encoded string columns. See Section 2.7.1, “Encoding String Columns”.

Table 2.5 Control Flow Functions and Operators

Name	VARLEN Support	Description
`CASE`	Yes	Case operator
`IF()`	Yes	If/else construct
`IFNULL()`	Yes	Null if/else construct
`NULLIF()`	Yes	Return NULL if expr1 = expr2

2.12.6 Data Masking and De-Identification Functions

The following table shows data masking and de-identification functions supported by HeatWave MySQL.

Table 2.6 Data Masking and De-Identification Functions

Name	Description
gen_range()	Generate random number within range
gen_rnd_email()	Generate random email address
gen_rnd_pan()	Generate random payment card Primary Account Number
gen_rnd_ssn()	Generate random US Social Security number
gen_rnd_us_phone()	Generate random US phone number
mask_inner()	Mask interior part of string
mask_outer()	Mask left and right parts of string
mask_pan()	Mask payment card Primary Account Number part of string
mask_pan_relaxed()	Mask payment card Primary Account Number part of string
mask_ssn()	Mask US Social Security number

2.12.7 Encryption and Compression Functions

The following table shows supported encryption and compression functions. Encryption and compression functions are supported with variable-length columns. These functions are not supported with dictionary-encoded columns.

Table 2.7 Encryption and Compression Functions

Name	Description
COMPRESS()	Return the result as a binary string.
MD5()	Calculate the MD5 checksum.
RANDOM_BYTES()	Return a random byte vector.
SHA(), SHA1()	Calculate an SHA-1 160-bit checksum.
SHA2()	Calculate an SHA-2 checksum.
UNCOMPRESS()	Uncompress a compressed string.
UNCOMPRESSED_LENGTH()	Return the length of a string before compression.

2.12.8 JSON Functions

The following table shows supported JSON functions.

Table 2.8 JSON Functions

Name	Description
->	Return value from JSON column after evaluating path; equivalent to JSON_EXTRACT().
->>	Return value from JSON column after evaluating path and unquoting the result; equivalent to JSON_UNQUOTE(JSON_EXTRACT()).
JSON_ARRAY()	Create JSON array
JSON_ARRAY_APPEND()	Append data to JSON document

Name	Description
JSON_ARRAY_INSERT()	Insert into JSON array
JSON_CONTAINS()	Whether JSON document contains specific object at path. Supported as of MySQL 9.0.0.
JSON_CONTAINS_PATH()	Whether JSON document contains any data at path. Supported as of MySQL 9.0.0.
JSON_DEPTH()	Maximum depth of JSON document
JSON_EXTRACT()	Return data from JSON document
JSON_INSERT()	Insert or update data into JSON document
JSON_KEYS()	Array of keys from JSON document. Supported as of MySQL 9.0.0.
JSON_LENGTH()	Number of elements in JSON document
JSON_MERGE()	Merge JSON documents, preserving all values of duplicate keys. Deprecated synonym for JSON_MERGE_PRESERVE()
JSON_MERGE_PATCH()	Merge JSON documents, replacing values of duplicate keys
JSON_MERGE_PRESERVE()	Merge JSON documents, preserving duplicate keys
JSON_OBJECT()	Create JSON object literal
JSON_OVERLAPS()	Compares two JSON documents, returns TRUE (1) if these have any key-value pairs or array elements in common, otherwise FALSE (0). Supported as of MySQL 9.0.0.
JSON_PRETTY()	Print a JSON document in human-readable format. Supported as of MySQL 9.0.0.
JSON_QUOTE()	Quote JSON document. Supported as of MySQL 9.0.0.
JSON_REMOVE()	Remove data from JSON document
JSON_REPLACE()	Replace values in JSON document
JSON_SCHEMA_VALID()	Validate JSON document against JSON schema; returns TRUE or 1 if document validates against schema, or FALSE or 0 if it does not
JSON_SCHEMA_VALIDATION_REPORT()	Validate JSON document against JSON schema; returns report in JSON format on outcome on validation including success or failure and reasons for failure
JSON_SEARCH()	Path to value within JSON document. Supported as of MySQL 9.0.0.
JSON_SET()	Insert data into JSON document
JSON_STORAGE_FREE()	Freed space within binary representation of JSON column value following partial update. Supported as of MySQL 9.0.0.
JSON_STORAGE_SIZE()	Space used for storage of binary representation of a JSON document. Supported as of MySQL 9.0.0.

Name	Description
JSON_TYPE()	Type of JSON value. Supported as of MySQL 9.0.0.
JSON_UNQUOTE()	Unquote JSON value
JSON_VALID()	Whether JSON value is valid. Supported as of MySQL 9.0.0.
JSON_VALUE()	Extract value from JSON document at location pointed to by path provided; return this value as VARCHAR(512) or specified type. Supported as of MySQL 9.0.0.
MEMBER OF()	Returns TRUE (1) if first operand matches any element of JSON array passed as second operand, otherwise returns FALSE (0). Supported as of MySQL 9.0.0.

2.12.9 Logical Operators

The following table shows supported logical operators.

Table 2.9 Logical Operators

Name	Description
AND, &&	Logical AND
NOT, !	Negates value
||, OR	Logical OR
XOR	Logical XOR

2.12.10 Mathematical Functions

The following table shows supported mathematical functions. These are not supported with dictionary-encoded columns.

Table 2.10 Mathematical Functions

Name	Description
ABS()	Return the absolute value.
ACOS()	Return the arc cosine.
ASIN()	Return the arc sine.
ATAN()	Return the arc tangent.
ATAN2(), ATAN()	Return the arc tangent of the two arguments
CEIL()	Return the smallest integer value not less than the argument. The function is not applied to BIGINT values. The input value is returned. CEIL() is a synonym for CEILING().
CEILING()	Return the smallest integer value not less than the argument. The function is not applied to BIGINT values. The input value is returned. CEILING() is a synonym for CEIL().
COS()	Return the cosine.
COT()	Return the cotangent.

Name	Description
CRC32()	Compute a cyclic redundancy check value.
DEGREES()	Convert radians to degrees.
EXP()	Raise to the power of.
FLOOR()	Return the largest integer value not greater than the argument. The function is not applied to BIGINT values. The input value is returned.
LN()	Return the natural logarithm of the argument.
LOG()	Return the natural logarithm of the first argument.
LOG10()	Return the base-10 logarithm of the argument.
LOG2()	Return the base-2 logarithm of the argument.
MOD()	Return the remainder.
PI()	Return the value of pi
POW()	Return the argument raised to the specified power
POWER()	Return the argument raised to the specified power.
RADIANS()	Return argument converted to radians.
RAND()	Return a random floating-point value.
ROUND()	Round the argument.
SIGN()	Return the sign of the argument.
SIN()	Return the sine of the argument.
SQRT()	Return the square root of the argument.
TAN()	Return the tangent of the argument.
TRUNCATE()	Truncate to specified number of decimal places.

2.12.11 String Functions and Operators

The following table shows supported string functions and operators. With the exception of the FORMAT() function, string functions and operators described in the following table are supported with variable-length encoded columns. Dictionary encoded columns are not supported.

Table 2.11 String Functions and Operators

Name	Description
ASCII()	Return numeric value of left-most character
BIN()	Return a string containing binary representation of a number
BIT_LENGTH()	Return length of argument in bits
CHAR_LENGTH()	Return number of characters in argument
CONCAT()	Return concatenated string
CONCAT_WS()	Return concatenated with separator
ELT()	Return string at index number
EXPORT_SET()	Return a string such that for every bit set in the value bits, it returns an on string (and for every unset bit, it returns an off string).

Name	Description
FIELD()	Index (position) of first argument in subsequent arguments
FIND_IN_SET()	Index (position) of first argument within second argument
FORMAT()	Return a number formatted to specified number of decimal places. *Does not support variable-length-encoded columns.*
FROM_BASE64()	Decode base64 encoded string and return result
GREATEST()	Return the largest argument
HEX()	Hexadecimal representation of decimal or string value
INSERT()	Return the index of the first occurrence of substring
INSTR()	Return the index of the first occurrence of substring
LEAST()	Return the smallest argument
LEFT()	Return the leftmost number of characters as specified
LENGTH()	Return the length of a string in bytes
LIKE	Simple pattern matching
LOCATE()	Return the position of the first occurrence of substring
LOWER()	Return the argument in lowercase
LPAD()	Return the string argument, left-padded with the specified string
LTRIM()	Remove leading spaces
MAKE_SET()	Return a set of comma-separated strings that have the corresponding bit in bits
MID()	Return a substring starting from the specified position
NOT LIKE	Negation of simple pattern matching
OCTET_LENGTH()	Synonym for LENGTH()
ORD()	Return character code for leftmost character of the argument
POSITION()	Synonym for LOCATE()
QUOTE()	Escape the argument for use in an SQL statement
REGEXP	Whether string matches regular expression
REGEXP_INSTR()	Starting index of substring matching regular expression
REGEXP_LIKE()	Whether string matches regular expression
REGEXP_REPLACE()	Replace substrings matching regular expression. Supports up to three arguments.
REGEXP_SUBSTR()	Return substring matching regular expression. Supports up to three arguments.

Name	Description
REPEAT()	Repeat a string the specified number of times
REPLACE()	Replace occurrences of a specified string
REVERSE()	Reverse the characters in a string
RIGHT()	Return the specified rightmost number of characters
RLIKE	Whether string matches regular expression
RPAD()	Append string the specified number of times
RTRIM()	Remove trailing spaces
SOUNDEX()	Return a soundex string
SPACE()	Return a string of the specified number of spaces
STRCMP()	Compare two strings
SUBSTR()	Return the substring as specified
SUBSTRING()	Return the substring as specified
SUBSTRING_INDEX()	Return a substring from a string before the specified number of occurrences of the delimiter
TO_BASE64()	Return the argument converted to a base-64 string
TRIM()	Remove leading and trailing spaces
UNHEX()	Return a string containing hex representation of a number
UPPER()	Convert to uppercase
WEIGHT_STRING()	Return the weight string for a string

2.12.12 Temporal Functions

The following table shows supported temporal functions.

As of MySQL 8.4.0, HeatWave supports named time zones such as MET or Europe/Amsterdam for CONVERT_TZ(). For a workaround before MySQL 8.4.0, see Section 2.18.3, “Functions and Operator Limitations”.

All functions support variable-length encoded string columns. See Section 2.7.1, “Encoding String Columns”.

Table 2.12 Temporal Functions

Name	Description
ADDDATE()	Add time values (intervals) to a date value
ADDTIME()	Add time
CONVERT_TZ()	Convert from one time zone to another.
CURDATE()	Return the current date
CURRENT_DATE(), CURRENT_DATE	Synonyms for CURDATE()
CURRENT_TIME(), CURRENT_TIME	Synonyms for CURTIME()
CURRENT_TIMESTAMP(), CURRENT_TIMESTAMP	Synonyms for NOW()
CURTIME()	Return the current time

Name	Description
DATE()	Extract the date part of a date or datetime expression
DATE_ADD()	Add time values (intervals) to a date value
DATE_FORMAT()	Format date as specified
DATE_SUB()	Subtract a time value (interval) from a date
DATEDIFF()	Subtract two dates
DAY()	Synonym for DAYOFMONTH()
DAYNAME()	Return the name of the weekday
DAYOFMONTH()	Return the day of the month (0-31)
DAYOFWEEK()	Return the weekday index of the argument
DAYOFYEAR()	Return the day of the year (1-366)
EXTRACT()	Extract part of a date
FROM_DAYS()	Convert a day number to a date
FROM_UNIXTIME()	Format Unix timestamp as a date
GET_FORMAT()	Return a date format string
HOUR()	Extract the hour
LAST_DAY()	Return the last day of the month for the argument
LOCALTIME(), LOCALTIME	Synonym for NOW()
LOCALTIMESTAMP, LOCALTIMESTAMP()	Synonym for NOW()
MAKEDATE()	Create a date from the year and day of year. Supports FLOAT, DOUBLE, INTEGER, and YEAR data types.
MAKETIME()	Create time from hour, minute, second
MICROSECOND()	Return the microseconds from argument
MINUTE()	Return the minute from the argument
MONTH()	Return the month from the date passed
MONTHNAME()	Return the name of the month
NOW()	Return the current date and time
PERIOD_ADD()	Add a period to a year-month
PERIOD_DIFF()	Return the number of months between periods
QUARTER()	Return the quarter from a date argument
SEC_TO_TIME()	Converts seconds to 'HH:MM:SS' format
SECOND()	Return the second (0-59)
STR_TO_DATE()	Convert a string to a date
SUBDATE()	Synonym for DATE_SUB() when invoked with three arguments
SUBTIME()	Subtract times
SYSDATE()	Return the time at which the function executes
TIME()	Extract the time portion of the expression passed

Name	Description
TIME_FORMAT()	Format as time.
TIME_TO_SEC()	Return the argument converted to seconds
TIMEDIFF()	Subtract time
TIMESTAMP()	With a single argument, this function returns the date or datetime expression; with two arguments, the sum of the arguments
TIMESTAMPADD()	Add an interval to a datetime expression
TIMESTAMPDIFF()	Subtract an interval from a datetime expression
TO_DAYS()	Return the date argument converted to days
TO_SECONDS()	Return the date or datetime argument converted to seconds since Year 0
UNIX_TIMESTAMP()	Return a Unix timestamp
UTC_DATE()	Return the current UTC date
UTC_TIME()	Return the current UTC time
UTC_TIMESTAMP()	Return the current UTC date and time
WEEK()	Return the week number. Restrictions apply. See Section 2.18.3, “Functions and Operator Limitations”
WEEKDAY()	Return the weekday index
WEEKOFYEAR()	Return the calendar week of the date (1-53)
YEAR()	Return the year
YEARWEEK()	Return the year and week

2.12.13 Vector Functions

The following table shows supported vector functions. These are not supported with dictionary-encoded columns. These were introduced in MySQL 9.0.0.

Table 2.13 Vector Functions

Name	Description
DISTANCE()	Calculates the distance between two vectors with the specified method
VECTOR_TO_STRING()	Get the string representation of a VECTOR column, given its value as a binary string
FROM_VECTOR()	A synonym for VECTOR_TO_STRING()
TO_VECTOR()	A synonym for STRING_TO_VECTOR()
VECTOR_DIM()	Get the length of a vector, that is, the number of entries it contains
VECTOR_DISTANCE	A synonym for DISTANCE()
VECTOR_TO_STRING()	Get the string representation of a VECTOR column, given its value as a binary string

- DISTANCE(*vector*, *vector*, *string*)

Calculates the distance between two vectors per the specified calculation method. It accepts the following arguments:

- A column of `VECTOR` data type.
- An input query of `VECTOR` data type.
- A string that specifies the distance metric. The supported values are `COSINE`, `DOT`, and `EUCLIDEAN`. The argument is a string, and requires quotation marks.

Examples:

```
mysql> SELECT DISTANCE(STRING_TO_VECTOR("[1.01231, 2.0123123, 3.0123123, 4.01231231]"), STRING_TO_VECTOR
+-----------------------------------------------------------------------------------------------------
| DISTANCE(STRING_TO_VECTOR("[1.01231, 2.0123123, 3.0123123, 4.01231231]"), STRING_TO_VECTOR("[1, 2, 3, 
+-----------------------------------------------------------------------------------------------------
|                                                                                                0.0000016
+-----------------------------------------------------------------------------------------------------
```

2.12.14 Window Functions

This section describes HeatWave window function support. For optimal performance, window functions in HeatWave utilize a massively parallel, partitioning-based algorithm. For general information about window functions, see Window Functions, in the *MySQL Reference Manual*.

HeatWave window function support includes support for:

- `WINDOW` and `OVER` clauses in conjunction with `PARTITION BY`, `ORDER BY`, and `WINDOW` frame specifications.
- Nonaggregate window functions supported by MySQL Server, as described in Window Function Descriptions.
- The following aggregate functions used as window functions:
  - `AVG()`
  - `COUNT()`
  - `MIN()`
  - `MAX()`
  - `STD()`.
  - `STDDEV()`.
  - `STDDEV_POP()`.
  - `STDDEV_SAMP()`.
  - `SUM()`
  - `VAR_POP()`.
  - `VAR_SAMP()`.
  - `VARIANCE()`.

2.13 SELECT Statement

`SELECT` statement clauses that are only available in the HeatWave secondary engine.

The `GROUP BY` clause permits the following:

- A `CUBE` modifier in addition to the `WITH ROLLUP` modifier.
- A `ROLLUP` modifier as a preferred alternative to the `WITH ROLLUP` modifier.

See: Section 2.12.1.1, “GROUP BY Modifiers”.

The `SELECT` statement includes the `QUALIFY` clause. This is between the `WINDOW` clause and the `ORDER BY` clause:

```
SELECT
    [ALL | DISTINCT | DISTINCTROW ]
    [HIGH_PRIORITY]
    [STRAIGHT_JOIN]
    [SQL_SMALL_RESULT] [SQL_BIG_RESULT] [SQL_BUFFER_RESULT]
    [SQL_NO_CACHE] [SQL_CALC_FOUND_ROWS]
    select_expr [, select_expr] ...
    [into_option]
    [FROM table_references
      [PARTITION partition_list]]
    [WHERE where_condition]
    [GROUP BY [CUBE] [ROLLUP] {col_name | expr | position}, ... [WITH ROLLUP]]
    [HAVING where_condition]
    [WINDOW window_name AS (window_spec)
        [, window_name AS (window_spec)] ...]
    [QUALIFY qualify_condition]
    [ORDER BY [CUBE] [ROLLUP] {col_name | expr | position}
      [ASC | DESC], ... [WITH ROLLUP]]
    [LIMIT {[offset,] row_count | row_count OFFSET offset}]
    [into_option]
    [FOR {UPDATE | SHARE}
        [OF tbl_name [, tbl_name] ...]
        [NOWAIT | SKIP LOCKED]
      | LOCK IN SHARE MODE]
    [into_option]

into_option: {
    INTO OUTFILE 'file_name'
        [CHARACTER SET charset_name]
        export_options
  | INTO DUMPFILE 'file_name'
  | INTO var_name [, var_name] ...
}
```

For a full explanation of the `SELECT` syntax, see: SELECT Statement.

In addition to constraints similar to the `HAVING` clause, the `QUALIFY` clause can also include predicates related to a window function.

Similar to the `HAVING` clause, the `QUALIFY` clause can refer to aliases mentioned in the `SELECT` list.

The `QUALIFY` clause requires the inclusion of at least one window function in the query. The window function can be part of any one of the following:

- The `SELECT` column list.
- A filter predicate of the `QUALIFY` clause.

The following query uses `SUM()` as a window function:

```
mysql> SELECT
          year, country, product, profit,
          SUM(profit) OVER() AS total_profit,
          SUM(profit) OVER(PARTITION BY country) AS country_profit
          FROM sales
          ORDER BY country, year, product, profit;
+------+---------+------------+--------+--------------+----------------+
| year | country | product    | profit | total_profit | country_profit |
+------+---------+------------+--------+--------------+----------------+
| 2000 | Finland | Computer   |   1500 |         7535 |           1610 |
| 2000 | Finland | Phone      |    100 |         7535 |           1610 |
| 2001 | Finland | Phone      |     10 |         7535 |           1610 |
| 2000 | India   | Calculator |     75 |         7535 |           1350 |
| 2000 | India   | Calculator |     75 |         7535 |           1350 |
| 2000 | India   | Computer   |   1200 |         7535 |           1350 |
| 2000 | USA     | Calculator |     75 |         7535 |           4575 |
| 2000 | USA     | Computer   |   1500 |         7535 |           4575 |
| 2001 | USA     | Calculator |     50 |         7535 |           4575 |
| 2001 | USA     | Computer   |   1200 |         7535 |           4575 |
| 2001 | USA     | Computer   |   1500 |         7535 |           4575 |
| 2001 | USA     | TV         |    100 |         7535 |           4575 |
| 2001 | USA     | TV         |    150 |         7535 |           4575 |
+------+---------+------------+--------+--------------+----------------+
```

The same query, but now with a `QUALIFY` clause that only returns results where `country_profit` is greater than 1,500:

```
mysql> SELECT
          year, country, product, profit,
          SUM(profit) OVER() AS total_profit,
          SUM(profit) OVER(PARTITION BY country) AS country_profit
          FROM sales
          QUALIFY  country_profit  > 1500
          ORDER BY country, year, product, profit;
+------+---------+------------+--------+--------------+----------------+
| year | country | product    | profit | total_profit | country_profit |
+------+---------+------------+--------+--------------+----------------+
| 2000 | Finland | Computer   |   1500 |         7535 |           1610 |
| 2000 | Finland | Phone      |    100 |         7535 |           1610 |
| 2001 | Finland | Phone      |     10 |         7535 |           1610 |
| 2000 | USA     | Calculator |     75 |         7535 |           4575 |
| 2000 | USA     | Computer   |   1500 |         7535 |           4575 |
| 2001 | USA     | Calculator |     50 |         7535 |           4575 |
| 2001 | USA     | Computer   |   1200 |         7535 |           4575 |
| 2001 | USA     | Computer   |   1500 |         7535 |           4575 |
| 2001 | USA     | TV         |    100 |         7535 |           4575 |
| 2001 | USA     | TV         |    150 |         7535 |           4575 |
+------+---------+------------+--------+--------------+----------------+
```

The `SELECT` statement includes the `TABLESAMPLE` clause, which only applies to base tables:

```
SELECT
    select_expr [, select_expr] ...
    [into_option]
    FROM table_references
      [PARTITION partition_list]
    TABLESAMPLE { SYSTEM | BERNOULLI } ( sample_percentage )

into_option: {
    INTO OUTFILE 'file_name'
        [CHARACTER SET charset_name]
        export_options
  | INTO DUMPFILE 'file_name'
  | INTO var_name [, var_name] ...
}
```

The TABLESAMPLE clause retrieves a sample of the data in a table with the SYSTEM or BERNOULLI sampling method.

The *sample_percentage* can be an integer, decimal or float numeric value.

SYSTEM sampling samples chunks, and each chunk has a random number. Bernoulli sampling samples rows, and each row has a random number. If the random number is less than or equal to the *sample_percentage*, the chunk or row is chosen for subsequent processing.

A TABLESAMPLE example that counts the number of rows from a join of a 10% sample of the LINEITEM table with a 10% sample of the ORDERS table. It applies TABLESAMPLE with the SYSTEM sampling method to both base tables.

```
mysql> SELECT COUNT(*)
        FROM LINEITEM TABLESAMPLE SYSTEM (10), ORDERS TABLESAMPLE SYSTEM (10)
        WHERE L_ORDERKEY=O_ORDERKEY;
```

2.14 String Column Encoding Reference

HeatWave supports two string column encoding types:

- Section 2.14.1, “Variable-length Encoding”
- Section 2.14.2, “Dictionary Encoding”

String column encoding is automatically applied when tables are loaded into HeatWave. Variable-length encoding is the default.

To use dictionary encoding, you must define the encoding type explicitly for individual string columns. See Section 2.7.1, “Encoding String Columns”.

2.14.1 Variable-length Encoding

Variable-length (VARLEN) encoding has the following characteristics:

- It is the default encoding type. No action is required to use variable-length encoding. It is applied to string columns by default when tables are loaded with the exception of string columns defined explicitly as dictionary-encoded columns.
- It minimizes the amount of data stored for string columns by efficiently storing variable length column values.
- It is more efficient than dictionary encoding with respect to storage and processing of string columns with a high number of distinct values relative to the cardinality of the table.
- It permits more operations involving string columns to be offloaded than dictionary encoding.
- It supports all character sets and collation types supported by the MySQL DB System. User defined character sets are not supported.
- VARLEN columns can be declared as NULL.

2.14.1.1 VARLEN Supported Expressions, Filters, Functions, and Operators

For supported functions and operators, refer to Section 2.12, “Supported Functions and Operators”.

VARLEN Supported Filters

- Column-to-column filters, excluding the <=> filter.

- Column-to-constant filters, excluding the <=> filter.

VARLEN Supported Relational Operators

- GROUP BY
- JOIN
- LIMIT
- ORDER BY

2.14.1.2 VARLEN Encoding Limits

- For information about HeatWave column limits and how they relate to VARLEN encoded columns, see Section 2.14.3, “Column Limits”.
- Only expressions with non-boolean types are supported.

2.14.1.3 VARLEN Column Memory Requirements

- For HeatWave nodes, a VARLEN encoded column value requires enough memory for the data plus two bytes for length information. Internal fragmentation or headers can affect the actual amount of memory required.
- There is no memory requirement on the MySQL DB System node, apart from a small memory footprint for metadata.

2.14.1.4 VARLEN Encoding and Performance

- The presence of VARLEN encoded VARCHAR or CHAR columns does not affect table load performance.
- Table load and change propagation operations perform more slowly on VARLEN encoded TEXT type columns than on VARLEN encoded VARCHAR columns.
- There are two main differences with respect to HeatWave result processing for variable-length encoding compared to dictionary encoding:
  - A dictionary decode operation is not required, which means that fewer CPU cycles are required.
  - Because VARLEN encoded columns use a larger number of bytes than dictionary-encoded columns, the network cost for sending results from HeatWave to the MySQL DB System is greater.

2.14.2 Dictionary Encoding

Dictionary encoding (SORTED) has the following characteristics:

- Best suited to string columns with a low number of distinct values relative to the cardinality of the table. Dictionary encoding reduces the space required for column values on the HeatWave nodes but requires space on the MySQL DB System node for dictionaries.
- Supports GROUP BY and ORDER BY operations on string columns.
- Supports only a subset of the operations supported by variable-length encoding such as LIKE with prefix expressions, and comparison with the exact same column. Dictionary-encoded columns cannot be compared in any way with other columns or constants, or with other dictionary-encoded columns.
- Does not support JOIN operations.

- Does not support operations that use string operators. Queries that use string operators on dictionary-encoded string columns are not offloaded.
- Does not support `LIKE` predicates.
- Does not support comparison with variable-length encoded columns.
- The dictionaries required to decode dictionary-encoded string columns must fit in MySQL DB System node memory. Dictionary size depends on the size of the column and the number of distinct values. Load operations for tables with dictionary-encoded string columns that have a high number of distinct values can fail if there is not enough available memory on the MySQL DB System node.

2.14.3 Column Limits

HeatWave has the following column limits:

- The maximum column width is 65532 bytes.
- The column limit for base relations, tables as loaded into HeatWave, is 1017.
- The column limit for intermediate relations (intermediate tables used by HeatWave when processing queries) is 1800.
- The actual column limit when running queries depends on factors such as MySQL limits, protocol limits, the total number of columns, column types, and column widths. For example, for any HeatWave physical operator, the maximum number of 65532-byte `VARLEN` encoded columns is 31 if the query only uses `VARLEN` encoded columns. On the other hand, HeatWave can produce a maximum of 1800 `VARLEN` encoded columns that are less than 1024 bytes in size if the query includes only `VARLEN` encoded columns. If the query includes only non-string columns such as `DOUBLE`, `INTEGER`, `DECIMAL`, and so on, 1800 columns are supported.

2.15 Troubleshooting

- *Problem:* Queries are not offloaded.
  - *Solution A:* Your query contains an unsupported predicate, function, operator, or has encountered some other limitation. See Section 2.3.1, “Query Prerequisites”.
  - *Solution B:* Query execution time is less than the query cost threshold.

    HeatWave is designed for fast execution of large analytic queries. Smaller, simpler queries, such as those that use indexes for quick lookups, often execute faster on the MySQL DB System. To avoid offloading inexpensive queries to HeatWave, the optimizer uses a query cost estimate threshold value. Only queries that exceed the threshold value on the MySQL DB System are considered for offload.

    The query cost threshold unit value is the same unit value used by the MySQL optimizer for query cost estimates. The threshold is 100000.00000. The ratio between a query cost estimate value and the actual time required to execute a query depends on the type of query, the type of hardware, and MySQL DB System configuration.

    To determine the cost of a query on the MySQL DB System:

    1. Disable `use_secondary_engine` to force MySQL DB System execution.

       ```
       mysql> SET SESSION use_secondary_engine=OFF;
       ```

    2. Run the query using `EXPLAIN`.

```
mysql> EXPLAIN select_query;
```

3. Query the `Last_query_cost` status variable. If the value is less than 100000.00000, the query cannot be offloaded.

```
mysql> SHOW STATUS LIKE 'Last_query_cost';
```

- *Solution C:* The table you are querying is not loaded. You can check the load status of a table in HeatWave by querying `LOAD_STATUS` data from HeatWave Performance Schema tables. For example:

```
mysql> USE performance_schema;
mysql> SELECT NAME, LOAD_STATUS FROM rpd_tables,rpd_table_id
          WHERE rpd_tables.ID = rpd_table_id.ID;
+------------------------------+--------------------+
| NAME                         | LOAD_STATUS        |
+------------------------------+--------------------+
| tpch.supplier                | AVAIL_RPDGSTABSTATE |
| tpch.partsupp                | AVAIL_RPDGSTABSTATE |
| tpch.orders                  | AVAIL_RPDGSTABSTATE |
| tpch.lineitem                | AVAIL_RPDGSTABSTATE |
| tpch.customer                | AVAIL_RPDGSTABSTATE |
| tpch.nation                  | AVAIL_RPDGSTABSTATE |
| tpch.region                  | AVAIL_RPDGSTABSTATE |
| tpch.part                    | AVAIL_RPDGSTABSTATE |
+------------------------------+--------------------+
```

For information about load statuses, see Section 7.3.9, “The rpd_tables Table”.

Alternatively, run the following statement:

```
mysql> ALTER TABLE tbl_name SECONDARY_LOAD;
```

The following error is reported if the table is already loaded:

```
mysql> ERROR 13331 (HY000): Table is already loaded.
```

- *Solution D:* The HeatWave Cluster has failed. To determine the status of the HeatWave Cluster, run the following statement:

```
mysql> SHOW GLOBAL STATUS LIKE 'rapid_plugin_bootstrapped';
+---------------------------------+------------+
| Variable_name                   | Value      |
+---------------------------------+------------+
| rapid_plugin_bootstrapped       | YES        |
+---------------------------------+------------+
```

See Chapter 6, *System and Status Variables* for `rapid_plugin_bootstrapped` status values.

If the HeatWave Cluster has failed, restart it in the HeatWave Console and reload the data if necessary. The HeatWave recovery mechanism should reload the data automatically.

- *Problem:* You have encountered an out-of-memory error when executing a query.

  Solution: HeatWave optimizes for network usage rather than memory. If you encounter out of memory errors when running a query, try running the query with the `MIN_MEM_CONSUMPTION` strategy by setting `rapid_execution_strategy` before executing the query:

  ```
  mysql> SET SESSION rapid_execution_strategy = MIN_MEM_CONSUMPTION;
  ```

  Also consider checking the size of your data by performing a cluster size estimate. If your data has grown substantially, you may require additional HeatWave nodes.

  - For HeatWave on OCI, see Generating a Node Count Estimate in the *HeatWave on OCI Service Guide*.
  - For HeatWave on AWS, see Estimating Cluster Size with HeatWave Autopilot in the *HeatWave on AWS Service Guide*.
  - For HeatWave for Azure, see Provisioning HeatWave Nodes in the *HeatWave for Azure Service Guide*.

  Avoid or rewrite queries that produce a Cartesian product. For more information, see Section 2.9.7, “Running Queries”.

- *Problem:* A table load operation fails with “ERROR HY000: Error while running parallel scan.”

  Solution: A `TEXT` type column value larger than 65532 bytes is rejected during `SECONDARY_LOAD` operations. Reduce the size of the `TEXT` type column value to less than 65532 bytes or exclude the column before loading the table. See Section 2.2.2.1, “Excluding Table Columns”.

- *Problem:* Change propagation fails with the following error: “Blob/text value of *n* bytes was encountered during change propagation but RAPID supports text values only up to 65532 bytes.”

  Solution: `TEXT` type values larger than 65532 bytes are rejected during change propagation. Reduce the size of `TEXT` type values to less than 65532 bytes. Should you encounter this error, check the change propagation status for the affected table. If change propagation is disabled, reload the table. See Section 2.2.7, “Change Propagation”.

- *Problem:* A warning was encountered when running Auto Parallel Load.

  Solution: When Auto Parallel Load encounters an issue that produces a warning, it automatically switches to `dryrun` mode to prevent further problems. In this case, the load statements generated by the Auto Parallel Load utility can still be obtained using the SQL statement provided in the utility output, but avoid those load statements or use them with caution, as they may be problematic.

  - If a warning message indicates that the HeatWave Cluster or service is not active or online, this means that the load cannot start because a HeatWave Cluster is not attached to the MySQL DB System or is not active. In this case, provision and enable a HeatWave Cluster and run Auto Parallel Load again.
  - If a warning message indicates that MySQL host memory is insufficient to load all of the tables, the estimated dictionary size for dictionary-encoded columns may be too large for MySQL host memory. Try changing column encodings to `VARLEN` to free space in MySQL host memory.
  - If a warning message indicates that HeatWave Cluster memory is insufficient to load all of the tables, the estimated table size is too large for HeatWave Cluster memory. Try excluding certain schemas or tables from the load operation or increase the size of the cluster.

  - If a warning message indicates that a concurrent table load is in progress, this means that another client session is currently loading tables into HeatWave. While the concurrent load operation is in progress, the accuracy of Auto Parallel Load estimates cannot be guaranteed. Wait until the concurrent load operation finishes before running Auto Parallel Load.

- *Problem:* During retrieval of the generated Auto Parallel Load or Advisor DDL statements, an error message indicates that the `heatwave_autopilot_report` table or the `heatwave_advisor_report` table or the `heatwave_load_report` table does not exist. For example:

```
mysql> SELECT log->>"$.sql" AS "SQL Script"
          FROM sys.heatwave_autopilot_report
          WHERE type = "sql"
          ORDER BY id;
ERROR 1146 (42S02): Table 'sys.heatwave_autopilot_report' does not exist

mysql> SELECT log->>"$.sql" AS "SQL Script"
          FROM sys.heatwave_advisor_report
          WHERE type = "sql"
          ORDER BY id;
ERROR 1146 (42S02): Table 'sys.heatwave_advisor_report' does not exist


mysql> SELECT log->>"$.sql" AS "SQL Script"
          FROM sys.heatwave_load_report
          WHERE type = "sql"
          ORDER BY id;
ERROR 1146 (42S02): Table 'sys.heatwave_load_report' does not exist
```

  Solution: This error can occur when querying a report table from a different session. Query the report table using the same session that issued the Auto Parallel Load or Advisor `CALL` statement. This error also occurs if the session used to call Auto Parallel Load or Advisor has timed out or was terminated. In this case, run Auto Load or Advisor again before querying the report table.

2.16 Metadata Queries

This section provides Information Schema and Performance Schema queries that you can use to retrieve HeatWave metadata.

> **Note**
>
> For queries that monitor HeatWave node status, memory usage, data loading, change propagation, and queries, see Section 7.1, “HeatWave MySQL Monitoring”.

2.16.1 Secondary Engine Definitions

To identify tables on the MySQL DB System that are defined with a secondary engine, query the `CREATE_OPTIONS` column of the `INFORMATION_SCHEMA.TABLES` table. The `CREATE_OPTIONS` column shows the `SECONDARY_ENGINE` clause, if defined. Tables with `SECONDARY_ENGINE="RAPID"` are loaded into HeatWave, and changes to them are automatically propagated to their counterpart tables in the HeatWave Cluster.

```
mysql> SELECT TABLE_SCHEMA, TABLE_NAME, CREATE_OPTIONS
          FROM INFORMATION_SCHEMA.TABLES
          WHERE CREATE_OPTIONS LIKE '%SECONDARY_ENGINE%' AND TABLE_SCHEMA LIKE 'tpch';
+--------------+------------+--------------------------+
| TABLE_SCHEMA | TABLE_NAME | CREATE_OPTIONS           |
+--------------+------------+--------------------------+
```

```
| tpch         | customer   | SECONDARY_ENGINE="RAPID" |
| tpch         | lineitem   | SECONDARY_ENGINE="RAPID" |
| tpch         | nation     | SECONDARY_ENGINE="RAPID" |
| tpch         | orders     | SECONDARY_ENGINE="RAPID" |
| tpch         | part       | SECONDARY_ENGINE="RAPID" |
| tpch         | partsupp   | SECONDARY_ENGINE="RAPID" |
| tpch         | region     | SECONDARY_ENGINE="RAPID" |
| tpch         | supplier   | SECONDARY_ENGINE="RAPID" |
+--------------+------------+--------------------------+
```

You can also view create options for an individual table using `SHOW CREATE TABLE`.

2.16.2 Excluded Columns

To identify table columns defined as `NOT SECONDARY` on the MySQL DB System, query the `EXTRA` column of the `INFORMATION_SCHEMA.COLUMNS` table. Columns defined with the `NOT SECONDARY` column attribute are not loaded into HeatWave when executing a table load operation.

```
mysql> SELECT COLUMN_NAME, EXTRA
          FROM INFORMATION_SCHEMA.COLUMNS
          WHERE TABLE_NAME LIKE 't1' AND EXTRA LIKE '%NOT SECONDARY%';
+-------------+---------------+
| COLUMN_NAME | EXTRA         |
+-------------+---------------+
| O_COMMENT   | NOT SECONDARY |
+-------------+---------------+
```

You can also view columns defined as `NOT SECONDARY` for an individual table using `SHOW CREATE TABLE`.

2.16.3 String Column Encoding

- To identify explicitly encoded string columns in tables on the MySQL DB System, query the `COLUMN_COMMENT` column of the `INFORMATION_SCHEMA.COLUMNS` table. For example:

  ```
  mysql> SELECT COLUMN_NAME, COLUMN_COMMENT
            FROM INFORMATION_SCHEMA.COLUMNS
            WHERE TABLE_NAME LIKE 'orders' AND COLUMN_COMMENT LIKE '%ENCODING%';
  +-----------------+-----------------------------+
  | COLUMN_NAME     | COLUMN_COMMENT              |
  +-----------------+-----------------------------+
  | O_CLERK         | RAPID_COLUMN=ENCODING=SORTED |
  | O_ORDERPRIORITY | RAPID_COLUMN=ENCODING=SORTED |
  | O_ORDERSTATUS   | RAPID_COLUMN=ENCODING=SORTED |
  | O_CLERK         | RAPID_COLUMN=ENCODING=SORTED |
  | O_ORDERPRIORITY | RAPID_COLUMN=ENCODING=SORTED |
  | O_ORDERSTATUS   | RAPID_COLUMN=ENCODING=SORTED |
  +-----------------+-----------------------------+
  ```

  You can also view explicitly defined column encodings for an individual table using `SHOW CREATE TABLE`.

- To view the dictionary size for dictionary-encoded columns, in bytes:

  ```
  mysql> USE performance_schema;
  mysql> SELECT rpd_table_id.TABLE_NAME, rpd_columns.COLUMN_ID, rpd_columns.DICT_SIZE_BYTES
            FROM rpd_table_id, rpd_columns
            WHERE rpd_table_id.ID = rpd_columns.TABLE_ID AND rpd_columns.DICT_SIZE_BYTES > 0
            ORDER BY rpd_table_id.TABLE_NAME;
  ------------+-----------+-----------------+
  | TABLE_NAME | COLUMN_ID | DICT_SIZE_BYTES |
  +------------+-----------+-----------------+
  ```

```
| orders     |         3 |        25165912 |
+------------+-----------+-----------------+
```

2.16.4 Data Placement

- To identify columns defined as data placement keys in tables on the MySQL DB System, query the `COLUMN_COMMENT` column of the `INFORMATION_SCHEMA.COLUMNS` table. For example:

```
mysql> SELECT COLUMN_NAME, COLUMN_COMMENT
          FROM INFORMATION_SCHEMA.COLUMNS
          WHERE TABLE_NAME LIKE 'orders' AND COLUMN_COMMENT LIKE '%DATA_PLACEMENT_KEY%';
+-----------------+----------------------------------+
| COLUMN_NAME     | COLUMN_COMMENT                   |
+-----------------+----------------------------------+
| O_CUSTKEY       | RAPID_COLUMN=DATA_PLACEMENT_KEY=1 |
+-----------------+----------------------------------+
```

  You can also view data placement keys for an individual table using `SHOW CREATE TABLE`.

- To identify columns defined as data placement keys in tables that are loaded in HeatWave, query the `DATA_PLACEMENT_INDEX` column of the `performance_schema.rpd_columns` table for columns with a `DATA_PLACEMENT_INDEX` value greater than 0, which indicates that the column is defined as a data placement key. For example:

```
mysql> SELECT TABLE_NAME, COLUMN_NAME, DATA_PLACEMENT_INDEX
          FROM performance_schema.rpd_columns r1
          JOIN performance_schema.rpd_column_id r2 ON r1.COLUMN_ID = r2.ID
          WHERE r1.TABLE_ID = (SELECT ID FROM performance_schema.rpd_table_id
          WHERE TABLE_NAME = 'orders') AND r2.TABLE_NAME = 'orders'
          AND r1.DATA_PLACEMENT_INDEX > 0 ORDER BY r1.DATA_PLACEMENT_INDEX;
+------------+--------------+----------------------+
| TABLE_NAME | COLUMN_NAME  | DATA_PLACEMENT_INDEX |
+------------+--------------+----------------------+
| orders     | O_TOTALPRICE |                    1 |
| orders     | O_ORDERDATE  |                    2 |
| orders     | O_COMMENT    |                    3 |
+------------+--------------+----------------------+
```

  For information about data placement key index values, see Section 2.7.2, “Defining Data Placement Keys”.

- To determine if data placement partitions were used by a `JOIN` or `GROUP BY` query, you can query the `QEP_TEXT` column of the `performance_schema.rpd_query_stats` table to view `prepart` data. (`prepart` is short for “pre-partitioning”.) The `prepart` data for a `GROUP BY` operation contains a single value; for example: `"prepart":#`, where `#` represents the number of HeatWave nodes. A value greater than 1 indicates that data placement partitions were used. The `prepart` data for a `JOIN` operation has two values that indicate the number of HeatWave nodes; one for each `JOIN` branch; for example: `"prepart":[#,#]`. A value greater than 1 for a `JOIN` branch indicates that the `JOIN` branch used data placement partitions. (A value of `"prepart":[1,1]` indicates that data placement partitions were not used by either `JOIN` branch.) `prepart` data is only generated if a `GROUP BY` or `JOIN` operation is executed. To query `QEP_TEXT` `prepart` data for the last executed query:

```
mysql> SELECT CONCAT( '"prepart":[', (JSON_EXTRACT(QEP_TEXT->>"$**.prepart", '$[0][0]')),
          "," ,(JSON_EXTRACT(QEP_TEXT->>"$**.prepart", '$[0][1]')) , ']' )
          FROM performance_schema.rpd_query_stats
          WHERE query_id = (select max(query_id)
          FROM performance_schema.rpd_query_stats);
+-------------------------------------------------------------------------------+
| concat( '"prepart":[', (JSON_EXTRACT(QEP_TEXT->>"$**.prepart", '$[0][0]')), |
|"," ,(JSON_EXTRACT(QEP_TEXT->>"$**.prepart", '$[0][1]')) , ']' )              |
+-------------------------------------------------------------------------------+
| "prepart":[2,2]                                                               |
```

```
+--------------------------------------------------------------------------+
```

2.17 Bulk Ingest Data to MySQL Server

MySQL includes a bulk load extension to the `LOAD DATA` statement that is only available in HeatWave MySQL. It adds the following features:

- The optimized loading of data sorted by primary key.
- The optimized loading of unsorted data.
- The optimized loading of data from an object store.
- The optimized loading of data from a series of files.

MySQL 8.4.0 adds the following features:

- Loading a MySQL Shell dump file.
- Loading ZSTD compressed CSV files.
- Monitor bulk load progress with the Performance Schema.

MySQL 9.0.0 adds the following features:

- Large data support and additional data types.

Use a second session to monitor bulk load progress:

- If the data is sorted, there is a single stage: `loading`.
- If the data is unsorted, there are two stages: `sorting` and `loading`.

See: Section 2.18.8, “Bulk Ingest Data to MySQL Server Limitations”.

Bulk Ingest Data Type Support

`LOAD DATA` with `ALGORITHM=BULK` supports the following data types:

- `INT`, `SMALLINT`, `TINYINT`, and `BIGINT`
- `CHAR` and `BINARY`.
- `VARCHAR` and `VARBINARY`. Before MySQL 9.0.0 the record must fit in the page as there is no large data support.
- `NUMERIC` and `DECIMAL`.
- `UNSIGNED NUMERIC` and `DECIMAL`.
- `DOUBLE` and `FLOAT`.
- `DATE` and `DATETIME`.

As of MySQL 9.0.0, `LOAD DATA` with `ALGORITHM=BULK` supports the following data types:

- `BIT`.

- ENUM.
- JSON.
- SET.
- TIMESTAMP.
- YEAR.
- TINYBLOB, BLOB, MEDIUMBLOB, and LONGBLOB.
- TINYTEXT, TEXT, MEDIUMTEXT, and LONGTEXT.
- GEOMETRY, GEOMETRYCOLLECTION, POINT, MULTIPOINT, LINESTRING, MULTILINESTRING, POLYGON, and MULTIPOLYGON,

For the data types that HeatWave supports, see: Section 2.10, “Supported Data Types”

See: Section 2.18.8, “Bulk Ingest Data to MySQL Server Limitations”.

Bulk Ingest Syntax

```
mysql> LOAD DATA
    [LOW_PRIORITY | CONCURRENT]
    [FROM]
    INFILE | URL | S3 'file_prefix' | 'options' [COUNT N]
    [IN PRIMARY KEY ORDER]
    INTO TABLE tbl_name
    [CHARACTER SET charset_name] [COMPRESSION = {'ZSTD'}]
    [{FIELDS | COLUMNS}
        [TERMINATED BY 'string']
        [[OPTIONALLY] ENCLOSED BY 'char']
        [ESCAPED BY 'char']
    ]
    [LINES
        [TERMINATED BY 'string']
    ]
    [IGNORE number {LINES | ROWS}]
    [PARALLEL = number]
    [MEMORY = M]
    [ALGORITHM = BULK]

options: {
     JSON_OBJECT("key","value"["key","value"] ...)
          "key","value": {
          "url-prefix","prefix"
          ["url-sequence-start",0]
          ["url-suffix","suffix"]
          ["url-prefix-last-append","@"]
          ["is-dryrun",{true|false}]
          }
}
```

The additional LOAD DATA clauses are:

- FROM: Makes the statement more readable.
- URL: A URL accessible with a HTTP GET request.
- S3: The AWS S3 file location.

  This requires the user privilege LOAD_FROM_S3.

See: Section 2.18.8, “Bulk Ingest Data to MySQL Server Limitations”.

- `COUNT`: The number of files in a series of files.

  For `COUNT 5` and *`file_prefix`* set to `data.csv.`, the five files would be: `data.csv.1`, `data.csv.2`, `data.csv.3`, `data.csv.4`, and `data.csv.5`.

- `IN PRIMARY KEY ORDER`: Use when the data is already sorted. The values should be in ascending order within the file.

  For a file series, the primary keys in each file must be disjoint and in ascending order from one file to the next.

- `PARALLEL`: The number of concurrent threads to use. A typical value might be 16, 32 or 48. The default value is 16.

  `PARALLEL` does not require `CONCURRENT`.

- `MEMORY`: The amount of memory to use. A typical value might be 512M or 4G. The default value is 1G.

- `ALGORITHM`: Set to `BULK` for bulk load. The file format is CSV.

- `COMPRESSION`: The file compression algorithm. Bulk load supports the ZSTD algorithm.

- `options` is a JSON object literal that includes:

  - `url-prefix`: The common URL prefix for the files to load.

  - `url-sequence-start`: The sequence number for the first file.

    The default value is 1, and the minimum value is 0. The value cannot be a negative number. The value can be a string or a number, for example, "134", or "default".

  - `url-suffix`: The file suffix.

  - `url-prefix-last-append`: The string to append to the prefix of the last file.

    This supports MySQL Shell dump files.

  - `is-dryrun`: Set to `true` to run basic checks and report if bulk load is possible on the given table. The default value is `false`.

    To enable `is-dryrun`, use any of the following values: `true`, `"true"`, `"1"`, `"on"` or `1`.

    To disable `is-dryrun`, use any of the following values: `false`, `"false"`, `"0"`, `"off"` or `0`.

`LOAD DATA` with `ALGORITHM=BULK` does not support these clauses:

```
LOAD DATA
    [LOCAL]
    [REPLACE | IGNORE]
    [PARTITION (partition_name [, partition_name] ...)]
    ]
    [LINES
        [STARTING BY 'string']
    ]
    [(col_name_or_user_var
        [, col_name_or_user_var] ...)]
    [SET col_name={expr | DEFAULT}
```

```
        [, col_name={expr | DEFAULT}] ...]
```

Syntax Examples

- An example that loads unsorted data from AWS S3 with 48 concurrent threads and 4G of memory:

```
mysql> GRANT LOAD_FROM_S3 ON *.* TO load_user@localhost;

mysql> LOAD DATA FROM S3 's3-us-east-1://innodb-bulkload-dev-1/lineitem.tbl'
          INTO TABLE lineitem
          FIELDS TERMINATED BY "|"
          OPTIONALLY ENCLOSED BY '"'
          LINES TERMINATED BY '\n'
          PARALLEL = 48
          MEMORY = 4G
          ALGORITHM=BULK;
```

- An example that loads eight files of sorted data from AWS S3. The `file_prefix` ends with a period. The files are `lineitem.tbl.1`, `lineitem.tbl.2`, ... `lineitem.tbl.8`:

```
mysql> GRANT LOAD_FROM_S3 ON *.* TO load_user@localhost;

mysql> LOAD DATA FROM S3 's3-us-east-1://innodb-bulkload-dev-1/lineitem.tbl.' COUNT 8
          IN PRIMARY KEY ORDER
          INTO TABLE lineitem
          FIELDS TERMINATED BY "|"
          OPTIONALLY ENCLOSED BY '"'
          LINES TERMINATED BY '\n'
          ALGORITHM=BULK;
```

- An example that performs a dry run on a sequence of MySQL Shell dump files compressed with the ZSTD algorithm:

```
mysql> GRANT LOAD_FROM_URL ON *.* TO load_user@localhost;

mysql> LOAD DATA FROM URL
          '{"url-prefix","https://example.com/bucket/test@lineitem@","url-sequence-start",0,"url-suffix"
          COUNT 20
          INTO TABLE lineitem
          CHARACTER SET ???? COMPRESSION = {'ZSTD'}
          FIELDS TERMINATED BY "|"
          OPTIONALLY ENCLOSED BY '"'
          LINES TERMINATED BY '\n'
          IGNORE 20000 LINES
          ALGORITHM=BULK;
```

- An example that monitors bulk load progress in a second session.

  - Review the list of stages with the following query:

```
mysql> SELECT NAME, ENABLED, TIMED FROM performance_schema.setup_instruments
          WHERE ENABLED='YES' AND NAME LIKE "stage/bulk_load%";
```

  - Enable the `events_stages_current` with the following query:

```
mysql> UPDATE performance_schema.setup_consumers
          SET ENABLED = 'YES' WHERE NAME LIKE 'events_stages_current';
```

  - Use one session to run bulk load, and monitor progress in a second session:

```
mysql> SELECT thread_id, event_id, event_name, WORK_ESTIMATED,  WORK_COMPLETED
          FROM performance_schema.events_stages_current;
--------------
SELECT thread_id, event_id, event_name, WORK_ESTIMATED,  WORK_COMPLETED FROM performance_schema.events
```

```
--------------

+-----------+----------+---------------------------------+----------------+----------------+
| thread_id | event_id | event_name                      | WORK_ESTIMATED | WORK_COMPLETED |
+-----------+----------+---------------------------------+----------------+----------------+
|        49 |        5 | stage/bulk_load_unsorted/sorting |     1207551343 |      583008145 |
+-----------+----------+---------------------------------+----------------+----------------+
1 row in set (0.00 sec)
```

2.18 HeatWave MySQL Limitations

This section lists functionality that is not supported by HeatWave. It is not an exhaustive list with respect to data types, functions, operators, and SQL modes. For data types, functions, operators, and SQL modes that are supported by HeatWave, see Section 2.10, “Supported Data Types”, Section 2.12, “Supported Functions and Operators”, and Section 2.11, “Supported SQL Modes”. If a particular data type, function, operator, or SQL mode does not appear in those tables and lists, it should be considered unsupported.

2.18.1 Change Propagation Limitations

- Cascading changes triggered by a foreign key constraint.

2.18.2 Data Type Limitations

- Spatial data types. See Spatial Data Types.
- String and text data types::
  - `BINARY`
  - `VARBINARY`
- Decimal values with a precision greater than 18 in expression operators, with the exception of the following:
  - Arithmetic operators, see: Section 2.12.2, “Arithmetic Operators”.
  - `CAST()`
  - `COALESCE()`
  - `CASE`
  - `IF()`
  - `NULLIF()`
  - `ABS()`
  - `CEILING()`
  - `FLOOR()`
  - `ROUND()`
  - `TRUNCATE()`
  - `GREATEST()`
  - `LEAST()`

- `ENUM` type columns as part of a `UNION`, `EXCEPT`, `EXCEPT ALL`, `INTERSECT`, or `INTERSECT ALL` `SELECT` list or as a `JOIN` key, except when used inside a supported expression.
- `ENUM` type columns as part of a non-top level `UNION ALL SELECT` list or as a `JOIN` key, except when used inside a supported expression.
- `ENUM` type support is limited to:
  - Comparison with string or numeric constants, and other numeric, non-temporal expressions (numeric columns, constants, and functions with a numeric result).
  - Comparison operators (<, <=, <=>, =, >=, >, and `BETWEEN`) with numeric arguments.
  - Comparison operators (=, <=>, and <>) with string constants.
  - *`enum_col`* `IS [NOT] {NULL|TRUE|FALSE}`
  - The `IN()` function in combination with numeric arguments (constants, functions, or columns) and string constants.
  - `COUNT()`, `SUM()`, and `AVG()` aggregation functions on `ENUM` columns. The functions operate on the numeric index value, not the associated string value.
  - `CAST(`*`enum_col`* `AS {[N]CHAR [(X)]|SIGNED|UNSIGNED|FLOAT|DOUBLE|DECIMAL [(M,N)]})`. The numeric index value is cast, not the associated string value.
  - `CAST(`*`enum_col`*`) AS {[N]CHAR}` is supported only in the `SELECT` list and when it is not nested in another expression.
- Temporal types are supported only with strict SQL mode. See Strict SQL Mode.

2.18.3 Functions and Operator Limitations

- Bit functions and operators.
- `COALESCE()` as a `JOIN` predicate.
- `CONVERT_TZ()` with named time zones before MySQL 8.4.0. Only datetime values are supported. Rewrite queries that use named time zones with equivalent datetime values. For example:

  ```
  mysql> SELECT CONVERT_TZ(O_ORDERDATE, 'UTC','EST') FROM tpch.orders;
  ```

  Rewrite as:

  ```
  mysql> SELECT CONVERT_TZ(O_ORDERDATE, '+00:00','-05:00') FROM tpch.orders;
  ```

- Only `UNIX_TIMESTAMP()` and `FROM_UNIXTIME()` functions support a *`timezone`* value specified as an offset from UTC in the form of *`[H]H:MM`* and prefixed with a + or -. For example:

  ```
  mysql> SET time_zone = timezone;
  ```

  For information about time zone offsets, see MySQL Server Time Zone Support.
- `COUNT(NULL)` in cases where it is used as an input argument for non-aggregate operators.
- All known limitations for `COUNT(DISTINCT)` also apply to `HLL()`.
- Full-text search functions.
- XML, Spatial, and other domain specific functions.

- The following JSON functions:
  - `JSON_TABLE()`
- Loadable Functions.
- `GROUP_CONCAT()` with:
  - Dictionary-encoded string columns, `RAPID_COLUMN=ENCODING=SORTED`. See Section 2.14.2, “Dictionary Encoding”.
  - As of MySQL 9.0.1-u1, the GROUP_CONCAT() function supports CUBE and ROLLUP non-primitive GROUP BY transformation options.
  - As of MySQL 9.0.1-u1 `COUNT(DISTINCT)` function supports CUBE, ROLLUP and WITH ROLLUP options.
  - `JSON_SET()`
  - `JSON_TABLE()`
- `COUNT(DISTINCT)` with:
  - As of MySQL 9.0.1-u1 `COUNT(DISTINCT)` function supports CUBE, ROLLUP and WITH ROLLUP options.
- A `CASE` control flow operator or `IF()` function that contains columns not within an aggregation function and not part of the `GROUP BY` key.
- String functions and operators on columns that are not `VARLEN` encoded. See Section 2.7.1, “Encoding String Columns”.
- The `AVG()` aggregate function with enumeration and temporal data types.

- The following aggregate functions with the enumeration data type:
  - `STD()`
  - `STDDEV()`
  - `STDDEV_POP()`
  - `STDDEV_SAMP()`
  - `SUM()`
  - `VARIANCE()`
  - `VAR_POP()`
  - `VAR_SAMP()`

  With the exception of `SUM()`, the same aggregate functions within a semi-join predicate due to the nondeterministic nature of floating-point results and potential mismatches. For example, the following use is not supported:

  ```
  mysql> SELECT FROM A WHERE a1 IN (SELECT VAR_POP(b1) FROM B);
  ```

  The same aggregate functions with numeric data types other than those supported by HeatWave. See Section 2.10, “Supported Data Types”.

- `WEEK(date[,mode])` does not support the `default_week_format` system variable.

  To use the `mode` argument, the *`mode`* value must be defined explicitly.

2.18.4 Index Hint and Optimizer Hint Limitations

- Index and optimizer hints. See Index Hints, and Optimizer Hints.
- Semijoin strategies other than `FIRSTMATCH`.

  MySQL attempts to enforce the `FIRSTMATCH` strategy and ignores all other semijoin strategies specified explicitly as subquery optimizer hints. However, MySQL may still select the `DUPSWEEDOUT` semijoin strategy during `JOIN` order optimization, even if an equivalent plan could be offered using the `FIRSTMATCH` strategy. A plan that uses the `DUPSWEEDOUT` semijoin strategy would produce incorrect results if executed on HeatWave.

  For general information about subquery optimizer hints, see Subquery Optimizer Hints.

2.18.5 Join Limitations

- Antijoins, with the exception of supported `IN` and `EXISTS` antijoin variants listed below.
- `EXISTS` semijoins and antijoins are supported in the following variants only:
  - `SELECT ... WHERE ... EXISTS (...)`
  - `SELECT ... WHERE ... EXISTS (...) IS TRUE`
  - `SELECT ... WHERE ... EXISTS (...) IS NOT FALSE`
  - `SELECT ... WHERE ... NOT EXISTS (...) IS FALSE`

  - `SELECT ... WHERE ... NOT EXISTS (...) IS NOT TRUE`

  Depending on transformations and optimizations performed by MySQL, other variants of `EXISTS` semijoins may or may not be offloaded.

- `IN` semijoins and antijoins other than the following variants:

  - `SELECT ... WHERE ... IN (...)`
  - `SELECT ... WHERE ... IN (...) IS TRUE`
  - `SELECT ... WHERE ... NOT IN (...) IS FALSE`

  Depending on transformations and optimizations performed by MySQL, other variants of `IN` semijoins may or may not be offloaded.

- A query with a supported semijoin or antijoin condition may be rejected for offload due to how MySQL optimizes and transforms the query.

- Semijoin and antijoin queries use the best plan found after evaluating the first 10000 possible plans, or after investigating 10000 possible plans since the last valid plan.

  The plan evaluation count is reset to zero after each derived table, after an outer query, and after each subquery. The plan evaluation limit is required because the `DUPSWEEDOUT` join strategy, which is not supported by HeatWave, may be used as a fallback strategy by MySQL during join order optimization (for related information, see FIRSTMATCH). The plan evaluation limit prevents too much time being spent evaluating plans in cases where MySQL generates numerous plans that use the `DUPSWEEDOUT` semijoin strategy.

- Outer join queries without an equality condition defined for the two tables.

- Some outer join queries with `IN ... EXISTS` sub-queries (semi-joins) in the `ON` clause.

2.18.6 Partition Selection Limitations

Before MySQL 8.4.0, HeatWave does not support explicit partition selection.

As of MySQL 8.4.0, HeatWave supports InnoDB partitions with the following limitations:

- HeatWave cannot load partitions with Auto Parallel Load.
- HeatWave cannot load partitions from a snapshot of the database.
- HeatWave cannot load partitions from a table that contains dictionary-encoded columns.
- HeatWave maintains partition information in memory. This information is not available during a restart, and the automatic reload on restart has to reload the entire table.
- HeatWave can unload up to 1,000 partitions in a single statement. Use additional statements to unload more than 1,000 partitions.
- HeatWave cannot load or unload partitions during the recovery process.

2.18.7 Variable Limitations

- A `group_concat_max_len` session variable with a value that exceeds the maximum HeatWave column length (65532 bytes).

- The `sql_select_limit` as a global variable.

  It is only supported as a session variable.

- `time_zone` and `timestamp` variable settings are not passed to HeatWave when queries are offloaded.

2.18.8 Bulk Ingest Data to MySQL Server Limitations

- HeatWave on OCI does not support `LOAD DATA` with `ALGORITHM=BULK`.
- HeatWave on AWS does support `LOAD DATA` with `ALGORITHM=BULK`, but does not support the `INFILE` and `URL` clauses.
- `LOAD DATA` with `ALGORITHM=BULK` has the following limitations:
  - It locks the target table exclusively and does not allow other operations on the table.
  - It does not support automatic rounding or truncation of the input data. It will fail if the input data requires rounding or truncation in order to be loaded.
  - It does not support temporary tables.
  - It is atomic but not transactional. It commits any transaction that is already running. On failure the `LOAD DATA` statement is completely rolled back.
  - It cannot execute when the target table is explicitly locked by a `LOCK TABLES` statement.
- The target table for `LOAD DATA` with `ALGORITHM=BULK` has the following limitations:
  - It must be empty. The state of the table should be as though it has been freshly created. If the table has instantly added/dropped column, call TRUNCATE before calling `LOAD DATA` with `ALGORITHM=BULK`.
  - It must not be partitioned.
  - It must not contain secondary indexes.
  - It must be in a file_per_tablespace, and must not be in a shared tablespace.
  - It must have the default row format, `ROW_FORMAT=DYNAMIC`. Use `ALTER TABLE` to make any changes to the table after `LOAD DATA` with `ALGORITHM=BULK`.
  - It must contain a primary key, but the primary key must not have a prefix index.
  - It must not contain virtual or stored generated columns.
  - It must not contain foreign keys.
  - It must not contain `CHECK` constraints.
  - It must not contain triggers.
  - It is not replicated to other nodes.
  - It must not use a secondary engine. Set the secondary engine after the after `LOAD DATA` with `ALGORITHM=BULK`. See: Section 2.2.2.2, “Defining the Secondary Engine”.

2.18.9 Other Limitations

- Most non-default MySQL DB System SQL modes.

  For a list of supported SQL modes, see Section 2.11, “Supported SQL Modes”.

- The `gb18030_chinese_ci` character set and collation.

- The `WITH ROLLUP` modifier in `GROUP BY` clauses in the following cases:

  - In queries that contain distinct aggregations.

  - In queries that contain duplicate `GROUP BY` keys.

- Cursors inside stored programs are not supported before MySQL 9.0.0, see: Cursors.

- `UNION ALL` queries with an `ORDER BY` or `LIMIT` clause, between dictionary-encoded columns, or between `ENUM` columns.

  `EXCEPT`, `EXCEPT ALL`, `INTERSECT`, `INTERSECT ALL`, and `UNION` queries with or without an `ORDER BY` or `LIMIT` clause, between dictionary-encoded columns, or between `ENUM` columns.

  `EXCEPT`, `EXCEPT ALL`, `INTERSECT`, `INTERSECT ALL`, `UNION` and `UNION ALL` subqueries with or without an `ORDER BY` or `LIMIT` clause, between dictionary-encoded columns, between `ENUM` columns, or specified in an `IN` or `EXISTS` clause.

- Comparison predicates, `GROUP BY`, `JOIN`, and so on, if the key column is `DOUBLE PRECISION`.

- Queries with an impossible `WHERE` condition (queries known to have an empty result set).

  For example, the following query is not offloaded:

  ```
  mysql> SELECT AVG(c1) AS value FROM t1 WHERE c1 IS NULL;
  ```

- Primary keys with column prefixes.

- Virtual generated columns.

- Queries that are executed as part of a trigger.

- Queries that call a stored program.

- Queries that are executed as part of a stored program.

- Queries that are part of a multi-statement transaction.

- Materialized views.

  Only nonmaterialized views are supported. See Section 2.3.10, “Using Views”.

- Partial query offload for regular `SELECT` queries.

  If all elements of the query are supported, the entire query is offloaded; otherwise, the query is executed on the MySQL DB System by default.

  HeatWave supports `CREATE TABLE ... SELECT` and `INSERT ... SELECT` statements where only the `SELECT` portion of the operation is offloaded to HeatWave. See Section 2.3, “Running Queries”.

- Named time zones are not supported before MySQL 8.4.0.

- Row widths in intermediate and final query results that exceed 4MB in size.

  A query that exceeds this row width limit is not offloaded to HeatWave for processing.

- Consecutive filter operations on derived tables.

  For example, the following query is not supported:

```
mysql> SELECT * FROM (SELECT * FROM t1 WHERE x < 7) tt1,
         (SELECT * FROM t1 WHERE x < y) tt2
         WHERE  tt1.x > 5 AND tt1.x = tt2.x;
```

  The query uses a filter for table `tt1` in the table scan of table `t1` (`x < 7`) followed by a consecutive filter on table `tt1` (`tt1.x > 5`) in the `WHERE` clause.

- Recursive common table expressions.

- Operations involving `ALTER TABLE` such as loading, unloading, or recovering data when MySQL Server is running in `SUPER_READ_ONLY` mode.

  MySQL Server is placed in `SUPER_READ_ONLY` mode when MySQL Server disk space drops below a set amount for a specific duration. For information about thresholds that control this behavior and how to disable `SUPER_READ_ONLY` mode, see Resolving SUPER_READ_ONLY and OFFLINE_MODE Issue in the *HeatWave on OCI Service Guide*.

Chapter 3 HeatWave AutoML

Table of Contents

3.1 HeatWave AutoML Features

HeatWave AutoML makes it easy to use machine learning, whether you are a novice user or an experienced ML practitioner. You provide the data, and HeatWave AutoML analyzes the characteristics of the data and creates an optimized machine learning model that you can use to generate predictions and explanations. An ML model makes predictions by identifying patterns in your data and applying those patterns to unseen data. HeatWave AutoML *explanations* help you understand how predictions are made, such as which features of a dataset contribute most to a prediction.

3.1.1 HeatWave AutoML Supervised Learning

HeatWave AutoML supports supervised machine learning. That is, it creates a machine learning model by analyzing a labeled dataset to learn patterns that enable it to predict labels based on the features of the dataset. For example, this guide uses the *Census Income Data Set* in its examples, where features such as age, education, occupation, country, and so on, are used to predict the income of an individual (the label).

Once a model is created, it can be used on unseen data, where the label is unknown, to make predictions. In a business setting, predictive models have a variety of possible applications such as predicting customer churn, approving or rejecting credit applications, predicting customer wait times, and so on.

HeatWave AutoML supports both classification and regression models. A classification model predicts discrete values, such as whether an email is spam or not, whether a loan application should be approved or rejected, or what product a customer might be interested in purchasing. A regression model predicts continuous values, such as customer wait times, expected sales, or home prices, for example. The model type is selected during training, with classification being the default type.

3.1.2 HeatWave AutoML Ease of Use

HeatWave AutoML is purpose-built for ease of use. It requires no machine learning expertise, specialized tools, or algorithms. With HeatWave AutoML and a set of training data, you can train a predictive machine learning model with a single call to the `ML_TRAIN` SQL routine; for example:

```
CALL sys.ML_TRAIN('heatwaveml_bench.census_train', 'revenue', NULL, @census_model);
```

The ML_TRAIN routine leverages Oracle AutoML technology to automate training of machine learning models. For information about Oracle AutoML, see Section 3.1.4, “Oracle AutoML”.

You can use a model created by ML_TRAIN with other HeatWave AutoML routines to generate predictions and explanations; for example, this call to the ML_PREDICT_TABLE routine generates predictions for a table of input data:

```
CALL sys.ML_PREDICT_TABLE('heatwaveml_bench.census_test', @census_model,
'heatwaveml_bench.census_predictions');
```

All HeatWave AutoML operations are initiated by running CALL or SELECT statements, which can be easily integrated into your applications. HeatWave AutoML routines reside in the MySQL sys schema and can be run from any MySQL client or application that is connected to a MySQL DB System with a HeatWave Cluster. HeatWave AutoML routines include:

- ML_TRAIN: Trains a machine learning model for a given training dataset.
- ML_PREDICT_ROW: Makes predictions for one or more rows of data.
- ML_PREDICT_TABLE: Makes predictions for a table of data.
- ML_EXPLAIN_ROW: Explains predictions for one or more rows of data.
- ML_EXPLAIN_TABLE: Explains predictions for a table of data.
- ML_SCORE: Computes the quality of a model.
- ML_MODEL_LOAD: Loads a machine learning model for predictions and explanations.
- ML_MODEL_UNLOAD: Unloads a machine learning model.

In addition, with HeatWave AutoML, there is no need to move or reformat your data. Data and machine learning models never leave the HeatWave Service, which saves you time and effort while keeping your data and models secure.

3.1.3 HeatWave AutoML Workflow

A typical HeatWave AutoML workflow is described below:

1. When the ML_TRAIN routine is called, HeatWave AutoML calls the MySQL DB System where the training data resides. The training data is sent from the MySQL DB System and distributed across the HeatWave Cluster, which performs machine learning computation in parallel. See Section 3.5, “Training a Model”.
2. HeatWave AutoML analyzes the training data, trains an optimized machine learning model, and stores the model in a model catalog on the MySQL DB System. See Section 3.14.1, “The Model Catalog”.
3. HeatWave AutoML ML_PREDICT_* and ML_EXPLAIN_* routines use the trained model to generate predictions and explanations on test or unseen data. See Section 3.7, “Predictions”, and Section 3.8, “Explanations”.
4. Predictions and explanations are returned to the MySQL DB System and to the user or application that issued the query.

Optionally, the ML_SCORE routine can be used to compute the quality of a model to ensure that predictions and explanations are reliable. See Section 3.14.6, “Scoring Models”.

HeatWave AutoML shares resources with HeatWave MySQL. For information about concurrent queries see: Section 2.3.3, “Auto Scheduling”.

3.1.4 Oracle AutoML

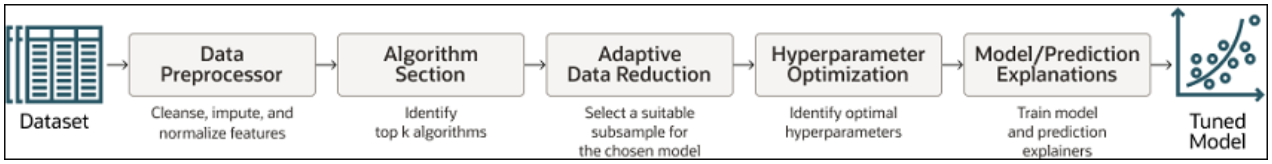
The HeatWave AutoML `ML_TRAIN` routine leverages Oracle AutoML technology to automate the process of training a machine learning model. Oracle AutoML replaces the laborious and time consuming tasks of the data analyst whose workflow is as follows:

1. Selecting a model from a large number of viable candidate models.
2. For each model, tuning hyperparameters.
3. Selecting only predictive features to speed up the pipeline and reduce over-fitting.
4. Ensuring the model performs well on unseen data (also called generalization).

Oracle AutoML automates this workflow, providing you with an optimal model given a time budget. The Oracle AutoML pipeline used by the HeatWave AutoML `ML_TRAIN` routine has these stages:

- Data preprocessing
- Algorithm selection
- Adaptive data reduction
- Hyperparameter optimization
- Model and prediction explanations

Figure 3.1 Oracle AutoML Pipeline



Oracle AutoML also produces high quality models very efficiently, which is achieved through a scalable design and intelligent choices that reduce trials at each stage in the pipeline.

- *Scalable design*: The Oracle AutoML pipeline is able to exploit both HeatWave internode and intranode parallelism, which improves scalability and reduces runtime.
- *Intelligent choices reduce trials in each stage*: Algorithms and parameters are chosen based on dataset characteristics, which ensures that the model is accurate and efficiently selected. This is achieved using meta-learning throughout the pipeline.

For additional information about Oracle AutoML, refer to Yakovlev, Anatoly, et al. "Oracle AutoML: A Fast and Predictive AutoML Pipeline." Proceedings of the VLDB Endowment 13.12 (2020): 3166-3180.

3.2 HeatWave AutoML Prerequisites

As of MySQL 9.0.0, HeatWave AutoML can support large models that are only limited by the amount of memory defined by the shape.

- An operational MySQL DB System.
  - For HeatWave on OCI, see Creating a DB System in the *HeatWave on OCI Service Guide*.
  - For HeatWave on AWS, see Creating a DB System in the *HeatWave on AWS Service Guide*.
  - For HeatWave for Azure, see Provisioning HeatWave in the *HeatWave for Azure Service Guide*.

- An operational HeatWave Cluster.
  - For HeatWave on OCI, see Adding a HeatWave Cluster in the *HeatWave on OCI Service Guide*.
  - For HeatWave on AWS, see Creating a HeatWave Cluster in the *HeatWave on AWS Service Guide*.
  - For HeatWave for Azure, see Provisioning HeatWave Nodes in the *HeatWave for Azure Service Guide*.
- MySQL Shell 8.0.22 or higher.
  - For HeatWave on OCI, see Connecting to a DB System in the *HeatWave on OCI Service Guide*.
  - For HeatWave on AWS, see Connecting with MySQL Shell in the *HeatWave on AWS Service Guide*.
- The MySQL account that will train a model does not have a period character (".") in its name; for example, a user named `'joesmith'@'%'` is permitted to train a model, but a user named `'joe.smith'@'%'` is not. For more information about this requirement, see Section 3.19, "HeatWave AutoML Limitations".
- The MySQL account that will use HeatWave AutoML has been granted the following privileges:
  - `SELECT` and `ALTER` privileges on the schema that contains the machine learning datasets; for example:

    ```
    mysql> GRANT SELECT, ALTER ON schema_name.* TO 'user_name'@'%';
    ```

  - `SELECT` and `EXECUTE` on the MySQL `sys` schema where HeatWave AutoML routines reside; for example:

    ```
    mysql> GRANT SELECT, EXECUTE ON sys.* TO 'user_name'@'%';
    ```

3.3 Getting Started

Once you have access to a MySQL DB System with a HeatWave Cluster, and you have obtained the MySQL user privileges described in Section 3.2, "HeatWave AutoML Prerequisites", you can start using HeatWave AutoML.

Proceed through the following steps to prepare data, train a model, make predictions, and generate explanations:

1. Prepare and load training and test data. See Section 3.4, "Preparing Data".
2. Train a machine learning model. See Section 3.5, "Training a Model".
3. Make predictions with test data using a trained model. See Section 3.7, "Predictions".
4. Run explanations on test data using a trained model to understand how predictions are made. See Section 3.8, "Explanations".
5. Score your machine learning model to assess its reliability. See Section 3.14.6, "Scoring Models".
6. View a model explanation to understand how the model makes predictions. See Section 3.14.7, "Model Explanations".

Alternatively, you can jump ahead to the *Iris Data Set Machine Learning Quickstart*, which provides a quick run-through of HeatWave AutoML capabilities using a simple, well-known machine learning data set. See Section 8.4, "Iris Data Set Machine Learning Quickstart".

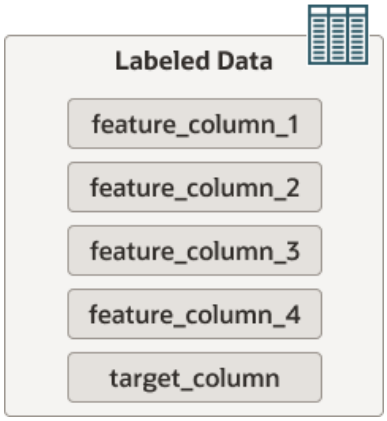
3.4 Preparing Data

HeatWave AutoML works with labeled and unlabeled data. Labeled data is used to train and score machine learning models. Unlabeled data is required when generating predictions and explanations.

3.4.1 Labeled Data

Labeled data has feature columns and a target column (the *label*), as illustrated in the following diagram:

Figure 3.2 Labeled Data



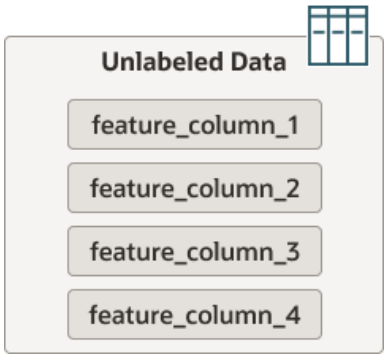
Feature columns contain the input variables used to train the machine learning model. The target column contains *ground truth values* or, in other words, the correct answers. A labeled dataset with ground truth values is required to train a machine learning model. In the context of this guide, the labeled dataset used to train a machine learning model is referred as the *training dataset*.

A labeled dataset with ground truth values is also used to score a model (compute its accuracy and reliability). This dataset should have the same columns as the *training dataset* but with a different set of data. In the context of this guide, the labeled dataset used to score a model is referred as the *validation dataset*.

3.4.2 Unlabeled Data

Unlabeled data has feature columns but no target column (no answers), as illustrated below:

Figure 3.3 Unlabeled Data



Unlabeled data is required to generate predictions and explanations. It must have exactly the same feature columns as the training dataset but no target column. In the context of this guide, the unlabeled data used

for predictions and explanations is referred to as the *test dataset*. Test data starts as labeled data but the label is removed for the purpose of trialing the machine learning model.

The “unseen data” that you will eventually use with your model to make predictions is also unlabeled data. Like the *test dataset*, unseen data must have exactly the same feature columns as the training dataset but no target column.

For examples of training, validation, and test dataset tables and how they are structured, see Section 3.4.4, “Example Data”, and Section 8.4, “Iris Data Set Machine Learning Quickstart”.

3.4.3 General Data Requirements

General requirements for HeatWave AutoML data include the following:

- Each dataset must reside in a single table on the MySQL DB System. HeatWave AutoML routines such as `ML_TRAIN`, `ML_PREDICT_TABLE`, and `ML_EXPLAIN_TABLE` operate on a single table.

  For information about loading data into a MySQL DB System, see Importing and Exporting Databases in the *HeatWave on OCI Service Guide*.

- Tables used with HeatWave AutoML must not exceed 10 GB, 100 million rows, or 1017 columns.

- Table columns must use supported data types. For supported data types and recommendations for how to handle unsupported types, see Section 3.17, “Supported Data Types”.

- NaN (Not a Number) values are not recognized by MySQL and should be replaced by `NULL`.

- The target column in a training dataset for a classification model must have at least two distinct values, and each distinct value should appear in at least five rows. For a regression model, only a numeric target column is permitted.

> **Note**
>
> The `ML_TRAIN` routine ignores columns missing more than 20% of its values and columns with the same value in each row. Missing values in numerical columns are replaced with the average value of the column, standardized to a mean of 0 and with a standard deviation of 1. Missing values in categorical columns are replaced with the most frequent value, and either one-hot or ordinal encoding is used to convert categorical values to numeric values. The input data as it exists in the MySQL database is not modified by `ML_TRAIN`.

3.4.4 Example Data

Examples in this guide use the *Census Income Data Set*.

Dua, D. and Graff, C. (2019). UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information.

> **Note**
>
> *Census Income Data Set* examples demonstrate `classification` training and inference. HeatWave AutoML also supports `regression` training and inference for datasets suited for that purpose. The `ML_TRAIN task` parameter defines whether the machine learning model is trained for `classification` or `regression`.

To replicate the examples in this guide, perform the following steps to create the required schema and tables. Python 3 and MySQL Shell are required.

1. Create the following schema and tables on the MySQL DB System by executing the following statements:

```
mysql> CREATE SCHEMA heatwaveml_bench;

mysql> USE heatwaveml_bench;

mysql> CREATE TABLE census_train (
          age INT, workclass VARCHAR(255),
          fnlwgt INT, education VARCHAR(255),
          `education-num` INT,
          `marital-status` VARCHAR(255),
          occupation VARCHAR(255),
          relationship VARCHAR(255),
          race VARCHAR(255),
          sex VARCHAR(255),
          `capital-gain` INT,
          `capital-loss` INT,
          `hours-per-week` INT,
          `native-country` VARCHAR(255),
          revenue VARCHAR(255));

mysql> CREATE TABLE `census_test` LIKE `census_train`;
```

2. Navigate to the *HeatWave AutoML Code for Performance Benchmarks* GitHub repository at https://github.com/oracle-samples/heatwave-ml.

