Abstract

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

For information about creating and managing HeatWave Clusters, refer to the MySQL Database Service Guide.

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

For information about the latest MySQL 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: 2022-06-10 (revision: 73474)
# Table of Contents

Preface and Legal Notices ........................................................................................................... v  
1 Overview .................................................................................................................................. 1  
2 HeatWave .................................................................................................................................. 5  
   2.1 Before You Begin ............................................................................................................... 6  
   2.2 Loading Data .................................................................................................................... 6  
      2.2.1 Prerequisites ........................................................................................................... 7  
      2.2.2 Loading Data Manually .......................................................................................... 7  
      2.2.3 Loading Data Using Auto Parallel Load ............................................................... 8  
      2.2.4 Monitoring Load Progress ...................................................................................... 14  
      2.2.5 Checking Load Status ............................................................................................. 14  
      2.2.6 Data Compression .................................................................................................. 14  
      2.2.7 Change Propagation ................................................................................................. 14  
2.3 Running Queries .................................................................................................................. 15  
2.4 Modifying Tables ................................................................................................................. 21  
2.5 Unloading Tables .................................................................................................................. 23  
2.6 Table Load and Query Example ......................................................................................... 23  
2.7 Workload Optimization ....................................................................................................... 25  
   2.7.1 Encoding String Columns ......................................................................................... 25  
   2.7.2 Defining Data Placement Keys .................................................................................. 26  
   2.7.3 Workload Optimization using Advisor ...................................................................... 28  
2.8 Best Practices ....................................................................................................................... 41  
2.9 Supported Data Types ......................................................................................................... 50  
2.10 Supported SQL Modes ....................................................................................................... 51  
   2.11 Supported Functions and Operators ............................................................................. 51  
      2.11.1 Aggregate Functions ............................................................................................... 51  
      2.11.2 Arithmetic Operators .............................................................................................. 52  
      2.11.3 Cast Functions and Operators ............................................................................... 52  
      2.11.4 Comparison Functions and Operators .................................................................. 52  
      2.11.5 Control Flow Functions and Operators ............................................................... 53  
      2.11.6 Date and Time Functions ....................................................................................... 54  
      2.11.7 Logical Operators .................................................................................................. 56  
      2.11.8 Mathematical Functions ....................................................................................... 56  
      2.11.9 String Functions and Operators .......................................................................... 57  
      2.11.10 Window Functions ............................................................................................... 58  
2.12 String Column Encoding Reference ................................................................................. 59  
   2.12.1 Variable-length Encoding ......................................................................................... 59  
   2.12.2 Dictionary Encoding ................................................................................................. 60  
   2.12.3 Column Limits ......................................................................................................... 60  
2.13 Monitoring .......................................................................................................................... 61  
2.14 Troubleshooting ................................................................................................................ 66  
2.15 Metadata Queries .............................................................................................................. 69  
2.16 Limitations .......................................................................................................................... 72  
3 HeatWave ML .......................................................................................................................... 79  
   3.1 Before You Begin ............................................................................................................. 81  
   3.2 Getting Started ................................................................................................................. 82  
   3.3 Preparing Data .................................................................................................................. 82  
   3.4 Training a Model .............................................................................................................. 85  
   3.5 Predictions ....................................................................................................................... 86  
   3.6 Explanations ...................................................................................................................... 88  
   3.7 Managing Models ............................................................................................................. 89  
      3.7.1 The Model Catalog ................................................................................................. 89  
      3.7.2 Loading Models ....................................................................................................... 91  
      3.7.3 Unloading Models .................................................................................................... 91  
      3.7.4 Viewing Models ..................................................................................................... 92  
      3.7.5 Scoring Models ....................................................................................................... 92
Preface and Legal Notices

This is the user manual for MySQL HeatWave.

Legal Notices

Copyright © 1997, 2022, Oracle and/or its affiliates.

This software and related documentation are provided under a license agreement containing restrictions on use and disclosure and are protected by intellectual property laws. Except as expressly permitted in your license agreement or allowed by law, you may not use, copy, reproduce, translate, broadcast, modify, license, transmit, distribute, exhibit, perform, publish, or display any part, in any form, or by any means. Reverse engineering, disassembly, or decompilation of this software, unless required by law for interoperability, is prohibited.

The information contained herein is subject to change without notice and is not warranted to be error-free. If you find any errors, please report them to us in writing.

If this is software or related documentation that is delivered to the U.S. Government or anyone licensing it on behalf of the U.S. Government, then the following notice is applicable:

U.S. GOVERNMENT END USERS: Oracle programs (including any operating system, integrated software, any programs embedded, installed or activated on delivered hardware, and modifications of such programs) and Oracle computer documentation or other Oracle data delivered to or accessed by U.S. Government end users are "commercial computer software" or "commercial computer software documentation" pursuant to the applicable Federal Acquisition Regulation and agency-specific supplemental regulations. As such, the use, reproduction, duplication, release, display, disclosure, modification, preparation of derivative works, and/or adaptation of i) Oracle programs (including any operating system, integrated software, any programs embedded, installed or activated on delivered hardware, and modifications of such programs), ii) Oracle computer documentation and/or iii) other Oracle data, is subject to the rights and limitations specified in the license contained in the applicable contract. The terms governing the U.S. Government’s use of Oracle cloud services are defined by the applicable contract for such services. No other rights are granted to the U.S. Government.

This software or hardware is developed for general use in a variety of information management applications. It is not developed or intended for use in any inherently dangerous applications, including applications that may create a risk of personal injury. If you use this software or hardware in dangerous applications, then you shall be responsible to take all appropriate fail-safe, backup, redundancy, and other measures to ensure its safe use. Oracle Corporation and its affiliates disclaim any liability for any damages caused by use of this software or hardware in dangerous applications.

Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners.

Intel and Intel Inside are trademarks or registered trademarks of Intel Corporation. All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc. AMD, Epyc, and the AMD logo are trademarks or registered trademarks of Advanced Micro Devices. UNIX is a registered trademark of The Open Group.

This software or hardware and documentation may provide access to or information about content, products, and services from third parties. Oracle Corporation and its affiliates are not responsible for and expressly disclaim all warranties of any kind with respect to third-party content, products, and services unless otherwise set forth in an applicable agreement between you and Oracle. Oracle Corporation and its affiliates will not be responsible for any loss, costs, or damages incurred due to your access to or use of third-party content, products, or services, except as set forth in an applicable agreement between you and Oracle.

This documentation is NOT distributed under a GPL license. Use of this documentation is subject to the following terms:
You may create a printed copy of this documentation solely for your own personal use. Conversion to other formats is allowed as long as the actual content is not altered or edited in any way. You shall not publish or distribute this documentation in any form or on any media, except if you distribute the documentation in a manner similar to how Oracle disseminates it (that is, electronically for download on a Web site with the software) or on a CD-ROM or similar medium, provided however that the documentation is disseminated together with the software on the same medium. Any other use, such as any dissemination of printed copies or use of this documentation, in whole or in part, in another publication, requires the prior written consent from an authorized representative of Oracle. Oracle and/or its affiliates reserve any and all rights to this documentation not expressly granted above.

Documentation Accessibility

For information about Oracle's commitment to accessibility, visit the Oracle Accessibility Program website at
https://www.oracle.com/corporate/accessibility/.

Access to Oracle Support for Accessibility

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit
https://www.oracle.com/corporate/accessibility/learning-support.html#support-tab.
Chapter 1 Overview

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.


When you enable a HeatWave Cluster, analytics queries that meet certain prerequisites are automatically offloaded from the MySQL DB System to the HeatWave Cluster for accelerated processing, enabling you to 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 HeatWave's analytics capabilities, see Chapter 2, HeatWave.

Enabling a HeatWave Cluster also provides access to HeatWave Machine Learning (ML), which is a fully managed, highly scalable, cost-efficient, machine learning solution for data stored in MySQL. HeatWave ML 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 ML, 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.

With HeatWave ML, data and models never leave the MySQL Database Service, saving you time and effort while keeping your data and models secure. HeatWave ML is optimized for HeatWave shapes and scaling, and all HeatWave ML processing is performed on the HeatWave Cluster. ML computation is distributed among HeatWave nodes, taking advantage of HeatWave's scalability and massively parallel processing capabilities. For more information about HeatWave's machine learning capabilities, see Chapter 3, HeatWave ML.

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 supports up to 64 nodes.

Data that is loaded into HeatWave is automatically persisted to OCI Object Storage for fast recovery in case of a HeatWave Cluster failure.

HeatWave network traffic is fully encrypted.
HeatWave Architectural Features

- **In-Memory Hybrid-Columnar Format**
- **Massively Parallel Architecture**
- **Push-Based Vectorized Query Processing**
- **Scale-Out Data Management**
- **Native MySQL Integration**

**In-Memory Hybrid-Columnar Format**

HeatWave stores data in main memory in a hybrid columnar format. HeatWave’s 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**

HeatWave’s massively parallel architecture is enabled by 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.
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 to OCI Object Storage for fast recovery in case of a HeatWave node or cluster failure. Data is automatically restored by the HeatWave Storage Layer when HeatWave 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. The time required to reload data is constant regardless of data size or HeatWave Cluster size.

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.

MySQL Autopilot

MySQL 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 MySQL Autopilot to execute its core capabilities. MySQL Autopilot makes the HeatWave query optimizer increasingly intelligent as more queries are executed, resulting in continually improving system performance.

Autopilot focuses on four aspects of the HeatWave service life cycle:

- System Setup
- Data Load
- Query Execution
System Setup

- **Failure Handling**

**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 Encoding**

  Determines the optimal representation of columns loaded into HeatWave by analyzing HeatWave query history, which improves query performance and minimizes the required cluster size. See Section 2.7.3.1, “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 2.7.3.2, “Auto Data Placement”.

**Query Execution**

- **Auto Query Plan Improvement**

  Uses statistics from previously executed queries to improve future query execution plans. See Auto Query Plan Improvement.

- **Auto Query Time Estimation**

  Estimates query execution time, allowing you to determine how a query might perform without having to run the query. Runtime estimates are provided by the Advisor Query Insights feature. See Section 2.7.3.3, “Query Insights”.

- **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.

- **Auto Scheduling**

  Prioritizes queries in an intelligent way to reduce overall query execution wait times. See Auto Scheduling.

**Failure Handling**

- **Auto Error Recovery**

  Auto Error Recovery provisions new HeatWave nodes and reloads data from the HeatWave storage layer if one or more HeatWave nodes becomes unresponsive due to a software or hardware failure. See HeatWave Cluster Failure and Recovery.
Chapter 2 HeatWave

Table of Contents

2.1 Before You Begin ................................................................. 6
2.2 Loading Data ........................................................................ 6
  2.2.1 Prerequisites .................................................................. 7
  2.2.2 Loading Data Manually ................................................. 7
  2.2.3 Loading Data Using Auto Parallel Load ...................... 8
  2.2.4 Monitoring Load Progress ......................................... 14
  2.2.5 Checking Load Status .................................................. 14
  2.2.6 Data Compression ......................................................... 14
  2.2.7 Change Propagation ....................................................... 14
2.3 Running Queries ................................................................... 15
2.4 Modifying Tables .................................................................. 21
2.5 Unloading Tables ................................................................. 23
2.6 Table Load and Query Example ........................................... 23
2.7 Workload Optimization ....................................................... 25
  2.7.1 Encoding String Columns ............................................. 25
  2.7.2 Defining Data Placement Keys .................................... 26
  2.7.3 Workload Optimization using Advisor ....................... 28
2.8 Best Practices ....................................................................... 41
2.9 Supported Data Types ......................................................... 50
2.10 Supported SQL Modes ....................................................... 51
2.11 Supported Functions and Operators ....................................... 51
  2.11.1 Aggregate Functions ..................................................... 51
  2.11.2 Arithmetic Operators ................................................... 52
  2.11.3 Cast Functions and Operators ...................................... 52
  2.11.4 Comparison Functions and Operators ....................... 52
  2.11.5 Control Flow Functions and Operators ..................... 53
  2.11.6 Date and Time Functions ............................................. 54
  2.11.7 Logical Operators ........................................................ 56
  2.11.8 Mathematical Functions .............................................. 56
  2.11.9 String Functions and Operators .................................. 57
  2.11.10 Window Functions ...................................................... 58
2.12 String Column Encoding Reference ....................................... 59
  2.12.1 Variable-length Encoding .......................................... 59
  2.12.2 Dictionary Encoding .................................................... 60
  2.12.3 Column Limits ............................................................ 60
2.13 Monitoring ......................................................................... 61
2.14 Troubleshooting ................................................................. 66
2.15 Metadata Queries ............................................................... 69
2.16 Limitations ......................................................................... 72

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".
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”.

Data loaded into HeatWave, including propagated changes, are automatically persisted by the HeatWave Storage Layer to OCI Object Storage for a fast recovery in case of a HeatWave node or cluster failure.

After running a number of queries, you can use the HeatWave Advisor to optimize your workload. Advisor analyzes your data and query history to provide string column encoding and data placement recommendations. See Section 2.7.3, “Workload Optimization using Advisor”.

2.1 Before You Begin

Before you begin using HeatWave, the following is assumed:

• You have an operational DB System and you are able to connect to it using a MySQL client. If not, see Creating a DB System, and Connecting to a DB System, in the MySQL Database Service Guide.

• Your MySQL DB System has an operational HeatWave Cluster. If not, see Adding a HeatWave Cluster, in the MySQL Database Service Guide.

2.2 Loading Data

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 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”.

When data is loaded into HeatWave, it is read from InnoDB using batched, multi-threaded reads. It is then converted into columnar format and sent over the network and distributed among the HeatWave nodes in horizontal slices. Data is partitioned by the table’s primary key unless data placement keys are defined. 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, 4MB of memory (the default heap segment size) is allocated from the root heap. This memory requirement should be considered when loading a large number of tables. For example, with a root heap of approximately 400GB available to HeatWave, loading 100K tables would consume all available root heap memory (100K x 4GB = 400GB). From MySQL 8.0.30, the default heap segment size is reduced from 4MB to a default of 64KB per table, reducing the amount of memory that must be allocated from the root heap for each loaded table.

As of MySQL 8.0.28-u1, 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. For instructions, see Section 2.2.6, “Data Compression”.

6
Prerequisites

Note
DDL operations are not permitted on tables that are loaded in HeatWave. To alter the definition of a table, you must unload the table and remove the SECONDARY_ENGINE attribute before performing the DDL operation. See Section 2.4, “Modifying Tables”.

For related best practices, see Section 2.8, “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 Importing and Exporting Databases in the MySQL Database 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:

  ```sql
  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:

  ```sql
  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.

- Table column width cannot exceed 65532 bytes.

- The number of columns per table cannot exceed 900.

2.2.2 Loading Data Manually

Loading data manually involves the following steps:

1. Excluding columns with unsupported data types. See Excluding Table Columns.

2. Defining RAPID as the secondary engine for tables you want to load. See Defining the Secondary Engine.

3. Optionally, applying string column encoding and data placement workload optimizations. For more information, see Section 2.7, “Workload Optimization”.

4. Loading tables using ALTER TABLE ... SECONDARY_LOAD statements. See Loading Tables.

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 2.9, “Supported Data Types”.
Loading Data Using Auto Parallel Load

Optionally, you can also exclude columns that are not relevant to the queries you intend to run. 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.

```sql
mysql> ALTER TABLE orders MODIFY description BLOB NOT SECONDARY;
```

```sql
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”.

**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:

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

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

**Loading Tables**

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

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

The `SECONDARY_LOAD` clause has these properties:

- It is considered a local operation and is therefore omitted from the binary log.
- Data is read using the `READ COMMITTED` isolation level.

**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.
- 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:

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

Auto Parallel Load is described under the following topics in this section:

- Auto Parallel Load Requirements
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.

Auto Parallel Load Syntax

```sql
CALL sys.heatwave_load (db_list, [options]);
```

**db_list** specifies the schemas to load. The list is specified as a `JSON_ARRAY`. Specifying an empty array is permitted for viewing the Auto Parallel Load command-line help (see Auto Parallel Load Command-Line Help). Otherwise, one or more valid schema names are required.

**options** are specified as key-value pairs in JSON object format. If an option is not specified, the default setting is used. If no options are specified, `NULL` can be specified in place of the option's argument.

For syntax examples, see Auto Parallel Load Examples.

Auto Parallel Load options include:

- **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.

- **output**: Defines how Auto Parallel Load produces output. Permitted values are:
  - `normal`: The default. Produces summarized output that is sent to `stdout` and to the `heatwave_load_report` table. (See The Auto Parallel Load Report Table.)
Loading Data Using Auto Parallel Load

• **silent**: Sends output to the `heatwave_load_report` table only. (See The Auto Parallel Load 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 Auto Parallel Load Examples.

• **compact**: Produces compact output.

• **help**: Displays Auto Parallel Load command-line help. See 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 Excluding Table Columns.

• **exclude_list**: Defines a list of database objects (schemas, tables, and columns) to exclude from the load script. Names must be fully qualified without backticks, as in the following example:

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

  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 excluded automatically.

• **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.

• **check**: The default. Checks if there is enough memory on the MySQL node for dictionary-encoded columns and is 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, 4MB of memory (the default heap segment size) must be allocated from the root heap for variable-length column encoding overhead. From MySQL 8.0.30, the default heap segment size is reduced from 4GB to a default of 64KB per table. 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 Memory Estimation for String Column Encoding.

Running Auto Parallel Load

An Auto Parallel Load call that loads a single schema ("tpch") is written as follows:

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

Specifying NULL in place of options means that Auto Parallel Load is run with the default option settings.

You can run Auto Parallel Load in "dryrun" mode first to check for errors and warnings and to inspect the generated load script. An Auto Parallel Load call in "dryrun" mode that specifies a single schema ("tpch") is written as follows:

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

In "dryrun" mode, Auto Parallel Load sends the load script to the heatwave_load_report table only. It does not load data into HeatWave.

If Auto Parallel Load fails with an error, inspect the errors by querying the heatwave_load_report table:

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

When Auto Parallel Load finishes running, use the following query to check for warnings:

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

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

```sql
SELECT log->>"$.sql" AS "Load Script" FROM sys.heatwave_load_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:

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

Note

You can retrieve DDL statements in a table or use the following statements to produce a list of DDL statements that you can easily copy and paste:

```sql
SET SESSION group_concat_max_len = 1000000;
SELECT GROUP_CONCAT(log->>"$.sql" SEPARATOR ' ') FROM sys.heatwave_load_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.

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 want to ensure that there is sufficient memory for string column encoding before attempting a load operation. Insufficient memory can cause a load failure.

```sql
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.

