HeatWave User Guide

Abstract

This is the *HeatWave User Guide*. This document provides information and procedures about loading data into HeatWave and running queries. For information about creating and managing a HeatWave cluster, refer to the MySQL Database Service User Guide.

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

For legal information, see the Legal Notices.

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

Document generated on: 2021-10-18 (revision: 71116)

Table of Contents

Preface and Legal Notices

This is the *HeatWave User Guide*. This document provides information and procedures about loading data into HeatWave and running queries. For information about creating and managing a HeatWave cluster, refer to the MySQL Database Service User Guide.

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

Legal Notices

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 Introduction

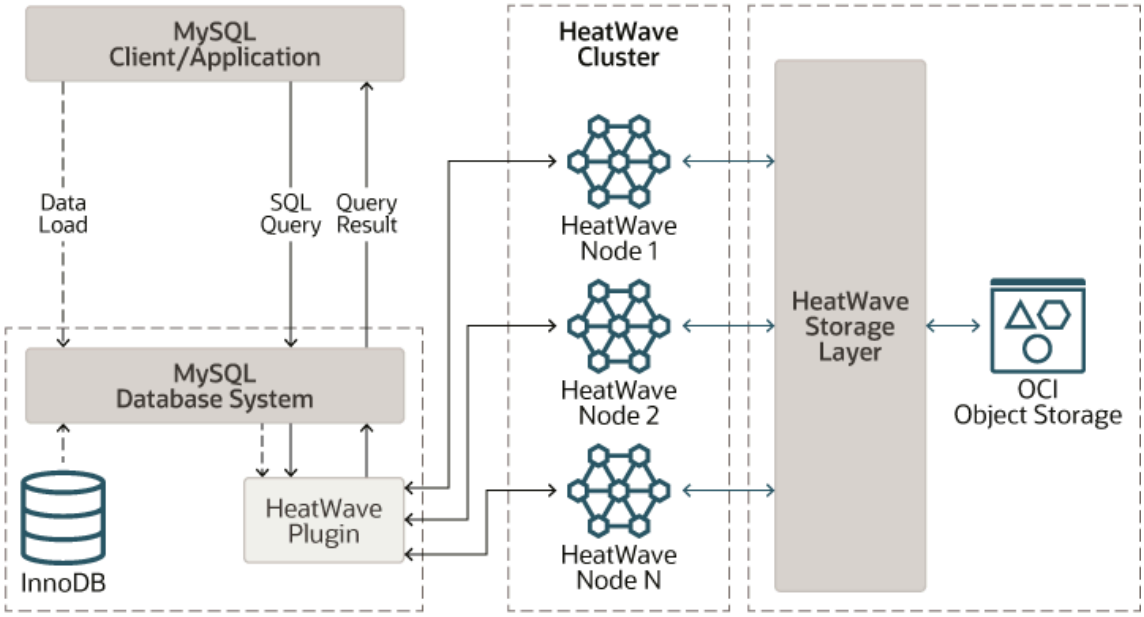
Table of Contents

HeatWave is a distributed, scalable, shared-nothing, in-memory, hybrid columnar, query processing engine designed for extreme performance. It is enabled when you add a HeatWave cluster to a MySQL DB System.

A HeatWave cluster includes a MySQL DB System node and two or more HeatWave nodes. The MySQL DB System node has a HeatWave plugin that is responsible for cluster management, loading data into the HeatWave cluster, query scheduling, and returning query results to the MySQL DB System. HeatWave nodes store data in memory and process analytics queries. Each HeatWave node contains an instance of the HeatWave query processing engine (`RAPID`).

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.

Figure 1.1 HeatWave Architecture



Queries are issued from a MySQL client or application that is connected to the MySQL DB System node. Clients and applications do not connect to HeatWave directly. Supported queries are automatically offloaded from the MySQL DB System to HeatWave for accelerated processing. Results are returned to the MySQL DB System node and to the MySQL client or application that issued the query. For more information, see Chapter 6, *Running Queries*.

Loading data into HeatWave involves preparing tables on the MySQL DB System and executing load commands. Preparing tables includes tasks such as excluding columns, defining string column encodings, adding data placement keys, and marking the tables as secondary engine candidates. (`InnoDB` is the primary engine.) For more information, see Chapter 3, *Preparing Data*, and Chapter 4, *Loading Data*.