3. Follow the `README.md` instructions to create `census_train.csv` and `census_test.csv` data files. In summary, the instructions are:

   a. Install the required Python packages:

   ```
   $> pip install pandas==1.2.3 numpy==1.22.2 unlzw3==0.2.1 sklearn==1.0.2
   ```

   b. Download or clone the repository, which includes the census source data and preprpocessing script.

   c. Run the `preprocess.py` script to create the `census_train.csv` and `census_test.csv` data files.

   ```
   $> python3 heatwave-ml/preprocess.py --benchmark census
   ```

   > **Note**
   >
   > Do not run the benchmark as instructed in the `README.md` file. The benchmark script removes the schema and data at the end of processing.

4. Start MySQL Shell with the `--mysql` option to open a `ClassicSession`, which is required when using the Parallel Table Import Utility.

```
$> mysqlsh --mysql Username@IPAddressOfMySQLDBSystemEndpoint
```

5. Load the data from the `.csv` files into the MySQL DB System using the following commands:

```
MySQL>JS> util.importTable("census_train.csv",{table: "census_train",
               dialect: "csv-unix", skipRows:1})

MySQL>JS> util.importTable("census_test.csv",{table: "census_test",
               dialect: "csv-unix", skipRows:1})
```

6. Create a validation table:

```
mysql> CREATE TABLE `census_validate` LIKE `census_test`;
```

```
mysql> INSERT INTO `census_validate` SELECT * FROM `census_test`;
```

7. Modify the `census_test` table to remove the target `` `revenue` `` column:

```
mysql> ALTER TABLE `census_test` DROP COLUMN `revenue`;
```

Other Example Data Sets

For other example data sets to use with HeatWave AutoML, refer to the HeatWave AutoML Code for Performance Benchmarks GitHub repository.

3.4.5 Example Text Data

HeatWave AutoML supports text data types. To create a sample text data set, use the `fetch_20newsgroups` data set from scikit-learn. This also uses the pandas Python library.

```
$> from sklearn.datasets import fetch_20newsgroups
$> import pandas as pd

$> categories = ['alt.atheism', 'talk.religion.misc', 'comp.graphics', 'sci.space']

$> newsgroups_train = fetch_20newsgroups(subset='train', remove=('headers', 'footers', 'quotes'), categori
$> df = pd.DataFrame([newsgroups_train.data, newsgroups_train.target.tolist()]).T
$> df.columns = ['text', 'target']
$> targets = pd.DataFrame( newsgroups_train.target_names)
$> targets.columns=['category']
$> out = pd.merge(df, targets, left_on='target', right_index=True).drop('target', axis=1)
$> out = out[(out.text != '')] #remove empty strings
$> out.to_csv('20newsgroups_train.csv', index=False)

$> newsgroups_test = fetch_20newsgroups(subset='test', remove=('headers', 'footers', 'quotes'), categories=
$> df = pd.DataFrame([newsgroups_test.data, newsgroups_test.target.tolist()]).T
$> df.columns = ['text', 'target']
$> targets = pd.DataFrame( newsgroups_test.target_names)
$> targets.columns=['category']
$> out = pd.merge(df, targets, left_on='target', right_index=True).drop('target', axis=1)
$> out = out[(out.text != '')] #remove empty strings
$> out.to_csv('20newsgroups_test.csv', index=False)
```

Then load the csv files into MySQL:

```
mysql> DROP TABLE IF EXISTS `20newsgroups_train`;
mysql> DROP TABLE IF EXISTS `20newsgroups_test`;
mysql> CREATE TABLE `20newsgroups_train` (`text` LONGTEXT DEFAULT NULL, `target` VARCHAR(255) DEFAULT NULL
mysql> CREATE TABLE `20newsgroups_test` LIKE `20newsgroups_train`;

mysql-js> util.importTable("20newsgroups_train.csv",{table: "20newsgroups_train", dialect: "csv-unix", ski
mysql-js> util.importTable("20newsgroups_test.csv",{table: "20newsgroups_test", dialect: "csv-unix", skipR
```

3.5 Training a Model

Run the `ML_TRAIN` routine on a training dataset to produce a trained machine learning model.

`ML_TRAIN` supports training of classification, regression, and forecasting models. Use a classification model to predict discrete values. Use a regression model to predict continuous values. Use a forecasting model to create timeseries forecasts for temporal data.

The time required to train a model can take a few minutes to a few hours depending on the number of rows and columns in the dataset, specified `ML_TRAIN` parameters, and the size of the HeatWave Cluster. HeatWave AutoML supports tables up to 10 GB in size with a maximum of 100 million rows and or 1017 columns.

ML_TRAIN stores machine learning models in the MODEL_CATALOG table. See Section 3.14.1, “The Model Catalog”.

For ML_TRAIN option descriptions, see Section 3.16.1, “ML_TRAIN”.

The training dataset used with ML_TRAIN must reside in a table on the MySQL DB System. For an example training dataset, see Section 3.4.4, “Example Data”.

The following example runs ML_TRAIN on the heatwaveml_bench.census_train training dataset:

```
mysql> CALL sys.ML_TRAIN('heatwaveml_bench.census_train', 'revenue',
         JSON_OBJECT('task', 'classification'), @census_model);
```

Where:

- heatwaveml_bench.census_train is the fully qualified name of the table that contains the training dataset (*schema_name.table_name*).
- revenue is the name of the target column, which contains ground truth values.
- JSON_OBJECT('task', 'classification') specifies the machine learning task type.

  Specify NULL instead of JSON options to use the default classification task type.

  When using the regression task type, only a numeric target column is permitted.

  For the forecasting task type, see Section 3.9, “Forecasting”.

  For the anomaly_detection task type, see Section 3.10, “Anomaly Detection”

  For the recommendation task type, see Section 3.11, “Recommendations”
- @census_model is the name of the user-defined session variable that stores the model handle for the duration of the connection. User variables are written as *@var_name*. Some of the examples in this guide use @census_model as the variable name. Any valid name for a user-defined variable is permitted, for example @my_model).

After ML_TRAIN trains a model, the model is stored in the model catalog. To retrieve the generated model handle, query the specified session variable; for example:

```
mysql> SELECT @census_model;
+-------------------------------------------------+
| @census_model                                   |
+-------------------------------------------------+
| heatwaveml_bench.census_train_user1_1636729526  |
+-------------------------------------------------+
```

> **Tip**
>
> While using the same connection used to execute ML_TRAIN, specify the session variable, for example @census_model, in place of the model handle in other HeatWave AutoML routines, but the session variable data is lost when the current session is terminated. If you need to look up a model handle, you can do so by querying the model catalog table. See Section 3.14.8, “Model Handles”.

The quality and reliability of a trained model can be assessed using the ML_SCORE routine. For more information, see Section 3.14.6, “Scoring Models”. ML_TRAIN displays the following message if a trained model has a low score: Model Has a low training score, expect low quality model explanations.

3.5.1 Advanced ML_TRAIN Options

The `ML_TRAIN` routine provides advanced options to influence model selection and training.

- The `model_list` option permits specifying the type of model to be trained. If more than one type of model specified, the best model type is selected from the list. For a list of supported model types, see Section 3.16.13, “Model Types”. This option cannot be used together with the `exclude_model_list` option.

  The following example trains either an `XGBClassifier` or `LGBMClassifier` model.

```
mysql> CALL sys.ML_TRAIN('heatwaveml_bench.census_train', 'revenue',
          JSON_OBJECT('task','classification', 'model_list',
          JSON_ARRAY('XGBClassifier', 'LGBMClassifier')), @census_model);
```

- The `exclude_model_list` option specifies types of models that should not be trained. Specified model types are excluded from consideration. For a list of model types you can specify, see Section 3.16.13, “Model Types”. This option cannot be used together with the `model_list` option.

  The following example excludes the `LogisticRegression` and `GaussianNB` models.

```
mysql> CALL sys.ML_TRAIN('heatwaveml_bench.census_train', 'revenue',
          JSON_OBJECT('task','classification',
          'exclude_model_list', JSON_ARRAY('LogisticRegression', 'GaussianNB')),
          @census_model);
```

- The `optimization_metric` option specifies a scoring metric to optimize for. See: Section 3.16.14, “Optimization and Scoring Metrics”.

  The following example optimizes for the `neg_log_loss` metric.

```
mysql> CALL sys.ML_TRAIN('heatwaveml_bench.census_train', 'revenue',
          JSON_OBJECT('task','classification', 'optimization_metric', 'neg_log_loss'),
          @census_model);
```

- The `exclude_column_list` option specifies feature columns to exclude from consideration when training a model.

  The following example excludes the `'age'` column from consideration when training a model for the `census` dataset.

```
mysql> CALL sys.ML_TRAIN('heatwaveml_bench.census_train', 'revenue',
          JSON_OBJECT('task','classification', 'exclude_column_list', JSON_ARRAY('age')),
          @census_model);
```

3.6 Training Explainers

After the `ML_TRAIN` routine, use the `ML_EXPLAIN` routine to train prediction explainers and model explainers for HeatWave AutoML. In earlier releases, the `ML_TRAIN` routine trains the default Permutation Importance model and prediction explainers.

Explanations help you understand which features have the most influence on a prediction. Feature importance is presented as a value ranging from -1 to 1. A positive value indicates that a feature contributed toward the prediction. A negative value indicates that the feature contributed toward a different prediction; for example, if a feature in a loan approval model with two possible predictions ('approve' and 'reject') has a negative value for an 'approve' prediction, that feature would have a positive value for a 'reject' prediction. A value of 0 or near 0 indicates that the feature value has no impact on the prediction to which it applies.

Prediction explainers are used when you run the ML_EXPLAIN_ROW and ML_EXPLAIN_TABLE routines to generate explanations for specific predictions. You must train a prediction explainer for the model before you can use those routines. The ML_EXPLAIN routine can train these prediction explainers:

- The Permutation Importance prediction explainer, specified as permutation_importance, is the default prediction explainer, which explains the prediction for a single row or table.
- The SHAP prediction explainer, specified as shap, uses feature importance values to explain the prediction for a single row or table.

Model explainers are used when you run the ML_EXPLAIN routine to explain what the model learned from the training dataset. The model explainer provides a list of feature importances to show what features the model considered important based on the entire training dataset. The ML_EXPLAIN routine can train these model explainers:

- The Partial Dependence model explainer, specified as partial_dependence, shows how changing the values of one or more columns will change the value that the model predicts. When you train this model explainer, you need to specify some additional options.
- The SHAP model explainer, specified as shap, produces global feature importance values based on Shapley values.
- The Fast SHAP model explainer, specified as fast_shap, is a subsampling version of the SHAP model explainer which usually has a faster runtime.
- The Permutation Importance model explainer, specified as permutation_importance, is the default model explainer.

The model explanation is stored in the model catalog along with the machine learning model (see Section 3.14.1, “The Model Catalog”). If you run ML_EXPLAIN again for the same model handle and model explainer, the field is overwritten with the new result.

Before you run ML_EXPLAIN, you must load the model, for example:

```
mysql> CALL sys.ML_MODEL_LOAD('ml_data.iris_train_user1_1636729526', NULL);
```

The following example runs ML_EXPLAIN to train the SHAP model explainer and the Permutation Importance prediction explainer for the model:

```
mysql> CALL sys.ML_EXPLAIN('ml_data.iris_train', 'class', 'ml_data.iris_train_user1_1636729526',
         JSON_OBJECT('model_explainer', 'shap', 'prediction_explainer', 'permutation_importance'));
```

Where:

- ml_data.iris_train is the fully qualified name of the table that contains the training dataset (*schema_name.table_name*).
- class is the name of the target column, which contains ground truth values.
- ml_data.iris_train_*user1*_1636729526 is the model handle for the model in the model catalog. You can use a session variable to specify the model handle instead, written as *@var_name*.
- JSON is a list of key-value pairs naming the model explainer and prediction explainer that are to be trained for the model. In this case, model_explainer specifies shap for the SHAP model explainer, and prediction_explainer specifies permutation_importance for the Permutation Importance model explainer.

This example runs ML_EXPLAIN to train the Partial Dependence model explainer (which requires extra options) and the SHAP prediction explainer for the model:

```
mysql> CALL sys.ML_EXPLAIN('ml_data.iris_train', 'class', @iris_model,
          JSON_OBJECT('columns_to_explain', JSON_ARRAY('sepal width'),
          'target_value', 'Iris-setosa', 'model_explainer',
          'partial_dependence', 'prediction_explainer', 'shap'));
```

Where:

- `columns_to_explain` identifies the `sepal width` column for the explainer to explain how changing the value in this column affects the model. You can identify more than one column in the JSON array.
- `target_value` is a valid value that the target column containing ground truth values (in this case, `class`) can take.

For the full `ML_EXPLAIN` option descriptions, see Section 3.16.2, “ML_EXPLAIN”.

3.7 Predictions

Predictions are generated by running `ML_PREDICT_ROW` or `ML_PREDICT_TABLE` on unlabeled data; that is, it must have the same feature columns as the data used to train the model but no target column.

`ML_PREDICT_ROW` generates predictions for one or more rows of data. `ML_PREDICT_TABLE` generates predictions for an entire table of data and saves the results to an output table.

3.7.1 Row Predictions

`ML_PREDICT_ROW` generates predictions for one or more rows of data specified in `JSON` format. It is invoked using a `SELECT` statement. For `ML_PREDICT_ROW` parameter descriptions, see Section 3.16.5, “ML_PREDICT_ROW”.

Before running `ML_PREDICT_ROW`, ensure that the model you want to use is loaded; for example:

```
mysql> CALL sys.ML_MODEL_LOAD(@census_model, NULL);
```

For more information about loading models, see Section 3.14.3, “Loading Models”.

The following example runs `ML_PREDICT_ROW` on a single row of unlabeled data, which is assigned to a `@row_input` session variable:

```
mysql> SET @row_input = JSON_OBJECT(
          "age", 25,
          "workclass", "Private",
          "fnlwgt", 226802,
          "education", "11th",
          "education-num", 7,
          "marital-status", "Never-married",
          "occupation", "Machine-op-inspct",
          "relationship", "Own-child",
          "race", "Black",
          "sex", "Male",
          "capital-gain", 0,
          "capital-loss", 0,
          "hours-per-week", 40,
          "native-country", "United-States");

mysql> SELECT sys.ML_PREDICT_ROW(@row_input, @census_model, NULL);
```

where:

- `@row_input` is a session variable containing a row of unlabeled data. The data is specified in `JSON` key-value format. The column names must match the feature column names in the training dataset.
- `@census_model` is the session variable that contains the model handle.

ML_PREDICT_ROW returns a JSON object containing a Prediction key with the predicted value and the features values used to make the prediction.

You can also run ML_PREDICT_ROW on multiple rows of data selected from a table. For an example, refer to the syntax examples in Section 3.16.5, “ML_PREDICT_ROW”.

3.7.2 Table Predictions

ML_PREDICT_TABLE generates predictions for an entire table of unlabeled data and saves the results to an output table. Predictions are performed in parallel. For parameter and option descriptions, see Section 3.16.6, “ML_PREDICT_TABLE”.

ML_PREDICT_TABLE is a compute intensive process. Limiting operations to batches of 10 to 100 rows by splitting large tables into smaller tables is recommended. Use batch processing with the batch_size option. See: Section 3.15, “Progress tracking”.

Before running ML_PREDICT_TABLE, ensure that the model you want to use is loaded; for example:

```
mysql> CALL sys.ML_MODEL_LOAD(@census_model, NULL);
```

For more information about loading models, see Section 3.14.3, “Loading Models”.

The following example creates a table with 10 rows of unlabeled test data and generates predictions for that table:

```
mysql> CREATE TABLE heatwaveml_bench.census_test_subset AS SELECT *
          FROM heatwaveml_bench.census_test
          LIMIT 10;

mysql> CALL sys.ML_PREDICT_TABLE('heatwaveml_bench.census_test_subset',
          @census_model, 'heatwaveml_bench.census_predictions');
```

where:

- heatwaveml_bench.census_test_subset is the fully qualified name of the test dataset table (*schema_name.table_name*). The table must have the same feature column names as the training dataset but no target column.
- @census_model is the session variable that contains the model handle.
- heatwaveml_bench.census_predictions is the output table where predictions are stored. The table is created if it does not exist. A fully qualified table name must be specified (*schema_name.table_name*). If the table already exists, an error is returned.

To view ML_PREDICT_TABLE results, query the output table; for example:

```
mysql> SELECT * FROM heatwaveml_bench.census_predictions;
```

ML_PREDICT_TABLE populates the output table with predictions and the features used to make each prediction.

3.8 Explanations

Explanations are generated by running ML_EXPLAIN_ROW or ML_EXPLAIN_TABLE on unlabeled data; that is, it must have the same feature columns as the data used to train the model but no target column.

Explanations help you understand which features have the most influence on a prediction. Feature importance is presented as a value ranging from -1 to 1. A positive value indicates that a feature contributed toward the prediction. A negative value indicates that the feature contributed toward a different

prediction; for example, if a feature in a loan approval model with two possible predictions ('approve' and 'reject') has a negative value for an 'approve' prediction, that feature would have a positive value for a 'reject' prediction. A value of 0 or near 0 indicates that the feature value has no impact on the prediction to which it applies.

`ML_EXPLAIN_ROW` generates explanations for one or more rows of data. `ML_EXPLAIN_TABLE` generates explanations on an entire table of data and saves the results to an output table. `ML_EXPLAIN_*` routines limit explanations to the 100 most relevant features.

After the `ML_TRAIN` routine, use the `ML_EXPLAIN` routine to train prediction explainers and model explainers for HeatWave AutoML. You must train prediction explainers in order to use `ML_EXPLAIN_ROW` and `ML_EXPLAIN_TABLE`. In earlier releases, the `ML_TRAIN` routine trains the default Permutation Importance model and prediction explainers. See Section 3.6, “Training Explainers”.

3.8.1 Row Explanations

`ML_EXPLAIN_ROW` explains predictions for one or more rows of unlabeled data. It is invoked using a `SELECT` statement. For `ML_EXPLAIN_ROW` parameter descriptions, see Section 3.16.7, “ML_EXPLAIN_ROW”.

Before running `ML_EXPLAIN_ROW`, ensure that the model you want to use is loaded; for example:

```
mysql> CALL sys.ML_MODEL_LOAD(@census_model, NULL);
```

For more information about loading models, see Section 3.14.3, “Loading Models”.

The following example generates explanations for a single row of unlabeled data, which is assigned to a `@row_input` session variable:

```
mysql> SET @row_input = JSON_OBJECT(
          "age", 25,
          "workclass", "Private",
          "fnlwgt", 226802,
          "education", "11th",
          "education-num", 7,
          "marital-status", "Never-married",
          "occupation", "Machine-op-inspct",
          "relationship", "Own-child",
          "race", "Black",
          "sex", "Male",
          "capital-gain", 0,
          "capital-loss", 0,
          "hours-per-week", 40,
          "native-country", "United-States");

mysql> SELECT sys.ML_EXPLAIN_ROW(@row_input, @census_model,
          JSON_OBJECT('prediction_explainer', 'permutation_importance')));
```

where:

- `@row_input` is a session variable containing a row of unlabeled data. The data is specified in `JSON` key-value format. The column names must match the feature column names in the training dataset.
- `@census_model` is the session variable that contains the model handle.
- `prediction_explainer` provides the name of the prediction explainer that you have trained for this model, either the Permutation Importance prediction explainer or the SHAP prediction explainer. You train this using the `ML_EXPLAIN` routine (see Section 3.6, “Training Explainers”).

`ML_EXPLAIN_ROW` output includes a prediction, the features used to make the prediction, and a weighted numerical value that indicates feature importance, in the following format: `"feature_attribution":`

value. The output includes a Notes field that identifies features with the greatest impact on predictions and reports a warning if the model is low quality.

You can also run ML_EXPLAIN_ROW on multiple rows of data selected from a table. For an example, refer to the syntax examples in Section 3.16.7, “ML_EXPLAIN_ROW”.

3.8.2 Table Explanations

ML_EXPLAIN_TABLE explains predictions for an entire table of unlabeled data and saves results to an output table. Explanations are performed in parallel. For parameter and option descriptions, see Section 3.16.8, “ML_EXPLAIN_TABLE”.

ML_EXPLAIN_TABLE is a compute intensive process. Limiting operations to batches of 10 to 100 rows by splitting large tables into smaller tables is recommended. Use batch processing with the batch_size option. See: Section 3.15, “Progress tracking”.

The following example creates a table with 10 rows of data selected from the census_test dataset and generates explanations for that table.

Before running ML_EXPLAIN_TABLE, ensure that the model you want to use is loaded; for example:

```
mysql> CALL sys.ML_MODEL_LOAD(@census_model, NULL);
```

For more information about loading models, see Section 3.14.3, “Loading Models”.

The following example creates a table with 10 rows of unlabeled test data and generates explanations for that table:

```
mysql> CREATE TABLE heatwaveml_bench.census_test_subset AS SELECT *
          FROM heatwaveml_bench.census_test
          LIMIT 10;

mysql> CALL sys.ML_EXPLAIN_TABLE('heatwaveml_bench.census_test_subset',
          @census_model, 'heatwaveml_bench.census_explanations',
          JSON_OBJECT('prediction_explainer', 'shap'));
```

where:

- heatwaveml_bench.census_test_subset is the fully qualified name of the input table (*schema_name.table_name*). The table must have the same feature column names as the training dataset but no target column.
- @census_model is the session variable that contains the model handle.
- heatwaveml_bench.census_explanations is the table where explanation data is stored. The table is created if it does not exist. A fully qualified table name must be specified (*schema_name.table_name*). If the table already exists, an error is returned.
- prediction_explainer provides the name of the prediction explainer that you have trained for this model, either the Permutation Importance prediction explainer or the SHAP prediction explainer. You train this using the ML_EXPLAIN routine (see Section 3.6, “Training Explainers”).

To view ML_EXPLAIN_TABLE results, query the output table; for example:

```
mysql> SELECT * FROM heatwaveml_bench.census_explanations;
```

The ML_EXPLAIN_TABLE output table includes the features used to make the explanations, the explanations, and *feature*_attribution columns that provide a weighted numerical value that indicates feature importance.

ML_EXPLAIN_TABLE output also includes a Notes field that identifies features with the greatest impact on predictions. ML_EXPLAIN_TABLE reports a warning if the model is low quality.

3.9 Forecasting

To generate a forecast, run ML_TRAIN and specify a forecasting task as a JSON object literal. ML_PREDICT_TABLE can then predict the values for the selected column. Use ML_SCORE to score the model quality.

Create a timeseries forecast for a single column with a numeric data type. In forecasting terms, this is a univariate endogenous variable.

HeatWave AutoML supports multivariate endogenous forecasting models, and exogenous forecasting models.

3.9.1 Training a Forecasting Model

Run the ML_TRAIN routine to create a forecasting model, and use the following JSON *options*:

ML_TRAIN does not require target_column_name for forecasting, and it can be set to NULL.

- task: forecasting: Specifies the machine learning task.
- datetime_index: '*column*' The column name for a datetime column that acts as an index for the forecast variable. The column can be one of the supported datetime column types, DATETIME, TIMESTAMP, DATE, TIME, and YEAR, or an auto-incrementing index.
- endogenous_variables: JSON_ARRAY('*column*'[,'*column*'] ...) The column or columns to be forecast. One of these columns must also be specified as the target_column_name.
- exogenous_variables: JSON_ARRAY('*column*'[,'*column*'] ...) The column or columns of independent, non-forecast, predictive variables. For example, for sales forecasting these variables might be advertising expenditure, occurrence of promotional events, weather, or holidays.
- include_column_list: JSON_ARRAY('*column*'[,'*column*'] ...) include_column_list can include exogenous_variables.

See Section 3.5, “Training a Model”, and for full details of all the *options*, see ML_TRAIN.

Syntax Examples

- An ML_TRAIN example that specifies the forecasting task type and the additional required parameters datetime_index and endogenous_variables:

```
mysql> CALL sys.ML_TRAIN('ml_data.opsd_germany_daily_train', 'consumption',
          JSON_OBJECT('task', 'forecasting', 'datetime_index', 'ddate',
          'endogenous_variables', JSON_ARRAY('consumption')),
          @forecast_model);
```

3.9.2 Using a Forecasting Model

To produce a forecast, run the ML_PREDICT_TABLE routine on data with the same columns as the training model. datetime_index must be included. exogenous_variables must also be included, if used. Any extra columns, for example endogenous_variables, are ignored for the prediction, but included in the return table.

For instructions to use the ML_PREDICT_TABLE and ML_SCORE routines, see Section 3.7, “Predictions”, and Section 3.14.6, “Scoring Models”. For the complete list of option descriptions, see ML_PREDICT_TABLE and ML_SCORE.

`ML_SCORE` does not require `target_column_name` for forecasting, and it can be set to `NULL`.

`ML_PREDICT_ROW` cannot be used with forecasting models.

Syntax Examples

- A forecasting example with univariate `endogenous_variables`:

```
mysql> CALL sys.ML_TRAIN('mlcorpus.opsd_germany_daily_train', 'consumption',
       JSON_OBJECT('task', 'forecasting',
       'datetime_index', 'ddate',
       'endogenous_variables', JSON_ARRAY('consumption')),
       @forecast_model);
Query OK, 0 rows affected (11.51 sec)

mysql> CALL sys.ML_MODEL_LOAD(@forecast_model, NULL);
Query OK, 0 rows affected (1.07 sec)

mysql> CALL sys.ML_PREDICT_TABLE('mlcorpus.opsd_germany_daily_test',
          @forecast_model, 'mlcorpus.opsd_germany_daily_train_predictions1');
Query OK, 0 rows affected (1.50 sec)

mysql> SELECT * FROM opsd_germany_daily_train_predictions1 LIMIT 5;
+------------+---------+--------+------------+------------+
| ddate      | wind    | solar  | wind_solar | Prediction |
+------------+---------+--------+------------+------------+
| 2017-12-01 |  52.323 | 19.266 |     71.589 |    1528.13 |
| 2017-12-02 | 126.274 | 16.459 |    142.733 |    1333.16 |
| 2017-12-03 |  387.49 | 12.411 |    399.901 |     1276.1 |
| 2017-12-04 | 479.798 | 10.747 |    490.545 |    1602.18 |
| 2017-12-05 | 611.488 | 10.953 |    622.441 |    1615.38 |
+------------+---------+--------+------------+------------+
5 rows in set (0.01 sec)

mysql> CALL sys.ML_SCORE('mlcorpus.opsd_germany_daily_test', 'consumption',
          @forecast_model, 'neg_sym_mean_abs_percent_error', @score);
Query OK, 0 rows affected (1.40 sec)

mysql> SELECT @score;
+----------------------+
| @score               |
+----------------------+
| -0.07764234393835068 |
+----------------------+
1 row in set (0.00 sec)
```

- A forecasting example with univariate `endogenous_variables` and `exogenous_variables`:

```
mysql> CALL sys.ML_TRAIN('mlcorpus.opsd_germany_daily_train', 'consumption',
       JSON_OBJECT('task', 'forecasting',
       'datetime_index', 'ddate',
       'endogenous_variables', JSON_ARRAY('consumption'),
       'exogenous_variables', JSON_ARRAY('wind', 'solar', 'wind_solar')),
       @forecast_model);
Query OK, 0 rows affected (11.51 sec)

mysql> CALL sys.ML_MODEL_LOAD(@forecast_model, NULL);
Query OK, 0 rows affected (0.87 sec)

mysql> CALL sys.ML_PREDICT_TABLE('mlcorpus.opsd_germany_daily_test',
          @forecast_model, 'mlcorpus.opsd_germany_daily_train_predictions2', NULL);
Query OK, 0 rows affected (1.30 sec)

mysql> SELECT * FROM opsd_germany_daily_train_predictions2 LIMIT 5;
+-------------------+------------+--------------------+--------------------+--------------------+-----------
| _4aad19ca6e_pk_id | ddate      | consumption        | wind               | solar              | wind_solar
```

```
+------------------+------------+--------------------+--------------------+--------------------+-------
|                1 | 2015-12-30 | 1496.9310000000005 |  578.6919999999999 |             33.549 |
|                2 | 2015-12-31 |          1533.091 |  586.7679999999999 |             33.653 |
|                3 | 2016-01-01 | 1521.9320000000002 |            385.009 | 44.772999999999996 |  429.7
|                4 | 2016-01-02 |          1518.605 | 283.66299999999995 |              47.09 | 330.75
+------------------+------------+--------------------+--------------------+--------------------+-------
4 rows in set (0.00 sec)

mysql> CALL sys.ML_SCORE('mlcorpus.opsd_germany_daily_test', 'consumption',
          @forecast_model, 'neg_sym_mean_abs_percent_error', @score);
Query OK, 0 rows affected (1.11 sec)

mysql> SELECT @score;
+---------------------+
| @score              |
+---------------------+
| -0.06471854448318481 |
+---------------------+
1 row in set (0.00 sec)
```

- A forecasting example with multivariate `endogenous_variables` and `exogenous_variables`:

```
mysql> CALL sys.ML_TRAIN('mlcorpus.opsd_germany_daily_train', 'consumption',
          JSON_OBJECT('task', 'forecasting',
          'datetime_index', 'ddate',
          'endogenous_variables', JSON_ARRAY('consumption', 'wind'),
          'exogenous_variables', JSON_ARRAY('solar')),
          @forecast_model);
Query OK, 0 rows affected (27.84 sec)

mysql> CALL sys.ML_MODEL_LOAD(@forecast_model, NULL);
Query OK, 0 rows affected (0.92 sec)

mysql> CALL sys.ML_PREDICT_TABLE('mlcorpus.opsd_germany_daily_test',
          @forecast_model, 'mlcorpus.opsd_germany_daily_train_predictions3', NULL);
Query OK, 0 rows affected (2.79 sec)

mysql> SELECT * FROM opsd_germany_daily_train_predictions3 LIMIT 5;
+------------------+------------+--------------------+--------------------+--------------------+-------
| _4aad19ca6e_pk_id | ddate      | consumption        | wind               | solar              | wind_s
+------------------+------------+--------------------+--------------------+--------------------+-------
|                1 | 2015-12-30 | 1496.9310000000005 |  578.6919999999999 |             33.549 |
|                2 | 2015-12-31 |          1533.091 |  586.7679999999999 |             33.653 |
|                3 | 2016-01-01 | 1521.9320000000002 |            385.009 | 44.772999999999996 |  429.7
|                4 | 2016-01-02 |          1518.605 | 283.66299999999995 |              47.09 | 330.75
+------------------+------------+--------------------+--------------------+--------------------+-------
4 rows in set (0.00 sec)

mysql> CALL sys.ML_SCORE('mlcorpus.opsd_germany_daily_test', 'consumption',
          @forecast_model, 'neg_sym_mean_abs_percent_error', @score);
Query OK, 0 rows affected (2.62 sec)

mysql> SELECT @score;
+---------------------+
| @score              |
+---------------------+
| -0.43969136476516724 |
+---------------------+
1 row in set (0.00 sec)
```

- A forecasting example with multivariate `endogenous_variables`, `exogenous_variables` and `include_column_list`:

```
mysql> CALL sys.ML_TRAIN('mlcorpus.opsd_germany_daily_train', 'consumption',
          JSON_OBJECT('task', 'forecasting',
          'datetime_index', 'ddate',
          'endogenous_variables', JSON_ARRAY('consumption', 'wind'),
```

```
          'exogenous_variables', JSON_ARRAY('solar'),
          'include_column_list', JSON_ARRAY('solar')),
          @forecast_model);
Query OK, 0 rows affected (24.42 sec)

mysql> CALL sys.ML_MODEL_LOAD(@forecast_model, NULL);
Query OK, 0 rows affected (0.85 sec)

mysql> CALL sys.ML_PREDICT_TABLE('mlcorpus.opsd_germany_daily_test',
          @forecast_model, 'mlcorpus.opsd_germany_daily_train_predictions4', NULL);
Query OK, 0 rows affected (3.12 sec)

mysql> SELECT * FROM opsd_germany_daily_train_predictions4 LIMIT 5;
+------------------+------------+--------------------+--------------------+--------------------+-----------
| _4aad19ca6e_pk_id | ddate      | consumption        | wind               | solar              | wind_solar
+------------------+------------+--------------------+--------------------+--------------------+-----------
|                1 | 2015-12-30 | 1496.9310000000005 |  578.6919999999999 |             33.549 |
|                2 | 2015-12-31 |           1533.091 |  586.7679999999999 |             33.653 |
|                3 | 2016-01-01 | 1521.9320000000002 |            385.009 | 44.772999999999996 |  429.78199
|                4 | 2016-01-02 |           1518.605 | 283.66299999999995 |              47.09 | 330.753000
+------------------+------------+--------------------+--------------------+--------------------+-----------
4 rows in set (0.00 sec)

mysql> CALL sys.ML_SCORE('mlcorpus.opsd_germany_daily_test', 'consumption',
          @forecast_model, 'neg_sym_mean_abs_percent_error', @score);
Query OK, 0 rows affected (2.73 sec)

mysql> SELECT @score;
+---------------------+
| @score              |
+---------------------+
| -0.4276188611984253 |
+---------------------+
1 row in set (0.00 sec)
```

3.9.3 Prediction Intervals

MySQL 8.4.0 introduces prediction intervals for forecasting models, which generate upper and lower bounds on predictions for forecasting based on level of confidence. For example, for a prediction interval of 0.95 with a lower bound of 25 units and an upper bound of 65 units, you are 95% confident that product ABC will sell between 25 and 65 units on a randomly selected day.

When using prediction intervals, the `prediction_interval` option is included for the `ML_PREDICT_TABLE` routine, which specifies a level of confidence. Predictions provide three outputs corresponding to each endogenous variable: the forecasted value, a lower bound, and an upper bound. The `prediction_interval` option can be provided as a JSON_OBJECT.

For the `prediction_interval` option:

- The default value is 0.95.
- The data type for this value must be FLOAT.
- The value must be greater than 0 and less than 1.0.

3.9.3.1 Using Forecasting Models with Prediction Intervals

Before You Begin

- Connect to your HeatWave Database System
- Complete the steps to train the model with forecast modeling. See Section 3.9.1, “Training a Forecasting Model”

Using Prediction Intervals

To use a forecasting model with prediction intervals:

1. Use the `ML_MODEL_LOAD` routine to load the forecasting model:

```
mysql> CALL sys.ML_MODEL_LOAD(@forecast_model, NULL);
```

2. Use the `ML_PREDICT_TABLE` routine to generate forecasting predictions with prediction intervals:

```
mysql> CALL sys.ML_PREDICT_TABLE('schema_name.`input_table_name`', @forecast_model, 'schema_name.`outpu
JSON_OBJECT('prediction_interval', 0.95));
```

   Where:

   - `schema_name` is the database name that contains the table. Update this with the appropriate database.
   - `` `input_table_name` `` is the input table that contains the training dataset. Update this with the appropriate input table.
   - `@forecast_model` is the session variable that contains the model handle. Update this as needed.
   - `` `output_table_name` `` is the output table that will have the predictions. No existing table can have the same name.
   - `JSON_OBJECT('prediction_interval', 0.95)` includes the prediction interval option at 95% certainty.

For every endogenous variable included in the trained forecasting model, `prediction_interval_EndogVar` is added to the `ml_results` JSON. `EndogVar` is the endogenous variable name. The lower and upper bounds are also included.

See the following example:

```
mysql> select ml_results from schema_name.output_table_name limit 1;
+-------------------------------------------------------------------------------------------------------------
| ml_results
+-------------------------------------------------------------------------------------------------------------
| {"predictions": {"C1": 616.911, "C2": 456.851, "prediction_interval_C1": [250.507, 850.329], "prediction_
+-------------------------------------------------------------------------------------------------------------
```

Where:

- `C1` is the first endogenous variable, and `C2` is the second endogenous variable.
- The lower and upper bounds for `C1` are 250.507 and 850.329.
- The lower and upper bounds for `C2` are 150.461 and 750.164.

3.10 Anomaly Detection

HeatWave AutoML includes anomaly detection, which is also known as outlier detection. Anomaly detection is the data mining task that finds unusual patterns in data. It is particularly useful for the following applications:

- Financial fraud detection.
- Network intrusion detection for cyber security.
- Detecting life-threatening medical conditions.

For anomaly detection, HeatWave AutoML uses Generalized kth Nearest Neighbors, GkNN, which is a model developed at Oracle. It is a single ensemble algorithm that outperforms state-of-the-art models on public benchmarks. It can identify common anomaly types, such as local, global, and clustered anomalies, and can achieve an AUC score that is similar to, or better than, when identifying the following:

- Global anomalies compared to KNN, with an optimal k hyperparameter value.
- Local anomalies compared to LOF, with an optimal k hyperparameter value.
- Clustered anomalies.

Optimal k hyperparameter values would be extremely difficult to set without labels and knowledge of the use-case.

Other algorithms would require training and comparing scores from at least three algorithms to address global and local anomalies, ignoring clustered anomalies: LOF for local, KNN for global, and another generic method to establish a 2/3 voting mechanism.

MySQL 8.4.0 introduces support for two additional models:

- Principal Component Analysis (PCA)
- The internally developed Generalized Local Outlier Factor (GLOF)

3.10.1 Anomaly Detection Model Types

There are two types of anomaly detection types: unsupervised and semi-supervised.

Unsupervised Anomaly Detection

When running an unsupervised anomaly detection model the machine learning algorithm requires no labeled data. When training the model, the `target_column_name` parameter must be set to `NULL`.

Semi-supervised Anomaly Detection

MySQL 9.0.1-u1 introduces support for semi-supervised learning. This type of machine learning algorithm uses a specific set of labeled data along with unlabeled data to detect anomalies. To enable this, use the `experimental` and `semisupervised` options. The `target_column_name` parameter must specify a column whose only allowed values are 0, 1, and NULL. All rows will be used to train the unsupervised component, while the rows with a value different than NULL will be used to train the supervised component.

3.10.2 Training an Anomaly Detection Model

Run the `ML_TRAIN` routine to create an anomaly detection model.

If running an unsupervised model, the `target_column_name` parameter must be set to `NULL`.

If running a semi-supervised model, the `target_column_name` parameter must specify a column whose only allowed values are 0, 1, and NULL. All rows will be used to train the unsupervised component, while the rows with a value different than NULL will be used to train the supervised component.

Use the following `JSON` *`options`*:

- `task`: This must be set to `anomaly_detection`.
- `contamination`: A new option which represents an estimate of the percentage of outliers in the training table.

  - The contamination factor is calculated as: estimated number of rows with anomalies/total number of rows in the training table.
  - The contamination value must be greater than 0 and less than 0.5. The default value is 0.01.
- As of MySQL 8.4.0, `model_list` is supported to allow the selection of the Principal Component Analysis (PCA) model and Generalized Local Outlier Factor (GLOF) model. If no option is specified, the default model is Generalized kth Nearest Neighbors (GkNN). Selecting more than one model or an unsupported model produces an error.

MySQL 9.0.1-u1 introduces the following options to train a semi-supervised anomaly detection model:

- `experimental`: Must be set to `semisupervised` to enable semi-supervised training.
- `supervised_submodel_options`: Allows you to set optional override parameters for the supervised model component. The only model supported is `DistanceWeightedKNNClassifier`. The following parameters are supported:
  - `n_neighbors`: Sets the desired k value that checks the k closest neighbors for each unclassified point. The default value is 5 and the value must be an integer greater than 0.
  - `min_labels`: Sets the minimum number of labeled data points required to train the supervised component. If fewer labeled data points are provided during training of the model, `ML_TRAIN` fails. The default value is 20 and the value must be an integer greater than 0.
- `ensemble_score`: This option specifies the metric to use to score the ensemble of unsupervised and supervised components. It identifies the optimal weight between the two components based on the metric. The supported metrics are `accuracy`, `precision`, `recall`, and `f1`. The default metric is `f1`.

The following options are not supported for anomaly detection:

- `exclude_model_list`
- `optimization_metric`
- Before MySQL 8.4.0, `model_list` is not supported because the only supported algorithm model is Generalized kth Nearest Neighbors (GkNN). As of MySQL 8.4.0, `model_list` is supported to allow the selection of the Principal Component Analysis (PCA) model and Generalized Local Outlier Factor (GLOF) model.

See Section 3.5, “Training a Model”, and for full details of all the *`options`*, see `ML_TRAIN`.

Syntax Examples for Unsupervised Learning

- An `ML_TRAIN` example that specifies the `anomaly_detection` task type:

```
mysql> CALL sys.ML_TRAIN('mlcorpus_anomaly_detection.volcanoes-b3_anomaly_train',
          NULL, JSON_OBJECT('task', 'anomaly_detection',
          'exclude_column_list', JSON_ARRAY('target')),
          @anomaly);
Query OK, 0 rows affected (46.59 sec)
```

- An `ML_TRAIN` example that specifies the `anomaly_detection` task with a `contamination` option. Access the model catalog metadata to check the value of the `contamination` option.

```
mysql> CALL sys.ML_TRAIN('mlcorpus_anomaly_detection.volcanoes-b3_anomaly_train',
          NULL, JSON_OBJECT('task', 'anomaly_detection', 'contamination', 0.013,
          'exclude_column_list', JSON_ARRAY('target')),
          @anomaly_with_contamination);
Query OK, 0 rows affected (50.22 sec)
```

```
mysql> SELECT JSON_EXTRACT(model_metadata, '$.contamination')
          FROM ML_SCHEMA_root.MODEL_CATALOG
          WHERE model_handle = @anomaly_with_contamination;
+-----------------------------------------------+
| JSON_EXTRACT(model_metadata, '$.contamination') |
+-----------------------------------------------+
| 0.013000000268220901                          |
+-----------------------------------------------+
1 row in set (0.00 sec)
```

Syntax Examples for Semi-Supervised Learning

- An `ML_TRAIN` example that enables semi-supervised learning using all defaults. The *`target_column_name`* is set to `target`. The `experimental` option is set to `semisupervised`.

```
mysql> CALL sys.ML_TRAIN('mlcorpus.anomaly_train_with_partial_target', "target",
CAST('{"task": "anomaly_detection", "experimental": {"semisupervised": {}}}'
as JSON), @semisupervised_model);
```

- An `ML_TRAIN` example that enables semi-supervised learning with additional options.

```
mysql> CALL sys.ML_TRAIN('mlcorpus.`anomaly_train_with_partial_target`',
 "target", CAST('{"task": "anomaly_detection", "experimental": {"semisupervised":
 {"supervised_submodel_options": {""min_labels": 10, "n_neighbors": 3},
 "ensemble_score": "recall"}}}' as JSON), @semisupervised_model_options);
```

  Where:

  - The `supervised_submodel_options` parameter `min_labels` is set to 10.
  - The `supervised_submodel_options` parameter `n_neighbors` is set to 3.
  - The `ensemble_score` option is set to the `recall` metric.

Syntax Examples for Model Selection

- An `ML_TRAIN` example that selects the PCA algorithm model.

```
mysql> CALL sys.ML_TRAIN('mlcorpus_anomaly_detection_v1.`
volcanoes-b3_anomaly_train`', NULL,
JSON_OBJECT('task', 'anomaly_detection', 'exclude_column_list',
JSON_ARRAY('target'), 'model_list', JSON_ARRAY('PCA')), @anomaly_pca);
```

- An `ML_TRAIN` example that selects the GLOF algorithm model.

```
mysql> CALL sys.ML_TRAIN('mlcorpus_anomaly_detection_v1.
`volcanoes-b3_anomaly_train`', NULL,
JSON_OBJECT('task', 'anomaly_detection', 'exclude_column_list',
JSON_ARRAY('target'), 'model_list', JSON_ARRAY('GLOF')), @anomaly_glof);
```

- An `ML_TRAIN` example that does not specify an algorithm model for the `model_list` option. If no model is specified, the default model GkNN is used.

```
mysql> CALL sys.ML_TRAIN('mlcorpus_anomaly_detection_v1.
`volcanoes-b3_anomaly_train`', NULL,
JSON_OBJECT('task', 'anomaly_detection', 'exclude_column_list',
JSON_ARRAY('target'), 'model_list', JSON_ARRAY()), @anomaly_empty_list);
```

3.10.3 Using an Anomaly Detection Model

Anomaly detection models produce anomaly scores, which indicate the probability that a row has an anomaly.

To detect anomalies, run the `ML_PREDICT_ROW` or `ML_PREDICT_TABLE` routines on data with the same columns as the training model.

For `ML_SCORE` the `target_column_name` column must only contain the anomaly scores as an integer: `1`: an anomaly or `0` normal.

Anomaly Detection Model Options

Anomaly detection uses a threshold to convert anomaly scores to: an anomaly, which is set to `1`, or normal, which is set to `0`. There are two methods to set the threshold:

- A threshold value derived from the `ML_TRAIN` `contamination` option.

  Threshold = (1 - `contamination`)-th percentile of all the anomaly scores.

  The default `contamination` value is 0.01. The default `threshold` value based on the default `contamination` value is the 0.99-th percentile of all the anomaly scores.

- Set the threshold to a specific value.

  The `ML_PREDICT_TABLE`, `ML_PREDICT_ROW`, and `ML_SCORE` routines include a `threshold` option: 0 < `threshold` < 1.

The following additional options are available:

- An alternative to `threshold` is `topk`. The results include the top K rows with the highest anomaly scores. The `ML_PREDICT_TABLE` and `ML_SCORE` routines include the `topk` option, which is an integer between 1 and the table length.
- `ML_SCORE` includes an options parameter in `JSON` format. The options are `threshold` and `topk`.
- When running a semi-supervised model, the `ML_PREDICT_ROW`, `ML_PREDICT_TABLE`, and `ML_SCORE` routines have the `supervised_submodel_weight` option. It allows you to override the `ensemble_score` weighting estimated during `ML_TRAIN` with a new value. The value must be greater than 0 and less than 1.0.

Syntax Examples for Unsupervised Learning

- An anomaly detection example that uses the `roc_auc` metric for `ML_SCORE`.

```
mysql> CALL sys.ML_TRAIN('mlcorpus_anomaly_detection.volcanoes-b3_anomaly_train',
          NULL, JSON_OBJECT('task', 'anomaly_detection',
          'exclude_column_list', JSON_ARRAY('target')),
          @anomaly);
Query OK, 0 rows affected (46.59 sec)

mysql> CALL sys.ML_MODEL_LOAD(@anomaly, NULL);
Query OK, 0 rows affected (3.23 sec)

mysql> CALL sys.ML_PREDICT_TABLE('mlcorpus_anomaly_detection.volcanoes-b3_anomaly_train',
          @anomaly, 'mlcorpus_anomaly_detection.volcanoes-predictions',NULL);
Query OK, 0 rows affected (10.28 sec)

mysql> SELECT * FROM mlcorpus_anomaly_detection.volcanoes-predictions LIMIT 5;
+-------------------+------+------+----------+--------+-----------------------------------------------
| _4aad19ca6e_pk_id | V1   | V2   | V3       | target | ml_results
+-------------------+------+------+----------+--------+-----------------------------------------------
|                 1 |  128 |  802 |  0.47255 |      0 | {'predictions': {'is_anomaly': 0}, 'probabilitie
|                 2 |  631 |  642 | 0.387302 |      0 | {'predictions': {'is_anomaly': 0}, 'probabilitie
|                 3 |  438 |  959 | 0.556034 |      0 | {'predictions': {'is_anomaly': 0}, 'probabilitie
|                 4 |  473 |  779 | 0.407626 |      0 | {'predictions': {'is_anomaly': 0}, 'probabilitie
```

```
|                 5 |   67 |  933 | 0.383843 |      0 | {'predictions': {'is_anomaly': 0}, 'probabilities':
+-------+------+------+----------+--------+-------------------------------------------------------------------------
5 rows in set (0.00 sec)

mysql> CALL sys.ML_SCORE('mlcorpus_anomaly_detection.volcanoes-b3_anomaly_train',
          'target', @anomaly, 'roc_auc', @score, NULL);
Query OK, 0 rows affected (5.84 sec)

mysql> SELECT @score;
+--------------------+
| @score             |
+--------------------+
| 0.7465642094612122 |
+--------------------+
1 row in set (0.00 sec)
```

- An `ML_PREDICT_ROW` example that uses default options.

```
mysql> SELECT sys.ML_PREDICT_ROW('{"V1": 438.0, "V2": 959.0, "V3": 0.556034}', @anomaly, NULL);
+-------------------------------------------------------------------------------------------------------------------
| sys.ML_PREDICT_ROW('{"V1": 438.0, "V2": 959.0, "V3": 0.556034}', @anomaly, NULL)
+-------------------------------------------------------------------------------------------------------------------
| {"V1": 438.0, "V2": 959.0, "V3": 0.556034, "ml_results": "{'predictions': {'is_anomaly': 0}, 'probabilitie
+-------------------------------------------------------------------------------------------------------------------
1 row in set (5.35 sec)
```

- An `ML_PREDICT_TABLE` example that uses the `threshold` option set to 1%. All rows shown have probabilities of being an anomaly above 1%, and are predicted to be anomalies.

```
mysql> CALL sys.ML_PREDICT_TABLE('mlcorpus_anomaly_detection.volcanoes-b3_anomaly_train',
          @anomaly, 'mlcorpus_anomaly_detection.volcanoes-predictions_threshold',
          JSON_OBJECT('threshold', 0.01));
Query OK, 0 rows affected (12.77 sec)

mysql> SELECT * FROM mlcorpus_anomaly_detection.volcanoes-predictions_threshold LIMIT 5;
+------------------+------+------+----------+--------+-----------------------------------------------------
| _4aad19ca6e_pk_id | V1   | V2   | V3       | target | ml_results
+------------------+------+------+----------+--------+-----------------------------------------------------
|                1 |  128 |  802 |  0.47255 |      0 | {'predictions': {'is_anomaly': 1}, 'probabilities':
|                2 |  631 |  642 | 0.387302 |      0 | {'predictions': {'is_anomaly': 1}, 'probabilities':
|                3 |  438 |  959 | 0.556034 |      0 | {'predictions': {'is_anomaly': 1}, 'probabilities':
|                4 |  473 |  779 | 0.407626 |      0 | {'predictions': {'is_anomaly': 1}, 'probabilities':
|                5 |   67 |  933 | 0.383843 |      0 | {'predictions': {'is_anomaly': 1}, 'probabilities':
+------------------+------+------+----------+--------+-----------------------------------------------------
5 rows in set (0.00 sec)
```

- An `ML_SCORE` example that uses the `accuracy` metric with a `threshold` set to 90%.

```
mysql> CALL sys.ML_SCORE('mlcorpus_anomaly_detection.volcanoes-b3_anomaly_train',
          'target', @anomaly, 'accuracy', @score, JSON_OBJECT('threshold', 0.9));
Query OK, 0 rows affected (1.86 sec)

mysql> SELECT @score;
+--------------------+
| @score             |
+--------------------+
| 0.9791129231452942 |
+--------------------+
1 row in set (0.00 sec)
```

- An `ML_SCORE` example that uses the `precision_at_k` metric with a `topk` value of 10.

```
mysql> CALL sys.ML_SCORE('mlcorpus_anomaly_detection.volcanoes-b3_anomaly_train',
          'target', @anomaly, 'precision_at_k', @score, JSON_OBJECT('topk', 10));
Query OK, 0 rows affected (5.84 sec)
```

```
mysql> SELECT @score;
+--------------------+
| @score             |
+--------------------+
| 0                  |
+--------------------+
1 row in set (0.00 sec)
```

Syntax Examples for Semi-Supervised Learning

- An `ML_PREDICT_ROW` example that overrides the `ensemble_score` value from the `ML_TRAIN` routine to a new value of 0.8

```
mysql> SET @row_input = JSON_OBJECT('V1', 250, 'V2', 525, 'V3', 0.438976);
mysql> SELECT sys.ML_PREDICT_ROW(@row_input, @semsup_gknn,
CAST('{"experimental": {"semisupervised": {"supervised_submodel_weight": 0.8}}}' as JSON));
```

- An `ML_PREDICT_TABLE` example that overrides the `ensemble_score` value from the `ML_TRAIN` routine to a new value of 0.5.

```
mysql> CALL sys.ML_PREDICT_TABLE('mlcorpus.anomaly_train',
 @semsup_gknn, 'mlcorpus.preds_gknn_weighted',
CAST('{"experimental": {"semisupervised": {"supervised_submodel_weight": 0.5}}}' as JSON));
```

- An `ML_SCORE` example that overrides the `ensemble_score` value from the `ML_TRAIN` routine to a new value of 0.5.

```
mysql> CALL sys.ML_SCORE('mlcorpus.anomaly_train_with_target', "target",
@semsup_gknn, 'precision_at_k', @semsup_score_gknn_weighted,
CAST('{"topk": 10, "experimental":
{"semisupervised": {"supervised_submodel_weight": 0.5}}}' as JSON));
```

3.11 Recommendations

Recommendation models find patterns in user behavior to recommend new products based on prior behavior and preferences. Common examples include a streaming service recommending movies and shows based on past viewing history, or an online shopping site recommending products based on prior purchases.

The main goal of recommendation models is to recommend either items that a user will like, or recommend users who may like a specific item. HeatWave AutoML includes recommendation models that can recommend the following:

- The rating that a user will give to an item.
- Users who will like an item.
- Items that a user will like.
- Identify similar items.
- Identify similar users.

3.11.1 Recommendation Model Types

The following recommendation model types are available:

Recommendation Models with Explicit Feedback

Recommendation models that use explicit feedback collect data on users that directly provide ratings on items. The user ratings can be positive or negative. The recommendation models then use the feedback to

generate predicted ratings for users and items. The ratings are specific values, and the higher the value, the better the rating. See Recommendation Models to review models that support explicit feedback.

Recommendation Models with Implicit Feedback

Recommendation models that use implicit feedback collect data on users' behaivor, such as past purchases, clicks, and view times. Users do not have to explicitly express their taste about an item. When a user interacts with an item, the implication is that they prefer it to an item that they do not interact with. Therefore, only positive observations are available. The non-observed user-item interactions are a blend of negative feedback (the user doesn't like the item) or missing values (the user might be interested in the item). The recommendation model generates rankings for users and items. Rankings are a comparative measure, and the lower the value, the better the ranking. Because A is better than B, the ranking for A has a lower value than the ranking for B. HeatWave AutoML derives rankings based on ratings from implicit feedback for all ratings that are at or above the feedback threshold.

Implicit feedback data can be in the following formats:

- Unary data: Only records if an interaction occurred or not. This type of data often uses a value of 1 to represent an interaction, such as a click or view. Non-interactions can be represented by a value of 0 or missing values.
- Binary data: Explicitly categorizes interactions as positive or negative, such as users expressing likes or dislikes.
- Numerical data: Provides more granular information about the interaction, such as how long a user watched a video or how many times a user listened to a song. If numerical data is used for implicit feedback, it is important to set the `feedback_threshold` option during training to distinguish what constitutes positive feedback. This threshold determines what value is equivalent to a positive interaction. For example, if users are tracked by how many times they have interacted with an item, you might set the `feedback_threshold` with a value of 3, which means that positive feedback is represented by users that interact with the item more than three times.

Implicit feedback uses BPR: Bayesian Personalized Ranking from Implicit Feedback, which is a matrix factorization model that ranks user-item pairs. The recommendation models use this data to generate rankings for users and items.

Content-Based Recommendation Models

Content-based recommendation models allow you to include item descriptions in the input of the recommendation model. This helps the model provide more accurate representations of items. Currently, content-based recommendation models can only be used with implicit feedback. When training a content-based recommendation model, you can use the Collaborative Topic Regression model, which combines the ideas of matrix factorization models and topic modeling using Latent Dirichlet Allocation (LDA).

3.11.2 Training a Recommendation Model

When training a recommendation model, there are two mandatory options: `users` and `items`.

For content-based recommendation models, the `item_metadata` option is also required, which specifies the table that has item descriptions. This table must only have two columns: one corresponding to the `item_id`, and the other with a `TEXT` data type (TINYTEXT, TEXT, MEDIUMTEXT, LONGTEXT) that has the description of the item.

Run the `ML_TRAIN` routine to create a recommendation model, and use the following `JSON` *`options`*:

- `task`: `recommendation`: Specifies the machine learning task.

- `users`: Specifies the column name corresponding to the user ids.
- `items`: Specifies the column name corresponding to the item ids.
- `feedback`: The type of feedback for a recommendation model, `explicit`, the default, or `implicit`.
- `feedback_threshold`: The feedback threshold for a recommendation model that uses implicit feedback. It represents the threshold required to be considered positive feedback. For example, if numerical data records the number of times users interact with an item, you might set a threshold with a value of 3. This means users would need to interact with an item more than three times to be considered positive feedback.
- `item_metadata`: To be used with content-based recommendation models using implicit feedback. It is a JSON object that can have the `table_name` option as a key, which specifies the table that has item descriptions.
  - `table_name`: To be used with the `item_metadata` option. It specifies the table name that has item descriptions for content-based recommendation models. It must be a string in a fully qualified format (schema_name.table_name) that specifies the table name.

If the `users` or `items` column contains `NULL` values, the corresponding rows will be dropped and will not be considered during training.

See Section 3.5, “Training a Model”, and for full details of all the *`options`*, see `ML_TRAIN`.

Before You Begin

Review Section 3.2, “HeatWave AutoML Prerequisites”.

Syntax Examples for Explicit Feedback

- An `ML_TRAIN` example that specifies the `recommendation` task type.

```
mysql> CALL sys.ML_TRAIN('table_train', 'target_column_feature',
          JSON_OBJECT('task', 'recommendation',
          'users', 'user_column_feature',
          'items', 'item_column_feature'),
          @model);
```

- An `ML_TRAIN` example that specifies the `SVD` model type.

```
mysql> CALL sys.ML_TRAIN('mlcorpus.foursquare_NYC_train', 'rating',
          JSON_OBJECT('task', 'recommendation',
          'users', 'user_id',
          'items', 'item_id',
          'model_list', JSON_ARRAY('SVD')),
          @model);
Query OK, 0 rows affected (11.31 sec)

mysql> SELECT model_type FROM ML_SCHEMA_root.MODEL_CATALOG
          WHERE model_handle=@model;
+------------+
| model_type |
+------------+
| SVD        |
+------------+
1 row in set (0.00 sec)
```

- An `ML_TRAIN` example that specifies the `SVDpp` model type.

```
mysql> CALL sys.ML_TRAIN('mlcorpus.foursquare_NYC_train', 'rating',
          JSON_OBJECT('task', 'recommendation',
          'users', 'user_id',
          'items', 'item_id',
          'model_list',  JSON_ARRAY('SVDpp')),
          @model);
Query OK, 0 rows affected (13.97 sec)

mysql> SELECT model_type FROM ML_SCHEMA_root.MODEL_CATALOG
          WHERE model_handle=@model;
+------------+
| model_type |
+------------+
| SVDpp      |
+------------+
1 row in set (0.00 sec)
```

- An `ML_TRAIN` example that specifies the `NMF` model type.

```
mysql> CALL sys.ML_TRAIN('mlcorpus.foursquare_NYC_train', 'rating',
          JSON_OBJECT('task', 'recommendation',
          'users', 'user_id',
          'items', 'item_id',
          'model_list', JSON_ARRAY('NMF')),
          @model);
Query OK, 0 rows affected (12.28 sec)

mysql> SELECT model_type FROM ML_SCHEMA_root.MODEL_CATALOG
          WHERE model_handle=@model;
+------------+
| model_type |
+------------+
| NMF        |
+------------+
1 row in set (0.00 sec)
```

- An `ML_TRAIN` example that specifies three models for the `model_list` option.

```
mysql> SET @allowed_models = JSON_ARRAY('SVD', 'SVDpp', 'NMF');

mysql> CALL sys.ML_TRAIN('mlcorpus.foursquare_NYC_train', 'rating',
          JSON_OBJECT('task', 'recommendation',
          'users', 'user_id',
          'items', 'item_id',
          'model_list', CAST(@allowed_models AS JSON)),
          @model);
Query OK, 0 rows affected (14.88 sec)

mysql> SELECT model_type FROM ML_SCHEMA_root.MODEL_CATALOG
          WHERE model_handle=@model;
+------------+
| model_type |
+------------+
| SVD        |
+------------+
1 row in set (0.00 sec)
```

- An `ML_TRAIN` example that specifies five models for the `exclude_model_list` option.

```
mysql> SET @exclude_models= JSON_ARRAY('NormalPredictor', 'Baseline', 'SlopeOne', 'CoClustering', 'SVD');

mysql> CALL sys.ML_TRAIN('mlcorpus.foursquare_NYC_train', 'rating',
          JSON_OBJECT('task', 'recommendation',
          'users', 'user_id',
          'items', 'item_id',
          'exclude_model_list', CAST(@exclude_models AS JSON)),
```

```
          @model);
Query OK, 0 rows affected (14.71 sec)

mysql> SELECT model_type FROM ML_SCHEMA_root.MODEL_CATALOG
          WHERE model_handle=@model;
+------------+
| model_type |
+------------+
| SVDpp      |
+------------+
1 row in set (0.00 sec)
```

Syntax Example for Implicit Feedback

- An `ML_TRAIN` example that specifies the `recommendation` task with implicit feedback.

```
mysql> SELECT * FROM training_table LIMIT 10;
+---------+---------+--------+
| user_id | item_id | rating |
+---------+---------+--------+
| 836     | 226     |      1 |
| 3951    | 14918   |      1 |
| 1048    | 2498    |      1 |
| 4312    | 2559    |      1 |
| 4882    | 12930   |      1 |
| 3226    | 8400    |      1 |
| 3455    | 5120    |      1 |
| 830     | 12537   |      1 |
| 4525    | 791     |      1 |
| 2303    | 14243   |      1 |
+---------+---------+--------+

mysql> CALL sys.ML_TRAIN('mlcorpus.training_table', 'rating',
          JSON_OBJECT('task', 'recommendation',
          'users', 'user_id',
          'items', 'item_id',
          'feedback', 'implicit'),
          @model);
Query OK, 0 rows affected (2 min 13.6415 sec)
```

Syntax Example for Content-Based Recommendation Model

- An `ML_TRAIN` example that trains a content-based recommendation model by specifying a table with item descriptions (`mlcorpus_recsys.\`citeulike_items_sample`). The optimization metric `hit_ratio_at_k` is used. The model must use implicit feedback.

```
mysql> CALL sys.ML_TRAIN('mlcorpus_recsys.`citeulike_train_sample`', 'rating',
 JSON_OBJECT('task', 'recommendation', 'model_list',
JSON_ARRAY('CTR'), 'users', 'user_id', 'items', 'item_id','feedback', 'implicit', 'optimization_metric',
'item_metadata', JSON_OBJECT('table_name', 'mlcorpus_recsys.`citeulike_items_sample`')),
@model);
```

3.11.3 Using a Recommendation Model

Once you train and load a recommendation model, you can start generating predictions (rows or tables) and scores for the model.

Generating Predictions and Scores

To generate predictions on the trained model, run the `ML_PREDICT_ROW` or `ML_PREDICT_TABLE` routine. Run the routines on the data with the same columns as the training model.

When generating predictions:

- A table with the same name as the output table for `ML_PREDICT_TABLE` must not already exist.
- `NULL` values for any row in the `users` or `items` columns will cause an error.

When generating scores for a recommendation model, run the `ML_SCORE` routine. You can use any of the recommendation metrics to score a recommendation model. You can use the `metric` parameter to specify a ratings metric for a recommendation model that uses explicit feedback, or a ranking metric to use with a recommendation model that uses implicit or explicit feedback. See: Recommendation Model Metrics.

For instructions on generating predictions and scores, see Section 3.7, “Predictions”, and Section 3.14.6, “Scoring Models”. For the complete list of option descriptions for predictions and scores, see `ML_PREDICT_ROW`, `ML_PREDICT_TABLE`, and `ML_SCORE`.

`ML_EXPLAIN`, `ML_EXPLAIN_ROW` and `ML_EXPLAIN_TABLE` do not support recommendation models. A call to any of these routines with a recommendation model will produce an error.

Options for Generating Predictions and Scores

The *`options`* for `ML_PREDICT_ROW` and `ML_PREDICT_TABLE` include the following:

- `topk`: The number of recommendations to provide. The default is `3`.
- `recommend`: Specifies what to recommend. Permitted values are:
  - `ratings`: Predicts ratings that users will give. This is the default value.
  - `items`: Recommends items for users.
  - `users`: Recommends users for items.
  - `users_to_items`: This is the same as `items`.
  - `items_to_users`: This is the same as `users`.
  - `items_to_items`: Recommends similar items for items.
  - `users_to_users`: Recommends similar users for users.
- `remove_seen`: If `true`, the model will not repeat existing interactions from the training table. It only applies to the recommendations `items`, `users`, `users_to_items`, and `items_to_users`.

The *`options`* for `ML_SCORE` include the following:

- `threshold`: The optional threshold that defines positive feedback, and a relevant sample. Only use with ranking metrics. It can be used for either explicit or implicit feedback.
- `topk`: The optional top K rows to recommend. Only use with ranking metrics.
- `remove_seen`: If `true`, the model will not repeat existing interactions from the training table.

Output Values

Recommendation models can recommend the following for explicit and implicit feedback:

- The rating or ranking that a user will give to an item.

  - For known users and known items, the output includes the predicted rating or ranking a user will give for an item for a given pair of `user_id` and `item_id`.
  - For a known user with a new item, the prediction is the global average rating or ranking. The routines can add a user bias if the model includes it.
  - For a new user with a known item, the prediction is the global average rating or ranking. The routines can add an item bias if the model includes it.
  - For a new user with a new item, the prediction is the global average rating or ranking.
- Users that will like an item.
  - For known users and known items, the output includes a list of users that will most likely give a high rating to an item and will also predict the ratings.
  - For a new item, and an explicit feedback model, the prediction is the global top K users who have provided the average highest ratings.

    For a new item, and an implicit feedback model, the prediction is the global top K users with the highest number of interactions.
  - For an item that has been tried by all known users, the prediction is an empty list because it is not possible to recommend any other users. Set `remove_seen` to `false` to repeat existing interactions from the training table.
- Items that a user will like.
  - For known users and known items, the output includes a list of items that the user will most likely give a high rating and the predicted rating.
  - For a new user, and an explicit feedback model, the prediction is the global top K items that received the average highest ratings.

    For a new user, and an implicit feedback model, the prediction is the global top K items with the highest number of interactions.
  - For a user who has tried all known items, the prediction is an empty list because it is not possible to recommend any other items. Set `remove_seen` to `false` to repeat existing interactions from the training table.
- Items similar to another item.
  - For known items, the output includes a list of predicted items that have similar ratings and are appreciated by similar users.
  - The predictions are expressed in cosine similarity, and range from 0, very dissimilar, to 1, very similar.
  - For a new item, there is no information to provide a prediction. This will produce an error.
- Users similar to another user.
  - For known users, the output includes a list of predicted users that have similar behavior and taste.
  - The predictions are expressed in cosine similarity, and range from 0, very dissimilar, to 1, very similar.
  - For a new user, there is no information to provide a prediction. This will produce an error.

Before You Begin

1. Complete the steps for Section 3.11.2, “Training a Recommendation Model”
2. Once the model is trained, run the `ML_MODEL_LOAD` routine.

Syntax Examples for Explicit Feedback

- An `ML_PREDICT_TABLE` example that predicts the ratings for particular users and items. This is the default option for `recommend`, with *`options`* set to `NULL`.

```
mysql> CALL sys.ML_PREDICT_TABLE('mlcorpus.retailrocket-transactionto_to_predict',
          @model, 'mlcorpus.table_predictions', NULL);
Query OK, 0 rows affected (0.7589 sec)

mysql> SELECT * FROM  table_predictions;
+------------------+---------------+---------+---------+--------+----------------------------------+
| _4aad19ca6e_pk_id | timestamp     | user_id | item_id | rating | ml_results                       |
+------------------+---------------+---------+---------+--------+----------------------------------+
|                1 | 1436670000000 | 836347  | 64154   |      1 | {"predictions": {"rating": 1.0}}  |
|                2 | 1441250000000 | 435603  | 335366  |      1 | {"predictions": {"rating": 1.04}} |
|                3 | 1439670000000 | 1150086 | 314062  |      1 | {"predictions": {"rating": 1.03}} |
+------------------+---------------+---------+---------+--------+----------------------------------+
3 rows in set (0.00 sec)
```

  The output table displays the following recommendations:

  - User 836347 is predicted to give a rating of 1.0 for item 64154.
  - User 435603 is predicted to give a rating of 1.04 for item 335366.
  - User 1150086 is predicted to give a rating of 1.03 for item 314062.

  The values in the `rating` column refer to the past rating the `user_id` gave to the `item_id`. They are not relevant to the values in `ml_results`.

- A more complete example for the top 3 users that will like particular items.

```
mysql> SELECT * FROM train_table;
+---------+------------+--------+
| user_id | item_id    | rating |
+---------+------------+--------+
| user_1  | good_movie |      5 |
| user_1  | bad_movie  |      1 |
| user_2  | bad_movie  |      1 |
| user_3  | bad_movie  |      0 |
| user_4  | bad_movie  |      0 |
+---------+------------+--------+
5 rows in set (0.00 sec)

mysql> CALL sys.ML_TRAIN('mlcorpus.train_table', 'rating',
          JSON_OBJECT('task', 'recommendation', 'users', 'user_id', 'items', 'item_id'),
          @model);
Query OK, 0 rows affected (11.39 sec)

mysql> CALL sys.ML_MODEL_LOAD(@model, NULL);
Query OK, 0 rows affected (0.98 sec)

mysql> SELECT * FROM items_table;
+------------+
| item_id    |
+------------+
| good_movie |
| bad_movie  |
```

```
| new_movie  |
+------------+
3 rows in set (0.00 sec)

mysql> CALL sys.ML_PREDICT_TABLE('mlcorpus.items_table',
          @model, 'mlcorpus.user_recommendation',
          JSON_OBJECT("recommend", "users", "topk", 3));
Query OK, 0 rows affected (1.21 sec)

mysql> SELECT * FROM user_recommendation;
+-------------------+------------+-------------------------------------------------------------------------
| _4aad19ca6e_pk_id | item_id    | ml_results
+-------------------+------------+-------------------------------------------------------------------------
|                 1 | good_movie | {"predictions": {"user_id": ["user_2", "user_3", "user_4"], "rating":
|                 2 | bad_movie  | {"predictions": {"user_id": [], "rating": []}}
|                 3 | new_movie  | {"predictions": {"user_id": ["user_1", "user_2", "user_3"], "rating":
+-------------------+------------+-------------------------------------------------------------------------
3 rows in set (0.0004 sec)
```

The training table shows that users have given a rating with a scale of 1 to 5 for a `good_movie` and a `bad_movie`. There is an additional `new_movie` item in the `item_id` column. After running `PREDICT_TABLE`, the predicted ratings for the users are generated for `good_movie` and `new_movie`. There are no generated predictions for `bad_movie` because all the users have already rated the movie and the `remove_seen` option is set to the default value of `true`. To generate predictions for `bad_movie`, set `remove_seen` to `false`.

- An `ML_PREDICT_TABLE` example for the top 3 users that will like particular items with the `items_to_users` option.

```
mysql> CALL sys.ML_PREDICT_TABLE('mlcorpus.ml-100k',
          @model, 'mlcorpus.item_to_users_recommendation',
          JSON_OBJECT("recommend", "items_to_users", "topk", 3));
Query OK, 0 rows affected (21.2070 sec)

mysql> SELECT * FROM mlcorpus.item_to_users_recommendation LIMIT 5;
+-------------------+---------+---------+--------+-----------+-----------------------------------------
| _4aad19ca6e_pk_id | user_id | item_id | rating | timestamp | ml_results
+-------------------+---------+---------+--------+-----------+-----------------------------------------
|                 1 | 846     | 524     |      3 | 883948000 | {"predictions": {"user_id": ["7", "164",
|                 2 | 138     | 474     |      5 | 879024000 | {"predictions": {"user_id": ["7", "164",
|                 3 | 840     | 609     |      4 | 891205000 | {"predictions": {"user_id": ["7", "164",
|                 4 | 660     | 402     |      3 | 891201000 | {"predictions": {"user_id": ["7", "164",
|                 5 | 154     | 89      |      5 | 879139000 | {"predictions": {"user_id": ["7", "164",
+-------------------+---------+---------+--------+-----------+-----------------------------------------
```

The output table displays the following recommendations:

- Users 7, 164, and 894 are predicted to like items 524, 474, 609, and 402.
- Users 7, 164, and 151 are predicted to like item 89.
- The predicted ratings each user will give for the respective items display after the predicted users. For example, for item 524, user 7 is predicted to give a rating of 4.05, user 164 is predicted to give a rating of 3.94, and user 894 is predicted to give a rating of 3.91.

The values in the `rating` column refer to past ratings for respective user-item pairs. They are not relevant to the values in `ml_results`.

- An `ML_PREDICT_ROW` example for the top 3 items that a particular user will like.

```
mysql> SELECT sys.ML_PREDICT_ROW('{"user_id": "836347"}', @model,
          JSON_OBJECT("recommend", "items", "topk", 3));
+-----------------------------------------------------------------------------------------------------------
```

```
| sys.ML_PREDICT_ROW('{"user_id": "836347"}', @model,  JSON_OBJECT("recommend", "items", "topk", 3))
+-----------------------------------------------------------------------------------------------------
| {"user_id": "836347", "ml_results": "{"predictions": {"item_id": ["119736", "396042", "224549"], "rating":
+-----------------------------------------------------------------------------------------------------
1 row in set (0.31 sec)
```

User `836347` is predicted to like items `119736`, `396042`, and `224549`. The predicted ratings that the user will give for each item are also included after `rating`.

- An `ML_PREDICT_ROW` example for the top 3 items similar to another item.

```
mysql> SELECT sys.ML_PREDICT_ROW('{"item_id": "524"}', @model,
          JSON_OBJECT("recommend", "items_to_items", "topk", 3));
+-----------------------------------------------------------------------------------------------------
| sys.ML_PREDICT_ROW('{"item_id": "524"}', @model,  JSON_OBJECT("recommend", "items_to_items", "topk", 3))
+-----------------------------------------------------------------------------------------------------
| {"item_id": "524", "ml_results": "{"predictions": {"item_id": ["665", "633", "378"], "similarity": [1.0, 1
+-----------------------------------------------------------------------------------------------------
```

Item `524` is predicted to be most similar to items `665`, `633`, and `378`. The items have the highest similarity value of 1.0.

- An `ML_PREDICT_ROW` example for the top 3 users similar to another user.

```
mysql> SELECT sys.ML_PREDICT_ROW('{"user_id": "846"}', @model,
          JSON_OBJECT("recommend", "users_to_users", "topk", 3));
+-----------------------------------------------------------------------------------------------------
| sys.ML_PREDICT_ROW('{"user_id": "846"}', @model,  JSON_OBJECT("recommend", "users_to_users", "topk", 3))
+-----------------------------------------------------------------------------------------------------
| {"user_id": "846", "ml_results": "{"predictions": {"user_id": ["62", "643", "172"], "similarity": [0.7413,
+-----------------------------------------------------------------------------------------------------
1 row in set (0.2373 sec)
```

User `846` is predicted to be most similar to users `62`, `643`, and `172`. The respective similarity values for each user are included after `similarity`.

- An `ML_SCORE` example:

```
mysql> CALL sys.ML_SCORE('mlcorpus.ipinyou-click_test',
          'rating', @model, 'neg_mean_squared_error', @score, NULL);
Query OK, 0 rows affected (1 min 18.29 sec)

mysql> SELECT @score;
+----------------------+
| @score               |
+----------------------+
| -0.23571448028087616 |
+----------------------+
1 row in set (0.00 sec)
```

Syntax Examples for Implicit Feedback

- An `ML_PREDICT_ROW` example that predicts the ranking for a particular user and item.

```
mysql> SELECT sys.ML_PREDICT_ROW('{"user_id": "836", "item_id": "226"}', @model,  NULL);
+-----------------------------------------------------------------------------------+
| sys.ML_PREDICT_ROW('{"user_id": "836", "item_id": "226"}', @model,  NULL)         |
+-----------------------------------------------------------------------------------+
| {"item_id": "226", "user_id": "836", "ml_results": {"predictions": {"rating": 2.46}}} |
+-----------------------------------------------------------------------------------+
1 row in set (0.1390 sec)
```

The predicted ranking of item `226` and user `836` is 2.46.

- An `ML_PREDICT_TABLE` example that predicts the rankings for particular users and items.

```
mysql> CALL sys.ML_PREDICT_TABLE('mlcorpus.test_table', @model, 'mlcorpus.table_predictions', NULL);

mysql> SELECT * FROM mlcorpus.table_predictions LIMIT 10;
+------------------+---------+---------+--------+-----------------------------------+
| _4aad19ca6e_pk_id | user_id | item_id | rating | ml_results                        |
+------------------+---------+---------+--------+-----------------------------------+
|                1 | 1026    | 13763   |      1 | {"predictions": {"rating": 1.25}}  |
|                2 | 992     | 16114   |      1 | {"predictions": {"rating": -0.15}} |
|                3 | 1863    | 4527    |      1 | {"predictions": {"rating": 0.42}}  |
|                4 | 3725    | 3981    |      1 | {"predictions": {"rating": 3.11}}  |
|                5 | 3436    | 5854    |      1 | {"predictions": {"rating": 0.45}}  |
|                6 | 2236    | 13608   |      1 | {"predictions": {"rating": -0.35}} |
|                7 | 5230    | 1181    |      1 | {"predictions": {"rating": 1.96}}  |
|                8 | 1684    | 10140   |      1 | {"predictions": {"rating": -0.42}} |
|                9 | 3438    | 8022    |      1 | {"predictions": {"rating": -0.11}} |
|               10 | 1536    | 7578    |      1 | {"predictions": {"rating": 0.1}}   |
+------------------+---------+---------+--------+-----------------------------------+
10 rows in set (0.0004 sec)
```

  The predicted rankings are displayed in `ml_results`. The values in the `rating` column refer to past interactions for the user-item pair. They are not relevant to the values in `ml_results`.

- An `ML_PREDICT_ROW` example that recommends the top 3 users that will like a particular item and includes existing interactions from the training table.

```
mysql> SELECT sys.ML_PREDICT_ROW('{"item_id": "13763"}', @model,  JSON_OBJECT("recommend", "users", "top
+-----------------------------------------------------------------------------------------------------------
| sys.ML_PREDICT_ROW('{"item_id": "13763"}', @model,  JSON_OBJECT("recommend", "users", "topk", 3, "remo
+-----------------------------------------------------------------------------------------------------------
| {"item_id": "13763", "ml_results": {"predictions": {"rating": [1.26, 1.26, 1.26], "user_id": ["4590",
+-----------------------------------------------------------------------------------------------------------
1 row in set (0.3098 sec)
```

  Users `4590`, `1822`, and `3585` are predicted to like item `13763`. The respective ranking of each user is included after `rating`.

- An `ML_PREDICT_ROW` example that recommends the top 3 items that a particular user will like.

```
mysql> SELECT sys.ML_PREDICT_ROW('{"user_id": "1026"}', @model,  JSON_OBJECT("recommend", "items", "topk
+-----------------------------------------------------------------------------------------------------------
| sys.ML_PREDICT_ROW('{"user_id": "1026"}', @model,  JSON_OBJECT("recommend", "items", "topk", 3))
+-----------------------------------------------------------------------------------------------------------
| {"user_id": "1026", "ml_results": {"predictions": {"rating": [3.43, 3.37, 3.18], "item_id": ["10", "14
+-----------------------------------------------------------------------------------------------------------
1 row in set (0.6586 sec)
```

  User `1026` is predicted to like items `10`, `14`, and `11`. The respective ranking for each item is included after `rating`.

- An `ML_PREDICT_ROW` example for the top 3 items similar to another item.

```
mysql> SELECT sys.ML_PREDICT_ROW('{"item_id": "13763"}', @model,  JSON_OBJECT("recommend", "items_to_ite
+-----------------------------------------------------------------------------------------------------------
| sys.ML_PREDICT_ROW('{"item_id": "13763"}', @model,  JSON_OBJECT("recommend", "items_to_items", "topk",
+-----------------------------------------------------------------------------------------------------------
| {"item_id": "13763", "ml_results": {"predictions": {"item_id": ["13751", "13711", "13668"], "similarit
+-----------------------------------------------------------------------------------------------------------
1 row in set (0.4607 sec)
```

  Item `13763` is predicted to be most similar to items `13751`, `13711`, and `13668`. The respective similarity scores for each item is included after `similarity`.