### The Auto Parallel Load Report Table

When Auto Parallel Load is run, output including Auto Parallel Load execution logs and the generated load script is sent 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

The `heatwave_load_report` table can be queried after running Auto Parallel Load, as in the following examples:

- View error information in case Auto Parallel Load stops unexpectedly:
  ```sql
  SELECT log FROM sys.heatwave_load_report WHERE type="error";
  ```

- View warnings to find out why tables cannot be loaded:
  ```sql
  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:
  ```sql
  SELECT log->>'$.sql' AS "Load Script" FROM sys.heatwave_load_report
  WHERE type = "sql" ORDER BY id;
  ```

- View the number of load commands generated:
  ```sql
  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:
  ```sql
  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 that can be copied and pasted 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.)
  ```sql
  SET SESSION group_concat_max_len = 1000000;
  SELECT GROUP_CONCAT(log->>'$.sql' SEPARATOR ' ') FROM sys.heatwave_load_report
  WHERE type = "sql" ORDER BY id;
  ```

### Auto Parallel Load Command-Line Help

To view Auto Parallel Load command-line help, issue the following statement:
The command-line help provides usage documentation for the Auto Parallel Load utility.

**Auto Parallel Load Examples**

- **Load multiple schemas. No options are specified, which means that the default options are used.**

```sql
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.**

```sql
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:**

```sql
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`.**

```sql
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%');
CALL sys.heatwave_load(JSON_ARRAY('schema_customer_1'), @exc_list);
```

- **Load all schemas with tables that start with an "hw" prefix.**

```sql
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 NOT IN
                               ('mysql','information_schema','performance_schema','sys')
                         AND table_name NOT LIKE 'hw%');
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:**

```sql
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_load_report WHERE type="warn"
                        AND stage="VERIFICATION" and log like "%Unsupported%";
    SELECT Count(*) AS "Total Load commands Generated"
                        FROM sys.heatwave_load_report WHERE type = "sql" ORDER BY id;
END //
DELIMITER ;
CALL auto_load_wrapper();
```
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.

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

<table>
<thead>
<tr>
<th>VARIABLE_VALUE</th>
</tr>
</thead>
<tbody>
<tr>
<td>100.000000</td>
</tr>
</tbody>
</table>

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:

```sql
mysql> USE performance_schema;
mysql> SELECT NAME, LOAD_STATUS FROM rpd_tables, rpd_table_id
    -> WHERE rpd_tables.ID = rpd_table_id.ID;
```

<table>
<thead>
<tr>
<th>NAME</th>
<th>LOAD_STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>tpch.supplier</td>
<td>AVAIL_RPDGSTABSTATE</td>
</tr>
<tr>
<td>tpch.partsupp</td>
<td>AVAIL_RPDGSTABSTATE</td>
</tr>
<tr>
<td>tpch.orders</td>
<td>AVAIL_RPDGSTABSTATE</td>
</tr>
<tr>
<td>tpch.lineitem</td>
<td>AVAIL_RPDGSTABSTATE</td>
</tr>
<tr>
<td>tpch.customer</td>
<td>AVAIL_RPDGSTABSTATE</td>
</tr>
<tr>
<td>tpch.nation</td>
<td>AVAIL_RPDGSTABSTATE</td>
</tr>
<tr>
<td>tpch.region</td>
<td>AVAIL_RPDGSTABSTATE</td>
</tr>
<tr>
<td>tpch.part</td>
<td>AVAIL_RPDGSTABSTATE</td>
</tr>
</tbody>
</table>

The `AVAIL_RPDGSTABSTATE` status indicates that the table is loaded. For information about load statuses, see Section 5.8, “The rpd_tables Table”.

2.2.6 Data Compression

As of MySQL 8.0.28-u3, 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 your data.

While data compression results in smaller HeatWave Clusters, 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, you can disable data compression by setting the `rapid_compression` session variable to `OFF` before loading data:

```sql
SET SESSION rapid_compression=OFF;
```

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, 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, “Unloading Tables”, and Section 2.2, “Loading Data”.

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

```sql
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.

```sql
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 |
+---------------+-------------------------------+
```

Change propagation does not support cascading changes triggered by a foreign key constraint.

A `time_zone` setting other than `SYSTEM (+00:00)` is not supported when propagating changes to tables containing `TIMESTAMP` columns. Propagating changes in this scenario causes incorrect data to be stored in `TIMESTAMP` columns, leading to incorrect query results.

## 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 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.
Query Prerequisites

For information about connecting to a MySQL DB System, see Connecting to a DB System, in the MySQL Database Service Guide.

Running queries and other query-related topics are described in the following sections:

- Query Prerequisites
- Running Queries
- Auto Scheduling
- Auto Query Plan Improvement
- Debugging Queries
- Query Runtimes and Estimates
- CREATE TABLE ... SELECT Statements
- INSERT ... SELECT Statements
- Using Views

For related best practices, see Section 2.8, “Best Practices”.

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 CREATE TABLE ... SELECT Statements, and INSERT ... SELECT Statements.

- All tables accessed by the query must be defined with RAPID as the secondary engine. See Defining the Secondary Engine.

- All tables accessed by the query must be loaded in HeatWave. See Section 2.2, “Loading Data”.

- `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 2.11, “Supported Functions and Operators”.

- Queries must avoid known limitations. See Section 2.16, “Limitations”.

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

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".

```
Auto Scheduling

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 2.14, “Troubleshooting”, or try debugging the query using the procedure described in Debugging Queries.

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

```sql
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;
```

<table>
<thead>
<tr>
<th>O_ORDERPRIORITY</th>
<th>ORDER_COUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-URGENT</td>
<td>2017573</td>
</tr>
<tr>
<td>2-HIGH</td>
<td>2015859</td>
</tr>
<tr>
<td>3-MEDIUM</td>
<td>2013174</td>
</tr>
<tr>
<td>4-NOT SPECIFIED</td>
<td>2014476</td>
</tr>
<tr>
<td>5-LOW</td>
<td>2013674</td>
</tr>
</tbody>
</table>

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.

```sql
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;
```

<table>
<thead>
<tr>
<th>O_ORDERPRIORITY</th>
<th>ORDER_COUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-URGENT</td>
<td>2017573</td>
</tr>
<tr>
<td>2-HIGH</td>
<td>2015859</td>
</tr>
<tr>
<td>3-MEDIUM</td>
<td>2013174</td>
</tr>
<tr>
<td>4-NOT SPECIFIED</td>
<td>2014476</td>
</tr>
<tr>
<td>5-LOW</td>
<td>2013674</td>
</tr>
</tbody>
</table>

5 rows in set (8.91 sec)

Note

Concurrently issued queries are prioritized for execution. For information about query prioritization, see Auto Scheduling.

Auto Scheduling

HeatWave 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.

When HeatWave is idle, an arriving query is scheduled immediately for execution. It is not queued. A query is queued only if a preceding query is running on HeatWave.
Auto Query Plan Improvement

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.

For a query that you can run to view the HeatWave query history including query start time, end time, and wait time in the scheduling queue, see Section 2.13, “Monitoring”.

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, JOINs, GROUP BY operations, and so on.

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.

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\G
```

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

```
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 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                     |
| TIMEDIFF(O_ORDERDATE,'2000:01:01 00:00:00.000001') | [{"Reason": "Function timediff is not 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:

```
["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:

```
["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 2.14, “Troubleshooting”.

For a query that attempts to access a column defined as `NOT_SECONDARY`, the following reason is reported:

```
["Reason": "Column risk_assessment is marked as NOT SECONDARY."]
```

Columns defined as `NOT_SECONDARY` are excluded when a table is loaded into HeatWave. See Excluding Table Columns.

**Query Runtimes and Estimates**

You can view HeatWave query runtimes and runtime estimates using HeatWave Advisor Query Insights feature or by querying 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.
CREATE TABLE ... SELECT Statements

Using Query Insights

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

```sql
CALL sys.heatwave_advisor(JSON_OBJECT("query_insights", TRUE));
```

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

```sql
CALL sys.heatwave_advisor(JSON_OBJECT("query_insights", TRUE,
"query_session_id", JSON_ARRAY(connection_id())));
```

For additional information about using Query Insights, see Section 2.7.3.3, “Query Insights”.

Using the rpd_query_stats Table

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

```sql
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:

```sql
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:

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

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:

```sql
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.

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 HeatWave and inserts the result set into the orders2 table on the MySQL DB System:

```sql
mysql> INSERT INTO orders2 SELECT * FROM orders;
```
Using Views

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.
- `SELECT .. UNION ALL` queries are not offloaded if the `INSERT` table is the same as the `SELECT` table because MySQL Server uses a temporary table in this case, which cannot be offloaded.
- `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.

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.

```sql
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:

```sql
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

Modifying a table that is loaded in HeatWave requires unloading, modifying, and reloading the table, as described in the following procedure:

1. Unload the table from HeatWave; for example:

   ```sql
   ALTER TABLE orders SECONDARY_UNLOAD;
   ```

2. Set the `SECONDARY_ENGINE` attribute to `NULL`; for example:

   ```sql
   ALTER TABLE orders SECONDARY_ENGINE = NULL;
   ```

3. Modify the table. The following examples demonstrate adding a previously excluded column, modifying or removing a column encoding, and modifying or removing a data placement key.
Examples are based on the orders table describes in Section 2.6, "Table Load and Query Example".

- **Adding a previously excluded column**

  Columns are excluded by specifying the `NOT SECONDARY` column attribute in a `CREATE TABLE` or `ALTER TABLE` statement; for example:

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

  To include a previously excluded column the next time the table is loaded, modify the column definition to remove the `NOT SECONDARY` column attribute; for example:

  ```sql
  ALTER TABLE orders MODIFY `O_COMMENT` varchar(79) NOT NULL;
  ```

- **Modifying or removing a column encoding**

  String column encoding is defined by specifying the `RAPID_COLUMN=ENCODING={SORTED|VARLEN}` keyword string in a column comment; for example:

  ```sql
  ALTER TABLE orders MODIFY `O_COMMENT` VARCHAR(79) COLLATE utf8mb4_bin NOT NULL COMMENT 'RAPID_COLUMN=ENCODING=SORTED';
  ```

  To modify the column encoding, alter the column comment; for example:

  ```sql
  ALTER TABLE orders MODIFY `O_COMMENT` VARCHAR(79) COLLATE utf8mb4_bin NOT NULL COMMENT 'RAPID_COLUMN=ENCODING=VARLEN';
  ```

  To remove a column encoding, remove the `RAPID_COLUMN=ENCODING={SORTED|VARLEN}` keyword string from the column comment. The next time the table is loaded, the string column encoding defaults to `VARLEN` encoding, which is the default string column encoding.

  The following example removes the column comment entirely, but if there are other column comments that you want to keep, you need only remove the encoding keyword string.

  ```sql
  ALTER TABLE orders MODIFY `O_COMMENT` VARCHAR(79) COLLATE utf8mb4_bin NOT NULL;
  ```

- **Modifying or removing a data placement key**

  A data placement key is defined by specifying the `RAPID_COLUMN=DATA_PLACEMENT_KEY=N` keyword string in a column comment; for example:

  ```sql
  ALTER TABLE orders MODIFY `O_ORDERDATE` DATE NOT NULL COMMENT 'RAPID_COLUMN=DATA_PLACEMENT_KEY=1';
  ```

  To modify a data placement key, modify the data placement keyword string:

  ```sql
  ALTER TABLE orders MODIFY `O_ORDERDATE` DATE NOT NULL COMMENT 'RAPID_COLUMN=DATA_PLACEMENT_KEY=2';
  ```

  To remove a data placement key, modify the column comment to remove the `RAPID_COLUMN=DATA_PLACEMENT_KEY=N` keyword string. The following example removes the column comment entirely, but if there are other column comments that you want to keep, you need only remove the data placement keyword string.

  ```sql
  ALTER TABLE orders MODIFY `O_ORDERDATE` DATE NOT NULL;
  ```

4. After making the desired changes to the table, set the `SECONDARY_ENGINE` attribute back to `RAPID`; for example:

   ```sql
   ALTER TABLE orders SECONDARY_ENGINE = RAPID;
   ```

5. Reload the table; for example:

   ```sql
   ALTER TABLE orders SECONDARY_LOAD;
   ```
2.5 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 orders SECONDARY_UNLOAD;
```

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

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.

```
# 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

# 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;

# 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';

# Define RAPID as the secondary engine for the table
mysql> ALTER TABLE orders SECONDARY_ENGINE RAPID;

# 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) 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` int NOT NULL,
  `O_COMMENT` varchar(79) NOT NULL,
  PRIMARY KEY (`O_ORDERKEY`) ) ENGINE=RAPID DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_bin
```
Table Load and Query Example

```sql
'0_ORDERKEY' int NOT NULL,
'0_CUSTKEY' int NOT NULL,
'0_ORDERSTATUS' char(1) COLLATE utf8mb4_bin NOT NULL COMMENT 'RAPID_COLUMN=ENCODING=SORTED',
'0_TOTALPRICE' decimal(15,2) NOT NULL,
'0_ORDERDATE' date NOT NULL,
'0_ORDERPRIORITY' char(15) COLLATE utf8mb4_bin NOT NULL COMMENT 'RAPID_COLUMN=ENCODING=SORTED',
'0_CLERK' char(15) COLLATE utf8mb4_bin NOT NULL COMMENT 'RAPID_COLUMN=ENCODING=SORTED',
'0_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

# Load the table into HeatWave
mysql> ALTER TABLE orders SECONDARY_LOAD;

# 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)

# 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;

+-----------------+-------------+
<table>
<thead>
<tr>
<th>O_ORDERPRIORITY</th>
<th>ORDER_COUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-URGENT</td>
<td>2017573</td>
</tr>
<tr>
<td>2-HIGH</td>
<td>2015859</td>
</tr>
<tr>
<td>3-MEDIUM</td>
<td>2013174</td>
</tr>
<tr>
<td>4-NOT SPECIFIED</td>
<td>2014476</td>
</tr>
<tr>
<td>5-LOW</td>
<td>2013674</td>
</tr>
</tbody>
</table>
+-----------------+-------------+
5 rows in set, 0.04 sec

# 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;

+-----------------+-------------+
<table>
<thead>
<tr>
<th>O_ORDERPRIORITY</th>
<th>ORDER_COUNT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-URGENT</td>
<td>2017573</td>
</tr>
<tr>
<td>2-HIGH</td>
<td>2015859</td>
</tr>
<tr>
<td>3-MEDIUM</td>
<td>2013174</td>
</tr>
</tbody>
</table>
+-----------------+-------------+
3 rows in set, 0.04 sec
```

### 2.7 Workload Optimization

This section describes optimizations that you can apply to your data to improve HeatWave query performance. Optimizations include using dictionary encoding for certain string columns and defining data placements keys to optimize for **JOIN** and **GROUP BY** query performance. For information about applying these optimizations, see Section 2.7.1, “Encoding String Columns”, and Section 2.7.2, “Defining Data Placement Keys”.

You can determine where to apply the encoding and data placement optimizations yourself or run the Advisor utility for recommendations. The Advisor **Auto Encoding** feature provides string column encoding recommendations. The Advisor **Auto Data Placement** feature recommends data placement keys. See Section 2.7.3, “Workload Optimization using Advisor”.

Advisor also includes a **Query Insights** feature that provides query runtimes and runtime estimates, which can be used to optimize queries, troubleshoot, and perform workload cost estimations. See Section 2.7.3.3, “Query Insights”.

#### 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:

```sql
CREATE TABLE orders (name VARCHAR(100) COMMENT 'RAPID_COLUMN=ENCODING=SORTED');
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 the Advisor utility after loading tables into HeatWave and running queries. For more information, see Section 2.7.3, “Workload Optimization using Advisor”.

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

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>
<thead>
<tr>
<th>Encoding Type</th>
<th>Expression, Filter, Function, and Operator Support</th>
<th>Best Suited To</th>
<th>Space Required On</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable-length (VARLEN)</td>
<td>Supports JOIN operations, string functions and operators, and LIKE predicates. See Section 2.12.1, “Variable-length Encoding”.</td>
<td>Columns with a high number of distinct values</td>
<td>HeatWave nodes</td>
</tr>
<tr>
<td>Dictionary (SORTED)</td>
<td>Does not support JOIN operations, string functions and operators, or LIKE predicates.</td>
<td>Columns with a low number of distinct values</td>
<td>MySQL DB System node</td>
</tr>
</tbody>
</table>

For additional information about string column encoding, see Section 2.12, “String Column Encoding Reference”.

2.7.2 Defining Data Placement Keys

When data is loaded into HeatWave, it is partitioned by the table's 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 the Advisor utility after loading tables into HeatWave and running queries. For more information, see Section 2.7.3, “Workload Optimization using Advisor”.

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

RAPIDCOLUMN=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 \texttt{CREATE TABLE} or \texttt{ALTER TABLE} statement:

\begin{verbatim}
CREATE TABLE orders (date DATE COMMENT 'RAPID_COLUMN=DATA_PLACEMENT_KEY=1');
ALTER TABLE orders MODIFY date DATE COMMENT 'RAPID_COLUMN=DATA_PLACEMENT_KEY=1';
\end{verbatim}

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.

\begin{verbatim}
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'
};
\end{verbatim}

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

\textbf{Note}

Other information is permitted in column comments. For example, it is permitted to specify a column description alongside a data placement keyword string:

\begin{verbatim}
COMMENT 'column_description RAPID_COLUMN=DATA_PLACEMENT_KEY=1'
\end{verbatim}

To modify or remove a data placement key, refer to the procedure described in Section 2.4, “Modifying Tables”.

Usage notes:

- \texttt{JOIN} and \texttt{GROUP BY} query optimizations are only applied if at least one of the \texttt{JOIN} or \texttt{GROUP BY} relations has a key that matches the defined data placement key.
- If a \texttt{JOIN} operation can be executed with or without the \texttt{JOIN} and \texttt{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 2.9, “Supported Data Types”.
- A data placement key column cannot be defined as a \texttt{NOT SECONDARY} column. See Excluding Table Columns.
2.7.3 Workload Optimization using Advisor

This section describes the Advisor utility, which provides the following optimization capabilities:

- **Auto Encoding**
  Recommends string column encodings for improving query performance and reducing the amount of memory required on HeatWave nodes. See Section 2.7.3.1, “Auto Encoding”.

- **Auto Data Placement**
  Recommends data placement keys for optimizing **JOIN** and **GROUP BY** query performance. See Section 2.7.3.2, “Auto Data Placement”.

- **Query Insights**
  Provides runtimes for successfully executed queries and runtime estimates for **EXPLAIN** queries, queries cancelled using 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. See Section 2.7.3.3, “Query Insights”.

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. Running Advisor involves issuing a **CALL** statement for the stored procedure with optional arguments.

```
CALL sys.heatwave_advisor (options);
```

Issue the following statement to view Advisor command-line help:

```
CALL sys.heatwave_advisor (JSON_OBJECT("output","help"));
```

Advisor can be run from any MySQL client or connector.

**Advisor Requirements**

- To run Advisor, the HeatWave Cluster must be active.
- 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.

2.7.3.1 Auto Encoding

**Auto Encoding** provides string column encoding recommendations. Choosing the right string column encodings can improve the performance of queries accessing those columns. The type of encoding applied to string columns also affects the amount of memory required on HeatWave nodes. 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, which may not be the optimal encoding choice in all cases. Auto Encoding generates string column encoding recommendations by analyzing column data, HeatWave query history, and available MySQL node memory. For more information about string column encoding, see Section 2.7.1, “Encoding String Columns”.

```
Auto Encoding Syntax

CALL sys.heatwave_advisor ([options]);

options: {
  JSON_OBJECT("key","value":[,"key","value"] ...
  "key","value":
  ["output",{"normal"|"silent"|"help"}]
  ["target_schema",JSON_ARRAY([,"schema_name"])
  ["exclude_query",JSON_ARRAY("query_id"),...
  ["query_session_id",JSON_ARRAY("query_session_id"),...
  ["auto_enc",JSON_OBJECT(auto_enc_option)]
}

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

For syntax examples, see Section 2.7.3.4, “Advisor Examples”.

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

• **output**: Defines how Advisor produces output. Permitted values are:
  • **normal**: The default. Produces summarized output that is sent to stdout and to the heatwave_advisor_report table. (See Section 2.7.3.5, “Advisor Report Table”.)
  • **silent**: Sends output to the heatwave_advisor_report table only. (See Section 2.7.3.5, “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 Advisor command-line help. See Section 2.7.3.4, “Advisor Examples”.

• **target_schema**: Defines one or more schemas for Advisor to analyze. The list is specified in JSON-array format. If a target schema is not specified, all schemas in the HeatWave Cluster are analyzed. 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 your 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 2.7.3.4, “Advisor 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 2.7.3.4, “Advisor Examples”.

• **auto_enc**: Defines settings for Auto Encoding, which provides string column encoding recommendations. Settings include:
  • **mode**: Defines the operational mode. Permitted values are:
    • **off**: The default. Disables the Auto Encoding feature.
    • **recommend**: The Auto Encoding feature recommends string column encodings.

  • **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.
Running Auto Encoding

Auto Encoding is enabled by specifying the `auto_enc` option in `recommend` mode. See Auto Encoding Syntax.

**Note**

If you intend to run Advisor for both encoding and data placement recommendations, it is recommended that you run Auto Encoding first, apply the recommended encodings, rerun your 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 2.12.2, “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.

```sql
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”.)

```sql
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)
```
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.)

```
SET SESSION group_concat_max_len = 1000000;
```
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 2.12.2, “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.

2.7.3.2 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”.

Auto Data Placement Syntax

```sql
CALL sys.heatwave_advisor([options]);
```
For syntax examples, see Section 2.7.3.4, “Advisor Examples”.

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

- **output**: Defines how Advisor produces output. Permitted values are:
  - **normal**: The default. Produces summarized output that is sent to `stdout` and to the `heatwave_advisor_report` table. (See Section 2.7.3.5, “Advisor Report Table”.)
  - **silent**: Sends output to the `heatwave_advisor_report` table only. (See Section 2.7.3.5, “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 Advisor command-line help. See Section 2.7.3.4, “Advisor Examples”.

- **target_schema**: Defines one or more schemas for Advisor to analyze. The list is specified in JSON-array format. If a target schema is not specified, all schemas in HeatWave are analyzed. 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 your 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 2.7.3.4, “Advisor 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 2.7.3.4, “Advisor Examples”.

- **auto_dp**: Defines settings for the Data Placement feature, which recommends data placement keys. Settings 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.

### Running Auto Data Placement

**Note**

If you intend to run Advisor for both encoding and data placement recommendations, it is recommended that you run Auto Encoding first, apply the recommended encodings, rerun your 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.

```
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 |
+-------------------------------+ 6 rows in set (0.01 sec)
+----------------------------------------------------------------------+
| AUTO DATA PLACEMENT                                                  |                                                                                         |                                                                      |
+----------------------------------------------------------------------+
| Auto Data Placement Configuration:                                   |                                                                                         |                                                                      |
|   Minimum benefit threshold: 1%                                       |                                                                                         |                                                                      |
| Producing Data Placement suggestions for current setup:              |                                                                                         |                                                                      |
|   Tables Loaded: 8                                                   |                                                                                         |                                                                      |
|     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)                                           | 34                                      |
Total Data Placement suggestions produced for 2 tables

<table>
<thead>
<tr>
<th>TABLE</th>
<th>DATA PLACEMENT</th>
<th>DATA PLACEMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME</td>
<td>CURRENT KEY</td>
<td>SUGGESTED KEY</td>
</tr>
<tr>
<td>--------</td>
<td>-------------------------</td>
<td>-------------------------</td>
</tr>
<tr>
<td><code>tpch_1024</code>.LINEITEM</td>
<td>L_ORDERKEY, L_LINENUMBER</td>
<td>L_ORDERKEY</td>
</tr>
<tr>
<td><code>tpch_1024</code>.SUPPLIER</td>
<td>S_SUPPKEY</td>
<td>S_NATIONKEY</td>
</tr>
</tbody>
</table>

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 prior to 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.

2.7.3.3 Query Insights

Query Insights provides:
Workload Optimization using Advisor

- 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.

Query Insights Syntax

```sql
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)]

For syntax examples, see Section 2.7.3.4, “Advisor Examples”.

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

- **output**: Defines how Advisor produces output. Permitted values are:
  - normal: The default. Produces summarized output that is sent to stdout and to the heatwave_advisor_report table. (See Section 2.7.3.5, “Advisor Report Table”.)
  - silent: Sends output to the heatwave_advisor_report table only. (See Section 2.7.3.5, “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 Advisor command-line help. See Section 2.7.3.4, “Advisor Examples”.

- **target_schema**: Defines one or more schemas. The list is specified in JSON-array format. If a target schema is not specified, all schemas in HeatWave are considered.

- **exclude_query**: Defines the IDs of queries to exclude. To identify query IDs, query the performance_schema.rpd_query_stats table. For a query example, see Section 2.7.3.4, “Advisor 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 2.7.3.4, “Advisor Examples”.

- **query_insights**: Provides runtimes 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. The default setting is FALSE.

Running Query Insights

For Query Insights to provide runtime data, a query history must be available. Query Insights provides 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:

```sql
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 Query Insights. 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 Query Insights data, see Query Insights Data.

```sql
mysql> CALL sys.heatwave_advisor(JSON_OBJECT("query_insights", TRUE));
<table>
<thead>
<tr>
<th>INITIALIZING HEATWAVE ADVISOR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version: 1.12</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Output Mode: normal</td>
</tr>
<tr>
<td>Excluded Queries: 0</td>
</tr>
<tr>
<td>Target Schemas: All</td>
</tr>
</tbody>
</table>
+-------------------------------+
6 rows in set (0.01 sec)

<p>| ANALYZING LOADED DATA         |
| Total 8 tables loaded in HeatWave for 1 schemas |
| Tables excluded by user: 0 (within target schemas) |</p>
<table>
<thead>
<tr>
<th>SCHEMA</th>
<th>TABLES</th>
<th>COLUMNS</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>tpch128</code></td>
<td>8</td>
<td>61</td>
</tr>
</tbody>
</table>
+---------------------------------------------------------+
8 rows in set (0.02 sec)

<p>| QUERY INSIGHTS |
| Queries executed on Heatwave: 4 |
| Session IDs (as filter): None |</p>
<table>
<thead>
<tr>
<th>QUERY-ID</th>
<th>SESSION-ID</th>
<th>QUERY-STRING</th>
<th>EXEC-RUNTIME</th>
<th>COMMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>32</td>
<td>SELECT COUNT(*) FROM tpch128.LINEITEM</td>
<td>0.628</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>32</td>
<td>SELECT COUNT(*) FROM tpch128.ORDERS</td>
<td>0.114 (est.)</td>
<td>Explain.</td>
</tr>
<tr>
<td>3</td>
<td>32</td>
<td>SELECT COUNT(*) FROM tpch128.ORDERS, tpch128.LINEITEM</td>
<td>5.207 (est.)</td>
<td>Out of memory error during query execution in RAPID.</td>
</tr>
<tr>
<td>4</td>
<td>32</td>
<td>SELECT COUNT(*) FROM tpch128.SUPPLIER, tpch128.LINEITEM</td>
<td>3.478 (est.)</td>
<td>Operation was interrupted by the user.</td>
</tr>
<tr>
<td>TOTAL ESTIMATED: 3</td>
<td>EXEC-RUNTIME: 8.798 sec</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TOTAL EXECUTED: 1</td>
<td>EXEC-RUNTIME: 0.628 sec</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Retrieve detailed query statistics using the query below:
```sql
SELECT log FROM sys.heatwave_advisor_report WHERE stage = "QUERY_INSIGHTS" AND type = "info";
```
For Query Insights **CALL** statements examples that filter results by schema and session ID, see Section 2.7.3.4, “Advisor Examples”.

### Query Insights Data

Query Insights 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 query statistics from the **sys.heatwave_advisor_report** table. For an example of the detailed statistics, see Running Query Insights.
Workload Optimization using Advisor

Query Insights data can be retrieved in machine readable format for use in scripts; see Section 2.7.3.4, “Advisor Examples”. Query Insights data can also be retrieved in JSON format or SQL table format by querying the `sys.heatwave_advisor_report` table. See Section 2.7.3.5, “Advisor Report Table”.

2.7.3.4 Advisor Examples

This section provides Advisor CALL statement examples that you can reference when creating your own statements.

Note

Examples may specify schemas, columns, connection IDs, and other objects that are not be present on your HeatWave instance. Such examples must be modified to work with your data.

- Advisor Command-line Help
- Auto Encoding Examples
- Auto Data Placement Examples
- Query Insights Examples

Advisor Command-line Help

- To view Advisor command-line help:

```
CALL sys.heatwave_advisor(JSON_OBJECT("output","help"));
```

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

Auto Encoding Examples

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

```
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.

```
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"))));
```

Auto Data Placement Examples

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

```
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.

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

- Running the Advisor with the data placement `max_combinations` and `benefit_threshold` parameters. For information about these options, see Auto Data Placement Syntax.

```
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:

```sql
SELECT query_id, LEFT(query_text,160) FROM performance_schema.rpd_query_stats;
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:

```sql
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
  ));
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.

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

### Query Insights Examples

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

```sql
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:

```sql
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:

```sql
CALL sys.heatwave_advisor(JSON_OBJECT("query_insights", TRUE,
"query_session_id", JSON_ARRAY(8)));
```

• This example demonstrates how to invoke the Query Insights Advisor in silent output mode, which is useful if the output is consumed by a script, for example.

```sql
CALL sys.heatwave_advisor(JSON_OBJECT("query_insights",TRUE,"output","silent"));
```

### 2.7.3.5 Advisor Report Table

When running Advisor, detailed output is sent 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

The `heatwave_advisor_report` table can be queried after running Advisor, as in the following examples:

• View Advisor warning information:
Best Practices

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

• View error information in case Advisor stops unexpectedly:
SELECT log FROM sys.heatwave_advisor_report WHERE type = "error";

• View the generated DDL statements for Advisor recommendations:
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 that can be copied and pasted 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.)

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

• Retrieve Query Insights data in JSON format:
SELECT log FROM sys.heatwave_advisor_report WHERE stage = "QUERY_INSIGHTS" AND type = "info";

• Retrieve Query Insights data in SQL table format:
```sql
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 Best Practices

HeatWave best practices are described under the following topics in this section:

• Preparing Data
• Provisioning
• Importing Data into the MySQL DB System
• Inbound Replication
• Loading Data
• Auto Encoding and Auto Data Placement
• Running Queries
• Monitoring
• Reloading Data

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 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 the HeatWave Encoding Advisor to obtain encoding recommendations. See Section 2.7.3.1, “Auto Encoding”.

• Data is partitioned by the table’s 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 2.7.3.2, “Auto Data Placement”.

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 refer to Generating a Node Count Estimate, in the MySQL Database Service Guide.

Perform a cluster size estimate:

• When adding a HeatWave Cluster to a 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.

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. See Importing and Exporting Databases in the MySQL Database Service Guide.

Inbound Replication

• 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. For more information, see Replication, in the MySQL Database Service User Guide.

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

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

However, DDL operations are not permitted on tables defined with a secondary engine. Before replicating a DDL operation from an on-premise instance to a table on the DB System that is defined with a secondary engine, you must set the SECONDARY_ENGINE option to NULL; for example:

```
ALTER TABLE orders SECONDARY_ENGINE = NULL;
```

Setting the SECONDARY_ENGINE option to NULL removes the SECONDARY_ENGINE option from the table definition and unloads the table from HeatWave. To reload the table into HeatWave after the DDL operation is replicated, specify the SECONDARY_LOAD option in an ALTER TABLE statement.

```
ALTER TABLE orders SECONDARY_LOAD;
```

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”.

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, “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”.

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.

```sql
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”.

• **Load tables concurrently**

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

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

Session 2:
```sql
mysql> ALTER TABLE parts SECONDARY_LOAD;
```

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

Session 4:
```sql
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.
Auto Encoding and Auto Data Placement

The Advisor utility analyzes your data and HeatWave query history to provide string column encoding and data placement key recommendations. Consider re-running Advisor for updated recommendations when queries change, when data changes significantly, and after reloading modified tables.

In all cases, re-run your queries before running Advisor. See Section 2.7.3, “Workload Optimization using Advisor”.

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 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:

```sql
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”.

```sql
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
```

```sql
*************************** 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: 100.00
  Extra: Using where
```

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

```sql
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\G
*************************** 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 ***************************
```
Running Queries

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