- `ML_SCORE` examples for the four metrics suitable for a recommendation model with implicit feedback, with `threshold` set to `3`, `topk` set to `50` and including existing interactions from the training table with `remove_seen` set to `false`.

```
mysql> CALL sys.ML_MODEL_LOAD(@model, NULL);

mysql> SET @options = JSON_OBJECT('threshold', 3, 'topk', 50, 'remove_seen', false);
Query OK, 0 rows affected (0.00 sec)

mysql> CALL sys.ML_SCORE('mlcorpus.recsys_item_1', 'target', @model, 'precision_at_k', @score, @options);
Query OK, 0 rows affected (2.03 sec)

mysql> SELECT @score;
+--------------------+
| @score1            |
+--------------------+
| 0.03488215431571007 |
+--------------------+
1 row in set (0.00 sec)

mysql> CALL sys.ML_SCORE('mlcorpus.recsys_item_1', 'target', @model, 'recall_at_k', @score, @options);
Query OK, 0 rows affected (2.35 sec)

mysql> SELECT @score;
+--------------------+
| @score             |
+--------------------+
| 0.24304823577404022 |
+--------------------+
1 row in set (0.00 sec)

mysql> CALL sys.ML_SCORE('mlcorpus.recsys_item_1', 'target', @model, 'hit_ratio_at_k', @score, @options);
Query OK, 0 rows affected (2.30 sec)

mysql> SELECT @score;
+--------------------+
| @score             |
+--------------------+
| 0.18799902498722076 |
+--------------------+
1 row in set (0.00 sec)

mysql> CALL sys.ML_SCORE('mlcorpus.recsys_item_1', 'target', @model, 'ndcg_at_k', @score, @options);
Query OK, 0 rows affected (2.35 sec)

mysql> SELECT @score;
+--------------------+
| @score             |
+--------------------+
| 0.12175655364990234 |
+--------------------+
1 row in set (0.00 sec)
```

3.12 HeatWave AutoML and Lakehouse

HeatWave AutoML routines can load data directly from Object Storage with Lakehouse. Lakehouse must be enabled. See Chapter 5, *HeatWave Lakehouse*.

`ML_TRAIN`, `ML_MODEL_LOAD`, `ML_EXPLAIN` `ML_PREDICT_TABLE`, `ML_EXPLAIN_TABLE`, and `ML_SCORE` routines require no changes.

`ML_PREDICT_ROW` and `ML_EXPLAIN_ROW` routines cannot use a `FROM` clause.

Loading data from Lakehouse into HeatWave and unloading:

- If the Lakehouse table had not been loaded into HeatWave before a HeatWave AutoML command, then the data will be unloaded after the command.
- If the Lakehouse table had been loaded into HeatWave before a HeatWave AutoML command, then the data will remain in HeatWave after the command.

HeatWave AutoML commands operate on data loaded into HeatWave. If the original Lakehouse data in Object Storage is deleted or modified this will not affect a HeatWave AutoML command, until the data is unloaded from HeatWave.

Syntax Examples

The following examples use data from: Bank Marketing. The target column is `y`.

- A `CREATE TABLE` example with Lakehouse details that loads the training dataset.

```
mysql> CREATE TABLE bank_marketing_lakehouse_train(
        age int,
        job varchar(255),
        marital varchar(255),
        education varchar(255),
        default1 varchar(255),
        balance float,
        housing varchar(255),
        loan varchar(255),
        contact varchar(255),
        day int,
        month varchar(255),
        duration float,
        campaign int,
        pdays float,
        previous float,
        poutcome varchar(255),
        y varchar(255)
    )
 ENGINE=LAKEHOUSE
 SECONDARY_ENGINE=RAPID
  ENGINE_ATTRIBUTE='{"dialect": {"format": "csv",
                                 "skip_rows": 1,
                                 "field_delimiter":",",
                                 "record_delimiter":"\\n"}}',
                    "file": [{  "region": "region",
                                 "namespace": "namespace",
                                 "bucket": "bucket",
                                 "prefix": "mlbench/bank_marketing_train.csv"}]'
```

- An `ALTER TABLE` example with Lakehouse details that loads the test dataset.

```
mysql> CREATE TABLE bank_marketing_lakehouse_train(
        age int,
        job varchar(255),
        marital varchar(255),
        education varchar(255),
        default1 varchar(255),
        balance float,
        housing varchar(255),
        loan varchar(255),
        contact varchar(255),
        day int,
        month varchar(255),
        duration float,
        campaign int,
        pdays float,
        previous float,
        poutcome varchar(255),
```

```
        y varchar(255)
    );

mysql> ALTER TABLE bank_marketing_lakehouse_test
  ENGINE=LAKEHOUSE
  SECONDARY_ENGINE=RAPID
  ENGINE_ATTRIBUTE='{"dialect": {"format": "csv",
                                 "skip_rows": 1,
                                 "field_delimiter":",",
                                 "record_delimiter":"\\n"}}',
                   "file": [{  "region": "region",
                                 "namespace": "namespace",
                                 "bucket": "bucket",
                                 "prefix": "mlbench/bank_marketing_test.csv"}]';
```

- `ML_TRAIN`, `ML_MODEL_LOAD`, and `ML_SCORE` examples that use the Lakehouse data.

```
mysql> CALL sys.ML_TRAIN('ml_data.bank_marketing_lakehouse_train', 'y', NULL, @bank_model);

mysql> CALL sys.ML_MODEL_LOAD(@bank_model, NULL);

mysql> CALL sys.ML_SCORE('ml_data.bank_marketing_lakehouse_test', 'y', @bank_model,
       'balanced_accuracy', @score, NULL);
```

- `ML_PREDICT_TABLE`, `ML_EXPLAIN`, and `ML_EXPLAIN_TABLE` examples that use the Lakehouse data. `ML_EXPLAIN_TABLE` might take a long time for a large data set, but this is shape dependent.

```
mysql> CALL sys.ML_PREDICT_TABLE('ml_data.bank_marketing_lakehouse_test', @bank_model,
       'ml_data.bank_marketing_lakehouse_test_predictions', NULL);

mysql>  CALL sys.ML_EXPLAIN('ml_data.bank_marketing_lakehouse_test', 'y', @bank_model,
       JSON_OBJECT('prediction_explainer', 'permutation_importance'));

mysql> CALL sys.ML_EXPLAIN_TABLE('ml_data.bank_marketing_lakehouse_test', @bank_model,
       'ml_data.bank_marketing_lakehouse_test_explanations',
       JSON_OBJECT('prediction_explainer', 'permutation_importance'));
```

- Examples for `ML_PREDICT_ROW` and `ML_EXPLAIN_ROW` that insert data directly, and avoid the `FROM` clause.

```
mysql> SELECT sys.ML_PREDICT_ROW('{
       "age": 37,
       "job": "admin.",
       "marital": "married",
       "education": "unknown",
       "default1": "no",
       "balance": 734,
       "housing": "yes",
       "loan": "no",
       "contact": "unknown",
       "day": 21,
       "month": "may",
       "duration": 1106,
       "campaign": 1,
       "pdays": -1,
       "previous": 0,
       "poutcome":
       "unknown",
       "y": "no"}',
       @bank_model, NULL);

mysql> SELECT sys.ML_EXPLAIN_ROW('{
       "age": 37,
       "job": "admin.",
       "marital": "married",
       "education": "unknown",
```

```
        "default1": "no",
        "balance": 734,
        "housing": "yes",
        "loan": "no",
        "contact": "unknown",
        "day": 21,
        "month": "may",
        "duration": 1106,
        "campaign": 1,
        "pdays": -1,
        "previous": 0,
        "poutcome":
        "unknown",
        "y": "no"}',
        @bank_model,
        JSON_OBJECT('prediction_explainer', 'permutation_importance'));
```

- Examples for `ML_PREDICT_ROW` and `ML_EXPLAIN_ROW` that insert data directly with a `JSON` object, and avoid the `FROM` clause.

```
mysql> SET @row_input = JSON_OBJECT(
        'age', 37,
        'job', 'admin.',
        'marital', 'married',
        'education', 'unknown',
        'default1', 'no',
        'balance', 734,
        'housing', 'yes',
        'loan', 'no',
        'contact', 'unknown',
        'day', 21,
        'month', 'may',
        'duration', 1106,
        'campaign', 1,
        'pdays', -1,
        'previous', 0,
        'poutcome', 'unknown',
        'y', 'no');

mysql> SELECT sys.ML_PREDICT_ROW(@row_input, @bank_model, NULL);

mysql> SELECT sys.ML_EXPLAIN_ROW(@row_input, @bank_model,
        JSON_OBJECT('prediction_explainer', 'permutation_importance'));
```

- Examples for `ML_PREDICT_ROW` and `ML_EXPLAIN_ROW` that copies four rows to an InnoDB table, and then uses a `FROM` clause.

```
mysql> ALTER TABLE bank_marketing_lakehouse_test SECONDARY_LOAD;

mysql> CREATE TABLE bank_marketing_lakehouse_test_innoDB
        AS SELECT * from bank_marketing_lakehouse_test LIMIT 4;

mysql> SET @row_input = JSON_OBJECT(
        'age', bank_marketing_lakehouse_test_innoDB.age,
        'job', bank_marketing_lakehouse_test_innoDB.job,
        'marital', bank_marketing_lakehouse_test_innoDB.marital,
        'education', bank_marketing_lakehouse_test_innoDB.education,
        'default1', bank_marketing_lakehouse_test_innoDB.default1,
        'balance', bank_marketing_lakehouse_test_innoDB.balance,
        'housing', bank_marketing_lakehouse_test_innoDB.housing,
        'loan', bank_marketing_lakehouse_test_innoDB.loan,
        'contact', bank_marketing_lakehouse_test_innoDB.contact,
        'day', bank_marketing_lakehouse_test_innoDB.day,
        'month', bank_marketing_lakehouse_test_innoDB.month,
        'duration', bank_marketing_lakehouse_test_innoDB.duration,
        'campaign', bank_marketing_lakehouse_test_innoDB.campaign,
```

```
        'pdays', bank_marketing_lakehouse_test_innoDB.pdays,
        'previous', bank_marketing_lakehouse_test_innoDB.previous,
        'poutcome', bank_marketing_lakehouse_test_innoDB.poutcome);

mysql> SELECT sys.ML_PREDICT_ROW(@row_input, @bank_model, NULL);
        FROM bank_marketing_lakehouse_test_innoDB LIMIT 4;

mysql> SELECT sys.ML_EXPLAIN_ROW(@row_input, @bank_model,
        JSON_OBJECT('prediction_explainer', 'permutation_importance'))
        FROM bank_marketing_lakehouse_test_innoDB LIMIT 4;
```

3.13 Topic Modeling

MySQL 9.0.1-u1 introduces topic modeling, which is an unsupervised machine learning technique that's capable of scanning a set of documents, detecting word and phrase patterns within them, and automatically clustering word groups and similar expressions that best characterize the documents.

Topic modeling works with existing HeatWave AutoML routines.

> **Note**
>
> Topic modeling is not supported with the following HeatWave AutoML routines:
>
> - `ML_EXPLAIN`
> - `ML_EXPLAIN_TABLE`
> - `ML_EXPLAIN_ROW`
> - `ML_SCORE`

3.13.1 Training a Model with Topic Modeling

To add topic modeling when using the `ML_TRAIN` routine, you need to use the `document_column` parameter in the options argument as a key-value pair. This represents the name of the column that contains the text that topic modeling training will use to generate topics and tags as output. The output is an array of word groups that best characterize the text.

When HeatWave AutoML runs topic modeling, the operation is based on a single algorithm that does not require the tuning of hyperparameters. Moreover, topic modeling is an unsupervised task, which means there are no labels. Therefore, the following options are not supported for topic modeling:

- `model_list`
- `optimization_metric`
- `exclude_model_list`
- `exclude_column_list`
- `include_column_list`

The following example runs `ML_TRAIN` and includes the option to add topic modeling to the training:

```
mysql> CALL sys.ML_TRAIN('schema_name.table_name', NULL, JSON_OBJECT('task', 'topic_modeling',
 'document_column', 'column_name'), @topic_modeling);
```

Where:

- `schema_name` is the database name that contains the table. Update this with the appropriate database.

- `table_name` is the table name that contains the data to analyze. Update this with the appropriate table name.
- The target column argument is set to `NULL` because topic modeling is an unsupervised task and does not need labeled data to train the model.
- `JSON_OBJECT('task', 'topic_modeling', 'document_column', 'column_name')` specifies the machine learning task and text to train.
- The `task` must be set to `topic_modeling.`
- The `document_column` represents the name of the column that contains the text to train. Update `column_name` with the appropriate column name.
- `@topic_modeling` is the name of the user-defined session variable that stores the model handle for the duration of the connection. You can customize this name to your preference.

Once the model is trained, you can start using it for topic modeling in table and row predictions.

3.13.2 Table Predictions with Topic Modeling

This section describes how to generate predictions for a table of data with topic modeling.

Before You Begin

- Connect to your HeatWave Database System.
- Complete the steps to train the model with topic modeling. See Section 3.13.1, “Training a Model with Topic Modeling”.

Generating Predictions with Topic Modeling

To generate predictions on a table with topic modeling:

1. Use the `ML_MODEL_LOAD` routine to load the model with topic modeling:

```
mysql> CALL sys.ML_MODEL_LOAD(@topic_modeling, NULL);
```

2. Use the `ML_PREDICT_TABLE` routine to generate predictions with topic modeling:

```
mysql> CALL sys.ML_PREDICT_TABLE('schema_name.input_table_name', @topic_modeling,
'schema_name.output_table_name', NULL);
```

Where:

- `schema_name` is the database name that contains the table. Update this with the appropriate database.
- `` `input_table_name` `` is the input table that contains the training dataset. Update this with the appropriate input table.
- `@topic_modeling` is the session variable that contains the model handle. Update this as needed.
- `` `output_table_name` `` is the output table that will have the predictions. No existing table can have the same name.
- The target column argument is set to `NULL` because topic modeling is an unsupervised task and does not need labeled data to train the model.

Once the output table of predictions is generated:

- The output table will have the same columns as the input table with the added column of `ml_results`
- The `ml_results` JSON object literal contains the array of word groups that represent the trained text.
- Every row in `ml_results` has predictions and every prediction has the new key `tags` which has the generated word groups.

To modify the number of word groups in the ml_results column, you can set the `topk` option. This option must be an integer greater or equal to one.

The following example generates predictions on a table with topic modeling and uses the `topk` option to limit the number of word groups to five:

```
mysql> CALL sys.ML_PREDICT_TABLE('schema_name.input_table_name', @topic_modeling,
'schema_name.output_table_name',JSON_OBJECT('topk', 5));
```

3.13.3 Row Predictions with Topic Modeling

This section describes how to generate predictions on a row of data with topic modeling.

Before You Begin

- Connect to your HeatWave Database System.
- Complete the steps to train the model with topic modeling. See Training a Model with Topic Modeling.

Generating Predictions with Topic Modeling

To generate predictions on a row of data with topic modeling:

1. Use the `ML_MODEL_LOAD` routine to load the model with topic modeling:

   ```
   mysql> CALL sys.ML_MODEL_LOAD(@topic_modeling, NULL);
   ```

2. Set the row data to generate a prediction for a JSON object literal. The following example includes the row data from the `description` column:

   ```
   mysql> SET @row_input = JSON_OBJECT('description', "Presidential Jet|Taichung Top Ten
   Souvenirs|HWC Heiwo Coffee Product features: rich floral and lemon aroma, bright acid
   value; Layered, from sour to sweet, unique lemon fragrance. Product Description:
   Origin|Ethiopia Ethiopia Roast Level|Medium Light Processing Method|Washed Flavor
   DescriptionTastingNotes|Floralscent, lemon, citrus, tropical fruit, lemon, Tangerine,
   Tropical fruit Brewing method: It is recommended to brew and drink by hand Brewing
   recommended temperature: medium and shallow roasted Yekashefi 90-92°C Hand
   brewing thickness suggestion: a little thicker than No. 2 sugar, so that the coffee powder
   can be extracted completely Product ingredients: 100% Arabica coffee beans Weight: 1/2
   pound ");
   ```

   `@row_input` is a session variable containing a row of unlabeled data. The data is specified in `JSON` key-value format. The column names must match the feature column names in the training dataset.

3. Use the `ML_PREDICT_ROW` routine to generate the predictions with topic modeling:

   ```
   mysql> select sys.ML_PREDICT_ROW(@row_input, @topic_modeling, NULL);
   ```

   `@topic_modeling` is the session variable that contains the model handle. Update this name as needed.

To modify the number of word groups in the ml_results column, you can set the `topk` option. This option must be an integer greater or equal to one.

The following example generates predictions on the row of data with topic modeling and uses the `topk` option to limit the number of word groups to ten:

```
mysql> select sys.ML_PREDICT_ROW(@row_input, @topic_modeling, JSON_OBJECT('topk',10));
```

3.14 Managing Models

3.14.1 The Model Catalog

HeatWave AutoML stores machine learning models in a model catalog in HeatWave MySQL. A model catalog is an InnoDB table named `MODEL_CATALOG`. HeatWave AutoML creates a model catalog for any user that creates a machine learning model.

The `MODEL_CATALOG` table is created in a schema named `ML_SCHEMA_`*`user_name`*, where the *`user_name`* is the name of the owning user.

When a user creates a model, the `ML_TRAIN` routine creates the model catalog schema and table if they do not exist. `ML_TRAIN` inserts the model as a row in the `MODEL_CATALOG` table at the end of training.

A model catalog is accessible only to the owning user unless the user grants privileges on the model catalog to another user. This means that HeatWave AutoML routines can only use models that are accessible to the user running the routines. For information about granting model catalog privileges, see Section 3.14.10, “Sharing Models”.

A database administrator can manage a model catalog table as they would a regular MySQL table.

As of MySQL 9.0.0, HeatWave AutoML can support large models that are only limited by the amount of memory defined by the shape. The changes to the model catalog and model metadata are as follows:

- Models are not stored in the `model_object` column in the `MODEL_CATALOG` table. This column is set to `NULL`.
- Models are chunked and stored uncompressed in a new `model_object_catalog` table. Each chunk is saved with the same `model_handle`.
- `model_metadata` includes the number of chunks.

A call to one of the following routines will upgrade the model catalog, and store the model in the `model_object_catalog` table:

- `ML_TRAIN`
- `ML_MODEL_LOAD`
- `ML_EXPLAIN`
- `ML_MODEL_IMPORT`
- `ML_MODEL_EXPORT`

If the call to one of these routines is not successful or is aborted, then the previous model catalog will still be available.

As of MySQL 9.0.0, the model catalog can support these model formats:

- HWMLv1.0

A model trained by HeatWave AutoML and stored compressed.

- HWMLv2.0

  A model trained by HeatWave AutoML and stored uncompressed.

- ONNXv1.0

  An ONNX model verified by HeatWave AutoML and stored compressed.

- ONNXv2.0

  An ONNX model verified by HeatWave AutoML and stored uncompressed.

3.14.1.1 The Model Catalog Table

MySQL 8.1.0 deprecates several columns, and replaces them with additional `model_metadata`. `model_metadata` supports HeatWave AutoML and ONNX models.

The `MODEL_CATALOG` table has the following columns:

- `model_id`

  A primary key, and a unique auto-incrementing numeric identifier for the model.

- `model_handle`

  A name for the model. The model handle must be unique in the model catalog. The model handle is generated or set by the user when the `ML_TRAIN` routine is executed on a training dataset. The generated `model_handle` format is *`schemaName_tableName_userName_No`*, as in the following example: `heatwaveml_bench.census_train_user1_1636729526`.

  > **Note**
  >
  > The format of the generated model handle is subject to change.

- `model_object`

  A string in JSON format containing the serialized HeatWave AutoML model.

  As of MySQL 9.0.0, models are stored in the `model_object_catalog` table, and the `model_object` is set to `NULL`.

- `model_owner`

  The user who initiated the `ML_TRAIN` query to create the model.

- `build_timestamp`

  A timestamp indicating when the model was created (in UNIX epoch time). A model is created when the `ML_TRAIN` routine finishes executing.

  MySQL 8.1.0 deprecates `build_timestamp`, and replaces it with `build_timestamp` in `model_metadata`. A future release will remove it.

- `target_column_name`

  The name of the column in the training table that was specified as the target column.

MySQL 8.1.0 deprecates `target_column_name`, and replaces it with `target_column_name` in `model_metadata`. A future release will remove it.

- `train_table_name`

  The name of the input table specified in the `ML_TRAIN` query.

  MySQL 8.1.0 deprecates `train_table_name`, and replaces it with `train_table_name` in `model_metadata`. A future release will remove it.

- `model_object_size`

  The model object size, in bytes.

- `model_type`

  The type of model (algorithm) selected by `ML_TRAIN` to build the model.

  MySQL 8.1.0 deprecates `model_type`, because it is the same as `algorithm_name` in `model_metadata`. A future release will remove it.

- `task`

  The task type specified in the `ML_TRAIN` query.

  MySQL 8.1.0 deprecates `task`, and replaces it with `task` in `model_metadata`. A future release will remove it.

- `column_names`

  The feature columns used to train the model.

  MySQL 8.1.0 deprecates `column_names`, and replaces it with `column_names` in `model_metadata`. A future release will remove it.

- `model_explanation`

  The model explanation generated during training. See Section 3.14.7, “Model Explanations”.

  MySQL 8.1.0 deprecates `model_explanation`, and replaces it with `model_explanation` in `model_metadata`. A future release will remove it.

- `last_accessed`

  The last time the model was accessed. HeatWave AutoML routines update this value to the current timestamp when accessing the model.

  MySQL 8.1.0 deprecates `last_accessed`, because it is no longer used. A future release will remove it.

- `model_metadata`

  Metadata for the model. If an error occurs during training or you cancel the training operation, HeatWave AutoML records the error status in this column. See Section 3.14.1.3, “Model Metadata”

- `notes`

  Use this column to record notes about the trained model. It also records any error messages that occur during model training.

MySQL 8.1.0 deprecates `notes`, and replaces it with `notes` in `model_metadata`. A future release will remove it.

3.14.1.2 The Model Object Catalog Table

Introduced in MySQL 9.0.0 to support large models. The `model_object_catalog` table has the following columns:

- `chunk_id`

  A primary key, and an auto-incrementing numeric identifier for the chunk. `chunk_id` is unique for the chunks sharing the same `model_handle`.

- `model_handle`

  A primary key, and a foreign key that references `model_handle` in the `MODEL_CATALOG` table.

- `model_object`

  A string in JSON format containing the serialized HeatWave AutoML model.

3.14.1.3 Model Metadata

Metadata for the model. It is a column in the model catalog, see Section 3.14.1.1, “The Model Catalog Table”, and a parameter in `ML_MODEL_IMPORT`. The default value for `model_metadata` is `NULL`.

`model_metadata` has several fields that replace deprecated columns in the model catalog, and fields that support ONNX model import, see: Section 3.14.2, “ONNX Model Import”.

`model_metadata` contains the following metadata as key-value pairs in JSON format:

- `task: string`

  The task type specified in the `ML_TRAIN` query. The default is `classification` when used with `ML_MODEL_IMPORT`.

- `build_timestamp: number`

  A timestamp indicating when the model was created, in UNIX epoch time. A model is created when the `ML_TRAIN` routine finishes executing.

- `target_column_name: string`

  The name of the column in the training table that was specified as the target column.

- `train_table_name: string`

  The name of the input table specified in the `ML_TRAIN` query.

- `column_names: JSON array`

  The feature columns used to train the model.

- `model_explanation: JSON object literal`

  The model explanation generated during training. See Section 3.14.7, “Model Explanations”.

- `notes: string`

The `notes` specified in the `ML_TRAIN` query. It also records any error messages that occur during model training.

- `format: string`

  The model serialization format. `HWMLv1.0` for a HeatWave AutoML model or `ONNX` for a ONNX model. The default is `ONNX` when used with `ML_MODEL_IMPORT`.

- `status: string`

  The status of the model. The default is `Ready` when used with `ML_MODEL_IMPORT`.

  - `Creating`

    The model is still being created.

  - `Ready`

    The model is trained and active.

  - `Error`

    Either training was canceled or an error occurred during training. Any error message appears in the `notes` column. The error message also appears in `model_metadata notes`.

- `model_quality: string`

  The quality of the model object. Either `low` or `high`.

- `training_time: number`

  The time in seconds taken to train the model.

- `algorithm_name: string`

  The name of the chosen algorithm.

- `training_score: number`

  The cross-validation score achieved for the model by training.

- `n_rows: number`

  The number of rows in the training table.

- `n_columns: number`

  The number of columns in the training table.

- `n_selected_rows: number`

  The number of rows selected by adaptive sampling.

- `n_selected_columns: number`

  The number of columns selected by feature selection.

- `optimization_metric: string`

The optimization metric used for training.

- `selected_column_names: JSON array`

  The names of the columns selected by feature selection.

- `contamination: number`

  The contamination factor for a model.

- `options: JSON object literal`

  The `options` specified in the `ML_TRAIN` query.

- `training_params: JSON object literal`

  Internal task dependent parameters used during `ML_TRAIN`.

- `onnx_inputs_info: JSON object literal`

  Information about the format of the ONNX model inputs. This only applies to ONNX models. See Section 3.14.2, “ONNX Model Import”.

  Do not provide `onnx_inputs_info` if the model is not ONNX format. This will cause an error.

  - `data_types_map: JSON object literal`

    This maps the data type of each column to an ONNX model data type. The default value is:

    ```
    JSON_OBJECT("tensor(int64)": "int64", "tensor(float)": "float32", "tensor(string)": "str_")
    ```

- `onnx_outputs_info: JSON object literal`

  Information about the format of the ONNX model outputs. This only applies to ONNX models. See Section 3.14.2, “ONNX Model Import”.

  Do not provide `onnx_outputs_info` if the model is not ONNX format, or if `task` is `NULL`. This will cause an error.

  - `predictions_name: string`

    This name determines which of the ONNX model outputs is associated with predictions.

  - `prediction_probabilities_name: string`

    This name determines which of the ONNX model outputs is associated with prediction probabilities.

  - `labels_map: JSON object literal`

    This maps prediction probabilities to predictions, known as labels.

- `training_drift_metric: JSON object literal`

  Contains data drift information about the training data, see: Section 3.14.11, “Data Drift Detection”. This only applies to classification and regression models.

  - `mean: number`

    The mean value of drift metrics of all the training data. ≥ `0`.

- `variance: number`

  The variance value of drift metrics of all the training data. ≥ `0`.

  Both `mean` and `variance` should be low. To avoid divide by zero, the lowest value for both is `1e-10`.

- `chunks: number`

  The total number of chunks that the model has been split into. This was added in MySQL 9.0.0.

3.14.2 ONNX Model Import

HeatWave AutoML supports the upload of pre-trained models in ONNX, Open Neural Network Exchange, format to the model catalog. Load them with the `ML_MODEL_IMPORT` routine. After import, all the HeatWave AutoML routines can be used with ONNX models.

Models in ONNX format, `.onnx`, cannot be loaded directly into a MySQL table. They require string serialization and conversion to Base64 encoding before you use the `ML_MODEL_IMPORT` routine.

HeatWave AutoML supports these types of ONNX model:

- An ONNX model that has only one input and it is the entire MySQL table.
- An ONNX model that has more than one input and each input is one column in the MySQL table.

For example, HeatWave AutoML does not support an ONNX model that takes more than one input and each input is associated with more than one column in the MySQL table.

The first dimension of the input to the ONNX model provided by the ONNX model `get_inputs()` API should be the batch size. This should be `None`, a string, or an integer. `None` or string indicate a variable batch size and an integer indicates a fixed batch size. Examples of input shapes:

```
[None, 2]

['batch_size', 2, 3]

[1, 14]
```

All other dimensions should be integers. For example, HeatWave AutoML does not support an input shape similar to the following:

```
input shape = ['batch_size', 'sequence_length']
```

The output of an ONNX model is a list of results. The ONNX API documentation defines the results as a numpy array, a list, a dictionary or a sparse tensor. HeatWave AutoML only supports a numpy array, a list, and a dictionary.

- Numpy array examples:

```
array(['Iris-virginica', 'Iris-versicolor', 'Iris-versicolor', 'Iris-versicolor'], dtype=object)

array([0, 2, 0, 0], dtype=int64)

array([[0.8896357 , 0.11036429],
       [0.28360802, 0.716392  ],
       [0.9404001 , 0.05959991],
       [0.5655978 , 0.43440223]], dtype=float32)

array([[0.96875435],
       [1.081366  ],
```

```
       [0.5736201 ],
       [0.90711355]], dtype=float32)
```

- Simple list examples:

```
['Iris-virginica', 'Iris-versicolor', 'Iris-versicolor', 'Iris-versicolor']

[0, 2, 0, 0]
```

- List of lists examples:

```
[[0.8896357 , 0.110364],
[0.28360802, 0.716392],
[0.9404001 , 0.059599],
[0.5655978 , 0.434402]]

[[[0.8896357] , [0.110364]],
[[0.28360802], [0.716392]],
[[0.9404001] , [0.059599]],
[[0.5655978] , [0.434402]]]

[[0.968754],
[1.081366],
[0.573620],
[0.907113]]

[[[0.968754]],
[[1.081366]],
[[0.573620]],
[[0.907113]]]
```

- Dictionary examples:

```
{'Iris-setosa': 0.0, 'Iris-versicolor': 0.0, 'Iris-virginica': 0.999}

{0: 0.1, 1: 0.9}
```

- List of dictionaries examples:

```
[{'Iris-setosa': 0.0, 'Iris-versicolor': 0.0, 'Iris-virginica': 0.999},
{'Iris-setosa': 0.0, 'Iris-versicolor': 0.999, 'Iris-virginica': 0.0},
{'Iris-setosa': 0.0, 'Iris-versicolor': 0.589, 'Iris-virginica': 0.409},
{'Iris-setosa': 0.0, 'Iris-versicolor': 0.809, 'Iris-virginica': 0.190}]

[{0: 1.0, 1: 0.0, 2: 0.0},
{0: 0.0, 1: 0.0, 2: 1.0},
{0: 1.0, 1: 0.0, 2: 0.0},
{0: 1.0, 1: 0.0, 2: 0.0}]

[{0: 0.176, 1: 0.823},
{0: 0.176, 1: 0.823},
{0: 0.264, 1: 0.735},
{0: 0.875, 1: 0.124}]

[{0: 0.176, 1: 0.823},
{0: 0.176, 1: 0.823},
{0: 0.264, 1: 0.735},
{0: 0.875, 1: 0.124}]

[{0: 0.176, 1: 0.823},
{0: 0.176, 1: 0.823},
{0: 0.264, 1: 0.735},
{0: 0.875, 1: 0.124}]
```

For classification and regression tasks, HeatWave AutoML only supports model explainers and scoring for variable batch sizes.

For forecasting, anomaly detection and recommendation tasks, HeatWave AutoML does not support model explainers and scoring. The prediction column must contain a JSON object literal of name value keys. For example, for three outputs:

```
{output1: value1, output2: value2, output3: value3}
```

3.14.2.1 ONNX Model Metadata

For `model_metadata`, see: Section 3.14.1.3, “Model Metadata”. This includes `onnx_inputs_info` and `onnx_outputs_info`.

`onnx_inputs_info` includes `data_types_map`. See Section 3.14.1.3, “Model Metadata” for the default value.

`onnx_outputs_info` includes `predictions_name`, `prediction_probabilities_name`, and `labels_map`.

Use the `data_types_map` to map the data type of each column to an ONNX model data type. For example, to convert inputs of the type `tensor(float)` to `float64`:

```
data_types_map = {"tensor(float)": "float64"}
```

HeatWave AutoML first checks the user `data_types_map`, and then the default `data_types_map` to check if the data type exists. HeatWave AutoML supports the following numpy data types:

Table 3.1 Supported numpy data types

`str_`	`unicode_`	`int8`	`int16`	`int32`	`int64`	`int_`	`uint16`
`uint32`	`uint64`	`byte`	`ubyte`	`short`	`ushort`	`intc`	`uintc`
`uint`	`longlong`	`ulonglong`	`intp`	`uintp`	`float16`	`float32`	`float64`
`half`	`single`	`longfloat`	`double`	`longdouble`	`bool_`	`datetime64`	`complex_`
`complex64`	`complex128`	`complex256`	`csingle`	`cdouble`	`clongdouble`		

The use of any other numpy data type will cause an error.

Use `predictions_name` to determine which of the ONNX model outputs is associated with predictions. Use `prediction_probabilities_name` to determine which of the ONNX model outputs is associated with prediction probabilities. Use use a `labels_map` to map prediction probabilities to predictions, known as labels.

For regression tasks, if the ONNX model generates only one output, then `predictions_name` is optional. If the ONNX model generates more than one output, then `predictions_name` is required. Do not provide `prediction_probabilities_name` as this will cause an error.

For classification tasks use `predictions_name` or `prediction_probabilities_name` or both. Failure to provide at least one will cause an error. The model explainers SHAP, Fast SHAP and Partial Dependence require `prediction_probabilities_name`.

Only use a `labels_map` with classification tasks. A `labels_map` requires `predictions_probabilities_name`. The use of a `labels_map` with any other task, or with `predictions_name` or without `predictions_probabilities_name` will cause an error.

An example of a `predictions_probabilities_name` with a `labels_map` produces these labels:

```
predictions_probabilities_name = array([[0.35, 0.50, 0.15],
                                        [0.10, 0.20, 0.70],
                                        [0.90, 0.05, 0.05],
                                        [0.55, 0.05, 0.40]], dtype=float32)
```

```
labels_map = {0:'Iris-virginica', 1:'Iris-versicolor', 2:'Iris-setosa'}

labels=['Iris-versicolor', 'Iris-setosa', 'Iris-virginica', 'Iris-virginica']
```

Do not provide `predictions_name` or `prediction_probabilities_name` when the task is `NULL` as this will cause an error.

HeatWave AutoML adds a note for ONNX models that have inputs with four dimensions about the reshaping of data to a suitable shape for an ONNX model. This would typically be for ONNX models that are trained on image data. An example of this note added to the `ml_results` column:

```
mysql> CALL sys.ML_PREDICT_TABLE('mlcorpus_v5.mnist_test_temp', @model,
          'mlcorpus_v5.`mnist_predictions`', NULL);
Query OK, 0 rows affected (20.6296 sec)

mysql> SELECT ml_results FROM mnist_predictions;;
+-----------------------------------------------------------------------------------------------------------------
| ml_results
+-----------------------------------------------------------------------------------------------------------------
| {'predictions': {'prediction': 7}, 'Notes': 'Input data is reshaped into (1, 28, 28).', 'probabilities': {0:
+-----------------------------------------------------------------------------------------------------------------
```

3.14.2.2 How to Import ONNX Models

Follow these steps to import a model in ONNX format to the model catalog:

1. Convert the `.onnx` file containing the model to Base64 encoding and carry out string serialization. Do this with the Python base64 module. The following example converts the file `iris.onnx`:

   ```
   $> python -c "import onnx; import base64;
   $> open('iris_base64.onnx', 'wb').write(
   $> base64.b64encode(onnx.load('iris.onnx').SerializeToString()))"
   ```

2. Connect to the MySQL DB System for the HeatWave Cluster as a client, and create a temporary table to upload the model. For example:

   ```
   mysql> CREATE TEMPORARY TABLE onnx_temp (onnx_string LONGTEXT);
   ```

3. Use a `LOAD DATA INFILE` statement to load the preprocessed `.onnx` file into the temporary table. For example:

   ```
   mysql> LOAD DATA INFILE 'iris_base64.onnx'
             INTO TABLE onnx_temp
             CHARACTER SET binary
             FIELDS TERMINATED BY '\t'
             LINES TERMINATED BY '\r' (onnx_string);
   ```

4. Select the uploaded model from the temporary table into a session variable. For example:

   ```
   mysql> SELECT onnx_string FROM onnx_temp INTO @onnx_encode;
   ```

5. Call the `ML_MODEL_IMPORT` routine to import the ONNX model into the model catalog. For example:

   ```
   mysql> CALL sys.ML_MODEL_IMPORT(@onnx_encode, NULL, 'iris_onnx');
   ```

   In this example, the model handle is `iris_onnx`, and the optional model metadata is omitted and set to `NULL`. For details of the supported metadata for imported ONNX models, see `ML_MODEL_IMPORT` and Section 3.14.1.3, “Model Metadata”.

After import, all the HeatWave AutoML routines can be used with the ONNX model. It is added to the model catalog and can be managed in the same ways as a model created by HeatWave AutoML.

ONNX Import Examples

- A classification task example:

```
mysql> SET @model := 'sklearn_pipeline_classification_3_onnx';
Query OK, 0 rows affected (0.0003 sec)

mysql> SET @model_metadata := JSON_OBJECT('task','classification',
         'onnx_outputs_info',JSON_OBJECT('predictions_name','label','prediction_probabilities_name', 'p
Query OK, 0 rows affected (0.0003 sec)

mysql> CALL sys.ML_MODEL_IMPORT(@onnx_encode_sklearn_pipeline_classification_3, @
         model_metadata, @model);
Query OK, 0 rows affected (1.2438 sec)

mysql> CALL sys.ML_MODEL_LOAD(@model, NULL);
Query OK, 0 rows affected (0.5372 sec)

mysql> CALL sys.ML_PREDICT_TABLE('mlcorpus.classification_3_predict', @model,
         'mlcorpus.predictions', NULL);
Query OK, 0 rows affected (0.8743 sec)

mysql> SELECT * FROM mlcorpus.predictions;
+------------------+----+----+-----+------------+-----------------------------------------------------
| _4aad19ca6e_pk_id | f1 | f2 | f3  | Prediction | ml_results
+------------------+----+----+-----+------------+-----------------------------------------------------
|                1 | a  | 20 | 1.2 |          0 | {"predictions": {"prediction": 0}, "probabilities": {
|                2 | b  | 21 | 3.6 |          1 | {"predictions": {"prediction": 1}, "probabilities": {
|                3 | c  | 19 | 7.8 |          1 | {"predictions": {"prediction": 1}, "probabilities": {
|                4 | d  | 18 |   9 |          0 | {"predictions": {"prediction": 0}, "probabilities": {
|                5 | e  | 17 | 3.6 |          1 | {"predictions": {"prediction": 1}, "probabilities": {
+------------------+----+----+-----+------------+-----------------------------------------------------
5 rows in set (0.0005 sec)

mysql> CALL sys.ML_SCORE('mlcorpus.classification_3_table','target', @model,
         'accuracy', @score, NULL);
Query OK, 0 rows affected (0.9573 sec)

mysql> SELECT @score;
+--------+
| @score |
+--------+
|      1 |
+--------+
1 row in set (0.0003 sec)

mysql> CALL sys.ML_EXPLAIN('mlcorpus.classification_3_table', 'target', @model,
         JSON_OBJECT('model_explainer', 'shap', 'prediction_explainer', 'shap'));
Query OK, 0 rows affected (10.1771 sec)

mysql> SELECT model_explanation FROM ML_SCHEMA_root.MODEL_CATALOG
         WHERE model_handle=@model;
+-------------------------------------------------------+
| model_explanation                                     |
+-------------------------------------------------------+
| {"shap": {"f1": 0.0928, "f2": 0.0007, "f3": 0.0039}} |
+-------------------------------------------------------+
1 row in set (0.0005 sec)

mysql> CALL sys.ML_EXPLAIN_TABLE('mlcorpus.classification_3_predict', @model,
         'mlcorpus.explanations_shap', JSON_OBJECT('prediction_explainer', 'shap'));
Query OK, 0 rows affected (7.6577 sec)

mysql> SELECT * FROM mlcorpus.explanations_shap;
+------------------+----+----+-----+------------+----------------+----------------+----------------+--
| _4aad19ca6e_pk_id | f1 | f2 | f3  | Prediction | f1_attribution | f2_attribution  | f3_attribution | m
+------------------+----+----+-----+------------+----------------+----------------+----------------+--
```

```
|                 1 | a   | 20 | 1.2 |          0 |       0.116909 |     0.000591494 |    -0.00524929 | {"pre
|                 2 | b   | 21 | 3.6 |          1 |      0.0772133 |     -0.00110559 |     0.00219658 | {"pre
|                 3 | c   | 19 | 7.8 |          1 |      0.0781372 | 0.0000000913938 |    -0.00324671 | {"pre
|                 4 | d   | 18 |   9 |          0 |       0.115209 |    -0.000592354 |     0.00639341 | {"pre
|                 5 | e   | 17 | 3.6 |          1 |      0.0767679 |      0.00110463 |     0.00219425 | {"pre
+-------------------+-----+----+-----+------------+----------------+-----------------+----------------+------
5 rows in set (0.0005 sec)
```

- A regression task example:

```
mysql> SET @model := 'sklearn_pipeline_regression_2_onnx';
Query OK, 0 rows affected (0.0003 sec)

mysql> set @model_metadata := JSON_OBJECT('task','regression',
         'onnx_outputs_info',JSON_OBJECT('predictions_name','variable'));
Query OK, 0 rows affected (0.0003 sec)

mysql> CALL sys.ML_MODEL_IMPORT(@onnx_encode_sklearn_pipeline_regression_2,
         @model_metadata, @model);
Query OK, 0 rows affected (1.0652 sec)

mysql> CALL sys.ML_MODEL_LOAD(@model, NULL);
Query OK, 0 rows affected (0.5141 sec)

mysql> CALL sys.ML_PREDICT_TABLE('mlcorpus.regression_2_table',
         @model, 'mlcorpus.predictions', NULL);
Query OK, 0 rows affected (0.8902 sec)

mysql> SELECT * FROM mlcorpus.predictions;
+-------------------+----+----+-----+--------+------------+-------------------------------------------------
| _4aad19ca6e_pk_id | f1 | f2 | f3  | target | Prediction | ml_results
+-------------------+----+----+-----+--------+------------+-------------------------------------------------
|                 1 | a  | 20 | 1.2 |   22.4 |     22.262 | {"predictions": {"prediction": 22.26203918457031
|                 2 | b  | 21 | 3.6 |   32.9 |    32.4861 | {"predictions": {"prediction": 32.48611450195312
|                 3 | c  | 19 | 7.8 |   56.8 |    56.2482 | {"predictions": {"prediction": 56.24815368652344
|                 4 | d  | 18 |   9 |   31.8 |       31.8 | {"predictions": {"prediction": 31.80000114440918
|                 5 | e  | 17 | 3.6 |   56.4 |    55.9861 | {"predictions": {"prediction": 55.98611450195312
+-----+----+----+-----+--------+------------+-------------------------------------------------+
5 rows in set (0.0005 sec)

mysql> CALL sys.ML_SCORE('mlcorpus.regression_2_table','target', @model,
         'r2', @score, NULL);
Query OK, 0 rows affected (0.8688 sec)

mysql> SELECT @score;
+--------------------+
| @score             |
+--------------------+
| 0.9993192553520203 |
+--------------------+
1 row in set (0.0003 sec)

mysql> CALL sys.ML_EXPLAIN('mlcorpus.regression_2_table', 'target', @model,
         JSON_OBJECT('model_explainer', 'partial_dependence',
         'columns_to_explain', JSON_ARRAY('f1'), 'prediction_explainer', 'shap'));
Query OK, 0 rows affected (9.9860 sec)

mysql> SELECT model_explanation FROM ML_SCHEMA_root.MODEL_CATALOG
         WHERE model_handle=@model;
+-----------------------------------------------------------------------------------------------------------------
| model_explanation
+-----------------------------------------------------------------------------------------------------------------
| {"partial_dependence": {"f1": {"0": "a", "1": "b", "2": "c", "3": "d", "4": "e"}, "Mean": {"0": 28.9969997
+-----------------------------------------------------------------------------------------------------------------
1 row in set (0.0005 sec)

mysql> CALL sys.ML_EXPLAIN_TABLE('mlcorpus.regression_2_predict', @model,
```

```
          'mlcorpus.explanations', JSON_OBJECT('prediction_explainer', 'shap'));
Query OK, 0 rows affected (8.2625 sec)

mysql> SELECT * FROM mlcorpus.explanations;
+------------------+----+----+-----+------------+----------------+----------------+----------------+---
| _4aad19ca6e_pk_id | f1 | f2 | f3  | Prediction | f1_attribution | f2_attribution | f3_attribution | ml
+------------------+----+----+-----+------------+----------------+----------------+----------------+---
|                1 | a  | 20 | 1.2 |     22.262 |       -10.7595 |       -4.25162 |       -2.48331 | {"
|                2 | b  | 21 | 3.6 |    32.4861 |        2.33657 |       -8.50325 |        -1.1037 | {"
|                3 | c  | 19 | 7.8 |    56.2482 |        14.8361 |              0 |        1.65554 | {"
|                4 | d  | 18 |   9 |       31.8 |       -15.2433 |        4.25162 |        3.03516 | {"
|                5 | e  | 17 | 3.6 |    55.9861 |        8.83008 |        8.50325 |        -1.1037 | {"
+------------------+----+----+-----+------------+----------------+----------------+----------------+---
5 rows in set (0.0006 sec)
```

- An example with task set to `NULL`.

```
mysql> SET @model := 'tensorflow_recsys_onnx';

mysql> CALL sys.ML_MODEL_IMPORT(@onnx_encode_tensorflow_recsys, NULL, @model);
Query OK, 0 rows affected (1.0037 sec)

mysql> CALL sys.ML_MODEL_LOAD(@model, NULL);
Query OK, 0 rows affected (0.5116 sec)

mysql> CALL sys.ML_PREDICT_TABLE('mlcorpus.recsys_predict', @model,
          'mlcorpus.predictions', NULL);
Query OK, 0 rows affected (0.8271 sec)

mysql> SELECT * FROM mlcorpus.predictions;
+------------------+---------+-------------+--------------------------+-------------------------------
| _4aad19ca6e_pk_id | user_id | movie_title | Prediction               | ml_results
+------------------+---------+-------------+--------------------------+-------------------------------
|                1 | a       | A           | {"output_1": ["0.7558"]} | {"predictions": {"prediction":
|                2 | b       | B           | {"output_1": ["1.0443"]} | {"predictions": {"prediction":
|                3 | c       | A           | {"output_1": ["0.8483"]} | {"predictions": {"prediction":
|                4 | d       | B           | {"output_1": ["1.2986"]} | {"predictions": {"prediction":
|                5 | e       | C           | {"output_1": ["1.1568"]} | {"predictions": {"prediction":
+------------------+---------+-------------+--------------------------+-------------------------------
5 rows in set (0.0005 sec)
```

3.14.3 Loading Models

A model must be loaded from the model catalog into HeatWave before running HeatWave AutoML routines other than `ML_TRAIN`. A model remains loaded and can be called repetitively by HeatWave AutoML routines until it is unloaded using the `ML_MODEL_UNLOAD` routine or until the HeatWave Cluster is restarted.

A model can only be loaded by the MySQL user that created the model. For more information, see Section 3.14.10, “Sharing Models”.

HeatWave can load multiple models but to avoid taking up too much space in memory, limit the number of loaded models to three.

For `ML_MODEL_LOAD` parameter descriptions, see Section 3.16.10, “ML_MODEL_LOAD”.

The following example loads a HeatWave AutoML model from the model catalog:

```
mysql> CALL sys.ML_MODEL_LOAD(@census_model, NULL);
```

where:

- `@census_model` is the session variable that contains the model handle.

- NULL is specified in place of the user name of the model owner.

A fully qualified ML_MODEL_LOAD call that specifies the model handle and user name of the owner is as follows:

```
mysql> CALL sys.ML_MODEL_LOAD('heatwaveml_bench.census_train_user1_1636729526', user1);
```

To look up a model handle, see Section 3.14.8, “Model Handles”.

The user parameter is ignored, and it can be set to NULL.

3.14.4 Unloading Models

The ML_MODEL_UNLOAD routine unloads a model from HeatWave AutoML. For ML_MODEL_UNLOAD parameter descriptions, see Section 3.16.11, “ML_MODEL_UNLOAD”.

It is permitted to load multiple models but to avoid taking up too much space in memory, limit the number of loaded models to three.

The following example unloads a model:

```
mysql> CALL sys.ML_MODEL_UNLOAD('heatwaveml_bench.census_train_user1_1636729526');
```

where:

- heatwaveml_bench.census_train_*user1*_1636729526 is the model handle.

To look up a model handle, see Section 3.14.8, “Model Handles”.

3.14.5 Viewing Models

To view the models in your model catalog, query the MODEL_CATALOG table; for example:

```
mysql> SELECT model_id, model_handle, model_owner FROM ML_SCHEMA_user1.MODEL_CATALOG;
```

where:

- model_id is a unique numeric identifier for the model.
- model_owner is the user that created the model.
- model_handle is the handle by which the model is called.
- ML_SCHEMA_*user1*.MODEL_CATALOG is the fully qualified name of the MODEL_CATALOG table. The schema is named for the owning user.

> **Note**
>
> The example above retrieves data from only a few MODEL_CATALOG table columns. For other columns you can query, see Section 3.14.1, “The Model Catalog”.

3.14.6 Scoring Models

ML_SCORE scores a model by generating predictions using the feature columns in a labeled dataset as input and comparing the predictions to ground truth values in the target column of the labeled dataset.

The dataset used with ML_SCORE should have the same feature columns as the dataset used to train the model but the data sample should be different from the data used to train the model; for example, you might reserve 20 to 30 percent of a labeled dataset for scoring.

`ML_SCORE` returns a computed metric indicating the quality of the model. A value of `None` is reported if a score for the specified or default metric cannot be computed. If an invalid metric is specified, the following error message is reported: `Invalid data for the metric. Score could not be computed.`

Models with a low score can be expected to perform poorly, producing predictions and explanations that cannot be relied upon. A low score typically indicates that the provided feature columns are not a good predictor of the target values. In this case, consider adding more rows or more informative features to the training dataset.

You can also run `ML_SCORE` on the training dataset and a labeled test dataset and compare results to ensure that the test dataset is representative of the training dataset. A high score on a training dataset and low score on a test dataset indicates that the test data set is not representative of the training dataset. In this case, consider adding rows to the training dataset that better represent the test dataset.

HeatWave AutoML supports a variety of scoring metrics to help you understand how your model performs across a series of benchmarks. For `ML_SCORE` parameter descriptions and supported metrics, see Section 3.16.9, “ML_SCORE”.

Before running `ML_SCORE`, ensure that the model you want to use is loaded; for example:

```
mysql> CALL sys.ML_MODEL_LOAD(@census_model, NULL);
```

For information about loading models, see Section 3.14.3, “Loading Models”.

The following example runs `ML_SCORE` to compute model quality using the `balanced_accuracy` metric:

```
mysql> CALL sys.ML_SCORE('heatwaveml_bench.census_validate', 'revenue',
         @census_model, 'balanced_accuracy', @score);
```

where:

- `heatwaveml_bench.census_validate` is the fully qualified name of the validation dataset table (*`schema_name.table_name`*).
- `revenue` is the name of the target column containing ground truth values.
- `@census_model` is the session variable that contains the model handle.
- `balanced_accuracy` is the scoring metric. For other supported scoring metrics, see Section 3.16.9, “ML_SCORE”.
- `@score` is the user-defined session variable that stores the computed score. The `ML_SCORE` routine populates the variable. User variables are written as *`@var_name`*. The examples in this guide use `@score` as the variable name. Any valid name for a user-defined variable is permitted, for example `@my_score`.

To retrieve the computed score, query the `@score` session variable.

```
mysql> SELECT @score;
+--------------------+
| @score             |
+--------------------+
| 0.8188666105270386 |
+--------------------+
```

3.14.7 Model Explanations

Use the `ML_EXPLAIN` routine to train a model explainer for the model and produce a model explanation. The model explanation is stored in the `model_explanation` column in the `MODEL_CATALOG` table.

A model explanation helps you identify the features that are most important to the model overall. Feature importance is presented as a numerical value ranging from 0 to 1. Higher values signify higher feature importance, lower values signify lower feature importance, and a 0 value means that the feature does not influence the model.

The following example retrieves the model explanation for the census model:

```
mysql> SELECT model_explanation FROM ML_SCHEMA_user1.MODEL_CATALOG
          WHERE model_handle=@census_model;
```

where:

- `ML_SCHEMA_user1.MODEL_CATALOG` is the fully qualified name of the `MODEL_CATALOG` table. The schema is named for the user that created the model.
- `@census_model` is the session variable that contains the model handle.

3.14.8 Model Handles

When `ML_TRAIN` trains a model, it generates a model handle, which is required when running other HeatWave AutoML routines. You can replace the generated model handle with a model handle of your choice, which must be unique in the model catalog.

The model handle is stored temporarily in a user-defined session variable specified in the `ML_TRAIN` call. In the following example, `@census_model` is defined as the model handle session variable:

```
mysql> CALL sys.ML_TRAIN('heatwaveml_bench.census_train', 'revenue', NULL, @census_model);
```

To use your own model handle instead of a generated one, set the value of the session variable before calling the `ML_TRAIN` routine, like this:

```
mysql> SET @census_model = 'census_test';
mysql> CALL sys.ML_TRAIN('heatwaveml_bench.census_train', 'revenue', NULL, @census_model);
```

If you set a model handle that already appears in the model catalog, the `ML_TRAIN` routine returns an error.

While the connection used to run `ML_TRAIN` remains active, that connection can retrieve the model handle by querying the session variable; for example:

```
mysql> SELECT @census_model;
+-------------------------------------------------+
| @census_model                                   |
+-------------------------------------------------+
| heatwaveml_bench.census_train_user1_1636729526  |
+-------------------------------------------------+
```

> **Note**
>
> The format of the generated model handle is subject to change.

While the session variable remains populated with the model handle, it can be specified in place of the model handle when running other `ML_*` routines. However, once the connection is terminated, the session variable data is lost. In this case, you can look up the model handle by querying the model catalog table; for example:

```
mysql> SELECT model_handle, model_owner, train_table_name
          FROM ML_SCHEMA_user1.MODEL_CATALOG;
+-------------------------------------------------+-------------+-------------------------------+
| model_handle                                    | model_owner | train_table_name              |
+-------------------------------------------------+-------------+-------------------------------+
| heatwaveml_bench.census_train_user1_1636729526 |     user1   | heatwaveml_bench.census_train |
```

```
+-----------------------------------------------+------------+------------------------------+
```

You can specify the model handle in `ML_ROUTINE_*` calls directly; for example:

```
mysql> SELECT sys.ML_PREDICT_ROW(@row_input, 'heatwaveml_bench.census_train_user1_1636729526');
```

Alternatively, you can reassign a model handle to a session variable; for example:

- To assign a model handle to a session variable named `@my_model`:

  ```
  mysql> SET @my_model = 'heatwaveml_bench.census_train_user1_1636729526';
  ```

- To assign a model handle to a session variable named `@my_model` for the most recently trained model:

  ```
  mysql> SET @my_model = (SELECT model_handle FROM ML_SCHEMA_user1.MODEL_CATALOG
           ORDER BY model_id DESC LIMIT 1);
  ```

  The most recently trained model is the last model inserted into the `MODEL_CATALOG` table. It has the most recently assigned `model_id`, which is a unique auto-incrementing numeric identifier.

3.14.9 Deleting Models

A model can be deleted by the owning user or users that have been granted the required privileges on the `MODEL_CATALOG` table.

To delete a model from the model catalog, issue a query similar to the following:

```
mysql> DELETE FROM ML_SCHEMA_user1.MODEL_CATALOG WHERE model_id = 3;
```

where:

- `ML_SCHEMA_user1.MODEL_CATALOG` is the fully qualified name of the `MODEL_CATALOG` table. The schema is named for the user that created the model.
- `model_id = 3` is the ID of the model you want to delete.

3.14.10 Sharing Models

A user can only load their own model, the `ML_MODEL_LOAD user` parameter is ignored, and it is not possible to share a model. See Section 2.3.8, “CREATE TABLE ... SELECT Statements” for an alternative method.

Sharing a model requires granting model catalog privileges to another user. You can only share a model with another MySQL user on the same MySQL DB System.

To grant model catalog privileges, issue a statement similar to the following:

```
mysql> GRANT SELECT, ALTER, INSERT, CREATE, UPDATE, DROP, GRANT OPTION
         ON ML_SCHEMA_user1.MODEL_CATALOG
         TO 'user2'@'%';
```

where:

- `ML_SCHEMA_user1.MODEL_CATALOG` is the fully qualified name of the `MODEL_CATALOG` table. The schema is named for the user that created the model.
- `'user2'@'%'` is the user you want to grant access to.

> **Note**
>
> The user that is granted model catalog privileges must also have the privileges required to use HeatWave AutoML and the `CREATE` privilege on the schema where

> `ML_PREDICT_TABLE` or `ML_EXPLAIN_TABLE` results are written. See Section 3.2, "HeatWave AutoML Prerequisites".

After a model catalog is shared with another user, that user can access models in the catalog when running `ML_*` routines. For example, `'user2'@'%'` in the example above might assign a model handle from the `user1` model catalog to a session variable, and call that session variable from a `ML_PREDICT_TABLE` routine. The model owner is responsible for loading a model shared with other users.

```
mysql> SET @my_model = (SELECT model_handle
  FROM ML_SCHEMA_user1.MODEL_CATALOG
  WHERE train_table_name LIKE '%census_train%');

mysql> SELECT @my_model;
+--------------------------------------------------+
| @my_model                                        |
+--------------------------------------------------+
| heatwaveml_bench.census_train_user1_1648167434   |
+--------------------------------------------------+

mysql> CALL sys.ML_PREDICT_TABLE('heatwaveml_bench.census_test_subset', @my_model,
'heatwaveml_bench.census_predictions');
```

3.14.11 Data Drift Detection

HeatWave AutoML includes data drift detection for classification and regression models.

Machine learning typically makes an assumption that the training data and test data are similar. Over time, the similarity between the training data and the test data can decrease. This is known as data drift.

HeatWave AutoML monitors data drift with the following additions to the model catalog and to the `ML_PREDICT_ROW` and `ML_PREDICT_TABLE` routines:

- The `model_metadata` column in the model catalog includes the `training_drift_metric` JSON object literal which contains `mean` and `variance` numeric values. See: Section 3.14.1.3, "Model Metadata".
- The `ML_PREDICT_ROW` and `ML_PREDICT_TABLE` *`options`* parameter includes the `additional_details` boolean value.
- The `ML_PREDICT_ROW` and `ML_PREDICT_TABLE` `ml_results` column includes the `drift` JSON object literal which contains the `metric` numeric value and the `attribution_percent` JSON object literal. `attribution_percent` records up to 3 features with the highest attribution percentage values for each result.

To use data drift detection, follow this process:

1. During training, the `ML_TRAIN` routine records the `training_drift_metric`. Once training is complete, review the `mean` and `variance` values.

   `mean` and `variance` indicate the quality of the trained drift detector, and both values should be low. `mean` is more important, and if it is greater than 1.0, then drift evaluation for the test results might not be reliable.

2. Set the `additional_details` option to `true` for `ML_PREDICT_ROW` and `ML_PREDICT_TABLE` to record `drift` in `ml_results`.

3. Run `ML_PREDICT_ROW` or `ML_PREDICT_TABLE`, and review `drift` in `ml_results`.

   `metric` indicates the similarity between training and test data. A low value indicates similar values. A value grater than 1.0 indicates data drift, and the prediction results will be questionable.

`attribution_percent` indicates the top three features that contribute to data drift for each result. The higher the percentage value, the greater the contribution.

Syntax Examples

- The `model_metadata` includes the `training_drift_metric` JSON object literal.

```
mysql> CALL sys.ML_TRAIN('mlcorpus_v5.`titanic_train`', 'survived', NULL, @model);
Query OK, 0 rows affected (1 min 11.7002 sec)

mysql> SELECT JSON_PRETTY(model_metadata) FROM ML_SCHEMA_root.MODEL_CATALOG WHERE model_handle=@model;
+-----------------------------------------------------+
| JSON_PRETTY(model_metadata)                         |
+-----------------------------------------------------+
| {
  "task": "classification",
  "notes": null,
  "format": "HWMLv1.0",
  "n_rows": 916,
  "status": "Ready",
  "options": {
    "model_explainer": "permutation_importance",
    "prediction_explainer": "permutation_importance"
  },
  "n_columns": 13,
  "column_names": [
    "pclass",
    "name",
    "sex",
    "age",
    "sibsp",
    "parch",
    "ticket",
    "fare",
    "cabin",
    "embarked",
    "boat",
    "body",
    "home.dest"
  ],
  "contamination": null,
  "model_quality": "high",
  "training_time": 57.53120040893555,
  "algorithm_name": "XGBClassifier",
  "training_score": -0.07736892998218536,
  "build_timestamp": 1699468966,
  "n_selected_rows": 732,
  "training_params": {
    "recommend": "ratings",
    "force_use_X": false,
    "recommend_k": 3
  },
  "train_table_name": "mlcorpus_v5.titanic_train",
  "model_explanation": {
    "permutation_importance": {
      "age": 0.0,
      "sex": 0.0,
      "boat": 0.4445,
      "body": 0.0,
      "fare": 0.0,
      "name": 0.0,
      "cabin": 0.0,
      "parch": 0.0,
      "sibsp": 0.0,
      "pclass": 0.0,
```

```
      "ticket": 0.0,
      "embarked": 0.0,
      "home.dest": 0.0
    }
  },
  "n_selected_columns": 2,
  "target_column_name": "survived",
  "optimization_metric": "neg_log_loss",
  "selected_column_names": [
    "boat",
    "sex"
  ],
  "training_drift_metric": {
    "mean": 0.278,
    "variance": 0.2356
  }
} |
+-----------------------------------------------------+
1 row in set (0.0004 sec)
```

- A `ML_PREDICT_TABLE` example with `additional_details` set to `true`.

```
mysql> CALL sys.ML_TRAIN('mlcorpus_v5.`diamonds_train`', 'price', JSON_OBJECT('task','regression'), @model);
Query OK, 0 rows affected (7 min 47.9567 sec)

mysql> CALL sys.ML_MODEL_LOAD(@model, NULL);
Query OK, 0 rows affected (0.7665 sec)

mysql> DROP TABLE IF EXISTS diamonds_predictions_experiment_results;
Query OK, 0 rows affected (0.0106 sec)

mysql> CALL sys.ML_PREDICT_TABLE('mlcorpus_v5.`diamonds_test`', @model, 'mlcorpus_v5.`diamonds_predictions_e
Query OK, 0 rows affected (28.5353 sec)

mysql> SELECT ml_results FROM diamonds_predictions_experiment_results
          WHERE JSON_EXTRACT(ml_results, '$.drift.metric') > 0.5
          LIMIT 10;
+-------------------------------------------------------------------------------------------------------------
| ml_results
+-------------------------------------------------------------------------------------------------------------
| {"predictions": {"price": 4769.22265625}, "drift": {"metric": 0.69, "attribution_percent": {"cut": 100.0,
| {"predictions": {"price": 2610.075439453125}, "drift": {"metric": 0.57, "attribution_percent": {"color": 9
| {"predictions": {"price": 2725.368896484375}, "drift": {"metric": 0.54, "attribution_percent": {"cut": 100
| {"predictions": {"price": 7102.55224609375}, "drift": {"metric": 2.49, "attribution_percent": {"z": 64.53,
| {"predictions": {"price": 3622.7236328125}, "drift": {"metric": 0.55, "attribution_percent": {"color": 81.
| {"predictions": {"price": 3879.93701171875}, "drift": {"metric": 2.24, "attribution_percent": {"z": 70.23,
| {"predictions": {"price": 566.2338256835938}, "drift": {"metric": 0.67, "attribution_percent": {"color": 9
| {"predictions": {"price": 2495.825439453125}, "drift": {"metric": 0.64, "attribution_percent": {"cut": 100
| {"predictions": {"price": 421.9180603027344}, "drift": {"metric": 0.58, "attribution_percent": {"color": 1
| {"predictions": {"price": 325.4655456542969}, "drift": {"metric": 0.53, "attribution_percent": {"color": 1
+-------------------------------------------------------------------------------------------------------------
10 rows in set (0.0048 sec)
```

- A `ML_PREDICT_ROW` example with `additional_details` set to `true`.

```
mysql> SELECT JSON_OBJECT('carat', `diamonds_test`.`carat`, 'cut', `diamonds_test`.`cut`, 'color', `diamonds
          AS obj
          FROM `diamonds_test`
          WHERE JSON_UNQUOTE(JSON_OBJECT(JSON_OBJECT('carat', `diamonds_test`.`carat`, 'cut', `diamonds_test
          LIMIT 1
          INTO @row;
Query OK, 1 row affected (0.0033 sec)

mysql> SELECT sys.ML_PREDICT_ROW(@row, @model, JSON_OBJECT('additional_details', TRUE)) FROM `diamonds_test`
+-------------------------------------------------------------------------------------------------------------
| sys.ML_PREDICT_ROW(@row, @model, JSON_OBJECT('additional_details', TRUE))
+-------------------------------------------------------------------------------------------------------------
```

```
| {"x": 9.1000003815, "y": 8.970000267, "z": 5.6700000763, "cut": "Premium", "carat": 3.0099999905, "col
+-----------------------------------------------------------------------------------------------------
1 row in set (0.1670 sec)
```

3.15 Progress tracking

HeatWave AutoML includes progress tracking. Use a second MySQL Client window to track progress with calls to the Performance Schema.

Progress tracking supports all HeatWave AutoML routines:

- ML_TRAIN.
- ML_EXPLAIN.
- ML_MODEL_IMPORT for models in ONNX format. Progress tracking does not support ML_MODEL_IMPORT for models in HeatWave AutoML format.
- ML_PREDICT_ROW.
- ML_PREDICT_TABLE.
- ML_EXPLAIN_ROW.
- ML_EXPLAIN_TABLE.
- ML_SCORE.
- ML_MODEL_LOAD.
- ML_MODEL_UNLOAD.

For each of these routines, progress tracking tracks each individual operation, stages within routines, and includes a completed percentage value.

Syntax Examples

- Run ML_TRAIN from the first MySQL Client window:

```
mysql>  CALL sys.ML_TRAIN('mlcorpus_v5.`titanic_train`', 'survived', NULL, @model);
Query OK, 0 rows affected (1 min 28.7369 sec)
```

From the second MySQL Client window, run the following successive queries:

1. At the start of the ML_TRAIN operation.

```
mysql> SELECT * FROM performance_schema.rpd_query_stats;
+----------+--------------+---------------+---------------+-----------------------------------------
| QUERY_ID | STATEMENT_ID | CONNECTION_ID | QUERY_TEXT    | QEXEC_TEXT
+----------+--------------+---------------+---------------+-----------------------------------------
|        1 |   4294967295 |    4294967295 | ML_LOAD_TABLE | {"status": "Completed", "completedSteps":
|        2 |   4294967295 |    4294967295 | ML_TRAIN      | {"options": null, "operation": "training"
+----------+--------------+---------------+---------------+-----------------------------------------
2 rows in set (0.0005 sec)
```

2. During the ML_TRAIN operation.

```
mysql> SELECT * FROM performance_schema.rpd_query_stats;
+----------+--------------+---------------+---------------+-----------------------------------------
| QUERY_ID | STATEMENT_ID | CONNECTION_ID | QUERY_TEXT    | QEXEC_TEXT
+----------+--------------+---------------+---------------+-----------------------------------------
|        1 |   4294967295 |    4294967295 | ML_LOAD_TABLE | {"status": "Completed", "completedSteps":
|        2 |   4294967295 |    4294967295 | ML_TRAIN      | {"status": "In Progress", "completedSteps
```

```
+----------+--------------+--------------+--------------+-----------------------------------------
2 rows in set (0.0005 sec)
```

3. When the ML_TRAIN operation is complete.

```
mysql> SELECT * FROM performance_schema.rpd_query_stats;
+----------+--------------+---------------+-------------------+-----------------------------------------
| QUERY_ID | STATEMENT_ID | CONNECTION_ID | QUERY_TEXT        | QEXEC_TEXT
+----------+--------------+---------------+-------------------+-----------------------------------------
|        1 |   4294967295 |    4294967295 | ML_LOAD_TABLE     | {"status": "Completed", "completedSteps":
|        2 |   4294967295 |    4294967295 | ML_TRAIN          | {"status": "Completed", "completedSteps":
|        3 |   4294967295 |    4294967295 | ML_MODEL_OBJECT   | {"status": "Completed", "completedSteps":
|        4 |   4294967295 |    4294967295 | ML_MODEL_METADATA | {"status": "Completed", "completedSteps":
|        5 |   4294967295 |    4294967295 | ML_UNLOAD_MODEL   | {"status": "Completed", "completedSteps":
+----------+--------------+---------------+-------------------+-----------------------------------------
5 rows in set (0.0005 sec)
```

- As an example for other operations, run ML_EXPLAIN from the first MySQL Client window:

```
mysql> CALL sys.ML_MODEL_LOAD(@model, NULL);
Query OK, 0 rows affected (0.5951 sec)

mysql> CALL sys.ML_EXPLAIN('mlcorpus_v5.`titanic_train`', 'survived', @model,
                    JSON_OBJECT('model_explainer', 'fast_shap', 'prediction_explainer', 'shap'));
Query OK, 0 rows affected (8 min 35.2447 sec)
```

From the second MySQL Client window, run the following successive queries:

1. At the start of the ML_EXPLAIN operation. The first five rows relate to the progress of ML_TRAIN.

```
mysql> SELECT * FROM performance_schema.rpd_query_stats;
+----------+--------------+---------------+-------------------+-----------------------------------------
| QUERY_ID | STATEMENT_ID | CONNECTION_ID | QUERY_TEXT        | QEXEC_TEXT
+----------+--------------+---------------+-------------------+-----------------------------------------
|        6 |   4294967295 |    4294967295 | ML_LOAD_TABLE     | {"status": "Completed", "completedSteps":
|        7 |   4294967295 |    4294967295 | ML_LOAD_TABLE     | {"status": "Completed", "completedSteps":
|        8 |   4294967295 |    4294967295 | ML_EXPLAIN        | {"options": {"model_explainer": "fast_sha
+----------+--------------+---------------+-------------------+-----------------------------------------
8 rows in set (0.0005 sec)
```

2. During the ML_EXPLAIN operation.

```
mysql> SELECT * FROM performance_schema.rpd_query_stats;
+----------+--------------+---------------+-------------------+-----------------------------------------
| QUERY_ID | STATEMENT_ID | CONNECTION_ID | QUERY_TEXT        | QEXEC_TEXT
+----------+--------------+---------------+-------------------+-----------------------------------------
|        6 |   4294967295 |    4294967295 | ML_LOAD_TABLE     | {"status": "Completed", "completedSteps":
|        7 |   4294967295 |    4294967295 | ML_LOAD_TABLE     | {"status": "Completed", "completedSteps":
|        8 |   4294967295 |    4294967295 | ML_EXPLAIN        | {"status": "In Progress", "completedSteps
+----------+--------------+---------------+-------------------+-----------------------------------------
8 rows in set (0.0005 sec)
```

3. When the ML_EXPLAIN operation is complete.

```
mysql> SELECT * FROM performance_schema.rpd_query_stats;
+----------+--------------+---------------+-------------------+-----------------------------------------
| QUERY_ID | STATEMENT_ID | CONNECTION_ID | QUERY_TEXT        | QEXEC_TEXT
+----------+--------------+---------------+-------------------+-----------------------------------------
|        6 |   4294967295 |    4294967295 | ML_LOAD_TABLE     | {"status": "Completed", "completedSteps":
|        7 |   4294967295 |    4294967295 | ML_LOAD_TABLE     | {"status": "Completed", "completedSteps":
|        8 |   4294967295 |    4294967295 | ML_EXPLAIN        | {"status": "Completed", "completedSteps":
|        9 |   4294967295 |    4294967295 | ML_MODEL_OBJECT   | {"status": "Completed", "completedSteps":
|       10 |   4294967295 |    4294967295 | ML_MODEL_METADATA | {"status": "Completed", "completedSteps":
+----------+--------------+---------------+-------------------+-----------------------------------------
10 rows in set (0.0005 sec)
```

- Run `ML_PREDICT_ROW` from the first MySQL Client window:

```
mysql> SELECT sys.ML_PREDICT_ROW(
          JSON_OBJECT('pclass',`titanic_test`.`pclass`,
                      'name',`titanic_test`.`name`,
                      'sex',`titanic_test`.`sex`,
                      'age',`titanic_test`.`age`,
                      'sibsp',`titanic_test`.`sibsp`,
                      'parch',`titanic_test`.`parch`,
                      'ticket',`titanic_test`.`ticket`,
                      'fare',`titanic_test`.`fare`,
                      'cabin',`titanic_test`.`cabin`,
                      'embarked',`titanic_test`.`embarked`,
                      'boat',`titanic_test`.`boat`,
                      'body',`titanic_test`.`body`,
                      'home.dest',`titanic_test`.`home.dest` ),
          @model, NULL) FROM `titanic_test` LIMIT 4;
+-----------------------------------------------------------------------------------------------------------
| sys.ML_PREDICT_ROW(JSON_OBJECT('pclass',`titanic_test`.`pclass`,'name',`titanic_test`.`name`,'sex',`ti
+-----------------------------------------------------------------------------------------------------------
| {"age": 20.0, "sex": "male", "boat": null, "body": 89.0, "fare": 9.2250003815, "name": "Olsvigen, Mr. T
| {"age": 4.0, "sex": "female", "boat": "2", "body": null, "fare": 22.0249996185, "name": "Kink-Heilmann
| {"age": 42.0, "sex": "male", "boat": null, "body": 120.0, "fare": 7.6500000954, "name": "Humblen, Mr. A
| {"age": 45.0, "sex": "male", "boat": "7", "body": null, "fare": 29.7000007629, "name": "Chevre, Mr. Pa
+-----------------------------------------------------------------------------------------------------------
4 rows in set (1.1977 sec)
```

  From the second MySQL Client window, run the following query:

```
mysql> SELECT * FROM performance_schema.rpd_query_stats;
+----------+--------------+---------------+-------------------+-------------------------------------
| QUERY_ID | STATEMENT_ID | CONNECTION_ID | QUERY_TEXT        | QEXEC_TEXT
+----------+--------------+---------------+-------------------+-------------------------------------
|       23 |   4294967295 |    4294967295 | ML_PREDICT_ROW    | {"status": "Completed", "completedSteps"
|       24 |   4294967295 |    4294967295 | ML_PREDICT_ROW    | {"status": "Completed", "completedSteps"
|       25 |   4294967295 |    4294967295 | ML_PREDICT_ROW    | {"status": "Completed", "completedSteps"
|       26 |   4294967295 |    4294967295 | ML_PREDICT_ROW    | {"status": "Completed", "completedSteps"
+----------+--------------+---------------+-------------------+-------------------------------------
26 rows in set (0.0005 sec)
```

- Call `ML_PREDICT_TABLE` with a `batch_size` of 2 on a table with 4 rows:

```
mysql> CALL sys.ML_MODEL_LOAD(@model, NULL);
Query OK, 0 rows affected (0.5957 sec)

mysql> CREATE TABLE `titanic_test_temp` AS SELECT * FROM `titanic_test` LIMIT 4;
Query OK, 4 rows affected (0.0200 sec)

mysql> CALL sys.ML_PREDICT_TABLE('mlcorpus_v5.`titanic_test_temp`', @model,
          'mlcorpus_v5.`titanic_predictions`', JSON_OBJECT('batch_size',2));
Query OK, 0 rows affected (3.6509 sec)

mysql> SELECT * FROM titanic_predictions;
+-------------------+--------+-----------------------------------------+--------+-----+-------+-------+
| _4aad19ca6e_pk_id | pclass | name                                    | sex    | age | sibsp | parch |
+-------------------+--------+-----------------------------------------+--------+-----+-------+-------+
|                 1 |      3 | Olsvigen, Mr. Thor Anderson             | male   |  20 |     0 |     0 |
|                 2 |      3 | Kink-Heilmann, Miss. Luise Gretchen     | female |   4 |     0 |     2 |
|                 3 |      3 | Humblen, Mr. Adolf Mathias Nicolai Olsen | male   |  42 |     0 |     0 |
|                 4 |      1 | Chevre, Mr. Paul Romaine                | male   |  45 |     0 |     0 |
+-------------------+--------+-----------------------------------------+--------+-----+-------+-------+
4 rows in set (0.0006 sec)
```

  From the second MySQL Client window, run the following query:

```
mysql> SELECT * FROM performance_schema.rpd_query_stats;
```

```
+----------+--------------+---------------+-------------------+-----------------------------------------
| QUERY_ID | STATEMENT_ID | CONNECTION_ID | QUERY_TEXT        | QEXEC_TEXT
+----------+--------------+---------------+-------------------+-----------------------------------------
|        8 |       397099 |            10 | ML_LOAD_TABLE     | {"status": "Completed", "completedSteps": []
|        9 |       397224 |            10 | ML_LOAD_TABLE     | {"status": "Completed", "completedSteps": []
|       10 |       595247 |            10 | ML_PREDICT_TABLE  | {"status": "Completed", "completedSteps": []
|       11 |       595318 |            10 | ML_LOAD_TABLE     | {"status": "Completed", "completedSteps": []
|       12 |       793401 |            10 | ML_PREDICT_TABLE  | {"status": "Completed", "completedSteps": []
+----------+--------------+---------------+-------------------+-----------------------------------------
12 rows in set (0.0011 sec)
```

- To extract the completed percentage for the `ML_TRAIN` operation:

```
mysql> SELECT QEXEC_TEXT INTO @progress_json
                    FROM performance_schema.rpd_query_stats
                    WHERE QUERY_TEXT='ML_TRAIN'
                    ORDER BY QUERY_ID
                    DESC limit 1;
Query OK, 1 row affected (0.0009 sec)

mysql> SELECT JSON_EXTRACT(@progress_json, '$.completionPercentage');
Query OK, 1 row affected (0.0009 sec)
+-------------------------------------------------------+
| JSON_EXTRACT(@progress_json, '$.completionPercentage') |
+-------------------------------------------------------+
| 63                                                    |
+-------------------------------------------------------+
1 row in set (0.0005 sec)
```

- To extract the status for the `ML_TRAIN` operation:

```
mysql> SELECT QEXEC_TEXT INTO @progress_json
          FROM performance_schema.rpd_query_stats
          WHERE QUERY_TEXT='ML_TRAIN'
          ORDER BY QUERY_ID
          DESC limit 1;
Query OK, 1 row affected (0.0009 sec)

mysql> SELECT JSON_EXTRACT(@progress_json, '$.status');
Query OK, 1 row affected (0.0009 sec)
+----------------------------------------+
| JSON_EXTRACT(@progress_json, '$.status') |
+----------------------------------------+
| "In Progress"                          |
+----------------------------------------+
1 row in set (0.0006 sec)
```

3.16 HeatWave AutoML Routines

HeatWave AutoML routines reside in the MySQL `sys` schema.

Examples in this section are based on the *Iris Data Set*. See Section 8.4, “Iris Data Set Machine Learning Quickstart”.

3.16.1 ML_TRAIN

Run the `ML_TRAIN` routine on a labeled training dataset to produce a trained machine learning model.

MySQL 9.0.0 introduces support for large models that changes how HeatWave AutoML stores models, see: Section 3.14.1, “The Model Catalog”. `ML_TRAIN` upgrades older models.

ML_TRAIN Syntax

```
mysql> CALL sys.ML_TRAIN ('table_name', 'target_column_name', [options], model_handle);
```

```
options: {
     JSON_OBJECT("key","value"["key","value"] ...)
          "key","value": {
          ['task', {'classification'|'regression'|'forecasting'|'anomaly_detection'|'recommendation'}|NULL
          ['datetime_index', 'column']
          ['endogenous_variables', JSON_ARRAY('column'['column'] ...)]
          ['exogenous_variables', JSON_ARRAY('column'['column'] ...)]
          ['model_list', JSON_ARRAY('model'['model'] ...)]
          ['exclude_model_list', JSON_ARRAY('model'['model'] ...)]
          ['optimization_metric', 'metric']
          ['include_column_list', JSON_ARRAY('column'['column'] ...)]
          ['exclude_column_list', JSON_ARRAY('column'['column'] ...)]
          ['contamination',  'contamination factor']
          ['users', 'users_column']
          ['items', 'items_column']
          ['notes', 'notes_text']
          ['feedback', {'explicit'  ['implicit'}]
          ['feedback_threshold', 'threshold']
          ['item_metadata', JSON_OBJECT('table_name'['schema_name.table_name'] ...)]
          }
}
```

> **Note**
>
> The MySQL account that runs `ML_TRAIN` cannot have a period character (".") in its name; for example, a user named `'joesmith'@'%'` is permitted to train a model, but a user named `'joe.smith'@'%'` is not. For more information about this limitation, see Section 3.19, “HeatWave AutoML Limitations”.