```sql
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\G
```
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. See Generating a Node Count Estimate, in the MySQL Database 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:
     ```sql
     mysql> SET SESSION rapid_execution_strategy = MIN_MEM_CONSUMPTION;
     ```
   - Using a SET_VAR optimizer hint when issuing a query:
     ```sql
     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, “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 Excluding Table Columns.


### Monitoring

The following monitoring practices are recommended:

- You can 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 MySQL Database Service Metrics, in the MySQL Database Service User Guide.

- 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”.

### 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. Consider setting up a MySQL Database Service event notification or Service Connector Hub notification to let you know when an update has occurred.

- For information about MySQL DB System maintenance, see Maintenance in the MySQL Database Service Guide.

- For information about MySQL Database Service events, see MySQL Database Service Events.

- For information about Service Connector Hub, see Service Connector Hub.

- For table load instructions, see Section 2.2, “Loading Data”.

**Tip**

Instead of loading data into HeatWave manually, consider using the Auto Parallel Load utility, which prepares and loads data for you using an
Supported Data Types

When the HeatWave Cluster is restarted due to a DB System restart. Data in the HeatWave Cluster is lost in this case, requiring reload.

2.9 Supported Data Types

HeatWave supports the following data types. Columns with unsupported data types must be excluded (defined as NOT SECONDARY) before loading a table. See Excluding Table Columns.

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

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

For limitations related to TIMESTAMP data type support, see Section 2.16, “Limitations”.

- String data types:
  - CHAR
  - VARCHAR
  - TEXT-types including TEXT, TINYTEXT, MEDIUMTEXT, and LONGTEXT.
  - ENUM

For ENUM limitations, see Section 2.16, “Limitations”.

- optimized number of parallel load threads. See Section 2.2.3, “Loading Data Using Auto Parallel Load”.
2.10 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

2.11 Supported Functions and Operators

This section describes functions and operators supported by HeatWave.

2.11.1 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>
<thead>
<tr>
<th>Name</th>
<th>VARLEN Support</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVG()</td>
<td></td>
<td>Return the average value of the argument</td>
</tr>
<tr>
<td>COUNT()</td>
<td>Yes</td>
<td>Return a count of the number of rows returned</td>
</tr>
<tr>
<td>COUNT(DISTINCT)</td>
<td></td>
<td>Return the count of a number of different values</td>
</tr>
<tr>
<td>MAX()</td>
<td>Yes</td>
<td>Return the maximum value</td>
</tr>
<tr>
<td>MIN()</td>
<td>Yes</td>
<td>Return the minimum value</td>
</tr>
<tr>
<td>STD()</td>
<td></td>
<td>Return the population standard deviation</td>
</tr>
<tr>
<td>STDDEV()</td>
<td></td>
<td>Return the population standard deviation</td>
</tr>
<tr>
<td>STDDEV_POP()</td>
<td></td>
<td>Return the population standard deviation</td>
</tr>
<tr>
<td>STDDEV_SAMP()</td>
<td></td>
<td>Return the sample standard deviation</td>
</tr>
<tr>
<td>SUM()</td>
<td></td>
<td>Return the sum</td>
</tr>
<tr>
<td>VAR_POP()</td>
<td></td>
<td>Return the population standard variance</td>
</tr>
<tr>
<td>VAR_SAMP()</td>
<td></td>
<td>Return the sample variance</td>
</tr>
<tr>
<td>VARIANCE()</td>
<td></td>
<td>Return the population standard variance</td>
</tr>
</tbody>
</table>
2.11.2 Arithmetic Operators

The following table shows supported arithmetic operators. Arithmetic operators are not supported with variable-length encoded string columns. See Section 2.7.1, “Encoding String Columns”.

Table 2.3 Arithmetic Operators

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>DIV</td>
<td>Integer division</td>
</tr>
<tr>
<td>/</td>
<td>Division operator</td>
</tr>
<tr>
<td>-</td>
<td>Minus operator</td>
</tr>
<tr>
<td>%, MOD</td>
<td>Modulo operator</td>
</tr>
<tr>
<td>+</td>
<td>Addition operator</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication operator</td>
</tr>
<tr>
<td>-</td>
<td>Change the sign of the argument</td>
</tr>
</tbody>
</table>

2.11.3 Cast Functions and Operators

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

- CAST() to DECIMAL.
- CAST() to YEAR.
- CAST() of VARLEN, DATE, DATETIME, TIME, and YEAR column values to DOUBLE.
- CAST() of VARLEN DECIMAL and DOUBLE column values to temporal types such as DATE, DATETIME, TIME, and YEAR.
- CAST() of TIME, DATETIME, TIMESTAMP, and DATE values to REAL, TIME, DATETIME, DATE, and YEAR.
- CAST() of values from DATETIME, TIMESTAMP, DATE, and TIME types to DOUBLE.
- CAST() of temporal types to VARCHAR.
- CAST() of DECIMAL and INTEGER types to DECIMAL. For example:

  \[
  \text{CAST(c1 AS DECIMAL(5,2))}
  \]

- CAST() of INTEGER values to SIGNED and UNSIGNED.
- 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.
- CAST() of FLOAT and DOUBLE values to DECIMAL.

2.11.4 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

<table>
<thead>
<tr>
<th>Name</th>
<th>VARLEN Support</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BETWEEN ... AND ...</td>
<td>Yes</td>
<td>Check whether a value is within a range of values</td>
</tr>
<tr>
<td>COALESCE ()</td>
<td>Yes</td>
<td>Return the first non-NULL argument. Not supported as a JOIN predicate.</td>
</tr>
</tbody>
</table>
## 2.11.5 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**

<table>
<thead>
<tr>
<th>Name</th>
<th>VARLEN Support</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>Yes</td>
<td>Equal operator</td>
</tr>
<tr>
<td>&lt;=&gt;</td>
<td></td>
<td>NULL-safe equal to operator</td>
</tr>
<tr>
<td>&gt;</td>
<td>Yes</td>
<td>Greater than operator</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Yes</td>
<td>Greater than or equal operator</td>
</tr>
<tr>
<td>GREATEST()</td>
<td>Yes</td>
<td>Return the largest argument.</td>
</tr>
<tr>
<td>IN()</td>
<td>Yes</td>
<td>Check whether a value is within a set of values.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>expr IN (value,...)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>comparisions where the expression is a single value and compared values are constants of the same data type and encoding are optimized for performance. For example, the following IN() comparison is optimized:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>SELECT * FROM Customers WHERE Country IN ('Germany', 'France', 'Spain');</td>
</tr>
<tr>
<td>IS</td>
<td></td>
<td>Test a value against a boolean</td>
</tr>
<tr>
<td>IS NOT</td>
<td></td>
<td>Test a value against a boolean</td>
</tr>
<tr>
<td>IS NOT NULL</td>
<td>Yes</td>
<td>NOT NULL value test</td>
</tr>
<tr>
<td>IS NULL</td>
<td>Yes</td>
<td>NULL value test</td>
</tr>
<tr>
<td>ISNULL()</td>
<td>Yes</td>
<td>Test whether the argument is NULL</td>
</tr>
<tr>
<td>LEAST()</td>
<td>Yes</td>
<td>Return the smallest argument.</td>
</tr>
<tr>
<td>&lt;</td>
<td>Yes</td>
<td>Less than operator</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Yes</td>
<td>Less than or equal operator</td>
</tr>
<tr>
<td>LIKE</td>
<td>Yes</td>
<td>Simple pattern matching</td>
</tr>
<tr>
<td>NOT BETWEEN ... AND ...</td>
<td>Yes</td>
<td>Check whether a value is not within a range of values</td>
</tr>
<tr>
<td>!=,&lt;&gt;</td>
<td>Yes</td>
<td>Not equal operator</td>
</tr>
<tr>
<td>NOT IN()</td>
<td>Yes</td>
<td>Check whether a value is not within a set of values</td>
</tr>
<tr>
<td>NOT LIKE</td>
<td>Yes</td>
<td>Negation of simple pattern matching</td>
</tr>
<tr>
<td>STRCMP()</td>
<td>Yes</td>
<td>Compare two strings.</td>
</tr>
</tbody>
</table>
### 2.11.6 Date and Time Functions

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

#### Table 2.6 Date and Time Functions

<table>
<thead>
<tr>
<th>Name</th>
<th>VARLEN Support</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ADDDATE()</strong></td>
<td></td>
<td>Add time values (intervals) to a date value</td>
</tr>
<tr>
<td><strong>ADDTIME()</strong></td>
<td>Yes</td>
<td>Add time</td>
</tr>
<tr>
<td><strong>CONVERT_TZ()</strong></td>
<td></td>
<td>Convert from one time zone to another. Conversion of named timezones is not supported. For a workaround, see Section 2.16, “Limitations”.</td>
</tr>
<tr>
<td><strong>CURDATE()</strong></td>
<td></td>
<td>Return the current date</td>
</tr>
<tr>
<td><strong>CURRENT_DATE()</strong>, <strong>CURRENT_DATE</strong></td>
<td></td>
<td>Synonyms for CURDATE()</td>
</tr>
<tr>
<td><strong>CURRENT_TIME()</strong>, <strong>CURRENT_TIME</strong></td>
<td></td>
<td>Synonyms for CURTIME()</td>
</tr>
<tr>
<td><strong>CURRENT_TIMESTAMP()</strong>, <strong>CURRENT_TIMESTAMP</strong></td>
<td></td>
<td>Synonyms for NOW()</td>
</tr>
<tr>
<td><strong>CURTIME()</strong></td>
<td></td>
<td>Return the current time</td>
</tr>
<tr>
<td><strong>DATE()</strong></td>
<td>Yes</td>
<td>Extract the date part of a date or datetime expression</td>
</tr>
<tr>
<td><strong>DATE_ADD()</strong></td>
<td>Yes</td>
<td>Add time values (intervals) to a date value</td>
</tr>
<tr>
<td><strong>DATE_FORMAT()</strong></td>
<td>Yes</td>
<td>Format date as specified</td>
</tr>
<tr>
<td><strong>DATE_SUB()</strong></td>
<td></td>
<td>Subtract a time value (interval) from a date</td>
</tr>
<tr>
<td><strong>DATEDIFF()</strong></td>
<td>Yes</td>
<td>Subtract two dates</td>
</tr>
<tr>
<td><strong>DAY()</strong></td>
<td></td>
<td>Synonym for DAYOFMONTH()</td>
</tr>
<tr>
<td><strong>DAYNAME()</strong></td>
<td>Yes</td>
<td>Return the name of the weekday</td>
</tr>
<tr>
<td><strong>DAYOFMONTH()</strong></td>
<td>Yes</td>
<td>Return the day of the month (0-31)</td>
</tr>
<tr>
<td><strong>DAYOFWEEK()</strong></td>
<td></td>
<td>Return the weekday index of the argument</td>
</tr>
<tr>
<td><strong>DAYOFYEAR()</strong></td>
<td>Yes</td>
<td>Return the day of the year (1-366)</td>
</tr>
<tr>
<td><strong>EXTRACT()</strong></td>
<td></td>
<td>Extract part of a date</td>
</tr>
<tr>
<td><strong>FROM_UNIXTIME()</strong></td>
<td></td>
<td>Format Unix timestamp as a date</td>
</tr>
<tr>
<td><strong>HOUR()</strong></td>
<td>Yes</td>
<td>Extract the hour</td>
</tr>
<tr>
<td>Name</td>
<td>VARLEN Support</td>
<td>Description</td>
</tr>
<tr>
<td>----------------------</td>
<td>----------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>LAST_DAY()</td>
<td>Yes</td>
<td>Return the last day of the month for the argument</td>
</tr>
<tr>
<td>LOCALTIME(), LOCALTIME</td>
<td></td>
<td>Synonym for NOW()</td>
</tr>
<tr>
<td>LOCALTIMESTAMP, LOCALTIMESTAMP()</td>
<td></td>
<td>Synonym for NOW()</td>
</tr>
<tr>
<td>MAKEDATE()</td>
<td></td>
<td>Create a date from the year and day of year. Supports FLOAT, DOUBLE, INTEGER, and YEAR type values.</td>
</tr>
<tr>
<td>MICROSECOND()</td>
<td>Yes</td>
<td>Return the microseconds from argument</td>
</tr>
<tr>
<td>MINUTE()</td>
<td>Yes</td>
<td>Return the minute from the argument</td>
</tr>
<tr>
<td>MONTH()</td>
<td>Yes</td>
<td>Return the month from the date passed</td>
</tr>
<tr>
<td>MONTHNAME()</td>
<td>Yes</td>
<td>Return the name of the month</td>
</tr>
<tr>
<td>NOW()</td>
<td></td>
<td>Return the current date and time</td>
</tr>
<tr>
<td>QUARTER()</td>
<td>Yes</td>
<td>Return the quarter from a date argument</td>
</tr>
<tr>
<td>SECOND()</td>
<td></td>
<td>Return the second (0-59)</td>
</tr>
<tr>
<td>STR_TO_DATE()</td>
<td>Yes</td>
<td>Convert a string to a date</td>
</tr>
<tr>
<td>SUBTIME()</td>
<td>Yes</td>
<td>Subtract times</td>
</tr>
<tr>
<td>TIME()</td>
<td>Yes</td>
<td>Extract the time portion of the expression passed</td>
</tr>
<tr>
<td>TIME_FORMAT()</td>
<td>Yes</td>
<td>Format as time.</td>
</tr>
<tr>
<td>TIME_TO_SEC()</td>
<td></td>
<td>Return the argument converted to seconds</td>
</tr>
<tr>
<td>TIMESTAMP()</td>
<td>Yes</td>
<td>With a single argument, this function returns the date or datetime expression; with two arguments, the sum of the arguments</td>
</tr>
<tr>
<td>TIMESTAMPADD()</td>
<td></td>
<td>Add an interval to a datetime expression</td>
</tr>
<tr>
<td>TIMESTAMPDIFF()</td>
<td>Yes</td>
<td>Subtract an interval from a datetime expression</td>
</tr>
<tr>
<td>TO_DAYS()</td>
<td>Yes</td>
<td>Return the date argument converted to days</td>
</tr>
<tr>
<td>TO_SECONDS()</td>
<td>Yes</td>
<td>Return the date or datetime argument converted to seconds since Year 0</td>
</tr>
<tr>
<td>UNIX_TIMESTAMP()</td>
<td></td>
<td>Return a Unix timestamp</td>
</tr>
<tr>
<td>WEEK()</td>
<td>Yes</td>
<td>Return the week number. Restrictions apply. See Section 2.16, “Limitations”</td>
</tr>
<tr>
<td>WEEKDAY()</td>
<td></td>
<td>Return the weekday index</td>
</tr>
</tbody>
</table>
### Logical Operators

<table>
<thead>
<tr>
<th>Name</th>
<th>VARLEN Support</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>WEEKOFYEAR()</td>
<td></td>
<td>Return the calendar week of the date (1-53)</td>
</tr>
<tr>
<td>YEAR()</td>
<td>Yes</td>
<td>Return the year</td>
</tr>
<tr>
<td>YEARWEEK()</td>
<td></td>
<td>Return the year and week</td>
</tr>
</tbody>
</table>

#### 2.11.7 Logical Operators

The following table shows supported logical operators.

**Table 2.7 Logical Operators**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AND, &amp;&amp;</td>
<td>Logical AND</td>
</tr>
<tr>
<td>NOT, !</td>
<td>Negates value</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>XOR</td>
<td>Logical XOR</td>
</tr>
</tbody>
</table>

#### 2.11.8 Mathematical Functions

The following table shows supported mathematical functions. Mathematical functions are not supported with variable-length or dictionary-encoded columns.

**Table 2.8 Mathematical Functions**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS()</td>
<td>Return the absolute value</td>
</tr>
<tr>
<td>ACOS()</td>
<td>Return the arc cosine</td>
</tr>
<tr>
<td>ASIN()</td>
<td>Return the arc sine</td>
</tr>
<tr>
<td>ATAN()</td>
<td>Return the arc tangent</td>
</tr>
<tr>
<td>CEIL()</td>
<td>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().</td>
</tr>
<tr>
<td>CEILING()</td>
<td>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().</td>
</tr>
<tr>
<td>COS()</td>
<td>Return the cosine</td>
</tr>
<tr>
<td>COT()</td>
<td>Return the cotangent</td>
</tr>
<tr>
<td>DEGREES()</td>
<td>Convert radians to degrees</td>
</tr>
<tr>
<td>EXP()</td>
<td>Raise to the power of</td>
</tr>
<tr>
<td>FLOOR()</td>
<td>Return the largest integer value not greater than the argument. The function is not applied to BIGINT values. The input value is returned.</td>
</tr>
<tr>
<td>LN()</td>
<td>Return the natural logarithm of the argument</td>
</tr>
<tr>
<td>LOG()</td>
<td>Return the natural logarithm of the first argument</td>
</tr>
<tr>
<td>LOG10()</td>
<td>Return the base-10 logarithm of the argument</td>
</tr>
<tr>
<td>MOD()</td>
<td>Return the remainder</td>
</tr>
<tr>
<td>RADIANS()</td>
<td>Return argument converted to radians</td>
</tr>
<tr>
<td>ROUND()</td>
<td>Round the argument</td>
</tr>
</tbody>
</table>
### 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 2.9 String Functions and Operators**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASCII()</td>
<td>Return numeric value of left-most character</td>
</tr>
<tr>
<td>BIT_LENGTH()</td>
<td>Return length of argument in bits</td>
</tr>
<tr>
<td>CHAR_LENGTH()</td>
<td>Return number of characters in argument</td>
</tr>
<tr>
<td>CONCAT()</td>
<td>Return concatenated string</td>
</tr>
<tr>
<td>CONCAT_WS()</td>
<td>Return concatenated with separator</td>
</tr>
<tr>
<td>FIND_IN_SET()</td>
<td>Index (position) of first argument within second argument</td>
</tr>
<tr>
<td>FORMAT()</td>
<td>Return a number formatted to specified number of decimal places. <em>Does not support variable-length-encoded columns.</em></td>
</tr>
<tr>
<td>FROM_BASE64()</td>
<td>Decode base64 encoded string and return result</td>
</tr>
<tr>
<td>GREATEST()</td>
<td>Return the largest argument. Not supported with temporal columns.</td>
</tr>
<tr>
<td>HEX()</td>
<td>Hexadecimal representation of decimal or string value</td>
</tr>
<tr>
<td>INSERT()</td>
<td>Return the index of the first occurrence of substring</td>
</tr>
<tr>
<td>INSTR()</td>
<td>Return the index of the first occurrence of substring</td>
</tr>
<tr>
<td>LEAST()</td>
<td>Return the smallest argument. Not supported with temporal columns.</td>
</tr>
<tr>
<td>LEFT()</td>
<td>Return the leftmost number of characters as specified</td>
</tr>
<tr>
<td>LENGTH()</td>
<td>Return the length of a string in bytes</td>
</tr>
<tr>
<td>LIKE</td>
<td>Simple pattern matching</td>
</tr>
<tr>
<td>LOCATE()</td>
<td>Return the position of the first occurrence of substring</td>
</tr>
<tr>
<td>LOWER()</td>
<td>Return the argument in lowercase</td>
</tr>
<tr>
<td>LPAD()</td>
<td>Return the string argument, left-padded with the specified string</td>
</tr>
<tr>
<td>LTRIM()</td>
<td>Remove leading spaces</td>
</tr>
<tr>
<td>NOT LIKE</td>
<td>Negation of simple pattern matching</td>
</tr>
<tr>
<td>OCTET_LENGTH()</td>
<td>Synonym for <code>LENGTH()</code></td>
</tr>
</tbody>
</table>
### Window Functions

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ORD()</td>
<td>Return character code for leftmost character of the argument</td>
</tr>
<tr>
<td>POSITION()</td>
<td>Synonym for LOCATE()</td>
</tr>
<tr>
<td>REPEAT()</td>
<td>Repeat a string the specified number of times</td>
</tr>
<tr>
<td>QUOTE()</td>
<td>Escape the argument for use in an SQL statement</td>
</tr>
<tr>
<td>REGEXP()</td>
<td>Whether string matches regular expression</td>
</tr>
<tr>
<td>REGEXP_LIKE()</td>
<td>Whether string matches regular expression</td>
</tr>
<tr>
<td>REGEXP_REPLACE()</td>
<td>Replace substrings matching regular expression. Supports up to three arguments.</td>
</tr>
<tr>
<td>REGEXP_SUBSTR()</td>
<td>Return substring matching regular expression. Supports up to three arguments.</td>
</tr>
<tr>
<td>REPLACE()</td>
<td>Replace occurrences of a specified string</td>
</tr>
<tr>
<td>REVERSE()</td>
<td>Reverse the characters in a string</td>
</tr>
<tr>
<td>RIGHT()</td>
<td>Return the specified rightmost number of characters</td>
</tr>
<tr>
<td>RLIKE</td>
<td>Whether string matches regular expression</td>
</tr>
<tr>
<td>RPAD()</td>
<td>Append string the specified number of times</td>
</tr>
<tr>
<td>RTRIM()</td>
<td>Remove trailing spaces</td>
</tr>
<tr>
<td>STRCMP()</td>
<td>Compare two strings</td>
</tr>
<tr>
<td>SUBSTR()</td>
<td>Return the substring as specified</td>
</tr>
<tr>
<td>SUBSTRING()</td>
<td>Return the substring as specified</td>
</tr>
<tr>
<td>SUBSTRING_INDEX()</td>
<td>Return a substring from a string before the specified number of occurrences of the delimiter</td>
</tr>
<tr>
<td>TO_BASE64()</td>
<td>Return the argument converted to a base-64 string</td>
</tr>
<tr>
<td>TRIM()</td>
<td>Remove leading and trailing spaces</td>
</tr>
<tr>
<td>UNHEX()</td>
<td>Return a string containing hex representation of a number</td>
</tr>
<tr>
<td>UPPER()</td>
<td>Convert to uppercase</td>
</tr>
</tbody>
</table>

#### 2.11.10 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:
  - **COUNT()**
  - **SUM()**
  - **AVG()**
2.12 String Column Encoding Reference

HeatWave supports two string column encoding types:

- Section 2.12.1, “Variable-length Encoding”
- Section 2.12.2, “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”.

2.12.1 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.

2.12.1.1 VARLEN Supported Expressions, Filters, Functions, and Operators

For supported functions and operators, refer to Section 2.11, “Supported Functions and Operators”.

VARLEN Supported Filters

- Column-to-column filters, excluding the <==> filter. Both columns must be VARLEN-encoded.
  - Column-to-column filters must use columns that are encoded with the same character set and collation.
- Column-to-constant filters, excluding the <==> filter.
  - The character set and collation of the constant variable must match the character set and collation of the constant.

VARLEN Supported Relational Operators

- GROUP BY
- JOIN
- LIMIT
- ORDER BY
2.12.1.2 VARLEN Encoding Limits

- For information about HeatWave column limits and how they relate to VARLEN-encoded columns, see Section 2.12.3, "Column Limits".
- Only expressions with non-boolean types are supported.

2.12.1.3 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.

2.12.1.4 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.

2.12.2 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.

2.12.3 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 900.

• 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.

2.13 Monitoring

This section provides queries that you can use to monitor HeatWave.

Note

You can also monitor HeatWave from the OCI Console using MySQL Database Service Metrics. For more information, see MySQL Database Service Metrics.

Queries are organized into the following categories:

• HeatWave Node Status
• HeatWave Memory Usage
• Data Load Progress and Status
• Change Propagation
• Query Execution
• Query History and Statistics
• Scanned Data

HeatWave Node Status

To view the status of each HeatWave node:

```sql
mysql> SELECT ID, STATUS FROM performance_schema.rpd_nodes;
```

<table>
<thead>
<tr>
<th>ID</th>
<th>STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>AVAIL_RNSTATE</td>
</tr>
<tr>
<td>1</td>
<td>AVAIL_RNSTATE</td>
</tr>
</tbody>
</table>

For column descriptions, see Section 5.4, “The rpd_nodes Table”.

HeatWave Memory Usage

To view the memory usage for each HeatWave node:

```sql
mysql> SELECT ID, MEMORY_USAGE, MEMORY_TOTAL, BASEREL_MEMORY_USAGE FROM performance_schema.rpd_nodes;
```

<table>
<thead>
<tr>
<th>ID</th>
<th>MEMORY_USAGE</th>
<th>MEMORY_TOTAL</th>
<th>BASEREL_MEMORY_USAGE</th>
</tr>
</thead>
</table>
For column descriptions, see Section 5.4, “The rpd_nodes Table”.

Data Load Progress and Status

- 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.

  ```sql
  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:

  ```sql
  USE performance_schema;
  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 5.8, “The rpd_tables Table”.

- To view the amount of data loaded in HeatWave for a table, in bytes:

  ```sql
  USE performance_schema;
  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:

  ```sql
  USE performance_schema;
  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 |
Change Propagation

- To determine if change propagation is enabled globally, query the `rapid_change_propagation_status` variable:

```sql
mysql> SELECT VARIABLE_VALUE FROM performance_schema.global_status
WHERE VARIABLE_NAME = 'rapid_change_propagation_status';
```

- 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.

```sql
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';
```

Query Execution

- To view the number of queries offloaded to the HeatWave Cluster for execution since the last time HeatWave Cluster was started:

```sql
mysql> SELECT VARIABLE_VALUE
FROM performance_schema.global_status
WHERE VARIABLE_NAME = 'rapid_query_offload_count';
```

- From MySQL 8.0.29, 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:

```sql
mysql> SELECT CURRENT_SCHEMA, LEFT(DIGEST_TEXT, 50), EXECUTION_ENGINE
FROM performance_schema.events_statements_history
WHERE CURRENT_SCHEMA='tpch';
```
From MySQL 8.0.29, 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;
+----------------------+
<table>
<thead>
<tr>
<th>SUM(COUNT_SECONDARY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>25</td>
</tr>
</tbody>
</table>
```

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'
```

<table>
<thead>
<tr>
<th>TOTAL_EXECUTIONS: 25</th>
<th>PRIMARY_ENGINE: 5</th>
<th>SECONDARY_ENGINE: 20</th>
</tr>
</thead>
</table>

**Query History and Statistics**

To view the HeatWave query history including query start time, end time, and wait time in the scheduling queue, as discussed in Auto Scheduling.

```
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)
```

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.

```sql
mysql> SELECT COUNT(*) FROM performance_schema.rpd_query_stats;
+----------+
| count(*) |
+----------+
|     1000 |
+----------+
```

To view query IDs for the first and last successfully executed queries:

```sql
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:

```sql
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 |
+------------+---------+-------------+----------------------------+
```

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:

```sql
mysql> SHOW GLOBAL STATUS LIKE 'hw_data_scanned';
+-----------------+-------+
| Variable_name   | Value |
+-----------------+-------+
```

---

Scanned Data

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:
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:

```sql
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;
```

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:

```sql
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;
```

```sql
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;
```

2.14 Troubleshooting

- **Problem**: Queries are not offloaded.
  - **Solution A**: Your query contains an unsupported predicate, function, operator, or has encountered some other limitation. See **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.
   ```sql
   SET SESSION use_secondary_engine=OFF;
   ```

2. Run the query using `EXPLAIN`.
   ```sql
   EXPLAIN select_query;
   ```

3. Query the `Last_query_cost` status variable. If the value is less than 100000.00000, the query cannot be offloaded.
   ```sql
   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:

   ```sql
   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 5.8, “The rpd_tables Table”.

   Alternatively, run the following statement:
   ```sql
   mysql> ALTER TABLE tbl_name SECONDARY_LOAD;
   ```

   The following error is reported if the table is already loaded:
   ```sql
   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:
   ```sql
   mysql> SHOW GLOBAL STATUS LIKE 'rapid_plugin_bootstrapped';
   +------------------------+--------+
   | Variable_name          | Value  |
   +------------------------+--------+
   | rapid_plugin_bootstrapped | YES    |
   ```
Troubleshooting

See for Chapter 4, System and Status Variables for rapid_plugin_bootstrapped status values.

If the HeatWave Cluster has failed, restart it in the Console and reload the data if necessary. The HeatWave recovery mechanism should reload the data automatically.

- **Problem:** You cannot alter the table definition to exclude a column, define a string column encoding, or define data placement keys.

  **Solution:** Column attributes must be defined before or at the same that you define a secondary engine for a table. Defining a column attribute is not possible after a table is defined with a secondary engine, as DDL operations are not permitted on tables defined with a secondary engine. If you need to perform a DDL operation on a table that is defined with a secondary engine, remove the SECONDARY ENGINE option first:

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

  For more information, see Section 2.4, “Modifying Tables”.

- **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 prior to executing the query:

  ```
  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. See Generating a Node Count Estimate, in the MySQL Database Service Guide.

  Avoid or rewrite queries that produce a Cartesian product. For more information, see Running Queries.

- **Problem:** A table load operation fails with “ERROR HY000: Error while running parallel scan.”

  **Solution:** TEXT-type values larger than 65532 bytes are rejected during SECONDARY_LOAD operations. Reduce the size of TEXT-type values to less than 65532 bytes or exclude the column before loading the table. See 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 utility’s output, but those load statements should be avoided or used 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:** When attempting to retrieve generated Auto Parallel Load or Advisor DDL statements, an 
error message indicates that the **heatwave_advisor_report** or **heatwave_load_report** table 
does not exist. For example:

```sql
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' doesn't 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' doesn't 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.

### 2.15 Metadata Queries

This section provides Information Schema and Performance Schema queries that you can use to 
retrieve HeatWave metadata.

#### Note

For queries that you can use to monitor HeatWave node status, memory usage, 
data loading, change propagation, and queries, see [Section 2.13, “Monitoring”](#).

Metadata queries are organized into the following categories.

- **Secondary Engine Definitions**
- **Excluded Columns**
- **String Column Encoding**
- **Data Placement**

**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.

```sql
mysql> SELECT TABLE_SCHEMA, TABLE_NAME, CREATE_OPTIONS FROM INFORMATION_SCHEMA.TABLES 
WHERE CREATE_OPTIONS LIKE '%SECONDARY_ENGINE%' AND TABLE_SCHEMA LIKE 'tpch';
```

<table>
<thead>
<tr>
<th>TABLE_SCHEMA</th>
<th>TABLE_NAME</th>
<th>CREATE_OPTIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>--------------</td>
<td>------------</td>
<td>----------------</td>
</tr>
</tbody>
</table>
You can also view create options for an individual table using **SHOW CREATE TABLE**.

**Note**  
You can use the `show_create_table_skip_secondary_engine` variable to exclude the `SECONDARY ENGINE` clause from **SHOW CREATE TABLE** output, and from **CREATE TABLE** statements dumped by the `mysqldump` utility.  
`mysqldump` also provides a `--show-create-skip-secondary-engine` option that enables the `show_create_table_skip_secondary_engine` system variable for the duration of the dump operation. It may be necessary to exclude the `SECONDARY ENGINE` option from **CREATE TABLE** statements when creating a dump file, as DDL operations cannot be performed on tables defined with a secondary engine.

### 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. For example:

```
mysql> SELECT COLUMN_NAME, EXTRA FROM INFORMATION_SCHEMA.COLUMNS
WHERE TABLE_NAME LIKE 't1' AND EXTRA LIKE '%NOT SECONDARY%';
```

<table>
<thead>
<tr>
<th>COLUMN_NAME</th>
<th>EXTRA</th>
</tr>
</thead>
<tbody>
<tr>
<td>O_COMMENT</td>
<td>NOT SECONDARY</td>
</tr>
</tbody>
</table>

You can also view columns defined as `NOT SECONDARY` for an individual table using **SHOW CREATE TABLE**.

### 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%';
```

<table>
<thead>
<tr>
<th>COLUMN_NAME</th>
<th>COLUMN_COMMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>O_CLERK</td>
<td>RAPID_COLUMN=ENCODING=SORTED</td>
</tr>
<tr>
<td>O_ORDERPRIORITY</td>
<td>RAPID_COLUMN=ENCODING=SORTED</td>
</tr>
<tr>
<td>O_ORDERSTATUS</td>
<td>RAPID_COLUMN=ENCODING=SORTED</td>
</tr>
<tr>
<td>O_CLERK</td>
<td>RAPID_COLUMN=ENCODING=SORTED</td>
</tr>
<tr>
<td>O_ORDERPRIORITY</td>
<td>RAPID_COLUMN=ENCODING=SORTED</td>
</tr>
<tr>
<td>O_ORDERSTATUS</td>
<td>RAPID_COLUMN=ENCODING=SORTED</td>
</tr>
</tbody>
</table>

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
```

70
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:

```sql
mysql> SELECT COLUMN_NAME, COLUMN_COMMENT FROM INFORMATION_SCHEMA.COLUMNS
WHERE TABLE_NAME LIKE 'orders' AND COLUMN_COMMENT LIKE '%DATA_PLACEMENT_KEY%';
```

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:

```sql
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;
```

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:

```sql
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);
```

71
2.16 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 2.9, “Supported Data Types”, Section 2.11, “Supported Functions and Operators”, and Section 2.10, “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.

- **Functions:**
  - Bit functions and operators.
  - `CAST()` as `SIGNED` and `UNSIGNED` on temporal values. For supported `CAST()` operations, see Section 2.11.3, “Cast Functions and Operators”.
  - `COALESCE()` as a `JOIN` predicate.
  - `CONVERT_TZ()` with named timezones such as 'MET' or 'Europe/Amsterdam'. Only datetime values are supported. A query that uses named timezones can be rewritten using equivalent datetime values; for example, `SELECT CONVERT_TZ(O_ORDERDATE, 'UTC','EST')` FROM `tpch.orders` can be rewritten as `SELECT CONVERT_TZ(O_ORDERDATE, '+00:00','-05:00')` FROM `tpch.orders`.
- **MySQL Enterprise Data Masking and De-Identification and De-Identification Functions**
- Full-text search functions.
- XML, JSON, Spatial, and other domain specific functions.
- Encryption and compression functions.
- Loadable Functions.
  - `GREATEST()` and `LEAST()` functions with temporal data type columns.
  - A `CASE` control flow operator or `IF()` function that contains columns not within an aggregation function and not part of the `GROUP BY` key.
- Date functions on the `YEAR` type.
- String functions and operators on columns that are not `VARLEN`-encoded. See Section 2.7.1, “Encoding String Columns”.
- In some cases, comparison functions with a mixture of string and non-string arguments due to HeatWave returning incorrect results.
- The `AVG()` aggregate function with enumeration and temporal data types.
- The following aggregate functions with enumeration, string, and temporal data types:
  - `STD()`
  - `STDDEV()`
  - `STDDEV_POP()`
  - `STDDEV_SAMP()`
  - `SUM()`
• VAR_POP()

• VAR_SAMP()

• VARIANCE()

With the exception of SUM(), the same aggregate functions within a semi-join predicate due to the undeterministic nature of floating-point results and potential mismatches. For example, the following use is not supported:

```sql
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 2.9, “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.

• Data types:

  • Spatial data types. See Spatial Data Types.

  • Decimal values with a precision greater than 18 in expression operations, with the exception of ABS() expression operations.

  • ENUM type columns as part of a UNION or 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.

• Character sets and collations:

  • The gb18030_chinese_ci character set and collation.
Limitations

- Variables:
  - `time_zone` and `timestamp` variable settings are not passed to HeatWave when queries are offloaded.

  - A `time_zone` setting other than `SYSTEM (+00:00)` is not supported when propagating changes to tables containing `TIMESTAMP` columns. Propagating changes in this scenario causes incorrect data to be stored in `TIMESTAMP` columns, leading to incorrect query results.

  - The `sql_select_limit` as a global variable. It is only supported as a session variable.

- JOIN types:
  - Antijoins, with the exception of supported `IN` and `EXISTS` antijoin variants listed below.

  - Implicit casting (query cast injection) of the `YEAR` type to other types. It can only be joined with itself.

  - Implicit casting (query cast injection) of the `VARCHAR` type to types other than `DATETIME`, `TIMESTAMP`, and `DATE`.

  - Temporal to numeric implicit casting (query cast injection). Therefore, temporal types cannot be joined with numeric types.

  - `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.
- 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.

- **SQL modes:**
  - Most non-default MySQL DB System SQL modes. For a list of supported SQL modes, see Section 2.10, "Supported SQL Modes".

- **Other:**
  - 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.
    - `COUNT(NULL)` in cases where it is used as an input argument for non-aggregate operators.
  - `UNION ALL` queries with an `ORDER BY` or `LIMIT` clause, between different column types, between dictionary-encoded columns, or between `ENUM` columns.
  - `UNION` queries with or without an `ORDER BY` or `LIMIT` clause, between different column types, between dictionary-encoded columns, or between `ENUM` columns.
  - `UNION` and `UNION ALL` subqueries with or without an `ORDER BY` or `LIMIT` clause, between different column types, 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`.
  - Type conversion on relational data. For example, `SELECT CONCAT(2, L_COMMENT) from LINEITEM;` is not supported.
  - Queries with an impossible `WHERE` condition (queries known to have an empty result set). For example, the following query is not offloaded:
    ```sql
    SELECT AVG(cl) AS value FROM t1 WHERE cl IS NULL;
    ```
  - Querying of `YEAR` type data using expressions and other functions. For example, the following queries are not offloaded:
    ```sql
    SELECT YEAR(d) + 1 FROM t1;
    ```
SELECT YEAR(d) + c1 FROM t1; # where c1 is an integer column

- String operations involving columns with different collations.
- Explicit partition selection. See Partition Selection.
- 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 multi-query transaction.
- Materialized views. Only nonmaterialized views are supported. See Using Views.
- `UNION` queries executed on a view.

- 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”.)

- `SET timezone = timezone`, with the `timezone` value specified as an offset from UTC in the form of [H]H:MM and prefixed with a + or - is supported only by the `UNIX_TIMESTAMP()` and `FROM_UNIXTIME()` functions. Named time zones are not supported. For information about time zone offsets, see MySQL Server Time Zone Support.

- A `time_zone` setting other than `SYSTEM (+00:00)` is not supported when propagating changes to tables containing `TIMESTAMP` columns. Propagating changes in this scenario causes incorrect data to be stored in `TIMESTAMP` columns, leading to incorrect query results.

- 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:

```sql
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.

- `UNION ALL` statements inside derived table expressions or common table expressions. For example, the following query, which defines a `UNION ALL` clause within `WITH` clause, is not supported.

```sql
WITH
c1 AS (SELECT a, b FROM table1 UNION ALL (SELECT a, b FROM table2)),
c2 AS (SELECT c, d FROM table2)
SELECT b, d FROM cte1 JOIN cte2 WHERE cte1.a = cte2.c;
```

- Recursive common table expressions.

- Operations involving `ALTER TABLE` such 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, refer to the Health Monitor documentation, in the MySQL Database Service Guide.
Chapter 3 HeatWave ML

Table of Contents

3.1 Before You Begin ......................................................................................... 81
3.2 Getting Started ............................................................................................. 82
3.3 Preparing Data ............................................................................................... 82
3.4 Training a Model ............................................................................................ 85
3.5 Predictions ...................................................................................................... 86
3.6 Explanations .................................................................................................. 88
3.7 Managing Models ........................................................................................... 89
  3.7.1 The Model Catalog .................................................................................. 89
  3.7.2 Loading Models ....................................................................................... 91
  3.7.3 Unloading Models ................................................................................... 91
  3.7.4 Viewing Models ....................................................................................... 92
  3.7.5 Scoring Models ....................................................................................... 92
  3.7.6 Model Explanations ............................................................................... 93
  3.7.7 Model Handles ....................................................................................... 93
  3.7.8 Deleting Models ..................................................................................... 94
  3.7.9 Sharing Models ...................................................................................... 94
3.8 HeatWave ML Routines .................................................................................. 95
  3.8.1 ML_TRAIN .............................................................................................. 95
  3.8.2 ML_PREDICT_ROW .............................................................................. 96
  3.8.3 ML_PREDICT_TABLE ............................................................................ 97
  3.8.4 ML_EXPLAIN_ROW ............................................................................ 97
  3.8.5 ML_EXPLAIN_TABLE ......................................................................... 98
  3.8.6 ML_SCORE .......................................................................................... 99
  3.8.7 ML_MODEL_LOAD .............................................................................. 100
  3.8.8 ML_MODEL_UNLOAD ....................................................................... 101
3.9 Supported Data Types ................................................................................... 101
3.10 Monitoring .................................................................................................... 102
3.11 HeatWave ML Error Messages ..................................................................... 103
3.12 Limitations ................................................................................................... 107

HeatWave ML makes it easy to use machine learning, whether you are a novice user or an experienced ML practitioner. You provide the data, and HeatWave ML 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 ML explanations help you understand how predictions are made, such as which features of a dataset contribute most to a prediction.

Supervised Learning

HeatWave ML 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 an individual’s income (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 ML 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.

Ease of Use

HeatWave ML is purpose-built for ease of use. It requires no machine learning expertise, specialized tools, or algorithms. With HeatWave ML and a set of training data, you can train a predictive machine learning (ML) 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 Oracle AutoML.

You can use a model created by ML_TRAIN with other HeatWave ML 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 ML operations are initiated by running CALL or SELECT statements, which can be easily integrated into your applications. HeatWave ML routines reside in the MySQL sys schema and can be run from any MySQL client or application that is connected to a DB System with a HeatWave Cluster. HeatWave ML 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 ML, there is no need to move or reformat your data. Data and machine learning models never leave the MySQL Database Service, which saves you time and effort while keeping your data and models secure.

HeatWave ML Workflow

The general HeatWave ML workflow is described below:

1. When the ML_TRAIN routine is called, HeatWave ML 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 3.4, “Training a Model”.

2. HeatWave ML 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 3.7.1, “The Model Catalog”.

3. HeatWave ML ML_PREDICT_* and ML_EXPLAIN_* routines use the trained model to generate predictions and explanations on test or unseen data. See Section 3.5, “Predictions”, and Section 3.6, “Explanations”.
Oracle AutoML

4. Predictions and explanations are returned to the 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 3.7.5, “Scoring Models”.

**Note**


Oracle AutoML

The HeatWave ML `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 ML `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.


3.1 Before You Begin

Before you begin using HeatWave ML, the following is assumed:
• 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 Database Service.

• Your MySQL DB System has an operational HeatWave Cluster. If not, complete the steps described in Adding a HeatWave Cluster.

• The MySQL account that you will use to 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 3.12, “Limitations”.

• The MySQL account that will use HeatWave ML has been granted the following privileges:

  • SELECT and ALTER privileges on the schema that contains the machine learning datasets; for example:

    ```sql
    GRANT SELECT, ALTER ON schema_name.* TO 'user_name'@'%;
    ```

  • SELECT and EXECUTE on the MySQL sys schema where HeatWave ML routines reside; for example:

    ```sql
    GRANT SELECT, EXECUTE ON sys.* TO 'user_name'@'%;
    ```

### 3.2 Getting Started

Once you have access to a DB System with a HeatWave Cluster, and you have obtained the MySQL user privileges described in Section 3.1, “Before You Begin”, you can start using HeatWave ML.

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 3.3, “Preparing Data”.

2. Train a machine learning model. See Section 3.4, “Training a Model”.

3. Make predictions with test data using a trained model. See Section 3.5, “Predictions”.

4. Run explanations on test data using a trained model to understand how predictions are made. See Section 3.6, “Explanations”.

5. Score your machine learning model to assess its reliability. See Section 3.7.5, “Scoring Models”.

6. View a model explanation to understand how the model makes predictions. See Section 3.7.6, “Model Explanations”.

Alternatively, you can jump ahead to the Iris Data Set Machine Learning Quickstart, which provides a quick run-through of HeatWave ML capabilities using a simple, well-known machine learning data set. See Section 6.3, “Iris Data Set Machine Learning Quickstart”.

### 3.3 Preparing Data

HeatWave ML works with labeled an unlabeled data. Labeled data is used to train and score machine learning models. Unlabelled data is required when generating predictions and explanations.

#### Labeled Data

Labeled data has feature columns and a target column (the label), as illustrated in the following diagram:
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 to 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 to as the *validation dataset*.

**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 *Example Data*, and *Section 6.3, “Iris Data Set Machine Learning Quickstart”*.

**General Data Requirements**

General requirements for HeatWave ML data include the following:

- Each dataset must reside in a single table on the MySQL DB System. HeatWave ML 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.

- Tables used with HeatWave ML must not exceed 10 GB, 100 million rows, or 900 columns.
- Table columns must use supported data types. For supported data types and recommendations for how to handle unsupported types, see Section 3.9, “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.

**Example Data**

Examples in this guide use the Census Income Data Set.


**Note**

Census Income Data Set examples demonstrate classification training and inference. HeatWave ML 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:

   ```sql
   CREATE SCHEMA heatwaveml_bench;
   USE heatwaveml_bench;
   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));
   ```
CREATE TABLE `census_test` LIKE `census_train`;


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 preprocessing 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> util.importTable("census_train.csv",{table: "census_train", dialect: "csv-unix", skipRows:1})
   MySQL> util.importTable("census_test.csv",{table: "census_test", dialect: "csv-unix", skipRows:1})
   ```

6. Create a validation table:

   ```
   CREATE TABLE `census_validate` LIKE `census_test`;
   INSERT INTO `census_validate` SELECT * FROM `census_test`;
   ```

7. Modify the `census_test` table to remove the target `revenue` column:

   ```
   ALTER TABLE `census_test` DROP COLUMN `revenue`;
   ```

### 3.4 Training a Model

The `ML_TRAIN` routine, when run on a training dataset, produces a trained machine learning (ML) model.

`ML_TRAIN` supports training of classification and regression models. A classification model is for predicting discrete values. A regression model is for predicting continuous values. The model type is specified by the `ML_TRAIN` task parameter.

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, the specified `ML_TRAIN` parameters, and the size of the HeatWave Cluster. HeatWave ML supports tables up to 10 GB in size with a maximum of 100 million rows and 900 columns.

`ML_TRAIN` stores the machine learning model in the `MODEL_CATALOG` table. See Section 3.7.1, “The Model Catalog”. 
For **ML_TRAIN** parameter and option descriptions, see Section 3.8.1, “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 Example Data.