The `ML_TRAIN` routine also runs the `ML_EXPLAIN` routine with the default Permutation Importance model for prediction explainers and model explainers. See Section 3.6, “Training Explainers”. To train other prediction explainers and model explainers use the `ML_EXPLAIN` routine with the preferred explainer after `ML_TRAIN`.

`ML_EXPLAIN` does not support the `anomaly_detection` and `recommendation` tasks, and `ML_TRAIN` does not run `ML_EXPLAIN`.

`ML_TRAIN` parameters:

- `table_name`: The name of the table that contains the labeled training dataset. The table name must be valid and fully qualified; that is, it must include the schema name, *`schema_name.table_name`*. The table cannot exceed 10 GB, 100 million rows, or 1017 columns.

- `target_column_name`: The name of the target column containing ground truth values.

  HeatWave AutoML does not support a text target column.

  Anomaly detection does not require labelled data, and `target_column_name` must be set to `NULL`.

  Forecasting does not require `target_column_name`, and it can be set to `NULL`.

- `model_handle`: The name of a user-defined session variable that stores the machine learning model handle for the duration of the connection. User variables are written as *`@var_name`*. Some of the examples in this guide use `@census_model` as the variable name. Any valid name for a user-defined variable is permitted, for example `@my_model`.

  If the `model_handle` variable was set to a value before calling `ML_TRAIN`, that model handle is used for the model. A model handle must be unique in the model catalog.

Otherwise, HeatWave AutoML generates a model handle. When `ML_TRAIN` finishes executing, retrieve the generated model handle by querying the session variable. See Section 3.14.8, “Model Handles”.

- *`options`*: Optional parameters specified as key-value pairs in `JSON` format. If an option is not specified, the default setting is used. If no options are specified, you can specify `NULL` in place of the `JSON` argument.

  - `task`: Specifies the machine learning task. Permitted values are:

    - `classification`: The default. Use this task type if the target is a discrete value.
    - `regression`: Use this task type if the target column is a continuous numerical value.
    - `forecasting`: Use this task type if the target column is a date-time column that requires a timeseries forecast. The `datetime_index` and `endogenous_variables` parameters are required with the `forecasting` task.
    - `anomaly_detection`: Use this task type to detect anomalies.
    - `recommendation`: Use this task type for recommendation models.

  - `datetime_index`: For forecasting tasks, the column name for a datetime column that acts as an index for the forecast variable. The column can be one of the supported datetime column types, `DATETIME`, `TIMESTAMP`, `DATE`, `TIME`, and `YEAR`, or an auto-incrementing index.

    The forecast models `SARIMAXForecaster`, `VARMAXForecaster`, and `DynFactorForecaster` cannot back test, that is forecast into training data, when using `exogenous_variables`. Therefore, the predict table must not overlap the `datetime_index` with the training table. The start date in the predict table must be a date immediately following the last date in the training table when `exogenous_variables` are used. For example, the predict table has to start with year 2024 if the training table with `YEAR` data type `datetime_index` ends with year 2023.

    The `datetime_index` for the predict table must not have missing dates after the last date in the training table. For example, the predict table has to start with year 2024 if the training table with `YEAR` data type `datetime_index` ends with year 2023. The predict table cannot start with year, for example, 2025 or 2030, because that would miss out 1 and 6 years, respectively.

    When *`options`* do not include `exogenous_variables` , the predict table can overlap the `datetime_index` with the training table. This supports back testing.

    The valid range of years for `datetime_index` dates must be between 1678 and 2261. It will cause an error if any part of the training table or predict table has dates outside this range. The last date in the training table plus the predict table length must still be inside the valid year range. For example, if the `datetime_index` in the training table has `YEAR` data type, and the last date is year 2023, the predict table length must be less than 238 rows: 2261 minus 2023 equals 238 rows.

  - `endogenous_variables`: For forecasting tasks, the column or columns to be forecast.

    Univariate forecasting models support a single numeric column, specified as a `JSON_ARRAY`. This column must also be specified as the `target_column_name`, because that field is required, but it is not used in that location.

    Multivariate forecasting models support multiple numeric columns, specified as a `JSON_ARRAY`. One of these columns must also be specified as the `target_column_name`.

    `endogenous_variables` cannot be text.

- `exogenous_variables`: For forecasting tasks, the column or columns of independent, non-forecast, predictive variables, specified as a `JSON_ARRAY`. These optional variables are not forecast, but help to predict the future values of the forecast variables. These variables affect a model without being affected by it. For example, for sales forecasting these variables might be advertising expenditure, occurrence of promotional events, weather, or holidays.

  `ML_TRAIN` will consider all supported models during the algorithm selection stage if *`options`* includes `exogenous_variables`, including models that do not support `exogenous_variables`.

  For example, if *`options`* includes univariate `endogenous_variables` with `exogenous_variables`, then `ML_TRAIN` will consider `NaiveForecaster`, `ThetaForecaster`, `ExpSmoothForecaster`, `ETSForecaster`, `STLwESForecaster`, `STLwARIMAForecaster`, and `SARIMAXForecaster`. `ML_TRAIN` will ignore `exogenous_variables` if the model does not support them.

  Similarly, if *`options`* includes multivariate `endogenous_variables` with `exogenous_variables`, then `ML_TRAIN` will consider `VARMAXForecaster` and `DynFactorForecaster`.

  If *`options`* also includes `include_column_list`, this will force `ML_TRAIN` to only consider those models that support `exogenous_variables`.

- `model_list`: The type of model to be trained. If more than one model is specified, the best model type is selected from the list. See Section 3.16.13, “Model Types”.

  This option cannot be used together with the `exclude_model_list` option. Before MySQL 8.4.0, `model_list` is not supported for `anomaly_detection` tasks. As of MySQL 8.4.0, you can select the Principal Component Analysis (PCA) model and Generalized Local Outlier Factor (GLOF) model

for `anomaly_detection` tasks. The default model is the Generalized kth Nearest Neighbors (GkNN) model.

- `exclude_model_list`: Model types that should not be trained. Specified model types are excluded from consideration during model selection. See Section 3.16.13, "Model Types".

  This option cannot be specified together with the `model_list` option, and it is not supported for `anomaly_detection` tasks.

- `optimization_metric`: The scoring metric to optimize for when training a machine learning model. The metric must be compatible with the `task` type and the target data. See Section 3.16.14, "Optimization and Scoring Metrics".

  This is not supported for `anomaly_detection` tasks.

- `include_column_list`: `ML_TRAIN` must include this list of columns.

  For `classification`, `regression`, `anomaly_detection` and `recommendation` tasks, `include_column_list` ensures that `ML_TRAIN` will not drop these columns.

  For `forecasting` tasks, `include_column_list` can only include `exogenous_variables`. If `include_column_list` contains at least one `exogenous_variables`, this will force `ML_TRAIN` to only consider those models that support `exogenous_variables`.

  All columns in `include_column_list` must be included in the training table.

- `exclude_column_list`: Feature columns of the training dataset to exclude from consideration when training a model. Columns that are excluded using `exclude_column_list` do not need to be excluded from the dataset used for predictions.

  The `exclude_column_list` cannot contain any columns provided in `endogenous_variables`, `exogenous_variables`, and `include_column_list`.

- `contamination`: The optional contamination factor for use with the `anomaly_detection` task. 0 < `contamination` < 0.5. The default value is 0.1.

- `users`: Specifies the column name corresponding to the user ids.

  This must be a valid column name, and it must be different from the `items` column name.

- `items`: Specifies the column name corresponding to the item ids.

  This must be a valid column name, and it must be different from the `users` column name.

- `notes`: Add notes to the `model_metadata`.

- `feedback`: The type of feedback for a recommendation model, `explicit`, the default, or `implicit`.

- `feedback_threshold`: The feedback threshold for a recommendation model with implicit feedback. All ratings at or above the `feedback_threshold` are implied to provide positive feedback. All ratings below the `feedback_threshold` are implied to provide negative feedback. The default value is 1.

- `item_metadata:` When training a content-based recommendation model, this option specifies the table that has item descriptions. This table must have two columns: one specifying the `item_id` and the other with a text description of the item. You must include the `table_name` option as a JSON object and provide it in a fully qualified format as *`schema_name.table_name`*.

Syntax Examples

- An `ML_TRAIN` example that uses the `classification` task option implicitly (`classification` is the default if not specified explicitly):

```
mysql> CALL sys.ML_TRAIN('ml_data.iris_train', 'class',
         NULL, @iris_model);
```

- An `ML_TRAIN` example that specifies the `classification` task type explicitly, and sets a model handle instead of letting HeatWave AutoML generate one:

```
mysql> SET @iris_model = 'iris_manual';
mysql> CALL sys.ML_TRAIN('ml_data.iris_train', 'class',
         JSON_OBJECT('task', 'classification'),
         @iris_model);
```

- An `ML_TRAIN` example that specifies the `regression` task type:

```
mysql> CALL sys.ML_TRAIN('employee.salary_train', 'salary',
         JSON_OBJECT('task', 'regression'), @salary_model);
```

- An `ML_TRAIN` example that specifies the `model_list` option. This example trains either an `XGBClassifier` or `LGBMClassifier` model.

```
mysql> CALL sys.ML_TRAIN('ml_data.iris_train', 'class',
         JSON_OBJECT('task','classification',
         'model_list', JSON_ARRAY('XGBClassifier', 'LGBMClassifier')),
         @iris_model);
```

- An `ML_TRAIN` example that specifies the `exclude_model_list` option. In this example, `LogisticRegression` and `GaussianNB` models are excluded from model selection.

```
mysql> CALL sys.ML_TRAIN('ml_data.iris_train', 'class',
         JSON_OBJECT('task','classification',
         'exclude_model_list', JSON_ARRAY('LogisticRegression', 'GaussianNB')),
         @iris_model);
```

- An `ML_TRAIN` example that specifies the `optimization_metric` option.

```
mysql> CALL sys.ML_TRAIN('ml_data.iris_train', 'class',
         JSON_OBJECT('task','classification', 'optimization_metric', 'neg_log_loss') ,
         @iris_model);
```

- An `ML_TRAIN` example that specifies the `exclude_column_list` option.

```
mysql> CALL sys.ML_TRAIN('ml_data.iris_train', 'class',
         JSON_OBJECT('task','classification',
         'exclude_column_list', JSON_ARRAY('sepal length', 'petal length')),
         @iris_model);
```

- An `ML_TRAIN` example that adds `notes` to the `model_metadata`.

```
mysql> CALL sys.ML_TRAIN('ml_data.iris_train', 'class',
         JSON_OBJECT('task','classification', 'notes', 'testing2'),
         @iris_model);
Query OK, 0 rows affected (1 min 42.53 sec)

mysql>  SELECT model_metadata FROM ML_SCHEMA_root.MODEL_CATALOG
         WHERE model_handle=@iris_model;
+-----------------------------------------------------------------------------------------------------------
| model_metadata
+-----------------------------------------------------------------------------------------------------------
| {"task": "classification", "notes": "testing2", "format": "HWMLv1.0", "n_rows": 105, "status": "Ready"
+-----------------------------------------------------------------------------------------------------------
```

```
1 row in set (0.00 sec)
```

- An ML_TRAIN example that stores the model in the model_object_catalog.

```
mysql> CALL sys.ML_TRAIN('mlcorpus.iris_train', 'class', NULL, @iris_model);
Query OK, 0 rows affected (32.18 sec)

mysql> SELECT model_object, model_object_size
          FROM ML_SCHEMA_user1.MODEL_CATALOG
          WHERE model_handle=@iris_model;
+--------------+-------------------+
| model_object | model_object_size |
+--------------+-------------------+
| NULL         |            346866 |
+--------------+-------------------+
1 row in set (0.00 sec)

mysql> SELECT model_metadata->>'$.format', model_metadata->>'$.chunks'
          FROM ML_SCHEMA_user1.MODEL_CATALOG
          WHERE model_handle=@iris_model;
+-----------------------------+-----------------------------+
| model_metadata->>'$.format' | model_metadata->>'$.chunks' |
+-----------------------------+-----------------------------+
| HWMLv2.0                    | 1                           |
+-----------------------------+-----------------------------+
1 row in set (0.00 sec)

mysql> SELECT chunk_id, length(model_object)
          FROM ML_SCHEMA_user1.model_object_catalog
          WHERE model_handle=@iris_model;
+----------+----------------------+
| chunk_id | length(model_object) |
+----------+----------------------+
|        1 |               346866 |
+----------+----------------------+
1 row in set (0.00 sec)
```

See also:

- Section 3.9.1, “Training a Forecasting Model”
- Section 3.10.2, “Training an Anomaly Detection Model”
- Section 3.11.2, “Training a Recommendation Model”
- Section 3.14.1, “The Model Catalog”.
- Section 3.15, “Progress tracking”.

3.16.2 ML_EXPLAIN

Running the ML_EXPLAIN routine on a model and dataset trains a prediction explainer and model explainer, and adds a model explanation to the model catalog.

MySQL 9.0.0 introduces support for large models that changes how HeatWave AutoML stores models, see: Section 3.14.1, “The Model Catalog”. ML_EXPLAIN upgrades older models.

ML_EXPLAIN does not support recommendation models, and a call with a recommendation model will produce an error.

ML_EXPLAIN does not support anomaly detection, and a call with an anomaly detection model will produce an error.

ML_EXPLAIN Syntax

```
mysql> CALL sys.ML_EXPLAIN ('table_name', 'target_column_name',
          model_handle_variable, [options]);

options: {
     JSON_OBJECT("key","value"["key","value"] ...)
          "key","value": {
          ['model_explainer', {'permutation_importance'|'partial_dependence'|'shap'|'fast_shap'}| NULL]
          ['prediction_explainer', {'permutation_importance'|'shap'}]
          ['columns_to_explain', JSON_ARRAY('column'['column'] ...)]
          ['target_value', 'target_class']
          }
}
```

Run the `ML_EXPLAIN` routine before `ML_EXPLAIN_ROW` and `ML_EXPLAIN_TABLE` routines. The `ML_TRAIN` routine also runs the `ML_EXPLAIN` routine with the default Permutation Importance model. It is only necessary to use the `ML_EXPLAIN` routine to train prediction explainers and model explainers with a different model. See Section 3.6, “Training Explainers”.

`ML_EXPLAIN` parameters:

- `table_name`: The name of the table that contains the labeled training dataset. The table name must be valid and fully qualified; that is, it must include the schema name (*`schema_name`*.*`table_name`*). Use `NULL` for help. Use the dataset that the model is trained on - running `ML_EXPLAIN` on a dataset that the model has not been trained on produces errors or unreliable explanations.
- `target_column_name`: The name of the target column in the training dataset containing ground truth values.
- `model_handle`: A string containing the model handle for the model in the model catalog. Use `NULL` for help. The model explanation is stored in this model metadata. The model must be loaded first, for example:

  ```
  mysql> CALL sys.ML_MODEL_LOAD('ml_data.iris_train_user1_1636729526', NULL);
  ```

  If you run `ML_EXPLAIN` again with the same model handle and model explainer, the model explanation field is overwritten with the new result.
- *`options`*: Optional parameters specified as key-value pairs in `JSON` format. If an option is not specified, the default setting is used. If you specify `NULL` in place of the `JSON` argument, the default Permutation Importance model explainer is trained, and no prediction explainer is trained. The available options are:
  - `model_explainer`: Specifies the model explainer. Valid values are:
    - `permutation_importance`: The default model explainer.
    - `shap`: The SHAP model explainer, which produces global feature importance values based on Shapley values.
    - `fast_shap`: The Fast SHAP model explainer, which is a subsampling version of the SHAP model explainer that usually has a faster runtime.
    - `partial_dependence`: Explains how changing the values in one or more columns will change the value predicted by the model. The following additional arguments are required for the `partial_dependence` model explainer:
      - `columns_to_explain`: a JSON array of one or more column names in the table specified by *`table_name`*. The model explainer explains how changing the value in this column or columns affects the model.

    - `target_value`: a valid value that the target column containing ground truth values, as specified by `target_column_name`, can take.
- `prediction_explainer`: Specifies the prediction explainer. Valid values are:
    - `permutation_importance`: The default prediction explainer.
    - `shap`: The SHAP prediction explainer, which produces global feature importance values based on Shapley values.

Syntax Examples

- Load the model first:

```
mysql> CALL sys.ML_MODEL_LOAD('ml_data.iris_train_user1_1636729526', NULL);
```

- Running `ML_EXPLAIN` to train the SHAP prediction explainer and the Fast SHAP model explainer:

```
mysql> CALL sys.ML_EXPLAIN('ml_data.iris_train', 'class',
          'ml_data.iris_train_user1_1636729526',
          JSON_OBJECT('model_explainer', 'fast_shap', 'prediction_explainer', 'shap'));
```

- Run `ML_EXPLAIN` and use `NULL` for the options trains the default Permutation Importance model explainer and no prediction explainer:

```
mysql> CALL sys.ML_EXPLAIN('ml_data.iris_train', 'class',
          'ml_data.iris_train_user1_1636729526', NULL);
```

- Running `ML_EXPLAIN` to train the Partial Dependence model explainer (which requires extra options) and the SHAP prediction explainer:

```
mysql> CALL sys.ML_EXPLAIN('ml_data.iris_train', 'class', @iris_model,
          JSON_OBJECT('columns_to_explain', JSON_ARRAY('sepal width'),
          'target_value', 'Iris-setosa', 'model_explainer', 'partial_dependence',
          'prediction_explainer', 'shap'));
```

- Viewing the model explanation, in this case produced by the Permutation Importance model explainer:

```
mysql> SELECT model_explanation FROM MODEL_CATALOG WHERE model_handle = @iris_model;
+-------------------------------------------------------------------------------------------------------------
| model_explanation
+-------------------------------------------------------------------------------------------------------------
| {"permutation_importance": {"petal width": 0.5926, "sepal width": 0.0, "petal length": 0.0423, "sepal leng
+-------------------------------------------------------------------------------------------------------------
```

- An `ML_EXPLAIN` example that stores the model in the `model_object_catalog`.

```
mysql> SET @explain_option = JSON_OBJECT('model_explainer', 'shap', 'prediction_explainer', 'shap');
Query OK, 0 rows affected (0.00 sec)

mysql> CALL sys.ML_EXPLAIN('mlcorpus.iris_train', 'class', @iris_model, @explain_option);
Query OK, 0 rows affected (11.51 sec)

mysql> SELECT model_object, model_object_size
          FROM ML_SCHEMA_user1.MODEL_CATALOG
          WHERE model_handle=@iris_model;
+--------------+--------------------+
| model_object | model_object_size |
+--------------+--------------------+
| NULL         |             348954 |
+--------------+--------------------+
1 row in set (0.00 sec)
```

```
mysql> SELECT model_metadata->>'$.format', model_metadata->>'$.chunks'
          FROM ML_SCHEMA_user1.MODEL_CATALOG
          WHERE model_handle=@iris_model;
+-----------------------------+-----------------------------+
| model_metadata->>'$.format' | model_metadata->>'$.chunks' |
+-----------------------------+-----------------------------+
| HWMLv2.0                    | 1                           |
+-----------------------------+-----------------------------+
1 row in set (0.00 sec)


mysql> SELECT chunk_id, length(model_object)
          FROM ML_SCHEMA_user1.model_object_catalog
          WHERE model_handle=@iris_model;
+----------+----------------------+
| chunk_id | length(model_object) |
+----------+----------------------+
|        1 |               348954 |
+----------+----------------------+
1 row in set (0.00 sec)
```

3.16.3 ML_MODEL_EXPORT

MySQL 9.0.0 introduces the `ML_MODEL_EXPORT` routine. Use this routine to export a model from the model catalog to a user defined table.

MySQL 9.0.0 also introduces support for large models that changes how HeatWave AutoML stores models, see: Section 3.14.1, “The Model Catalog”. `ML_MODEL_EXPORT` upgrades older models.

The output table will have these columns:

- `chunk_id`:

  `INT AUTO_INCREMENT PRIMARY KEY`

- `model_object`:

  `LONGTEXT DEFAULT NULL`

- `model_metadata`:

  `JSON`

  See Section 3.14.1.3, “Model Metadata”.

`ML_MODEL_EXPORT` should work regardless of `model_metadata.status`:

- If there is no corresponding row in the `model_object_catalog` for an existing `model_handle` in the `MODEL_CATALOG`:

  There should be only one row in the output table with `chunk_id` = 0, `model_object` = NULL and `model_metadata` = `MODEL_CATALOG.model_metadata`.

- If there is at least one row in the `model_object_catalog` for an existing `model_handle` in the `MODEL_CATALOG`:

  - There should be N rows in the output table with `chunk_id` being 1 to N.
  - `ML_MODEL_EXPORT` copies the `model_object` from `model_object_catalog` to the output table.

- `model_metadata` in the row with `chunk_id` = 1 should be the same as in the `MODEL_CATALOG`.

If a user subsequently uses `ML_MODEL_IMPORT` to import the model into a different MySQL DB System, the results will depend upon the MySQL version:

- If the MySQL DB System has MySQL 9.0.0 or greater, the import should work.
- If the MySQL DB System has a MySQL version before Mysql 9.0.0:
  - If the model format is HWMLv2.0 or ONNXv2.0, then the import will fail.
  - If the model format is HWMLv1.0, ONNXv1.0 or ONNX, make the following changes:
    - Drop `model_metadata.chunks`.
    - Change the model format to ONNX if the model format is ONNXv1.0.

    The import should then succeed.

ML_MODEL_EXPORT Syntax

As of MySQL 9.0.0, `ML_MODEL_EXPORT` can export a model from the `model_object_catalog` to a user defined table.

```
mysql> CALL sys.ML_MODEL_EXPORT (model_handle, output_table_name);
```

`ML_MODEL_EXPORT` parameters:

- `model_handle`: The model handle for the model.
- `output_table_name`: The name for the output table.

Syntax Examples

- An example that exports a HeatWave AutoML model with metadata:

```
mysql> CALL sys.ML_MODEL_EXPORT(@iris_model, 'ML_SCHEMA_user1.model_export');
Query OK, 0 rows affected (0.06 sec)

mysql> SHOW CREATE TABLE ML_SCHEMA_user1.model_export;
+--------------+-----------------------------------------------------------------------+
| Table        | Create Table                                                          |
+--------------+-----------------------------------------------------------------------+
| model_export | CREATE TABLE `model_export` (
  `chunk_id` int NOT NULL AUTO_INCREMENT,
  `model_object` longtext,
  `model_metadata` json DEFAULT NULL,
  PRIMARY KEY (`chunk_id`)
) ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci |
+--------------+-----------------------------------------------------------------------+
1 row in set (0.00 sec)
```

3.16.4 ML_MODEL_IMPORT

Use the `ML_MODEL_IMPORT` routine to import a pre-trained model into the model catalog. HeatWave AutoML supports the import of HeatWave AutoML and ONNX, Open Neural Network Exchange, format models. After import, all the HeatWave AutoML routines can be used with an ONNX model.

Models in ONNX format, `.onnx`. cannot be loaded directly into a MySQL table. They require string serialization and conversion to Base64 binary encoding. Before running `ML_MODEL_IMPORT`, follow

the instructions in Section 3.14.2, “ONNX Model Import” to pre-process and then load the model into a temporary table for import to HeatWave.

MySQL 9.0.0 introduces support for large models that changes how HeatWave AutoML stores models, see: Section 3.14.1, “The Model Catalog”. `ML_MODEL_IMPORT` upgrades older models.

MySQL 9.0.0 also supports model import from a table. The supported model import formats are ONNX, and all the formats supported by the model catalog. The default import format is ONNX. HeatWave AutoML verifies models with the ONNX format, and stores them as ONNXv2.0.

The table should have the following columns, and their recommended parameters:

- `chunk_id`:

  `INT AUTO_INCREMENT PRIMARY KEY`

- `model_object`:

  `LONGTEXT NOT NULL`

- `model_metadata`:

  `JSON DEFAULT NULL`

  See Section 3.14.1.3, “Model Metadata”.

The table must meet the following criteria:

- There must be one row, and only one row, in the table with `chunk_id` = 1.
- The `model_metadata` corresponding to `chunk_id` = 1 must have the correct JSON key, value pair for the model format.

`ML_MODEL_IMPORT` will store the `model_metadata` corresponding to `chunk_id` = 1 in the model catalog, and ignore the `model_metadata` from other rows.

If `chunks` in the `model_metadata` corresponding to `chunk_id` = 1 is not set, it will be set to the number of rows in the input table.

If `ML_MODEL_IMPORT` fails or is cancelled, there will be no change to the `MODEL_CATALOG` and to the `model_object_catalog`.

ML_MODEL_IMPORT Syntax

As of MySQL 9.0.0, `ML_MODEL_IMPORT` can import a model from a table as well as a a preprocessed model object. This uses an alternative syntax for `model_metadata`:

```
mysql> CALL sys.ML_MODEL_IMPORT (model_object, model_metadata, model_handle);

model_metadata: {
     JSON_OBJECT("key","value"[,"key","value"] ...)
          "key","value": {
          ['schema', 'schema']
          ['table', 'table']
          }
}

model_metadata: {
     JSON_OBJECT("key","value"[,"key","value"] ...)
          "key","value": {
          ['task', {'classification'|'regression'|'forecasting'|'anomaly_detection'|'recommendation'}|NULL
          ['build_timestamp', 'timestamp']
```

```
          ['target_column_name', 'column']
          ['train_table_name', 'table']
          ['column_names', JSON_ARRAY('column'[,'column'] ...)]
          ['model_explanation', ml_explain_options]
          ['notes', 'notes']
          ['format', 'format']
          ['status', {'creating'|'ready'|'error'}|NULL]
          ['model_quality', 'quality']
          ['training_time', 'time']
          ['algorithm_name', 'algorithm']
          ['training_score', 'score']
          ['n_rows', 'rows']
          ['n_columns', 'columns']
          ['n_selected_rows', 'rows']
          ['n_selected_columns', 'columns']
          ['optimization_metric', 'metric']
          ['selected_column_names', JSON_ARRAY('column'[,'column'] ...)]
          ['contamination', 'contamination']
          ['options', ml_train_options]
          ['training_params', ml_train_params]
          ['onnx_inputs_info', data_types_map]
          ['onnx_outputs_info', labels_map]
          ['training_drift_metric', JSON_OBJECT('mean', 'value', 'variance', 'value')]
          ['chunks', 'chunks']
}
```

Before MySQL 9.0.0, use `ML_MODEL_IMPORT` to import a preprocessed model object:

```
mysql> CALL sys.ML_MODEL_IMPORT (model_object, model_metadata, model_handle);

model_metadata: {
     JSON_OBJECT("key","value"[,"key","value"] ...)
          "key","value": {
          ['task', {'classification'|'regression'|'forecasting'|'anomaly_detection'|'recommendation'}|NULL]
          ['build_timestamp', 'timestamp']
          ['target_column_name', 'column']
          ['train_table_name', 'table']
          ['column_names', JSON_ARRAY('column'[,'column'] ...)]
          ['model_explanation', ml_explain_options]
          ['notes', 'notes']
          ['format', 'format']
          ['status', {'creating'|'ready'|'error'}|NULL]
          ['model_quality', 'quality']
          ['training_time', 'time']
          ['algorithm_name', 'algorithm']
          ['training_score', 'score']
          ['n_rows', 'rows']
          ['n_columns', 'columns']
          ['n_selected_rows', 'rows']
          ['n_selected_columns', 'columns']
          ['optimization_metric', 'metric']
          ['selected_column_names', JSON_ARRAY('column'[,'column'] ...)]
          ['contamination', 'contamination']
          ['options', ml_train_options]
          ['training_params', ml_train_params]
          ['onnx_inputs_info', data_types_map]
          ['onnx_outputs_info', labels_map]
          ['training_drift_metric', JSON_OBJECT('mean', 'value', 'variance', 'value')]
          }
}
```

`ML_MODEL_IMPORT` parameters:

- `model_object`:
  - To import a model from a table: Set to `NULL`.

  - To import a model object: The preprocessed model object.

- `model_metadata`:

  - To import a model from a table:

    - `schema`: The name of the schema.

    - `table`: The name of the table.

  - To import a model object: An optional JSON object literal that contains key-value pairs with model metadata. See Section 3.14.1.3, “Model Metadata”.

- `model_handle`: The model handle for the model. The model is stored in the model catalog under this name and accessed using it. Specify a model handle that does not already exist in the model catalog. Set to `NULL` for HeatWave AutoML to generate a unique model handle.

Syntax Examples

- An example that imports a HeatWave AutoML model with metadata:

```
mysql> SET @hwml_model = "hwml_model";
Query OK, 0 rows affected (0.00 sec)

mysql> CALL sys.ML_TRAIN('mlcorpus.input_train', 'target', NULL, @hwml_model);
Query OK, 0 rows affected (23.36 sec)

mysql> SET @hwml_object = (SELECT model_object FROM ML_SCHEMA_root.MODEL_CATALOG
          WHERE model_handle=@hwml_model);
Query OK, 0 rows affected (0.00 sec)

mysql> SET @hwml_metadata = (SELECT model_metadata FROM ML_SCHEMA_root.MODEL_CATALOG
          WHERE model_handle=@hwml_model);
Query OK, 0 rows affected (0.00 sec)

mysql> CALL sys.ML_MODEL_IMPORT(@hwml_object, @hwml_metadata, @imported_model);
Query OK, 0 rows affected (0.01 sec)

mysql> model_metadata->>'$.task' AS task, model_owner,
          model_metadata->>'$.build_timestamp' AS build_timestamp,
          model_metadata->>'$.target_column_name' AS target_column_name,
          model_metadata->>'$.train_table_name' AS train_table_name, model_object_size,
          model_metadata->>'$.model_explanation' AS model_explanation
          FROM ML_SCHEMA_root.MODEL_CATALOG;
+----------------+----------------+-----------------+--------------------+----------------------+-------
| task           | model_owner    | build_timestamp | target_column_name | train_table_name     | model_
+----------------+----------------+-----------------+--------------------+----------------------+-------
| classification | root           |      1679202888 | target             | mlcorpus.input_train |
| classification | root           |      1679202888 | target             | mlcorpus.input_train |
+----------------+----------------+-----------------+--------------------+----------------------+-------
2 rows in set (0.00 sec)


mysql> SELECT model_metadata->>'$.column_names' AS column_names
          FROM ML_SCHEMA_root.MODEL_CATALOG
          WHERE model_handle LIKE 'hwml%';
+-----------------------------------------------------------------------------------------------+
| column_names                                                                                  |
+-----------------------------------------------------------------------------------------------+
| ["C0", "C1", "C2", "C3", "C4", "C5", "C6", "C7", "C8", "C9", "C10", "C11", "C12", "C13"] |
| ["C0", "C1", "C2", "C3", "C4", "C5", "C6", "C7", "C8", "C9", "C10", "C11", "C12", "C13"] |
+-----------------------------------------------------------------------------------------------+
2 rows in set (0.00 sec)
```

- An example that imports an ONNX model without specifying metadata:

```
mysql> CALL sys.ML_MODEL_IMPORT(@onnx_encode, NULL, 'onnx_test');
```

- An example that exports a model to a table, switches users, and then imports the model from that table:

```
mysql> CALL sys.ML_MODEL_EXPORT(@iris_model, 'ML_SCHEMA_user1.model_export');
Query OK, 0 rows affected (0.06 sec)

mysql> SHOW CREATE TABLE ML_SCHEMA_user1.model_export;
+--------------+-----------------------------------------------------------------------+
| Table        | Create Table                                                          |
+--------------+-----------------------------------------------------------------------+
| model_export | CREATE TABLE `model_export` (
  `chunk_id` int NOT NULL AUTO_INCREMENT,
  `model_object` longtext,
  `model_metadata` json DEFAULT NULL,
  PRIMARY KEY (`chunk_id`)
) ENGINE=InnoDB AUTO_INCREMENT=2 DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_0900_ai_ci |
+--------------+-----------------------------------------------------------------------+
1 row in set (0.00 sec)

# switch to user2

mysql> CALL sys.ML_MODEL_IMPORT(NULL, JSON_OBJECT('schema', 'ML_SCHEMA_user1', 'table', 'model_export'), @ir
Query OK, 0 rows affected (0.19 sec)

mysql> CALL sys.ML_MODEL_LOAD(@iris_export, NULL);
Query OK, 0 rows affected (0.63 sec)

mysql> SELECT model_object, model_object_size
          FROM ML_SCHEMA_user2.MODEL_CATALOG
          WHERE model_handle=@iris_export;
+--------------+-------------------+
| model_object | model_object_size |
+--------------+-------------------+
| NULL         |            348954 |
+--------------+-------------------+
1 row in set (0.00 sec)


mysql> SELECT chunk_id, LENGTH(model_object)
          FROM ML_SCHEMA_user2.model_object_catalog
          WHERE model_handle=@iris_export;
+----------+----------------------+
| chunk_id | LENGTH(model_object) |
+----------+----------------------+
|        1 |               348954 |
+----------+----------------------+
1 row in set (0.00 sec)
```

- An example that imports a model in ONNX format from a table:

```
mysql> DROP TABLE IF EXISTS model_table;

mysql> CREATE TABLE model_table (
          chunk_id INT AUTO_INCREMENT PRIMARY KEY,
          model_object LONGTEXT NOT NULL,
          model_metadata JSON DEFAULT NULL);

mysql> LOAD DATA INFILE '/onnx_examples/x00'
          INTO TABLE model_table
          CHARACTER SET binary
          FIELDS TERMINATED BY '\t'
          LINES TERMINATED BY '\r'
          (model_object);
Query OK, 1 row affected (34.96 sec)
```

```
Records: 1  Deleted: 0  Skipped: 0  Warnings: 0

mysql> LOAD DATA INFILE '/onnx_examples/x01'
          INTO TABLE model_table
          CHARACTER SET binary
          FIELDS TERMINATED BY '\t'
          LINES TERMINATED BY '\r'
          (model_object);
Query OK, 1 row affected (32.74 sec)
Records: 1  Deleted: 0  Skipped: 0  Warnings: 0

mysql> LOAD DATA INFILE '/onnx_examples/x02'
          INTO TABLE model_table
          CHARACTER SET binary
          FIELDS TERMINATED BY '\t'
          LINES TERMINATED BY '\r'
          (model_object);
Query OK, 1 row affected (11.90 sec)
Records: 1  Deleted: 0  Skipped: 0  Warnings: 0

mysql> SET @model_metadata = JSON_OBJECT('task','classification','onnx_outputs_info',JSON_OBJECT('predic

mysql> UPDATE mlcorpus.model_table
          SET model_metadata=@model_metadata
          WHERE chunk_id=1;

mysql> CALL sys.ML_MODEL_IMPORT(NULL, JSON_OBJECT('schema', 'mlcorpus', 'table', 'model_table'), @onnx_m
Query OK, 0 rows affected (18 min 7.29 sec)

mysql> CALL sys.ML_MODEL_LOAD(@onnx_model, NULL);
Query OK, 0 rows affected (6 min 51.37 sec)

mysql> SELECT COUNT(*)
          FROM ML_SCHEMA_root.model_object_catalog
          WHERE model_handle=@onnx_model;
+----------+
| COUNT(*) |
+----------+
|        3 |
+----------+
1 row in set (0.01 sec)

mysql> SELECT SUM(LENGTH(model_object))
          FROM ML_SCHEMA_root.model_object_catalog
          WHERE model_handle=@onnx_model;
+--------------------------+
| SUM(LENGTH(model_object)) |
+--------------------------+
|               2148494845 |
+--------------------------+
1 row in set (57.36 sec)
```

3.16.5 ML_PREDICT_ROW

`ML_PREDICT_ROW` generates predictions for one or more rows of unlabeled data specified in `JSON` format. Invoke `ML_PREDICT_ROW` with a `SELECT` statement.

`ML_PREDICT_ROW` requires a loaded model to run. See Section 3.14.3, “Loading Models”.

`ML_PREDICT_ROW` supports data drift detection for classification and regression models with the following:

- The *`options`* parameter includes the `additional_details` boolean value.
- The `ml_results` column includes the `drift` JSON object literal.

See: Section 3.14.11, “Data Drift Detection”.

ML_PREDICT_ROW Syntax

```
mysql> SELECT sys.ML_PREDICT_ROW(input_data, model_handle), [options]);

options: {
     JSON_OBJECT("key","value"["key","value"] ...)
          "key","value": {
          ['threshold', 'N']
          ['topk', 'N']
          ['recommend', {'ratings'|'items'|'users'|'users_to_items'|'items_to_users'|'items_to_items'|'users_t
          ['remove_seen', {'true'|'false'}]
          ['additional_details', {'true'|'false'}]
          }
}
```

A call to `ML_PREDICT_ROW` can include columns that were not present during `ML_TRAIN`. A table can include extra columns, and still use the HeatWave AutoML model. This allows side by side comparisons of target column labels, ground truth, and predictions in the same table. `ML_PREDICT_ROW` ignores any extra columns, and appends them to the results.

`ML_PREDICT_ROW` parameters:

- `input_data`: Specifies the data to generate predictions for.

  Specify a single row of data in `JSON` format:

  ```
  mysql> SELECT sys.ML_PREDICT_ROW(JSON_OBJECT("column_name", value,
            "column_name", value, ...),
            model_handle, options);
  ```

  To run `ML_PREDICT_ROW` on multiple rows of data, specify the columns as key-value pairs in `JSON` format and select from a table:

  ```
  mysql> SELECT sys.ML_PREDICT_ROW(JSON_OBJECT("output_col_name", schema.`input_col_name`,
            "output_col_name", schema.`input_col_name`, ...),
            model_handle, options)
            FROM input_table_name LIMIT N;
  ```

- `model_handle`: Specifies the model handle or a session variable that contains the model handle.
- *`options`*: A set of options in `JSON` format.

  This parameter only supports the `recommendation` and `anomaly_detection` tasks. For all other tasks, set this parameter to `NULL`.

  - `threshold`: The optional threshold for use with the `anomaly_detection` task to convert anomaly scores to `1`: an anomaly or `0`: normal. 0 < `threshold` < 1. The default value is (1 - `contamination`)-th percentile of all the anomaly scores.
  - `topk`: Use with the `recommendation` task to specify the number of recommendations to provide. A positive integer. The default is `3`.
  - `recommend`: Use with the `recommendation` task to specify what to recommend. Permitted values are:
    - `ratings`: Use this option to predict ratings. This is the default value.

      The target column is `prediction`, and the values are `float`.

      The input table must contain at least two columns with the same names as the user column and item column from the training model.

- `items`: Use this option to recommend items for users.

  The target column is `item_recommendation`, and the values are:

  ```
  JSON_OBJECT("column_item_id_name", JSON_ARRAY("item_1", ... , "item_k"),
              "column_rating_name" , JSON_ARRAY(rating_1, ..., rating_k))
  ```

  The input table must at least contain a column with the same name as the user column from the training model.

- `users`: Use this option to recommend users for items.

  The target column is `user_recommendation`, and the values are:

  ```
  JSON_OBJECT("column_user_id_name", JSON_ARRAY("user_1", ... , "user_k"),
              "column_rating_name" , JSON_ARRAY(rating_1, ..., rating_k))
  ```

  The input table must at least contain a column with the same name as the item column from the training model.

- `users_to_items`: This is the same as `items`.

- `items_to_users`: This is the same as `users`.

- `items_to_items`: Use this option to recommend similar items for items.

  The target column is `item_recommendation`, and the values are:

  ```
  JSON_OBJECT("column_item_id_name", JSON_ARRAY("item_1", ... , "item_k"))
  ```

  The input table must at least contain a column with the same name as the item column from the training model.

- `users_to_users`: Use this option to recommend similar users for users.

  The target column is `user_recommendation`, and the values are:

  ```
  JSON_OBJECT("column_user_id_name", JSON_ARRAY("user_1", ... , "user_k"))
  ```

  The input table must at least contain a column with the same name as the user column from the training model.

- `remove_seen`: If the input table overlaps with the training table, and `remove_seen` is `true`, then the model will not repeat existing interactions. The default is `true`. Set `remove_seen` to `false` to repeat existing interactions from the training table.

- `additional_details`: Set to `true` for `ml_results` to include the JSON object literal, `drift`.

Syntax Examples

- To run `ML_PREDICT_ROW` on a single row of data use a select statement. The results include the `ml_results` field, which uses `JSON` format:

```
mysql> SELECT sys.ML_PREDICT_ROW(JSON_OBJECT("sepal length", 7.3, "sepal width", 2.9,
          "petal length", 6.3, "petal width", 1.8), @iris_model, NULL);
+-----------------------------------------------------------------------------------------------------
| sys.ML_PREDICT_ROW('{"sepal length": 7.3, "sepal width": 2.9, "petal length": 6.3, "petal width": 1.8}
+-----------------------------------------------------------------------------------------------------
| {"Prediction": "Iris-virginica", "ml_results": "{'predictions': {'class': 'Iris-virginica'}, 'probabil
```

```
+-----------------------------------------------------------------------------------------------------------------
1 row in set (1.12 sec)
```

- To run ML_PREDICT_ROW on five rows of data selected from an input table:

```
mysql> SELECT sys.ML_PREDICT_ROW(JSON_OBJECT("sepal length", iris_test.`sepal length`,
          "sepal width", iris_test.`sepal width`, "petal length", iris_test.`petal length`,
          "petal width", iris_test.`petal width`), @iris_model, NULL)
          FROM ml_data.iris_test LIMIT 5;
```

See also:

- Section 3.9.2, “Using a Forecasting Model”
- Section 3.10.3, “Using an Anomaly Detection Model”
- Section 3.11.3, “Using a Recommendation Model”
- Section 3.14.11, “Data Drift Detection”

3.16.6 ML_PREDICT_TABLE

ML_PREDICT_TABLE generates predictions for an entire table of unlabeled data and saves the results to an output table. HeatWave AutoML performs the predictions in parallel.

ML_PREDICT_TABLE is a compute intensive process. Limiting operations to batches of 10 to 100 rows by splitting large tables into smaller tables is recommended.

A loaded model is required to run ML_PREDICT_TABLE. See Section 3.14.3, “Loading Models”.

The returned table includes a primary key:

- If the input table has a primary key, the output table will have the same primary key.
- If the input table does not have a primary key, the output table will have a new primary key column that auto increments.
  - As of MySQL 8.4.1, the name of the new primary key column is _4aad19ca6e_pk_id. The input table must not have a column with the name _4aad19ca6e_pk_id that is not a primary key.
  - Before MySQL 8.4.1, the name of the new primary key column is _id. The input table must not have a column with the name _id that is not a primary key.

The returned table includes the ml_results column which contains the prediction results and the data. The combination of results and data must be less than 65,532 characters.

ML_PREDICT_TABLE supports data drift detection for classification and regression models with the following:

- The *options* parameter includes the additional_details boolean value.
- The ml_results column includes the drift JSON object literal.

See: Section 3.14.11, “Data Drift Detection”.

ML_PREDICT_TABLE Syntax

```
mysql> CALL sys.ML_PREDICT_TABLE(table_name, model_handle, output_table_name), [options]);

options: {
     JSON_OBJECT("key","value"["key","value"] ...)
          "key","value": {
```

```
          ['threshold', 'N']
          ['topk', 'N']
          ['recommend', {'ratings'|'items'|'users'|'users_to_items'|'items_to_users'|'items_to_items'|'user
          ['remove_seen', {'true'|'false'}]
          ['batch_size', 'N']
          ['additional_details', {'true'|'false'}]
          }
}
```

A call to `ML_PREDICT_TABLE` can include columns that were not present during `ML_TRAIN`. A table can include extra columns, and still use the HeatWave AutoML model. This allows side by side comparisons of target column labels, ground truth, and predictions in the same table. `ML_PREDICT_TABLE` ignores any extra columns, and appends them to the results.

`ML_PREDICT_TABLE` parameters:

- `table_name`: Specifies the fully qualified name of the input table (*`schema_name.table_name`*). The input table should contain the same feature columns as the training dataset but no target column.
- `model_handle`: Specifies the model handle or a session variable containing the model handle
- `output_table_name`: Specifies the table where predictions are stored. The table is created if it does not exist. A fully qualified table name must be specified (*`schema_name.table_name`*). If the table already exists, an error is returned.
- *`options`*: A set of options in `JSON` format.

  This parameter supports the `recommendation` and `anomaly_detection` tasks. For all other tasks, set this parameter to `NULL`.

  - `threshold`: The optional threshold for use with the `anomaly_detection` task to convert anomaly scores to `1`: an anomaly or `0`: normal. 0 < `threshold` < 1. The default value is (1 - `contamination`)-th percentile of all the anomaly scores.
  - `topk`: The optional top K rows for use with the `anomaly_detection` and `recommendation` tasks. A positive integer between 1 and the table length.

    For the `anomaly_detection` task, the results include the top K rows with the highest anomaly scores. If `topk` is not set, `ML_PREDICT_TABLE` uses `threshold`.

    For an `anomaly_detection` task, do not set both `threshold` and `topk`. Use `threshold` or `topk`, or set *`options`* to `NULL`.

    For the `recommendation` task, the number of recommendations to provide. The default is `3`.

    A `recommendation` task with implicit feedback can use both `threshold` and `topk`.
  - `recommend`: Use with the `recommendation` task to specify what to recommend. Permitted values are:
    - `ratings`: Use this option to predict ratings. This is the default value.

      The target column is `prediction`, and the values are `float`.

      The input table must contain at least two columns with the same names as the user column and item column from the training model.
    - `items`: Use this option to recommend items for users.

The target column is `item_recommendation`, and the values are:

```
JSON_OBJECT("column_item_id_name", JSON_ARRAY("item_1", ... , "item_k"),
            "column_rating_name" , JSON_ARRAY(rating_1, ..., rating_k))
```

The input table must at least contain a column with the same name as the user column from the training model.

- `users`: Use this option to recommend users for items.

  The target column is `user_recommendation`, and the values are:

  ```
  JSON_OBJECT("column_user_id_name", JSON_ARRAY("user_1", ... , "user_k"),
              "column_rating_name" , JSON_ARRAY(rating_1, ..., rating_k))
  ```

  The input table must at least contain a column with the same name as the item column from the training model.

- `users_to_items`: This is the same as `items`.
- `items_to_users`: This is the same as `users`.
- `items_to_items`: Use this option to recommend similar items for items.

  The target column is `item_recommendation`, and the values are:

  ```
  JSON_OBJECT("column_item_id_name", JSON_ARRAY("item_1", ... , "item_k"))
  ```

  The input table must at least contain a column with the same name as the item column from the training model.

- `users_to_users`: Use this option to recommend similar users for users.

  The target column is `user_recommendation`, and the values are:

  ```
  JSON_OBJECT("column_user_id_name", JSON_ARRAY("user_1", ... , "user_k"))
  ```

  The input table must at least contain a column with the same name as the user column from the training model.

- `remove_seen`: If the input table overlaps with the training table, and `remove_seen` is `true`, then the model will not repeat existing interactions. The default is `true`. Set `remove_seen` to `false` to repeat existing interactions from the training table.
- `batch_size`: The size of each batch. 1 ≤ `batch_size` ≤ 1,000. The default is 1,000, and this provides the best results.
- `additional_details`: Set to `true` for `ml_results` to include the JSON object literal, `drift`.

Syntax Examples

- A typical usage example that specifies the fully qualified name of the table to generate predictions for, the session variable containing the model handle, and the fully qualified output table name:

```
mysql> CALL sys.ML_PREDICT_TABLE('ml_data.iris_test', @iris_model,
          'ml_data.iris_predictions', NULL);
```

To view ML_PREDICT_TABLE results, query the output table. The table shows the predictions and the feature column values used to make each prediction. The table includes the primary key, _4aad19ca6e_pk_id, and the ml_results column, which uses JSON format:

```
mysql> SELECT * from ml_data.iris_predictions LIMIT 5;
+------------------+--------------+-------------+--------------+-------------+-----------------+-------
| _4aad19ca6e_pk_id | sepal length | sepal width | petal length | petal width | class           | Predic
+------------------+--------------+-------------+--------------+-------------+-----------------+-------
|                1 |          7.3 |         2.9 |          6.3 |         1.8 | Iris-virginica  | Iris-v
|                2 |          6.1 |         2.9 |          4.7 |         1.4 | Iris-versicolor | Iris-v
|                3 |          6.3 |         2.8 |          5.1 |         1.5 | Iris-virginica  | Iris-v
|                4 |          6.3 |         3.3 |          4.7 |         1.6 | Iris-versicolor | Iris-v
|                5 |          6.1 |           3 |          4.9 |         1.8 | Iris-virginica  | Iris-v
+------------------+--------------+-------------+--------------+-------------+-----------------+-------
5 rows in set (0.00 sec)
```

See also:

- Section 3.9.2, “Using a Forecasting Model”
- Section 3.10.3, “Using an Anomaly Detection Model”
- Section 3.11.3, “Using a Recommendation Model”
- Section 3.15, “Progress tracking”.
- Section 3.14.11, “Data Drift Detection”

3.16.7 ML_EXPLAIN_ROW

The ML_EXPLAIN_ROW routine generates explanations for one or more rows of unlabeled data. ML_EXPLAIN_ROW is invoked using a SELECT statement.

ML_EXPLAIN_ROW limits explanations to the 100 most relevant features.

A loaded model, trained with a prediction explainer, is required to run ML_EXPLAIN_ROW. See Section 3.14.3, “Loading Models” and Section 3.16.2, “ML_EXPLAIN”.

ML_EXPLAIN_ROW does not support anomaly detection, and a call with an anomaly detection model will produce an error.

ML_EXPLAIN_ROW does not support recommendation models, and a call with a recommendation model will produce an error.

A call to ML_EXPLAIN_ROW can include columns that were not present during ML_TRAIN. A table can include extra columns, and still use the HeatWave AutoML model. This allows side by side comparisons of target column labels, ground truth, and explanations in the same table. ML_EXPLAIN_ROW ignores any extra columns, and appends them to the results.

ML_EXPLAIN_ROW Syntax

```
mysql> SELECT sys.ML_EXPLAIN_ROW(input_data, model_handle, [options]);

options: {
     JSON_OBJECT("key","value"[,"key","value"] ...)
          "key","value": {
          ['prediction_explainer', {'permutation_importance'|'shap'}|NULL]
          }
}
```

ML_EXPLAIN_ROW parameters:

- `input_data`: Specifies the data to generate explanations for. Data must be specified in `JSON` key-value format, where the key is a column name. The column names must match the feature column names in the table used to train the model. A single row of data can be specified as follows:

```
mysql> SELECT sys.ML_EXPLAIN_ROW(JSON_OBJECT("column_name", value, "column_name", value, ...)',
          model_handle, options);
```

  You can run `ML_EXPLAIN_ROW` on multiple rows of data by specifying the columns in `JSON` key-value format and selecting from an input table:

```
mysql> SELECT sys.ML_EXPLAIN_ROW(JSON_OBJECT("output_col_name", schema.`input_col_name`,
          output_col_name", schema.`input_col_name`, ...),
          model_handle, options)
          FROM input_table_name
          LIMIT N;
```

- `model_handle`: Specifies the model handle or a session variable containing the model handle.
- *`options`*: A set of options in `JSON` format. The available options are:
  - `prediction_explainer`: The name of the prediction explainer that you have trained for this model using `ML_EXPLAIN`. Valid values are:
    - `permutation_importance`: The default prediction explainer.
    - `shap`: The SHAP prediction explainer, which produces global feature importance values based on Shapley values.

Syntax Examples

- Run `ML_EXPLAIN_ROW` on a single row of data with the default Permutation Importance prediction explainer. The results include the `ml_results` field, which uses `JSON` format:

```
mysql> SELECT sys.ML_EXPLAIN_ROW(JSON_OBJECT("sepal length", 7.3, "sepal width", 2.9,
          "petal length", 6.3, "petal width", 1.8), @iris_model,
          JSON_OBJECT('prediction_explainer', 'permutation_importance'));
+-----------------------------------------------------------------------------------------------------------
| sys.ML_EXPLAIN_ROW(JSON_OBJECT("sepal length", 7.3, "sepal width", 2.9, "petal length", 6.3, "petal width"
| @iris_model, JSON_OBJECT('prediction_explainer', 'permutation_importance'))
+-----------------------------------------------------------------------------------------------------------
| {"Notes": "petal width (1.8) had the largest impact towards predicting Iris-virginica",
| "Prediction": "Iris-virginica", "ml_results": "{'attributions': {'petal length': 0.57, 'petal width': 0.73
| 'predictions': {'class': 'Iris-virginica'}, 'notes': 'petal width (1.8) had the largest impact towards
| predicting Iris-virginica'}", "petal width": 1.8, "sepal width": 2.9, "petal length": 6.3, "sepal length":
| "petal width_attribution": 0.73, "petal length_attribution": 0.57}
+-----------------------------------------------------------------------------------------------------------
1 row in set (5.92 sec)
```

- Run `ML_EXPLAIN_ROW` with the SHAP prediction explainer:

```
mysql> SELECT sys.ML_EXPLAIN_ROW(JSON_OBJECT('sepal length',`iris_test`.`sepal length`,
          'sepal width',`iris_test`.`sepal width`,'petal length',`iris_test`.`petal length`,
          'petal width',`iris_test`.`petal width` ), @iris_model,
          JSON_OBJECT('prediction_explainer', 'shap'))
          FROM `iris_test`
          LIMIT 4;
```

3.16.8 ML_EXPLAIN_TABLE

`ML_EXPLAIN_TABLE` explains predictions for an entire table of unlabeled data and saves results to an output table.

ML_EXPLAIN_TABLE is a compute intensive process. Limiting operations to batches of 10 to 100 rows by splitting large tables into smaller tables is recommended.

ML_EXPLAIN_TABLE limits explanations to the 100 most relevant features.

A loaded model, trained with a prediction explainer, is required to run ML_EXPLAIN_TABLE. See Section 3.14.3, “Loading Models” and Section 3.16.2, “ML_EXPLAIN”.

The returned table includes a primary key:

- If the input table has a primary key, the output table will have the same primary key.
- If the input table does not have a primary key, the output table will have a new primary key column that auto increments.
  - As of MySQL 8.4.1, the name of the new primary key column is _4aad19ca6e_pk_id. The input table must not have a column with the name _4aad19ca6e_pk_id that is not a primary key.
  - Before MySQL 8.4.1, the name of the new primary key column is _id. The input table must not have a column with the name _id that is not a primary key.

ML_EXPLAIN_TABLE does not support anomaly detection, and a call with an anomaly detection model will produce an error.

ML_EXPLAIN_TABLE does not support recommendation models, and a call with a recommendation model will produce an error.

A call to ML_EXPLAIN_TABLE can include columns that were not present during ML_TRAIN. A table can include extra columns, and still use the HeatWave AutoML model. This allows side by side comparisons of target column labels, ground truth, and explanations in the same table. ML_EXPLAIN_TABLE ignores any extra columns, and appends them to the results.

ML_EXPLAIN_TABLE Syntax

```
mysql> CALL sys.ML_EXPLAIN_TABLE(table_name, model_handle, output_table_name, [options]);

options: {
     JSON_OBJECT("key","value"[,"key","value"] ...)
          "key","value": {
          ['prediction_explainer', {'permutation_importance'|'shap'}|NULL]
          ['batch_size', 'N']
          }
}
```

ML_EXPLAIN_TABLE parameters:

- table_name: Specifies the fully qualified name of the input table (*schema_name.table_name*). The input table should contain the same feature columns as the table used to train the model but no target column.
- model_handle: Specifies the model handle or a session variable containing the model handle.
- output_table_name: Specifies the table where explanation data is stored. The table is created if it does not exist. A fully qualified table name must be specified (*schema_name.table_name*). If the table already exists, an error is returned.
- *options*: A set of options in JSON format. The available options are:
  - prediction_explainer: The name of the prediction explainer that you have trained for this model using ML_EXPLAIN. Valid values are:

  - `permutation_importance`: The default prediction explainer.
  - `shap`: The SHAP prediction explainer, which produces global feature importance values based on Shapley values.
- `batch_size`: The size of each batch. 1 ≤ `batch_size` ≤ 100. The default is 100, and this provides the best results.

Syntax Examples

- The following example generates explanations for a table of data with the default Permutation Importance prediction explainer. The `ML_EXPLAIN_TABLE` call specifies the fully qualified name of the table to generate explanations for, the session variable containing the model handle, and the fully qualified output table name.

```
mysql> CALL sys.ML_EXPLAIN_TABLE('ml_data.iris_test', @iris_model,
          'ml_data.iris_explanations',
          JSON_OBJECT('prediction_explainer', 'permutation_importance'));
```

  To view `ML_EXPLAIN_TABLE` results, query the output table. The `SELECT` statement retrieves explanation data from the output table. The table includes the primary key, `_4aad19ca6e_pk_id`, and the `ml_results` column, which uses `JSON` format:

```
mysql> SELECT * FROM ml_data.iris_explanations LIMIT 5;
+-------------------+--------------+-------------+--------------+-------------+-----------------+-----------
| _4aad19ca6e_pk_id | sepal length | sepal width | petal length | petal width | class           | Prediction
+-------------------+--------------+-------------+--------------+-------------+-----------------+-----------
|                 1 |          7.3 |         2.9 |          6.3 |         1.8 | Iris-virginica  | Iris-virgi
|                 2 |          6.1 |         2.9 |          4.7 |         1.4 | Iris-versicolor | Iris-versi
|                 3 |          6.3 |         2.8 |          5.1 |         1.5 | Iris-virginica  | Iris-versi
|                 4 |          6.3 |         3.3 |          4.7 |         1.6 | Iris-versicolor | Iris-versi
|                 5 |          6.1 |           3 |          4.9 |         1.8 | Iris-virginica  | Iris-virgi
+-------------------+--------------+-------------+--------------+-------------+-----------------+-----------
5 rows in set (0.00 sec)
```

- Run `ML_EXPLAIN_TABLE` with the SHAP prediction explainer:

```
mysql> CALL sys.ML_EXPLAIN_TABLE('ml_data.`iris_test_temp`', @model,
          'ml_data.`iris_explanations`',
          JSON_OBJECT('prediction_explainer', 'shap'));
```

See also: Section 3.15, “Progress tracking”.

3.16.9 ML_SCORE

`ML_SCORE` scores a model by generating predictions using the feature columns in a labeled dataset as input and comparing the predictions to ground truth values in the target column of the labeled dataset. The dataset used with `ML_SCORE` should have the same feature columns as the dataset used to train the model but the data should be different; for example, you might reserve 20 to 30 percent of the labeled training data for scoring.

`ML_SCORE` returns a computed metric indicating the quality of the model.

ML_SCORE Syntax

```
mysql> CALL sys.ML_SCORE(table_name, target_column_name, model_handle, metric, score, [options]);

options: {
     JSON_OBJECT("key","value"[,"key","value"] ...)
          "key","value": {
```

```
          ['threshold', 'N']
          ['topk', 'N']
          ['remove_seen', {'true'|'false'}]
          }
}
```

`ML_SCORE` parameters:

- `table_name`: Specifies the fully qualified name of the table used to compute model quality (*`schema_name.table_name`*). The table must contain the same columns as the training dataset.
- `target_column_name`: Specifies the name of the target column containing ground truth values.

  Forecasting does not require `target_column_name`, and it can be set to `NULL`.
- `model_handle`: Specifies the model handle or a session variable containing the model handle.
- `metric`: Specifies the name of the metric. See Section 3.16.14, “Optimization and Scoring Metrics”.
- `score`: Specifies the user-defined variable name for the computed score. The `ML_SCORE` routine populates the variable. User variables are written as *`@var_name`*. The examples in this guide use `@score` as the variable name. Any valid name for a user-defined variable is permitted, for example `@my_score`.
- *`options`*: A set of options in `JSON` format. This parameter only supports the `anomaly detection` and `recommendation` tasks. For all other tasks, set this parameter to `NULL`.
  - `threshold`: The optional threshold for use with the `anomaly_detection` and `recommendation` tasks.

    Use with the `anomaly_detection` task to convert anomaly scores to `1`: an anomaly or `0`: normal. 0 < `threshold` < 1. The default value is (1 - `contamination`)-th percentile of all the anomaly scores.

    Use with the `recommendation` task and ranking metrics to define positive feedback, and a relevant sample. All rankings at or above the `threshold` are implied to provide positive feedback. All rankings below the `threshold` are implied to provide negative feedback. The default value is 1.
  - `topk`: The optional top K rows for use with the `anomaly_detection` and `recommendation` tasks. A positive integer between 1 and the table length.

    For the `anomaly_detection` task the results include the top K rows with the highest anomaly scores. It is an integer between 1 and the table length. If `topk` is not set, `ML_SCORE` uses `threshold`.

    For an `anomaly_detection` task, do not set both `threshold` and `topk`. Use `threshold` or `topk`, or set *`options`* to `NULL`.

    For the `recommendation` task and ranking metrics, the number of recommendations to provide. The default is `3`.

    A `recommendation` task and ranking metrics can use both `threshold` and `topk`.
  - `remove_seen`: If the input table overlaps with the training table, and `remove_seen` is `true`, then the model will not repeat existing interactions. The default is `true`. Set `remove_seen` to `false` to repeat existing interactions from the training table.

Syntax Example

- The following example runs `ML_SCORE` on the `ml_data.iris_train` table to determine model quality:

```
mysql> CALL sys.ML_SCORE('ml_data.iris_validate', 'class', @iris_model,
          'balanced_accuracy', @score, NULL);

mysql> SELECT @score;
+--------------------+
| @score             |
+--------------------+
| 0.9583333134651184 |
+--------------------+
```

See also:

- Section 3.9.2, “Using a Forecasting Model”
- Section 3.10.3, “Using an Anomaly Detection Model”
- Section 3.11.3, “Using a Recommendation Model”

3.16.10 ML_MODEL_LOAD

The `ML_MODEL_LOAD` routine loads a model from the model catalog. A model remains loaded until the model is unloaded using the `ML_MODEL_UNLOAD` routine or until HeatWave AutoML is restarted by a HeatWave Cluster restart.

MySQL 9.0.0 introduces support for large models that changes how HeatWave AutoML stores models, see: Section 3.14.1, “The Model Catalog”. `ML_MODEL_LOAD` upgrades older models.

Use `ML_MODEL_ACTIVE` to check which models are active for which users. All active users and models share the amount of memory defined by the shape, and it might be necessary to schedule users. `ML_MODEL_LOAD` will raise an error if there are memory limitations.

To share models with other users, see: Section 3.14.10, “Sharing Models”.

ML_MODEL_LOAD Syntax

```
mysql> CALL sys.ML_MODEL_LOAD(model_handle, user);
```

`ML_MODEL_LOAD` parameters:

- `model_handle`: Specifies the model handle or a session variable containing the model handle. For how to look up a model handle, see Section 3.14.8, “Model Handles”.
- `user`: The MySQL user name of the model owner. Specify `NULL` if the model owner is the current user.

  This parameter is ignored, and it can be set to `NULL`.

Syntax Examples

- An `ML_MODEL_LOAD` call with `NULL` specified, indicating that the model belongs to the user executing the `ML_MODEL_LOAD` call:

  ```
  mysql> CALL sys.ML_MODEL_LOAD('ml_data.iris_train_user1_1636729526', NULL);
  ```

- An `ML_MODEL_LOAD` call that specifies a session variable containing the model handle:

  ```
  mysql> CALL sys.ML_MODEL_LOAD(@iris_model, NULL);
  ```

- An `ML_MODEL_LOAD` call that specifies the model owner:

  ```
  mysql> CALL sys.ML_MODEL_LOAD('ml_data.iris_train_user1_1636729526', user1);
  ```

3.16.11 ML_MODEL_UNLOAD

`ML_MODEL_UNLOAD` unloads a model from HeatWave AutoML.

As of MySQL 9.0.0, `ML_MODEL_UNLOAD` does not check whether the model specified is in the model catalog. If it is not, `ML_MODEL_UNLOAD` will succeed, but will not unload any model. Use `ML_MODEL_ACTIVE` to check which models are active, and owned by the user.

ML_MODEL_UNLOAD Syntax

```
mysql> CALL sys.ML_MODEL_UNLOAD(model_handle);
```

`ML_MODEL_UNLOAD` parameters:

- `model_handle`: Specifies the model handle.

Syntax Examples

- An `ML_MODEL_UNLOAD` call that specifies the model handle:

  ```
  mysql> CALL sys.ML_MODEL_UNLOAD('ml_data.iris_train_user1_1636729526');
  ```

- An `ML_MODEL_UNLOAD` call that specifies a session variable containing the model handle:

  ```
  mysql> CALL sys.ML_MODEL_UNLOAD(@iris_model);
  ```

3.16.12 ML_MODEL_ACTIVE

MySQL 9.0.0 introduces the `ML_MODEL_ACTIVE` routine. Use this routine to check which models are active for which users. All active users and models share the amount of memory defined by the shape, and it might be necessary to schedule users.

ML_MODEL_ACTIVE Syntax

As of MySQL 9.0.0, `ML_MODEL_ACTIVE` can report which models are active for which users.

```
mysql> CALL sys.ML_MODEL_ACTIVE (user, model_info);
```

`ML_MODEL_ACTIVE` parameters:

- `user`: The user to provide information for. Set to `current` or `all` or `NULL`. `NULL` is equivalent to `current`.
- `model_info`: The name of the JSON array that will contain the active user and model information. It will contain two JSON object literals:
  - A JSON object literal that will contain the following :
    - Key: `total model size(bytes)`
    - Value:
      - If `user`: is set to `current`: The sum of model sizes for the current user.
      - If `user`: is set to `all`: The sum of model sizes for all users.
  - A JSON object literal that will contain the following :
    - Key:

    - If `user`: is set to `current`: The model handle for an active model owned by the current user.
    - If `user`: is set to `all`: The name of a user who has active models.
  - Value:
    - If `user`: is set to `current`: The `model_metadata` for this model.
    - If `user`: is set to `all`: A list of JSON object literals:
      - Key: The model handle.
      - Value: Brief model metadata for that model.

Syntax Examples

- *`user1`* checks their own models:

```
mysql> CALL sys.ML_MODEL_ACTIVE('current', @model_info);
Query OK, 0 rows affected (0.10 sec)

mysql> SELECT JSON_PRETTY(@model_info);
+------------------------------------------------------------+
| JSON_PRETTY(@model_info)                                   |
+------------------------------------------------------------+
| [                                                          |
  {                                                          |
    "total model size(bytes)": 348954                        |
  },                                                         |
  {                                                          |
    "iris_export_user1": {                                   |
      "task": "classification",                              |
      "notes": "",                                           |
      "chunks": 1,                                           |
      "format": "HWMLv2.0",                                  |
      "n_rows": 120,                                         |
      "status": "Ready",                                     |
      "options": {                                           |
        "model_explainer": "permutation_importance, shap",   |
        "prediction_explainer": "shap"                       |
      },                                                     |
      "n_columns": 4,                                        |
      "pos_class": null,                                     |
      "column_names": [                                      |
        "sepal length",                                      |
        "sepal width",                                       |
        "petal length",                                      |
        "petal width"                                        |
      ],                                                     |
      "contamination": null,                                 |
      "model_quality": "high",                               |
      "training_time": 18.363686,                            |
      "algorithm_name": "ExtraTreesClassifier",              |
      "training_score": -0.10970368035588404,                |
      "build_timestamp": 1697524180,                         |
      "n_selected_rows": 96,                                 |
      "training_params": {                                   |
        "sp_arr": null,                                      |
        "timezone": null,                                    |
        "recommend": "ratings",                              |
        "force_use_X": false,                                |
        "recommend_k": 3,                                    |
        "remove_seen": true,                                 |
        "contamination": null,                               |
```

```
      "feedback_threshold": 1                                  |
    },                                                         |
    "train_table_name": "mlcorpus.iris_train",                 |
    "model_explanation": {                                     |
      "shap": {                                                |
        "petal width": 0.3139,                                 |
        "sepal width": 0.0296,                                 |
        "petal length": 0.2787,                                |
        "sepal length": 0.0462                                 |
      },                                                       |
      "permutation_importance": {                              |
        "petal width": 0.2301,                                 |
        "sepal width": 0.0056,                                 |
        "petal length": 0.2192,                                |
        "sepal length": 0.0056                                 |
      }                                                        |
    },                                                         |
    "model_object_size": 348954,                               |
    "n_selected_columns": 4,                                   |
    "target_column_name": "class",                             |
    "optimization_metric": "neg_log_loss",                     |
    "selected_column_names": [                                 |
      "petal length",                                          |
      "petal width",                                           |
      "sepal length",                                          |
      "sepal width"                                            |
    ]                                                          |
  }                                                            |
 }                                                             |
] |                                                           |
+-----------------------------------------------------------+
1 row in set (0.00 sec)
```

- *user1* checks their own models, and extracts specific information:

```
mysql> CALL sys.ML_MODEL_ACTIVE('current', @model_info);
Query OK, 0 rows affected (0.12 sec)

mysql> SELECT JSON_KEYS(JSON_EXTRACT(@model_info, '$[1]'));
+-------------------------------------------------------------+
| JSON_KEYS(JSON_EXTRACT(@model_info, '$[1]'))                |
+-------------------------------------------------------------+
| ["iris_export", "mlcorpus.iris_train_user1_1697524152037"] |
+-------------------------------------------------------------+
1 row in set (0.00 sec)

mysql> SELECT JSON_EXTRACT(@model_info, '$[0]');
+-----------------------------------+
| JSON_EXTRACT(@model_info, '$[0]')    |
+-----------------------------------+
| {"total model size(bytes)":  697908} |
+-----------------------------------+
1 row in set (0.01 sec)
```

- *user1* checks the models for all users:

```
mysql> CALL sys.ML_MODEL_ACTIVE('all', @model_info);
Query OK, 0 rows affected (0.11 sec)

mysql> SELECT JSON_PRETTY(@model_info);
+-----------------------------------------------------+
| JSON_PRETTY(@model_info)                            |
+-----------------------------------------------------+
| [                                                   |
  {                                                   |
    "total model size(bytes)": 1046862                |
  },                                                  |
```

```
  {                                                    |
    "user2": [                                         |
      {                                                |
        "iris_export_user2": {                         |
          "format": "HWMLv2.0",                        |
          "model_size(byte)": 348954                   |
        }                                              |
      }                                                |
    ],                                                 |
    "user1": [                                         |
      {                                                |
        "mlcorpus.iris_train_user1_1697524152037": {   |
          "format": "HWMLv2.0",                        |
          "model_size(byte)": 348954                   |
        }                                              |
      },                                               |
      {                                                |
        "iris_export": {                               |
          "format": "HWMLv2.0",                        |
          "model_size(byte)": 348954                   |
        }                                              |
      }                                                |
    ]                                                  |
  }                                                    |
] |                                                    |
+-----------------------------------------------------+
1 row in set (0.00 sec)
```

- *`user2`* checks the models for all users:

```
mysql> CALL sys.ML_MODEL_ACTIVE('all', @model_info);
Query OK, 0 rows affected (0.10 sec)

mysql> SELECT JSON_PRETTY(@model_info)
+-----------------------------------------------------+
| JSON_PRETTY(@model_info)                            |
+-----------------------------------------------------+
| [                                                   |
  {                                                    |
    "total model size(bytes)": 1046862                 |
  },                                                   |
  {                                                    |
    "user2": [                                         |
      {                                                |
        "iris_export_user2": {                         |
          "format": "HWMLv2.0",                        |
          "model_size(byte)": 348954                   |
        }                                              |
      }                                                |
    ],                                                 |
    "user1": [                                         |
      {                                                |
        "mlcorpus.iris_train_user1_1697524152037": {   |
          "format": "HWMLv2.0",                        |
          "model_size(byte)": 348954                   |
        }                                              |
      },                                               |
      {                                                |
        "iris_export": {                               |
          "format": "HWMLv2.0",                        |
          "model_size(byte)": 348954                   |
        }                                              |
      }                                                |
    ]                                                  |
  }                                                    |
] |                                                    |
```

```
+-------------------------------------------------+
1 row in set (0.01 sec)
```

3.16.13 Model Types

HeatWave AutoML supports the following training models. When training a model, use the `ML_TRAIN` `model_list` and `exclude_model_list` options to specify the training models to consider or exclude. Section 3.14.1.3, “Model Metadata” includes the `algorithm_name` field which defines the model type.

Classification Models

- LogisticRegression
- GaussianNB
- DecisionTreeClassifier
- RandomForestClassifier
- XGBClassifier
- LGBMClassifier
- SVC
- LinearSVC
- ExtraTreesClassifier

Regression Models

- DecisionTreeRegressor
- RandomForestRegressor
- LinearRegression
- LGBMRegressor
- XGBRegressor
- SVR
- LinearSVR
- ExtraTreesRegressor

Forecasting Models

Univariate endogenous models:

- NaiveForecaster
- ThetaForecaster
- ExpSmoothForecaster
- ETSForecaster
- `STLwESForecaster`: STLForecast with ExponentialSmoothing substructure

- `STLwARIMAForecaster`: STLForecast with ARIMA substructure

Univariate endogenous with exogenous models:

- SARIMAXForecaster

Multivariate endogenous with exogenous models:

- VARMAXForecaster

Univariate or multivariate endogenous with exogenous models:

- DynFactorForecaster

Anomaly Detection Models

The only model is GkNN. See: Section 3.10, “Anomaly Detection”.

Recommendation Models

Recommendation models that rate users or items to use with explicit feedback:

- Baseline
- CoClustering
- NormalPredictor
- SlopeOne
- Matrix factorization models:
  - SVD
  - SVDpp
  - NMF

Recommendation models that rank users or items to use with implicit feedback:

- BPR: Bayesian Personalized Ranking from Implicit Feedback
- CTR: Collaborative Topic Regression

3.16.14 Optimization and Scoring Metrics

The `ML_TRAIN` routine includes the `optimization_metric` option, and the `ML_SCORE` routine includes the `metric` option. Both of these options define a metric that must be compatible with the `task` type and the target data. Section 3.14.1.3, “Model Metadata” includes the `optimization_metric` field.

For more information about scoring metrics, see: scikit-learn.org. For more information about forecasting metrics, see: sktime.org and statsmodels.org.

Classification Metrics

Binary-only metrics:

- f1
- precision

- recall
- roc_auc

Binary and multi-class metrics:

- accuracy
- balanced_accuracy (`ML_SCORE` only)
- f1_macro
- f1_micro
- f1_samples (`ML_SCORE` only)
- f1_weighted
- neg_log_loss
- precision_macro
- precision_micro
- precision_samples (`ML_SCORE` only)
- precision_weighted
- recall_macro
- recall_micro
- recall_samples (`ML_SCORE` only)
- recall_weighted

Regression Metrics

- neg_mean_absolute_error
- neg_mean_squared_error
- neg_mean_squared_log_error
- neg_median_absolute_error
- r2

Forecasting Metrics

- neg_max_absolute_error
- neg_mean_absolute_error
- neg_mean_abs_scaled_error
- neg_mean_squared_error
- neg_root_mean_squared_error
- neg_root_mean_squared_percent_error

- neg_sym_mean_abs_percent_error

Anomaly Detection Metrics

Metrics for anomaly detection can only be used with the `ML_SCORE` routine. They cannot be used with the `ML_TRAIN` routine.

- roc_auc: You must not specify `threshold` or `topk` options.
- `precision_k`: An Oracle implementation of a common metric for fraud detection and lead scoring. You must use the `topk` option. You cannot use the `threshold` option.

The following metrics can use the `threshold` option, but cannot use the `topk` option:

- accuracy
- balanced_accuracy
- f1
- neg_log_loss
- precision
- recall

Recommendation Model Metrics

The following rating metrics can be used for explicit feedback:

- neg_mean_absolute_error
- neg_mean_squared_error
- neg_root_mean_squared_error
- r2

For recommendation models that use implicit feedback:

- If a user and item combination in the input table is not unique, the input table is grouped by user and item columns, and the result is the average of the rankings.
- If the input table overlaps with the training table, and `remove_seen` is `true`, which is the default setting, then the model will not repeat a recommendation and it ignores the overlap items.

The following ranking metrics can be used for implicit and explicit feedback:

- `precision_at_k` is the number of relevant `topk` recommended items divided by the total `topk` recommended items for a particular user:

  `precision_at_k` = (relevant `topk` recommended items) / (total `topk` recommended items)

  For example, if 7 out of 10 items are relevant for a user, and `topk` is 10, then `precision_at_k` is 70%.

  The `precision_at_k` value for the input table is the average for all users. If `remove_seen` is `true`, the default setting, then the average only includes users for whom the model can make a recommendation. If a user has implicitly ranked every item in the training table, the model cannot recommend any more items for that user, and they are ignored from the average calculation if `remove_seen` is `true`.

- `recall_at_k` is the number of relevant `topk` recommended items divided by the total relevant items for a particular user:

  `recall_at_k` = (relevant `topk` recommended items) / (total relevant items)

  For example, there is a total of 20 relevant items for a user. If `topk` is 10, and 7 of those items are relevant, then `recall_at_k` is 7 / 20 = 35%.

  The `recall_at_k` value for the input table is the average for all users.

- `hit_ratio_at_k` is the number of relevant `topk` recommended items divided by the total relevant items for all users:

  `hit_ratio_at_k` = (relevant `topk` recommended items, all users) / (total relevant items, all users)

  The average of `hit_ratio_at_k` for the input table is `recall_at_k`. If there is only one user, `hit_ratio_at_k` is the same as `recall_at_k`.

- `ndcg_at_k` is normalized discounted cumulative gain, which is the discounted cumulative gain of the relevant `topk` recommended items divided by the discounted cumulative gain of the relevant `topk` items for a particular user.

  The discounted gain of an item is the true rating divided by $\log_2(r+1)$ where `r` is the ranking of this item in the relevant `topk` items. If a user prefers a particular item, the rating is higher, and the ranking is lower.

  The `ndcg_at_k` value for the input table is the average for all users.

3.17 Supported Data Types

HeatWave AutoML supports the following data types:

- `FLOAT`
- `DOUBLE`
- `INT`
- `TINYINT`
- `SMALLINT`
- `MEDIUMINT`
- `BIGINT`
- `INT UNSIGNED`
- `TINYINT UNSIGNED`
- `SMALLINT UNSIGNED`
- `MEDIUMINT UNSIGNED`
- `BIGINT UNSIGNED`
- `VARCHAR`
- `CHAR`

- `DATE`.
- `TIME`.
- `DATETIME`.
- `TIMESTAMP`.
- `YEAR`.
- `TINYTEXT`.
- `TEXT`.
- `MEDIUMTEXT`.
- `LONGTEXT`.

HeatWave AutoML uses TfidfVectorizer to pre-process `TINYTEXT`, `TEXT`, `MEDIUMTEXT`, and `LONGTEXT` and appends the results to the data set. HeatWave AutoML has the following limitations for text usage:

- The `ML_PREDICT_TABLE` `ml_results` column contains the prediction results and the data. This combination must be less than 65,532 characters.
- HeatWave AutoML only supports datasets in the English language.
- HeatWave AutoML does not support text columns with `NULL` values.
- HeatWave AutoML does not support a text target column.
- HeatWave AutoML does not support recommendation tasks with a text column.
- For the forecasting task, `endogenous_variables` cannot be text.

`DECIMAL` data type columns are not supported. Remove them or convert them to `FLOAT`.

3.18 HeatWave AutoML Error Messages

Each error message includes an error number, SQLSTATE value, and message string, as described in Error Message Sources and Elements.

- Error number: `ML001016`; SQLSTATE: `HY000`

  Message: Only classification, regression, and forecasting tasks are supported.

  Example: `ERROR HY000: ML001016: Only classification, regression, and forecasting tasks are supported.`

  Check the `task` option in the `ML_TRAIN` call to ensure that it is specified correctly.

- Error number: `ML001031`; SQLSTATE: `HY000`

  Message: Running as a classification task. % classes have less than % samples per class, and cannot be trained on. Maybe it should be trained as a regression task instead of a classification task. Or the task ran on the default setting - classification, due to an incorrect JSON task argument.

  Example: `ERROR HY000: ML001031: Running as a classification task. 189 classes have less than 5 samples per class, and cannot be trained on. Maybe it should`

be trained as a regression task instead of a classification task. Or the task ran on the default setting - classification, due to an incorrect JSON task argument.

If a classification model is intended, add more samples to the data to increase the minority class count; that is, add more rows with the under-represented target column value. If a classification model was not intended, run ML_TRAIN with the regression task option.

- Error number: ML001051; SQLSTATE: HY000

  Message: One or more rows contain all NaN values. Imputation is not possible on such rows.

  Example: ERROR HY000: ML001051: One or more rows contain all NaN values. Imputation is not possible on such rows.

  MySQL does not support NaN values. Replace with NULL.

- Error number: ML001052; SQLSTATE: HY000

  Message: All columns are dropped. They are constant, mostly unique, or have a lot of missing values!

  Example: ERROR HY000: ML001052: All columns are dropped. They are constant, mostly unique, or have a lot of missing values!

  ML_TRAIN ignores columns with certain characteristics such as columns missing more than 20% of values and columns containing the same single value. See Section 3.4, “Preparing Data”.

- Error number: ML001053; SQLSTATE: HY000

  Message: Unlabeled samples detected in the training data. (Values in target column can not be NULL).

  Example: ERROR HY000: ML001053: Unlabeled samples detected in the training data. (Values in target column can not be NULL).

  Training data must be labeled. See Section 3.4, “Preparing Data”.

- Error number: ML003000; SQLSTATE: HY000

  Message: Number of offloaded datasets has reached the limit!

  Example: ERROR HY000: ML003000: Number of offloaded datasets has reached the limit!

- Error number: ML003011; SQLSTATE: HY000

  Message: Columns of provided data need to match those used for training. Provided - ['%', '%', '%'] vs Trained - ['%', '%'].

  Example: ERROR HY000: ML003011: Columns of provided data need to match those used for training. Provided - ['petal length', 'petal width', 'sepal length', 'sepal width'] vs Trained - ['petal length', 'sepal length', 'sepal width'].

  The input data columns do not match the columns of training dataset used to train the model. Compare the input data to the training data to identify the discrepancy.

- Error number: ML003012; SQLSTATE: HY000

  Message: The table (%) is NULL or has not been loaded.

Example: `ERROR HY000: ML003012: The table (mlcorpus.iris_train) is NULL or has not been loaded.`

There is no data in the specified table.

- Error number: `ML003014`; SQLSTATE: `HY000`

  Message: The size of model generated is larger than the maximum allowed.

  Example: `ERROR HY000: ML003014: The size of model generated is larger than the maximum allowed.`

  Models greater than 900 MB in size are not supported.

- Error number: `ML003015`; SQLSTATE: `HY000`

  Message: The input column types do not match the column types of dataset which the model was trained on. ['%', '%'] vs ['%', '%'].

  Example: `ERROR HY000: ML003015: The input column types do not match the column types of dataset which the model was trained on. ['numerical', 'numerical', 'categorical', 'numerical'] vs ['numerical', 'numerical', 'numerical', 'numerical'].`

- Error number: `ML003016`; SQLSTATE: `HY000`

  Message: Missing argument \"row_json\" in input JSON -> dict_keys(['%', '%']).

  Example: `ERROR HY000: ML003016: Missing argument \"row_json\" in input JSON -> dict_keys(['operation', 'user_name', 'table_name', 'schema_name', 'model_handle']).`

- Error number: `ML003017`; SQLSTATE: `HY000`

  Message: The corresponding value of row_json should be a string!

  Example: `ERROR HY000: ML003017: The corresponding value of row_json should be a string!`

- Error number: `ML003018`; SQLSTATE: `HY000`

  Message: The corresponding value of row_json is NOT a valid JSON!

  Example: `ERROR HY000: ML003018: The corresponding value of row_json is NOT a valid JSON!`

- Error number: `ML003019`; SQLSTATE: `HY000`

  Message: Invalid data for the metric (%). Score could not be computed.

  Example: `ERROR HY000: ML003019: Invalid data for the metric (roc_auc). Score could not be computed.`

  The scoring metric is legal and supported, but the data provided is not suitable to calculate such a score. For example: ROC_AUC for multi-class classification. Try a different scoring metric.

- Error number: `ML003020`; SQLSTATE: `HY000`

Message: Unsupported scoring function (%) for current task (%).

Example: `ERROR HY000: ML003020: Unsupported scoring function (accuracy) for current task (regression).`

The scoring metric is legal and supported, but the task provided is not suitable to calculate such a score; for example: Using the `accuracy` metric for a `regression` model.

- Error number: `ML003021`; SQLSTATE: `HY000`

  Message: Cannot train a regression task with a non-numeric target column.

  Example: `ERROR HY000: ML003021: Cannot train a regression task with a non-numeric target column.`

  `ML_TRAIN` was run with the regression task type on a training dataset with a non-numeric target column. Regression models require a numeric target column.

- Error number: `ML003022`; SQLSTATE: `HY000`

  Message: At least 2 target classes are needed for classification task

  Example: `ERROR HY000: ML003022: At least 2 target classes are needed for classification task.`

  `ML_TRAIN` was run with the classification task type on a training dataset where the target column did not have at least two possible values.

- Error number: `ML003023`; SQLSTATE: `3877 (HY000)`

  Message: Unknown option given. Allowed options for training are: ['task', 'model_list', 'exclude_model_list', 'optimization_metric', 'exclude_column_list', 'datetime_index', 'endogenous_variables', 'exogenous_variables', 'positive_class', 'users', 'items', 'user_columns', 'item_columns'].

  Example: `ERROR 3877 (HY000): ML003023: Unknown option given. Allowed options for training are: ['task', 'model_list', 'exclude_model_list', 'optimization_metric', 'exclude_column_list', 'datetime_index', 'endogenous_variables', 'exogenous_variables', 'positive_class', 'users', 'items', 'user_columns', 'item_columns'].`

  The `ML_TRAIN` call specified an unknown option.

- Error number: `ML003024`; SQLSTATE: `HY000`

  Message: Not enough available memory, unloading any RAPID tables will help to free up memory.

  Example: `ERROR HY000: ML003024: Not enough available memory, unloading any RAPID tables will help to free up memory.`

  There is not enough memory on the HeatWave Cluster to perform the operation. Try unloading data that was loaded for analytics to free up space.

  The recommended node shape for HeatWave AutoML functionality is HeatWave.256GB. The HeatWave.16GB node shape might not have enough memory to train the model with large data sets. If this error message appears with the smaller node shape and HeatWave AutoML, use the larger shape instead.

- Error number: `ML003027`; SQLSTATE: `3877 (HY000)`

  Message: JSON attribute (item_columns) must be in JSON_ARRAY type.

  Example: `ERROR 3877 (HY000): ML003027: JSON attribute (item_columns) must be in JSON_ARRAY type.`

  Specify the `item_columns` JSON attribute as a JSON array.

- Error number: `ML003027`; SQLSTATE: `3877 (HY000)`

  Message: JSON attribute (user_columns) must be in JSON_ARRAY type.

  Example: `ERROR 3877 (HY000): ML003027: JSON attribute (user_columns) must be in JSON_ARRAY type.`

  Specify the `user_columns` JSON attribute as a JSON array.

- Error number: `ML003039`; SQLSTATE: `HY000`

  Message: Not all user specified columns are present in the input table - missing columns are {%}.

  Example: `ERROR HY000: ML003039: Not all user specified columns are present in the input table - missing columns are {C4}.`

  The syntax includes a column that is not available.

- Error number: `ML003047`; SQLSTATE: `HY000`

  Message: All columns cannot be excluded. User provided exclude_column_list is ['%', '%'].

  Example: `ERROR HY000: ML003047: All columns cannot be excluded. User provided exclude_column_list is ['C0', 'C1', 'C2', 'C3'].`

  The syntax includes an `exclude_column_list` that attempts to exclude too many columns.

- Error number: `ML003048`; SQLSTATE: `HY000`

  Message: exclude_column_list JSON attribute must be of JSON_ARRAY type.

  Example: `ERROR HY000: ML003048: exclude_column_list JSON attribute must be of JSON_ARRAY type.`

  The syntax includes a malformed `JSON_ARRAY` for the `exclude_column_list`.

- Error number: `ML003048`; SQLSTATE: `HY000`

  Message: include_column_list JSON attribute must be of JSON_ARRAY type.

  Example: `ERROR HY000: ML003048: include_column_list JSON attribute must be of JSON_ARRAY type.`

  The syntax includes a malformed `JSON_ARRAY` for the `include_column_list`.

- Error number: `ML003049`; SQLSTATE: `HY000`

  Message: One or more columns in include_column_list ([%]) does not exist. Existing columns are (['%', '%']).

  Example: `ERROR HY000: ML003049: One or more columns in include_column_list ([C15]) does not exist. Existing columns are (['C0', 'C1', 'C2', 'C3']).`

  The syntax includes an `include_column_list` that expects a column that does not exist.

- Error number: `ML003050`; SQLSTATE: `HY000`

  Message: include_column_list must be a subset of exogenous_variables for forecasting task.

  Example: `ERROR HY000: ML003050: include_column_list must be a subset of exogenous_variables for forecasting task.`

  The syntax for a forecasting task includes an `include_column_list` that expects one or more columns that are not defined by `exogenous_variables`.

- Error number: `ML003052`; SQLSTATE: `HY000`

  Message: Target column provided % is one of the independent variables used to train the model [%, %, %].

  Example: `ERROR HY000: ML003052: Target column provided LSTAT is one of the independent variables used to train the model [RM, RAD, LSTAT].`

  The syntax defines a `target_column_name` that is one of the independent variables used to train the model.

- Error number: `ML003053`; SQLSTATE: `HY000`

  Message: datetime_index must be specified by the user for forecasting task and must be a column in the training table.

  Example: `ERROR HY000: ML003053: datetime_index must be specified by the user for forecasting task and must be a column in the training table.`

  The syntax for a forecasting task must include `datetime_index`, and this must be a column in the training table.

- Error number: `ML003054`; SQLSTATE: `HY000`

  Message: endogenous_variables must be specified by the user for forecasting task and must be column(s) in the training table.

  Example: `ERROR HY000: ML003054: endogenous_variables must be specified by the user for forecasting task and must be column(s) in the training table.`

  The syntax for a forecasting task must include the `endogenous_variables` option, and these must be a column or columns in the training table.

- Error number: `ML003055`; SQLSTATE: `HY000`

  Message: endogenous_variables / exogenous_variables option must be of JSON_ARRAY type.

  Example: `ERROR HY000: ML003055: endogenous_variables / exogenous_variables option must be of JSON_ARRAY type.`

  The syntax for a forecasting task includes `endogenous_variables` or `exogenous_variables` that do not have valid `JSON` format.

- Error number: `ML003056`; SQLSTATE: `HY000`

  Message: exclude_column_list cannot contain any of endogenous or exogenous variables for forecasting task.

  Example: `ERROR HY000: ML003056: exclude_column_list cannot contain any of endogenous or exogenous variables for forecasting task.`

  The syntax for a forecasting task includes `exclude_column_list` that contains columns that are also in `endogenous_variables` or `exogenous_variables`.

- Error number: `ML003057`; SQLSTATE: `HY000`

  Message: endogenous and exogenous variables may not have any common columns for forecasting task.

  Example: `ERROR HY000: ML003057: endogenous and exogenous variables may not have any common columns for forecasting task.`

  The syntax for a forecasting task includes `endogenous_variables` and `exogenous_variables`, and they have one or more columns in common.

- Error number: `ML003058`; SQLSTATE: `HY000`

  Message: Can not train a forecasting task with non-numeric endogenous_variables column(s).

  Example: `ERROR HY000: ML003058: Can not train a forecasting task with non-numeric endogenous_variables column(s).`

  The syntax for a forecasting task includes `endogenous_variables` and some of the columns are not defined as numeric.

- Error number: `ML003059`; SQLSTATE: `HY000`

  Message: User provided list of models ['ThetaForecaster', 'ETSForecaster', 'SARIMAXForecaster', 'ExpSmoothForecaster'] does not include any supported models for the task. Supported models for the given task and table are ['DynFactorForecaster', 'VARMAXForecaster'].

  Example: `ERROR HY000: ML003059: User provided list of models ['ThetaForecaster', 'ETSForecaster', 'SARIMAXForecaster', 'ExpSmoothForecaster'] does not include any supported models for the task. Supported models for the given task and table are ['DynFactorForecaster', 'VARMAXForecaster'].`

  The syntax for a forecasting task includes multivariate `endogenous_variables`, but the provided models only support univariate `endogenous_variables`.

- Error number: `ML003060`; SQLSTATE: `HY000`

Message:: endogenous_variables may not contain repeated column names ['%1', '%2', '%1'].

Example: `ERROR HY000: ML003060: endogenous_variables may not contain repeated column names ['wind', 'solar', 'wind'].`

The syntax for a forecasting task includes `endogenous_variables` with a repeated column.

- Error number: `ML003061`; SQLSTATE: `HY000`

  Message: exogenous_variables may not contain repeated column names ['consumption', 'wind_solar', 'consumption'].

  Example: `ERROR HY000: ML003061: exogenous_variables may not contain repeated column names ['consumption', 'wind_solar', 'consumption'].`

  The syntax for a forecasting task includes `exogenous_variables` with a repeated column.

- Error number: `ML003062`; SQLSTATE: `HY000`

  Message: endogenous_variables argument must not be NULL.

  Example: `ERROR HY000: ML003062: endogenous_variables argument must not be NULL.`

  The syntax for a forecasting task includes `endogenous_variables` with a NULL argument.

- Error number: `ML003063`; SQLSTATE: `HY000`

  Message: exogenous_variables argument must not be NULL when provided by user.

  Example: `ERROR HY000: ML003063: exogenous_variables argument must not be NULL when provided by user.`

  The syntax for a forecasting task includes user provided `exogenous_variables` with a NULL argument.

- Error number: `ML003064`; SQLSTATE: `HY000`

  Message: Cannot exclude all models.

  Example: `ERROR HY000: ML003064: Cannot exclude all models.`

  The syntax for a forecasting task must include at least one model.

- Error number: `ML003065`; SQLSTATE: `HY000`

  Message: Prediction table cannot have overlapping datetime_index with train table when exogenous_variables are used. It can only forecast into future.

  Example: `ERROR HY000: ML003065: Prediction table cannot have overlapping datetime_index with train table when exogenous_variables are used. It can only forecast into future.`

  The syntax for a forecasting task includes `exogenous_variables` and the prediction table contains values in the `datetime_index` column that overlap with values in the `datetime_index` column in the training table.

- Error number: `ML003066`; SQLSTATE: `HY000`

Message: datetime_index for test table must not have missing dates after the last date in training table. Please ensure test table starts on or before 2034-01-01 00:00:00. Currently, start date in the test table is 2036-01-01 00:00:00.

Example: `ERROR HY000: ML003066: datetime_index for test table must not have missing dates after the last date in training table. Please ensure test table starts on or before 2034-01-01 00:00:00. Currently, start date in the test table is 2036-01-01 00:00:00.`

The syntax for a forecasting task includes a prediction table that contains values in the `datetime_index` column that leave a gap to the values in the `datetime_index` column in the training table.

- Error number: `ML003067`; SQLSTATE: `HY000`

  Message: datetime_index for forecasting task must be between year 1678 and 2261.

  Example: `ERROR HY000: ML003067: datetime_index for forecasting task must be between year 1678 and 2261.`

  The syntax for a forecasting task includes values in a `datetime_index` column that are outside the date range from 1678 to 2261.

- Error number: `ML003068`; SQLSTATE: `HY000`

  Message: Last date of datetime_index in the training table 2151-01-01 00:00:00 plus the length of the table 135 must be between year 1678 and 2261.

  Example: `ERROR HY000: ML003068: Last date of datetime_index in the training table 2151-01-01 00:00:00 plus the length of the table 135 must be between year 1678 and 2261.`

  The syntax for a forecasting task includes a prediction table that has too many rows, and the values in the `datetime_index` column would be outside the date range from 1678 to 2261.

- Error number: `ML003070`; SQLSTATE: `3877 (HY000)`

  Message: For recommendation tasks both user and item column names should be provided.

  Example: `ERROR 3877 (HY000): ML003070: For recommendation tasks both user and item column names should be provided.`

- Error number: `ML003071`; SQLSTATE: `HY000`

  Message: contamination must be numeric value greater than 0 and less than 0.5.

  Example: `ERROR HY000: ML003071: contamination must be numeric value greater than 0 and less than 0.5.`

- Error number: `ML003071`; SQLSTATE: `3877 (HY000)`

  Message: item_columns can not contain repeated column names ['C4', 'C4'].

  Example: `ERROR 3877 (HY000): ML003071: item_columns can not contain repeated column names ['C4', 'C4'].`

- Error number: `ML003071`; SQLSTATE: `3877 (HY000)`

Message: user_columns can not contain repeated column names ['C4', 'C4'].

Example: `ERROR 3877 (HY000): ML003071: user_columns can not contain repeated column names ['C4', 'C4'].`

- Error number: `ML003072`; SQLSTATE: `HY000`

  Message: Can not use more than one threshold method.

  Example: `ERROR HY000: ML003072: Can not use more than one threshold method.`

- Error number: `ML003072`; SQLSTATE: `3877 (HY000)`

  Message: Target column C3 can not be specified as a user or item column.

  Example: `ERROR 3877 (HY000): ML003072: Target column C3 can not be specified as a user or item column.`

- Error number: `ML003073`; SQLSTATE: `HY000`

  Message: topk must be an integer value between 1 and length of the table, inclusively (1 <= topk <= 20).

  Example: `ERROR HY000: ML003073: topk must be an integer value between 1 and length of the table, inclusively (1 <= topk <= 20).`

- Error number: `ML003073`; SQLSTATE: `3877 (HY000)`

  Message: The users and items columns should be different.

  Example: `ERROR 3877 (HY000): ML003073: The users and items columns should be different.`

- Error number: `ML003074`; SQLSTATE: `HY000`

  Message: threshold must be a numeric value between 0 and 1, inclusively (0 <= threshold <= 1).

  Example: `ERROR HY000: ML003074: threshold must be a numeric value between 0 and 1, inclusively (0 <= threshold <= 1).`

- Error number: `ML003074`; SQLSTATE: `3877 (HY000)`

  Message: Unsupported ML Operation for recommendation task.

  Example: `ERROR 3877 (HY000): ML003074: Unsupported ML Operation for recommendation task.`

- Error number: `ML003075`; SQLSTATE: `HY000`

  Message: Unknown option given. This scoring metric only allows for these options: ['topk'].

  Example: `ERROR HY000: ML003075: Unknown option given. This scoring metric only allows for these options: ['topk'].`

- Error number: `ML003075`; SQLSTATE: `3877 (HY000)`

  Message: Unknown option given. Allowed options for recommendations are ['recommend', 'top'].

  Example: `ERROR 3877 (HY000): ML003075: Unknown option given. Allowed options for recommendations are ['recommend', 'top'].`

- Error number: `ML003076`; SQLSTATE: `HY000`

  Message: ML_EXPLAIN, ML_EXPLAIN_ROW and ML_EXPLAIN_TABLE are not supported for anomaly_detection task.

  Example: `ERROR HY000: ML003076: ML_EXPLAIN, ML_EXPLAIN_ROW and ML_EXPLAIN_TABLE are not supported for anomaly_detection task.`

- Error number: `ML003076`; SQLSTATE: `3877 (HY000)`

  Message: The recommend option should be provided when a value for topk is assigned.

  Example: `ERROR 3877 (HY000): ML003076: The recommend option should be provided when a value for topk is assigned.`

- Error number: `ML003077`; SQLSTATE: `HY000`

  Message: topk must be provided as an option when metric is set as precision_at_k.

  Example: `ERROR HY000: ML003077: topk must be provided as an option when metric is set as precision_at_k.`

- Error number: `ML003077`; SQLSTATE: `3877 (HY000)`

  Message: Unknown recommend value given. Allowed values for recommend are ['ratings', 'items', 'users'].

  Example: `ERROR 3877 (HY000): ML003077: Unknown recommend value given. Allowed values for recommend are ['ratings', 'items', 'users'].`

- Error number: `ML003078`; SQLSTATE: `HY000`

  Message: anomaly_detection only allows 0 (normal) and 1 (anomaly) for labels in target column with any metric used, and they have to be integer values.

  Example: `ERROR HY000: ML003078: anomaly_detection only allows 0 (normal) and 1 (anomaly) for labels in target column with any metric used, and they have to be integer values.`

- Error number: `ML003078`; SQLSTATE: `3877 (HY000)`

  Message: Should not provide a value for topk when the recommend option is set to ratings.

  Example: `ERROR 3877 (HY000): ML003078: Should not provide a value for topk when the recommend option is set to ratings.`

- Error number: `ML003079`; SQLSTATE: `3877 (HY000)`

  Message: Provided value for option topk is not a strictly positive integer.

  Example: `ERROR 3877 (HY000): ML003079: Provided value for option topk is not a strictly positive integer.`

- Error number: `ML003080`; SQLSTATE: `3877 (HY000)`

  Message: One or more rows contains NULL or empty values. Please provide inputs without NULL or empty values for recommendation.

  Example: `ERROR 3877 (HY000): ML003080: One or more rows contains NULL or empty values. Please provide inputs without NULL or empty values for recommendation.`

- Error number: `ML003081`; SQLSTATE: `3877 (HY000)`

  Message: Options should be NULL. Options are currently not supported for this task classification.

  Example: `ERROR 3877 (HY000): ML003081: options should be NULL. Options are currently not supported for this task classification.`

- Error number: `ML003082`; SQLSTATE: `3877 (HY000)`

  Message: All supported models are excluded, but at least one model should be included.

  Example: `ERROR 3877 (HY000): ML003082: All supported models are excluded, but at least one model should be included.`

- Error number: `ML003083`; SQLSTATE: `HY000`

  Message: Both user column name ['C3'] and item column name C0 must be provided as string.

  Example: `ERROR HY000: ML003083: Both user column name ['C3'] and item column name C0 must be provided as string.`

- Error number: `ML003105`; SQLSTATE: `3877 (HY000)`

  Message: Cannot recommend users to a user not present in the training table.

  Example: `ERROR: 3877 (HY000): ML003105: Cannot recommend users to a user not present in the training table.`

- Error number: `ML003106`; SQLSTATE: `3877 (HY000)`

  Message: Cannot recommend items to an item not present in the training table.

  Example: `ERROR 3877 (HY000): ML003106: Cannot recommend items to an item not present in the training table.`

- Error number: `ML003107`; SQLSTATE: `3877 (HY000)`

  Message: Users to users recommendation is not supported, please retrain your model.

  Example: `ERROR 3877 (HY000): ML003107: Users to users recommendation is not supported, please retrain your model.`

- Error number: `ML003108`; SQLSTATE: `3877 (HY000)`

  Message: Items to items recommendation is not supported, please retrain your model.

  Example: `ERROR 3877 (HY000): ML003108: Items to items recommendation is not supported, please retrain your model.`

- Error number: `ML003109`; SQLSTATE: `HY000`

Message: Invalid Model format.

Example: `HY000: ML003109: Invalid Model format.`

- Error number: `ML003111`; SQLSTATE: `HY000`

  Message: Unknown option given. Allowed options are ['batch_size'].

  Example: `ERROR HY000: ML003111: Unknown option given. Allowed options are ['batch_size'].`

- Error number: `ML003112`; SQLSTATE: `HY000`

  Message: Unknown option given. Allowed options for anomaly detection are [X, Y, ...].

  Example: `ERROR HY000: ML003112: Unknown option given. Allowed options for anomaly detection are [X, Y, ...].`

- Error number: `ML003114`; SQLSTATE: `HY000`

  Message: Threshold must be a numeric value.

  Example: `ERROR HY000: ML003114: Threshold must be a numeric value.`

- Error number: `ML003115`; SQLSTATE: `HY000`

  Message: Empty input table after applying threshold.

  Example: `ERROR HY000: ML003115: Empty input table after applying threshold.`

- Error number: `ML003116`; SQLSTATE: `HY000`

  Message: The feedback_threshold option can only be set for implicit feedback.

  Example: `ERROR HY000: ML003116: The feedback_threshold option can only be set for implicit feedback.`

- Error number: `ML003117`; SQLSTATE: `HY000`

  Message: The remove_seen option can only be used with the following recommendation ['items', 'users', 'users_to_items', 'items_to_users'].

  Example: `ERROR HY000: ML003117: The remove_seen option can only be used with the following recommendation ['items', 'users', 'users_to_items', 'items_to_users'].`

- Error number: `ML003118`; SQLSTATE: `HY000`

  Message: The remove_seen option must be set to either True or False. Provided *`input`*.

  Example: `ERROR HY000: ML003118: The remove_seen option must be set to either True or False. Provided input.`

- Error number: `ML003119`; SQLSTATE: `HY000`

  Message: The feedback option must either be set to explicit or implicit. Provided *`input`*.

  Example: `ERROR HY000: ML003119: The feedback option must either be set to explicit or implicit. Provided input.`

- Error number: `ML003120`; SQLSTATE: `HY000`

  Message: The input table needs to contain strictly more than one unique item.

  Example: `ERROR HY000: ML003120: The input table needs to contain strictly more than one unique item.`

- Error number: `ML003121`; SQLSTATE: `HY000`

  Message: The input table needs to contain at least one unknown or negative rating.

  Example: `ERROR HY000: ML003121: The input table needs to contain at least one unknown or negative rating.`

- Error number: `ML003122`; SQLSTATE: `HY000`

  Message: The feedback_threshold option must be numeric.

  Example: `ERROR HY000: ML003122: The feedback_threshold option must be numeric.`

- Error number: `ML003123`; SQLSTATE: `HY000`

  Message: User and item columns should contain strings.

  Example: `ERROR HY000: ML003123: User and item columns should contain strings.`

- Error number: `ML003124`; SQLSTATE: `HY000`

  Message: Calculation for precision_at_k metric could not complete because there are no recommended items.

  Example: `ERROR HY000: ML003124: Calculation for precision_at_k metric could not complete because there are no recommended items.`

- Error number: `ML004002`; SQLSTATE: `HY000`

  Message: Output format of onnx model is not supported (output_name={%},output_shape={%},output_type={%}).

  Example: `HY000: ML004002: Output format of onnx model is not supported (output_name={%},output_shape={%},output_type={%}).`

- Error number: `ML004003`; SQLSTATE: `HY000`

  Message: This ONNX model only supports fixed batch size=%.

  Example: `HY000: ML004003: This ONNX model only supports fixed batch size=%.`

- Error number: `ML004005`; SQLSTATE: `HY000`

  Message: The type % in data_types_map is not supported.

  Example: `HY000: ML004005: The type % in data_types_map is not supported.`

- Error number: `ML004006`; SQLSTATE: `HY000`

  Message: ML_SCORE is not supported for an onnx model that does not support batch inference.

  Example: `HY000: ML004006: ML_SCORE is not supported for an onnx model that does not support batch inference.`

- Error number: `ML004007`; SQLSTATE: `HY000`

  Message: ML_EXPLAIN is not supported for an onnx model that does not support batch inference.

  Example: `HY000: ML004007: ML_EXPLAIN is not supported for an onnx model that does not support batch inference.`

- Error number: `ML004008`; SQLSTATE: `HY000`

  Message: onnx model input type=% is not supported! Providing the appropriate types map using 'data_types_map' in model_metadata may resolve the issue.

  Example: `HY000: ML004008: onnx model input type=% is not supported! Providing the appropriate types map using 'data_types_map' in model_metadata may resolve the issue.`

- Error number: `ML004009`; SQLSTATE: `HY000`

  Message: Input format of onnx model is not supported (onnx_input_name={%}, expected_input_shape={%}, expected_input_type={%}, data_shape={%}).

  Example: `HY000: ML004009: Input format of onnx model is not supported (onnx_input_name={%}, expected_input_shape={%}, expected_input_type={%}, data_shape={%}).`

- Error number: `ML004010`; SQLSTATE: `HY000`

  Message: Output being sparse tensor with batch size > 1 is not supported.

  Example: `HY000: ML004010: Output being sparse tensor with batch size > 1 is not supported.`

- Error number: `ML004010`; SQLSTATE: `3877 (HY000)`

  Message: Received data exceeds maximum allowed length 943718400.

  Example: `ERROR 3877 (HY000): ML004010: Received data exceeds maximum allowed length 943718400.`

- Error number: `ML004011`; SQLSTATE: `HY000`

  Message: predictions_name=% is not valid.

  Example: `HY000: ML004011: predictions_name=% is not valid.`

- Error number: `ML004012`; SQLSTATE: `HY000`

Message: prediction_probabilities_name=% is not valid.

Example: `HY000: ML004012: prediction_probabilities_name=% is not valid.`

- Error number: `ML004013`; SQLSTATE: `HY000`

  Message: predictions_name should be provided when task=regression and onnx model generates more than one output.

  Example: `HY000: ML004013: predictions_name should be provided when task=regression and onnx model generates more than one output.`

- Error number: `ML004014`; SQLSTATE: `HY000`

  Message: Missing expected JSON key (%)

  Example: `ERROR HY000: ML004014: Missing expected JSON key (schema_name).`

- Error number: `ML004014`; SQLSTATE: `HY000`

  Message: Incorrect labels_map. labels_map should include the key %

  Example: `HY000: ML004014: Incorrect labels_map. labels_map should include the key %`

- Error number: `ML004015`; SQLSTATE: `HY000`

  Message: Expected JSON string type value for key (%)

  Example: `ERROR HY000: ML004015: Expected JSON string type value for key (schema_name).`

- Error number: `ML004015`; SQLSTATE: `HY000`

  Message: When task=classification, if the user does not provide prediction_probabilities_name for the onnx model, ML_EXPLAIN method=% will not be supported.

  Example: `HY000: ML004015: When task=classification, if the user does not provide prediction_probabilities_name for the onnx model, ML_EXPLAIN method=% will not be supported.` % can be "shap", "fast_shap" or "partial_dependence"

- Error number: `ML004016`; SQLSTATE: `HY000`

  Message: Given JSON (%, %, %, %, %) is larger than maximum permitted size.

  Example: `ERROR HY000: ML004016: Given JSON (prediction_row prediction_row prediction_row prediction_row prediction_row prediction_row prediction_row prediction_row) is larger than maximum permitted size.`

- Error number: `ML004016`; SQLSTATE: `HY000`

  Message: Invalid base64-encoded ONNX string.

  Example: `HY000: ML004016: Invalid base64-encoded ONNX string.`

- Error number: `ML004017`; SQLSTATE: `HY000`

  Message: Invalid ONNX model.

Example: `HY000: ML004017: Invalid ONNX model.`

- Error number: `ML004017`; SQLSTATE: `3877 (HY000)`

  Message: Input value to plugin variable is too long.

  Example: `ERROR 3877 (HY000): ML004017: Input value to plugin variable is too long.`

- Error number: `ML004018`; SQLSTATE: `HY000`

  Message: Parsing JSON arg: Invalid value. failed!

  Example: `ERROR HY000: ML004018: Parsing JSON arg: Invalid value. failed!`

- Error number: `ML004018`; SQLSTATE: `HY000`

  Message: There are issues in running inference session for the onnx model. This might have happened due to inference on inputs with incorrect names, shapes or types.

  Example: `HY000: ML004018: There are issues in running inference session for the onnx model. This might have happened due to inference on inputs with incorrect names, shapes or types.`

- Error number: `ML004019`; SQLSTATE: `HY000`

  Message: Expected JSON object type value for key (%)

  Example: `ERROR HY000: ML004019: Expected JSON object type value for key (JSON root).`

- Error number: `ML004019`; SQLSTATE: `HY000`

  Message: The computed predictions do not have the right format. This might have happened because the provided predictions_name is not correct.

  Example: `HY000: ML004019: The computed predictions do not have the right format. This might have happened because the provided predictions_name is not correct.`

- Error number: `ML004020`; SQLSTATE: `HY000`

  Message: Operation was interrupted by the user.

  Example: `ERROR HY000: ML004020: Operation was interrupted by the user.`

  If a user-initiated interruption, `Ctrl-C`, is detected during the first phase of HeatWave AutoML model and table load where a MySQL parallel scan is used in the HeatWave plugin to read data as of MySQL database and send it to the HeatWave Cluster, error messaging is handled by the MySQL parallel scan function and directed to `ERROR 1317 (70100): Query execution was interrupted.`. The `ERROR 1317 (70100)` message is reported to the client instead of the `ML004020` error message.

- Error number: `ML004020`; SQLSTATE: `HY000`

  Message: The computed prediction probabilities do not have the right format. This might have happened because the provided prediction_probabilities_name is not correct.

  Example: `HY000: ML004020: The computed prediction probabilities do not have the right format. This might have happened because the provided prediction_probabilities_name is not correct.`

- Error number: `ML004021`; SQLSTATE: `HY000`

  Message: The onnx model and dataset do not match. The onnx model's input=% is not a column in the dataset.

  Example: `HY000: ML004021: The onnx model and dataset do not match. The onnx model's input=% is not a column in the dataset.`

- Error number: `ML004022`; SQLSTATE: `HY000`

  Message: The user does not have access privileges to %.

  Example: `ERROR HY000: ML004022: The user does not have access privileges to ml.foo.`

- Error number: `ML004022`; SQLSTATE: `HY000`

  Message: Labels in y_true and y_pred should be of the same type. Got y_true=% and y_pred=YYY. Make sure that the predictions provided by the classifier coincides with the true labels.

  Example: `HY000: ML004022: Labels in y_true and y_pred should be of the same type. Got y_true=% and y_pred=YYY. Make sure that the predictions provided by the classifier coincides with the true labels.`

- Error number: `ML004026`; SQLSTATE: `HY000`

  Message: A column (%) with an unsupported column type (%) detected!

  Example: `ERROR HY000: ML004026: A column (D1) with an unsupported column type (DATETIME) detected!`

- Error number: `ML004051`; SQLSTATE: `HY000`

  Message: Invalid operation.

  Example: `ERROR HY000: ML004051: Invalid operation.`

- Error number: `ML004999`; SQLSTATE: `HY000`

  Message: Error during Machine Learning.

  Example: `ERROR HY000: ML004999: Error during Machine Learning.`

- Error number: `ML006006`; SQLSTATE: `45000`

  Message: target_column_name should be NULL or empty.

  Example: `ERROR 45000: ML006006: target_column_name should be NULL or empty.`

- Error number: `ML006017`; SQLSTATE: `45000`

Message: model_handle already exists in the Model Catalog.

Example: `45000: ML006017: model_handle already exists in the Model Catalog.`

- Error number: `ML006020`; SQLSTATE: `45000`

  Message: model_metadata should be a JSON object.

  Example: `45000: ML006020: model_metadata should be a JSON object.`

- Error number: `ML006021`; SQLSTATE: `45000`

  Message: contamination has to be passed with anomaly_detection task.

  Example: `ERROR 45000: ML006021: contamination has to be passed with anomaly_detection task.`

- Error number: `ML006022`; SQLSTATE: `45000`

  Message: Unsupported task.

  Example: `ERROR 45000: ML006022: Unsupported task.`

- Error number: `ML006023`; SQLSTATE: `45000`

  Message: "No model object found" will be raised.

  Example: `45000: ML006023: "No model object found" will be raised.`

- Error number: `ML006027`; SQLSTATE: `1644 (45000)`

  Message: Received results exceed `max_allowed_packet`. Please increase it or lower input options value to reduce result size.

  Example: `ERROR 1644 (45000): ML006027: Received results exceed `max_allowed_packet`. Please increase it or lower input options value to reduce result size.`

- Error number: `ML006029`; SQLSTATE: `45000`

  Message: model_handle is not Ready.

  Example: `45000: ML006029: model_handle is not Ready.`

- Error number: `ML006030`; SQLSTATE: `45000`

  Message: onnx_inputs_info must be a json object.

  Example: `ERROR 45000: ML006030: onnx_inputs_info must be a json object.`

- Error number: `ML006031`; SQLSTATE: `45000`

  Message: Unsupported format.

  Example: `45000: ML006031: Unsupported format.`

- Error number: `ML006031`; SQLSTATE: `45000`

  Message: onnx_outputs_info must be a json object.

Example: `ERROR 45000: ML006031: onnx_outputs_info must be a json object.`

- Error number: `ML006032`; SQLSTATE: `45000`

  Message: data_types_map must be a json object.

  Example: `ERROR 45000: ML006032: data_types_map must be a json object.`

- Error number: `ML006033`; SQLSTATE: `45000`

  Message: labels_map must be a json object.

  Example: `ERROR 45000: ML006033: labels_map must be a json object.`

- Error number: `ML006034`; SQLSTATE: `45000`

  Message: onnx_outputs_info must be provided for task=classification.

  Example: `ERROR 45000: ML006034: onnx_outputs_info must be provided for task=classification.`

- Error number: `ML006035`; SQLSTATE: `45000`

  Message: onnx_outputs_info must only be provided for classification and regression tasks.

  Example: `ERROR 45000: ML006035: onnx_outputs_info must only be provided for classification and regression tasks.`

- Error number: `ML006036`; SQLSTATE: `45000`

  Message: % is not a valid key in onnx_inputs_info.

  Example: `ERROR 45000: ML006036: % is not a valid key in onnx_inputs_info.`

- Error number: `ML006037`; SQLSTATE: `45000`

  Message: % is not a valid key in onnx_outputs_info.

  Example: `ERROR 45000: ML006037: % is not a valid key in onnx_outputs_info.`

- Error number: `ML006038`; SQLSTATE: `45000`

  Message: For task=classification, at least one of predictions_name and prediction_probabilities_name must be provided.

  Example: `ERROR 45000: ML006038: For task=classification, at least one of predictions_name and prediction_probabilities_name must be provided.`

- Error number: `ML006039`; SQLSTATE: `45000`

  Message: prediction_probabilities_name must only be provided for task=classification.

  Example: `ERROR 45000: ML006039: prediction_probabilities_name must only be provided for task=classification.`

- Error number: `ML006040`; SQLSTATE: `45000`

  Message: predictions_name must not be an empty string.

Example: `ERROR 45000: ML006040: predictions_name must not be an empty string.`

- Error number: `ML006041`; SQLSTATE: `45000`

  Message: prediction_probabilities_name must not be an empty string.

  Example: `ERROR 45000: ML006041: prediction_probabilities_name must not be an empty string.`

- Error number: `ML006042`; SQLSTATE: `45000`

  Message: labels_map must only be provided for task=classification.

  Example: `ERROR 45000: ML006042: labels_map must only be provided for task=classification.`

- Error number: `ML006043`; SQLSTATE: `45000`

  Message: When labels_map is provided, prediction_probabilities_name must also be provided.

  Example: `ERROR 45000: ML006043: When labels_map is provided, prediction_probabilities_name must also be provided.`

- Error number: `ML006044`; SQLSTATE: `45000`

  Message: When labels_map is provided, predictions_name must not be provided.

  Example: `ERROR 45000: ML006044: When labels_map is provided, predictions_name must not be provided.`

- Error number: `ML006045`; SQLSTATE: `45000`

  Message: ML_SCORE is not supported for a % task.

  Example: `ERROR 45000: ML006045: ML_SCORE is not supported for a % task.`

- Error number: `ML006046`; SQLSTATE: `45000`

  Message: ML_EXPLAIN is not supported for a % task.

  Example: `ERROR 45000: ML006046: ML_EXPLAIN is not supported for a % task.`

- Error number: `ML006047`; SQLSTATE: `45000`

  Message: onnx_inputs_info must only be provided when format='ONNX'.

  Example: `ERROR 45000: ML006047: onnx_inputs_info must only be provided when format='ONNX'.`

- Error number: `ML006048`; SQLSTATE: `45000`

  Message: onnx_outputs_info must only be provided when format='ONNX'.

  Example: `ERROR 45000: ML006048: onnx_outputs_info must only be provided when format='ONNX'.`

- Error number: `ML006049`; SQLSTATE: `45000`

  Message: The length of a key provided in onnx_inputs_info should not be greater than 32 characters.

Example: `ERROR 45000: ML006049: The length of a key provided in onnx_inputs_info should not be greater than 32 characters.`

- Error number: `ML006050`; SQLSTATE: `45000`

  Message: The length of a key provided in onnx_outputs_info should not be greater than 32 characters.

  Example: `ERROR 45000: ML006050: The length of a key provided in onnx_outputs_info should not be greater than 32 characters.`

- Error number: `ML006051`; SQLSTATE: `45000`

  Message: Invalid ONNX model.

  Example: `ERROR 45000: ML006051: Invalid ONNX model.`

- Error number: `ML006052`; SQLSTATE: `45000`

  Message: Input table is empty. Please provide a table with at least one row.

  Example: `ERROR 45000: ML006052: Input table is empty. Please provide a table with at least one row.`

- Error number: `ML006053`; SQLSTATE: `45000`

  Message: Insufficient access rights. Grant user with correct privileges (SELECT, DROP, CREATE, INSERT, ALTER) on input schema.

  Example: `ERROR 45000: ML006053: Insufficient access rights. Grant user with correct privileges (SELECT, DROP, CREATE, INSERT, ALTER) on input schema.`

- Error number: `ML006054`; SQLSTATE: `45000`

  Message: Input table already contains a column named '_id'. Please provide an input table without such column.

  Example: `ERROR 45000: ML006054: Input table already contains a column named '_id'. Please provide an input table without such column.`

- Error number: `ML006055`; SQLSTATE: `45000`

  Message: Options must be a JSON_OBJECT.

  Example: `ERROR 45000: ML006055: Options must be a JSON_OBJECT.`

- Error number: `ML006056`; SQLSTATE: `45000`

  Message: batch_size must be an integer between 1 and %.

  Example: `ERROR 45000: ML006056: batch_size must be an integer between 1 and %.`

- Error number: `ML006070`; SQLSTATE: `45000`

  Message: model_list is currently not supported for anomaly_detection.

  Example: `ERROR 45000: ML006070: model_list is currently not supported for anomaly_detection.`

- Error number: `ML006071`; SQLSTATE: `45000`

  Message: exclude_model_list is currently not supported for anomaly_detection.

  Example: `ERROR 45000: ML006071: exclude_model_list is currently not supported for anomaly_detection.`

- Error number: `ML006072`; SQLSTATE: `45000`

  Message: optimization_metric is currently not supported for anomaly_detection.

  Example: `ERROR 45000: ML006072: optimization_metric is currently not supported for anomaly_detection.`

3.19 HeatWave AutoML Limitations

The following limitations apply to HeatWave AutoML. For HeatWave limitations, see: Section 2.18, “HeatWave MySQL Limitations”.

- The `ML_TRAIN` routine does not support MySQL user names that contain a period; for example, a user named `'joe.smith'@'%'` cannot run the `ML_TRAIN` routine. The model catalog schema created by the `ML_TRAIN` procedure incorporates the user name in the schema name (e.g., `ML_SCHEMA_joesmith`), and a period is not a permitted schema name character.
- The table used to train a model cannot exceed 10 GB, 100 million rows, or 1017 columns.
- As of MySQL 9.0.0, the maximum model size is only limited by the amount of memory defined by the shape. `ML_TRAIN` will fail to execute and will raise an error if it attempts to train a model larger than this. Before MySQL 9.0.0, the limit was 900 MB.

  All active users and models share this memory, and it might be necessary to schedule users. See: `ML_MODEL_ACTIVE`.
- To avoid taking up too much space in memory, the number of loaded models should be limited to three.
- “Bring your own model” is not supported. Use of non-HeatWave AutoML models or manually modified HeatWave AutoML models can cause undefined behavior.
- There is currently no way to monitor HeatWave AutoML query progress. `ML_TRAIN` is typically the most time consuming routine. The time required to train a model depends on the number of rows and columns in the dataset and the specified `ML_TRAIN` parameters and options.

  `ML_EXPLAIN_TABLE` and `ML_PREDICT_TABLE` are compute intensive processes, with `ML_EXPLAIN_TABLE` being the most compute intensive. Limiting operations to batches of 10 to 100 rows by splitting large tables into smaller tables is recommended. Use batch processing with the `batch_size` option. See: Section 3.15, “Progress tracking”.
- Only `ML_TRAIN`, `ML_EXPLAIN_ROW`, and `ML_EXPLAIN_TABLE` support `Ctrl+C` interruption.
- `ML_EXPLAIN`, `ML_EXPLAIN_ROW`, and `ML_EXPLAIN_TABLE` routines limit explanations to the 100 most relevant features.
- The `ML_PREDICT_TABLE` `ml_results` column contains the prediction results and the data. This combination must be less than 65,532 characters.
- HeatWave AutoML only supports datasets in the English language.
- HeatWave AutoML does not support text columns with `NULL` values.

- HeatWave AutoML does not support a text target column.
- HeatWave AutoML does not support recommendation tasks with a text column.
- For the forecasting task, `endogenous_variables` cannot be text.
- Concurrent HeatWave analytics and HeatWave AutoML queries are not supported. A HeatWave AutoML query must wait for HeatWave analytics queries to finish, and vice versa. HeatWave analytics queries are given priority over HeatWave AutoML queries.
- HeatWave on AWS only supports HeatWave AutoML with the HeatWave.256GB node shape. To use HeatWave machine learning functionality, select that shape when creating a HeatWave Cluster.

Chapter 4 HeatWave GenAI

Table of Contents

This chapter describes HeatWave GenAI.

4.1 HeatWave GenAI Overview

HeatWave GenAI is a feature of HeatWave that lets you communicate with unstructured data in HeatWave using natural-language queries. It uses a familiar SQL interface which makes it is easy to use for content generation, summarization, and retrieval-augmented generation (RAG).

Using HeatWave GenAI, you can perform natural-language searches in a single step using either in-database or external large language models (LLMs). All the elements that are necessary to use HeatWave GenAI with proprietary data are integrated and optimized to work with each other.

> **Note**
>
> This chapter assumes that you are familiar with HeatWave Database Systems.

HeatWave GenAI includes the following:

- **In-Database and OCI Generative AI Service LLMs**

HeatWave GenAI uses large language models (LLMs) to enable natural language communication in multiple languages. You can use the capabilities of the LLMs to search data as well as generate or summarize content. However, as these LLMs are trained on public data, the responses to your queries are generated based on information available in the public data sources. To produce more relevant results, you can use the LLM capabilities of HeatWave GenAI with the vector store functionality to perform a vector search using RAG.

- **In-Database Vector Store**

  HeatWave GenAI provides an inbuilt vector store that you can use to store enterprise-specific proprietary content and perform vector or similarity across documents. Queries that you ask are automatically encoded with the same embedding model as the vector store without requiring any additional inputs or running a separate service. The vector store also provides valuable context for LLMs for RAG use cases.

- **Retrieval-Augmented Generation**

  HeatWave GenAI retrieves content from the vector store and provides it as context to the LLM along with the query. This process of generating an augmented prompt is called RAG, and it helps HeatWave GenAI produce more contextually relevant, personalised, and accurate results.

- **HeatWave Chat**

  This is an inbuilt chatbot that extends the LLMs capabilities as well as vector store and RAG functionalities of HeatWave GenAI to let you ask multiple follow-up questions about a topic in a single session. You can use HeatWave Chat to build customized chat applications by specifying custom settings, prompt, chat history length, and number of citations to be used for generating a response.

  HeatWave Chat also provides a graphical interface integrated with the Visual Studio Code plugin for MySQL Shell.

Benefits

HeatWave GenAI lets you integrate generative AI into the applications, providing an integrated end-to-end pipeline including vector store generation, vector search using RAG, and an inbuilt chatbot.

Some key benefits of using HeatWave GenAI are described below:

- The natural-language processing (NLP) capabilities of the LLMs let non-technical users have human-like conversations with the system in natural language.
- The in-database integration of LLMs and embedding generation eliminates the need for using external solutions, and ensures the security of the proprietary content.
- The in-database integration of LLMs, vector store, and embedding generation simplifies complexity of applications that use these features.
- The cost of running natural-language queries is significantly low as HeatWave GenAI is available at no additional cost for HeatWave users.
- HeatWave GenAI integrates with other in-database capabilities such as machine learning, analytics, and Lakehouse.

See Also

- HeatWave GenAI: Technical Overview

4.2 Getting Started with HeatWave GenAI

The sections in this topic describe how to get started with HeatWave GenAI.

4.2.1 Requirements

- To use HeatWave GenAI, you require a HeatWave Database System.

  The database system must meet the following requirements:

  - The database system must be version `9.0.0 - Innovation` or higher.
  - A HeatWave Cluster must be added to your database system. The shape of the cluster must be `HeatWave.512GB`.
  - HeatWave Lakehouse must be enabled on the database system.

  For more information, see Creating a DB System, Editing a DB System, and Connecting to the DB System.

- To create a vector store, you need an Oracle Cloud Infrastructure (OCI) Object Storage bucket for storing files that you want the vector store to ingest. Vector store can ingest files in the following formats: PPT, TXT, HTML, DOC, and PDF.

4.2.2 Supported Languages, Embedding Models, and LLMs

This section lists the languages, embedding models, and large language models (LLMs) that HeatWave GenAI supports.

Languages

In HeatWave `9.0.0` and later versions, HeatWave GenAI supports natural-language communication, ingesting documents, as well as generating text-based content in English (`en`).

In HeatWave `9.0.1-u1` and later versions, HeatWave GenAI supports natural-language communication, ingesting documents, as well as generating text-based content in multiple languages. The quality of the generated text outputs depends on the training and ability of the LLM to work with the language.

Following is a combined list of languages supported by HeatWave In-Database LLMs and OCI Generative AI Service LLMs:

- Arabic (`ar`)
- Bengali (`bn`)
- Brazilian Portuguese (`pt-br`)
- Burmese (`my`)
- Chinese (`zh`)
- Czech (`cs`)
- Dutch (`nl`)
- English (`en`)

- French (`fr`)
- German (`de`)
- Hebrew (`he`)
- Hindi (`hi`)
- Indonesian (`id`)
- Italian (`it`)
- Japanese (`ja`)
- Khmer (`km`)
- Korean (`ko`)
- Lao (`lo`)
- Malay (`ms`)
- Persian (`fa`)
- Polish (`pl`)
- Portuguese (`pt`)
- Spanish (`es`)
- Tagalog (`tl`)
- Thai (`th`)
- Turkish (`tr`)
- Urdu (`ur`)
- Vietnamese (`vi`)

However, not all LLMs support all the languages. To learn which LLM supports which language, see HeatWave In-Database LLMs and OCI Generative AI Service LLMs.

> **Note**
>
> To set the value of the `language` parameter in HeatWave GenAI routines that support this parameter, do not use the language name to specify the language. Use the two-letter `ISO 639-1` code for the language instead. For example, to use French, use the `ISO 639-1` code for French, which is `fr`.

HeatWave In-Database Embedding Models

HeatWave GenAI supports the following in-database embedding models for generating vector embeddings:

- `all_minilm_l12_v2`: in HeatWave `9.0.0` and later versions, HeatWave GenAI uses this embedding model, by default, for encoding English documents.
- `multilingual-e5-small`: in HeatWave `9.0.1-u1` and later versions, HeatWave GenAI uses this embedding model, by default, for encoding documents in all supported languages other than English.

OCI Generative AI Service Embedding Models

In HeatWave `9.0.1-u1` and later versions, HeatWave GenAI supports Oracle Cloud Infrastructure (OCI) Generative AI service embedding models for generating vector embeddings.

> **Note**
>
> These models are available only in the regions supported by the OCI Generative AI service. For more information, see Pretrained Foundational Models in Generative AI.

However, to use the OCI Generative AI service with HeatWave GenAI, you need to enable the database system to access OCI services. For more information, see Section 4.2.3, “Authenticating OCI Generative AI Service”.

HeatWave GenAI supports the following OCI Generative AI service embedding models:

- `cohere.embed-english-v3.0`: for encoding English text.
- `cohere.embed-multilingual-v3.0`: for encoding text in supported languages other than English.

HeatWave In-Database LLMs

HeatWave GenAI supports the following in-database LLMs:

- `mistral-7b-instruct-v1`
  - In HeatWave `9.0.0` and later versions, supports English.
  - In HeatWave `9.0.1-u1` and later versions, supports the following languages:
    - English (`en`)
    - French (`fr`)
    - German (`de`)
    - Spanish (`es`)
- `llama2-7b-v1`
  - In HeatWave `9.0.0`, supports English (`en`).
  - In HeatWave `9.0.1-u1` and later versions, support for `llama2-7b-v1` has been deprecated.
- `llama3-8b-instruct-v1`
  - This LLM is available in HeatWave `9.0.1-u1` and later versions.
  - It supports the following languages:
    - English (`en`)
    - French (`fr`)
    - German (`de`)
    - Hindi (`hi`)

- Italian (`it`)
- Portuguese (`pt`)
- Spanish (`es`)
- Thai (`th`)

OCI Generative AI Service LLMs

In HeatWave `9.0.1-u1` and later versions, HeatWave GenAI supports OCI Generative AI service LLMs.

> **Note**
>
> These LLMs are available only in the regions supported by the OCI Generative AI service. For more information, see Pretrained Foundational Models in Generative AI.

However, to use the OCI Generative AI service with HeatWave GenAI, you need to enable the database system to access OCI services. For more information, see Section 4.2.3, “Authenticating OCI Generative AI Service”.

HeatWave GenAI supports the following OCI Generative AI service LLMs:

- `cohere.command-r-plus`
  - In HeatWave `9.0.1-u1` and later versions, supports all available Languages.
- `cohere.command-r-16k`
  - In HeatWave `9.0.1-u1` and later versions, supports all available Languages.
- `meta.llama-3-70b-instruct`
  - In HeatWave `9.0.1-u1` and later versions, supports all available Languages.

4.2.3 Authenticating OCI Generative AI Service

To enable the database system to access OCI services, perform the following steps in OCI:

1. Create a new dynamic group or update a dynamic group's matching rules, and the following matching rule to include all MySQL DB Systems in the defined compartment:

   ```
   ALL{resource.type = 'mysqldbsystem', resource.compartment.id = 'ocid1.compartment.oc1..AlphanumericString'}
   ```

2. Add policies for the dynamic group that grant the dynamic group access to the target service's APIs:

   ```
   allow dynamic-group IdentityDomainName/GroupName to use generative-ai-chat in compartment CompartmentName
   allow dynamic-group IdentityDomainName/GroupName to use generative-ai-text-embedding in compartment Compart
   ```

   Replace the following:

   - *`IdentityDomainName`*: the identity domain name.

     > **Note**
     >
     > If the dynamic group belongs to the default identity domain, you can omit specifying the identity domain name.

- *GroupName*: the dynamic group name
- *CompartmentID*: the compartment ID of the database system

For more information, see Resource Principals.

In addition, for resource principal authentication, HeatWave automatically sets the `rapid_ml_genai` system variable. The default value of the variable is the compartment ID of the database system you are using.

To use a different compartment for accessing the OCI Generative AI service, you can set the `rapid_ml_genai_session` system variable to specify the ID of the compartment you want to use, as shown below:

```
SET rapid_ml_genai_session = '{"compartment_id": "ocid1.compartment.oc1..AlphanumericString"}';
```

The compartment ID specified in the `rapid_ml_genai_session` system variable must match the compartment ID you used to create the dynamic group and add the OCI Generative AI service authentication policy.

4.2.4 Quickstart: Setting Up a Help Chat

This quickstart shows how to use the vector store functionality and use HeatWave Chat to create a GenAI-powered Help chat that refers to the HeatWave user guide to respond to HeatWave related queries.

> **Note**
>
> This quickstart assumes that you are familiar with HeatWave Database Systems.

This quickstart contains the following sections:

- Before You Begin
- Setting Up the Object Storage Bucket
- Connecting to the Database System
- Setting Up the Vector Store
- Starting a Chat Session
- Cleaning Up

Before You Begin

Review the Requirements.

- To run this quickstart, you need HeatWave Database System version `9.0.0 - Innovation` or higher. For more information, see Creating a DB System and Editing a DB System.
- If not already done, add a HeatWave Cluster to your database system.
- If not already done, enable HeatWave Lakehouse on the database system.
- Enable the database system to access an OCI Object Storage bucket. For more information, see Resource Principals.

Setting Up the Object Storage Bucket

1. Download the HeatWave user guide PDF (A4) - 1.7Mb.

2. Create an Object Storage Bucket with the name `quickstart_bucket`.
3. Upload the PDF file to the Object Storage Bucket using the prefix `quickstart/` to create a new folder by the name *quickstart*.

Connecting to the Database System

- Connect to your HeatWave Database System.

```
mysqlsh -uAdmin -pPassword -hPrivateIP --sqlc
```

> **Note**
>
> X protocol is not supported for setting up a vector store using asynchronous load. To run this quickstart, ensure that you use the classic MySQL protocol while connecting to the database.

Replace the following:

- *`Admin`*: the database system admin name.
- *`Password`*: the database system password.
- *`PrivateIP`*: the private IP address of the database system.

Setting Up the Vector Store

1. Create and use a new database:

```
create database quickstart_db;
use quickstart_db;
```

2. To create a schema to be used for task management, run the following command:

```
select mysql_task_management_ensure_schema();
```

3. Create the vector table and load the source document:

```
call sys.VECTOR_STORE_LOAD('oci://quickstart_bucket@Namespace/quickstart/heatwave-en.a4.pdf', '{"table_name
```

Replace *`Namespace`* with the name of the Object Storage bucket namespace that you are using.

This creates an asynchronous task in the background which loads the vector embeddings into the specified vector store table `quickstart_embeddings`.

The routine output that is displayed contains a query with the task ID.

4. To track the progress of the task, run the task query displayed on the screen:

```
select id, name, message, progress, status, scheduled_time,estimated_completion_time, estimated_remaining_t
```

Replace *`TaskID`* with the displayed task ID.

The output looks similar to the following:

```
                      id: 1
                    name: Vector Store Loader
                 message: Task starting.
                 progress: 0
                   status: RUNNING
```

```
            scheduled_time: 2024-07-02 14:42:38
estimated_completion_time: 2024-07-22 10:19:53
  estimated_remaining_time: 52.50000
              progress_bar: __________
```

5. After the task status has changed to `Completed`, verify that embeddings are loaded in the vector store table:

```
select count(*) from quickstart_embeddings;
```

   If you see a numerical value in the output, your embeddings are successfully loaded in the vector store table.

Starting a Chat Session

1. Clear the previous chat history and states:

```
set @chat_options=NULL;
```

2. Ask your question using HeatWave Chat:

```
call sys.HEATWAVE_CHAT("What is HeatWave AutoML?");
```

   The `HEATWAVE_CHAT` routine automatically loads the LLM and runs a semantic search on the available vector stores and retrieves context, by default. The output is similar to the following:

```
|  HeatWave AutoML is a feature of MySQL HeatWave that makes it easy to use
machine learning, whether you are a novice user or an experienced ML practitioner.
It analyzes the characteristics of the data and creates an optimized machine
learning model that can be used to generate predictions and explanations. The
data and models never leave MySQL HeatWave, saving time and effort while keeping
the data and models secure. HeatWave AutoML is optimized for HeatWave shapes and
scaling, and all processing is performed on the HeatWave Cluster. |
```

3. Ask a follow-up question:

```
call sys.HEATWAVE_CHAT("How to set it up?");
```

   The output is similar to the following:

```
|  To set up HeatWave AutoML in MySQL HeatWave, you need to follow these steps:

1. Ensure that you have an operational MySQL DB System and are able to
connect to it using a MySQL client. If not, complete the steps described
in Getting Started with MySQL HeatWave.
2. Ensure that your MySQL DB System has an operational HeatWave Cluster.
If not, complete the steps described in Adding a HeatWave Cluster.
3. Obtain the MySQL user privileges described in Section 3.2, Before You Begin.
4. Prepare and load training and test data. See Section 3.4, Preparing Data.
5. Train a machine learning model. See Section 3.5, Training a Model.
6. Make predictions using the trained model. See Section 3.6, Making Predictions.
7. Generate explanations for the predictions made by the model. See Section
3.7, Generating Explanations.
8. Monitor and manage the performance of the model. See Section 3.8,
Monitoring and Managing Performance. |
```

   You can continue asking follow-up questions in the same chat session.

Cleaning Up

To avoid being billed for the resources that you created for this quickstart, perform the following steps:

1. Delete the database that you created:

```
drop database quickstart_db;
```

2. If you created a new database system, then delete the database system. For more information, see Deleting a DB System.
3. Delete `quickstart_bucket`. For more information, see Deleting the Object Storage Bucket.

4.3 Generating Text-Based Content

For generating text-based content and summarizing text, use the following routines:

- The `ML_MODEL_LOAD` routine loads a large language model (LLM) into the HeatWave Cluster.
- The `ML_GENERATE` routine uses the LLM to generate the text output.

The sections in this topic describe how to generate and summarize text-based content using HeatWave GenAI.

4.3.1 Generating New Content

This section describes how to generate new text-based content using HeatWave GenAI.

Before You Begin

- Connect to your HeatWave Database System.
- For Running Batch Queries, add the natural-language queries to a column in a new or existing table.

Generating Content

To generate text-based content using HeatWave GenAI, perform the following steps:

1. To load the LLM in HeatWave memory, use the `ML_MODEL_LOAD` routine:

   ```
   call sys.ML_MODEL_LOAD("LLM", NULL);
   ```

   Replace *`LLM`* with the name of the LLM that you want to use. To view the lists of supported LLMs, see HeatWave In-Database LLMs and OCI Generative AI Service LLMs.

   For example:

   ```
   call sys.ML_MODEL_LOAD("mistral-7b-instruct-v1", NULL);
   ```

   This step is optional. The `ML_GENERATE` routine loads the specified LLM too. But it takes a bit longer to load the LLM and generate the output when you run it for the first time.

2. To define your natural-language query, set the `@query` session variable:

   ```
   set @query="QueryInNaturalLanguage";
   ```

   Replace *`QueryInNaturalLanguage`* with a natural-language query of your choice. For example:

   ```
   set @query="Write an article on Artificial intelligence in 200 words.";
   ```

3. To generate text-based content, pass the query to the LLM using the `ML_GENERATE` routine with the `task` parameter set to `generation`:

   ```
   select sys.ML_GENERATE(@query, JSON_OBJECT("task", "generation", "model_id", "LLM", "language", "Language")
   ```

   Replace the following:

- *LLM*: LLM to use, which must be the same as the one you loaded in the previous step.
- *Language*: the two-letter ISO 639-1 code for the language you want to use. Default language is en, which is English. To view the list of supported languages, see Languages.

  > **Note**
  >
  > The language parameter is supported in HeatWave 9.0.1-u1 and later versions.

For example:

```
select sys.ML_GENERATE(@query, JSON_OBJECT("task", "generation", "model_id", "mistral-7b-instruct-v1",
```

Text-based content that is generated by the LLM in response to your query is printed as output. It looks similar to the text output shown below:

```
| {"text": " Artificial Intelligence, commonly referred to as AI, is a
rapidly growing field that focuses on creating intelligent machines capable
of performing tasks that typically require human intelligence. These tasks
include things like understanding natural language, recognizing images, and
making decisions.\n\nAI technology has come a long way in recent years, thanks
to advances in machine learning and deep learning algorithms. These algorithms
allow machines to learn from data and improve their performance over time.
This has led to the development of more advanced AI systems, such as virtual
assistants like Siri and Alexa, which can help users with tasks like setting
reminders and answering questions.\n\nAI is also being used in a variety of other
industries, including healthcare, finance, and transportation. In healthcare, AI
is being used to help doctors diagnose diseases and develop treatment plans. In
finance, AI is being used to detect fraud and make investment decisions. In
transportation, AI is being used to develop self-driving cars and improve traffic
flow.\n\nDespite the many benefits of AI, there are also concerns about its potential
impact on society. Some worry that AI could lead to job displacement and a loss of
privacy. Others worry that AI could be used for malicious purposes, such as
cyber attacks or surveillance.\n"} |
```

Running Batch Queries

To run multiple generation queries in parallel, use the ML_GENERATE_TABLE routine. This method is faster than running the ML_GENERATE routine multiple times.

> **Note**
>
> The ML_GENERATE_TABLE routine is supported in HeatWave 9.0.1-u1 and later versions.

To run batch queries using ML_GENERATE_TABLE, perform the following steps:

1. To load the LLM in HeatWave memory, use the ML_MODEL_LOAD routine:

   ```
   call sys.ML_MODEL_LOAD("LLM", NULL);
   ```

   Replace *LLM* with the name of the LLM that you want to use. To view the lists of supported LLMs, see HeatWave In-Database LLMs and OCI Generative AI Service LLMs.

   For example:

   ```
   call sys.ML_MODEL_LOAD("mistral-7b-instruct-v1", NULL);
   ```

   This step is optional. The ML_GENERATE_TABLE routine loads the specified LLM too. But it takes a bit longer to load the LLM and generate the output when you run it for the first time.

2. In the ML_GENERATE_TABLE routine, specify the table columns containing the input queries and for storing the generated text-based responses:

```
call sys.ML_GENERATE_TABLE("InputDBName.InputTableName.InputColumn", "OutputDBName.OutputTableName.OutputCo
```

Replace the following:

- *InputDBName*: the name of the database that contains the table column where your input queries are stored.
- *InputTableName*: the name of the table that contains the column where your input queries are stored.
- *InputColumn*: the name of the column that contains input queries.
- *OutputDBName*: the name of the database that contains the table where you want to store the generated outputs. This can be the same as the input database.
- *OutputTableName*: the name of the table where you want to create a new column to store the generated outputs. This can be the same as the input table. If the specified table doesn't exist, a new table is created.
- *OutputColumn*: the name for the new column where you want to store the output generated for the input queries.
- *LLM*: LLM to use, which must be the same as the LLM you loaded in the previous step.
- *Language*: the two-letter ISO 639-1 code for the language you want to use. Default language is en, which is English. To view the list of supported languages, see Languages.

For example:

```
call sys.ML_GENERATE_TABLE("demo_db.input_table.Input", "demo_db.output_table.Output", JSON_OBJECT("task",
```

To learn more about the available routine options, see ML_GENERATE_TABLE Syntax.

4.3.2 Summarizing Content

This section describes how to summarize exiting content using HeatWave GenAI.

Before You Begin

- Connect to your HeatWave Database System.
- For Running Batch Queries, add the natural-language queries to a column in a new or existing table.

Summarizing Content

To summarize text, perform the following steps:

1. To load the LLM in HeatWave memory, use the ML_MODEL_LOAD routine:

```
call sys.ML_MODEL_LOAD("LLM", NULL);
```

Replace *LLM* with the name of the LLM that you want to use. Summarization supports HeatWave In-Database LLMs only.

For example:

```
call sys.ML_MODEL_LOAD("mistral-7b-instruct-v1", NULL);
```

This step is optional. The `ML_GENERATE` routine loads the specified LLM too. But it takes a bit longer to load the LLM and generate the output when you run it for the first time.

2. To define the text that you want to summarize, set the `@text` session variable:

```
set @text="TextToSummarize";
```

Replace *`TextToSummarize`* with the text that you want to summarize.

For example:

```
set @text="Artificial Intelligence (AI) is a rapidly growing field that has the potential to
revolutionize how we live and work. AI refers to the development of computer systems that can
perform tasks that typically require human intelligence, such as visual perception, speech
recognition, decision-making, and language translation.\n\nOne of the most significant developments in
AI in recent years has been the rise of machine learning, a subset of AI that allows computers to learn
from data without being explicitly programmed. Machine learning algorithms can analyze vast amounts
of data and identify patterns, making them increasingly accurate at predicting outcomes and making
decisions.\n\nAI is already being used in a variety of industries, including healthcare, finance, and
transportation. In healthcare, AI is being used to develop personalized treatment plans for patients
based on their medical history and genetic makeup. In finance, AI is being used to detect fraud and mak
investment recommendations. In transportation, AI is being used to develop self-driving cars and improv
traffic flow.\n\nDespite the many benefits of AI, there are also concerns about its potential impact on
society. Some worry that AI could lead to job displacement, as machines become more capable of performi
tasks traditionally done by humans. Others worry that AI could be used for malicious ";
```

3. To generate the text summary, pass the original text to the LLM using the `ML_GENERATE` routine, with the `task` parameter set to `summarization`:

```
select sys.ML_GENERATE(@query, JSON_OBJECT("task", "summarization", "model_id", "LLM", "language", "Lan
```

Replace the following:

- *`LLM`*: LLM to use, which must be the same as the one you loaded in the previous step. To view the lists of supported LLMs, see HeatWave In-Database LLMs and OCI Generative AI Service LLMs.
- *`Language`*: the two-letter `ISO 639-1` code for the language you want to use. Default language is `en`, which is English. To view the list of supported languages, see Languages.

> **Note**
>
> The `language` parameter is supported in HeatWave `9.0.1-u1` and later versions.

For example:

```
select sys.ML_GENERATE(@text, JSON_OBJECT("task", "summarization", "model_id", "mistral-7b-instruct-v1"
```

A text summary generated by the LLM in response to your query is printed as output. It looks similar to the text output shown below:

```
| {"text": " Artificial Intelligence (AI) is a rapidly growing field with the potential to revolutioniz
how we live and work. It refers to computer systems that can perform tasks requiring human intelligence
as visual perception, speech recognition, decision-making, and language translation. Machine learning,
subset of AI, allows computers to learn from data without being explicitly programmed, making them incr
accurate at predicting outcomes and making decisions. AI is already being used in healthcare, finance,
transportation industries for personalized treatment plans, fraud detection, and self-driving cars. How
there are concerns about its potential impact on society, including job displacement and malicious use.
```

Running Batch Queries

To run multiple `summarization` queries in parallel, use the `ML_GENERATE_TABLE` routine. This method is faster than running the `ML_GENERATE` routine multiple times.

> **Note**
>
> The `ML_GENERATE_TABLE` routine is supported in HeatWave `9.0.1-u1` and later versions.

To run batch queries using `ML_GENERATE_TABLE`, perform the following steps:

1. To load the LLM in HeatWave memory, use the `ML_MODEL_LOAD` routine:

   ```
   call sys.ML_MODEL_LOAD("LLM", NULL);
   ```

   Replace *`LLM`* with the name of the LLM that you want to use. To view the lists of supported LLMs, see HeatWave In-Database LLMs and OCI Generative AI Service LLMs.

   For example:

   ```
   call sys.ML_MODEL_LOAD("mistral-7b-instruct-v1", NULL);
   ```

   This step is optional. The `ML_GENERATE_TABLE` routine loads the specified LLM too. But it takes a bit longer to load the LLM and generate the output when you run it for the first time.

2. In the `ML_GENERATE_TABLE` routine, specify the table columns containing the input queries and for storing the generated text summaries:

   ```
   call sys.ML_GENERATE_TABLE("InputDBName.InputTableName.InputColumn", "OutputDBName.OutputTableName.OutputCo
   ```

   Replace the following:

   - *`InputDBName`*: the name of the database that contains the table column where your input queries are stored.
   - *`InputTableName`*: the name of the table that contains the column where your input queries are stored.
   - *`InputColumn`*: the name of the column that contains input queries.
   - *`OutputDBName`*: the name of the database that contains the table where you want to store the generated outputs. This can be the same as the input database.
   - *`OutputTableName`*: the name of the table where you want to create a new column to store the generated outputs. This can be the same as the input table. If the specified table doesn't exist, a new table is created.
   - *`OutputColumn`*: the name for the new column where you want to store the output generated for the input queries.
   - *`LLM`*: LLM to use, which must be the same as the LLM you loaded in the previous step.
   - *`Language`*: the two-letter `ISO 639-1` code for the language you want to use. Default language is `en`, which is English. To view the list of supported languages, see Languages.

   For example:

   ```
   call sys.ML_GENERATE_TABLE("demo_db.input_table.Input", "demo_db.output_table.Output", JSON_OBJECT("task",
   ```

To learn more about the available routine options, see ML_GENERATE_TABLE Syntax.

4.4 Performing a Vector Search

Using the inbuilt vector store and retrieval-augmented generation (RAG), you can load and query unstructured documents stored in Object Storage using natural language within the HeatWave ecosystem.

The sections in this topic describe how to set up and perform a vector search.

4.4.1 HeatWave Vector Store Overview

This section describes the Vector Store functionality available in HeatWave.

About Vector Store

HeatWave vector store is a relational database that lets you load unstructured data to HeatWave Lakehouse. It automatically parses unstructured data formats, which include PDF, PPT, TXT, HTML, and DOC file formats, from Object Storage. Then, it segments the parsed data, creates vector embeddings, and stores them for HeatWave GenAI to perform semantic searches.

HeatWave vector store uses the native `VECTOR` data type to store unstructured data in a multi-dimensional space. Each point in a vector store represents the vector embedding of the corresponding data. Semantically similar data is placed closer in the vector space.

The large language models (LLMs) available in HeatWave GenAI are trained on publicly available data. Therefore, the responses generated by these LLMs are based on publicly available information. To generate content relevant to your proprietary data, you must store your proprietary enterprise data, which has been converted to vector embeddings, in a vector store. This enables the in-database retrieval-augmented generation (RAG) system to perform a semantic search in the proprietary data stored in the vector stores to find appropriate content, which is then fed to the LLM for generating more accurate and relevant responses.

About Vector Processing

To create vector embeddings, HeatWave GenAI uses in-database embedding models, which are encoders that converts sequence of words and sentences from documents into numerical representations. These numerical values are stored as vector embeddings in the vector store and capture the semantics of the data and relationships to other data.

A vector distance function measures the similarity between vectors by calculating the mathematical distance between two multi-dimensional vectors.

HeatWave GenAI encodes your queries using the same embedding model that is used to encode the ingested data to create the vector store. It then uses the right distance function to find relevant content with similar semantic meaning from the vector store to perform RAG.

4.4.2 Setting Up a Vector Store

This section describes how to generate vector embeddings for files or folders stored in Object Storage, and load the embeddings into a vector store table.

HeatWave GenAI supports the following methods to ingest files from the Object Storage bucket:

- Ingesting Files Using Asynchronous Load

- Ingesting Files Using Auto Parallel Load

Before You Begin

- If not already done, create an Oracle Cloud Infrastructure (OCI) Object Storage bucket for storing files that you want to ingest into the vector store.

  Then, upload the files to the Object Storage bucket.

  Vector store can ingest files in the following formats: PDF, PPT, TXT, HTML, and DOC.

- Connect to your HeatWave Database System.

```
mysqlsh -uAdmin -pPassword -hPrivateIP --sqlc
```

> **Note**
>
> X protocol is not supported for Ingesting Files Using Asynchronous Load. To set up a vector store using this method, ensure that you use the classic MySQL protocol while connecting to the database.

  Replace the following:

  - *`Admin`*: the database system admin name.
  - *`Password`*: the database system password.
  - *`PrivateIP`*: the private IP address of the database system.

Ingesting Files Using Asynchronous Load

The `VECTOR_STORE_LOAD` routine creates and loads vector embeddings asynchronously into the vector store. You can ingest the source files into the vector store using the following methods:

- Using the Uniform Resource Identifier
- Using a Pre-Authenticated Request

Using the Uniform Resource Identifier with Asynchronous Load

This section describes how to load source documents from the Object Storage bucket into the vector table using the uniform resource identifier (URI) of the object.

> **Note**
>
> To use this method, you need to enable the database system to access an Oracle Cloud Infrastructure Object Storage bucket. For more information, see Resource Principals.

To set up a new vector store using an object URI, perform the following steps:

1. To create the vector store table, use a new or existing database:

   ```
   use DBName;
   ```

   Replace *`DBName`* with the database name.

2. If you are loading a vector store table on a database system for the first time, call the following procedure to create a schema used for task management:

```
select mysql_task_management_ensure_schema();
```

3. Optionally, to specify a name for the vector store table and language to use, set the `@options` session variable:

   ```
   set @options = JSON_OBJECT("table_name", "VectorStoreTableName", "language", "Language");
   ```

   Replace the following:

   - *`VectorStoreTableName`*: the name you want for the vector store table.
   - *`Language`*: the two-letter `ISO 639-1` code for the language you want to use. Default language is `en`, which is English. To view the list of supported languages, see Languages.

     > **Note**
     >
     > The `language` parameter is supported in HeatWave `9.0.1-u1` and later versions.

   For example:

   ```
   set @options = JSON_OBJECT("table_name", "demo_embeddings", "language", "en");
   ```

   To learn more about the available routine options, see VECTOR_STORE_LOAD Syntax.

4. To ingest the file from the object storage, create vector embeddings, and load the vector embeddings into HeatWave, use the `VECTOR_STORE_LOAD` routine:

   ```
   call sys.VECTOR_STORE_LOAD("oci://BucketName@Namespace/Path/Filename", @options);
   ```

   Replace the following:

   - *`BucketName`*: the OCI Object Storage bucket name.
   - *`Namespace`*: the name of the Object Storage bucket namespace.
   - *`Path`*: path to the folder that contains the source file.
   - *`Filename`*: the filename with the file extension.

   For example:

   ```
   call sys.VECTOR_STORE_LOAD("oci://demo_bucket@demo_namespace/demo_folder/demo_file.pdf", @options);
   ```

   This creates an asynchronous task that runs in background and loads the vector embeddings into the specified vector store table. The output of the `VECTOR_STORE_LOAD` routine contains the following:

   - An ID of the task which was created.
   - A task query that you can use to track the progress of task.

   If HeatWave GenAI detects multiple files with the same or different file formats in a single load, it creates a separate table for every format it finds. The table name for each format is the specified or default table name followed by the format. For example, `demo_embeddings_pdf` is the name of the table that contains PDF files.

5. After the task is completed, verify that embeddings are loaded in the vector store table:

   ```
   select count(*) from VectorStoreTableName;
   ```

For example:

```
select count(*) from demo_embeddings;
```

If you see a numerical value in the output, your embeddings are successfully loaded in the vector store table.

Using a Pre-Authenticated Request with Asynchronous Load

This section describes how to ingest source documents from the object storage using pre-authenticated requests (PAR). Use this method if OCI Object Storage bucket access is not enabled on your database system.

> **Note**
>
> For confidential data, Using the Uniform Resource Identifier with Asynchronous Load is recommended for ingesting the source files into the vector store as it is a more secure method.

To set up a new vector store, perform the following steps:

1. To create the vector store table, use a new or existing database:

   ```
   use DBName;
   ```

   Replace *`DBName`* with the database name.

2. If you are loading a vector store table on a database system for the first time, call the following procedure to create a schema used for task management:

   ```
   select mysql_task_management_ensure_schema();
   ```

3. Optionally, to specify a name for the vector store table and language to use, set the `@options` session variable:

   ```
   set @options = JSON_OBJECT("table_name", "VectorStoreTableName", "language", "Language");
   ```

   Replace the following:

   - *`VectorStoreTableName`*: the name you want for the vector store table.
   - *`Language`*: the two-letter `ISO 639-1` code for the language you want to use. Default language is `en`, which is English. To view the list of supported languages, see Languages.

     > **Note**
     >
     > The `language` parameter is supported in HeatWave `9.0.1-u1` and later versions.

   For example:

   ```
   set @options = JSON_OBJECT("table_name", "demo_embeddings", "language", "en");
   ```

   To learn more about the available routine options, see VECTOR_STORE_LOAD Syntax.

4. To ingest the file from the object storage, create vector embeddings, and load the vector embeddings into HeatWave, use the `VECTOR_STORE_LOAD` routine:

   ```
   call sys.VECTOR_STORE_LOAD("PAR", @options);
   ```

Replace *PAR* with PAR of the bucket, folder, or file that you want to use to set up the vector store.

To learn how to create PAR for your object storage, see Creating a PAR Request in Object Storage.

> **Note**
>
> If you are creating a PAR for a folder or the object storage, then select *Enable Object Listing* in the *Create Pre-Authenticated Request* dialog to enable object listing.

For example:

```
call sys.VECTOR_STORE_LOAD("https://demo.objectstorage.us-ashburn-1.oci.customer-oci.com/p/demo-url/n/d
```

This creates a task that runs in background and loads the vector embeddings into the specified vector store table. The output of the `VECTOR_STORE_LOAD` routine contains the following:

- An ID of the task which was created.
- A task query that you can use to track the progress of task.

If HeatWave GenAI detects files with different file formats in a single load, it creates a separate table for every format it finds. The table name for each format is the specified or default table name followed by the format. For example, `demo_embeddings_pdf` is the name of the table that contains PDF files.

5. After the task is completed, verify that embeddings are loaded in the vector store table:

```
select count(*) from VectorStoreTableName;
```

For example:

```
select count(*) from demo_embeddings;
```

If you see a numerical value in the output, your embeddings are successfully loaded in the vector store table.

Ingesting Files Using Auto Parallel Load

The `HEATWAVE_LOAD` routine creates and loads vector embeddings into the vector store using auto parallel load.

To ingest files using the `HEATWAVE_LOAD` routine, perform the following steps:

1. In your HeatWave Database System, create and use a new database:

```
create database DBName;
use DBName;
```

Replace *DBName* with the name you want for the new database.

2. To ingest the file from the object store and create vector embeddings in a new vector store table, set the `@dl_tables` session variable:

```
set @dl_tables = '[
  {
    "db_name": "DBName",
    "tables": [
    {
        "table_name": "VectorStoreTableName",
        "engine_attribute": {
```

```
            "dialect": {"format": "FileFormat", "language": "Language"},
            "file": [
    {"par": "PAR"}
  ]
        }
      }
    ]
  }]';
```

Replace the following:

- *DBName*: the database name.
- *VectorStoreTableName*: the name you want for the vector store table where the vector embeddings are stored.
- *FileFormat*: the formats of the files to be ingested into the vector store table. The supported file formats are `html`, `pdf`, `ppt`, `pptx` `txt`, `doc`, and `docx`. To ingest multiple files with different unstructured data file formats into the vector store table in a single load, replace *FileFormat* with `auto_unstructured`.
- *Language*: the two-letter `ISO 639-1` code for the language you want to use. Default language is `en`, which is English. To view the list of supported languages, see Languages.