The following example runs **ML_TRAIN** on the heatwaveml_bench.census_train training dataset:

```
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. Supported types are classification (the default) and regression. NULL can be specified in place of the JSON_OBJECT if you intend to use the default classification task type. When using the regression task type, only a numeric target column is permitted.
- **@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 (e.g., @my_model).

After **ML_TRAIN** trains a model, the model is stored in the user’s 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**, you can specify the session variable (e.g., @census_model) in place of the model handle in other HeatWave ML 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 3.7.7, “Model Handles”.

The quality and reliability of a trained model can be assessed using the **ML_SCORE** routine. For more information, see Section 3.7.5, “Scoring Models”.

### 3.5 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.

#### 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 3.8.2, “ML_PREDICT_ROW”.

Before running **ML_PREDICT_ROW**, ensure that the model you want to use is loaded; for example:
CALL sys.ML_MODEL_LOAD(@census_model, NULL);

For more information about loading models, see Section 3.7.2, “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:

SET @row_input = JSON_OBJECT(
  "age", 25,
  "workclass", "Private",
  "fnlwgt", 226802,
  "education", "11th",
  "education-num", 7,
  "marital-status", "Never-married",
  "occupation", "Machine-op-inspt",
  "relationship", "Own-child",
  "race", "Black",
  "sex", "Male",
  "capital-gain", 0,
  "capital-loss", 0,
  "hours-per-week", 40,
  "native-country", "United-States";
SELECT sys.ML_PREDICT_ROW(@row_input, @census_model);

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 3.8.2, "ML_PREDICT_ROW".

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 ML_PREDICT_TABLE parameter descriptions, see Section 3.8.3, “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.

Before running ML_PREDICT_TABLE, ensure that the model you want to use is loaded; for example:

CALL sys.ML_MODEL_LOAD(@census_model, NULL);

For more information about loading models, see Section 3.7.2, “Loading Models”.

The following example creates a table with 10 rows of unlabeled test data and generates predictions for that table:

CREATE TABLE heatwaveml_bench.census_test_subset AS SELECT * FROM heatwaveml_bench.census_test LIMIT 10;

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.
Explanations

• **@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:

```
SELECT * FROM heatwaveml_bench.census_predictions;
```

**ML_PREDICT_TABLE** populates the output table with predictions and the features used to make each prediction.

3.6 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.

**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.

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_*** routines limit explanations to the 100 most relevant features.

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 3.8.4, "ML_EXPLAIN_ROW".

Before running **ML_EXPLAIN_ROW**, ensure that the model you want to use is loaded; for example:

```
CALL sys.ML_MODEL_LOAD(@census_model, NULL);
```

For more information about loading models, see Section 3.7.2, "Loading Models".

The following example generates explanations for a single row of unlabeled data, which is assigned to a **@row_input** session variable:

```
SET @row_input = JSON_OBJECT(
    "age", 25,
    "workclass", "Private",
    "fnlwgt", 226802,
    "education", "11th",
    "education-num", 7,
    "marital-status", "Never-married",
    "occupation", "Machine-op-inspect",
    "relationship", "Own-child",
    "race", "Black",
    "sex", "Male",
    "capital-gain", 0,
    "capital-loss", 0,
    "hours-per-week", 40,
    "native-country", "United-States");

SELECT sys.ML_EXPLAIN_ROW(@row_input, @census_model);
```
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_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_attribute": value.

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 3.8.4, “ML_EXPLAIN_ROW”.

### 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 **ML_EXPLAIN_TABLE** parameter descriptions, see Section 3.8.5, “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.

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:

```
CALL sys.ML_MODEL_LOAD(@census_model, NULL);
```

For more information about loading models, see Section 3.7.2, “Loading Models”.

The following example creates a table with 10 rows of unlabeled test data and generates explanations for that table:

```
CREATE TABLE heatwaveml_bench.census_test_subset AS SELECT * FROM heatwaveml_bench.census_test LIMIT 10;
CALL sys.ML_EXPLAIN_TABLE('heatwaveml_bench.census_test_subset', @census_model, 'heatwaveml_bench.census_explanations');
```

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.

To view **ML_EXPLAIN_TABLE** results, query the output table; for example:

```
SELECT * FROM heatwaveml_bench.census_explanations;
```

### 3.7 Managing Models

#### 3.7.1 The Model Catalog

HeatWave ML stores machine learning models in a model catalog in MySQL. A model catalog is a standard MySQL table named **MODEL_CATALOG**. HeatWave ML 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 ML routines can only use models that are accessible to the user running the routines. For information about granting model catalog privileges, see Section 3.7.9, “Sharing Models”.

A database administrator can manage a model catalog table as they would a regular MySQL table.

The Model Catalog Table

The `MODEL_CATALOG` table has the following columns:

- **model_id**
  A unique auto-incrementing numeric identifier for the model.

- **model_handle**
  The model handle. The model handle is generated when the `ML_TRAIN` routine is executed on a training dataset. The model_handle format is `schemaName_tableName_userName_No`, as in the following example: `heatwaveml_bench.census_train_user1_1636729526`.

  **Note**
  The format of the model handle is subject to change.

- **model_object**
  A string in JSON format containing the serialized HeatWave ML model.

- **model_owner**
  The user who initiated the `ML_TRAIN` routine 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.

- **target_column_name**
  The name of the column in the training table that was specified as the target column.

- **train_table_name**
  The name of the input table specified by the `ML_TRAIN` routine.

- **model_object_size**
  The model object size, in bytes.

- **model_type**
  The name of the algorithm used to build the model.

- **task**
The machine learning modeling task type that was specified in the ML_TRAIN query (classification or regression).

- **column_names**
  The feature columns used to train the model.

- **model_explanations**
  The model explanation generated during training. See Section 3.7.6, “Model Explanations”. This column was added in MySQL 8.0.29.

- **last_accessed**
  The last time the model was accessed. HeatWave ML routines update this value to the current timestamp when accessing the model.

### 3.7.2 Loading Models

A model must be loaded from the model catalog into HeatWave before running HeatWave ML routines other than ML_TRAIN. A model remains loaded and can be called repetitively by HeatWave ML 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 3.7.9, “Sharing Models”.

You 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 3.8.7, “ML_MODEL_LOAD”.

The following example loads a HeatWave ML model from the model catalog:

```sql
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 owning user's name is specified as follows:

```sql
CALL sys.ML_MODEL_LOAD('heatwaveml_bench.census_train_user1_1636729526', 'user1');
```

To look up a model handle, see Section 3.7.7, “Model Handles”.

### 3.7.3 Unloading Models

The ML_MODEL_UNLOAD routine unloads a model from HeatWave ML. For ML_MODEL_UNLOAD parameter descriptions, see Section 3.8.8, “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:

```sql
CALL sys.ML_MODEL_UNLOAD('heatwaveml_bench.census_train_user1_1636729526');
```

where:
• \textit{heatwaveml\_bench.census\_train\_user1\_1636729526} is the model handle.

To look up a model handle, see Section 3.7.7, “Model Handles”.

### 3.7.4 Viewing Models

To view the models in your model catalog, query the \texttt{MODEL\_CATALOG} table; for example:

```sql
SELECT model\_id, model\_handle, model\_owner FROM ML\_SCHEMA\_user1.MODEL\_CATALOG;
```

where:

• \texttt{model\_id} is a unique numeric identifier for the model.

• \texttt{model\_owner} is the user that created the model.

• \texttt{model\_handle} is the handle by which the model is called.

• \texttt{ML\_SCHEMA\_user1.MODEL\_CATALOG} is the fully qualified name of the \texttt{MODEL\_CATALOG} table. The schema is named for the owning user.

\begin{warning}
\textbf{Note}

The example above retrieves data from only a few \texttt{MODEL\_CATALOG} table columns. For other columns you can query, see Section 3.7.1, “The Model Catalog”.
\end{warning}

### 3.7.5 Scoring Models

\texttt{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 \texttt{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.

\texttt{ML\_SCORE} returns a computed metric indicating the quality of the model. A value of \texttt{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: \texttt{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 \texttt{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 dataset is not representative of the training dataset. In this case, consider adding rows to the training dataset that better represent the test dataset.

HeatWave ML supports a variety of scoring metrics to help you understand how your model performs across a series of benchmarks. For \texttt{ML\_SCORE} parameter descriptions and supported metrics, see Section 3.8.6, “ML\_SCORE”.

Before running \texttt{ML\_SCORE}, ensure that the model you want to use is loaded; for example:

```sql
CALL sys.ML\_MODEL\_LOAD(@census\_model, NULL);
```

For information about loading models, see Section 3.7.2, “Loading Models”.

The following example runs \texttt{ML\_SCORE} to compute model quality using the \texttt{balanced\_accuracy} metric:
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.
- `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 3.8.6, "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 (e.g., `@my_score`).

To retrieve the computed score, query the `@score` session variable.

```
SELECT @score;
+--------------------+
| @score             |
+--------------------+
| 0.8188666105270386 |
```

### 3.7.6 Model Explanations

From MySQL 8.0.29, a model explanation is generated when you train a machine learning model using the `ML_TRAIN` routine. 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:

```
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.

### 3.7.7 Model Handles

When `ML_TRAIN` trains a model, it generates a model handle, which is required when running other HeatWave ML routines. 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);
```

While the connection used to run `ML_TRAIN` remains active, that connection can retrieve the generated model handle by querying the session variable; for example:
Deleting Models

```
mysql> SELECT @census_model;
+--------------------------------------------------+
| @census_model                                    |
| heatwaveml_bench.census_train_user1_1636729526   |
+--------------------------------------------------+
```

**Note**
The format of the 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`:
  ```
  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:
  ```
  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.

### 3.7.8 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:

```
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.

### 3.7.9 Sharing Models

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:
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 ML and the **CREATE** privilege on the schema where **ML_PREDICT_TABLE** or **ML_EXPLAIN_TABLE** results are written. See Section 3.1, “Before You Begin”.

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.

```sql
mysql> SET @my_model = (SELECT model_handle
   FROM ML_SCHEMA_user1.MODEL_CATALOG
   WHERE train_table_name LIKE '%census_train%');

mysql> SELECT @my_model;
+--------------------------------------------------+
<table>
<thead>
<tr>
<th>@my_model</th>
</tr>
</thead>
<tbody>
<tr>
<td>heatwaveml_bench.census_train_user1_1648167434</td>
</tr>
</tbody>
</table>
+--------------------------------------------------+

mysql> CALL sys.ML_PREDICT_TABLE('heatwaveml_bench.census_test_subset', @my_model, 'heatwaveml_bench.census_predictions');
```

### 3.8 HeatWave ML Routines

HeatWave ML routines reside in the MySQL **sys** schema.

Examples in this section are based on the **Iris Data Set**. See Section 6.3, “Iris Data Set Machine Learning Quickstart”.

#### 3.8.1 ML_TRAIN

Running the **ML_TRAIN** routine on a labeled training dataset produces a trained machine learning model.

**ML_TRAIN Syntax**

```sql
CALL sys.ML_TRAIN ('table_name', 'target_column_name', [options], model_handle_variable);
```

options: {
  JSON_OBJECT('key','value'[, 'key','value'] ...)
  'key','value':
  [ 'task', [ 'classification'|'regression'|NULL]
  ]
}

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 3.12, "Limitations".

**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 900 columns.

- **target_column_name**: The name of the target column containing ground truth values.

- **model_handle_variable**: 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 (e.g., @my_model).

When **ML_TRAIN** finishes executing, you can retrieve the generated model handle by querying the session variable. See Section 3.7.7, "Model Handles".

- **options**: Optional parameters specified as key-value pairs in JSON_OBJECT() 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_OBJECT() 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.

**Syntax Examples**

- An **ML_TRAIN** example that uses the **classification** task option implicitly (classification is the default if not specified explicitly):

  ```sql
  CALL sys.ML_TRAIN('ml_data.iris_train', 'class', NULL, @iris_model);
  ```

- An **ML_TRAIN** example that specifies the **classification** task type explicitly:

  ```sql
  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:

  ```sql
  CALL sys.ML_TRAIN('employee.salary_train', 'salary', JSON_OBJECT('task', 'regression'), @salary_model);
  ```

### 3.8.2 ML_PREDICT_ROW

**ML_PREDICT_ROW** generates predictions for one or more rows of unlabeled data specified in JSON format. **ML_PREDICT_ROW** is invoked using a SELECT statement.

A loaded model is required to run **ML_PREDICT_ROW**. See Section 3.7.2, "Loading Models".

**ML_PREDICT_ROW Syntax**

```sql
SELECT ML_PREDICT_ROW(input_data, model_handle);
```

**ML_PREDICT_ROW** parameters:

- **input_data**: Specifies the data to generate predictions for. A single row of data can be specified explicitly in JSON format:

  ```sql
  SELECT sys.ML_PREDICT_ROW(JSON_OBJECT("column_name", value, "column_name", value, ...),
  ```
ML_PREDICT_ROW can be run on multiple rows of data by specifying the columns as key-value pairs in JSON format and selecting from a table:

```
SELECT sys.ML_PREDICT_ROW(JSON_OBJECT("output_col_name", schema."input_col_name", output_col_name, schema."input_col_name", ...), model_handle) FROM input_table_name LIMIT N;
```

- **model_handle**: Specifies the model handle or a session variable containing the model handle.

**Syntax Examples**

- Running `ML_PREDICT_ROW` on a single row of data:
  ```
  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);
  ```

- Running `ML_PREDICT_ROW` on five rows of data selected from an input table:
  ```
  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)
  FROM ml_data.iris_test LIMIT 5;
  ```

### 3.8.3 ML_PREDICT_TABLE

`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.

`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 3.7.2, “Loading Models”.

**ML_PREDICT_TABLE Syntax**

```
CALL sys.ML_PREDICT_TABLE(table_name, model_handle, output_table_name);
```

- **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.

**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:
  ```
  CALL sys.ML_PREDICT_TABLE('ml_data.iris_test', @iris_model, 'ml_data.iris_predictions');
  ```

### 3.8.4 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 is required to run **ML_EXPLAIN_ROW**. See Section 3.7.2, “Loading Models”.

### ML_EXPLAIN_ROW Syntax

```plaintext
SELECT sys.ML_EXPLAIN_ROW(input_data, model_handle);
```

**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:

  ```plaintext
  SELECT sys.ML_EXPLAIN_ROW(JSON_OBJECT("column_name", value, "column_name", value, ...'), model_handle);
  ```

  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:

  ```plaintext
  SELECT sys.ML_EXPLAIN_ROW(JSON_OBJECT("output_col_name", schema.`input_col_name`, output_col_name", schema.`input_col_name`, ...), model_handle) FROM input_table_name LIMIT N;
  ```

- **model_handle**: Specifies the model handle or a session variable containing the model handle.

### Syntax Examples

- Running **ML_EXPLAIN_ROW** on a single row of data:

  ```plaintext
  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);
  ```

- Running **ML_EXPLAIN_ROW** on five rows of data selected from an input table:

  ```plaintext
  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)
  FROM ml_data.iris_test LIMIT 5;
  ```

### 3.8.5 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 is required to run **ML_EXPLAIN_TABLE**. See Section 3.7.2, “Loading Models”.

### ML_EXPLAIN_TABLE Syntax

```plaintext
CALL sys.ML_EXPLAIN_TABLE(table_name, model_handle, output_table_name);
```

**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.
Syntax Examples

- The following example generates explanations for a table of data. 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. The `SELECT` statement retrieves explanation data from the output table.

```sql
CALL sys.ML_EXPLAIN_TABLE('ml_data.iris_test', @iris_model, 'ml_data.iris_explanations');
SELECT * FROM ml_data.iris_explanations;
```

### 3.8.6 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 your labeled training data for scoring.

**ML_SCORE** returns a computed metric indicating the quality of the model.

**ML_SCORE Syntax**

```sql
CALL sys.ML_SCORE(table_name, target_column_name, model_handle, metric, score);
```

**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.

- `model_handle`: Specifies the model handle or a session variable containing the model handle.

- `metric`: Specifies the name of the classification or regression type metric. Commonly used classification metrics are `accuracy`, `balanced_accuracy`, `precision`, and `recall`. Commonly used regression metrics are `neg_mean_squared_error`, `neg_absolute_squared_error`, and `r2`. For more information about scoring metrics, refer to [scikit-learn.org](http://scikit-learn.org).

- **classification metrics:**
  - `accuracy`: Computes the fraction of labels a model predicts correctly
  - `balanced_accuracy`: Computes the balanced accuracy for imbalanced datasets
  - `precision`: Computes precision
  - `recall`: Computes the recall
  - `f1`: Computes the F1 score
  - `f1_micro`: Computes the F1 score (micro-averaged)
  - `f1_macro`: Computes the F1 score (macro-averaged)
  - `f1_weighted`: Computes the F1 score (weighted)
  - `f1_samples`: Computes the F1 score (by multilabel sample)
  - `recall_micro`: Computes the recall (micro-averaged)
• **recall_macro**: Computes the recall (macro-averaged)
• **recall_weighted**: Computes the recall (weighted)
• **recall_samples**: Computes the recall (by multilabel sample)
• **precision_micro**: Computes precision (micro-averaged)
• **precision_macro**: Computes precision (macro-averaged)
• **precision_weighted**: Computes precision (weighted)
• **precision_samples**: Computes precision (by multilabel sample)

**regression metrics:**
• **neg_mean_squared_error**: Computes mean squared error regression loss.
• **neg_mean_absolute_error**: Computes mean absolute error regression loss.
• **r2**: Computes the $R^2$ (coefficient of determination) regression score function.
• **neg_mean_squared_log_error**: Computes mean squared logarithmic error regression loss.
• **neg_median_absolute_error**: Computes median absolute error regression loss.

• **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 (e.g., `@my_score`).

**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);
mysql> SELECT @score;
+--------------------+
<table>
<thead>
<tr>
<th>@score</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.9583333134651184</td>
</tr>
</tbody>
</table>
+--------------------+
```

### 3.8.7 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 ML is restarted by a HeatWave Cluster restart.

A model can only be loaded by the MySQL user that created the model or by a user that was granted privileges to access the owning user's model catalog. For more information, see Section 3.7.9, "Sharing Models".

You can load multiple models but to avoid taking up too much space in memory, the number of loaded models should be limited to three.

**ML_MODEL_LOAD Syntax**

```sql
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 3.7.7, “Model Handles”.
- **user**: The MySQL user name of the model owner. You can specify NULL in place of the user name if the model owner is the current user.

**Syntax Examples**

- An **ML_MODEL_LOAD** call with NULL specified, indicating that the model belongs to the user executing the **ML_MODEL_LOAD** call:

  ```
  CALL sys.ML_MODEL_LOAD('ml_data.iris_train_user1_1636729526', NULL);
  ```

- An **ML_MODEL_LOAD** call that specifies the model owner:

  ```
  CALL sys.ML_MODEL_LOAD('ml_data.iris_train_user1_1636729526', 'user1');
  ```

- An **ML_MODEL_LOAD** call that specifies a session variable containing the model handle:

  ```
  CALL sys.ML_MODEL_LOAD(@iris_model, NULL);
  ```

### 3.8.8 ML_MODEL_UNLOAD

**ML_MODEL_UNLOAD** unloads a model from HeatWave ML.

**ML_MODEL_UNLOAD** Syntax

```
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:

  ```
  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:

  ```
  CALL sys.ML_MODEL_UNLOAD(@iris_model);
  ```

### 3.9 Supported Data Types

HeatWave ML supports the following data types:

- **FLOAT**
- **DOUBLE**
- **INT**
- **TINYINT**
- **SMALLINT**
- **MEDIUMINT**
- **BIGINT**
- **INT UNSIGNED**
- **TINYINT UNSIGNED**
Monitoring

- SMALLINT UNSIGNED
- MEDIUMINT UNSIGNED
- BIGINT UNSIGNED
- VARCHAR
- CHAR

DATE, TIME, DATETIME, and TIMESTAMP data types are not supported. You can either remove them or convert them to CHAR or VARCHAR columns, or split them into separate day, month, and year columns and define them as numeric or string types.

DECIMAL data type columns are not supported. You can either remove them or convert them to FLOAT.

3.10 Monitoring

You can monitor HeatWave ML 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 ML. Possible values are ON and OFF.
  
  • ON: HeatWave ML is up and running.
  
  • OFF: HeatWave ML 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 ML status information to the ML_STATUS column of performance_schema.rpd_nodes table after each ML query. Possible values include:

  • UNAVAIL_MLSTATE: HeatWave ML is not available.
  
  • AVAIL_MLSTATE: HeatWave ML is available.
  
  • DOWN_MLSTATE: HeatWave ML 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:
If `rapid_ml_status` is OFF or `ML_STATUS` reports `DOWN_MLSTATE` for any HeatWave node, you can restart the HeatWave Cluster in the Oracle Cloud Infrastructure (OCI) Console but be aware that restarting interrupts any analytics queries that are running. See Starting, Stopping, or Restarting a HeatWave Cluster, in the MySQL Database Service Guide.

### 3.11 HeatWave ML 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 and regression tasks are supported

  Check the `task` option in your `ML_TRAIN` call to ensure that it was specified correctly.

- **Error number:** ML001031; SQLSTATE: HY000
  
  Message: samples per class, and cannot be trained on. For a real valued target column, the task parameter in the options JSON should be set to regression.

  **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 underrepresented 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

  **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 3.3, “Preparing Data”.

- **Error number:** ML001053; SQLSTATE: HY000
  
  Message: Unlabeled samples detected in the training data.
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 3.3, “Preparing Data”.

• Error number: ML003000; SQLSTATE: HY000
  Message: 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

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']"

Possible causes: The input data columns do not match the columns of training dataset used to train the model. Compare your input data to your training data to identify the discrepancy.

• Error number: ML003012; SQLSTATE: HY000
  Message: is NULL or has not been loaded

Example: ERROR HY000: "ML003012: 'The table (mlcorpus.iris_train) is NULL or has not been loaded.'"

Possible causes: 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

Possible causes: 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

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: in input JSON ->

Example: ERROR HY000: "ML003016: Missing argument "row_json" in input JSON -> dict_keys(['operation', 'user_name', 'table_name', 'schema_name', 'model_handle'])"

Possible causes: The syntax of your ML routine call is not valid.

• 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!"
Possible causes: The syntax of your ML routine call is not valid.

• 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!"

Possible causes: The syntax of your ML routine call is not valid.

• Error number: ML003019; SQLSTATE: HY000
  Message: Invalid data for the metric
  Example: ERROR HY000: "ML003019: Invalid data for the metric (roc_auc). Score could not be computed"

Possible causes: The scoring metric is legal and supported, but the data provided is not suitable to calculate such score; for example: ROC_AUC for multi-class classification. Try a different scoring metric.

• Error number: ML003020; SQLSTATE: HY000
  Message: Unsupported scoring function
  Example: ERROR HY000: "ML003020: Unsupported scoring function (accuracy) for current task (regression)."