  > **Note**
  >
  > The `language` parameter is supported in HeatWave `9.0.1-u1` and later versions.

- *PAR*: the pre-authenticated request (PAR) detail of the bucket, folder, or file that you want to use to set up the vector store.

  To learn how to create PAR for your object storage, see Creating a PAR request in Object Storage.

  > **Note**
  >
  > If you are creating a PAR for a folder or the object store, then select *Enable Object Listing* to enable object listing in the *Create Pre-Authenticated Request* dialog while creating the PAR.

For example:

```
set @dl_tables = '[
  {
    "db_name": "demo_db",
    "tables": [
    {
        "table_name": "demo_embeddings",
        "engine_attribute": {
            "dialect": {"format": "pdf", "language": "en"},
            "file": [
    {"par": "https://demo.objectstorage.us-ashburn-1.oci.customer-oci.com/p/demo-url/n/demo/b/demo_bucket/o
  ]
        }
      }
    ]
  }]';
```

3. To prepare for loading the vector embeddings into the HeatWave system, set the `@options` session variable:

```
set @options = JSON_OBJECT('mode', 'normal');
```

4. To load the vector embeddings into HeatWave, use the `HEATWAVE_LOAD` routine:

```
call sys.HEATWAVE_LOAD(CAST(@dl_tables AS JSON), @options);
```

   This creates and stores the vector embeddings in the specified vector store table.

5. Verify that embeddings are loaded in the vector store table:

```
select count(*) from VectorStoreTableName;
```

   For example:

```
select count(*) from demo_embeddings;
```

   If you see a numerical value in the output, your embeddings are successfully loaded in the table.

4.4.3 Updating the Vector Store

To keep up with the changes and updates in the documents in your object storage, you must update the vector embeddings loaded in the vector store table on a regular basis. This ensures that the responses generated by HeatWave GenAI are not only accurate, but also up-to-date. And it deletes the embeddings that are no longer useful.

You can update the vector store table using the following methods:

- Loading Data Incrementally into the Vector Store Table
- Deleting and Recreating the Vector Store Table

Before You Begin

- Complete the steps to set up a vector store.

Loading Data Incrementally into the Vector Store Table

Incremental load refreshes the vector embeddings for documents that have already been ingested into the vector store, and for the new documents that are available in the Object Storage bucket, it creates and loads the vector embeddings into the vector store. For documents that have been deleted from the Object Storage bucket, incremental load deletes the vector embeddings from the vector store.

> **Note**
>
> Incremental loading for vector store tables is supported in HeatWave `9.0.1-u1` and later versions.

To update the embeddings in the vector store table using incremental load, perform the following steps:

1. Check that the vector embeddings are loaded in the vector store table you want to update:

```
select count(*) from VectorStoreTableName;
```

   Replace *`VectorStoreTableName`* with the name of the vector store table you want to update.

   For example:

```
select count(*) from demo_embeddings;
```

If you see a numerical value in the output, the embeddings are loaded in the table.

2. To specify vector store table to update, set the `@dl_tables` session variable:

```
set @dl_tables = '[
  {
    "db_name": "DBName",
    "tables": [
    {
      "table_name": "VectorStoreTableName"
    }]
  }
]';
```

Replace the following:

- *`DBName`*: the name of database that contains the vector store table.
- *`VectorStoreTableName`*: the vector store table name.

For example:

```
set @dl_tables = '[
  {
    "db_name": "demo_db",
    "tables": [
    {
      "table_name": "demo_embeddings"
    }]
  }
]';
```

3. To enable incremental loading, set the `refresh_external_tables` parameter in the `@options` session variable:

```
set @options = JSON_OBJECT("mode", "normal", "refresh_external_tables", true);
```

4. To load the new and updated vector embeddings into the vector store, use the `HEATWAVE_LOAD` routine:

```
call sys.HEATWAVE_LOAD(CAST(@dl_tables AS JSON), @options);
```

This updates the vector store table with the new and updated embeddings.

5. Verify that the embeddings are updated in the vector store table:

```
select count(*) from VectorStoreTableName;
```

For example:

```
select count(*) from demo_embeddings;
```

If you see a numerical value in the output which is different than the one you saw in step 1, then the vector store table is successfully updated. In some cases, this value might not change even though the vector store table is successfully refreshed.

Deleting and Recreating the Vector Store Table

To delete and recreate the vector store table and vector embeddings, perform the following steps:

1. Delete the vector store table:

   ```
   drop table VectorStoreTableName;
   ```

   Replace *VectorStoreTableName* with the vector store table name.

2. To create new embeddings for the updated documents, repeat the steps to set up a vector store.

4.4.4 Running Retrieval-Augmented Generation

HeatWave retrieves content from the vector store and provide that as context to the LLM. This process is called as retrieval-augmented generation or RAG. This helps the LLM to produce more relevant and accurate results for your queries. The `ML_MODEL_LOAD` routine loads the LLM, and the `ML_RAG` routine runs RAG to generate accurate responses for your queries.

If the vector store tables contain information in different languages, then the `ML_RAG` routine filters the retrieved context using the embedding model name and the language used for ingesting files into the vector store table. Both these details are stored as metadata in the vector store tables.

Before You Begin

- Complete the steps to set up a vector store.
- For Running Batch Queries, add the natural-language queries to a column in a new or existing table.

Retrieving Context and Generating Relevant Content

To enter a natural-language query, retrieve the context, and generate accurate results using RAG, perform the following steps:

1. To load the LLM in HeatWave memory, use the `ML_MODEL_LOAD` routine:

   ```
   call sys.ML_MODEL_LOAD("LLM", NULL);
   ```

   Replace *LLM* with the name of the LLM that you want to use. To view the lists of supported LLMs, see HeatWave In-Database LLMs and OCI Generative AI Service LLMs.

   For example:

   ```
   call sys.ML_MODEL_LOAD("mistral-7b-instruct-v1", NULL);
   ```

   This step is optional. The `ML_RAG` routine loads the specified LLM too. But it takes a bit longer to load the LLM and generate the output when you run it for the first time.

2. To specify the table for retrieving the vector embeddings to use as context, set the `@options` session variable:

   ```
   set @options = JSON_OBJECT("vector_store", JSON_ARRAY("DBName.VectorStoreTableName"), "model_options",
   ```

   Replace the following:

   - *DBName*: the name of the database that contains the vector store table.
   - *VectorStoreTableName*: the name of the vector store table.
   - *Language*: the two-letter `ISO 639-1` code for the language you want to use. Default language is `en`, which is English. To view the list of supported languages, see Languages.

> **Note**
>
> The `language` parameter is supported in HeatWave `9.0.1-u1` and later versions.

For example:

```
set @options = JSON_OBJECT("vector_store", JSON_ARRAY("demo_db.demo_embeddings"), "model_options", JSON_OBJ
```

To learn more about the available routine options, see ML_RAG Syntax.

3. To define your natural-language query, set the session `@query` variable:

```
set @query="AddYourQuery";
```

Replace *`AddYourQuery`* with your natural-language query.

For example:

```
set @query="What is AutoML?";
```

4. To retrieve the augmented prompt, use the `ML_RAG` routine:

```
call sys.ML_RAG(@query,@output,@options);
```

5. Print the output:

```
select JSON_PRETTY(@output);
```

Text-based content that is generated by the LLM in response to your query is printed as output. The output generated by RAG is comprised of two parts:

- The text section contains the text-based content generated by the LLM as a response for your query.
- The citations section shows the segments and documents it referred to as context.

The output looks similar to the following:

```
"text": " AutoML is a machine learning technique that automates the process
of selecting, training, and evaluating machine learning models. It involves
using algorithms and techniques to automatically identify the best model
for a given dataset and optimize its hyperparameters without requiring manual
intervention from data analysts or ML practitioners. AutoML can be used in
various stages of the machine learning pipeline, including data preprocessing,
feature engineering, model selection, hyperparameter tuning,
and model evaluation.",
"citations": [
  {
    "segment": "Oracle AutoML also produces high quality models very efficiently,
    which is achieved through a scalable design and intelligent choices that
    reduce trials at each stage in the pipeline.\n Scalable design: The Oracle
    AutoML pipeline is able to exploit both HeatWave internode and intranode
    parallelism, which improves scalability and reduces runtime.",
    "distance": 0.4262576103210449,
    "document_name": "https://objectstorage.Region.oraclecloud.com/n/Namespace/b/BucketName/o/Path/Filename
  },
  {
    "segment": "The HeatWave AutoML ML_TRAIN routine leverages Oracle AutoML
    technology to automate the process of training a machine learning model.
    Oracle AutoML replaces the laborious and time consuming tasks of the data
    analyst whose workflow is as follows:\n1. Selecting a model from a large
    number of viable candidate models.\n2.\n99",
```

```
    "distance": 0.4311879277229309,
    "document_name": " https://objectstorage. Region.oraclecloud.com/n/Namespace/b/BucketName/o/Path/Fi
  },
  {
    "segment": "3.1 HeatWave AutoML Features HeatWave AutoML makes it easy to
    use machine learning, whether you are a novice user or an experienced ML
    practitioner. You provide the data, and HeatWave AutoML analyzes the
    characteristics of the data and creates an optimized machine learning model
    that you can use to generate predictions and explanations.",
    "distance": 0.4441382884979248,
    "document_name": "https://objectstorage. Region.oraclecloud.com/n/Namespace/b/BucketName/o/Path/Fil
  }
],
"vector_store": [
  "demo_db.demo_embeddings"
]
```

To continue running more queries in the same session, repeat steps 3 to 5.

Retrieving Context Without Generating Content

To enter a natural-language query and retrieve the context without generating a response for the query, perform the following steps:

1. To load the LLM in HeatWave memory, use the `ML_MODEL_LOAD` routine:

   ```
   call sys.ML_MODEL_LOAD("LLM", NULL);
   ```

   Replace *`LLM`* with the name of the LLM that you want to use. To view the lists of supported LLMs, see HeatWave In-Database LLMs and OCI Generative AI Service LLMs.

   For example:

   ```
   call sys.ML_MODEL_LOAD("mistral-7b-instruct-v1", NULL);
   ```

   This step is optional. The `ML_RAG` routine loads the specified LLM too. But it takes a bit longer to load the LLM and generate the output when you run it for the first time.

2. To specify the table for retrieving the vector embeddings and to skip generation of content, set the `@options` session variable:

   ```
   set @options = JSON_OBJECT("vector_store", JSON_ARRAY("DBName.VectorStoreTableName"), "skip_generate",
   ```

   Replace the following:

   - *`DBName`*: the name of the database that contains the vector store table.
   - *`VectorStoreTableName`*: the name of the vector store table.
   - *`Language`*: the two-letter `ISO 639-1` code for the language you want to use. Default language is `en`, which is English. To view the list of supported languages, see Languages.

     > **Note**
     >
     > The `language` parameter is supported in HeatWave `9.0.1-u1` and later versions.

   For example:

   ```
   set @options = JSON_OBJECT("vector_store", JSON_ARRAY("demo_db.demo_embeddings"), "skip_generate", true
   ```

3. To define your natural-language query, set the session `@query` variable:

```
set @query="AddYourQuery";
```

Replace *`AddYourQuery`* with your natural-language query.

For example:

```
set @query="What is AutoML?";
```

4. To retrieve the augmented prompt, use the `ML_RAG` routine:

```
call sys.ML_RAG(@query,@output,@options);
```

5. Print the output:

```
select JSON_PRETTY(@output);
```

Semantically similar text segments used as content for the query and the name of the documents they were found in are printed as output.

The output looks similar to the following:

```
{
    "citations": [
    {
      "segment": "Oracle AutoML also produces high quality models very efficiently,
      which is achieved through a scalable design and intelligent choices that
      reduce trials at each stage in the pipeline.\n Scalable design: The Oracle
      AutoML pipeline is able to exploit both HeatWave internode and intranode
      parallelism, which improves scalability and reduces runtime.",
      "distance": 0.4262576103210449,
      "document_name": "https://objectstorage.Region.oraclecloud.com/n/Namespace/b/BucketName/o/Path/Filena
    },
    {
      "segment": "The HeatWave AutoML ML_TRAIN routine leverages Oracle AutoML
      technology to automate the process of training a machine learning model.
      Oracle AutoML replaces the laborious and time consuming tasks of the data
      analyst whose workflow is as follows:\n1. Selecting a model from a large
      number of viable candidate models.\n2.\n99",
      "distance": 0.4311879277229309,
      "document_name": " https://objectstorage. Region.oraclecloud.com/n/Namespace/b/BucketName/o/Path/File
    },
    {
      "segment": "3.1 HeatWave AutoML Features HeatWave AutoML makes it easy to
      use machine learning, whether you are a novice user or an experienced ML
      practitioner. You provide the data, and HeatWave AutoML analyzes the
      characteristics of the data and creates an optimized machine learning model
      that you can use to generate predictions and explanations.",
      "distance": 0.4441382884979248,
      "document_name": "https://objectstorage. Region.oraclecloud.com/n/Namespace/b/BucketName/o/Path/Filen
    }
  ],
  "vector_store": [
    "demo_db.demo_embeddings"
  ]
}
```

To continue running more queries in the same session, repeat steps 3 to 5.

Running Batch Queries

To run multiple RAG queries in parallel, use the `ML_RAG_TABLE` routine. This method is faster than running the `ML_RAG` routine multiple times.

> **Note**
>
> The `ML_RAG_TABLE` routine is supported in HeatWave `9.0.1-u1` and later versions.

To run batch queries using `ML_RAG_TABLE`, perform the following steps:

1. To load the LLM in HeatWave memory, use the `ML_MODEL_LOAD` routine:

   ```
   call sys.ML_MODEL_LOAD("LLM", NULL);
   ```

   Replace *`LLM`* with the name of the LLM that you want to use. To view the lists of supported LLMs, see HeatWave In-Database LLMs and OCI Generative AI Service LLMs.

   For example:

   ```
   call sys.ML_MODEL_LOAD("mistral-7b-instruct-v1", NULL);
   ```

   This step is optional. The `ML_RAG_TABLE` routine loads the specified LLM too. But it takes a bit longer to load the LLM and generate the output when you run it for the first time.

2. To specify the table for retrieving the vector embeddings to use as context, set the `@options` session variable:

   ```
   set @options = JSON_OBJECT("vector_store", JSON_ARRAY("DBName.VectorStoreTableName"), "model_options",
   ```

   Replace the following:

   - *`DBName`*: the name of the database that contains the vector store table.
   - *`VectorStoreTableName`*: the name of the vector store table.
   - *`Language`*: the two-letter `ISO 639-1` code for the language you want to use. Default language is `en`, which is English. To view the list of supported languages, see Languages.

   For example:

   ```
   set @options = JSON_OBJECT("vector_store", JSON_ARRAY("demo_db.demo_embeddings"), "model_options", JSON
   ```

   To learn more about the available routine options, see ML_RAG_TABLE Syntax.

3. In the `ML_RAG_TABLE` routine, specify the table columns containing the input queries and for storing the generated outputs:

   ```
   call sys.ML_RAG_TABLE("InputDBName.InputTableName.InputColumn", "OutputDBName.OutputTableName.OutputCol
   ```

   Replace the following:

   - *`InputDBName`*: the name of the database that contains the table column where your input queries are stored.
   - *`InputTableName`*: the name of the table that contains the column where your input queries are stored.
   - *`InputColumn`*: the name of the column that contains input queries.
   - *`OutputDBName`*: the name of the database that contains the table where you want to store the generated outputs. This can be the same as the input database.

- *`OutputTableName`*: the name of the table where you want to create a new column to store the generated outputs. This can be the same as the input table. If the specified table doesn't exist, a new table is created.
- *`OutputColumn`*: the name for the new column where you want to store the output generated for the input queries.

For example:

```
call sys.ML_RAG_TABLE("demo_db.input_table.Input", "demo_db.output_table.Output", @options);
```

4.5 Running HeatWave Chat

You can use HeatWave Chat to simulate human-like conversations where you can get responses for multiple queries in the same session. HeatWave Chat is a conversational agent that utilizes large language models (LLMs) to understand inputs and responds in natural manner. It extends the text generation by using a chat history that lets you ask follow-up questions, and uses the vector search functionality to draw its knowledge from the inbuilt vector store. The responses generated by HeatWave Chat are quick and secure as all the communication and processing happens within the HeatWave service.

The sections in this topic describe how to run and manage HeatWave Chat.

4.5.1 Running HeatWave GenAI Chat

When you run HeatWave Chat, it automatically loads the `mistral-7b-instruct-v1` LLM.

By default, HeatWave Chat searches for an answer to a query across all ingested documents by automatically discovering available vector stores, and returns the answer along with relevant citations. you can limit the scope of search to specific document collections available in certain vector stores or specify documents to include in the search.

If the vector store tables contain information in different languages, then similar to `ML_RAG`, the `HEATWAVE_CHAT` routine also filters the retrieved context using the embedding model name and the language used for ingesting files into the vector store table.

If you do not have a vector store set up, then HeatWave Chat uses information available in public data sources to generate a response for your query.

Before You Begin

- If you want to extend the vector search functionality and ask specific questions about the information available in your proprietary documents that are stored in the vector store, complete the steps to set up a vector store.

Running the Chat

To run HeatWave Chat, perform the following steps:

1. To delete previous chat output and state, if any, reset the `@chat_options` session variable:

   ```
   set @chat_options=NULL;
   ```

   To use a language other than English, set the `language` model option of the `@chat_options` session variable:

   ```
   set @chat_options = JSON_OBJECT("model_options", JSON_OBJECT("language", "Language"));
   ```

Replace *Language* with the two-letter ISO 639-1 code for the language you want to use. Default language is en, which is English. To view the list of supported languages, see Languages.

For example, to use French set language to fr:

> **Note**
>
> The language parameter is supported in HeatWave 9.0.1-u1 and later versions.

```
set @chat_options = JSON_OBJECT("model_options", JSON_OBJECT("language", "fr"));
```

This resets the @chat_options session variables and specifies the language for the chat.

2. Then, add your query to HeatWave Chat by using the HEATWAVE_CHAT routine:

```
call sys.HEATWAVE_CHAT("YourQuery");
```

For example:

```
call sys.HEATWAVE_CHAT("What is HeatWave AutoML?");
```

The output looks similar to the following:

```
|  HeatWave AutoML is a feature of MySQL HeatWave that makes it easy
to use machine learning, whether you are a novice user or an
experienced ML practitioner. It analyzes the characteristics of the
data and creates an optimized machine learning model that can be used
to generate predictions and explanations. The data and models never
leave MySQL HeatWave, saving time and effort while keeping the data
and models secure. HeatWave AutoML is optimized for HeatWave shapes
and scaling, and all processing is performed on the HeatWave Cluster. |
```

Repeat this step to ask follow-up questions using the HEATWAVE_CHAT routine:

```
call sys.HEATWAVE_CHAT("What learning algorithms does it use?");
```

The output looks similar to the following:

```
|  HeatWave AutoML uses a variety of machine learning algorithms. It
leverages Oracle AutoML technology which includes a range of algorithms
such as decision trees, random forests, neural networks, and support vector
machines (SVMs). The specific algorithm used by HeatWave AutoML depends on
the characteristics of the data being analyzed and the goals of the model
being created. |
```

4.5.2 Viewing Chat Session Details

This section describes how to view a chat session details.

Before You Begin

- Complete the steps for Running the Chat.

Viewing Details

To view the chat session details, perform the following step:

- Inspect the @chat_options session variable:

```
select JSON_PRETTY(@chat_options);
```

The output includes the following details about a chat session:

- *Vector store tables:* in the database which were referenced by HeatWave Chat.
- *Text segments:* that were retrieved from the vector store and used as context to prepare responses for your queries.
- *Chat history:* which includes both your queries and responses generated by HeatWave Chat.
- *LLM details:* which was used by the routine to generate the responses.

The output looks similar to the following:


```
| {
  "tables": [
    {
      "table_name": "`demo_embeddings`",
      "schema_name": "`demo_db`"
    }
  ],
  "response": " HeatWave AutoML uses a variety of machine learning algorithms. It
  leverages Oracle AutoML technology which includes a range of algorithms such as
  decision trees, random forests, neural networks, and support vector machines (SVMs).
  The specific algorithm used by HeatWave AutoML depends on the characteristics of
  the data being analyzed and the goals of the model being created.",
  "documents": [
    {
      "id": "https://objectstorage.Region.oraclecloud.com/n/Namespace/b/BucketName/o/Path/heatwave-en.a4.pdf",
      "title": "heatwave-en.a4.pdf",
      "segment": "3.1 HeatWave AutoML Features HeatWave AutoML makes it easy to use
      machine learning, whether you are a novice user or an experienced ML practitioner.
      You provide the data, and HeatWave AutoML analyzes the characteristics of the
      data and creates an optimized machine learning model that you can use to
      generate predictions and explanations.",
      "distance": 0.18456566333770752
    },
    {
      "id": "https://objectstorage.Region.oraclecloud.com/n/Namespace/b/BucketName/o/Path/heatwave-en.a4.pdf",
      "title": "heatwave-en.a4.pdf",
      "segment": "The HeatWave AutoML ML_TRAIN routine leverages Oracle AutoML technology
      to automate the process of training a machine learning model. Oracle AutoML replaces
      the laborious and time consuming tasks of the data analyst whose workflow is as
      follows:\n1. Selecting a model from a large number of viable candidate models.\n2.
      For each model, tuning hyperparameters.\n3. Selecting only predictive features
      to speed up the pipeline and reduce over-fitting.\n99",
      "distance": 0.22687965631484985
    },
    {
      "id": "https://objectstorage.Region.oraclecloud.com/n/Namespace/b/BucketName/o/Path/heatwave-en.a4.pdf",
      "title": "heatwave-en.a4.pdf",
      "segment": "3.1.1 HeatWave AutoML Supervised Learning\nHeatWave AutoML supports supervised
      machine learning. That is, it creates a machine learning model by analyzing a labeled
      dataset to learn patterns that enable it to predict labels based on the features of the
      dataset. For example, this guide uses the Census Income Data Set in its examples, where
      features such as age, education, occupation, country, and so on, are used to predict the
      income of an individual (the label).",
      "distance": 0.2275727391242981
    }
  ],
  "chat_history": [
    {
      "user_message": "What is HeatWave AutoML?",
      "chat_query_id": "99471681-387f-11ef-96d7-020017331ed6",
      "chat_bot_message": " HeatWave AutoML is a feature of MySQL HeatWave that makes
      it easy to use machine learning, whether you are a novice user or an experienced ML
      practitioner. It analyzes the characteristics of the data and creates an optimized
```

```
      machine learning model that can be used to generate predictions and explanations. The data
      and models never leave MySQL HeatWave, saving time and effort while keeping the data
      and models secure. HeatWave AutoML is optimized for HeatWave shapes and scaling,
      and all processing is performed on the HeatWave Cluster."
    },
    {
      "user_message": "What learning algorithms does it use?",
      "chat_query_id": "c59140f5-387f-11ef-96d7-020017331ed6",
      "chat_bot_message": " HeatWave AutoML uses a variety of machine learning algorithms.
      It leverages Oracle AutoML technology which includes a range of algorithms such
      as decision trees, random forests, neural networks, and support vector machines (SVMs).
      The specific algorithm used by HeatWave AutoML depends on the characteristics of the
      data being analyzed and the goals of the model being created."
    }
  ],
  "model_options": {
    "model_id": "mistral-7b-instruct-v1"
  },
  "request_completed": true
} |
```

4.6 Generating Vector Embeddings

This section describes how to generate vector embeddings using the `ML_EMBED_ROW` HeatWave GenAI routine. Vector embeddings are a numerical representation of the text that capture the semantics of the data and relationships to other data. You can pass the text string in the routine manually or use data from tables in your database. To embed multiple rows of text stored in a table in a single run, you can even run a batch query.

Using this method, you can create vector embedding tables that you can use to perform similarity searches using the `DISTANCE()` function, without setting up a vector store.

> **Note**
>
> This method does not support embedding unstructured data. To learn how to create vector embeddings for unstructured data, see Section 4.4, “Performing a Vector Search”.

Before You Begin

- Connect to your HeatWave Database System.
- For Running Batch Queries, add the text that you want to embed to a column in a new or existing table.

Generating a Vector Embedding for Specified Text

To generate a vector embedding using HeatWave GenAI, perform the following steps:

1. To define the text that you want to encode, set the `@text` session variable:

   ```
   set @text="TextToEncode";
   ```

   Replace *`TextToEncode`* with the text that you want to encode. For example:

   ```
   set @text="HeatWave GenAI is a feature of HeatWave that lets you communicate with unstructured data in
   ```

2. To generate a vector embedding for the specified text, pass the text to the embedding model using the `ML_EMBED_ROW` routine:

   ```
   select sys.ML_EMBED_ROW(@text, JSON_OBJECT("model_id", "EmbeddingModel")) into @text_embedding;
   ```

Replace *EmbeddingModel* with ID of the embedding model you want to use. To view the lists of supported models, see HeatWave In-Database Embedding Models and OCI Generative AI Service Embedding Models.

For example:

```
select sys.ML_EMBED_ROW(@text, JSON_OBJECT("model_id", "all_minilm_l12_v2")) into @text_embedding;
```

The routine returns a VECTOR, and this commad stores it in the @text_embedding session variable.

3. Print the vector embedding stored in the @text_embedding session variable:

```
select @text_embedding;
```

The output, which is a binary representation of the specified text, looks similar to the following:

```
+-----------------------------------------------------+
| 0xEB0FE93C21737D3C4ED2F0BC6DCC06BD0668DBBB15D1ABBBDF3E03BD09DC21BC229512BD06B602BD4824F6BCFFEF70BDFFB53
```

Running Batch Queries

To encode multiple rows of text strings stored in a table column, in parallel, use the ML_EMBED_TABLE routine. This method is faster than running the ML_EMBED_ROW routine multiple times.

> **Note**
>
> The ML_EMBED_TABLE routine is supported in HeatWave 9.0.1-u1 and later versions.

- Run batch queries using ML_EMBED_TABLE:

```
call sys.ML_EMBED_TABLE("InputDBName.InputTableName.InputColumn", "OutputDBName.OutputTableName.OutputColumn
```

Replace the following:

- *InputDBName*: the name of the database that contains the table column where your input queries are stored.
- *InputTableName*: the name of the table that contains the column where your input queries are stored.
- *InputColumn*: the name of the column that contains input queries.
- *OutputDBName*: the name of the database that contains the table where you want to store the generated outputs. This can be the same as the input database.
- *OutputTableName*: the name of the table where you want to create a new column to store the generated outputs. This can be the same as the input table. If the specified table doesn't exist, a new table is created.
- *OutputColumn*: the name for the new column where you want to store the output generated for the input queries.
- *EmbeddingModel*: ID of the embedding model to use. To view the lists of supported models, see HeatWave In-Database Embedding Models and OCI Generative AI Service Embedding Models.

For example:

```
call sys.ML_EMBED_TABLE("demo_db.input_table.Input", "demo_db.output_table.Output", JSON_OBJECT("model_i
```

4.7 HeatWave GenAI Routines

HeatWave GenAI routines reside in the MySQL `sys` schema.

4.7.1 ML_GENERATE

The `ML_GENERATE` routine uses the specified large language model (LLM) to generate text-based content as a response for the given natural-language query.

ML_GENERATE Syntax

```
mysql> select sys.ML_GENERATE('QueryInNaturalLanguage', [options]);

options: {
  JSON_OBJECT('key','value'[,'key','value'] ...)
    'key','value': {
    ['task', {'generation'|'summarization'}]
    ['model_id', {'mistral-7b-instruct-v1'|'llama2-7b-v1'|'llama3-8b-instruct-v1'|'cohere.command-r-plus'|
    ['context', 'Context']
    ['language', 'Language']
    ['temperature', Temperature]
    ['max_tokens', MaxTokens]
    ['top_k', K]
    ['top_p', P]
    ['repeat_penalty', RepeatPenalty]
    ['frequency_penalty', FrequencyPenalty]
    ['presence_penalty', PresencePenalty]
    ['stop_sequences', JSON_ARRAY('StopSequence1'[,'StopSequence2'] ...)]
    }
}
```

Following are `ML_GENERATE` parameters:

- *`QueryInNaturalLanguage`*: specifies the natural-language query that is passed to the large language model (LLM) handle.
- `options`: specifies optional parameters as key-value pairs in JSON format. It can include the following parameters:
  - `task`: specifies the task expected from the LLM. Default value is `generation`. Possible values are:
    - `generation`: generates text-based content.
    - `summarization`: generates a summary for existing text-based content.
  - `model_id`: specifies the LLM to use for the task. Default value is `mistral-7b-instruct-v1`. Possible values are:
    - `mistral-7b-instruct-v1`
    - `llama2-7b-v1`. This LLM is no longer supported in HeatWave `9.0.1-u1` and later versions.
    - `llama3-8b-instruct-v1`. This LLM is available in HeatWave `9.0.1-u1` and later versions.
    - `cohere.command-r-plus`. This LLM is available in HeatWave `9.0.1-u1` and later versions.
    - `cohere.command-r-16k`. This LLM is available in HeatWave `9.0.1-u1` and later versions.

- `meta.llama-3-70b-instruct`. This LLM is available in HeatWave `9.0.1-u1` and later versions.

To view the lists of supported LLMs, see HeatWave In-Database LLMs and OCI Generative AI Service LLMs.

> **Note**
>
> The `summarization` task supports HeatWave In-Database LLMs only.

- `context`: specifies the context to be used for augmenting the query and guide the text generation of the LLM. Default value is `NULL`.
- `language`: specifies the language to be used for writing queries, ingesting documents, and generating the output. To set the value of the `language` parameter, use the two-letter `ISO 639-1` code for the language. This parameter is supported in HeatWave `9.0.1-u1` and later versions.

  Default value is `en`.

  For possible values, to view the list of supported languages, see Languages.
- `temperature`: specifies a non-negative float that tunes the degree of randomness in generation. Lower temperatures mean less random generations.

  Default value is `0`.

  - `0` for the HeatWave In-Database LLMs.
  - `0.3` for `cohere.command-r-plus` and `cohere.command-r-16k`.
  - `1.0` for `meta.llama-3-70b-instruct`.

  Possible values are float values between `0` and `5`.

  - `0` and `5` for the HeatWave In-Database LLMs.
  - `0` and `1` for `cohere.command-r-plus` and `cohere.command-r-16k`.
  - `0` and `2` for `meta.llama-3-70b-instruct`.

  It is suggested that:

  - To generate the same output for a particular prompt every time you run it, set the temperature to `0`.
  - To generate a random new statement for a particular prompt every time you run it, increase the temperature.
- `max_tokens`: specifies the maximum number of tokens to predict per generation using an estimate of three tokens per word. Default value is `256`. Possible values are:

  - For `mistral-7b-instruct-v1`, integer values between `1` and `8000`.
  - For `llama2-7b-v1`, integer values between`1` and `4096`.
  - For `llama3-8b-instruct-v1`, integer values between `1` and `4096`.
  - For `cohere.command-r-plus`, integer values between `1` and `4000`.

  - For `cohere.command-r-16k`, integer values between `1` and `4000`.
  - For `meta.llama-3-70b-instruct`, integer values between `1` and `8000`.

- `top_k`: specifies the number of top most likely tokens to consider for text generation at each step. Default value is `40`, which means that top 40 most likely tokens are considered for text generation at each step. Possible values are integer values between `0` and `32000`.
- `top_p`: specifies a number, `p`, and ensures that only the most likely tokens with the sum of probabilities `p` are considered for generation at each step. A higher value of `p` introduces more randomness into the output. Default value is `0.95`. Possible values are float values between `0` and `1`.
  - To disable this method, set to `1.0` or `0`.
  - To eliminate tokens with low likelihood, assign `p` a lower value. For example, if set to `0.1`, tokens within top 10% probability are included.
  - To include tokens with low likelihood, assign `p` a higher value. For example, if set to `0.9`, tokens within top 90% probability are included.

  If you are also specifying the `top_k` parameter, the LLM considers only the top tokens whose probabilities add up to `p` percent. It ignores the rest of the `k` tokens.
- `repeat_penalty`: assigns a penalty when a token appears repeatedly. High penalties encourage less repeated tokens and produce more random outputs. Default value is `1.1`. Possible values are float values between `0` and `2`.

  > **Note**
  >
  > This parameter is supported for HeatWave In-Database LLMs only.

- `frequency_penalty`: assigns a penalty when a token appears frequently. High penalties encourage less repeated tokens and produce more random outputs. Default value is `0`. Possible values are float values between `0` and `1`.
- `presence_penalty`: assigns a penalty to each token when it appears in the output to encourage generating outputs with tokens that haven't been used. This is similar to `frequency_penalty`, except that this penalty is applied equally to all tokens that have already appeared, irrespective of their exact frequencies.

  > **Note**
  >
  > This parameter is supported for OCI Generative AI Service LLMs only.

  Default value is `0`. Possible values are:

  - For `cohere.command-r-plus` and `cohere.command-r-16k`, float values between `0` and `1`.
  - For `meta.llama-3-70b-instruct`, float values between `-2` and `2`.
- `stop_sequences`: specifies a list of characters such as a word, a phrase, a newline, or a period that tells the LLM when to end the generated output. If you have more than one stop sequence, then the LLM stops when it reaches any of those sequences. Default value is `NULL`.

Syntax Examples

- Generating text-based content in English using the `mistral-7b-instruct-v1` model:

```
mysql> select sys.ML_GENERATE("What is AI?", JSON_OBJECT("task", "generation", "model_id", "mistral-7b-instr
```

- Summarizing English text using the `mistral-7b-instruct-v1` model:

```
mysql> select sys.ML_GENERATE(@text, JSON_OBJECT("task", "summarization", "model_id", "mistral-7b-instruct-v
```

Where, `@text` is set as shown below:

```
set @text="Artificial Intelligence (AI) is a rapidly growing field that has the potential to
revolutionize how we live and work. AI refers to the development of computer systems that can
perform tasks that typically require human intelligence, such as visual perception, speech
recognition, decision-making, and language translation.\n\nOne of the most significant developments in
AI in recent years has been the rise of machine learning, a subset of AI that allows computers to learn
from data without being explicitly programmed. Machine learning algorithms can analyze vast amounts
of data and identify patterns, making them increasingly accurate at predicting outcomes and making
decisions.\n\nAI is already being used in a variety of industries, including healthcare, finance, and
transportation. In healthcare, AI is being used to develop personalized treatment plans for patients
based on their medical history and genetic makeup. In finance, AI is being used to detect fraud and make
investment recommendations. In transportation, AI is being used to develop self-driving cars and improve
traffic flow.\n\nDespite the many benefits of AI, there are also concerns about its potential impact on
society. Some worry that AI could lead to job displacement, as machines become more capable of performing
tasks traditionally done by humans. Others worry that AI could be used for malicious ";
```

4.7.2 ML_GENERATE_TABLE

The `ML_GENERATE_TABLE` routine runs multiple text generation or summarization queries in a batch, in parallel. The output generated for every input query is the same as the output generated by the `ML_GENERATE` routine.

This routine is available in HeatWave `9.0.1-u1` and later versions.

ML_GENERATE_TABLE Syntax

```
mysql> call sys.ML_GENERATE_TABLE('InputTableColumn', 'OutputTableColumn', [options]);

options: {
  JSON_OBJECT('key','value'[,'key','value'] ...)
    'key','value': {
    ['task', {'generation'|'summarization'}]
    ['model_id', {'mistral-7b-instruct-v1'|'llama2-7b-v1'|'llama3-8b-instruct-v1'|'cohere.command-r-plus'|'coh
    ['context_column', 'ContextColumn']
    ['language', 'Language']
    ['temperature', Temperature]
    ['max_tokens', MaxTokens]
    ['top_k', K]
    ['top_p', P]
    ['repeat_penalty', RepeatPenalty]
    ['frequency_penalty', FrequencyPenalty]
    ['presence_penalty', PresencePenalty]
    ['stop_sequences', JSON_ARRAY('StopSequence1'[,'StopSequence2'] ...)]
    ['batch_size', BatchSize]
    }
}
```

Following are `ML_GENERATE_TABLE` parameters:

- *`InputTableColumn`*: specifies the names of the input database, table, and column that contains the natural-language queries. The *`InputTableColumn`* is specified in the following format: *`DBName.TableName.ColumnName`*.

  - The specified input table can be an internal or external table.
  - The specified input table must already exist, must not be empty, and must have a primary key.
  - The input column must already exist and must contain `text` or `varchar` values.
  - The input column must not be a part of the primary key and must not have `NULL` values or empty strings.
  - There must be no backticks used in the *`DBName`*, *`TableName`*, or *`ColumnName`* and there must be no period used in the *`DBName`* or *`TableName`*.
- *`OutputTableColumn`*: specifies the names of the database, table, and column where the generated text-based response is stored. The *`OutputTableColumn`* is specified in the following format: *`DBName`*.*`TableName`*.*`ColumnName`*.
  - The specified output table must be an internal table.
  - If the specified output table already exists, then it must be the same as the input table. And, the specified output column must not already exist in the input table. A new JSON column is added to the table. External tables are read only. So if input table is an external table, then it cannot be used to store the output.
  - If the specified output table doesn't exist, then a new table is created. The new output table has key columns which contains the same primary key values as the input table and a JSON column that stores the generated text-based responses.
  - There must be no backticks used in the *`DBName`*, *`TableName`*, or *`ColumnName`* and there must be no period used in the *`DBName`* or *`TableName`*.
- `options`: specifies optional parameters as key-value pairs in JSON format. It can include the following parameters:
  - `task`: specifies the task expected from the large language model (LLM). Default value is `generation`. Possible values are:
    - `generation`: generates text-based content.
    - `summarization`: generates a summary for existing text-based content.
  - `model_id`: specifies the LLM to use for the task. Default value is `mistral-7b-instruct-v1`. Possible values are:
    - `mistral-7b-instruct-v1`
    - `llama2-7b-v1`
    - `llama3-8b-instruct-v1`
    - `cohere.command-r-plus`
    - `cohere.command-r-16k`
    - `meta.llama-3-70b-instruct`

    To view the lists of supported LLMs, see HeatWave In-Database LLMs and OCI Generative AI Service LLMs.

> **Note**
>
> The `summarization` task supports HeatWave In-Database LLMs only.

- `context_column`: specifies the table column that contains the context to be used for augmenting the queries and guiding the text generation of the LLM. The column must be specified in the following

format: *`DBName`*.*`TableName`*.*`ColumnName`*. The specified column must be an existing column in the input table. Default value is `NULL`.

- `language`: specifies the language to be used for writing queries, ingesting documents, and generating the output. To set the value of the `language` parameter, use the two-letter `ISO 639-1` code for the language.

  Default value is `en`.

  For possible values, to view the list of supported languages, see Languages.

- `temperature`: specifies a non-negative float that tunes the degree of randomness in generation. Lower temperatures mean less random generations.

  Default value is `0`.

  - `0` for the HeatWave In-Database LLMs.
  - `0.3` for `cohere.command-r-plus` and `cohere.command-r-16k`.
  - `1.0` for `meta.llama-3-70b-instruct`.

  Possible values are float values between `0` and `5`.

  - `0` and `5` For the HeatWave In-Database LLMs.
  - `0` and `1` for `cohere.command-r-plus` and `cohere.command-r-16k`.
  - `0` and `2` for `meta.llama-3-70b-instruct`.

  It is suggested that:

  - To generate the same output for a particular prompt every time you run it, set the temperature to `0`.
  - To generate a random new statement for a particular prompt every time you run it, increase the temperature.

- `max_tokens`: specifies the maximum number of tokens to predict per generation using an estimate of three tokens per word. Default value is `256`. Possible values are:

  - For `mistral-7b-instruct-v1`, integer values between `1` and `8000`.
  - For `llama2-7b-v1`, integer values between`1` and `4096`.
  - For `llama3-8b-instruct-v1`, integer values between `1` and `4096`.
  - For `cohere.command-r-plus`, integer values between `1` and `4000`.
  - For `cohere.command-r-16k`, integer values between `1` and `4000`.
  - For `meta.llama-3-70b-instruct`, integer values between `1` and `8000`.

- `top_k`: specifies the number of top most likely tokens to consider for text generation at each step. Default value is `40`, which means that top 40 most likely tokens are considered for text generation at each step. Possible values are integer values between `0` and `32000`.

- `top_p`: specifies a number, `p`, and ensures that only the most likely tokens with the sum of probabilities `p` are considered for generation at each step. A higher value of `p` introduces more randomness into the output. Default value is `0.95`. Possible values are float values between `0` and `1`.

  - To disable this method, set to `1.0` or `0`.
  - To eliminate tokens with low likelihood, assign `p` a lower value. For example, if set to `0.1`, tokens within top 10% probability are included.
  - To include tokens with low likelihood, assign `p` a higher value. For example, if set to `0.9`, tokens within top 90% probability are included.

  If you are also specifying the `top_k` parameter, the LLM considers only the top tokens whose probabilities add up to `p` percent. It ignores the rest of the `k` tokens.

- `repeat_penalty`: assigns a penalty when a token appears repeatedly. High penalties encourage less repeated tokens and produce more random outputs. Default value is `1.1`. Possible values are float values between `0` and `2`.

  > **Note**
  >
  > This parameter is supported for HeatWave In-Database LLMs only.

- `frequency_penalty`: assigns a penalty when a token appears frequently. High penalties encourage less repeated tokens and produce more random outputs. Default value is `0`. Possible values are float values between `0` and `1`.

- `presence_penalty`: assigns a penalty to each token when it appears in the output to encourage generating outputs with tokens that haven't been used. This is similar to `frequency_penalty`, except that this penalty is applied equally to all tokens that have already appeared, irrespective of their exact frequencies.

  > **Note**
  >
  > This parameter is supported for OCI Generative AI Service LLMs only.

  Default value is `0`. Possible values are:

  - For `cohere.command-r-plus` and `cohere.command-r-16k`, float values between `0` and `1`.
  - For `meta.llama-3-70b-instruct`, float values between `-2` and `2`.

- `stop_sequences`: specifies a list of characters such as a word, a phrase, a newline, or a period that tells the LLM when to end the generated output. If you have more than one stop sequence, then the LLM stops when it reaches any of those sequences. Default value is `NULL`.

- `batch_size`: specifies the batch size for the routine. This parameter is supported for internal tables only. Default value is `1000`. Possible values are integer values between `1` and `1000`.

Syntax Examples

- Generating English text-based content in a batch using the `mistral-7b-instruct-v1` model:

```
mysql> call sys.ML_GENERATE_TABLE("demo_db.input_table.Input", "demo_db.output_table.Output", JSON_OBJECT("t
```

In this example, the routine takes input queries stored in the `demo_db.input_table.Input` column, and creates a new column `demo_db.output_table.Output` where it stores the generated text-based responses.

- Summarizing English text in a batch using the `mistral-7b-instruct-v1` model:

```
mysql> call sys.ML_GENERATE_TABLE("demo_db.input_table.Input", "demo_db.output_table.Output", JSON_OBJEC
```

In this example, the routine takes input queries stored in the `demo_db.input_table.Input` column, and creates a new column `demo_db.output_table.Output` where it stores the generated text summaries.

4.7.3 VECTOR_STORE_LOAD

The `VECTOR_STORE_LOAD` routine generates vector embedding for the specified files or folders that are stored in the Object Storage bucket, and loads the embeddings into a new vector store table.

This routine creates an asynchronous task which loads vector store tables in the background. It also returns a query that you can run to track the status of the vector store load task that is running in the background.

> **Note**
>
> It is recommended that you create five or less vector store load tasks at a time. Too many tasks running at the same time might cause overloading issues.

VECTOR_STORE_LOAD Syntax

```
mysql> call sys.VECTOR_STORE_LOAD('URI', [options]);

options: {
  JSON_OBJECT('key','value'[,'key','value'] ...)
    'key','value': {
    ['formats', JSON_ARRAY('Format1'[,'Format2'] ...)]
    ['schema_name', 'SchemaName']
    ['table_name', 'TableName']
    ['region', 'Region']
    ['task_name', 'TaskName']
    ['language', 'Language']
    ['description', 'Description']
    ['uris', JSON_ARRAY(JSON_OBJECT('uri','URI1','table_name','TableName1')[,JSON_OBJECT('uri','URI2','tab
    ['autopilot_disable_check', {true|false}]
    }
}
```

Following are `VECTOR_STORE_LOAD` parameters:

- *`URI`*: specifies the unique reference index (URI) or pre-authenticated request (PAR) of the Object Storage bucket files or folders to be ingested into the vector store.

  A URI is considered to be one of the following:

  - A glob pattern, if it contains at least one unescaped `?` or `*` character.
  - A prefix, if it is not a pattern and ends with a `/` character like a folder path.
  - A file path, if it is neither a glob pattern nor a prefix.

  To learn how to create PAR for your object storage, see Creating a PAR request in Object Storage.

- `options`: specifies optional parameters as key-value pairs in JSON format. It can include the following parameters:

  - `formats`: specifies the list of formats to be loaded. The supported file formats are `pdf`, `ppt`, `txt`, `html`, and `doc`. By default, the routine uses all the supported formats.

    If the routine detects multiple files with the same or different file formats in a single load, it creates a separate table for every format it finds. The table name for each format is the specified or default table name followed by the format: *`TableName_Format`*.

  - `schema_name`: specifies the name of the schema where the vector embeddings are to be loaded. By default, this procedure uses the current schema from the session.

  - `table_name`: specifies the name of the vector store table to create. By default, the routine generates a unique table name with format `vector_store_data_x`, where `x` is a counter.

  - `region`: specifies the region of the bucket. Default value is the region where the current DB System is running.

  - `task_name`: specifies a name for the loading task to be mentioned in the task status. Default value is `Vector Store Loader`.

  - `language`: specifies the text content language used in the files to be ingested into the vector store. To set the value of the `language` parameter, use the two-letter `ISO 639-1` code for the language. This parameter is supported in HeatWave `9.0.1-u1` and later versions.

    Default value is `en`.

    For possible values, to view the list of supported languages, see Languages.

  - `description`: specifies a description of document collection being loaded to be mentioned in the task status. Default value is `NULL`.

  - `uris`: specifies a list of additional URIs to include along with an optional name for the vector store table to be created for the specified URI. Default value is `NULL`.

    This parameter accepts the following values:

    - `uri`: specifies the additional URI. If only `uri` is provided, the routine uses the specified URI as an additional URI, and loads it into the main table `options.table_name` or the generated table with the unique table name.

    - `table_name`: if only `table_name` is provided, the routine loads the specified vector store table into HeatWave.

      If both `uri` and `table_name` are provided, the routine loads the specified URI into the specified table.

  - `autopilot_disable_check`: if set to `true`, turns autopilot checks off, so HeatWave GenAI does not perform checks when the load task is running. Therefore, checks such as memory checks for estimating memory needed for tables and checking memory capacity in HeatWave for the load to complete are not performed. Default value is `false`.

Syntax Examples

- Specifying the file to ingest using the URI in `VECTOR_STORE_LOAD`:

```
call sys.VECTOR_STORE_LOAD('oci://demo_bucket@demo_namespace/demo_folder/demo_file.pdf', '{"table_name":
```

- Specifying the file to ingest using the PAR in VECTOR_STORE_LOAD:

```
call sys.VECTOR_STORE_LOAD('https://demo.objectstorage.us-ashburn-1.oci.customer-oci.com/p/demo-url/n/de
```

- Tracking the progress of a load task by running the task query displayed as output for the VECTOR_STORE_LOAD routine:

```
select id, name, message, progress, status, scheduled_time,estimated_completion_time, estimated_remainin
```

The output looks similar to the following:

```
           id: 1
         name: Vector Store Loader
      message: Task starting.
      progress: 0
        status: RUNNING
scheduled_time: 2024-07-02 14:42:38
estimated_completion_time: 2024-07-22 10:19:53
estimated_remaining_time: 52.50000
  progress_bar: __________
```

- Getting more details about the load task by querying the task logs for the given task id:

```
select * from mysql_task_management.task_log where task_id = 1;
```

The output looks similar to the following:

```
+------------------------------------+---------+----------------------------+-----------------------+
| id                                 | task_id | log_time                   | message               |
+------------------------------------+---------+----------------------------+-----------------------+
| 0x11EF799F5D99054288CC020017091C01 |       1 | 2024-09-23 11:31:24.884514 | Task created by user. |
| 0x11EF799F5D99812188CC020017091C01 |       1 | 2024-09-23 11:31:24.887685 | Task starting.        |
| 0x11EF799F6390FD9788CC020017091C01 |       1 | 2024-09-23 11:31:34.898219 | Loading in progress... |
| 0x11EF799F668C172F88CC020017091C01 |       1 | 2024-09-23 11:31:39.899271 | Loading in progress... |
| 0x11EF799F6987348588CC020017091C01 |       1 | 2024-09-23 11:31:44.900419 | Loading in progress... |
| 0x11EF799F6C82547D88CC020017091C01 |       1 | 2024-09-23 11:31:49.901634 | Loading in progress... |
+------------------------------------+---------+----------------------------+-----------------------+
```

4.7.4 ML_RAG

The ML_RAG routine performs retrieval-augmented generation (RAG) by:

1. Taking a natural-language query.
2. Retrieving context from relevant documents using semantic search.
3. Generating a response that integrates information from the retrieved documents.

This routine provides detailed, accurate, and contextually relevant answers by augmenting a generative model with information retrieved from a comprehensive knowledge base.

ML_RAG Syntax

```
mysql> call sys.ML_RAG('QueryInNaturalLanguage', 'Output', [options]);

options: {
  JSON_OBJECT('key','value'['key','value'] ...)
    'key','value': {
    ['vector_store', JSON_ARRAY('VectorStoreTableName1'['VectorStoreTableName2'] ...)]
    ['schema', JSON_ARRAY('Schema1'['Schema2'] ...)]
```

```
    ['n_citations', NumberOfCitations]
    ['distance_metric', {'COSINE'|'DOT'|'EUCLIDEAN'}]
    ['document_name', JSON_ARRAY('DocumentName1'['DocumentName2'] ...)]
    ['skip_generate', {true|false}]
    ['model_options', JSON_OBJECT('Key1','Value1'['Key2','Value2'] ...)]
    ['exclude_vector_store', JSON_ARRAY('ExcludeVectorStoreTableName1'['ExcludeVectorStoreTableName2'] ...)]
    ['exclude_document_name', JSON_ARRAY('ExcludeDocumentName1'['ExcludeDocumentName2'] ...)]
    }
}
```

Following are `ML_RAG` parameters:

- *`QueryInNaturalLangugae`*: specifies the natural-language query.
- *`Output`*: stores the generated output. The output contains the following segments:
  - `text`: the generated text-based response.
  - `citations`: contains the following details:
    - `segment`: the textual content that is retrieved from the vector store through semantic search, and used as context generating the response.
    - `distance`: the distance between the query embedding the segment embedding.
    - `document_name`: the name of the document from which the segment is retrieved.
  - `vector_store`: the list of vector store tables used for context retrieval.
- `options`: specifies optional parameters as key-value pairs in JSON format. It can include the following parameters:
  - `vector_store`: specifies a list of loaded vector store tables to use for context retrieval. The routine ignores invalid table names. By default, the routine performs a global search across all the available vector store tables in the DB system.
  - `schema`: specifies a list of schemas to check for loaded vector store tables. By default, the routine performs a global search across all the available vector store tables in all the schemas that are available in the DB system.
  - `n_citations`: specifies the number of segments to consider for context retrieval. Default value is `3`. Possible values are integer values between `0` and `100`.
  - `distance_metric`: specifies the distance metrics to use for context retrieval. Default value is `COSINE`. Possible values are `COSINE`, `DOT`, and `EUCLIDEAN`.
  - `document_name`: limits the documents to use for context retrieval. Only the specified documents are used. By default, the routine performs a global search across all the available documents stored in all the available vector stores in the DB system.
  - `skip_generate`: specifies whether to skip generation of the text-based response, and only perform context retrieval from the available or specified vector stores, schemas, or documents. Default value is `false`.
  - `model_options`: additional options that you can set for generating the text-based response. These are the same options that are available in the `ML_GENERATE` routine, which alter the text-based response per the specified settings. However, the `context` option is not supported as an `ML_RAG` model option. Default value is `'{"model_id": "mistral-7b-instruct-v1"}'`.

- `exclude_vector_store`: specifies a list of loaded vector store tables to exclude from context retrieval. The routine ignores invalid table names. Default value is `NULL`. This option is available in HeatWave `9.0.1-u1` and later versions.
- `exclude_document_name`: specifies a list of documents to exclude from context retrieval. Default value is `NULL`. This option is available in HeatWave `9.0.1-u1` and later versions.

Syntax Examples

- Retrieving context and generating output:

```
call sys.ML_RAG("What is AutoML",@output,@options);
```

Where, `@options` is set to specify the vector store table to use using `vector_store` key, as shown below:

```
set @options = JSON_OBJECT("vector_store", JSON_ARRAY("demo_db.demo_embeddings"));
```

Print the output:

```
select JSON_PRETTY(@output);
```

The output of the routine looks similar to the following:

```
{
"citations": [
{
"segment": "Oracle AutoML also produces high quality models very efficiently,
which is achieved through a scalable design and intelligent choices that
reduce trials at each stage in the pipeline.\n Scalable design: The Oracle
AutoML pipeline is able to exploit both HeatWave internode and intranode
parallelism, which improves scalability and reduces runtime.",
"distance": 0.4262576103210449,
"document_name": "https://objectstorage.Region.oraclecloud.com/n/Namespace/b/BucketName/o/Path/Filename"
},
{
"segment": "The HeatWave AutoML ML_TRAIN routine leverages Oracle AutoML
technology to automate the process of training a machine learning model.
Oracle AutoML replaces the laborious and time consuming tasks of the data
analyst whose workflow is as follows:\n1. Selecting a model from a large
number of viable candidate models.\n2.\n99",
"distance": 0.4311879277229309,
"document_name": " https://objectstorage. Region.oraclecloud.com/n/Namespace/b/BucketName/o/Path/Filenam
},
{
"segment": "3.1 HeatWave AutoML Features HeatWave AutoML makes it easy to
use machine learning, whether you are a novice user or an experienced ML
practitioner. You provide the data, and HeatWave AutoML analyzes the
characteristics of the data and creates an optimized machine learning model
that you can use to generate predictions and explanations.",
"distance": 0.4441382884979248,
"document_name": "https://objectstorage. Region.oraclecloud.com/n/Namespace/b/BucketName/o/Path/Filename
}
],
"vector_store": [
"demo_db.demo_embeddings"
]
}
```

4.7.5 ML_RAG_TABLE

The ML_RAG_TABLE routine runs multiple retrieval-augmented generation (RAG) queries in a batch, in parallel. The output generated for every input query is the same as the output generated by the ML_RAG routine.

This routine is available in HeatWave 9.0.1-u1 and later versions.

ML_RAG_TABLE Syntax

```
mysql> call sys.ML_RAG_TABLE('InputTableColumn', 'OutputTableColumn', [options]);

options: {
  JSON_OBJECT('key','value'[,'key','value'] ...)
    'key','value': {
    ['vector_store', JSON_ARRAY('VectorStoreTableName1'[,'VectorStoreTableName2'] ...)]
    ['schema', JSON_ARRAY('Schema1'[,'Schema2'] ...)]
    ['n_citations', NumberOfCitations]
    ['distance_metric', {'COSINE'|'DOT'|'EUCLIDEAN'}]
    ['document_name', JSON_ARRAY('DocumentName1'[,'DocumentName2'] ...)]
    ['skip_generate', {true|false}]
    ['model_options', JSON_OBJECT('Key1','Value1'[,'Key2','Value2'] ...)]
    ['exclude_vector_store', JSON_ARRAY('ExcludeVectorStoreTableName1'[,'ExcludeVectorStoreTableName2'] ...)]
    ['exclude_document_name', JSON_ARRAY('ExcludeDocumentName1'[,'ExcludeDocumentName2'] ...)]
    ['batch_size', BatchSize]
    }
}
```

Following are ML_RAG_TABLE parameters:

- *InputTableColumn*: specifies the names of the input database, table, and column that contains the natural-language queries. The *InputTableColumn* is specified in the following format: *DBName*.*TableName*.*ColumnName*.
  - The specified input table can be an internal or external table.
  - The specified input table must already exist, must not be empty, and must have a primary key.
  - The input column must already exist and must contain text or varchar values.
  - The input column must not be a part of the primary key and must not have NULL values or empty strings.
  - There must be no backticks used in the *DBName*, *TableName*, or *ColumnName* and there must be no period used in the *DBName* or *TableName*.
- *OutputTableColumn*: specifies the names of the database, table, and column where the generated text-based response is stored. The *OutputTableColumn* is specified in the following format: *DBName*.*TableName*.*ColumnName*.
  - The specified output table must be an internal table.
  - If the specified output table already exists, then it must be the same as the input table. And, the specified output column must not already exist in the input table. A new JSON column is added to the table. External tables are read only. So if input table is an external table, then it cannot be used to store the output.
  - If the specified output table doesn't exist, then a new table is created. The new output table has key columns which contains the same primary key values as the input table and a JSON column that stores the generated text-based responses.

  - There must be no backticks used in the *`DBName`*, *`TableName`*, or *`ColumnName`* and there must be no period used in the *`DBName`* or *`TableName`*.

- `options`: specifies optional parameters as key-value pairs in JSON format. It can include the following parameters:

  - `vector_store`: specifies a list of loaded vector store tables to use for context retrieval. The routine ignores invalid table names. By default, the routine performs a global search across all the available vector store tables in the DB system.

  - `schema`: specifies a list of schemas to check for loaded vector store tables. By default, the routine performs a global search across all the available vector store tables in all the schemas that are available in the DB system.

  - `n_citations`: specifies the number of segments to consider for context retrieval. Default value is `3`. Possible values are integer values between `0` and `100`.

  - `distance_metric`: specifies the distance metrics to use for context retrieval. Default value is `COSINE`. Possible values are `COSINE`, `DOT`, and `EUCLIDEAN`.

  - `document_name`: limits the documents to use for context retrieval. Only the specified documents are used. By default, the routine performs a global search across all the available documents stored in all the available vector stores in the DB system.

  - `skip_generate`: specifies whether to skip generation of the text-based response, and only perform context retrieval from the available or specified vector stores, schemas, or documents. Default value is `false`.

  - `model_options`: additional options that you can set for generating the text-based response. These are the same options that are available in the `ML_GENERATE` routine, which alter the text-based response per the specified settings. However, the `context` option is not supported as an `ML_RAG_TABLE` model option. Default value is `'{"model_id": "mistral-7b-instruct-v1"}'`.

  - `exclude_vector_store`: specifies a list of loaded vector store tables to exclude from context retrieval. The routine ignores invalid table names. Default value is `NULL`.

  - `exclude_document_name`: specifies a list of documents to exclude from context retrieval. Default value is `NULL`.

  - `batch_size`: specifies the batch size for the routine. This parameter is supported for internal tables only. Default value is `1000`. Possible values are integer values between `1` and `1000`.

Syntax Examples

Running retrieval-augmented generation in a batch of 10:

```
mysql> call sys.ML_RAG_TABLE("demo_db.input_table.Input", "demo_db.output_table.Output", JSON_OBJECT("vecto
```

In this example, the routine performs RAG for 10 input queries stored in the `demo_db.input_table.Input` column, and creates a column of 10 rows `demo_db.output_table.Output` where it stores the generated outputs.

4.7.6 HEATWAVE_CHAT

The `HEATWAVE_CHAT` routine automatically calls the `ML_RAG` routine which loads an LLM and runs a semantic search on the available vector stores by default. If the routine cannot find a vector store, then

it calls the ML_GENERATE routine and uses information available in LLM training data, which is primarily information that is available in public data sources, to generate a response for the entered query.

HEATWAVE_CHAT Syntax

```
mysql> call sys.HEATWAVE_CHAT('QueryInNaturalLanguage');
```

The HEATWAVE_CHAT routine accepts one input parameter:

- *QueryInNaturalLanguage*: specifies the query in natural language.

For specifying additional chat options, the HEATWAVE_CHAT routine reserves a session variable, @chat_options. When you run the routine, it also updates the session variable @chat_options with any additional information that is used or collected by the routine to generate the response.

@chat_options Parameters

Following is a list of all the parameters that you can set in the @chat_options session variable:

- **Input only**: you can set these parameters to control the chat behaviour. The routine cannot change the values of these parameters.
  - schema_name: specifies the name of a schema. If set, the routine searches for vector store tables in this schema. This parameter cannot be used in combination with the tables parameter. Default value is NULL
  - report_progress: specifies whether information such as routine progress detail is to be reported. Default value is false.
  - skip_generate: specifies whether response generation is skipped. If set to true, the routine does not generate a response. Default value is false.
  - return_prompt: specifies whether to return the prompt that was passed to the ML_RAG or ML_GENERATE routines. Default value is false.
  - re_run: if set to true, it indicates that the request is a re-run of the previous request. For example, a re-run of a query with some different parameters. The new query and response replaces the last entry stored in the chat_history parameter. Default value is false.
  - include_document_uris: limits the documents used for context retrieval by including only the specified document URIs. Default value is NULL.
  - retrieve_top_k: specifies the context size. The default value is the value of the n_citations parameter of the ML_RAG routine. Possible values are integer values between 0 and 100.
  - chat_query_id: specifies the chat query ID to be printed with the chat_history in the GUI. This parameter is reserved for GUI use. By default, the routine generates random IDs.
  - history_length: specifies the maximum history length, which is the number of question and answers, to include in the chat history. The specified value must be greater than or equal to 0. Default value is 3.
- **Input-output**: both you and the routine can change the values of these parameters.
  - chat_history: JSON array that represents the current chat history. Default value is NULL.

    Syntax for each object in the chat_history array is as follows:

    ```
    JSON_OBJECT('key','value'[,'key','value'] ...)
    ```

```
  'key','value': {
  ['user_message','Message']
  ['chat_bot_message','Message']
  ['chat_query_id','ID']
}
```

Each parameter value in the array holds the following keys and their values:

- `user_message`: message entered by the user.
- `chat_bot_message`: message generated by the chat bot.
- `chat_query_id`: a query ID.

- `tables`: JSON array that represents the following:
  - For providing input, represents the list of vector store schema or table names to consider for context retrieval.
  - As routine output, represents the list of discovered vector store tables loaded to HeatWave, if any. Otherwise, it holds the same values as input.

  Default value is `NULL`.

  Syntax for each object in the `tables` array is as follows:

  ```
  JSON_OBJECT('key','value'[,'key','value'] ...)
    'key','value': {
    ['schema_name','SchemaName']
    ['table_name','TableName']
  }
  ```

  Each parameter values in the array holds the following keys and their values:

  - `schema_name`: name of the schema.
  - `table_name`: name of the vector store table.

- `task`: specifies the task performed by the LLM. Default value is `generation`. Possible value is `generation`.
- `model_options`: optional model parameters specified as key-value pairs in JSON format. These are the same options that are available in the `ML_GENERATE` routine, which alter the text-based response per the specified settings. Default value is `'{"model_id": "mistral-7b-instruct-v1"}'`.

- **Output only**: only the routine can set or change values of these parameters.
  - `info`: contains information messages such as routine progress information. Default value is `NULL`. This parameter is populated only if `report_progress` is set to `true`.
  - `error`: contains the error message if an error occurred. Default value is `NULL`.
  - `error_code`: contains the error code if an error occurred. Default value is `NULL`.
  - `prompt`: contains the prompt passed to the `ML_RAG` or `ML_GENERATE` routine. Default value is `NULL`. This parameter is populated only if `report_prompt` is set to `true`.
  - `documents`: contains the names of the documents as well as segments used as context by the LLM for response generation. Default value is `NULL`.
  - `request_completed`: set to `true` when a response is the last response message to a request. Default value is `NULL`.
  - `response`: contains the final response from the routine. Default value is `NULL`.

Syntax Examples

- Entering a natural-language query using the `HEATWAVE_CHAT` routine:

```
call sys.HEATWAVE_CHAT("What is Lakehouse?");
```

- Modifying chat parameters using the `@chat_options` session variable:
  - Modifying a chat parameter, `tables`, to specify the vector store table to use for context retrieval in the next chat session:

    ```
    set @chat_options = '{"tables": [{"table_name": "demo_embeddings", "schema_name": "demo_db"}]}';
    ```

    This example resets the chat session and uses the specified vector store table in the new chat session.
  - Modifying a chat parameter, `tables`, to specify the vector store table to use for context retrieval in the same chat session:

    ```
    set @chat_options = JSON_SET(@chat_options,'$.tables', JSON_ARRAY(JSON_OBJECT("table_name", "demo_embeddin
    ```

    This example uses the specified vector store table in the ongoing chat session. It does not reset the chat session.
  - Modifying a chat parameter, `temperature`, without resetting the chat session:

    ```
    set @chat_options = json_set(@chat_options, '$.model_options.temperature', 0.5);
    ```

- Viewing the chat parameters and session details:

```
select JSON_PRETTY(@chat_options);
```

For more information about the output generated by this command, see Section 4.5.2, “Viewing Chat Session Details”.

4.7.7 ML_EMBED_ROW

The `ML_EMBED_ROW` routine uses the specified embedding model to encode the specified text or query into a vector embedding. The routine returns a `VECTOR` that contains a numerical representation of the specified text.

ML_EMBED_ROW Syntax

```
mysql> select sys.ML_EMBED_ROW('Text', [options]);

options: {
  JSON_OBJECT('key','value'['key','value'] ...)
    'key','value': {
    ['model_id', {'all_minilm_l12_v2'|'multilingual-e5-small'|'cohere.embed-english-v3.0'|'cohere.embed-mu
    ['truncate', {true|false}]
    }
}
```

Following are `ML_EMBED_ROW` parameters:

- *`Text`*: specifies the text to encode.
- `options`: specifies optional parameters as key-value pairs in JSON format. It can include the following parameters:
  - `model_id`: specifies the embedding model to use for encoding the text. Default value is `all_minilm_l12_v2`. Possible values are:
    - `all_minilm_l12_v2`: for encoding English text.
    - `multilingual-e5-small`: for encoding text in supported languages other than English (`en`). This embedding model is available in HeatWave `9.0.1-u1` and later versions.
    - `cohere.embed-english-v3.0`: for encoding English text. This embedding model is available in HeatWave `9.0.1-u1` and later versions.
    - `cohere.embed-multilingual-v3.0`: for encoding text in supported languages other than English. This embedding model is available in HeatWave `9.0.1-u1` and later versions.

    To view the lists of supported models, see HeatWave In-Database Embedding Models and OCI Generative AI Service Embedding Models. To view the list of supported languages, see Languages.
  - `truncate`: specifies whether to truncate inputs longer than the maximum token size. Default value is `true`.

Syntax Examples

- Embedding an English query using the `all_minilm_l12_v2` embedding model, and store the generated embedding in the `@text_embedding` session variable:

```
mysql> select sys.ML_EMBED_ROW("What is artificial intelligence?", JSON_OBJECT("model_id", "all_minilm_l
```

4.7.8 ML_EMBED_TABLE

The `ML_EMBED_TABLE` routine runs multiple embedding generations in a batch, in parallel.

This routine is available in HeatWave `9.0.1-u1` and later versions.

ML_EMBED_TABLE Syntax

```
mysql> call sys.ML_EMBED_TABLE('InputTableColumn', 'OutputTableColumn', [options]);

options: {
  JSON_OBJECT('key','value'[,'key','value'] ...)
    'key','value': {
    ['model_id', {'all_minilm_l12_v2'|'multilingual-e5-small'|'cohere.embed-english-v3.0'|'cohere.embed-multil
    ['truncate', {true|false}]
    ['batch_size', BatchSize]
    }
}
```

Following are `ML_EMBED_TABLE` parameters:

- *`InputTableColumn`*: specifies the names of the input database, table, and column that contains the text to encode. The *`InputTableColumn`* is specified in the following format: *`DBName`*.*`TableName`*.*`ColumnName`*.
  - The specified input table can be an internal or external table.
  - The specified input table must already exist, must not be empty, and must have a primary key.
  - The input column must already exist and must contain `text` or `varchar` values.
  - The input column must not be a part of the primary key and must not have `NULL` values or empty strings.
  - There must be no backticks used in the *`DBName`*, *`TableName`*, or *`ColumnName`* and there must be no period used in the *`DBName`* or *`TableName`*.
- *`OutputTableColumn`*: specifies the names of the database, table, and column where the generated embeddings are stored. The *`OutputTableColumn`* is specified in the following format: *`DBName`*.*`TableName`*.*`ColumnName`*.
  - The specified output table must be an internal table.
  - If the specified output table already exists, then it must be the same as the input table. And, the specified output column must not already exist in the input table. A new `VECTOR` column is added to the table. External tables are read only. So if input table is an external table, then it cannot be used to store the output.
  - If the specified output table doesn't exist, then a new table is created. The new output table has key columns which contains the same primary key values as the input table and a `VECTOR` column that stores the generated embeddings.
  - There must be no backticks used in the *`DBName`*, *`TableName`*, or *`ColumnName`* and there must be no period used in the *`DBName`* or *`TableName`*.
- `options`: specifies optional parameters as key-value pairs in JSON format. It can include the following parameters:
  - `model_id`: specifies the embedding model to use for encoding the text. Default value is `all_minilm_l12_v2`. Possible values are:
    - `all_minilm_l12_v2`: for encoding English text.
    - `multilingual-e5-small`: for encoding text in supported languages other than English.

- `cohere.embed-english-v3.0`: for encoding English text.
- `cohere.embed-multilingual-v3.0`: for encoding text in supported languages other than English.

To view the lists of supported models, see HeatWave In-Database Embedding Models and OCI Generative AI Service Embedding Models. To view the list of supported languages, see Languages.

- `truncate`: specifies whether to truncate inputs longer than the maximum token size. Default value is `true`.
- `batch_size`: specifies the batch size for the routine. This parameter is supported for internal tables only. Default value is `1000`. Possible values are integer values between `1` and `1000`.

Syntax Examples

Generating embeddings for text stored in `demo_db.input_table.Input` and saving the generating embeddings in `demo_db.output_table.Output` using the `all_minilm_l12_v2` embedding model:

```
mysql> call sys.ML_EMBED_TABLE("demo_db.input_table.Input", "demo_db.output_table.Output", JSON_OBJECT("mo
```

4.8 Troubleshooting Issues and Errors

This section describes some commonly encountered issues and errors for HeatWave GenAI and their workarounds.

- *Issue:* When you try to verify whether the vector embeddings were correctly loaded, if you see a message which indicates that the vector embeddings or table did not load in HeatWave, then it could be due one of the following reasons:
  - The task that loads the vector embeddings into the vector store table might still be running.

    Workaround: Check the task status by using the query that was printed by the `VECTOR_STORE_LOAD` routine:

    ```
    SELECT * from mysql_task_management.task_status where id = TaskID;
    ```

    Or, to see the log messages, check the task logs table:

    ```
    SELECT * from mysql_task_management.task_log where task_id = TaskID;
    ```

    Replace *`TaskID`* with the ID for the task which was printed by the `VECTOR_STORE_LOAD` routine.
  - The folder you are trying to load might contain unsupported format files or the file that you are trying to load might be of an unsupported format.

    Workaround: The supported file formats are: PDF, TXT, PPT, HTML, and DOC.

    If you find unsupported format files, then try one of the following:

    - Delete the files with unsupported formats from the folder, and run the `VECTOR_STORE_LOAD` command again to load the vector embeddings into the vector store table again.
    - Move the files with supported formats to another folder, create a new PAR and run the `VECTOR_STORE_LOAD` command with the new PAR to load the vector embeddings into the vector store table again.

- *Issue:* the `VECTOR_STORE_LOAD` command fails unexpectedly.

  Workaround: Ensure that you use the `--sqlc` flag when you connect to your database system:

  ```
  mysqlsh -uAdmin -pPassword -hPrivateIP --sqlc
  ```

  Replace the following:

  - *`Admin`*: the admin name.
  - *`Password`*: the database system password.
  - *`PrivateIP`*: the private IP address of the database system.

  If you are still not able to ingest files using `VECTOR_STORE_LOAD` routine, then try Ingesting Files Using Auto Parallel Load.

Chapter 5 HeatWave Lakehouse

Table of Contents

This chapter describes HeatWave Lakehouse.

5.1 Overview

The Lakehouse feature of HeatWave enables query processing on data resident in Object Storage. The source data is read from Object Storage, transformed to the memory optimized HeatWave format, stored in the HeatWave persistence storage layer in Object Storage, and then loaded to HeatWave cluster memory.

- Provides in-memory query processing on data resident in Object Storage.
- Data is not loaded into the MySQL InnoDB storage layer.
- Supports structured and semi-structured relational data in the following file formats:
  - Avro.
  - CSV.
  - JSON.

    As of MySQL 8.4.0, Lakehouse supports Newline Delimited JSON.
  - Parquet.

See: Section 5.6.1, “Parquet Data Type Conversions”.

- With this feature, users can now analyse data in both InnoDB and an object store using familiar SQL syntax in the same query.

To use Lakehouse with HeatWave AutoML, see: Section 3.12, “HeatWave AutoML and Lakehouse”.

5.1.1 External Tables

External tables are non-InnoDB tables which do not store any data, but refer to data stored externally in Object Storage, in the following file formats:

- Avro.
- CSV.
- JSON.

  As of MySQL 8.4.0, Lakehouse supports Newline Delimited JSON.

- Parquet.

  See: Section 5.6.1, “Parquet Data Type Conversions”.

The external table stores the location of the data, see Section 5.4, “Access Object Storage”.

5.1.2 Lakehouse Engine

HeatWave Lakehouse introduces the `lakehouse` storage engine.

The Lakehouse Engine enables you to create tables which point to external data sources.

For HeatWave Lakehouse, `lakehouse` is the primary engine, and `rapid` is the secondary engine.

5.1.3 Data Storage

The source data is read from Object Storage, transformed to the memory optimized HeatWave format, stored in the HeatWave persistence storage layer in Object Storage, and then loaded to HeatWave cluster memory. This data is not directly accessed by end-users of the service. This memory optimized internal version of the data is retained only as long as the HeatWave Cluster has the Lakehouse option enabled.

The data is deleted if the external table is dropped or unloaded, or if the HeatWave Cluster is deleted.

5.2 Loading Structured Data to HeatWave Lakehouse

Lakehouse offers the same alternatives to load data as HeatWave, see: Section 2.2, “Loading Data to HeatWave MySQL”.

5.2.1 Prerequisites

Lakehouse requires the following:

- A HeatWave enabled MySQL DB System with Lakehouse support enabled, and a minimum 512GB shape. See: Adding a HeatWave Cluster in the *HeatWave on OCI Service Guide*.

For a replicated MySQL DB System, see Section 5.8, “HeatWave Lakehouse Limitations”.

- Access to Object Storage, see: Section 5.4, “Access Object Storage”.

5.2.2 Lakehouse External Table Syntax

Lakehouse uses external tables to load the data from Object Storage.

MySQL 9.0.1-u1 adds support for the following:

- `timestampe_format` as a `dialect` parameter allows you to customize the format for columns of the `DATETIME` and `TIMESTAMP` data types.
- `ENGINE_ATTRIBUTE` option for specific columns. Use this option to customize the format for columns of the `DATETIME` and `TIMESTAMP` data types. Formats set with this parameter override formats set with the `dialect` parameter.

```
mysql> CREATE TABLE table_name create_definition
(column_name ENGINE_ATTRIBUTE '
  {"date_format": "default",
   "time_format": "default",
   "timestamp_format": "default"
  }')
ENGINE=lakehouse SECONDARY_ENGINE=rapid ENGINE_ATTRIBUTE='{
  "dialect":
  {
    "format": "avro" | "csv" | "json" | "parquet",
    "check_constraints": true | false,
    "field_delimiter": "|",
    "record_delimiter": "|\n",
    "escape_character": "\\",
    "quotation_marks":"\"",
    "skip_rows": 0,
    "encoding": "utf8mb4",
    "date_format": "default",
    "time_format": "default",
    "timestamp_format": "default",
    "trim_spaces": true | false,
    "has_header": true | false,
    "is_strict_mode": true | false,
    "allow_missing_files": true | false
  },
  "file":
  [
    file_section [, file_section, file_section, ...]
  ]
}';

file_section: {
  "bucket": "bucket_name",
  "namespace": "namespace",
  "region": "region",
  ("prefix": "prefix") | ("name": "filename")| ("pattern" : "pattern"),
  "is_strict_mode": true | false,
  "allow_missing_files": true | false
}

or

file_section: {
  "par": "PAR URL",
  ("prefix": "prefix") | ("name": "filename")| ("pattern" : "pattern"),
  "is_strict_mode": true | false,
```

```
  "allow_missing_files": true | false
}
```

MySQL 8.4.0 adds support for the following:

- The JSON file format.
- Support for primary key and unique key constraint validation.
- `is_strict_mode` as a `dialect` parameter now supports all file formats. It can override the global `sql_mode`.
- `allow_missing_files` is a `dialect` parameter and a `file` parameter that can allow missing files or not.

```
mysql> CREATE TABLE table_name create_definition ENGINE=lakehouse SECONDARY_ENGINE=rapid
ENGINE_ATTRIBUTE='{
  "dialect":
  {
    "format": "avro" | "csv" | "json" | "parquet",
    "check_constraints": true | false,
    "field_delimiter": "|",
    "record_delimiter": "|\n",
    "escape_character": "\\",
    "quotation_marks":"\"",
    "skip_rows": 0,
    "encoding": "utf8mb4",
    "date_format": "auto",
    "time_format": "auto",
    "trim_spaces": true | false,
    "has_header": true | false,
    "is_strict_mode": true | false,
    "allow_missing_files": true | false
  },
  "file":
  [
    file_section [, file_section, file_section, ...]
  ]
}';

file_section: {
  "bucket": "bucket_name",
  "namespace": "namespace",
  "region": "region",
  ("prefix": "prefix") | ("name": "filename")| ("pattern" : "pattern"),
  "is_strict_mode": true | false,
  "allow_missing_files": true | false
}

or

file_section: {
  "par": "PAR URL",
  ("prefix": "prefix") | ("name": "filename")| ("pattern" : "pattern"),
  "is_strict_mode": true | false,
  "allow_missing_files": true | false
}
```

Before MySQL 8.4.0:

```
mysql> CREATE TABLE table_name create_definition ENGINE=lakehouse SECONDARY_ENGINE=rapid
ENGINE_ATTRIBUTE='{
  "dialect":
  {
    "format": "avro" | "csv" | "parquet",
```

```
    "field_delimiter": "|",
    "record_delimiter": "|\n",
    "escape_character": "\\",
    "quotation_marks":"\"",
    "skip_rows": 0,
    "encoding": "utf8mb4",
    "date_format": "auto",
    "time_format": "auto",
    "trim_spaces": true | false,
    "is_strict_mode": true | false,
    "has_header": true | false
  },
  "file":
  [
    file_section [, file_section, file_section, ...]
  ]
}';

file_section: {
  "bucket": "bucket_name",
  "namespace": "namespace",
  "region": "region",
  ("prefix": "prefix") | ("name": "filename") | ("pattern" : "pattern"),
  "is_strict_mode": true | false
}

or

file_section: {
  "par": "PAR URL",
  ("prefix": "prefix") | ("name": "filename") | ("pattern" : "pattern"),
  "is_strict_mode": true | false
}
```

- `ENGINE`: Set to `lakehouse`.
- `SECONDARY_ENGINE`: Set to `rapid`.
- `ENGINE_ATTRIBUTE`: JSON object literal. Defines the location of files, the file format, and how the file format is handled.

  Use key-value pairs in `JSON` format to specify *`options`*. Lakehouse uses the default setting if there is no defined option. Use `NULL` to specify no arguments.

  - `dialect`: Defines the data configuration. Optional.
    - `dialect` parameters that apply to all file formats:
      - `format`: `avro`, `csv`, `json` or `parquet`. The default is `csv`.

        It is not possible to define multiple formats per table.

        Tables created with `json` format must only have a single column that conforms to the `JSON` data type, see: The JSON Data Type.

        As of MySQL 8.4.0, `json` only supports Newline Delimited JSON files.

        For `parquet` see: Section 5.6.1, “Parquet Data Type Conversions”.

      - `check_constraints`: Whether to validate primary key and unique key constraints or not. The default is `true`. Supported as of MySQL 8.4.0.

        If set to `true`, then Lakehouse validates primary key and unique key constraints.

If set to `false`, then Lakehouse does not validate primary key and unique key constraints.

- `is_strict_mode`: Whether the loading takes place in strict mode, `true`, or non-strict mode, `false`. This setting overrides the global `sql_mode`. The default is the value of `sql_mode`. See Strict SQL Mode. The `file` common parameter `is_strict_mode` can override this setting.

  If set to `true`, then missing files, empty columns, formatting errors or parsing errors throw an error, and loading stops.

  If set to `false`, then missing files, empty columns, formatting errors or parsing errors display a warning, and loading continues.

  As of MySQL 8.4.0, the `dialect` parameter `is_strict_mode` applies to all file formats. Before MySQL 8.4.0, it only applies to the CSV file format. For Avro and Parquet file formats, use the `file` parameter `is_strict_mode` to define strict mode before MySQL 8.4.0.

- `allow_missing_files`: Whether to allow missing files or not. This overrides the `dialect` parameter `is_strict_mode` for missing files. Supported as of MySQL 8.4.0.

  If set to `true`, then any missing files do not throw an error, and loading does not stop, unless all files are missing.

  If set to `false`, then any missing files throw an error, and loading stops.

  A missing file is defined as:

  - With the `name` parameter: there is no file with that name.
  - With the `pattern` parameter: there are no files that match the pattern.
  - With the `prefix` parameter: there are no files with that prefix.

- `dialect` parameters that only apply to `csv` and `json`:

  The use of any of these parameters with `avro` or `parquet` will produce an error.

  - `encoding`: Defines the character encoding. The default is `"utf8mb4"`.
  - `record_delimiter`: Defines one or more characters used to delimit records. The maximum field delimiter length is 64 characters.

    The default for `csv` is `"|\n"`.

    The default for `json` is `"\n"`. The only alternative for `json` is `"\r\n"`.

- `dialect` parameters that only apply to `csv`:

  The use of any of these parameters with `avro`, `json` or `parquet` will produce an error.

  - `field_delimiter`: Defines one or more characters used to enclose fields. The maximum field delimiter length is 64 characters. The default is `"|"`.
  - `escape_character`: Defines one or more characters used to escape special characters. The default is `"\\"`.
  - `quotation_marks`: Defines one or more characters used to enclose fields. The default is `"\""`.

- `skip_rows`: The number of rows to skip at the start of the file. The maximum value is `20`. The default is `0`.

  See comments for `has_header`.

- `date_format`: The date format, see: `date_format`. This format is ignored during auto parallel load. See Section 5.2.4, “Loading Data Using Auto Parallel Load”.

- `time_format`: The time format, see: String and Numeric Literals in Date and Time Context. This format is ignored during auto parallel load. See Section 5.2.4, “Loading Data Using Auto Parallel Load”.

- `timestampe_format`: The timestamp format, see: `date_format`. Optionally, you can set timestamp formats for each column by using the column `ENGINE_ATTRIBUTE` option, which

overrides the format in the `dialect` parameter. This format is ignored during auto parallel load. See Section 5.2.4, “Loading Data Using Auto Parallel Load”.

- `trim_spaces`: Whether to remove leading and trailing spaces, or not. The default is `false`.
- `has_header`: Whether the CSV file has a header row, or not. The default is `false`.

  If `has_header` and `skip_rows` are both defined, Lakehouse first skips the number of rows, and then uses the next row as the header row.

- `file`: Defines the Object Storage files. Required.

  Lakehouse supports a maximum of 256 file locations. To define more than 256, store the files under the same bucket or use `prefix` or `pattern`.

  - `file` parameters for resource principals, see: Section 5.4.2, “Resource Principals”.
    - `bucket`: The bucket name.
    - `namespace`: The tenancy namespace
    - `region`: The region that the tenancy is in.
  - `file` parameters for pre-authenticated requests, see: Section 5.4.1, “Pre-Authenticated Requests”.
    - `par`: The PAR URL.

      Do not specify a `region`, `namespace` or `bucket` with `par`. That information is contained in the PAR URL and will generate an error if defined in separate parameters. See: Section 5.4.1.1, “Recommendations”.
  - `file` parameters that apply to all file formats:
    - Use one or more of the following parameters, unless the target defines a specific file:
      - `name`: A specific Object Storage file name.
      - `pattern`: A regular expression that defines a set of Object Storage files. The pattern follows the modified ECMAScript regular expression grammar, see: Modified ECMAScript regular expression grammar.
      - `prefix`: The prefix for a set of Object Storage files.

      See: Section 5.2.2.1, “File Name, Pattern and Prefix Examples”.
    - `is_strict_mode`: Whether the loading takes place in strict mode, `true`, or non-strict mode, `false`. This overrides the `dialect` parameter `is_strict_mode`.
    - `allow_missing_files`: Whether to allow missing files or not. This overrides the `file` parameter `is_strict_mode` for missing files, and the `dialect` parameter `allow_missing_files`. Supported as of MySQL 8.4.0.

      If set to `true`, then any missing files do not throw an error, and loading does not stop.

      If set to `false`, then any missing files throw an error, and loading stops.

5.2.2.1 File Name, Pattern and Prefix Examples

This is a series of examples that demonstrate the use of `name`, `pattern` and `prefix`. For simplicity, this ignores the file location details.

A directory with the name `input` contains 204 files:

- 100 files with names in a sequence from `file00.tbl` to `file99.tbl`.
- 104 files with names in a sequence from `fileaa.tbl` to `fileaz.tbl`, `fileba.tbl` to `filebz.tbl`, `fileca.tbl` to `filecz.tbl`, and `fileda.tbl` to `filedz.tbl`.

Use a `prefix` to load all 204 files from the `input` directory:

```
"file": [{"prefix": "input/"}]
```

Use one or more `name` to load individual files that do not fit into a convenient sequence:

```
"file": [{"name": "input/file25.tbl"}, {"name": "input/fileck.tbl"}]
```

Use a regular expression `pattern` to load specific file name sequences. The regular expression syntax states that certain characters require an escape character, see: Regular Expression Syntax.

The escape character is the backslash character, and it is a reserved character in both JSON and MySQL. Therefore, it is necessary to escape the backslash character twice, and specify `\\` for JSON, and `\\` for MySQL.

However, the regular expression escape sequence depends upon the `NO_BACKSLASH_ESCAPES` SQL mode:

- Use `\\.` to escape a period if `NO_BACKSLASH_ESCAPES` is enabled.
- Use `\\\\.` to escape a period if `NO_BACKSLASH_ESCAPES` is not enabled. The following examples use this sequence because it is the default mode.

To load all 100 files with a numeric suffix:

```
"file": [{"pattern" : "input/file\\\\d+\\\\.tbl"}]
```

To load all 104 files with an alphabetical suffix:

```
"file": [{"pattern" : "input/file[a-z]+\\\\.tbl"}]
```

To load 10 files, `file00.tbl`, `file10.tbl` ... `file90.tbl`:

```
"file": [{"pattern" : "input/file\\\\d0\\\\.tbl"}]
```

To load 24 files, `fileaa.tbl` to `fileaf.tbl`, `fileba.tbl` to `filebf.tbl`, `fileca.tbl` to `filecf.tbl`, and `fileda.tbl` to `filedf.tbl`,

```
"file": [{"pattern" : "input/file[a-d][a-f]\\\\.tbl"}]
```

5.2.3 Loading Data Manually

As of MySQL 8.4.0, Lakehouse supports primary key and unique key constraint validation. Multiple primary key and unique key constraints can increase load time. Set `check_constraints` to `false` to avoid primary key and unique key constraint validation. See: Section 5.2.2, “Lakehouse External Table Syntax”.

To load data manually, follow these steps:

1. Choose whether to use a pre-authenticated request or a resource principal, see: Section 5.2.3.1, “Manually Loading Data from External Storage”.
2. Use the Lakehouse `ENGINE_ATTRIBUTE` with `CREATE TABLE` statements, see: Section 5.2.2, “Lakehouse External Table Syntax”.
3. Review these steps: Section 2.2.2, “Loading Data Manually”.

Lakehouse extends Guided Load for external tables, see: Section 2.2.2, “Loading Data Manually”. This employs Autopilot to perform a series of pre-load validation checks, and includes the following:

- Detects any errors with `ENGINE_ATTRIBUTE` and reports them.
- Infers the schema and performs similar schema adjustments to those performed by Autopilot during Lakehouse Auto Parallel Load. If the inferred schema is not compatible with the defined schema, then Guided Load aborts the load. See: Section 5.2.4.1, “Lakehouse Auto Parallel Load Schema Inference”.
- Predicts the amount of memory required, and checks that this is available. If the required memory is not available, Guided Load aborts the load.

To monitor any issues encountered during this pre-load validation process, run the `SHOW WARNINGS` statement after the load command has finished.

5.2.3.1 Manually Loading Data from External Storage

Manually Loading Data with Pre-Authenticated Requests

> **Note**
>
> PARs can be used for any Object Storage data stored in any tenancy in the same region.

```
mysql> CREATE TABLE `CUSTOMER` (`C_CUSTKEY` int NOT NULL PRIMARY KEY, `C_NATIONKEY` int NOT NULL)
ENGINE=lakehouse
SECONDARY_ENGINE = RAPID
ENGINE_ATTRIBUTE='{"dialect": {"format": "csv”},
                   "file": [{"par": "https://objectstorage.../n/some_bucket/customer.tbl"}]}';
ALTER TABLE `CUSTOMER` SECONDARY_LOAD;
```

Manually Loading Data with Resource Principals

```
mysql> CREATE TABLE `CUSTOMER` (`C_CUSTKEY` int NOT NULL PRIMARY KEY,`C_NATIONKEY` int NOT NULL)
ENGINE=lakehouse
SECONDARY_ENGINE = RAPID
ENGINE_ATTRIBUTE='{"dialect": {"format": "csv”},
                   "file": [{"region": "regionName", "namespace": tenancyNamespace",
                   "bucket": "bucketName", "name": "customer.tbl"}]}';
ALTER TABLE `CUSTOMER` SECONDARY_LOAD;
```

5.2.4 Loading Data Using Auto Parallel Load

HeatWave Lakehouse extends Auto Parallel Load in two ways:

- Lakehouse Auto Parallel Load includes schema inference which analyzes the external data to infer the table structure.

- Lakehouse Auto Parallel Load uses the `external_tables` option to enable loading data from external sources. See: Section 5.2.4.2, “Lakehouse Auto Parallel Load with the external_tables Option”. Do not use as of MySQL 8.4.0. `external_tables` will be deprecated in a future release.
- As of MySQL 8.4.0, Lakehouse Auto Parallel Load uses the `db_object` with `table` or `exclude_tables` instead. See: Section 2.2.3.2, “Auto Parallel Load Syntax”.

Lakehouse Auto Parallel Load facilitates the process of loading data into HeatWave by automating many of the steps involved, including:

- All these steps: Section 2.2.3, “Loading Data Using Auto Parallel Load”.
- Defining `lakehouse` as the engine for tables that are to be loaded.
- Defining the `ENGINE_ATTRIBUTE` for tables that are to be loaded.

Lakehouse Auto Parallel Load includes Lakehouse Incremental Load that can refresh tables after an initial load.

5.2.4.1 Lakehouse Auto Parallel Load Schema Inference

Lakehouse Auto Parallel Load includes schema inference, and uses it in one of two ways:

- Lakehouse Auto Parallel Load analyzes the data, infers the table structure, and creates the database and all tables. This only requires the name of the database, the names of each table, the external file parameters, and then Lakehouse Auto Parallel Load generates the `CREATE DATABASE` and `CREATE TABLE` statements. For example, see: Section 5.2.5.1, “Load Configuration”.

  Lakehouse Auto Parallel Load uses header information from the external files to define the column names. If this is not available, Lakehouse Auto Parallel Load defines the column names sequentially: `col_1`, `col_2`, `col_3` ...
- If the tables are already defined, Lakehouse Auto Parallel Load analyzes the data, infers the table structure, and then modifies the structure to avoid errors during data load. For example, if a table defines a column with `TINYINT`, but Lakehouse Auto Parallel Load infers that the data requires `SMALLINT` `MEDIUMINT`, `INT`, or `BIGINT`, then Lakehouse Auto Parallel Load will modify the structure accordingly. If the inferred data type is incompatible with the table definition, Lakehouse Auto Parallel Load raises an error, and specifies the column as `NOT SECONDARY`.

5.2.4.2 Lakehouse Auto Parallel Load with the external_tables Option

For the full Auto Parallel Load syntax, see: Section 2.2.3, “Loading Data Using Auto Parallel Load”.

HeatWave Lakehouse extends Auto Parallel Load with the `external_tables` option. This is a JSON array that includes one or more `db_object`.

Do not use as of MySQL 8.4.0. Use `db_object` with `table` or `exclude_tables` instead. `external_tables` will be deprecated in a future release.

```
db_object: {
     JSON_OBJECT("key","value"["key","value"] ...)
          "key","value": {
          "db_name": "name",
          "tables": JSON_ARRAY(table [, table] ...)
          }
}

table: {
     JSON_OBJECT("key","value"["key","value"] ...)
          "key","value": {
```

```
          "table_name": "name",
          "sampling": true|false,
          "dialect": {dialect_section},
          "file": JSON_ARRAY(file_section [, file_section]...),
          }
}
```

- `db_object`: the details of one or more tables. Each `db_object` contains the following:
  - `db_name`: name of the database. If the database does not exist, Lakehouse Auto Parallel Load creates it during the load process.
  - `tables`: a JSON array of `table`. Each `table` contains the following:
    - `table_name`: the name of the table to load.
    - `sampling`: if set to `true`, the default setting, Lakehouse Auto Parallel Load infers the schema by sampling the data and collect statistics.

      If set to `false`, Lakehouse Auto Parallel Load performs a full scan to infer the schema and collect statistics. Depending on the size of the data, this can take a long time.

      Auto Parallel Load uses the inferred schema to generate `CREATE TABLE` statements. The statistics are used to estimate storage requirements and load times.
    - `dialect`: details about the file format. See the `dialect` parameter in Section 5.2.2, “Lakehouse External Table Syntax”.
    - `file`: the location of the data in Object Storage. This can use a pre-authenticated request or a resource principal, and can be a path to a file, a file prefix, or a file pattern. See the `file` parameter in Section 5.2.2, “Lakehouse External Table Syntax”, and see: Section 5.4, “Access Object Storage”.

Syntax Examples

While it is possible to define the entire load command on a single line, for readability the configuration is divided into option definitions using `SET`.

Define the following option sets:

- Define the name of the database which will store the data.

```
mysql> SET @db_list = '["tpch"]';
```

  This assumes that Lakehouse Auto Parallel Load will analyze the data, infer the table structure, and create the database and all tables. See: Section 5.2.4.2, “Lakehouse Auto Parallel Load with the external_tables Option”.

- Define the `db_object` parameters that will load data from three external sources with Avro, CSV and Parquet format files:

```
mysql> SET @ext_tables = '[
{
  "db_name": "tpch",
  "tables": [{
    "table_name": "supplier_pq",
    "dialect": {
      "format": "parquet"
    },
    "file": [{
```

```
      "prefix": "src_data/parquet/tpch/supplier/",
      "bucket": "myBucket",
      "namespace": "myNamespace",
      "region": "myRegion"
    }]
  },
  {
    "table_name": "nation_csv",
    "dialect": {
      "format": "csv",
      "field_delimiter": "|",
      "record_delimiter": "|\\n",
      "has_header": true
    },
    "file": [{
      "par": "https://objectstorage.../nation.csv"
    }]
  },
  {
    "table_name": "region_avro",
    "dialect": {
      "format": "avro"
    },
    "file": [{
      "par": "https://objectstorage.../region.avro"
    }]
  }]
}
]';
```

- Define the `@options` variable with `SET`:

```
mysql> SET @options = JSON_OBJECT('external_tables', CAST(@ext_tables AS JSON));
```

  Setting `mode` to `dryrun` generates the load script but does not create or load the external tables. For example:

```
mysql> SET @options = JSON_OBJECT('mode', 'dryrun', 'external_tables', CAST(@ext_tables AS JSON));
```

  To implement the changes as part of the load command, set `mode` to `normal`. This is the default, and it is not necessary to add it to the command.

Exclude columns from the loading process with the `exclude_list` option. See Section 2.2.3.2, “Auto Parallel Load Syntax”.

Lakehouse Auto Parallel Load infers the column names for Avro and Parquet files, and also for CSV files if `has_header` is `true`. For these situations, use the column names with the `exclude_list` option.

If the table already exists, but no data has been loaded, use the existing column names with the `exclude_list` option.

For CSV files if `has_header` is `false`, use the generated schema names with the `exclude_list` option. These are: `col_1`, `col_2`, `col_3` ...

5.2.5 How to Load Data from External Storage Using Auto Parallel Load

This section describes loading data with the Auto Parallel Load procedure and schema inference which is part of the procedure. This process does not require an existing external table definition and creates the external table based on schema inference.

The TPCH data set is used in this example, which loads data from external Avro, CSV and Parquet format files.

5.2.5.1 Load Configuration

While it is possible to define the entire load command on a single line, for readability the configuration is divided into option definitions using `SET`.

Define the following option sets:

- Define the `input_list` parameters that will load data from four external sources with Avro, CSV, JSON and Parquet format files:

```
mysql> SET @input_list = '[
  {"db_name": "tpch", "tables": [{
    "table_name": "supplier_pq",
    "engine_attribute": {
      "dialect": {"format": "parquet"},
      "file": [{
        "prefix": "src_data/parquet/tpch/supplier/",
        "bucket": "myBucket",
        "namespace": "myNamespace",
        "region": "myRegion"
      }]
    }
  },
  {
    "table_name": "customer_csv",
    "engine_attribute": {
      "dialect": {
        "format": "csv",
        "field_delimiter": "|",
        "record_delimiter": "|\\n",
        "has_header": true
      },
      "file": [{"par": "https://objectstorage.../customer.csv"}]
    }
  },
  {
    "table_name": "region_avro",
    "engine_attribute": {
      "dialect": {"format": "avro"},
      "file": [{"par": "https://objectstorage.../region.avro"}]
    }
  },
  {
    "table_name": "nation_json",
    "engine_attribute": {
      "dialect": {"format": "json"},
      "file": [{"par": "https://objectstorage.../nation.json"}]
    }
  }
]}
]';
```

- Define the `@options` variable with `SET`. Setting `mode` to `dryrun` generates the load script but does not create or load the external tables. For example:

```
mysql> SET @options = JSON_OBJECT('mode', 'dryrun');
```

  To implement the changes as part of the load command, set `mode` to `normal`. This is the default, and it is not necessary to add it to the command.

  Set `mode` to `validation` to validate the data files against the created table for any potential data errors. For example:

```
mysql> SET @options = JSON_OBJECT('mode', 'validation');
```

> **Note**
>
> `validation` requires the tables to be created first, and it does not load the data to the tables. To load the tables the `mode` must be set to `normal`.

Exclude columns from the loading process with the `exclude_columns` option. See Section 2.2.3.2, “Auto Parallel Load Syntax”.

Lakehouse Auto Parallel Load infers the column names for Avro, JSON and Parquet files, and also for CSV files if `has_header` is `true`. For these situations, use the column names with the `exclude_columns` option.

If the table already exists, but no data has been loaded, use the existing column names with the `exclude_columns` option.

For CSV files if `has_header` is `false`, use the generated schema names with the `exclude_columns` option. These are: `col_1`, `col_2`, `col_3` ...

5.2.5.2 Loading Lakehouse Data

Using the defined options, run the load procedure in the following way:

```
mysql> CALL sys.heatwave_load(CAST(@input_list AS JSON), @options);
```

This example is run without a defined mode, and defaults to normal mode, generating the script and running it. If the mode is set to dryrun, the script is generated and made available to examine in the `LOAD SCRIPT GENERATION` section of the Auto Parallel Load process.

The procedure initializes, runs, and displays a report. The report is divided into the following sections:

- `INITIALIZING HEATWAVE AUTO PARALLEL LOAD`: Lists the load mode, policy, and output mode.

  For example:

```
+-----------------------------------------+
| INITIALIZING HEATWAVE AUTO PARALLEL LOAD |
+-----------------------------------------+
| Version: 3.11                           |
|                                         |
| Load Mode: normal                       |
| Load Policy: disable_unsupported_columns |
| Output Mode: normal                     |
|                                         |
+-----------------------------------------+
6 rows in set (0.02 sec)
```

- `LAKEHOUSE AUTO SCHEMA INFERENCE`: Displays the details of the table, how many rows and columns it contains, its file size, and the name of the schema.

  For example:

```
+-------------------------------------------------------------------------------------------------------------
| LAKEHOUSE AUTO SCHEMA INFERENCE
+-------------------------------------------------------------------------------------------------------------
| Verifying external lakehouse tables: 4
|
| SCHEMA                   TABLE                    TABLE IS           RAW     NUM. OF      ESTIMATED
| NAME                     NAME                     CREATED      FILE SIZE     COLUMNS      ROW COUNT
| ------                   -----                    --------     ---------     -------      ---------
| `tpch`                   `customer_csv`           NO          232.71 GiB           8          1.5 B
| `tpch`                   `nation_json`            NO            3.66 KiB           1             25
| `tpch`                   `region_avro`            NO           476 bytes           3              9
```

```
| `tpch`                   `supplier_pq`            NO           7.46 GiB           7          100 M
|
| New schemas to be created: 1
| External lakehouse tables to be created: 4
|
+-------------------------------------------------------------------------------------------------------
13 rows in set (21.06 sec)
```

- `OFFLOAD ANALYSIS`: Displays an analysis of the number and name of the tables and columns which can be offloaded to HeatWave.

  For example:

```
+-------------------------------------------------------------------------+
| OFFLOAD ANALYSIS                                                        |
+-------------------------------------------------------------------------+
| Verifying input schemas: 1                                              |
| User excluded items: 0                                                  |
|                                                                         |
| SCHEMA                        OFFLOADABLE    OFFLOADABLE     SUMMARY OF |
| NAME                               TABLES        COLUMNS     ISSUES     |
| ------                        -----------    -----------     ---------- |
| `tpch`                                4             19                 |
|                                                                         |
| Total offloadable schemas: 1                                            |
|                                                                         |
+-------------------------------------------------------------------------+
10 rows in set (21.09 sec)
```

- `CAPACITY ESTIMATION`: Displays the HeatWave cluster and MySQL node memory requirement to process the data and an estimation of the load time.

  For example:

```
+-------------------------------------------------------------------------------------------------------
| CAPACITY ESTIMATION
+-------------------------------------------------------------------------------------------------------
| Default encoding for string columns: VARLEN (unless specified in the schema)
| Estimating memory footprint for 1 schema(s)
|
|                                TOTAL       ESTIMATED       ESTIMATED       TOTAL     DICTIONARY      VARLE
| SCHEMA                    OFFLOADABLE   HEATWAVE NODE      MYSQL NODE      STRING        ENCODED     ENCODE
| NAME                           TABLES       FOOTPRINT       FOOTPRINT     COLUMNS        COLUMNS     COLUMN
| ------                    -----------       ---------       ---------     -------     ----------     ------
| `tpch`                             4      193.39 GiB        1.44 MiB          12              0          1
|
| Sufficient MySQL host memory available to load all tables.
| Sufficient HeatWave cluster memory available to load all tables.
|
+-------------------------------------------------------------------------------------------------------
12 rows in set (21.10 sec)
```

  > **Note**
  >
  > If there is insufficient memory, update the nodes before proceeding with the load.

- `EXECUTING LOAD`: Displays information about the generated script and approximate loading time. For example:

```
+-------------------------------------------------------------------------------------------------------
| EXECUTING LOAD SCRIPT
+-------------------------------------------------------------------------------------------------------
| HeatWave Load script generated
|   Retrieve load script containing 9 generated DDL command(s) using the query below:
| Deprecation Notice: "heatwave_load_report" will be deprecated, please switch to "heatwave_autopilot_report
```

```
|   SELECT log->>"$.sql" AS "Load Script" FROM sys.heatwave_autopilot_report WHERE type = "sql" ORDER BY
|
| Adjusting load parallelism dynamically per internal/external table.
| Using current parallelism of 4 thread(s) as maximum for internal tables.
|
| Warning: Executing the generated script may alter column definitions and secondary engine flags in the
|
| Using SQL_MODE: ONLY_FULL_GROUP_BY,STRICT_TRANS_TABLES,NO_ZERO_IN_DATE,NO_ZERO_DATE,ERROR_FOR_DIVISION
|
| Proceeding to load 4 table(s) into HeatWave.
|
| Applying changes will take approximately 22.08 min
|
+-----------------------------------------------------------------------------------------------------
16 rows in set (22.08 sec)
```

- `SCHEMA CREATION`: Displays information about the schema creation process and duration.

  For example:

```
+----------------------------------+
| SCHEMA CREATION                  |
+----------------------------------+
| Schema `tpch` creation succeeded! |
| Warnings/errors encountered: 0    |
| Elapsed time: 2.62 ms             |
|                                  |
+----------------------------------+
4 rows in set (14.70 sec)
```

- `LOADING TABLE`: Displays information on the table load process.

  For example:

```
+---------------------------------------+
| TABLE LOAD                            |
+---------------------------------------+
| TABLE (1 of 4): `tpch`.`customer_csv`  |
| Commands executed successfully: 2 of 2 |
| Warnings encountered: 0                |
| Table load succeeded!                  |
|   Total columns loaded: 8              |
|   Elapsed time: 19.33 min              |
|                                       |
+---------------------------------------+
7 rows in set (19 min 41.74 sec)

+---------------------------------------+
| TABLE LOAD                            |
+---------------------------------------+
| TABLE (2 of 4): `tpch`.`nation_json`   |
| Commands executed successfully: 2 of 2 |
| Warnings encountered: 0                |
| Table load succeeded!                  |
|   Total columns loaded: 1              |
|   Elapsed time: 3.70 s                 |
|                                       |
+---------------------------------------+
7 rows in set (19 min 45.44 sec)

+---------------------------------------+
| TABLE LOAD                            |
+---------------------------------------+
| TABLE (3 of 4): `tpch`.`region_avro`   |
| Commands executed successfully: 2 of 2 |
| Warnings encountered: 0                |
```

```
| Table load succeeded!                  |
|   Total columns loaded: 3              |
|   Elapsed time: 3.79 s                 |
|                                        |
+----------------------------------------+
7 rows in set (19 min 49.24 sec)

+----------------------------------------+
| TABLE LOAD                             |
+----------------------------------------+
| TABLE (4 of 4): `tpch`.`supplier_pq`   |
| Commands executed successfully: 2 of 2 |
| Warnings encountered: 0                |
| Table load succeeded!                  |
|   Total columns loaded: 7              |
|   Elapsed time: 1.96 min               |
|                                        |
+----------------------------------------+
7 rows in set (21 min 46.98 sec)
```

- `LOAD SUMMARY`: Displays a summary of the load process.

  For example:

```
+--------------------------------------------------------------------------------+
| LOAD SUMMARY                                                                   |
+--------------------------------------------------------------------------------+
|                                                                                |
| SCHEMA                          TABLES       TABLES      COLUMNS         LOAD |
| NAME                            LOADED       FAILED       LOADED     DURATION |
| ------                          ------       ------      -------     -------- |
| `tpch`                               4            0           19    21.41 min |
|                                                                                |
+--------------------------------------------------------------------------------+
6 rows in set (21 min 46.98 sec)
```

5.2.5.3 Generated Load Script

The following is a typical load script generated by the Auto Parallel Load process. It creates a schema, then creates a table with columns matching those found in the source bucket.

```
mysql> SELECT log->>"$.sql" AS "Load Script"
        FROM sys.heatwave_autopilot_report
        WHERE type = "sql"
        ORDER BY id;

+------------------------------------------------------------------------------------------------------
| Load Script
+------------------------------------------------------------------------------------------------------
| CREATE DATABASE `tpch`;
| CREATE TABLE `tpch`.`customer_csv`(
|   `C_CUSTKEY` int unsigned NOT NULL,
|   `C_NAME` varchar(19) NOT NULL COMMENT 'RAPID_COLUMN=ENCODING=VARLEN',
|   `C_ADDRESS` varchar(40) NOT NULL COMMENT 'RAPID_COLUMN=ENCODING=VARLEN',
|   `C_NATIONKEY` tinyint unsigned NOT NULL,
|   `C_PHONE` varchar(15) NOT NULL COMMENT 'RAPID_COLUMN=ENCODING=VARLEN',
|   `C_ACCTBAL` decimal(6,2) NOT NULL,
|   `C_MKTSEGMENT` varchar(10) NOT NULL COMMENT 'RAPID_COLUMN=ENCODING=VARLEN',
|   `C_COMMENT` varchar(116) NOT NULL COMMENT 'RAPID_COLUMN=ENCODING=VARLEN'
| ) ENGINE=lakehouse SECONDARY_ENGINE=RAPID
|   ENGINE_ATTRIBUTE='{"file": [{"par": "https://objectstorage.../customer.csv"}],
|                      "dialect": {"format": "csv", "field_delimiter": "|", "record_delimiter": "|\\n"}}';
| ALTER TABLE /*+ AUTOPILOT_DISABLE_CHECK */ `tpch`.`customer_csv` SECONDARY_LOAD;
| CREATE TABLE `tpch`.`nation_json`(`col_1` json NOT NULL COMMENT 'RAPID_COLUMN=ENCODING=VARLEN')
| ENGINE=lakehouse SECONDARY_ENGINE=RAPID
| ENGINE_ATTRIBUTE='{"file": [{"par": "https://objectstorage.../nation.json"}],
```

```
|                     "dialect": {"format": "json"}}';
| ALTER TABLE /*+ AUTOPILOT_DISABLE_CHECK */ `tpch`.`nation_json` SECONDARY_LOAD;
| CREATE TABLE `tpch`.`region_avro`(
|   `R_REGIONKEY` tinyint unsigned NOT NULL,
|   `R_NAME` varchar(11) NOT NULL COMMENT 'RAPID_COLUMN=ENCODING=VARLEN',
|   `R_COMMENT` varchar(115) NOT NULL COMMENT 'RAPID_COLUMN=ENCODING=VARLEN'
| ) ENGINE=lakehouse SECONDARY_ENGINE=RAPID
| ENGINE_ATTRIBUTE='{"file": [{"par": "https://objectstorage.../region.avro"}],
|                     "dialect": {"format": "avro"}}';
| ALTER TABLE /*+ AUTOPILOT_DISABLE_CHECK */ `tpch`.`region_avro` SECONDARY_LOAD;
| CREATE TABLE `tpch`.`supplier_pq`(
|   `S_SUPPKEY` int,
|   `S_NAME` varchar(19) COMMENT 'RAPID_COLUMN=ENCODING=VARLEN',
|   `S_ADDRESS` varchar(40) COMMENT 'RAPID_COLUMN=ENCODING=VARLEN',
|   `S_NATIONKEY` int,
|   `S_PHONE` varchar(15) COMMENT 'RAPID_COLUMN=ENCODING=VARLEN',
|   `S_ACCTBAL` decimal(15,2),
|   `S_COMMENT` varchar(100) COMMENT 'RAPID_COLUMN=ENCODING=VARLEN'
| ) ENGINE=lakehouse SECONDARY_ENGINE=RAPID
| ENGINE_ATTRIBUTE='{"file": [{"prefix": "src_data/parquet/tpch/supplier/",
|                              "bucket": "myBucket",
|                              "namespace": "myNamespace",
|                              "region": "myRegion"}],
|                     "dialect": {"format": "parquet"}}';
| ALTER TABLE /*+ AUTOPILOT_DISABLE_CHECK */ `tpch`.`supplier_pq` SECONDARY_LOAD;
+-----------------------------------------------------------------------------------------------------------------------
```

> **Note**
>
> The output above is displayed in a readable format. The actual `CREATE TABLE` output is generated on a single line.

5.2.6 Lakehouse Incremental Load

As of MySQL 9.0.0, Lakehouse includes Lakehouse Incremental Load. After an initial data load, the data can become stale, and the external tables require a refresh. Subsequent calls to Auto Parallel Load will refresh the data.

Lakehouse Incremental Load uses the existing `ENGINE_ATTRIBUTE`, see: Section 5.2.2, “Lakehouse External Table Syntax”. It is not possible to change file parameters, file names, file patterns or file prefixes. However, it is possible to add or remove individual files if they still match the defined file pattern or file prefix.

After the initial data load, a subsequent call to Auto Parallel Load with the `refresh_external_tables` option set to `true` will refresh the data, see: Section 2.2.3.2, “Auto Parallel Load Syntax”.

If a subsequent call to Auto Parallel Load includes tables that have not yet been loaded to Lakehouse, then Auto Parallel Load will load them for the first time.

A call to Auto Parallel Load might contain both loaded and unloaded tables. Those that are unloaded will be loaded, and those that are already loaded will be refreshed.

5.3 Loading Unstructured Data to HeatWave Lakehouse

HeatWave Lakehouse lets you load unstructured data from Object Storage using a HeatWave Vector Store.

For more information, see Setting Up a Vector Store.

5.4 Access Object Storage

HeatWave on OCI can access external data stored on Oracle Cloud Infrastructure. HeatWave on AWS can access external data stored on AWS. Each service uses its own terminology, however the syntax is generic, and each service can use the same syntax for data stored on that service.

HeatWave Lakehouse provides two methods to access Object Storage:

- A pre-authenticated request.
- A resource principal.

See: Section 5.2.2, “Lakehouse External Table Syntax”.

5.4.1 Pre-Authenticated Requests

Pre-authenticated requests allow users to access a bucket or an object without having their own credentials. A pre-authenticated request is a unique URL, and anyone who has this URL can access the Object Storage resources with standard HTTP tools like curl and wget.

For more information about pre-authenticated requests, see the Oracle Cloud Infrastructure documentation at Using Pre-Authenticated Requests.

5.4.1.1 Recommendations

The following recommendations apply to pre-authenticated requests created for HeatWave Lakehouse:

- Only use read-only pre-authenticated requests with Lakehouse.
- Set a short expiration date for the pre-authenticated request URL that matches the data loading plan.
- Do not make a pre-authenticated request URL publicly accessible.
- If the target defines a bucket or uses a prefix or pattern:
  - Use **Enable Object Listing** when creating the pre-authenticated request in the HeatWave Console.
  - When creating the pre-authenticated request from the command line, include the `--access-type AnyObjectRead` parameter.

    Note

    Use a resource principal for access to more sensitive data in Object Storage as it is more secure.

5.4.1.2 Pre-Authenticated Request Examples

The following example creates a pre-authenticated request named MyOneObjectReadPAR for a bucket named MyParBucket, and grants read-only access to a single object in that bucket:

```
$>oci os preauth-request create --namespace MyNamespace --bucket-name MyParBucket \
--name MyOneObjectReadPAR --access-type AnyObjectRead \
--time-expires="2022-11-21T23:00:00+00:00" --object-name data-file-01.csv
```

This command returns a result similar to the following:

```
{
```

```
  "data": {
    "access-type": "AnyObjectRead",
    "access-uri": "/p/alphanumericString/n/namespace/b/bucketName/o/",
    "bucket-listing-action": "Deny",
    "id": "alphanumericString",
    "name": "MyOneObjectReadPAR",
    "object-name": "data-file-01.csv",
    "time-created": "2022-12-08T18:51:34.491000+00:00",
    "time-expires": "2022-12-09T23:07:04+00:00"
  }
}
```

The default value of `bucket-listing-action` is Deny. This allows access to the defined object, `data-file-01.csv`, only.

The following example creates a PAR named MyAllObjectsReadPAR,for a file named `data-file-01.csv`, in a bucket named MyParBucket, and grants read-only access to all objects in the bucket:

```
$>oci os preauth-request create --namespace MyNamespace --bucket-name MyParBucket \
--name MyAllObjectsReadPAR --access-type AnyObjectRead \
--time-expires="2022-11-21T23:00:00+00:00" --bucket-listing-action ListObjects
```

This command returns a result similar to the following:

```
{
  "data": {
    "access-type": "AnyObjectRead",
    "access-uri": "/p/alphanumericString/n/namespace/b/bucketName/o/",
    "bucket-listing-action": "ListObjects",
    "id": "alphanumericString",
    "name": "MyAllObjectsReadPAR",
    "object-name": null,
    "time-created": "2022-12-08T18:51:34.491000+00:00",
    "time-expires": "2022-12-09T23:07:04+00:00"
  }
}
```

The defined value of `bucket-listing-action` is ListObjects. This allows the listing of all objects in the named bucket.

All pre-authenticated requests created with the command line are available to view in the HeatWave Console. To view the pre-authenticated requests, navigate to the HeatWave Console page for the bucket and select **Pre-Authenticated Requests** on the **Resources** section. You can also list the PARs from the command line, with the following command:

```
$>oci os preauth-request list --bucket-name=name
```

5.4.2 Resource Principals

A resource principal consists of a temporary session token and secure credentials that enables the MySQL DB System to authenticate itself to other services. Using a resource principal to access services, the token stored with the credentials on HeatWave is only valid for the resources to which the dynamic group has been granted access. To use a resource principal, you or your tenancy administrator define the policies and a dynamic group that allows access to resources with a resource principal.

For HeatWave on OCI, see Resource Principals in the *HeatWave on OCI Service Guide*.

5.4.2.1 Configuring a Tenancy for Resource Principal Data Loading

This section describes how to define the dynamic group and policy required to enable the MySQL DB System to access an OCI Object Storage bucket.

> **Note**
>
> It is assumed you have read the prerequisites and instructions documented here: Managing Dynamic Groups and are familiar with Oracle Cloud Infrastructure Identity and Access Management (IAM) groups and policies.

Dynamic Group

Dynamic groups allow you to group MySQL DB Systems as principal actors, similar to user groups. You can then create policies to permit MySQL DB Systems in these groups to make API calls against services, such as Object Storage. Membership in the group is determined by a set of criteria called matching rules.

The following example shows a matching rule including all MySQL DB Systems in the defined compartment:

```
"ALL{resource.type='mysqldbsystem', resource.compartment.id = 'ocid1.compartment.oc1..alphanumericString' }"
```

Dynamic groups require a name, description, and matching rule.

For more information, see Writing Matching Rules to Define Dynamic Groups.

Policy

Policies define what your groups can and cannot do. For HeatWave Lakehouse to access Object Storage, you must define a policy which grants the dynamic group's resources access to buckets and their contents in a specific compartment.

For example, the following policy grants the dynamic group `Lakehouse-dynamicGroup` read-only access to the buckets and objects contained in those buckets in the compartment `Lakehouse-Data`:

```
allow dynamic-group Lakehouse-dynamicGroup to read buckets in compartment Lakehouse-Data
allow dynamic-group Lakehouse-dynamicGroup to read objects in compartment Lakehouse-Data
```

For more information, see Writing Policies for Dynamic Groups.

5.5 External Table Recovery

In the event of a problem where query processing stops or the HeatWave cluster stops, the external tables are recovered when the Cluster resumes and begins processing queries again using the HeatWave recovery process. See HeatWave Cluster Failure and Recovery in the *HeatWave on OCI Service Guide*.

5.6 Data Types

For information on supported data types, see Section 2.10, “Supported Data Types”. For limitations, see Section 2.18.2, “Data Type Limitations”.

5.6.1 Parquet Data Type Conversions

See: Parquet Types.

See: Section 5.8.5, “Lakehouse Limitations for the Parquet File Format”.

Table 5.1 Parquet Data Type Conversions

MySQL Data Type	Parquet Logical Type	Comments
`BOOL`	`STRING`	Valid values: `TRUE`, `FALSE`, `T`, `F`, `true`, `false`, `0`, `1`

MySQL Data Type	Parquet Logical Type	Comments
BOOL	INT	All signed and unsigned integers with any bit width, but the value must be 0 or 1
BOOL	None	Physical Type FLOAT or DOUBLE
CHAR, VARCHAR, TEXT	DECIMAL	The maximum precision is 38 and Lakehouse supports the Physical Type FIXED_LEN_BYTE_ARRAY
CHAR, VARCHAR, TEXT	STRING	
CHAR, VARCHAR, TEXT	INT	All signed and unsigned integers
CHAR, VARCHAR, TEXT	None	Physical Type FLOAT or DOUBLE
DATE	STRING	
DATE	DATE	
DATE	TIMESTAMP	isAdjustedToUTC = false, timeUnit = x: Lakehouse only loads the date portion
DATE	None	Physical Type INT96: Lakehouse only loads the date portion
DATETIME	STRING	DATETIME accepts the range from 0000-00-00 to 9999-12-12
DATETIME	DATE	DATETIME accepts the range from 0000-00-00 to 9999-12-12
DATETIME	TIME	isAdjustedToUTC = false, timeUnit = x: DATETIME accepts the range from 0000-00-00 to 9999-12-12
DATETIME	TIMESTAMP	isAdjustedToUTC = false, timeUnit = x: DATETIME accepts the range from 0000-00-00 to 9999-12-12
DATETIME	None	Physical Type INT96: DATETIME accepts the range from 0000-00-00 to 9999-12-12
DECIMAL	DECIMAL	The maximum precision is 38 and Lakehouse supports the Physical Type FIXED_LEN_BYTE_ARRAY
DECIMAL	STRING	
DECIMAL	INT	All signed and unsigned integers
DECIMAL	None	Physical Type FLOAT or DOUBLE
DOUBLE, FLOAT	DECIMAL	The maximum precision is 38 and Lakehouse supports the Physical Type FIXED_LEN_BYTE_ARRAY
DOUBLE, FLOAT	STRING	
DOUBLE, FLOAT	INT	All signed and unsigned integers
DOUBLE, FLOAT	None	Physical Type FLOAT or DOUBLE

MySQL Data Type	Parquet Logical Type	Comments
ENUM	STRING	
ENUM	ENUM	Physical Type BYTE_ARRAY or FIXED_LEN_BYTE_ARRAY
ENUM	INT	All signed and unsigned integers
ENUM	None	Physical Type FLOAT or DOUBLE
JSON	STRING	
JSON	JSON	
JSON	None	Physical Type BYTE_ARRAY, Binary encoded. The non-UTF-8 encoding should be JSON binary, see: The JSON Data Type.
TIME	STRING	
TIME	TIME	isAdjustedToUTC = false, timeUnit = x
TIMESTAMP	STRING	TIMESTAMP accepts the range from 1970-01-01 to 2038-01-19. Lakehouse might not load out of range values correctly. Use DATETIME instead.
TIMESTAMP	DATE	TIMESTAMP accepts the range from 1970-01-01 to 2038-01-19. Lakehouse might not load out of range values correctly. Use DATETIME instead.
TIMESTAMP	TIME	isAdjustedToUTC = false, timeUnit = x: TIMESTAMP accepts the range from 1970-01-01 to 2038-01-19. Lakehouse might not load out of range values correctly. Use DATETIME instead.
TIMESTAMP	TIMESTAMP	isAdjustedToUTC = false, timeUnit = x: TIMESTAMP accepts the range from 1970-01-01 to 2038-01-19. Lakehouse might not load out of range values correctly. Use DATETIME instead.
TIMESTAMP	None	Physical Type INT96: TIMESTAMP accepts the range from 1970-01-01 to 2038-01-19. Lakehouse might not load out of range values correctly. Use DATETIME instead.
All signed and unsigned TINYINT, SMALLINT,	DECIMAL	The maximum precision is 38 and Lakehouse supports the Physical Type FIXED_LEN_BYTE_ARRAY

MySQL Data Type	Parquet Logical Type	Comments
MEDIUMINT, INTEGER, INT, BIGINT		
All signed and unsigned TINYINT, SMALLINT, MEDIUMINT, INTEGER, INT, BIGINT	STRING	
All signed and unsigned TINYINT, SMALLINT, MEDIUMINT, INTEGER, INT, BIGINT	INT	All signed and unsigned integers
All signed and unsigned TINYINT, SMALLINT, MEDIUMINT, INTEGER, INT, BIGINT	None	Physical Type FLOAT or DOUBLE
YEAR	STRING	The value should be in the range 0 to 99 or the range 1901 to 2155
YEAR	INT	The value should be in the range 0 to 99 or the range 1901 to 2155
YEAR	None	Physical Type FLOAT or DOUBLE: The value should be in the range 0 to 99 or the range 1901 to 2155
YEAR	None	Physical Type INT96: The value should be in the range 1901 to 2155, Lakehouse only loads the year portion
YEAR	DATE	The value should be in the range 1901 to 2155, Lakehouse only loads the year portion
YEAR	TIMESTAMP	isAdjustedToUTC = false: The value should be in the range 1901 to 2155, Lakehouse only loads the year portion

5.7 HeatWave Lakehouse Error Messages

See MySQL 9.0 Error Message Reference for Lakehouse error messages.

Some Lakehouse error and warning messages include the URL to the external object. The URL can contain up to 1024 characters, and exceed the error message limit of 512 characters. As of MySQL 9.0.1-u1, the truncated message will include a MySQL command to access the full error message. For example:

- Error message:

```
ERROR HY000: Message was truncated, use
<SELECT data FROM performance_schema.error_log
WHERE thread_id=15 AND data LIKE 'ERR_LH_test.t1_long_name_6045%'
ORDER BY LOGGED DESC>
to obtain more detailed information about the warning/error
```

- Warning message:

```
Warning 6045 Message was truncated, use
<SELECT data FROM performance_schema.error_log
WHERE thread_id=15 AND data LIKE 'WARN_LH_test.t1_long_name_6045_0%'
ORDER BY LOGGED DESC>
to obtain more detailed information about the warning/error
```

As of MySQL 9.0.1-u1, it is possible to filter out Lakehouse warning messages from the console, MySQL Shell and error log following a load command. The console **Total Warnings** will include both the displayed and the filtered warnings count. This does not filter error messages.

Use the `lakehouse_filter_warning_codes_list` session variable to filter warning messages by error code. For example:

```
mysql> SET @@session.lakehouse_filter_warning_codes_list = '6017,5099,6018';
```

Use the `lakehouse_filter_warning_modes_list` session variable to filter warning messages by mode. For example:

```
mysql> SET @@session.lakehouse_filter_warning_modes_list =
       'NUMERIC_TRUNCATION,STRING_TRUNCATION,EMPTY_FILES,DUPLICATE_FILES';
```

The supported modes are:

- Mode: `AVRO_FILE_BLOCK_HEADER` filters the following warning messages:
  - `ER_LH_AVRO_CANNOT_PARSE_HEADER`
  - `ER_LH_AVRO_HEADER_METADATA_ERR`
  - `ER_LH_FORMAT_HEADER_NO_MAGIC_BYTES`
  - `ER_LH_AVRO_HEADER_NO_SCHEMA`
  - `ER_LH_AVRO_INVALID_BLOCK_RECORD_COUNT`
  - `ER_LH_AVRO_INVALID_BLOCK_SIZE`
  - `ER_LH_AVRO_NO_CODEC_IN_HEADER`
- Mode: `DUPLICATE_FILES` filters the following warning messages:
  - `ER_LH_DUPLICATE_FILE`
- Mode: `EMPTY_FILES` filters the following warning messages:
  - `ER_LH_EMPTY_FILE`
- Mode: `EMPTY_MISSING_COLUMNS` filters the following warning messages:
  - `ER_LH_COLUMN_MISMATCH_ERR`
  - `ER_LH_COLUMN_MAX_ERR`
  - `ER_LH_WARN_COL_MISSING_NOT_NULLABLE`
  - `ER_LH_COL_IS_EMPTY_WARN`
- Mode: `INFER_SKIPPED` filters the following warning messages:
  - `ER_LH_INFER_FILE_HAS_NO_DATA`

    - ER_LH_WARN_INFER_SKIPPED_FILES
    - ER_LH_WARN_INFER_SKIPPED_LINES
- Mode: MISSING_FILES filters the following warning messages:
    - ER_LH_NO_FILES_FOUND
- Mode: NULL_COLUMNS filters the following warning messages:
    - ER_LH_WARN_COL_MISSING_NOT_NULLABLE
- Mode: NUMERIC_PARSING for the int, decimal, float, and double data types filters the following warning messages:
    - ER_LH_BAD_VALUE
    - ER_LH_DECIMAL_OOM_ERR
    - ER_LH_DECIMAL_UNKNOWN_ERR
    - ER_LH_PARQUET_DECIMAL_CONVERSION_ERR
    - ER_LH_REAL_IS_NAN
- Mode: NUMERIC_TRUNCATION for the int, decimal, float, and double data types filters the following warning messages:
    - ER_LH_DECIMAL_PRECISION_EXCEEDS_SCHEMA
    - ER_LH_EXCEEDS_MAX
    - ER_LH_EXCEEDS_MIN
    - ER_LH_OUT_OF_RANGE
    - ER_LH_WARN_DECIMAL_ROUNDING
    - ER_LH_WARN_EXCEEDS_MIN_TRUNCATING
    - ER_LH_WARN_EXCEEDS_MAX_TRUNCATING
    - ER_LH_WARN_TRUNCATED
- Mode: STRING_PARSING for the char, enum, text, and varchar data types filters the following warning messages:
    - ER_LH_BAD_VALUE
    - ER_LH_CANNOT_CONVERT_STRING
- Mode: STRING_TRUNCATION for the char, text, and varchar data types filters the following warning messages:
    - ER_LH_STRING_TOO_LONG
- Mode: TEMPORAL_PARSING for the date, datetime, time, and timestamp data types filters the following warning messages:

    - `ER_LH_BAD_VALUE`
    - `ER_LH_DATETIME_FORMAT`
- Mode: `TEMPORAL_TRUNCATION` for the `date`, `datetime`, `time`, and `timestamp` data types filters the following warning messages:
    - `ER_LH_WARN_TRUNCATED`
- Mode: `PARQUET_SCHEMA` filters the following warning messages:
    - `ER_LH_PARQUET_CANNOT_LOCATE_SCHEMA`
    - `ER_LH_PARQUET_SCHEMA_MISMATCH`

5.8 HeatWave Lakehouse Limitations

The following limitations apply to Lakehouse. For HeatWave limitations, see: Section 2.18, “HeatWave MySQL Limitations”.

5.8.1 Lakehouse Limitations for all File Formats

- See Section 2.18.9, “Other Limitations” for all HeatWave MySQL query related limitations.
- Do not create Lakehouse tables on the source DB in a replicated MySQL DB System if any of the replicas are outside HeatWave on OCI or HeatWave on AWS. This will cause replication errors.
- Before MySQL 8.4.0-u2, a replication channel might fail if a HeatWave Cluster is added to a replica of a MySQL DB System, and later manually stopped.
- It is not possible to dump external tables using the MySQL Shell export utilities, such as `dumpInstance()`. External tables are not replicated to InnoDB storage and cannot be exported. To export InnoDB data from a Lakehouse enabled database, exclude the external tables with an `excludeTables` option.
- It is not possible to restore a backup from a Lakehouse enabled MySQL DB System to a standalone MySQL DB System.
- A Lakehouse enabled MySQL DB System can support a maximum of 512 nodes.
- Before MySQL 8.4.0, Lakehouse does not enforce any specified constraints. For example, Lakehouse does not enforce primary key uniqueness constraints. MySQL 8.4.0 removes this limitation.
- Before MySQL 9.0.1-u1, the limit for the Lakehouse error message count is 100. As of MySQL 9.0.1-u1, Lakehouse supports `max_error_count`.
- The `lakehouse_filter_warning_codes_list` session variable has a limit of 50 codes and 250 characters.
- HeatWave Lakehouse does not support the following:
    - DML statements:
        - `INSERT`
        - `UPDATE`
        - `DELETE`

    - `REPLACE`
- The `CREATE TABLESPACE` statement.
- The following options for the `CREATE TABLE` statement:
    - `AUTOEXTEND_SIZE`
    - `AVG_ROW_LENGTH`
    - `CHECKSUM`
    - `COMPRESS`
    - `CONNECTION`
    - `DATADIR`
    - `DELAY_KEY_WRITE`
    - `ENCRYPT`
    - `INDEXDIR`
    - `INSERT_METHOD`
    - `KEY_BLOCK_SIZE`
    - `MAX_ROWS`
    - `MIN_ROWS`
    - `PACK_KEYS`
    - `PASSWORD`
    - `ROW_FORMAT`
    - `STATS_AUTO_RECALC`
    - `STATS_PERSISTENT`
    - `STATS_SAMPLE_PAGES`
    - `UNION`
- The default expression for a column definition for the `CREATE TABLE` statement.
- Creating temporary tables.
- Creating `AUTO_INCREMENT` columns.

  Therefore, Lakehouse is not compatible with `REQUIRE_TABLE_PRIMARY_KEY_CHECK` = `GENERATE`.
- Creating triggers.
- Running `ANALYZE TABLE`.

- Running `ALTER TABLE` statements that construct indexes, `ADD` or `DROP` columns, or add enforced check constraints.
- `SELECT` statements without `RAPID` as the secondary engine.
- Hidden columns.
- Index construction.
- Keys with column prefixes.
- Running any statement with a `STORAGE` clause.
- The use of `CURRENT_TIMESTAMP()`, `CURRENT_TIMESTAMP` or `NOW()` as a default value for a timestamp column. Also an `UPDATE` statement with `CURRENT_TIMESTAMP()`, `CURRENT_TIMESTAMP` or `NOW()`. Enable `explicit_defaults_for_timestamp` to use `ALTER TABLE` and `CREATE TABLE` statements with Lakehouse tables that have a timestamp column.

5.8.2 Lakehouse Limitations for the Avro Format Files

- Avro file headers must be less than 1MB.
- The data in an Avro block must be no larger than 64MiB before applying any compression.
- Lakehouse only supports uncompressed Avro blocks, or Avro blocks compressed with Snappy or Deflate algorithms. Lakehouse does not support other compression algorithms.
- Lakehouse does not support these data types in Avro files: `Array`, `Map`, and nested records. Lakehouse marks columns with these data types as `NOT SECONDARY`, and does not load them.
- Lakehouse only supports an Avro union between one supported data type and `NULL`.
- The Avro `DECIMAL` data type is limited to the scale and precision supported by MySQL.
- Avro data with `FIXED` values or values in `BYTES` are only supported with the Avro `DECIMAL` data type.

5.8.3 Lakehouse Limitations for the CSV File Format

- MySQL does not recognise NaN values in CSV files. Replace all NaN values with NULL.
- Lakehouse does not support CSV files with more than 4MB per line.

5.8.4 Lakehouse Limitations for the JSON File Format

- As of MySQL 8.4.0, Lakehouse only supports Newline Delimited JSON.
- Tables created with `json` format must only have a single column that conforms to the `JSON` data type, see: The JSON Data Type.
- Lakehouse can only load 64KB of data for each line, and ignores any line with more than 64KB of data.

5.8.5 Lakehouse Limitations for the Parquet File Format

- NaN values in Parquet files are loaded as NULL.
- Lakehouse does not support Parquet files with a row group size of more than 500MB.

- Lakehouse does not support the following data types in Parquet files. Lakehouse marks columns with these data types as `NOT SECONDARY`, and does not load them.
  - `BSON`.
  - `ENUM`.
  - `Interval`.
  - `List`.
  - `Map`.
  - `Unknown`.
  - `UUID`.
- Do not use strict SQL mode if the inferred schema differs from the table schema.

Chapter 6 System and Status Variables

Table of Contents

6.1 System Variables

HeatWave maintains several variables that configure its operation. Variables are set when the HeatWave Cluster is enabled. Most HeatWave variable settings are managed by OCI and cannot be modified directly.

- `bulk_loader.data_memory_size`

Command-Line Format	`--bulk_loader.data_memory_size=#`
System Variable	`bulk_loader.data_memory_size`
Scope	Global
Dynamic	Yes
`SET_VAR` Hint Applies	No
Type	Integer
Default Value	`1073741824`
Minimum Value	`67108864`
Maximum Value	`1099511627776`

Specifies the amount of memory to use for `LOAD DATA` with `ALGORITHM=BULK`, in bytes. See: Section 2.17, “Bulk Ingest Data to MySQL Server”.

- `lakehouse_filter_warning_codes_list`

Command-Line Format	`--lakehouse_filter_warning_codes_list=codes`
Introduced	9.0.1
System Variable	`lakehouse_filter_warning_codes_list`
Scope	Session
Dynamic	Yes
`SET_VAR` Hint Applies	No
Type	String

Specifies the warning codes for Lakehouse to filter.

- `lakehouse_filter_warning_modes_list`

Command-Line Format	`--lakehouse_filter_warning_modes_list=modes`
Introduced	9.0.1
System Variable	`lakehouse_filter_warning_modes_list`

Scope	Session
Dynamic	Yes
SET_VAR Hint Applies	No
Type	String

Specifies the warning modes for Lakehouse to filter.

- rapid_compression

System Variable	rapid_compression
Scope	Session
Dynamic	Yes
SET_VAR Hint Applies	No
Type	Enumeration
Default Value	AUTO
Valid Values	ON OFF

Whether to enable or disable data compression before loading data into HeatWave. Data compression is enabled by default. The setting does not affect data that is already loaded. See Section 2.2.6, “Data Compression”.

The default option is AUTO which automatically chooses the best compression algorithm for each column.

- rapid_bootstrap

Command-Line Format	--rapid-bootstrap[={OFF|ON|IDLE}]
System Variable	rapid_bootstrap
Scope	Global
Dynamic	Yes
SET_VAR Hint Applies	No
Type	Enumeration
Default Value	OFF
Valid Values	IDLE SUSPEND ON

The setting for this variable is managed by OCI and cannot be modified directly. Defines the HeatWave Cluster bootstrap state. States include:

- OFF

  The HeatWave Cluster is not bootstrapped (not initialized).

- `IDLE`

  The HeatWave Cluster is idle (stopped).

- `SUSPENDED`

  The HeatWave Cluster is suspended. The `SUSPENDED` state is a transition state between `IDLE` and `ON` that facilitates planned restarts of the HeatWave Cluster.

- `ON`

  The HeatWave Cluster is bootstrapped (started).

- `rapid_dmem_size`

Command-Line Format	`--rapid-dmem-size=#`
System Variable	`rapid_dmem_size`
Scope	Global
Dynamic	Yes
`SET_VAR` Hint Applies	No
Type	Integer
Default Value	`2048`
Minimum Value	`512`
Maximum Value	`2097152`

The setting for this variable is managed by OCI and cannot be modified directly. Specifies the amount of DMEM available on each core of each node, in bytes.

- `rapid_memory_heap_size`

Command-Line Format	`--rapid-memory-heap-size=#`
System Variable	`rapid_memory_heap_size`
Scope	Global
Dynamic	Yes
`SET_VAR` Hint Applies	No
Type	Integer
Default Value	`unlimited`
Minimum Value	`67108864`
Maximum Value	`unlimited`

The setting for this variable is managed by OCI and cannot be modified directly. Defines the amount of memory available for the HeatWave plugin, in bytes. Ensures that HeatWave does not use more memory than is allocated to it.

- `rapid_execution_strategy`

Command-Line Format	`--rapid_execution_strategy[={MIN_RUNTIME|MIN_MEM_CONSUMPTION}]`
System Variable	`rapid_execution_strategy`
Scope	Session
Dynamic	No
`SET_VAR` Hint Applies	No
Type	Enumeration
Default Value	`MIN_RUNTIME`

Valid Values	`MIN_RUNTIME`
	
	`MIN_MEM_CONSUMPTION`

Specifies the query execution strategy to use. Minimum runtime (`MIN_RUNTIME`) or minimum memory consumption (`MIN_MEM_CONSUMPTION`).

HeatWave optimizes for network usage rather than memory. If you encounter out of memory errors when running a query, try running the query with the `MIN_MEM_CONSUMPTION` strategy by setting by setting `rapid_execution_strategy` prior to executing the query:

```
SET SESSION rapid_execution_strategy = MIN_MEM_CONSUMPTION;
```

See Section 2.15, “Troubleshooting”.

- `rapid_stats_cache_max_entries`

Command-Line Format	`--rapid-stats-cache-max-entries=#`
System Variable	`rapid_stats_cache_max_entries`
Scope	Global
Dynamic	Yes
`SET_VAR` Hint Applies	No
Type	Integer
Default Value	`65536`
Minimum Value	`0`
Maximum Value	`1048576`

The setting for this variable is managed by OCI and cannot be modified directly. Specifies the maximum number of entries in the statistics cache.

The number of entries permitted in the statistics cache by default is 65536, which is enough to store statistics for 4000 to 5000 unique queries of medium complexity.

For more information, see Section 2.3.4, “Auto Query Plan Improvement”.

- `show_create_table_skip_secondary_engine`

Command-Line Format	`--show-create-table-skip-secondary-engine[={OFF|ON}]`
System Variable	`show_create_table_skip_secondary_engine`
Scope	Session
Dynamic	Yes
`SET_VAR` Hint Applies	Yes
Type	Boolean
Default Value	`OFF`

Whether to exclude the `SECONDARY ENGINE` clause from `SHOW CREATE TABLE` output, and from `CREATE TABLE` statements dumped by the `mysqldump` utility.

`mysqldump` provides the `--show-create-skip-secondary-engine` option. When specified, it enables the `show_create_table_skip_secondary_engine` system variable for the duration of the dump operation.

- `use_secondary_engine`

System Variable	`use_secondary_engine`
Scope	Session
Dynamic	Yes
`SET_VAR` Hint Applies	Yes
Type	Enumeration
Default Value	`ON`

Valid Values	`OFF`
	`ON`
	`FORCED`

Whether to execute queries using the secondary engine. These values are permitted:

- `OFF`: Queries execute using the primary storage (InnoDB) on the MySQL DB System. Execution using the secondary engine (RAPID) is disabled.
- `ON`: Queries execute using the secondary engine (RAPID) when conditions warrant, falling back to the primary storage engine (InnoDB) otherwise. In the case of fallback to the primary engine, whenever that occurs during statement processing, the attempt to use the secondary engine is abandoned and execution is attempted using the primary engine.
- `FORCED`: Queries always execute using the secondary engine (RAPID) or fail if that is not possible. Under this mode, a query returns an error if it cannot be executed using the secondary engine, regardless of whether the tables that are accessed have a secondary engine defined.

6.2 Status Variables

Several status variables provide operational information about HeatWave. You can retrieve status data using `SHOW STATUS` syntax. For example:

```
mysql> SHOW STATUS LIKE 'rapid%';
+---------------------------------+------------+
| Variable_name                   | Value      |
+---------------------------------+------------+
| hw_data_scanned                 | 0          |
| rapid_change_propagation_status | ON         |
| rapid_cluster_status            | ON         |
| rapid_core_count                | 64         |
| rapid_heap_usage                | 58720397   |
| rapid_load_progress             | 100.000000 |
| rapid_ml_operation_count        | 2          |
| rapid_ml_status                 | ON         |
| rapid_plugin_bootstrapped       | YES        |
| rapid_preload_stats_status      | Available  |
| rapid_query_offload_count       | 46         |
| rapid_service_status            | ONLINE     |
+---------------------------------+------------+
```

- `hw_data_scanned`

  Tracks the amount of data scanned by successfully executed HeatWave queries. Data is tracked in megabytes and is a cumulative total of data scanned since the HeatWave Cluster was last started. The counter is reset to 0 when the HeatWave Cluster is restarted (when the `rapid_bootstrap` state changes from `OFF` or `IDLE` to `ON`.)

- `rapid_change_propagation_status`

  The change propagation status.

  A status of `ON` indicates that change propagation is enabled globally, permitting changes to `InnoDB` tables on the MySQL DB System to be propagated to their counterpart tables in the HeatWave Cluster.

- `rapid_cluster_status`

  The HeatWave Cluster status.

- `rapid_core_count`

  The HeatWave node core count. The value remains at 0 until all HeatWave nodes are started.

- `rapid_heap_usage`

  MySQL DB System node heap usage.

- `rapid_load_progress`

  A percentage value indicating the status of a table load operation.

- `rapid_ml_operation_count`

  A count of the number of HeatWave AutoML operations that a dbsystem executes. An operation typically includes multiple queries, and it increases for both successful and failed queries. It resets when a HeatWave node restarts.

- `rapid_ml_status`

  The status of HeatWave AutoML. Possible values are `ON` and `OFF`.

- `rapid_plugin_bootstrapped`

  The bootstrap mode.

- `rapid_preload_stats_status`

  Reports the state of preload statistics collection. Column-level statistics are collected for tables on the MySQL DB System during a cluster size estimate. You can perform a cluster estimate when adding or editing a HeatWave Cluster. States include `Not started`, `In progress`, and `Statistics collected`.

  Preload statistics are cached in the `rpd_preload_stats` Performance Schema table. See Section 7.3.6, “The rpd_preload_stats Table”.

- `rapid_query_offload_count`

  The number of queries offloaded to HeatWave for processing.

- `rapid_service_status`

  Reports the status of the cluster as it is brought back online after a node failure.

- `Secondary_engine_execution_count`

  The number of queries executed by HeatWave. Execution occurs if query processing using the secondary engine advances past the preparation and optimization stages. The variable is incremented regardless of whether query execution is successful.

- `rapid_n_external_tables`

  returns the following for two external tables loaded:

```
+-------------------------+-------+
| Variable_name           | Value |
+-------------------------+-------+
| rapid_n_external_tables | 2     |
+-------------------------+-------+
```

- `rapid_lakehouse_total_loaded_bytes`

  Total bytes of all external tables loaded in Heatwave.

- `rapid_lakehouse_health`

  - `OFFLINE`: Indicates an issue with the RAPID storage engine.
  - `RESOURCEPRINCIPALDOWN`: The resource principal token could not be retrieved.
  - `RESOURCEPRINCIPALNOTSET`: The resource principal endpoint was not set.
  - `LAKEHOUSEDISABLED`: Lakehouse is not currently enabled.
  - `ONLINE`: Lakehouse is enabled and running.

Chapter 7 HeatWave Performance and Monitoring

Table of Contents

The MySQL Performance Schema collects statistics on the usage of HeatWave. Use SQL queries to access this data and check the system status and performance.

The Auto Shape Prediction feature in HeatWave Autopilot uses MySQL statistics for the workload to assess the suitability of the current shape. Auto Shape Prediction provides prompts to upsize the shape and improve system performance, or to downsize the shape if the system is under-utilized. This feature is available for HeatWave on AWS only.

Monitor HeatWave from the HeatWave Console:

- For HeatWave on OCI, the HeatWave Console uses Metrics. See Metrics in the *HeatWave on OCI Service Guide*.
- HeatWave on AWS users can monitor HeatWave on the **Performance** page in the HeatWave Console.
- For HeatWave for Azure, select **Metrics** on the details page for the HeatWave Cluster to access Microsoft Azure Application Insights.

7.1 HeatWave MySQL Monitoring

This section provides queries that you can use to monitor HeatWave.

7.1.1 HeatWave Node Status Monitoring

To view the status of each HeatWave node:

```
mysql> SELECT ID, STATUS
          FROM performance_schema.rpd_nodes;
+----+---------------+
```

```
| ID | STATUS        |
+----+---------------+
|  0 | AVAIL_RNSTATE |
|  1 | AVAIL_RNSTATE |
+----+---------------+
```

For column descriptions, see Section 7.3.5, “The rpd_nodes Table”.

7.1.2 HeatWave Memory Usage Monitoring

To view the memory usage for each HeatWave node:

```
mysql> SELECT ID, MEMORY_USAGE, MEMORY_TOTAL, BASEREL_MEMORY_USAGE
          FROM performance_schema.rpd_nodes;
+----+--------------+--------------+----------------------+
| ID | MEMORY_USAGE | MEMORY_TOTAL | BASEREL_MEMORY_USAGE |
+----+--------------+--------------+----------------------+
|  0 |    115760258 | 515396075520 |            115760152 |
|  1 |    115845086 | 515396075520 |            115844980 |
+----+--------------+--------------+----------------------+
```

For column descriptions, see Section 7.3.5, “The rpd_nodes Table”.

7.1.3 Data Load Progress and Status Monitoring

- The time required to load a table into HeatWave depends on data size. You can monitor load progress by issuing the following query, which returns a percentage value indicating load progress.

  ```
  mysql> SELECT VARIABLE_VALUE
            FROM performance_schema.global_status
            WHERE VARIABLE_NAME = 'rapid_load_progress';
  +----------------+
  | VARIABLE_VALUE |
  +----------------+
  | 100.000000     |
  +----------------+
  ```

- To check the load status of tables in a particular schema:

  ```
  mysql> USE performance_schema;
  mysql> SELECT NAME, LOAD_STATUS
            FROM rpd_tables,rpd_table_id
            WHERE rpd_tables.ID = rpd_table_id.ID AND SCHEMA_NAME LIKE 'tpch';
  +-----------------------------+---------------------+
  | NAME                        | LOAD_STATUS         |
  +-----------------------------+---------------------+
  | tpch.supplier               | AVAIL_RPDGSTABSTATE |
  | tpch.partsupp               | AVAIL_RPDGSTABSTATE |
  | tpch.orders                 | AVAIL_RPDGSTABSTATE |
  | tpch.lineitem               | AVAIL_RPDGSTABSTATE |
  | tpch.customer               | AVAIL_RPDGSTABSTATE |
  | tpch.nation                 | AVAIL_RPDGSTABSTATE |
  | tpch.region                 | AVAIL_RPDGSTABSTATE |
  | tpch.part                   | AVAIL_RPDGSTABSTATE |
  +-----------------------------+---------------------+
  ```

  For information about load statuses, see Section 7.3.9, “The rpd_tables Table”.

- To view the amount of data loaded in HeatWave for a table, in bytes:

  ```
  mysql> USE performance_schema;
  mysql> SELECT rpd_table_id.TABLE_NAME, rpd_tables.NROWS,
            rpd_tables.LOAD_STATUS, rpd_tables.SIZE_BYTES
            FROM rpd_table_id, rpd_tables
  ```

```
          WHERE rpd_table_id.ID = rpd_tables.ID
          ORDER BY SIZE_BYTES;
-----------+---------+--------------------+------------+
| TABLE_NAME | NROWS   | LOAD_STATUS        | SIZE_BYTES |
+------------+---------+--------------------+------------+
| region     |       5 | AVAIL_RPDGSTABSTATE |    4194304 |
| nation     |      25 | AVAIL_RPDGSTABSTATE |    8388608 |
| part       |  200000 | AVAIL_RPDGSTABSTATE |   33554432 |
| customer   |  150000 | AVAIL_RPDGSTABSTATE |   41943040 |
| orders     | 1500000 | AVAIL_RPDGSTABSTATE |  226492416 |
+------------+---------+--------------------+------------+
```

- To view the time that a table load operation completed:

```
mysql> USE performance_schema;
mysql> SELECT rpd_table_id.TABLE_NAME, rpd_tables.NROWS,
          rpd_tables.LOAD_STATUS, rpd_tables.LOAD_END_TIMESTAMP
          FROM rpd_table_id, rpd_tables
          WHERE rpd_table_id.ID = rpd_tables.ID;
+------------+---------+--------------------+----------------------------+
| TABLE_NAME | NROWS   | LOAD_STATUS        | LOAD_END_TIMESTAMP         |
+------------+---------+--------------------+----------------------------+
| region     |       5 | AVAIL_RPDGSTABSTATE | 2021-12-06 14:32:15.209825 |
| part       |  200000 | AVAIL_RPDGSTABSTATE | 2021-12-06 14:32:07.594575 |
| customer   |  150000 | AVAIL_RPDGSTABSTATE | 2021-12-06 14:31:57.210649 |
| nation     |      25 | AVAIL_RPDGSTABSTATE | 2021-12-06 14:17:53.472208 |
| orders     | 1500000 | AVAIL_RPDGSTABSTATE | 2021-12-06 14:24:45.809931 |
+------------+---------+--------------------+----------------------------+
```

- To view the time that the last successful recovery took across all tables.

```
mysql> SELECT variable_value
          FROM performance_schema.global_status
          WHERE variable_name="rapid_recovery_time";
+----------------+
| variable_value |
+----------------+
| N/A            |
+----------------+
```

7.1.4 Change Propagation Monitoring

- To determine if change propagation is enabled globally, query the `rapid_change_propagation_status` variable:

```
mysql> SELECT VARIABLE_VALUE FROM performance_schema.global_status
          WHERE VARIABLE_NAME = 'rapid_change_propagation_status';
+----------------+
| VARIABLE_VALUE |
+----------------+
| ON             |
+----------------+
```

- To determine if change propagation is enabled for a particular table, query the `POOL_TYPE` data from the HeatWave Performance Schema tables. `RAPID_LOAD_POOL_TRANSACTIONAL` indicates that change propagation is enabled for the table. `RAPID_LOAD_POOL_SNAPSHOT` indicates that change propagation is disabled.

```
mysql> USE performance_schema;
mysql> SELECT NAME, POOL_TYPE FROM rpd_tables,rpd_table_id
          WHERE rpd_tables.ID = rpd_table_id.ID AND SCHEMA_NAME LIKE 'tpch';
+---------------+-------------------------------+
| NAME          | POOL_TYPE                     |
+---------------+-------------------------------+
| tpch.orders   | RAPID_LOAD_POOL_TRANSACTIONAL |
```

```
| tpch.region   | RAPID_LOAD_POOL_TRANSACTIONAL |
| tpch.lineitem | RAPID_LOAD_POOL_TRANSACTIONAL |
| tpch.supplier | RAPID_LOAD_POOL_TRANSACTIONAL |
| tpch.partsupp | RAPID_LOAD_POOL_TRANSACTIONAL |
| tpch.part     | RAPID_LOAD_POOL_TRANSACTIONAL |
| tpch.customer | RAPID_LOAD_POOL_TRANSACTIONAL |
+---------------+-------------------------------+
```

7.1.5 Query Execution Monitoring

- To view the number of queries offloaded to the HeatWave Cluster for execution since the last time HeatWave Cluster was started:

```
mysql> SELECT VARIABLE_VALUE
          FROM performance_schema.global_status
          WHERE VARIABLE_NAME = 'rapid_query_offload_count';
+----------------+
| VARIABLE_VALUE |
+----------------+
| 62             |
+----------------+
```

- The Performance Schema statement event tables (see Performance Schema Statement Event Tables), and the `performance_schema.threads` and `performance_schema.processlist` tables include an `EXECUTION_ENGINE` column that indicates whether a query was processed on the `PRIMARY` or `SECONDARY` engine, where the primary engine is InnoDB and the secondary engine is HeatWave. The `sys.processlist` and `sys.x$processlist` views in the MySQL `sys` Schema also include an `execution_engine` column.

  This query shows the schema, the first 50 characters of the query, and the execution engine that processed the query:

```
mysql> SELECT CURRENT_SCHEMA, LEFT(DIGEST_TEXT, 50), EXECUTION_ENGINE
          FROM performance_schema.events_statements_history
          WHERE CURRENT_SCHEMA='tpch';
+----------------+----------------------------------------------------+------------------+
| CURRENT_SCHEMA | LEFT(DIGEST_TEXT, 50)                              | EXECUTION_ENGINE |
+----------------+----------------------------------------------------+------------------+
| tpch           | SELECT COUNT(*) FROM tpch.LINEITEM                 | SECONDARY        |
+----------------+----------------------------------------------------+------------------+
```

- The Performance Schema statement summary tables (see Statement Summary Tables) include a `COUNT_SECONDARY` column that indicates the number of times a query was processed on the `SECONDARY` engine (HeatWave).

  This query retrieves the total number of secondary engine execution events from the `events_statements_summary_by_digest` table:

```
mysql> SELECT SUM(COUNT_SECONDARY)
          FROM performance_schema.events_statements_summary_by_digest;
+----------------------+
| SUM(COUNT_SECONDARY) |
+----------------------+
|                   25 |
+----------------------+
```

  This query counts all engine execution events for a particular schema and shows how many occurred on the primary engine (InnoDB) and how many occurred on the secondary engine (HeatWave):

```
mysql> SELECT SUM(COUNT_STAR) AS TOTAL_EXECUTIONS, SUM(COUNT_STAR) - SUM(COUNT_SECONDARY)
          AS PRIMARY_ENGINE, SUM(COUNT_SECONDARY) AS SECONDARY_ENGINE
          FROM performance_schema.events_statements_summary_by_digest
```

```
          WHERE SCHEMA_NAME LIKE 'tpch';
*************************** 1. row ***************************
TOTAL_EXECUTIONS: 25
  PRIMARY_ENGINE: 5
SECONDARY_ENGINE: 20
```

7.1.6 Query History and Statistics Monitoring

- To view the HeatWave query history including query start time, end time, and wait time in the scheduling queue, as discussed in Section 2.3.3, “Auto Scheduling”.

```
mysql> SELECT QUERY_ID,
          CONNECTION_ID,
          QUERY_START,
          QUERY_END,
          QUEUE_WAIT,
          SUBTIME(
              SUBTIME(QUERY_END, SEC_TO_TIME(RPD_EXEC / 1000)),
              SEC_TO_TIME(GET_RESULT / 1000)
          ) AS EXEC_START
      FROM (
              SELECT QUERY_ID,
                  STR_TO_DATE(
                      JSON_UNQUOTE(
                          JSON_EXTRACT(QEXEC_TEXT->>"$**.queryStartTime", '$[0]')
                      ),
                      '%Y-%m-%d %H:%i:%s.%f'
                  ) AS QUERY_START,
                  JSON_EXTRACT(QEXEC_TEXT->>"$**.timeBetweenMakePushedJoinAndRpdExec", '$[0]')
                  AS QUEUE_WAIT,
                  STR_TO_DATE(
                      JSON_UNQUOTE(
                          JSON_EXTRACT(QEXEC_TEXT->>"$**.queryEndTime", '$[0]')
                      ),
                      '%Y-%m-%d %H:%i:%s.%f'
                  ) AS QUERY_END,
                  JSON_EXTRACT(QEXEC_TEXT->>"$**.rpdExec.msec", '$[0]') AS RPD_EXEC,
                  JSON_EXTRACT(QEXEC_TEXT->>"$**.getResults.msec", '$[0]') AS GET_RESULT,
                  JSON_EXTRACT(QEXEC_TEXT->>"$**.thread", '$[0]') AS CONNECTION_ID
              FROM performance_schema.rpd_query_stats
          ) tmp;
```

The query returns the following data:

- `QUERY_ID`

  The ID assigned to the query by HeatWave. IDs are assigned in first in first out (FIFO) order.

- `CONNECTION_ID`

  The connection ID of the client that issued the query.

- `QUERY_START`

  The time the query was issued.

- `QUERY_END`

  The time the query finished executing.

- `QUEUE_WAIT`

  The amount of time the query waited in the scheduling queue.

- EXEC_START

  The time that HeatWave started executing the query.

- To view the number of records in the rpd_query_stats table. The rpd_query_stats table stores query compilation and execution statistics (the query history) for the last 1000 successfully executed queries.

```
mysql> SELECT COUNT(*) FROM performance_schema.rpd_query_stats;
+----------+
| count(*) |
+----------+
|     1000 |
+----------+
```

- To view query IDs for the first and last successfully executed queries:

```
mysql> SELECT MIN(QUERY_ID), MAX(QUERY_ID) FROM performance_schema.rpd_query_stats;
+---------------+---------------+
| MIN(QUERY_ID) | MAX(QUERY_ID) |
+---------------+---------------+
|             2 |          1001 |
+---------------+---------------+
```

- To view the query count for a table and the last time the table was queried:

```
mysql> USE performance_schema;
mysql> SELECT rpd_table_id.TABLE_NAME, rpd_tables.NROWS, rpd_tables.QUERY_COUNT,
          rpd_tables.LAST_QUERIED FROM rpd_table_id, rpd_tables
          WHERE rpd_table_id.ID = rpd_tables.ID;
+------------+---------+-------------+----------------------------+
| TABLE_NAME | NROWS   | QUERY_COUNT | LAST_QUERIED               |
+------------+---------+-------------+----------------------------+
| orders     | 1500000 |           1 | 2021-12-06 14:32:59.868141 |
+------------+---------+-------------+----------------------------+
```

7.1.7 Scanned Data Monitoring

HeatWave tracks the amount of data scanned by all queries and by individual queries.

To view a cumulative total of data scanned (in MBs) by all successfully executed HeatWave queries from the time the HeatWave Cluster was last started, query the hw_data_scanned global status variable. For example:

```
mysql> SHOW GLOBAL STATUS LIKE 'hw_data_scanned';
+-----------------+-------+
| Variable_name   | Value |
+-----------------+-------+
| hw_data_scanned | 66    |
+-----------------+-------+
```

The cumulative total does not include data scanned by failed queries, queries that were not offloaded to HeatWave, or EXPLAIN queries.

The hw_data_scanned value is reset to 0 only when the HeatWave Cluster is restarted.

If a subset of HeatWave nodes go offline, HeatWave retains the cumulative total of scanned data as long as the HeatWave Cluster remains in an active state. When the HeatWave Cluster becomes fully operational and starts processing queries again, HeatWave resumes tracking the amount of data scanned, adding to the cumulative total.