Possible causes: The scoring metric is legal and supported, but the task provided is not suitable to calculate such 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.
  Possible causes: 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.
  Possible causes: 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: HY000
  Message: Unknown option given. Allowed options for training are.
  Possible causes: 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.
  Possible causes: There is not enough memory on your HeatWave Cluster to perform the operation. Try unloading data that was loaded for analytics to free up space.

• Error number: ML004014; SQLSTATE: HY000
  Message: Missing expected JSON key
HeatWave ML Error Messages

Example: **ERROR HY000: ML004014**: Missing expected JSON key (schema_name).

- **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**: 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 prediction_row prediction_row) is larger than maximum permitted size

- **Error number**: ML004018 ; **SQLSTATE**: HY000
  
  **Message**: Parsing JSON arg: failed!

  Example: **ERROR HY000: ML004018**: Parsing JSON arg: Invalid value. failed!

- **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**: ML004020 ; **SQLSTATE**: HY000
  
  **Message**: Operation was interrupted by the user

  If a user-initiated interruption (Ctrl-C) is detected during the first phase of HeatWave ML model and table load where a MySQL parallel scan is used in the HeatWave plugin to read data from 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**: ML004022 ; **SQLSTATE**: HY000
  
  **Message**: The user doesn't have access privileges to %.

  Example: **ERROR HY000: ML004022**: The user doesn't have access privileges to ml.foo

- **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
3.12 Limitations

HeatWave ML has the following 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 (the *training dataset*) cannot exceed 10 GB, 100 million rows, or 900 columns.

- 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 ML models or manually modified HeatWave ML models can cause undefined behavior.

- Models greater than 900 MB in size are not supported. If a model being trained by the `ML_TRAIN` routine exceeds 900 MB, the `ML_TRAIN` query fails with an error.

- There is currently no way to monitor HeatWave ML 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.

- `Ctrl+C` interruption is supported only for `ML_TRAIN`, `ML_EXPLAIN_ROW`, and `ML_EXPLAIN_TABLE`.

- `{ML_EXPLAIN_}*` routines limit explanations to the 100 most relevant features.

Chapter 4 System and Status Variables

Table of Contents

4.1 System Variables ................................................................. 109
4.2 Status Variables ................................................................. 112

4.1 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.

- **rapid_compression**

<table>
<thead>
<tr>
<th>Introduced</th>
<th>8.0.29</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Variable</td>
<td>rapid_compression</td>
</tr>
<tr>
<td>Scope</td>
<td>Session</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Yes</td>
</tr>
<tr>
<td>SET_VAR Hint Applies</td>
<td>No</td>
</tr>
<tr>
<td>Type</td>
<td>Boolean</td>
</tr>
<tr>
<td>Default Value</td>
<td>ON</td>
</tr>
</tbody>
</table>

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”.

- **rapid_bootstrap**

  | Command-Line Format | --rapid-bootstrap=[OFF|ON|IDLE] |
  |---------------------|---------------------------------|
  | Introduced          | 8.0.17                          |
  | 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=#`
  Introduced: 8.0.17
  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=#`
  Introduced: 8.0.17
  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}]`
  Introduced: 8.0.22
  System Variable: `rapid_execution_strategy`
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 rapid_execution_strategy prior to executing the query:

```sql
SET SESSION rapid_execution_strategy = MIN_MEM_CONSUMPTION;
```

See Section 2.14, “Troubleshooting”.

- rapid_stats_cache_max_entries

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 Auto Query Plan Improvement.

- show_create_table_skip_secondary_engine
Status Variables

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.

Attempts a `mysqldump` operation with the `--show-create-skip-secondary-engine` option on a MySQL Server release prior to MySQL 8.0.18 that does not support the `show_create_table_skip_secondary_engine` variable causes an error.

• **use_secondary_engine**

<table>
<thead>
<tr>
<th>Introduced</th>
<th>8.0.13</th>
</tr>
</thead>
<tbody>
<tr>
<td>System Variable</td>
<td><code>use_secondary_engine</code></td>
</tr>
<tr>
<td>Scope</td>
<td>Session</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Yes</td>
</tr>
<tr>
<td><code>SET_VAR</code> Hint Applies</td>
<td>Yes</td>
</tr>
<tr>
<td>Type</td>
<td>Enumeration</td>
</tr>
<tr>
<td>Default Value</td>
<td>ON</td>
</tr>
<tr>
<td>Valid Values</td>
<td><code>OFF</code> <code>ON</code> <code>FORCED</code></td>
</tr>
</tbody>
</table>

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.

### 4.2 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%';
```

<table>
<thead>
<tr>
<th>Variable_name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>hw_data_scanned</td>
<td>0</td>
</tr>
<tr>
<td>rapid_change_propagation_status</td>
<td>ON</td>
</tr>
<tr>
<td>rapid_cluster_status</td>
<td>ON</td>
</tr>
<tr>
<td>rapid_core_count</td>
<td>64</td>
</tr>
<tr>
<td>rapid_heap_usage</td>
<td>58720397</td>
</tr>
<tr>
<td>rapid_load_progress</td>
<td>100.000000</td>
</tr>
<tr>
<td>rapid_ml_status</td>
<td>ON</td>
</tr>
<tr>
<td>rapid_plugin_bootstrapped</td>
<td>YES</td>
</tr>
</tbody>
</table>
Status Variables

<table>
<thead>
<tr>
<th>Status Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>rapid_preload_stats_status</td>
<td>Available</td>
</tr>
<tr>
<td>rapid_query_offload_count</td>
<td>46</td>
</tr>
<tr>
<td>rapid_service_status</td>
<td>ONLINE</td>
</tr>
</tbody>
</table>

- **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_status**
  The status of HeatWave ML. 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 5.5, “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.
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”.

5.1 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.

- **NAME**
  The fully qualified column name, including the schema, table, and column

- **SCHEMA_NAME**
  The schema name.

- **TABLE_NAME**
  The table name.

- **COLUMN_NAME**
  The column name.

The rpd_column_id table is read-only.

5.2 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**
The rpd_exec_stats Table

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 2.15, “Metadata Queries”.

• DICT_SIZE_BTYES
  The dictionary size per column, in bytes.

The rpd_columns table is read-only.

5.3 The rpd_exec_stats Table

Note
The Performance Schema table described here is available as of MySQL 8.0.24.

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.

The rpd_exec_stats table has these columns:

• QUERY_ID
  The query ID.

• NODE_ID
  The HeatWave node ID.

• EXEC_TEXT
  Query execution statistics.

5.4 The rpd_nodes Table

The rpd_nodes table provides information about HeatWave nodes.

The rpd_nodes table has these columns:

• ID
The rpd_nodes Table

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 (renamed from DRAM in MySQL 8.0.24)
  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 Machine Learning status. Possible status values include:
  • UNAVAIL_MLSTATE: HeatWave Machine Learning is not available.
  • AVAIL_MLSTATE: HeatWave Machine Learning is available.
  • DOWN_MLSTATE: HeatWave Machine Learning 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.

5.5 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 prior to 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.
5.6 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.

The rpd_query_stats table has these columns:

- **QUERY_ID**
  The query ID.

- **QUERY_TEXT**
  The query.

- **QEXEC_TEXT**
  Query execution log.

- **QKRN_TEXT**
  Logical query execution plan.

- **QEP_TEXT**
  Physical query execution plan.

Includes prepart data, which can be queried to determine if a JOIN or GROUP BY query used data placement partitions. See Section 2.15, “Metadata Queries”.

5.7 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.

5.8 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.
  - **UNAVAIL_RPDGSTABSTATE**
    The table is unavailable.

- **LOAD_PROGRESS**
  Value 120
The rpd_tables Table

The load progress of the table expressed as a percentage value.

- **SIZE_BYTES**
  The amount of data loaded for the table, in bytes.

- **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.

The rpd_tables table is read-only.
Chapter 6 HeatWave Quickstarts

Table of Contents

6.1 tpch Analytics Quickstart .............................................................. 123
6.2 AirportDB Analytics Quickstart ......................................................... 130
6.3 Iris Data Set Machine Learning Quickstart ........................................... 134

6.1 tpch Analytics Quickstart

The tpch Analytics Quickstart walks you through importing data into the DB System using the MySQL Shell Parallel Table Import Utility, manually loading data into HeatWave, and running queries.

The following topics are described:

- Prerequisites
- Generating tpch Sample Data
- Creating the Sample Database and Importing Data
- Loading tpch Data Into HeatWave
- Running tpch Queries
- Additional tpch Queries
- Unloading tpch Tables

Prerequisites

- MySQL Shell 8.0.22 or higher. See Connecting to the MySQL DB System with SSH and MySQL Shell.

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
Creating the 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's endpoint:

   ```bash
   $> 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>JS>

2. Change the MySQL Shell execution mode from JavaScript to SQL:

   ```bash
   MySQL>JS> \sql
   ```

3. Create the `tpch` sample database and tables:

   ```sql
   CREATE DATABASE tpch Character_set utf8mb4;
   USE tpch;

   CREATE TABLE nation ( N_NATIONKEY INTEGER primary key, 
                        N_NAME CHAR(25) NOT NULL, 
                        N_REGIONKEY INTEGER NOT NULL, 
                        N_COMMENT VARCHAR(152));

   CREATE TABLE region ( R_REGIONKEY INTEGER primary key, 
                         R_NAME CHAR(25) NOT NULL, 
                         R_COMMENT VARCHAR(152));
   ```
Creating the Sample Database and Importing Data

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 );

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);

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) );

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);

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);

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>SHOW TABLES;
+----------------+
| Tables_in_tpch |
+----------------+
| customer       |
| lineitem       |
| nation         |
| orders         |
+----------------+
Loading tpch Data Into HeatWave

To load the tpch sample data into the HeatWave Cluster:

1. Start MySQL Shell and connect to the MySQL DB System's 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.

   USE tpch;
   ALTER TABLE nation modify `N_NAME` CHAR(25) NOT NULL COMMENT 'RAPID_COLUMN=ENCODING=SORTED';
ALTER TABLE nation modify `N_COMMENT` VARCHAR(152) COMMENT 'RAPID_COLUMN=ENCODING=SORTED';
ALTER TABLE nation SECONDARY_ENGINE=RAPID;
ALTER TABLE nation SECONDARY_LOAD;
ALTER TABLE region modify `R_NAME` CHAR(25) NOT NULL COMMENT 'RAPID_COLUMN=ENCODING=SORTED';
ALTER TABLE region modify `R_COMMENT` VARCHAR(152) COMMENT 'RAPID_COLUMN=ENCODING=SORTED';
ALTER TABLE region SECONDARY_ENGINE=RAPID;
ALTER TABLE region SECONDARY_LOAD;
ALTER TABLE part modify `P_MFGR` CHAR(25) NOT NULL COMMENT 'RAPID_COLUMN=ENCODING=SORTED';
ALTER TABLE part modify `P_BRAND` CHAR(10) NOT NULL COMMENT 'RAPID_COLUMN=ENCODING=SORTED';
ALTER TABLE part modify `P_CONTAINER` CHAR(10) NOT NULL COMMENT 'RAPID_COLUMN=ENCODING=SORTED';
ALTER TABLE part modify `P_COMMENT` VARCHAR(23) NOT NULL COMMENT 'RAPID_COLUMN=ENCODING=SORTED';
ALTER TABLE part SECONDARY_ENGINE=RAPID;
ALTER TABLE part SECONDARY_LOAD;
ALTER TABLE supplier modify `S_NAME` CHAR(25) NOT NULL COMMENT 'RAPID_COLUMN=ENCODING=SORTED';
ALTER TABLE supplier modify `S_ADDRESS` VARCHAR(40) NOT NULL COMMENT 'RAPID_COLUMN=ENCODING=SORTED';
ALTER TABLE supplier modify `S_PHONE` CHAR(15) NOT NULL COMMENT 'RAPID_COLUMN=ENCODING=SORTED';
ALTER TABLE supplier SECONDARY_ENGINE=RAPID;
ALTER TABLE supplier SECONDARY_LOAD;
ALTER TABLE partsupp modify `PS_COMMENT` VARCHAR(199) NOT NULL COMMENT 'RAPID_COLUMN=ENCODING=SORTED';
ALTER TABLE partsupp SECONDARY_ENGINE=RAPID;
ALTER TABLE partsupp SECONDARY_LOAD;
ALTER TABLE customer modify `C_NAME` VARCHAR(25) NOT NULL COMMENT 'RAPID_COLUMN=ENCODING=SORTED';
ALTER TABLE customer modify `C_ADDRESS` VARCHAR(40) NOT NULL COMMENT 'RAPID_COLUMN=ENCODING=SORTED';
ALTER TABLE customer modify `C_MKTSEGMENT` CHAR(10) NOT NULL COMMENT 'RAPID_COLUMN=ENCODING=SORTED';
ALTER TABLE customer modify `C_COMMENT` VARCHAR(117) NOT NULL COMMENT 'RAPID_COLUMN=ENCODING=SORTED';
ALTER TABLE customer SECONDARY_ENGINE=RAPID;
ALTER TABLE customer SECONDARY_LOAD;
ALTER TABLE orders modify `O_ORDERSTATUS` CHAR(1) NOT NULL COMMENT 'RAPID_COLUMN=ENCODING=SORTED';
ALTER TABLE orders modify `O_ORDERPRIORITY` CHAR(15) NOT NULL COMMENT 'RAPID_COLUMN=ENCODING=SORTED';
ALTER TABLE orders modify `O_CLERK` CHAR(15) NOT NULL COMMENT 'RAPID_COLUMN=ENCODING=SORTED';
ALTER TABLE orders SECONDARY_ENGINE=RAPID;
ALTER TABLE orders SECONDARY_LOAD;
ALTER TABLE lineitem modify `L_RETURNFLAG` CHAR(1) NOT NULL COMMENT 'RAPID_COLUMN=ENCODING=SORTED';
ALTER TABLE lineitem modify `L_LINESTATUS` CHAR(1) NOT NULL COMMENT 'RAPID_COLUMN=ENCODING=SORTED';
ALTER TABLE lineitem modify `L_SHIPINSTRUCT` CHAR(25) NOT NULL COMMENT 'RAPID_COLUMN=ENCODING=SORTED';
ALTER TABLE lineitem modify `L_SHIPMODE` CHAR(10) NOT NULL COMMENT 'RAPID_COLUMN=ENCODING=SORTED';
ALTER TABLE lineitem modify `L_COMMENT` VARCHAR(44) NOT NULL COMMENT 'RAPID_COLUMN=ENCODING=SORTED';
ALTER TABLE lineitem SECONDARY_ENGINE=RAPID;
ALTER TABLE lineitem SECONDARY_LOAD;

4. Verify that the \texttt{tpch} sample database tables are loaded in the HeatWave Cluster by querying \texttt{LOAD\_STATUS} data from the HeatWave Performance Schema tables. Loaded tables have an \texttt{AVAIL\_RPDGSTABSTATE} load status.

MySQL> USE performance_schema;
MySQL> SELECT NAME, LOAD_STATUS FROM rpd_tables, rpd_table_id
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:

1. Start MySQL Shell and connect to the MySQL DB System's 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;
   ```

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 |
   +------------------+
   ```
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> SET SESSION use_secondary_engine=OFF;
MySQL> 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."

```sql
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."

```sql
SELECT l_orderkey, SUM(l_extendedprice * (1 - l_discount)) AS revenue, o_orderdate, o_shippriority FROM customer, orders WHERE l_orderkey = o_orderkey AND l_shipdate <= DATE '1998-12-01' - INTERVAL '90' DAY GROUP BY l_orderkey, o_orderdate, o_shippriority ORDER BY revenue DESC LIMIT 10;
```
Unloading tpch Tables

---

```
lineitem
WHERE
c_mktsegment = 'BUILDING'
  AND c_custkey = o_custkey
  AND 1_orderkey = o_orderkey
  AND o_orderdate < DATE '1995-03-15'
  AND 1_shipdate > DATE '1995-03-15'
GROUP BY 1_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 \( [(l_{\text{extendedprice}}*(1-l_{\text{discount}})) - (ps_{\text{supplycost}} * l_{\text{quantity}})] \) 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)."

```
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:

USE tpch;

ALTER TABLE customer SECONDARY_UNLOAD;
ALTER TABLE lineitem SECONDARY_UNLOAD;
ALTER TABLE nation SECONDARY_UNLOAD;
ALTER TABLE orders SECONDARY_UNLOAD;
ALTER TABLE part SECONDARY_UNLOAD;
ALTER TABLE partsupp SECONDARY_UNLOAD;
ALTER TABLE region SECONDARY_UNLOAD;
ALTER TABLE supplier SECONDARY_UNLOAD;

6.2 AirportDB Analytics Quickstart

The HeatWave airportdb Quickstart walks you through importing data into the DB System using the MySQL Shell Dump Load utility, loading data into HeatWave using Auto Parallel Load, and running queries.

The following topics are described:
Prerequisites

- Prerequisites
- Installing airportdb
- Loading airportdb into HeatWave
- Running airportdb Queries
- Additional airportdb Queries
- Unloading airportdb Tables

Prerequisites

- MySQL Shell 8.0.22 or higher. See Connecting to the MySQL DB System with SSH and MySQL Shell.

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 DB System, see Connecting to the MySQL DB System with SSH and MySQL Shell.

   ```
   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.

## Loading airportdb into HeatWave

To load the airportdb from the MySQL DB System into HeatWave:

1. Start MySQL Shell and connect to the MySQL DB System's Endpoint.
   
   ```bash
   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.
   
   ```sql
   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:

1. Start MySQL Shell and connect to the MySQL DB System's endpoint:
   
   ```bash
   $> mysqlsh Username@DBSystem_IP_Address_or_Host_Name
   ```

   MySQL Shell opens in JavaScript execution mode by default.