To view the amount of data scanned by an individual HeatWave query or to view an estimate for the amount of data that would be scanned by a query run with `EXPLAIN`, run the query and query the `totalBaseDataScanned` field in the `QKRN_TEXT` column of the `performance_schema.rpd_query_stats` table:

```
mysql> SELECT query_id,
          JSON_EXTRACT(JSON_UNQUOTE(qkrn_text->'$**.sessionId'),'$[0]') AS session_id,
          JSON_EXTRACT(JSON_UNQUOTE(qkrn_text->'$**.totalBaseDataScanned'), '$[0]') AS data_scanned,
          JSON_EXTRACT(JSON_UNQUOTE(qexec_text->'$**.error'),'$[0]') AS error_message
          FROM performance_schema.rpd_query_stats;
+----------+------------+--------------+---------------+
| query_id | session_id | data_scanned | error_message |
+----------+------------+--------------+---------------+
|        1 | 8          | 66           | ""            |
+----------+------------+--------------+---------------+
```

The example above retrieves any error message associated with the query ID. If a query fails or was interrupted, the number of bytes scanned by the failed or interrupted query and the associated error message are returned, as shown in the following examples:

```
mysql> SELECT query_id,
          JSON_EXTRACT(JSON_UNQUOTE(qkrn_text->'$**.sessionId'),'$[0]') AS session_id,
          JSON_EXTRACT(JSON_UNQUOTE(qkrn_text->'$**.totalBaseDataScanned'), '$[0]') AS data_scanned,
          JSON_EXTRACT(JSON_UNQUOTE(qexec_text->'$**.error'),'$[0]') AS error_message
          FROM performance_schema.rpd_query_stats;
+----------+------------+--------------+-----------------------------------------+
| query_id | session_id | data_scanned | error_message                           |
+----------+------------+--------------+-----------------------------------------+
|        1 | 8          | 461          | "Operation was interrupted by the user." |
+----------+------------+--------------+-----------------------------------------+
```

```
mysql> SELECT query_id,
          JSON_EXTRACT(JSON_UNQUOTE(qkrn_text->'$**.sessionId'),'$[0]') AS session_id,
          JSON_EXTRACT(JSON_UNQUOTE(qkrn_text->'$**.totalBaseDataScanned'), '$[0]') AS data_scanned,
          JSON_EXTRACT(JSON_UNQUOTE(qexec_text->'$**.error'),'$[0]') AS error_message
          FROM performance_schema.rpd_query_stats;
+----------+------------+--------------+-----------------------------------------------+
| query_id | session_id | data_scanned | error_message                                 |
+----------+------------+--------------+-----------------------------------------------+
|        1 | 8          | 987          | "Out of memory error during query execution in|
|          |            |              |  RAPID."                                      |
+----------+------------+--------------+-----------------------------------------------+
```

7.2 HeatWave AutoML Monitoring

You can monitor HeatWave AutoML status by querying the `rapid_ml_status` variable or by querying the `ML_STATUS` column of the `performance_schema.rpd_nodes` table.

- Querying the `rapid_ml_status` variable:

  The `rapid_ml_status` variable provides the status of HeatWave AutoML. Possible values are `ON` and `OFF`.

  - `ON`: HeatWave AutoML is up and running.
  - `OFF`: HeatWave AutoML is down.

  You can query the `rapid_ml_status` status variable directly or through the `performance_schema.global_status` table; for example:

  ```
  mysql> SHOW GLOBAL STATUS LIKE 'rapid_ml_status';
  +-----------------+-------+
  ```

```
| Variable_name   | Value |
+-----------------+-------+
| rapid_ml_status | ON    |
+-----------------+-------+
```

```
mysql> SELECT VARIABLE_NAME, VARIABLE_VALUE
          FROM performance_schema.global_status
          WHERE VARIABLE_NAME LIKE 'rapid_ml_status';
+-----------------+----------------+
| VARIABLE_NAME   | VARIABLE_VALUE |
+-----------------+----------------+
| rapid_ml_status | ON             |
+-----------------+----------------+
```

- Querying the `ML_STATUS` column of the `performance_schema.rpd_nodes` table.

  The HeatWave plugin writes HeatWave AutoML status information to the `ML_STATUS` column of `performance_schema.rpd_nodes` table after each ML query. Possible values include:

  - `UNAVAIL_MLSTATE`: HeatWave AutoML is not available.
  - `AVAIL_MLSTATE`: HeatWave AutoML is available.
  - `DOWN_MLSTATE`: HeatWave AutoML is down.

  `ML_STATUS` is reported for each HeatWave node.

  To following query retrieves `ID`, `STATUS`, and `ML_STATUS` for each HeatWave node from the `performance_schema.rpd_nodes` table:

```
mysql> SELECT ID, STATUS, ML_STATUS FROM performance_schema.rpd_nodes;
+----+---------------+---------------+
| ID | STATUS        | ML_STATUS     |
+----+---------------+---------------+
|  1 | AVAIL_RNSTATE | AVAIL_MLSTATE |
|  0 | AVAIL_RNSTATE | AVAIL_MLSTATE |
+----+---------------+---------------+
```

If `rapid_ml_status` is `OFF` or `ML_STATUS` reports `DOWN_MLSTATE` for any HeatWave node, you can restart the HeatWave Cluster in the HeatWave Console but be aware that restarting interrupts any analytics queries that are running. See Managing a HeatWave Cluster in the *HeatWave on OCI Service Guide*.

7.3 HeatWave Performance Schema Tables

HeatWave Performance Schema tables provide information about HeatWave nodes, and about tables and columns that are currently loaded in HeatWave.

Information about HeatWave nodes is available only when `rapid_bootstrap` mode is `ON`. Information about tables and columns is available only after tables are loaded in the HeatWave Cluster. See Section 2.2, “Loading Data to HeatWave MySQL”.

7.3.1 The rpd_column_id Table

The `rpd_column_id` table provides information about columns of tables that are loaded in HeatWave.

The `rpd_column_id` table has these columns:

- `ID`

A unique identifier for the column.

- `TABLE_ID`

  The ID of the table to which the column belongs.

- `COLUMN_NAME`

  The column name.

The `rpd_column_id` table is read-only.

7.3.2 The rpd_columns Table

The `rpd_columns` table provides column encoding information for columns of tables loaded in HeatWave.

The `rpd_columns` table has these columns:

- `TABLE_ID`

  A unique identifier for the table.

- `COLUMN_ID`

  A unique identifier for the table column.

- `NDV`

  The number of distinct values in the column.

- `ENCODING`

  The type of encoding used.

- `DATA_PLACEMENT_INDEX`

  The data placement key index ID associated with the column. Index value ranges from 1 to 16. For information about data placement key index values, see Section 2.7.2, “Defining Data Placement Keys”. NULL indicates that the column is not defined as a data placement key. For a `DATA_PLACEMENT_INDEX` query that identifies columns with data placement keys, see Section 2.16, “Metadata Queries”.

- `DICT_SIZE_BTYES`

  The dictionary size per column, in bytes.

The `rpd_columns` table is read-only.

7.3.3 The rpd_exec_stats Table

The `rpd_exec_stats` table stores query execution statistics produced by HeatWave nodes in `JSON` format. One row of execution statistics is stored for each node that participates in the query. The table stores a maximum of 1000 queries.

For HeatWave AutoML routines that include multiple sub-queries, such as `ML_TRAIN`, a new record is used for each query.

The `rpd_exec_stats` table has these columns:

- `QUERY_ID`

  The query ID. The counter auto-increments for each HeatWave MySQL and HeatWave AutoML query.

- `NODE_ID`

  The HeatWave node ID.

- `EXEC_TEXT`

  Query execution statistics. For HeatWave AutoML, this contains the HeatWave AutoML routine that the user runs.

7.3.4 The rpd_ml_stats Table

> **Note**
>
> The Performance Schema table described here is available as of MySQL 9.0.0. Support for HeatWave GenAI routines is available as of MySQL 9.0.1-u1.

The `rpd_ml_stats` table tracks the usage of successful HeatWave routines. These metrics reset whenever the respective DB system restarts.

The following HeatWave AutoML routines are tracked:

- `ML_TRAIN`
- `ML_EXPLAIN`
- `ML_PREDICT_ROW`
- `ML_PREDICT_TABLE`
- `ML_EXPLAIN_ROW`
- `ML_EXPLAIN_TABLE`

The following HeatWave GenAI routines are tracked:

- `ML_GENERATE`
- `ML_EMBED_ROW`

The `rpd_ml_stats` table has these columns:

- `STATUS_NAME`

  Identifies the type of meter tracking usage.

- `STATUS_VALUE`

  Displays metrics for metering. Content is displayed in JSON format.

Metrics in the table are entries as JSON values. The following metrics are used:

- `n_cells`

The total number of table cells processed by the HeatWave AutoML routine for all invocations.

- `n_cells_user_excluded`

  The total number of table cells manually excluded for the HeatWave AutoML routine.

- `n_blob_cells`

  The total number of table BLOB cells processed by the HeatWave AutoML routine for all invocations.

- `table_size_bytes`

  The total number of bytes of data processed by the HeatWave AutoML routine for all invocations.

- `blob_size_bytes`

  The total number of bytes of BLOB/TEXT data processed by the HeatWave AutoML routine for all invocations.

- `model_size_bytes`

  The total number of bytes of data for the HeatWave AutoML model that is trained. This includes any explainer models. This metric only applies to the `ML_TRAIN` and `ML_EXPLAIN` HeatWave AutoML routines. All other routines will display NULL values.

- `input_size_bytes`

  The cumulative size in bytes of all input string/document invocations ingested by the HeatWave GenAI routine.

- `context_size_bytes`

  The size in bytes of the context string referenced when generating the response. This metric only applies to the `ML_GENERATE` HeatWave GenAI routine since the `ML_EMBED_ROW` routine does not have context. The metric will still appear for `ML_EMBED_ROW`, but will display a value of 0.

- `output_size_bytes`

  The cumulative size in bytes of responses generated by all invocations for the HeatWave GenAI routine.

- `n_invocations`

  The total number of times the routine has been successfully invoked on the HeatWave cluster.

- `last_updated_timestamp`

  The POSIX timestamp of the last call.

7.3.5 The rpd_nodes Table

The `rpd_nodes` table provides information about HeatWave nodes.

The `rpd_nodes` table has these columns:

- `ID`

  A unique identifier for the HeatWave node.

- `CORES`

  The number of cores used by the HeatWave node.

- `MEMORY_USAGE`

  Node memory usage in bytes. The value is refreshed every four seconds. If a query starts and finishes in the four seconds between refreshes, the memory used by the query is not accounted for in the reported value.

- `MEMORY_TOTAL`

  The total memory in bytes allocated to the HeatWave node.

- `BASEREL_MEMORY_USAGE`

  The base relation memory footprint per node.

- `STATUS`

  The status of the HeatWave node. Possible statuses include:

  - `NOTAVAIL_RNSTATE`

    Not available.

  - `AVAIL_RNSTATE`

    Available.

  - `DOWN_RNSTATE`

    Down.

  - `SPARE_RNSTATE`

    Spare.

  - `DEAD_RNSTATE`

    The node is not operational.

- `IP`

  IP address of the HeatWave node.

- `PORT`

  The port on which the HeatWave node was started.

- `CLUSTER_EVENT_NUM`

  The number of cluster events such as node down, node up, and so on.

- `NUM_OBJSTORE_GETS`

  Number of `GET` requests from the node to the object store.

- `NUM_OBJSTORE_PUTS`

The number of PUT requests from the node to the object store.

- NUM_OBJSTORE_DELETES

  The number of DELETE requests from the node to the object store.

- ML_STATUS

  HeatWave AutoML status. Possible status values include:

  - UNAVAIL_MLSTATE: HeatWave AutoML is not available.
  - AVAIL_MLSTATE: HeatWave AutoML is available.
  - DOWN_MLSTATE: HeatWave AutoML declares the node is down.

The rpd_nodes table is read-only.

The rpd_nodes table may not show the current status for a new node or newly configured node immediately. The rpd_nodes table is updated after the node has successfully joined the cluster.

If additional nodes fail while node recovery is in progress, the newly failed nodes are not detected and their status is not updated in the performance_schema.rpd_nodes table until after the current recovery operation finishes and the nodes that failed previously have rejoined the cluster.

7.3.6 The rpd_preload_stats Table

The rpd_preload_stats table stores column level statistics collected from InnoDB tables. The statistics are used to estimate the number of HeatWave nodes required for a given dataset before loading the tables into the HeatWave Cluster.

The rpd_preload_stats table has these columns:

- TABLE_SCHEMA

  The schema name.

- TABLE_NAME

  The table name.

- COLUMN_NAME

  The column name.

- AVG_BYTE_WIDTH_INC_NULL

  The average byte width of the column. The average value includes NULL values.

The rpd_preload_stats table has the following characteristics:

- It is read-only. DDL, including TRUNCATE TABLE, is not permitted.
- It is truncated before new statistics are added.
- It cannot be dropped, truncated, or modified by DML statements directly.
- Statistics are not persisted when the server is shut down.

- It has a limit of 65536 rows.
- Statistics are approximate, based on an adaptive sample scan.
- Statistics are deterministic, provided that the data does not change.

7.3.7 The rpd_query_stats Table

The `rpd_query_stats` table stores query compilation and execution statistics produced by the HeatWave plugin in `JSON` format. One row of data is stored for each query. The table stores data for the last 1000 executed queries. Data is stored for successfully processed queries and failed queries.

For HeatWave AutoML routines that include multiple sub-queries, such as `ML_TRAIN`, a new record is used for each query.

The `rpd_query_stats` table has these columns:

- `CONNECTION_ID`

  The ID of the connection.

- `QUERY_ID`

  The query ID. The counter auto-increments for each HeatWave MySQL and HeatWave AutoML query.

- `QUERY_TEXT`

  The `RAPID` engine query or HeatWave AutoML query run by the user.

- `QEXEC_TEXT`

  Query execution log. For HeatWave AutoML, this contains the arguments the user passed to the HeatWave AutoML routine call.

- `QKRN_TEXT`

  Logical query execution plan. This field is not used for HeatWave AutoML queries.

- `QEP_TEXT`

  Physical query execution plan. This field is not used for HeatWave AutoML queries.

  Includes `prepart` data, which can be queried to determine if a `JOIN` or `GROUP BY` query used data placement partitions. See Section 2.16, “Metadata Queries”.

- `STATEMENT_ID`

  The global query ID.

7.3.8 The rpd_table_id Table

The `rpd_table_id` table provides the ID, name, and schema of the tables loaded in HeatWave.

The `rpd_table_id` table has these columns:

- `ID`

  A unique identifier for the table.

- NAME

  The full table name including the schema.

- SCHEMA_NAME

  The schema name.

- TABLE_NAME

  The table name.

The rpd_table_id table is read-only.

7.3.9 The rpd_tables Table

The rpd_tables table provides the system change number (SCN) and load pool type for tables loaded in HeatWave.

The rpd_tables table has these columns:

- ID

  A unique identifier for the table.

- SNAPSHOT_SCN

  The system change number (SCN) of the table snapshot. The SCN is an internal number that represents a point in time according to the system logical clock that the table snapshot was transactionally consistent with the source table.

- PERSISTED_SCN

  The SCN up to which changes are persisted.

- POOL_TYPE

  The load pool type of the table. Possible values are RAPID_LOAD_POOL_SNAPSHOT and RAPID_LOAD_POOL_TRANSACTIONAL.

- DATA_PLACEMENT_TYPE

  The data placement type.

- NROWS

  The number of rows that are loaded for the table. The value is set initially when the table is loaded, and updated as changes are propagated.

- LOAD_STATUS

  The load status of the table. Statuses include:

  - NOLOAD_RPDGSTABSTATE

    The table is not yet loaded.

  - LOADING_RPDGSTABSTATE

The table is being loaded.

- `AVAIL_RPDGSTABSTATE`

  The table is loaded and available for queries.

- `UNLOADING_RPDGSTABSTATE`

  The table is being unloaded.

- `INRECOVERY_RPDGSTABSTATE`

  The table is being recovered. After completion of the recovery operation, the table is placed back in the `UNAVAIL_RPDGSTABSTATE` state if there are pending recoveries.

- `STALE_RPDGSTABSTATE`

  A failure during change propagation, and the table has become stale. See Section 2.2.7, “Change Propagation”

- `UNAVAIL_RPDGSTABSTATE`

  The table is unavailable.

- `LOAD_PROGRESS`

  The load progress of the table expressed as a percentage value.

  For Lakehouse, the following values are returned:

  - 10%: the initialization phase is complete.
  - 10-70%: the transformation to native HeatWave format is in progress.
  - 70% - 80%: the transformation to native HeatWave format is complete and the aggregation phase is in progress.
  - 80-99%: the recovery phase is in progress.
  - 100%: data load is complete.

- `SIZE_BYTES`

  The amount of data loaded for the table, in bytes.

- `TRANSFORMATION_BYTES`:

  The total size of raw Lakehouse data transformed, in bytes.

- `NROWS`:

  The number of rows loaded to the external table.

- `QUERY_COUNT`

  The number of queries that referenced the table.

- `LAST_QUERIED`

The timestamp of the last query that referenced the table.

- `LOAD_START_TIMESTAMP`

  The load start timestamp for the table.

- `LOAD_END_TIMESTAMP`

  The load completion timestamp for the table.

- `RECOVERY_SOURCE`

  Indicates the source of the last successful recovery for a table. The values are `MySQL`, that is InnoDB, or `ObjectStorage`.

- `RECOVERY_START_TIMESTAMP`

  The timestamp when the latest successful recovery started.

- `RECOVERY_END_TIMESTAMP`

  The timestamp when the latest successful recovery ended.

The `rpd_tables` table is read-only.

Chapter 8 HeatWave Quickstarts

Table of Contents

8.1 HeatWave Quickstart Prerequisites

- An operational MySQL DB System.
  - For HeatWave on OCI, see Creating a DB System in the *HeatWave on OCI Service Guide*.
  - For HeatWave on AWS, see Creating a DB System in the *HeatWave on AWS Service Guide*.
  - For HeatWave for Azure, see Provisioning HeatWave in the *HeatWave for Azure Service Guide*.
- An operational HeatWave Cluster.
  - For HeatWave on OCI, see Adding a HeatWave Cluster in the *HeatWave on OCI Service Guide*.
  - For HeatWave on AWS, see Creating a HeatWave Cluster in the *HeatWave on AWS Service Guide*.
  - For HeatWave for Azure, see Provisioning HeatWave Nodes in the *HeatWave for Azure Service Guide*.
- MySQL Shell 8.0.22 or higher.
  - For HeatWave on OCI, see Connecting to a DB System in the *HeatWave on OCI Service Guide*.
  - For HeatWave on AWS, see Connecting with MySQL Shell in the *HeatWave on AWS Service Guide*.

8.2 tpch Analytics Quickstart

The *tpch Analytics Quickstart* shows how to import data into the MySQL DB System using the MySQL Shell Parallel Table Import Utility, manually loading data into HeatWave, and running queries.

This quickstart contains the following sections:

- tpch Prerequisites
- Generating tpch Sample Data
- Creating the tpch Sample Database and Importing Data
- Loading tpch Data Into HeatWave MySQL
- Running tpch Queries
- Additional tpch Queries
- Unloading tpch Tables

tpch Prerequisites

Review the HeatWave Quickstart Requirements.

Generating tpch Sample Data

Examples in this Quickstart use the `tpch` sample database, which is an ad-hoc decision support database derived from the TPC Benchmark™ H (TPC-H) specification. For an overview of the `tpch` schema, refer to the *Logical Database Design* section of the specification document.

The following instructions describe how to generate `tpch` sample data using the `dbgen` utility. The instructions assume you are on a Linux system that has `gcc` and `make` libraries installed.

To generate `tpch` sample data:

1. Download the TPC-H tools zip file from TPC Download Current.
2. Extract the zip file to a location on your system.
3. Change to the `dbgen` directory and make a copy of the makefile template.

```
$> cd 2.18.0/dbgen
$> cp makefile.suite makefile
```

4. Configure the following settings in the makefile:

```
$> CC = gcc
$> DATABASE= ORACLE
$> MACHINE = LINUX
$> WORKLOAD = TPCH
```

5. Run `make` to build the `dbgen` utility:

```
$> make
```

6. Issue the following `dbgen` command to generate a 1GB set of data files for the `tpch` database:

```
$> ./dbgen -s 1
```

The operation may take a few minutes. When finished, the following data files appear in the working directory, one for each table in the `tpch` database:

```
$> ls -1 *.tbl
customer.tbl
lineitem.tbl
nation.tbl
orders.tbl
partsupp.tbl
part.tbl
region.tbl
supplier.tbl
```

Creating the tpch Sample Database and Importing Data

This topic describes how to create the tpch sample database on the MySQL DB System and import the sample data. The sample data must be available on the MySQL DB System before it can be loaded into the HeatWave Cluster.

Sample database creation and import operations are performed using MySQL Shell. The MySQL Shell Parallel Table Import Utility provides fast data import for large data files. The utility analyzes an input data file, divides it into chunks, and uploads the chunks to the target MySQL DB System using parallel

connections. The utility is capable of completing a large data import many times faster than a standard single-threaded upload using a `LOAD DATA` statement. For additional information, see Parallel Table Import Utility.

To create the tpch sample database on the MySQL DB System and import data:

1. Start MySQL Shell and connect to the MySQL DB System endpoint:

```
$> mysqlsh --mysql Username@DBSystem_IP_Address_or_Host_Name
```

   The `--mysql` option opens a `ClassicSession`, which is required when using the MySQL Shell Parallel Table Import Utility.

   MySQL Shell opens in JavaScript execution mode by default.

```
MySQL>JS>
```

2. Change the MySQL Shell execution mode from JavaScript to SQL:

```
MySQL>JS> \sql
```

3. Create the `tpch` sample database and tables:

```
mysql> CREATE DATABASE tpch character set utf8mb4;
mysql> USE tpch;

mysql> CREATE TABLE nation  ( N_NATIONKEY INTEGER primary key,
    N_NAME       CHAR(25) NOT NULL,
    N_REGIONKEY  INTEGER NOT NULL,
    N_COMMENT    VARCHAR(152));

mysql> CREATE TABLE region  ( R_REGIONKEY INTEGER primary key,
    R_NAME       CHAR(25) NOT NULL,
    R_COMMENT    VARCHAR(152));

mysql> CREATE TABLE part  ( P_PARTKEY INTEGER primary key,
    P_NAME        VARCHAR(55) NOT NULL,
    P_MFGR        CHAR(25) NOT NULL,
    P_BRAND       CHAR(10) NOT NULL,
    P_TYPE        VARCHAR(25) NOT NULL,
    P_SIZE        INTEGER NOT NULL,
    P_CONTAINER   CHAR(10) NOT NULL,
    P_RETAILPRICE DECIMAL(15,2) NOT NULL,
    P_COMMENT     VARCHAR(23) NOT NULL );

mysql> CREATE TABLE supplier  ( S_SUPPKEY INTEGER primary key,
    S_NAME        CHAR(25) NOT NULL,
    S_ADDRESS     VARCHAR(40) NOT NULL,
    S_NATIONKEY   INTEGER NOT NULL,
    S_PHONE       CHAR(15) NOT NULL,
    S_ACCTBAL     DECIMAL(15,2) NOT NULL,
    S_COMMENT     VARCHAR(101) NOT NULL);

mysql> CREATE TABLE partsupp  ( PS_PARTKEY INTEGER NOT NULL,
    PS_SUPPKEY     INTEGER NOT NULL,
    PS_AVAILQTY    INTEGER NOT NULL,
    PS_SUPPLYCOST  DECIMAL(15,2)  NOT NULL,
    PS_COMMENT     VARCHAR(199) NOT NULL, primary key (ps_partkey, ps_suppkey) );

mysql> CREATE TABLE customer  ( C_CUSTKEY INTEGER primary key,
    C_NAME        VARCHAR(25) NOT NULL,
    C_ADDRESS     VARCHAR(40) NOT NULL,
    C_NATIONKEY   INTEGER NOT NULL,
    C_PHONE       CHAR(15) NOT NULL,
    C_ACCTBAL     DECIMAL(15,2)   NOT NULL,
```

```
    C_MKTSEGMENT  CHAR(10) NOT NULL,
    C_COMMENT     VARCHAR(117) NOT NULL);

mysql> CREATE TABLE orders  ( O_ORDERKEY INTEGER primary key,
    O_CUSTKEY        INTEGER NOT NULL,
    O_ORDERSTATUS    CHAR(1) NOT NULL,
    O_TOTALPRICE     DECIMAL(15,2) NOT NULL,
    O_ORDERDATE      DATE NOT NULL,
    O_ORDERPRIORITY  CHAR(15) NOT NULL,
    O_CLERK          CHAR(15) NOT NULL,
    O_SHIPPRIORITY   INTEGER NOT NULL,
    O_COMMENT        VARCHAR(79) NOT NULL);

mysql> CREATE TABLE lineitem ( L_ORDERKEY INTEGER NOT NULL,
    L_PARTKEY     INTEGER NOT NULL,
    L_SUPPKEY     INTEGER NOT NULL,
    L_LINENUMBER  INTEGER NOT NULL,
    L_QUANTITY    DECIMAL(15,2) NOT NULL,
    L_EXTENDEDPRICE  DECIMAL(15,2) NOT NULL,
    L_DISCOUNT    DECIMAL(15,2) NOT NULL,
    L_TAX         DECIMAL(15,2) NOT NULL,
    L_RETURNFLAG  CHAR(1) NOT NULL,
    L_LINESTATUS  CHAR(1) NOT NULL,
    L_SHIPDATE    DATE NOT NULL,
    L_COMMITDATE  DATE NOT NULL,
    L_RECEIPTDATE DATE NOT NULL,
    L_SHIPINSTRUCT CHAR(25) NOT NULL,
    L_SHIPMODE     CHAR(10) NOT NULL,
    L_COMMENT      VARCHAR(44) NOT NULL,
    primary key(L_ORDERKEY,L_LINENUMBER));
```

4. Verify that the `tpch` schema and tables were created:

```
MySQL>SQL> SHOW TABLES;
+----------------+
| Tables_in_tpch |
+----------------+
| customer       |
| lineitem       |
| nation         |
| orders         |
| part           |
| partsupp       |
| region         |
| supplier       |
+----------------+
```

5. Change back to JavaScript execution mode to use the Parallel Table Import Utility:

```
MySQL>SQL> \js
```

6. Execute the following operations to import the data into the `tpch` database on the MySQL DB System.

> **Note**
>
> For information about the `util.importTable()` options used in the following commands, see Parallel Table Import Utility. The number of parallel threads specified using the `threads` option depends on the number of CPU cores of the shape. It is assumed that sample data fields are terminated by the pipe "`|`" character.

```
MySQL>JS> util.importTable("nation.tbl",{table: "nation", fieldsTerminatedBy:"|",
bytesPerChunk:"100M", threads:16})

MySQL>JS> util.importTable("region.tbl",{table: "region", fieldsTerminatedBy:"|",
```

```
bytesPerChunk:"100M", threads:16})

MySQL>JS> util.importTable("part.tbl",{table: "part", fieldsTerminatedBy:"|",
bytesPerChunk:"100M", threads:16})

MySQL>JS> util.importTable("supplier.tbl",{table: "supplier", fieldsTerminatedBy:"|" ,
bytesPerChunk:"100M", threads:16})

MySQL>JS> util.importTable("partsupp.tbl",{table: "partsupp", fieldsTerminatedBy:"|",
bytesPerChunk:"100M", threads:16})

MySQL>JS> util.importTable("customer.tbl",{table: "customer", fieldsTerminatedBy:"|",
bytesPerChunk:"100M", threads:16})

MySQL>JS> util.importTable("orders.tbl",{table: "orders", fieldsTerminatedBy:"|",
bytesPerChunk:"100M", threads:16})

MySQL>JS> util.importTable("lineitem.tbl",{table: "lineitem", fieldsTerminatedBy:"|",
bytesPerChunk:"100M", threads:16})
```

Loading tpch Data Into HeatWave MySQL

To load the `tpch` sample data into the HeatWave Cluster:

> **Note**
>
> For HeatWave on AWS, load data into HeatWave using the HeatWave Console. See Manage Data in HeatWave with Workspaces in the *HeatWave on AWS Service Guide*.

1. Start MySQL Shell and connect to the MySQL DB System endpoint:

   ```
   $> mysqlsh Username@DBSystem_IP_Address_or_Host_Name
   ```

   MySQL Shell opens in JavaScript execution mode by default.

   ```
   MySQL>JS>
   ```

2. Change the MySQL Shell execution mode to SQL:

   ```
   MySQL>JS> \sql
   ```

3. Execute the following operations to prepare the `tpch` sample database tables and load them into the HeatWave Cluster. The operations performed include defining string column encodings, defining the secondary engine, and executing `SECONDARY_LOAD` operations.

   ```
   mysql> USE tpch;

   mysql> ALTER TABLE nation modify `N_NAME` CHAR(25) NOT NULL COMMENT 'RAPID_COLUMN=ENCODING=SORTED';
   mysql> ALTER TABLE nation modify `N_COMMENT` VARCHAR(152) COMMENT 'RAPID_COLUMN=ENCODING=SORTED';
   mysql> ALTER TABLE nation SECONDARY_ENGINE=RAPID;
   mysql> ALTER TABLE nation SECONDARY_LOAD;

   mysql> ALTER TABLE region modify `R_NAME` CHAR(25) NOT NULL COMMENT 'RAPID_COLUMN=ENCODING=SORTED';
   mysql> ALTER TABLE region modify `R_COMMENT` VARCHAR(152) COMMENT 'RAPID_COLUMN=ENCODING=SORTED';
   mysql> ALTER TABLE region SECONDARY_ENGINE=RAPID;
   mysql> ALTER TABLE region SECONDARY_LOAD;

   mysql> ALTER TABLE part modify `P_MFGR` CHAR(25) NOT NULL COMMENT 'RAPID_COLUMN=ENCODING=SORTED';
   mysql> ALTER TABLE part modify `P_BRAND` CHAR(10) NOT NULL COMMENT 'RAPID_COLUMN=ENCODING=SORTED';
   mysql> ALTER TABLE part modify `P_CONTAINER` CHAR(10) NOT NULL COMMENT 'RAPID_COLUMN=ENCODING=SORTED';
   mysql> ALTER TABLE part modify `P_COMMENT` VARCHAR(23) NOT NULL COMMENT 'RAPID_COLUMN=ENCODING=SORTED';
   mysql> ALTER TABLE part SECONDARY_ENGINE=RAPID;
   mysql> ALTER TABLE part SECONDARY_LOAD;
   ```

```
mysql> ALTER TABLE supplier modify `S_NAME` CHAR(25) NOT NULL COMMENT 'RAPID_COLUMN=ENCODING=SORTED';
mysql> ALTER TABLE supplier modify `S_ADDRESS` VARCHAR(40) NOT NULL COMMENT 'RAPID_COLUMN=ENCODING=SORTED';
mysql> ALTER TABLE supplier modify `S_PHONE` CHAR(15) NOT NULL COMMENT 'RAPID_COLUMN=ENCODING=SORTED';

mysql> ALTER TABLE supplier SECONDARY_ENGINE=RAPID;
mysql> ALTER TABLE supplier SECONDARY_LOAD;
mysql> ALTER TABLE partsupp modify `PS_COMMENT` VARCHAR(199) NOT NULL COMMENT 'RAPID_COLUMN=ENCODING=SORTED
mysql> ALTER TABLE partsupp SECONDARY_ENGINE=RAPID;
mysql> ALTER TABLE partsupp SECONDARY_LOAD;

mysql> ALTER TABLE customer modify `C_NAME` VARCHAR(25) NOT NULL COMMENT 'RAPID_COLUMN=ENCODING=SORTED';
mysql> ALTER TABLE customer modify `C_ADDRESS` VARCHAR(40) NOT NULL COMMENT 'RAPID_COLUMN=ENCODING=SORTED';
mysql> ALTER TABLE customer modify `C_MKTSEGMENT` CHAR(10) NOT NULL COMMENT 'RAPID_COLUMN=ENCODING=SORTED';
mysql> ALTER TABLE customer modify `C_COMMENT` VARCHAR(117) NOT NULL COMMENT 'RAPID_COLUMN=ENCODING=SORTED'

mysql> ALTER TABLE customer SECONDARY_ENGINE=RAPID;
mysql> ALTER TABLE customer SECONDARY_LOAD;

mysql> ALTER TABLE orders modify `O_ORDERSTATUS` CHAR(1) NOT NULL COMMENT 'RAPID_COLUMN=ENCODING=SORTED';
mysql> ALTER TABLE orders modify `O_ORDERPRIORITY` CHAR(15) NOT NULL COMMENT 'RAPID_COLUMN=ENCODING=SORTED'
mysql> ALTER TABLE orders modify `O_CLERK` CHAR(15) NOT NULL COMMENT 'RAPID_COLUMN=ENCODING=SORTED';
mysql> ALTER TABLE orders SECONDARY_ENGINE=RAPID;
mysql> ALTER TABLE orders SECONDARY_LOAD;

mysql> ALTER TABLE lineitem modify `L_RETURNFLAG` CHAR(1) NOT NULL COMMENT 'RAPID_COLUMN=ENCODING=SORTED';
mysql> ALTER TABLE lineitem modify `L_LINESTATUS` CHAR(1) NOT NULL COMMENT 'RAPID_COLUMN=ENCODING=SORTED';
mysql> ALTER TABLE lineitem modify `L_SHIPINSTRUCT` CHAR(25) NOT NULL COMMENT 'RAPID_COLUMN=ENCODING=SORTED
mysql> ALTER TABLE lineitem modify `L_SHIPMODE` CHAR(10) NOT NULL COMMENT 'RAPID_COLUMN=ENCODING=SORTED';
mysql> ALTER TABLE lineitem modify `L_COMMENT` VARCHAR(44) NOT NULL COMMENT 'RAPID_COLUMN=ENCODING=SORTED';
mysql> ALTER TABLE lineitem SECONDARY_ENGINE=RAPID;
mysql> ALTER TABLE lineitem SECONDARY_LOAD;
```

4. Verify that the `tpch` sample database tables are loaded in the HeatWave Cluster by querying `LOAD_STATUS` data from the HeatWave Performance Schema tables. Loaded tables have an `AVAIL_RPDGSTABSTATE` load status.

```
MySQL>SQL> USE performance_schema;
MySQL>SQL> SELECT NAME, LOAD_STATUS
          FROM rpd_tables,rpd_table_id
          WHERE rpd_tables.ID = rpd_table_id.ID;
+------------------------------+---------------------+
| NAME                         | LOAD_STATUS         |
+------------------------------+---------------------+
| tpch.supplier                | AVAIL_RPDGSTABSTATE |
| tpch.partsupp                | AVAIL_RPDGSTABSTATE |
| tpch.orders                  | AVAIL_RPDGSTABSTATE |
| tpch.lineitem                | AVAIL_RPDGSTABSTATE |
| tpch.customer                | AVAIL_RPDGSTABSTATE |
| tpch.nation                  | AVAIL_RPDGSTABSTATE |
| tpch.region                  | AVAIL_RPDGSTABSTATE |
| tpch.part                    | AVAIL_RPDGSTABSTATE |
+------------------------------+---------------------+
```

Running tpch Queries

This topic describes how to query `tpch` data in the HeatWave Cluster. After tables are loaded into the HeatWave Cluster, queries that qualify are automatically offloaded to the HeatWave Cluster for accelerated processing. To run queries:

> **Note**
>
> For HeatWave on AWS, run queries from the *Query Editor* in the HeatWave Console. See Running Queries in the *HeatWave on AWS Service Guide*.

1. Start MySQL Shell and connect to the MySQL DB System endpoint:

```
$> mysqlsh Username@DBSystem_IP_Address_or_Host_Name
```

   MySQL Shell opens in JavaScript execution mode by default.

```
MySQL>JS>
```

2. Change the MySQL Shell execution mode to SQL:

```
MySQL>JS> \sql
```

3. Change to the `tpch` database:

```
MySQL>SQL> USE tpch;
Default schema set to `tpch`.Fetching table and column names from `tpch` for
auto-completion... Press ^C to stop.
```

4. Before running a query, use `EXPLAIN` to verify that the query can be offloaded to the HeatWave Cluster. For example:

```
MySQL>SQL> EXPLAIN SELECT SUM(l_extendedprice * l_discount) AS revenue
                FROM lineitem
                WHERE l_shipdate >= date '1994-01-01';
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: lineitem
   partitions: NULL
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 56834662
     filtered: 33.33
        Extra: Using where; Using secondary engine RAPID
```

   If the query can be offloaded, the `Extra` column in the `EXPLAIN` output reports `Using secondary engine RAPID`.

5. After verifying that the query can be offloaded, run the query and note the execution time.

```
MySQL>SQL> SELECT SUM(l_extendedprice * l_discount) AS revenue
                FROM lineitem
                WHERE l_shipdate >= date '1994-01-01';
+------------------+
| revenue          |
+------------------+
| 82752894454.9036 |
+------------------+
1 row in set (0.04 sec)
```

6. To compare the HeatWave execution time with MySQL DB System execution time, disable the `use_secondary_engine` variable to see how long it takes to run the same query on the MySQL DB System. For example:

```
MySQL>SQL> SET SESSION use_secondary_engine=OFF;
MySQL>SQL> SELECT SUM(l_extendedprice * l_discount) AS revenue
                FROM lineitem
                WHERE l_shipdate >= date '1994-01-01';
+------------------+
| revenue          |
+------------------+
| 82752894454.9036 |
```

```
+------------------+
1 row in set (24.20 sec)
```

For other `tpch` sample database queries that you can run, see Additional tpch Queries. For more information about running queries, refer to Section 2.3, “Running Queries”.

Additional tpch Queries

This topic provides additional tpch queries that you can run to test the HeatWave Cluster.

- TPCH-Q1: Pricing Summary Report Query

  As described in the TPC Benchmark™ H (TPC-H) specification: "The Pricing Summary Report Query provides a summary pricing report for all lineitems shipped as of a given date. The date is within 60 - 120 days of the greatest ship date contained in the database. The query lists totals for extended price, discounted extended price, discounted extended price plus tax, average quantity, average extended price, and average discount. These aggregates are grouped by `RETURNFLAG` and `LINESTATUS`, and listed in ascending order of `RETURNFLAG` and `LINESTATUS`. A count of the number of lineitems in each group is included."

```
mysql> SELECT
          l_returnflag,
          l_linestatus,
          SUM(l_quantity) AS sum_qty,
          SUM(l_extendedprice) AS sum_base_price,
          SUM(l_extendedprice * (1 - l_discount)) AS sum_disc_price,
          SUM(l_extendedprice * (1 - l_discount) * (1 + l_tax)) AS sum_charge,
          AVG(l_quantity) AS avg_qty,
          AVG(l_extendedprice) AS avg_price,
          AVG(l_discount) AS avg_disc,
          COUNT(*) AS count_order
      FROM
          lineitem
      WHERE
          l_shipdate <= DATE '1998-12-01' - INTERVAL '90' DAY
      GROUP BY l_returnflag , l_linestatus
      ORDER BY l_returnflag , l_linestatus;
```

- TPCH-Q3: Shipping Priority Query

  As described in the TPC Benchmark™ H (TPC-H) specification: "The Shipping Priority Query retrieves the shipping priority and potential revenue, defined as the sum of `l_extendedprice * (1-l_discount)`, of the orders having the largest revenue among those that had not been shipped as of a given date. Orders are listed in decreasing order of revenue. If more than 10 unshipped orders exist, only the 10 orders with the largest revenue are listed."

```
mysql> SELECT
          l_orderkey,
          SUM(l_extendedprice * (1 - l_discount)) AS revenue,
          o_orderdate,
          o_shippriority
      FROM
          customer,
          orders,
          lineitem
      WHERE
          c_mktsegment = 'BUILDING'
              AND c_custkey = o_custkey
              AND l_orderkey = o_orderkey
              AND o_orderdate < DATE '1995-03-15'
              AND l_shipdate > DATE '1995-03-15'
      GROUP BY l_orderkey , o_orderdate , o_shippriority
```

```
      ORDER BY revenue DESC , o_orderdate
      LIMIT 10;
```

- TPCH-Q9: Product Type Profit Measure Query

  As described in the TPC Benchmark™ H (TPC-H) specification: "The Product Type Profit Measure Query finds, for each nation and each year, the profit for all parts ordered in that year that contain a specified substring in their names and that were filled by a supplier in that nation. The profit is defined as the sum of `[(l_extendedprice*(1-l_discount)) - (ps_supplycost * l_quantity)]` for all lineitems describing parts in the specified line. The query lists the nations in ascending alphabetical order and, for each nation, the year and profit in descending order by year (most recent first). "

```
mysql> SELECT
          nation, o_year, SUM(amount) AS sum_profit
      FROM
          (SELECT
              n_name AS nation,
                  YEAR(o_ORDERdate) AS o_year,
                  l_extendedprice * (1 - l_discount) - ps_supplycost * l_quantity AS amount
          FROM
              part
          STRAIGHT_JOIN partsupp
          STRAIGHT_JOIN lineitem
          STRAIGHT_JOIN supplier
          STRAIGHT_JOIN orders
          STRAIGHT_JOIN nation
          WHERE
              s_suppkey = l_suppkey
                  AND ps_suppkey = l_suppkey
                  AND ps_partkey = l_partkey
                  AND p_partkey = l_partkey
                  AND o_ORDERkey = l_ORDERkey
                  AND s_nationkey = n_nationkey
                  AND p_name LIKE '%green%') AS profit
      GROUP BY nation , o_year
      ORDER BY nation , o_year DESC;
```

Unloading tpch Tables

`tpch` tables can be unloaded from HeatWave using the following statements:

> **Note**
>
> For HeatWave on AWS, unload data from HeatWave using the HeatWave Console. See Manage Data in HeatWave with Workspaces in the *HeatWave on AWS Service Guide*.

```
mysql> USE tpch;

mysql> ALTER TABLE customer SECONDARY_UNLOAD;
mysql> ALTER TABLE lineitem SECONDARY_UNLOAD;
mysql> ALTER TABLE nation SECONDARY_UNLOAD;
mysql> ALTER TABLE orders SECONDARY_UNLOAD;
mysql> ALTER TABLE part SECONDARY_UNLOAD;
mysql> ALTER TABLE partsupp SECONDARY_UNLOAD;
mysql> ALTER TABLE region SECONDARY_UNLOAD;
mysql> ALTER TABLE supplier SECONDARY_UNLOAD;
```

8.3 AirportDB Analytics Quickstart

The *HeatWave airportdb Quickstart* shows how to import data into the MySQL DB System using the MySQL Shell Dump Load utility, loading data into HeatWave using Auto Parallel Load, and running queries.

For an online workshop that demonstrates HeatWave using the `airportdb` sample database, see Turbocharge Business Insights with HeatWave Service and HeatWave.

This quickstart contains the following sections:

- AirportDB Prerequisites
- Installing AirportDB
- Loading AirportDB into HeatWave MySQL
- Running AirportDB Queries
- Additional AirportDB Queries
- Unloading AirportDB Tables

AirportDB Prerequisites

Review the HeatWave Quickstart Requirements.

Installing AirportDB

The installation procedure involves downloading the `airportdb` database to a Compute instance and importing the data from the Compute instance into the MySQL DB System using the MySQL Shell Dump Loading utility. For information about this utility, see Dump Loading Utility.

To install the `airportdb` database:

1. Download the `airportdb` sample database and unpack it. The `airportdb` sample database is provided for download as a compressed `tar` or Zip archive. The download is approximately 640 MBs in size.

   ```
   $> wget https://downloads.mysql.com/docs/airport-db.tar.gz
   $> tar xvzf airport-db.tar.gz
   ```

   or

   ```
   $> wget https://downloads.mysql.com/docs/airport-db.zip
   $> unzip airport-db.zip
   ```

   Unpacking the compressed `tar` or Zip archive results in a single directory named `airport-db`, which contains the data files.

2. Start MySQL Shell and connect to the MySQL DB System Endpoint. For additional information about connecting to a MySQL DB System, see Connecting to a DB System in the *HeatWave on OCI Service Guide*.

   ```
   $> mysqlsh Username@DBSystem_IP_Address_or_Host_Name
   ```

3. Load the `airportdb` database into the MySQL DB System using the MySQL Shell Dump Loading Utility.

```
MySQL>JS> util.loadDump("airport-db", {threads: 16, deferTableIndexes: "all",
                ignoreVersion: TRUE})
```

> **Note**
>
> The `deferTableIndexes: "all"` option defers creating secondary indexes until after the table data is loaded, which significantly reduces load times. If you intend to use `airportdb` with HeatWave, which does not use secondary indexes, you can avoid creating secondary indexes by specifying the `loadIndexes: "FALSE"` option instead of `deferTableIndexes: "all"`. For more information about MySQL Dump Load options, see Dump Loading Utility.

After the data is imported into the MySQL DB System, you can load the tables into HeatWave. For instructions, see Loading AirportDB into HeatWave MySQL.

Loading AirportDB into HeatWave MySQL

To load the `airportdb` from the MySQL DB System into HeatWave:

> **Note**
>
> For HeatWave on AWS, load data into HeatWave using the HeatWave Console. See Manage Data in HeatWave with Workspaces in the *HeatWave on AWS Service Guide*.

1. Start MySQL Shell and connect to the MySQL DB System Endpoint.

   ```
   $> mysqlsh Username@DBSystem_IP_Address_or_Host_Name
   ```

2. Change the MySQL Shell execution mode to SQL and run the following Auto Parallel Load command to load the `airportdb` tables into HeatWave.

   ```
   MySQL>JS> \sql
   MySQL>SQL> CALL sys.heatwave_load(JSON_ARRAY('airportdb'), NULL);
   ```

   For information about the Auto Parallel Load utility, see Section 2.2.3, “Loading Data Using Auto Parallel Load”.

Running AirportDB Queries

After airportdb sample database tables are loaded into the HeatWave Cluster, queries that qualify are automatically offloaded to the HeatWave Cluster for accelerated processing. To run queries:

> **Note**
>
> For HeatWave on AWS, run queries from the *Query Editor* in the HeatWave Console. See Running Queries in the *HeatWave on AWS Service Guide*.

1. Start MySQL Shell and connect to the MySQL DB System endpoint:

   ```
   $> mysqlsh Username@DBSystem_IP_Address_or_Host_Name
   ```

   MySQL Shell opens in JavaScript execution mode by default.

   ```
   MySQL>JS>
   ```

2. Change the MySQL Shell execution mode to SQL:

```
MySQL>JS> \sql
```

3. Change to the `airportdb` database.

```
MySQL>SQL> USE airportdb;
Default schema set to `airportdb`.Fetching table and column names from `airportdb` for
auto-completion... Press ^C to stop.
```

4. Before running a query, use `EXPLAIN` to verify that the query can be offloaded to the HeatWave Cluster. For example:

```
MySQL>SQL> EXPLAIN SELECT booking.price, count(*)
                FROM booking
                WHERE booking.price > 500
                GROUP BY booking.price
                ORDER BY booking.price LIMIT
                10;
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: booking
   partitions: NULL
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 54081693
     filtered: 33.32999801635742
        Extra: Using where; Using temporary; Using filesort; Using secondary engine RAPID
```

   If the query can be offloaded, the `Extra` column in the `EXPLAIN` output reports `Using secondary engine RAPID`.

5. After verifying that the query can be offloaded, run the query and note the execution time.

```
MySQL>SQL> SELECT booking.price, count(*)
                FROM booking
                WHERE booking.price > 500
                GROUP BY booking.price
                ORDER BY booking.price
                LIMIT 10;
+--------+----------+
| price  | count(*) |
+--------+----------+
| 500.01 |      860 |
| 500.02 |     1207 |
| 500.03 |     1135 |
| 500.04 |     1010 |
| 500.05 |     1016 |
| 500.06 |     1039 |
| 500.07 |     1002 |
| 500.08 |     1095 |
| 500.09 |     1117 |
| 500.10 |     1106 |
+--------+----------+
10 rows in set (0.0537 sec)
```

6. To compare the HeatWave execution time with MySQL DB System execution time, disable the `use_secondary_engine` variable to see how long it takes to run the same query on the MySQL DB System; for example:

```
MySQL>SQL> SET SESSION use_secondary_engine=OFF;
MySQL>SQL> SELECT booking.price, count(*)
```

```
                FROM booking
                WHERE booking.price > 500
                GROUP BY booking.price
                ORDER BY booking.price
                LIMIT 10;
+--------+----------+
| price  | count(*) |
+--------+----------+
| 500.01 |      860 |
| 500.02 |     1207 |
| 500.03 |     1135 |
| 500.04 |     1010 |
| 500.05 |     1016 |
| 500.06 |     1039 |
| 500.07 |     1002 |
| 500.08 |     1095 |
| 500.09 |     1117 |
| 500.10 |     1106 |
+--------+----------+
10 rows in set (9.3859 sec)
```

For other `airportdb` sample database queries that you can run, see Additional AirportDB Queries. For more information about running queries, see Section 2.3, “Running Queries”.

Additional AirportDB Queries

This topic provides additional airportdb queries that you can run to test the HeatWave Cluster.

- Query 1: Number of Tickets > $500.00, Grouped By Price

```
mysql> SELECT
           booking.price,
           count(*)
       FROM
           booking
       WHERE
           booking.price > 500
       GROUP BY
           booking.price
       ORDER BY
           booking.price
       LIMIT
           10;
```

- Query 2: Average Age of Passengers By Country, Per Airline

```
mysql> SELECT
           airline.airlinename,
           AVG(datediff(departure,birthdate)/365.25) as avg_age,
           count(*) as nb_people
       FROM
           booking, flight, airline, passengerdetails
       WHERE
           booking.flight_id=flight.flight_id AND
           airline.airline_id=flight.airline_id AND
           booking.passenger_id=passengerdetails.passenger_id AND
           country IN ("SWITZERLAND", "FRANCE", "ITALY")
       GROUP BY
           airline.airlinename
       ORDER BY
           airline.airlinename, avg_age
       LIMIT 10;
```

- Query 3: Most Tickets Sales by Airline for Departures from US Airports

```
mysql> SELECT
```

```
        airline.airlinename,
        SUM(booking.price) as price_tickets,
        count(*) as nb_tickets
    FROM
        booking, flight, airline, airport_geo
    WHERE
        booking.flight_id=flight.flight_id AND
        airline.airline_id=flight.airline_id AND
        flight.from=airport_geo.airport_id AND
        airport_geo.country = "UNITED STATES"
    GROUP BY
        airline.airlinename
    ORDER BY
        nb_tickets desc, airline.airlinename
    LIMIT 10;
```

Unloading AirportDB Tables

`airportdb` tables can be unloaded from HeatWave using the following statements:

> **Note**
>
> For HeatWave on AWS, unload data from HeatWave using the HeatWave Console. See Manage Data in HeatWave with Workspaces in the *HeatWave on AWS Service Guide*.

```
mysql> USE airportdb;

mysql> ALTER TABLE booking SECONDARY_UNLOAD;
mysql> ALTER TABLE flight SECONDARY_UNLOAD;
mysql> ALTER TABLE flight_log SECONDARY_UNLOAD;
mysql> ALTER TABLE airport SECONDARY_UNLOAD;
mysql> ALTER TABLE airport_reachable SECONDARY_UNLOAD;
mysql> ALTER TABLE airport_geo SECONDARY_UNLOAD;
mysql> ALTER TABLE airline SECONDARY_UNLOAD;
mysql> ALTER TABLE flightschedule SECONDARY_UNLOAD;
mysql> ALTER TABLE airplane SECONDARY_UNLOAD;
mysql> ALTER TABLE airplane_type SECONDARY_UNLOAD;
mysql> ALTER TABLE employee SECONDARY_UNLOAD;
mysql> ALTER TABLE passenger SECONDARY_UNLOAD;
mysql> ALTER TABLE passengerdetails SECONDARY_UNLOAD;
mysql> ALTER TABLE weatherdata SECONDARY_UNLOAD;
```

8.4 Iris Data Set Machine Learning Quickstart

This tutorial illustrates an end-to-end example of creating and using a predictive machine learning model using HeatWave AutoML. It steps through preparing data, using the `ML_TRAIN` routine to train a model, and using `ML_PREDICT_*` and `ML_EXPLAIN_*` routines to generate predictions and explanations. The tutorial also demonstrates how to assess the quality of a model using the `ML_SCORE` routine, and how to view a model explanation to understand how the model works.

For an online workshop based on this tutorial, see Get started with MySQL HeatWave AutoML.

The tutorial uses the publicly available Iris Data Set from the UCI Machine Learning Repository.

Dua, D. and Graff, C. (2019). UCI Machine Learning Repository [http://archive.ics.uci.edu/ml]. Irvine, CA: University of California, School of Information.

This quickstart contains the following sections:

- Overview
- Before You Begin

Overview

The *Iris Data Set* has the following data, where the sepal and petal features are used to predict the `class` label, which is the type of Iris plant:

- sepal length (cm)
- sepal width (cm)
- petal length (cm)
- petal width (cm)
- class. Possible values include:
  - Iris Setosa
  - Iris Versicolour
  - Iris Virginica

Data is stored in the MySQL database in the following schema and tables:

- `ml_data` schema: The schema containing training and test dataset tables.
- `iris_train` table: The training dataset (labeled). Includes feature columns (sepal length, sepal width, petal length, petal width) and a populated `class` target column with ground truth values.
- `iris_test` table: The test dataset (unlabeled). Includes feature columns (sepal length, sepal width, petal length, petal width) but no target column.
- `iris_validate` table: The validation dataset (labeled). Includes feature columns (sepal length, sepal width, petal length, petal width) and a populated `class` target column with ground truth values.

Before You Begin

Review the HeatWave Quickstart Requirements.

Creating, Training, and Testing Datasets

Create the example schema and tables on the MySQL DB System with the following statements:

```
mysql> CREATE SCHEMA ml_data;

USE ml_data;

CREATE TABLE `iris_train` (
          `sepal length` float DEFAULT NULL,
```

```
          `sepal width` float DEFAULT NULL,
          `petal length` float DEFAULT NULL,
          `petal width` float DEFAULT NULL,
          `class` varchar(16) DEFAULT NULL);

INSERT INTO iris_train VALUES(6.4,2.8,5.6,2.2,'Iris-virginica');
INSERT INTO iris_train VALUES(5.0,2.3,3.3,1.0,'Iris-setosa');
INSERT INTO iris_train VALUES(4.9,2.5,4.5,1.7,'Iris-virginica');
INSERT INTO iris_train VALUES(4.9,3.1,1.5,0.1,'Iris-versicolor');
INSERT INTO iris_train VALUES(5.7,3.8,1.7,0.3,'Iris-versicolor');
INSERT INTO iris_train VALUES(4.4,3.2,1.3,0.2,'Iris-versicolor');
INSERT INTO iris_train VALUES(5.4,3.4,1.5,0.4,'Iris-versicolor');
INSERT INTO iris_train VALUES(6.9,3.1,5.1,2.3,'Iris-virginica');
INSERT INTO iris_train VALUES(6.7,3.1,4.4,1.4,'Iris-setosa');
INSERT INTO iris_train VALUES(5.1,3.7,1.5,0.4,'Iris-versicolor');
INSERT INTO iris_train VALUES(5.2,2.7,3.9,1.4,'Iris-setosa');
INSERT INTO iris_train VALUES(6.9,3.1,4.9,1.5,'Iris-setosa');
INSERT INTO iris_train VALUES(5.8,4.0,1.2,0.2,'Iris-versicolor');
INSERT INTO iris_train VALUES(5.4,3.9,1.7,0.4,'Iris-versicolor');
INSERT INTO iris_train VALUES(7.7,3.8,6.7,2.2,'Iris-virginica');
INSERT INTO iris_train VALUES(6.3,3.3,4.7,1.6,'Iris-setosa');
INSERT INTO iris_train VALUES(6.8,3.2,5.9,2.3,'Iris-virginica');
INSERT INTO iris_train VALUES(7.6,3.0,6.6,2.1,'Iris-virginica');
INSERT INTO iris_train VALUES(6.4,3.2,5.3,2.3,'Iris-virginica');
INSERT INTO iris_train VALUES(5.7,4.4,1.5,0.4,'Iris-versicolor');
INSERT INTO iris_train VALUES(6.7,3.3,5.7,2.1,'Iris-virginica');
INSERT INTO iris_train VALUES(6.4,2.8,5.6,2.1,'Iris-virginica');
INSERT INTO iris_train VALUES(5.4,3.9,1.3,0.4,'Iris-versicolor');
INSERT INTO iris_train VALUES(6.1,2.6,5.6,1.4,'Iris-virginica');
INSERT INTO iris_train VALUES(7.2,3.0,5.8,1.6,'Iris-virginica');
INSERT INTO iris_train VALUES(5.2,3.5,1.5,0.2,'Iris-versicolor');
INSERT INTO iris_train VALUES(5.8,2.6,4.0,1.2,'Iris-setosa');
INSERT INTO iris_train VALUES(5.9,3.0,5.1,1.8,'Iris-virginica');
INSERT INTO iris_train VALUES(5.4,3.0,4.5,1.5,'Iris-setosa');
INSERT INTO iris_train VALUES(6.7,3.0,5.0,1.7,'Iris-setosa');
INSERT INTO iris_train VALUES(6.3,2.3,4.4,1.3,'Iris-setosa');
INSERT INTO iris_train VALUES(5.1,2.5,3.0,1.1,'Iris-setosa');
INSERT INTO iris_train VALUES(6.4,3.2,4.5,1.5,'Iris-setosa');
INSERT INTO iris_train VALUES(6.8,3.0,5.5,2.1,'Iris-virginica');
INSERT INTO iris_train VALUES(6.2,2.8,4.8,1.8,'Iris-virginica');
INSERT INTO iris_train VALUES(6.9,3.2,5.7,2.3,'Iris-virginica');
INSERT INTO iris_train VALUES(6.5,3.2,5.1,2.0,'Iris-virginica');
INSERT INTO iris_train VALUES(5.8,2.8,5.1,2.4,'Iris-virginica');
INSERT INTO iris_train VALUES(5.1,3.8,1.5,0.3,'Iris-versicolor');
INSERT INTO iris_train VALUES(4.8,3.0,1.4,0.3,'Iris-versicolor');
INSERT INTO iris_train VALUES(7.9,3.8,6.4,2.0,'Iris-virginica');
INSERT INTO iris_train VALUES(5.8,2.7,5.1,1.9,'Iris-virginica');
INSERT INTO iris_train VALUES(6.7,3.0,5.2,2.3,'Iris-virginica');
INSERT INTO iris_train VALUES(5.1,3.8,1.9,0.4,'Iris-versicolor');
INSERT INTO iris_train VALUES(4.7,3.2,1.6,0.2,'Iris-versicolor');
INSERT INTO iris_train VALUES(6.0,2.2,5.0,1.5,'Iris-virginica');
INSERT INTO iris_train VALUES(4.8,3.4,1.6,0.2,'Iris-versicolor');
INSERT INTO iris_train VALUES(7.7,2.6,6.9,2.3,'Iris-virginica');
INSERT INTO iris_train VALUES(4.6,3.6,1.0,0.2,'Iris-versicolor');
INSERT INTO iris_train VALUES(7.2,3.2,6.0,1.8,'Iris-virginica');
INSERT INTO iris_train VALUES(5.0,3.3,1.4,0.2,'Iris-versicolor');
INSERT INTO iris_train VALUES(6.6,3.0,4.4,1.4,'Iris-setosa');
INSERT INTO iris_train VALUES(6.1,2.8,4.0,1.3,'Iris-setosa');
INSERT INTO iris_train VALUES(5.0,3.2,1.2,0.2,'Iris-versicolor');
INSERT INTO iris_train VALUES(7.0,3.2,4.7,1.4,'Iris-setosa');
INSERT INTO iris_train VALUES(6.0,3.0,4.8,1.8,'Iris-virginica');
INSERT INTO iris_train VALUES(7.4,2.8,6.1,1.9,'Iris-virginica');
INSERT INTO iris_train VALUES(5.8,2.7,5.1,1.9,'Iris-virginica');
INSERT INTO iris_train VALUES(6.2,3.4,5.4,2.3,'Iris-virginica');
INSERT INTO iris_train VALUES(5.0,2.0,3.5,1.0,'Iris-setosa');
INSERT INTO iris_train VALUES(5.6,2.5,3.9,1.1,'Iris-setosa');
INSERT INTO iris_train VALUES(6.7,3.1,5.6,2.4,'Iris-virginica');
```

```
INSERT INTO iris_train VALUES(6.3,2.5,5.0,1.9,'Iris-virginica');
INSERT INTO iris_train VALUES(6.4,3.1,5.5,1.8,'Iris-virginica');
INSERT INTO iris_train VALUES(6.2,2.2,4.5,1.5,'Iris-setosa');
INSERT INTO iris_train VALUES(7.3,2.9,6.3,1.8,'Iris-virginica');
INSERT INTO iris_train VALUES(4.4,3.0,1.3,0.2,'Iris-versicolor');
INSERT INTO iris_train VALUES(7.2,3.6,6.1,2.5,'Iris-virginica');
INSERT INTO iris_train VALUES(6.5,3.0,5.5,1.8,'Iris-virginica');
INSERT INTO iris_train VALUES(5.0,3.4,1.5,0.2,'Iris-versicolor');
INSERT INTO iris_train VALUES(4.7,3.2,1.3,0.2,'Iris-versicolor');
INSERT INTO iris_train VALUES(6.6,2.9,4.6,1.3,'Iris-setosa');
INSERT INTO iris_train VALUES(5.5,3.5,1.3,0.2,'Iris-versicolor');
INSERT INTO iris_train VALUES(7.7,3.0,6.1,2.3,'Iris-virginica');
INSERT INTO iris_train VALUES(6.1,3.0,4.9,1.8,'Iris-virginica');
INSERT INTO iris_train VALUES(4.9,3.1,1.5,0.1,'Iris-versicolor');
INSERT INTO iris_train VALUES(5.5,2.4,3.8,1.1,'Iris-setosa');
INSERT INTO iris_train VALUES(5.7,2.9,4.2,1.3,'Iris-setosa');
INSERT INTO iris_train VALUES(6.0,2.9,4.5,1.5,'Iris-setosa');
INSERT INTO iris_train VALUES(6.4,2.7,5.3,1.9,'Iris-virginica');
INSERT INTO iris_train VALUES(5.4,3.7,1.5,0.2,'Iris-versicolor');
INSERT INTO iris_train VALUES(6.1,2.9,4.7,1.4,'Iris-setosa');
INSERT INTO iris_train VALUES(6.5,2.8,4.6,1.5,'Iris-setosa');
INSERT INTO iris_train VALUES(5.6,2.7,4.2,1.3,'Iris-setosa');
INSERT INTO iris_train VALUES(6.3,3.4,5.6,2.4,'Iris-virginica');
INSERT INTO iris_train VALUES(4.9,3.1,1.5,0.1,'Iris-versicolor');
INSERT INTO iris_train VALUES(6.8,2.8,4.8,1.4,'Iris-setosa');
INSERT INTO iris_train VALUES(5.7,2.8,4.5,1.3,'Iris-setosa');
INSERT INTO iris_train VALUES(6.0,2.7,5.1,1.6,'Iris-setosa');
INSERT INTO iris_train VALUES(5.0,3.5,1.3,0.3,'Iris-versicolor');
INSERT INTO iris_train VALUES(6.5,3.0,5.2,2.0,'Iris-virginica');
INSERT INTO iris_train VALUES(6.1,2.8,4.7,1.2,'Iris-setosa');
INSERT INTO iris_train VALUES(5.1,3.5,1.4,0.3,'Iris-versicolor');
INSERT INTO iris_train VALUES(4.6,3.1,1.5,0.2,'Iris-versicolor');
INSERT INTO iris_train VALUES(6.5,3.0,5.8,2.2,'Iris-virginica');
INSERT INTO iris_train VALUES(4.6,3.4,1.4,0.3,'Iris-versicolor');
INSERT INTO iris_train VALUES(4.6,3.2,1.4,0.2,'Iris-versicolor');
INSERT INTO iris_train VALUES(7.7,2.8,6.7,2.0,'Iris-virginica');
INSERT INTO iris_train VALUES(5.9,3.2,4.8,1.8,'Iris-setosa');
INSERT INTO iris_train VALUES(5.1,3.8,1.6,0.2,'Iris-versicolor');
INSERT INTO iris_train VALUES(4.9,3.0,1.4,0.2,'Iris-versicolor');
INSERT INTO iris_train VALUES(4.9,2.4,3.3,1.0,'Iris-setosa');
INSERT INTO iris_train VALUES(4.5,2.3,1.3,0.3,'Iris-versicolor');
INSERT INTO iris_train VALUES(5.8,2.7,4.1,1.0,'Iris-setosa');
INSERT INTO iris_train VALUES(5.0,3.4,1.6,0.4,'Iris-versicolor');
INSERT INTO iris_train VALUES(5.2,3.4,1.4,0.2,'Iris-versicolor');
INSERT INTO iris_train VALUES(5.3,3.7,1.5,0.2,'Iris-versicolor');
INSERT INTO iris_train VALUES(5.0,3.6,1.4,0.2,'Iris-versicolor');
INSERT INTO iris_train VALUES(5.6,2.9,3.6,1.3,'Iris-setosa');
INSERT INTO iris_train VALUES(4.8,3.1,1.6,0.2,'Iris-versicolor');
INSERT INTO iris_train VALUES(6.3,2.7,4.9,1.8,'Iris-virginica');
INSERT INTO iris_train VALUES(5.7,2.8,4.1,1.3,'Iris-setosa');
INSERT INTO iris_train VALUES(5.0,3.0,1.6,0.2,'Iris-versicolor');
INSERT INTO iris_train VALUES(6.3,3.3,6.0,2.5,'Iris-virginica');
INSERT INTO iris_train VALUES(5.0,3.5,1.6,0.6,'Iris-versicolor');
INSERT INTO iris_train VALUES(5.5,2.6,4.4,1.2,'Iris-setosa');
INSERT INTO iris_train VALUES(5.7,3.0,4.2,1.2,'Iris-setosa');
INSERT INTO iris_train VALUES(4.4,2.9,1.4,0.2,'Iris-versicolor');
INSERT INTO iris_train VALUES(4.8,3.0,1.4,0.1,'Iris-versicolor');
INSERT INTO iris_train VALUES(5.5,2.4,3.7,1.0,'Iris-setosa');

CREATE TABLE `iris_test` LIKE `iris_train`;

INSERT INTO iris_test VALUES(5.9,3.0,4.2,1.5,'Iris-setosa');
INSERT INTO iris_test VALUES(6.9,3.1,5.4,2.1,'Iris-virginica');
INSERT INTO iris_test VALUES(5.1,3.3,1.7,0.5,'Iris-versicolor');
INSERT INTO iris_test VALUES(6.0,3.4,4.5,1.6,'Iris-setosa');
INSERT INTO iris_test VALUES(5.5,2.5,4.0,1.3,'Iris-setosa');
INSERT INTO iris_test VALUES(6.2,2.9,4.3,1.3,'Iris-setosa');
```

```
INSERT INTO iris_test VALUES(5.5,4.2,1.4,0.2,'Iris-versicolor');
INSERT INTO iris_test VALUES(6.3,2.8,5.1,1.5,'Iris-virginica');
INSERT INTO iris_test VALUES(5.6,3.0,4.1,1.3,'Iris-setosa');
INSERT INTO iris_test VALUES(6.7,2.5,5.8,1.8,'Iris-virginica');
INSERT INTO iris_test VALUES(7.1,3.0,5.9,2.1,'Iris-virginica');
INSERT INTO iris_test VALUES(4.3,3.0,1.1,0.1,'Iris-versicolor');
INSERT INTO iris_test VALUES(5.6,2.8,4.9,2.0,'Iris-virginica');
INSERT INTO iris_test VALUES(5.5,2.3,4.0,1.3,'Iris-setosa');
INSERT INTO iris_test VALUES(6.0,2.2,4.0,1.0,'Iris-setosa');
INSERT INTO iris_test VALUES(5.1,3.5,1.4,0.2,'Iris-versicolor');
INSERT INTO iris_test VALUES(5.7,2.6,3.5,1.0,'Iris-setosa');
INSERT INTO iris_test VALUES(4.8,3.4,1.9,0.2,'Iris-versicolor');
INSERT INTO iris_test VALUES(5.1,3.4,1.5,0.2,'Iris-versicolor');
INSERT INTO iris_test VALUES(5.7,2.5,5.0,2.0,'Iris-virginica');
INSERT INTO iris_test VALUES(5.4,3.4,1.7,0.2,'Iris-versicolor');
INSERT INTO iris_test VALUES(5.6,3.0,4.5,1.5,'Iris-setosa');
INSERT INTO iris_test VALUES(6.3,2.9,5.6,1.8,'Iris-virginica');
INSERT INTO iris_test VALUES(6.3,2.5,4.9,1.5,'Iris-setosa');
INSERT INTO iris_test VALUES(5.8,2.7,3.9,1.2,'Iris-setosa');
INSERT INTO iris_test VALUES(6.1,3.0,4.6,1.4,'Iris-setosa');
INSERT INTO iris_test VALUES(5.2,4.1,1.5,0.1,'Iris-versicolor');
INSERT INTO iris_test VALUES(6.7,3.1,4.7,1.5,'Iris-setosa');
INSERT INTO iris_test VALUES(6.7,3.3,5.7,2.5,'Iris-virginica');
INSERT INTO iris_test VALUES(6.4,2.9,4.3,1.3,'Iris-setosa');

CREATE TABLE `iris_validate` LIKE `iris_test`;

INSERT INTO `iris_validate` SELECT * FROM `iris_test`;
```

Before MySQL 8.0.32, drop the `class` column from `iris_test`.

```
mysql> ALTER TABLE `iris_test` DROP COLUMN `class`;
```

Training the Model

Train the model with `ML_TRAIN`. Since this is a classification dataset, use the `classification` task to create a classification model:

```
mysql> CALL sys.ML_TRAIN('ml_data.iris_train', 'class',
         JSON_OBJECT('task', 'classification'), @iris_model);
```

When the training operation finishes, the model handle is assigned to the `@iris_model` session variable, and the model is stored in the model catalog. View the entry in the model catalog with the following query. Replace *`user1`* with the MySQL account name:

```
mysql> SELECT model_id, model_handle, train_table_name FROM ML_SCHEMA_user1.MODEL_CATALOG;
+----------+---------------------------------------+--------------------+
| model_id | model_handle                          | train_table_name   |
+----------+---------------------------------------+--------------------+
|        1 | ml_data.iris_train_user1_1648140791   | ml_data.iris_train |
+----------+---------------------------------------+--------------------+
```

MySQL 8.0.31 does not run the `ML_EXPLAIN` routine with the default Permutation Importance model after `ML_TRAIN`. For MySQL 8.0.31, run `ML_EXPLAIN` and use `NULL` for the options:

```
mysql> CALL sys.ML_EXPLAIN('ml_data.iris_train', 'class',
         'ml_data.iris_train_user1_1648140791', NULL);
```

Loading the Model

Load the model into HeatWave AutoML using `ML_MODEL_LOAD` routine:

```
mysql> CALL sys.ML_MODEL_LOAD(@iris_model, NULL);
```

A model must be loaded before you can use it. The model remains loaded until you unload it or the HeatWave Cluster is restarted.

Making Predictions

You can make predictions on a single row of data or on the table of data.

- Make a prediction for a single row of data using the `ML_PREDICT_ROW` routine. In this example, data is assigned to a `@row_input` session variable, and the variable is called by the routine. The model handle is called using the `@iris_model` session variable:

```
mysql> SET @row_input = JSON_OBJECT(
           "sepal length", 7.3,
           "sepal width", 2.9,
           "petal length", 6.3,
           "petal width", 1.8);

mysql> SELECT sys.ML_PREDICT_ROW(@row_input, @iris_model, NULL);
+-----------------------------------------------------------------------------------------------------------------
| sys.ML_PREDICT_ROW('{"sepal length": 7.3, "sepal width": 2.9, "petal length": 6.3, "petal width": 1.8}
+-----------------------------------------------------------------------------------------------------------------
| {"Prediction": "Iris-virginica", "ml_results": "{'predictions': {'class': 'Iris-virginica'}, 'probabil
+-----------------------------------------------------------------------------------------------------------------
1 row in set (1.12 sec)
```

  Before MySQL 8.0.32, the `ML_PREDICT_ROW` routine does not include options, and the results do not include the `ml_results` field:

```
mysql> SELECT sys.ML_PREDICT_ROW(@row_input, @iris_model);
+-------------------------------------------------------------------------+
| sys.ML_PREDICT_ROW(@row_input, @iris_model)                             |
+-------------------------------------------------------------------------+
| {"Prediction": "Iris-virginica", "petal width": 1.8, "sepal width": 2.9,  |
| "petal length": 6.3, "sepal length": 7.3}                               |
+-------------------------------------------------------------------------+
```

  Based on the feature inputs that were provided, the model predicts that the Iris plant is of the class `Iris-virginica`. The feature values used to make the prediction are also shown.

- Make predictions for a table of data using the `ML_PREDICT_TABLE` routine. The routine takes data from the `iris_test` table as input and writes the predictions to an `iris_predictions` output table.

```
mysql> CALL sys.ML_PREDICT_TABLE('ml_data.iris_test', @iris_model,
          'ml_data.iris_predictions', NULL);
```

  To view `ML_PREDICT_TABLE` results, query the output table; for example:

```
mysql> SELECT * from ml_data.iris_predictions LIMIT 5;
+-----+--------------+-------------+--------------+-------------+-----------------+-----------------+---
| _id | sepal length | sepal width | petal length | petal width | class           | Prediction      | ml
+-----+--------------+-------------+--------------+-------------+-----------------+-----------------+---
|   1 |          7.3 |         2.9 |          6.3 |         1.8 | Iris-virginica  | Iris-virginica  | {'
|   2 |          6.1 |         2.9 |          4.7 |         1.4 | Iris-versicolor | Iris-versicolor | {'
|   3 |          6.3 |         2.8 |          5.1 |         1.5 | Iris-virginica  | Iris-versicolor | {'
|   4 |          6.3 |         3.3 |          4.7 |         1.6 | Iris-versicolor | Iris-versicolor | {'
|   5 |          6.1 |           3 |          4.9 |         1.8 | Iris-virginica  | Iris-virginica  | {'
+-----+--------------+-------------+--------------+-------------+-----------------+-----------------+---
5 rows in set (0.00 sec)
```

  Before MySQL 8.0.32, the `ML_PREDICT_TABLE` routine does not include options, and the results do not include the `ml_results` column:

```
mysql> CALL sys.ML_PREDICT_TABLE('ml_data.iris_test', @iris_model,
          'ml_data.iris_predictions');

mysql> SELECT * FROM ml_data.iris_predictions LIMIT 3;
*************************** 1. row ***************************
sepal length: 7.3
 sepal width: 2.9
petal length: 6.3
 petal width: 1.8
  Prediction: Iris-virginica
*************************** 2. row ***************************
sepal length: 6.1
 sepal width: 2.9
petal length: 4.7
 petal width: 1.4
  Prediction: Iris-versicolor
*************************** 3. row ***************************
sepal length: 6.3
 sepal width: 2.8
petal length: 5.1
 petal width: 1.5
  Prediction: Iris-virginica
```

The table shows the predictions and the feature column values used to make each prediction.

Generating Explanations

After creating predictions (either on a single row of data or the table of data), you can generate explanations to understand how the predictions were made and review which features had the most influence on predictions.

- Generate an explanation for a prediction made on a row of data using the `ML_EXPLAIN_ROW` routine with the Permutation Importance prediction explainer:

```
mysql> SELECT sys.ML_EXPLAIN_ROW(JSON_OBJECT("sepal length", 7.3, "sepal width", 2.9,  "petal
length", 6.3, "petal width", 1.8), @iris_model, JSON_OBJECT('prediction_explainer',
'permutation_importance'));
+-----------------------------------------------------------------------------------------------------------------
| sys.ML_EXPLAIN_ROW(JSON_OBJECT("sepal length", 7.3, "sepal width", 2.9, "petal length", 6.3, "petal width"
+-----------------------------------------------------------------------------------------------------------------
| {"Notes": "petal width (1.8) had the largest impact towards predicting Iris-virginica", "Prediction": "Iri
+-----------------------------------------------------------------------------------------------------------------
1 row in set (5.92 sec)
```

Before MySQL 8.0.32, the results do not include the `ml_results` field:

```
+-----------------------------------------------------------------------------+
| sys.ML_EXPLAIN_ROW(JSON_OBJECT("sepal length", 7.3, "sepal width", 2.9,     |
          "petal length", 6.3, "petal width", 1.8), @iris_model)              |
+-----------------------------------------------------------------------------+
| {"Prediction": "Iris-virginica", "petal width": 1.8, "sepal width": 2.9,    |
| "petal length": 6.3, "sepal length": 7.3, "petal width_attribution": 0.73,  |
| "petal length_attribution": 0.57}                                           |
+-----------------------------------------------------------------------------+
```

The attribution values show which features contributed most to the prediction, with petal length and petal width being the most important features. The other features have a 0 value indicating that they did not contribute to the prediction.

- Generate explanations for predictions made for a table of data using the `ML_EXPLAIN_TABLE` routine with the Permutation Importance prediction explainer.

Feature importance is presented as an attribution value ranging from -1 to 1. A positive value indicates that a feature contributed toward the prediction. A negative value indicates that the feature contributes positively towards one of the other possible predictions.

```
mysql> CALL sys.ML_EXPLAIN_TABLE('ml_data.iris_test', @iris_model,
'ml_data.iris_explanations', JSON_OBJECT('prediction_explainer',
'permutation_importance'));
```

To view `ML_EXPLAIN_TABLE` results, query the output table; for example:

```
mysql> SELECT * FROM ml_data.iris_explanations LIMIT 5;
+-----+--------------+-------------+--------------+-------------+-----------------+-----------------+---
| _id | sepal length | sepal width | petal length | petal width | class           | Prediction      | No
+-----+--------------+-------------+--------------+-------------+-----------------+-----------------+---
|   1 |          7.3 |         2.9 |          6.3 |         1.8 | Iris-virginica  | Iris-virginica  | pe
|   2 |          6.1 |         2.9 |          4.7 |         1.4 | Iris-versicolor | Iris-versicolor | pe
|   3 |          6.3 |         2.8 |          5.1 |         1.5 | Iris-virginica  | Iris-versicolor | pe
|   4 |          6.3 |         3.3 |          4.7 |         1.6 | Iris-versicolor | Iris-versicolor | pe
|   5 |          6.1 |           3 |          4.9 |         1.8 | Iris-virginica  | Iris-virginica  | pe
+-----+--------------+-------------+--------------+-------------+-----------------+-----------------+---
5 rows in set (0.00 sec)
```

Before MySQL 8.0.32, the output table does not include the `ml_results` column:

```
mysql> SELECT * FROM ml_data.iris_explanations LIMIT 3;
*************************** 1. row ***************************
            sepal length: 7.3
             sepal width: 2.9
            petal length: 6.3
             petal width: 1.8
              Prediction: Iris-virginica
petal length_attribution: 0.57
 petal width_attribution: 0.73
*************************** 2. row ***************************
            sepal length: 6.1
             sepal width: 2.9
            petal length: 4.7
             petal width: 1.4
              Prediction: Iris-versicolor
petal length_attribution: 0.14
 petal width_attribution: 0.6
*************************** 3. row ***************************
            sepal length: 6.3
             sepal width: 2.8
            petal length: 5.1
             petal width: 1.5
              Prediction: Iris-virginica
petal length_attribution: -0.25
 petal width_attribution: 0.31
3 rows in set (0.0006 sec)
```

Scoring the Model

Score the model with `ML_SCORE` to assess the reliability of the model. This example uses the `balanced_accuracy` metric, which is one of the many scoring metrics that HeatWave AutoML supports.

```
mysql> CALL sys.ML_SCORE('ml_data.iris_validate', 'class', @iris_model, 'balanced_accuracy',
@score, NULL);
```

Before MySQL 8.2.0, there is no options parameter available. So, the *`NULL`* parameter is not required.

```
mysql> CALL sys.ML_SCORE('ml_data.iris_validate', 'class', @iris_model,
          'balanced_accuracy', @score);
```

To retrieve the computed score, query the `@score` session variable.

```
mysql> SELECT @score;
+--------------------+
| @score             |
+--------------------+
| 0.9583333134651184 |
+--------------------+
```

Unloading the Model

Unload the model using `ML_MODEL_UNLOAD`:

```
mysql> CALL sys.ML_MODEL_UNLOAD(@iris_model);
```

To avoid consuming too much space, it is good practice to unload a model when you are finished using it.