2. Change the MySQL Shell execution mode to SQL:
   
   ```bash
   MySQL>JS> \sql
   ```

3. Change to the airportdb database.
   
   ```sql
   MySQL>SQL> USE airportdb;
   ```

4. Before running a query, use `EXPLAIN` to verify that the query can be offloaded to the HeatWave Cluster. For example:
   
   ```sql
   MySQL>SQL> EXPLAIN SELECT booking.price, count(*) FROM booking WHERE booking.price > 500 GROUP BY booking.price ORDER BY booking.price LIMIT 10;
   ```

   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.

```sql
SELECT booking.price, count(*) FROM booking WHERE booking.price > 500
GROUP BY booking.price ORDER BY booking.price LIMIT 10;
```

<table>
<thead>
<tr>
<th>price</th>
<th>count(*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>500.01</td>
<td>860</td>
</tr>
<tr>
<td>500.02</td>
<td>1207</td>
</tr>
<tr>
<td>500.03</td>
<td>1135</td>
</tr>
<tr>
<td>500.04</td>
<td>1010</td>
</tr>
<tr>
<td>500.05</td>
<td>1016</td>
</tr>
<tr>
<td>500.06</td>
<td>1039</td>
</tr>
<tr>
<td>500.07</td>
<td>1002</td>
</tr>
<tr>
<td>500.08</td>
<td>1095</td>
</tr>
<tr>
<td>500.09</td>
<td>1117</td>
</tr>
<tr>
<td>500.10</td>
<td>1106</td>
</tr>
</tbody>
</table>

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:

```sql
SET SESSION use_secondary_engine=OFF;
SELECT booking.price, count(*) FROM booking WHERE booking.price > 500
GROUP BY booking.price ORDER BY booking.price LIMIT 10;
```

<table>
<thead>
<tr>
<th>price</th>
<th>count(*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>500.01</td>
<td>860</td>
</tr>
<tr>
<td>500.02</td>
<td>1207</td>
</tr>
<tr>
<td>500.03</td>
<td>1135</td>
</tr>
<tr>
<td>500.04</td>
<td>1010</td>
</tr>
<tr>
<td>500.05</td>
<td>1016</td>
</tr>
<tr>
<td>500.06</td>
<td>1039</td>
</tr>
<tr>
<td>500.07</td>
<td>1002</td>
</tr>
<tr>
<td>500.08</td>
<td>1095</td>
</tr>
<tr>
<td>500.09</td>
<td>1117</td>
</tr>
<tr>
<td>500.10</td>
<td>1106</td>
</tr>
</tbody>
</table>

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**

```sql
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**

```sql
SELECT
```
Unloading airportdb Tables

```
SELECT airline.airlinername,
       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.airlinername
ORDER BY airline.airlinername, avg_age
LIMIT 10;
```

- Query 3: Most Tickets Sales by Airline for Departures from US Airports

```
SELECT airline.airlinername,
       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.airlinername
ORDER BY nb_tickets desc, airline.airlinername
LIMIT 10;
```

### Unloading airportdb Tables

airportdb tables can be unloaded from HeatWave using the following statements:

```
USE airportdb;
ALTER TABLE booking SECONDARY_UNLOAD;
ALTER TABLE flight SECONDARY_UNLOAD;
ALTER TABLE flight_log SECONDARY_UNLOAD;
ALTER TABLE airport SECONDARY_UNLOAD;
ALTER TABLE airport_reachable SECONDARY_UNLOAD;
ALTER TABLE airport_geo SECONDARY_UNLOAD;
ALTER TABLE airline SECONDARY_UNLOAD;
ALTER TABLE flightschedule SECONDARY_UNLOAD;
ALTER TABLE airplane SECONDARY_UNLOAD;
ALTER TABLE airplane_type SECONDARY_UNLOAD;
ALTER TABLE employee SECONDARY_UNLOAD;
ALTER TABLE passenger SECONDARY_UNLOAD;
ALTER TABLE passengerdetails SECONDARY_UNLOAD;
ALTER TABLE weatherdata SECONDARY_UNLOAD;
```

### 6.3 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 ML. It steps you 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 your model works.

The tutorial uses the publicly available Iris Data Set from the UCI Machine Learning Repository.

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.

This tutorial assumes that you have met the prerequisites outlined in Section 3.1, “Before You Begin”.

1. Create the example schema and tables on the MySQL DB System by executing the following statements:

```sql
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');
```

<table>
<thead>
<tr>
<th>Species</th>
<th>Sepal Length</th>
<th>Sepal Width</th>
<th>Petal Length</th>
<th>Petal Width</th>
<th>Species</th>
</tr>
</thead>
<tbody>
<tr>
<td>Iris-virginica</td>
<td>6.7</td>
<td>3.3</td>
<td>5.7</td>
<td>2.1</td>
<td>Iris-virginica</td>
</tr>
<tr>
<td>Iris-virginica</td>
<td>6.4</td>
<td>2.8</td>
<td>5.6</td>
<td>2.1</td>
<td>Iris-virginica</td>
</tr>
<tr>
<td>Iris-versicolor</td>
<td>5.4</td>
<td>3.9</td>
<td>1.3</td>
<td>0.4</td>
<td>Iris-versicolor</td>
</tr>
<tr>
<td>Iris-virginica</td>
<td>6.1</td>
<td>2.6</td>
<td>6.5</td>
<td>1.4</td>
<td>Iris-virginica</td>
</tr>
<tr>
<td>Iris-virginica</td>
<td>7.2</td>
<td>3.0</td>
<td>5.8</td>
<td>1.6</td>
<td>Iris-virginica</td>
</tr>
<tr>
<td>Iris-versicolor</td>
<td>5.2</td>
<td>3.5</td>
<td>1.5</td>
<td>0.2</td>
<td>Iris-versicolor</td>
</tr>
<tr>
<td>Iris-setosa</td>
<td>5.8</td>
<td>2.6</td>
<td>1.5</td>
<td>1.2</td>
<td>Iris-setosa</td>
</tr>
<tr>
<td>Iris-virginica</td>
<td>5.9</td>
<td>3.0</td>
<td>5.1</td>
<td>1.8</td>
<td>Iris-virginica</td>
</tr>
<tr>
<td>Iris-setosa</td>
<td>5.4</td>
<td>3.0</td>
<td>4.5</td>
<td>1.5</td>
<td>Iris-setosa</td>
</tr>
<tr>
<td>Iris-setosa</td>
<td>6.7</td>
<td>3.0</td>
<td>5.0</td>
<td>1.7</td>
<td>Iris-setosa</td>
</tr>
<tr>
<td>Iris-versicolor</td>
<td>6.3</td>
<td>2.3</td>
<td>4.4</td>
<td>1.7</td>
<td>Iris-versicolor</td>
</tr>
<tr>
<td>Iris-setosa</td>
<td>5.2</td>
<td>2.5</td>
<td>5.1</td>
<td>3.0</td>
<td>Iris-setosa</td>
</tr>
<tr>
<td>Iris-setosa</td>
<td>5.0</td>
<td>3.0</td>
<td>5.1</td>
<td>2.4</td>
<td>Iris-setosa</td>
</tr>
<tr>
<td>Iris-setosa</td>
<td>5.8</td>
<td>2.8</td>
<td>5.1</td>
<td>2.4</td>
<td>Iris-setosa</td>
</tr>
<tr>
<td>Iris-setosa</td>
<td>6.8</td>
<td>3.0</td>
<td>5.5</td>
<td>2.1</td>
<td>Iris-setosa</td>
</tr>
<tr>
<td>Iris-virginica</td>
<td>7.2</td>
<td>3.0</td>
<td>5.0</td>
<td>2.1</td>
<td>Iris-virginica</td>
</tr>
<tr>
<td>Iris-virginica</td>
<td>5.9</td>
<td>3.2</td>
<td>5.7</td>
<td>2.3</td>
<td>Iris-virginica</td>
</tr>
<tr>
<td>Iris-virginica</td>
<td>6.5</td>
<td>3.2</td>
<td>5.1</td>
<td>2.0</td>
<td>Iris-virginica</td>
</tr>
<tr>
<td>Iris-virginica</td>
<td>5.8</td>
<td>2.8</td>
<td>5.2</td>
<td>2.2</td>
<td>Iris-virginica</td>
</tr>
<tr>
<td>Iris-setosa</td>
<td>5.1</td>
<td>3.8</td>
<td>1.5</td>
<td>0.3</td>
<td>Iris-setosa</td>
</tr>
<tr>
<td>Iris-setosa</td>
<td>4.8</td>
<td>3.0</td>
<td>1.4</td>
<td>0.3</td>
<td>Iris-setosa</td>
</tr>
<tr>
<td>Iris-setosa</td>
<td>7.9</td>
<td>3.8</td>
<td>6.4</td>
<td>2.0</td>
<td>Iris-setosa</td>
</tr>
<tr>
<td>Iris-versicolor</td>
<td>6.2</td>
<td>2.7</td>
<td>5.1</td>
<td>1.9</td>
<td>Iris-versicolor</td>
</tr>
<tr>
<td>Iris-setosa</td>
<td>5.8</td>
<td>2.8</td>
<td>5.1</td>
<td>2.4</td>
<td>Iris-setosa</td>
</tr>
<tr>
<td>Iris-setosa</td>
<td>5.1</td>
<td>3.8</td>
<td>1.5</td>
<td>0.3</td>
<td>Iris-setosa</td>
</tr>
<tr>
<td>Iris-versicolor</td>
<td>4.8</td>
<td>3.2</td>
<td>1.6</td>
<td>0.2</td>
<td>Iris-versicolor</td>
</tr>
<tr>
<td>Iris-versicolor</td>
<td>5.0</td>
<td>3.3</td>
<td>1.4</td>
<td>0.2</td>
<td>Iris-versicolor</td>
</tr>
<tr>
<td>Iris-setosa</td>
<td>6.6</td>
<td>3.0</td>
<td>4.4</td>
<td>1.4</td>
<td>Iris-setosa</td>
</tr>
<tr>
<td>Iris-setosa</td>
<td>6.1</td>
<td>2.8</td>
<td>4.0</td>
<td>1.3</td>
<td>Iris-setosa</td>
</tr>
<tr>
<td>Iris-versicolor</td>
<td>5.0</td>
<td>2.9</td>
<td>4.0</td>
<td>1.3</td>
<td>Iris-versicolor</td>
</tr>
<tr>
<td>Iris-versicolor</td>
<td>6.4</td>
<td>2.9</td>
<td>4.8</td>
<td>1.4</td>
<td>Iris-versicolor</td>
</tr>
<tr>
<td>Iris-versicolor</td>
<td>5.7</td>
<td>2.8</td>
<td>4.5</td>
<td>1.3</td>
<td>Iris-versicolor</td>
</tr>
<tr>
<td>Iris-setosa</td>
<td>6.0</td>
<td>2.7</td>
<td>5.1</td>
<td>1.6</td>
<td>Iris-setosa</td>
</tr>
<tr>
<td>Iris-setosa</td>
<td>5.0</td>
<td>3.5</td>
<td>1.3</td>
<td>0.2</td>
<td>Iris-setosa</td>
</tr>
<tr>
<td>Iris-versicolor</td>
<td>6.6</td>
<td>2.9</td>
<td>4.6</td>
<td>1.3</td>
<td>Iris-versicolor</td>
</tr>
<tr>
<td>Iris-versicolor</td>
<td>5.5</td>
<td>3.5</td>
<td>1.3</td>
<td>0.2</td>
<td>Iris-versicolor</td>
</tr>
<tr>
<td>Iris-setosa</td>
<td>5.6</td>
<td>2.5</td>
<td>3.9</td>
<td>1.1</td>
<td>Iris-setosa</td>
</tr>
<tr>
<td>Iris-versicolor</td>
<td>6.7</td>
<td>3.1</td>
<td>5.6</td>
<td>2.4</td>
<td>Iris-versicolor</td>
</tr>
<tr>
<td>Iris-versicolor</td>
<td>6.2</td>
<td>2.7</td>
<td>5.1</td>
<td>1.9</td>
<td>Iris-versicolor</td>
</tr>
<tr>
<td>Iris-versicolor</td>
<td>5.7</td>
<td>3.0</td>
<td>5.1</td>
<td>1.8</td>
<td>Iris-versicolor</td>
</tr>
<tr>
<td>Iris-versicolor</td>
<td>6.0</td>
<td>3.0</td>
<td>5.0</td>
<td>1.5</td>
<td>Iris-versicolor</td>
</tr>
<tr>
<td>Iris-versicolor</td>
<td>5.5</td>
<td>2.4</td>
<td>3.8</td>
<td>1.1</td>
<td>Iris-versicolor</td>
</tr>
<tr>
<td>Iris-versicolor</td>
<td>6.1</td>
<td>2.9</td>
<td>4.2</td>
<td>1.3</td>
<td>Iris-versicolor</td>
</tr>
<tr>
<td>Iris-versicolor</td>
<td>6.2</td>
<td>2.2</td>
<td>4.5</td>
<td>1.5</td>
<td>Iris-versicolor</td>
</tr>
<tr>
<td>Iris-versicolor</td>
<td>7.3</td>
<td>2.9</td>
<td>6.3</td>
<td>1.8</td>
<td>Iris-versicolor</td>
</tr>
<tr>
<td>Iris-versicolor</td>
<td>4.4</td>
<td>3.0</td>
<td>1.3</td>
<td>0.2</td>
<td>Iris-versicolor</td>
</tr>
<tr>
<td>Iris-versicolor</td>
<td>7.2</td>
<td>3.6</td>
<td>6.1</td>
<td>1.2</td>
<td>Iris-versicolor</td>
</tr>
<tr>
<td>Iris-versicolor</td>
<td>6.5</td>
<td>3.0</td>
<td>5.5</td>
<td>1.8</td>
<td>Iris-versicolor</td>
</tr>
<tr>
<td>Iris-versicolor</td>
<td>5.0</td>
<td>3.4</td>
<td>1.5</td>
<td>0.2</td>
<td>Iris-versicolor</td>
</tr>
<tr>
<td>Iris-versicolor</td>
<td>6.0</td>
<td>3.2</td>
<td>1.3</td>
<td>0.2</td>
<td>Iris-versicolor</td>
</tr>
<tr>
<td>Iris-versicolor</td>
<td>6.6</td>
<td>2.9</td>
<td>4.6</td>
<td>1.3</td>
<td>Iris-versicolor</td>
</tr>
<tr>
<td>Iris-versicolor</td>
<td>6.2</td>
<td>2.9</td>
<td>4.6</td>
<td>1.3</td>
<td>Iris-versicolor</td>
</tr>
<tr>
<td>Iris-versicolor</td>
<td>5.5</td>
<td>3.5</td>
<td>1.3</td>
<td>0.2</td>
<td>Iris-versicolor</td>
</tr>
<tr>
<td>Iris-versicolor</td>
<td>7.7</td>
<td>3.0</td>
<td>6.1</td>
<td>1.2</td>
<td>Iris-versicolor</td>
</tr>
<tr>
<td>Iris-versicolor</td>
<td>6.1</td>
<td>3.0</td>
<td>4.9</td>
<td>1.8</td>
<td>Iris-versicolor</td>
</tr>
<tr>
<td>Iris-versicolor</td>
<td>4.9</td>
<td>3.1</td>
<td>1.5</td>
<td>0.1</td>
<td>Iris-versicolor</td>
</tr>
<tr>
<td>Iris-versicolor</td>
<td>5.5</td>
<td>2.4</td>
<td>3.8</td>
<td>1.1</td>
<td>Iris-versicolor</td>
</tr>
<tr>
<td>Iris-versicolor</td>
<td>6.3</td>
<td>2.9</td>
<td>4.2</td>
<td>1.3</td>
<td>Iris-versicolor</td>
</tr>
<tr>
<td>Iris-versicolor</td>
<td>6.0</td>
<td>2.9</td>
<td>4.5</td>
<td>1.5</td>
<td>Iris-versicolor</td>
</tr>
<tr>
<td>Iris-versicolor</td>
<td>6.4</td>
<td>2.7</td>
<td>5.3</td>
<td>1.9</td>
<td>Iris-versicolor</td>
</tr>
<tr>
<td>Iris-versicolor</td>
<td>5.4</td>
<td>3.7</td>
<td>1.5</td>
<td>0.2</td>
<td>Iris-versicolor</td>
</tr>
<tr>
<td>Iris-versicolor</td>
<td>6.1</td>
<td>2.9</td>
<td>4.7</td>
<td>1.4</td>
<td>Iris-versicolor</td>
</tr>
<tr>
<td>Iris-versicolor</td>
<td>6.5</td>
<td>2.8</td>
<td>4.6</td>
<td>1.5</td>
<td>Iris-versicolor</td>
</tr>
<tr>
<td>Iris-versicolor</td>
<td>5.6</td>
<td>2.7</td>
<td>4.2</td>
<td>1.3</td>
<td>Iris-versicolor</td>
</tr>
<tr>
<td>Iris-versicolor</td>
<td>6.3</td>
<td>3.4</td>
<td>5.6</td>
<td>2.4</td>
<td>Iris-versicolor</td>
</tr>
<tr>
<td>Iris-versicolor</td>
<td>4.9</td>
<td>3.1</td>
<td>1.5</td>
<td>0.1</td>
<td>Iris-versicolor</td>
</tr>
<tr>
<td>Iris-versicolor</td>
<td>6.8</td>
<td>2.8</td>
<td>4.8</td>
<td>1.4</td>
<td>Iris-versicolor</td>
</tr>
<tr>
<td>Iris-versicolor</td>
<td>5.7</td>
<td>2.8</td>
<td>4.5</td>
<td>1.3</td>
<td>Iris-versicolor</td>
</tr>
<tr>
<td>Iris-versicolor</td>
<td>6.0</td>
<td>2.7</td>
<td>5.1</td>
<td>1.6</td>
<td>Iris-versicolor</td>
</tr>
<tr>
<td>Iris-versicolor</td>
<td>5.0</td>
<td>3.5</td>
<td>1.3</td>
<td>0.2</td>
<td>Iris-versicolor</td>
</tr>
<tr>
<td>Iris-versicolor</td>
<td>6.5</td>
<td>3.0</td>
<td>5.2</td>
<td>2.0</td>
<td>Iris-versicolor</td>
</tr>
</tbody>
</table>
2. Train the model using `ML_TRAIN`. Since this is a classification dataset, the `classification` task is specified to create a classification model:

```sql
mysql> CALL sys.ML_TRAIN('ml_data.iris_train', 'class', 137;
```
When the training operation finishes, the model handle is assigned to the `@iris_model` session variable, and the model is stored in your model catalog. You can view the entry in your model catalog using the following query, where `user1` is your MySQL account name:

```sql
SELECT model_id, model_handle, train_table_name FROM ML_SCHEMA_user1.MODEL_CATALOG;
```

```
+----------+---------------------------------------+--------------------+
<table>
<thead>
<tr>
<th>model_id</th>
<th>model_handle</th>
<th>train_table_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>ml_data.iris_train_user1_1648140791</td>
<td>ml_data.iris_train</td>
</tr>
</tbody>
</table>
+----------+---------------------------------------+--------------------+
```

3. Load the model into HeatWave ML using `ML_MODEL_LOAD` routine:

```sql
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.

4. 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:

```sql
SET @row_input = JSON_OBJECT(
  "sepal length", 7.3,
  "sepal width", 2.9,
  "petal length", 6.3,
  "petal width", 1.8);

SELECT sys.ML_PREDICT_ROW(@row_input, @iris_model);
```

```
+--------------------------------------------------------------------------------------------------+
<table>
<thead>
<tr>
<th>sys.ML_PREDICT_ROW(@row_input, @iris_model)</th>
</tr>
</thead>
<tbody>
<tr>
<td>{&quot;Prediction&quot;: &quot;Iris-virginica&quot;, &quot;petal width&quot;: 1.8, &quot;sepal width&quot;: 2.9,</td>
</tr>
<tr>
<td>&quot;petal length&quot;: 6.3, &quot;sepal length&quot;: 7.3}</td>
</tr>
</tbody>
</table>
+--------------------------------------------------------------------------------------------------+
```

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.

5. Now, generate an explanation for the same row of data using the `ML_EXPLAIN_ROW` routine to understand how the prediction was made:

```sql
SELECT sys.ML_EXPLAIN_ROW(@row_input, @iris_model);
```

```
+--------------------------------------------------------------------------------------------------+
<table>
<thead>
<tr>
<th>sys.ML_EXPLAIN_ROW(@row_input, @iris_model)</th>
</tr>
</thead>
<tbody>
<tr>
<td>{&quot;Prediction&quot;: &quot;Iris-virginica&quot;, &quot;petal width&quot;: 1.8, &quot;sepal width&quot;: 2.9,</td>
</tr>
<tr>
<td>&quot;petal length&quot;: 6.3, &quot;sepal length&quot;: 7.3, &quot;petal width_attribution&quot;: 0.2496,</td>
</tr>
<tr>
<td>&quot;petal length_attribution&quot;: 0.9997}</td>
</tr>
</tbody>
</table>
+--------------------------------------------------------------------------------------------------+
```

The attribution values show which features contributed most to the prediction, with petal length and pedal width being the most important features. The other features have a 0 value indicating that they did not contribute to the prediction.

6. 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.

```sql
CALL sys.ML_PREDICT_TABLE('ml_data.iris_test', @iris_model, 'ml_data.iris_predictions');
```

To view `ML_PREDICT_TABLE` results, query the output table; for example:

```sql
SELECT * FROM iris_predictions LIMIT 3
```
The table shows the predictions and the feature column values used to make each prediction.

7. Generate explanations for the same table of data using the `ML_EXPLAIN_TABLE` routine.

```sql
mysql> CALL sys.ML_EXPLAIN_TABLE('ml_data.iris_test', @iris_model, 'ml_data.iris_explanations');
```

To view `ML_EXPLAIN_TABLE` results, query the output table; for example:

```sql
mysql> SELECT * FROM iris_explanations LIMIT 3;
```

8. Score the model using `ML_SCORE` to assess the model's reliability. This example uses the `balanced_accuracy` metric, which is one of the many scoring metrics supported by HeatWave ML.

```sql
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.

```sql
mysql> SELECT @score;
```

<table>
<thead>
<tr>
<th>@score</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.9583333134651184</td>
</tr>
<tr>
<td>-------------------</td>
</tr>
</tbody>
</table>
9. **Unload the model using** `ML_MODEL_UNLOAD`:

```sql
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.