The *Auto Parallel Load* utility facilitates the process of preparing and loading tables by automating required steps and optimizing the number of parallel load threads. See Section 4.1, “Auto Parallel Load”.

When HeatWave loads a table, the data is shareded and distributed among HeatWave nodes. Once 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 4.2, “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 queries on the data, 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 Chapter 8, *Workload Optimization using Advisor*.

1.1 Architecture

This section provides an overview of HeatWave architectural features.

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

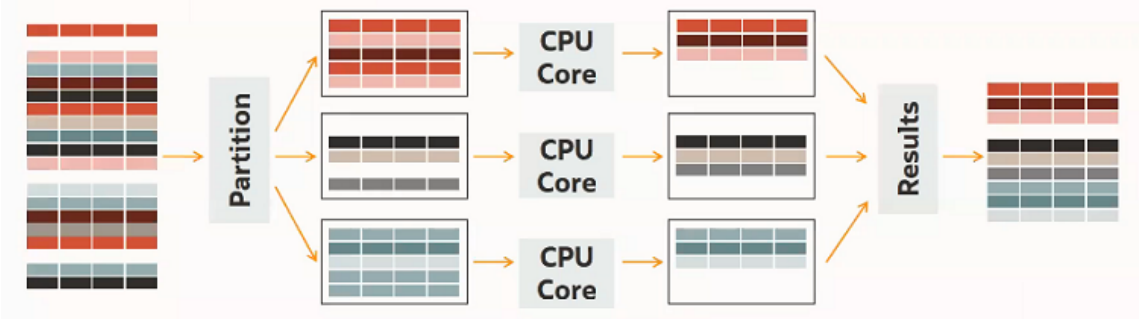
In-Memory Hybrid-Columnar Format

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.

Figure 1.2 HeatWave Massively Parallel Architecture



Push-Based Vectorized Query Processing

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

Scale-Out Data Management

When 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 Database Service provides a single data management platform for OLTP and analytics. HeatWave is designed as a pluggable MySQL storage engine, which enables management of both the MySQL Database Service and HeatWave using the same interfaces, including the OCI console, REST API, and the command line.

Changes to the 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 node that is part of the HeatWave cluster. Users connect through 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.

Once a query is submitted to the MySQL database, the MySQL query optimizer transparently decides if the query should be offloaded to HeatWave for accelerated execution, based on whether query offload prerequisites are met. Offloaded queries are pushed to HeatWave nodes for processing. Query results are sent back to the MySQL database node and to the issuing client or application.

1.2 MySQL Autopilot

MySQL Autopilot automates many of the most important and often challenging aspects of achieving high query performance at scale, including provisioning, loading data, query execution, 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. These machine learning models are then 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

- Failure Handling

System Setup

- *Auto Provisioning*

  Estimates the number of HeatWave nodes required for a workload by sampling the data, which means that manual cluster size estimations are not necessary. See HeatWave Cluster Size Estimates.

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 4.1, “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 8.1, “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 8.2, “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 8.3, “Query Insights”.

- *Auto Change Propagation*

  Intelligently determines the optimal time when changes in MySQL DB System should be propagated to the HeatWave Storage Layer.

- *Auto Scheduling*

  Identifies short running queries and prioritizes them over long running queries in an intelligent way to reduce overall query execution wait times. See Auto Scheduling.

Failure Handling

- *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 Before You Begin

Before you begin using HeatWave, the following is assumed:

- You have an operational MySQL Database Service and you are able to connect to it using a MySQL client. If not, complete the steps described in Getting Started with MySQL Database Service, in the MySQL Database Service User Guide.
- The data you want to query using HeatWave is available on the MySQL DB System. Data must be available on the MySQL DB System before it can be loaded into the HeatWave cluster. For information about importing data into a MySQL DB System, see Importing and Exporting Databases, in the MySQL Database Service User Guide.
- You, or your group, have been granted the `mysql-heatwave` policies described in Policy Details for MySQL Database Service, in the MySQL Database Service User Guide.
- You have added a HeatWave cluster to your MySQL DB System. For instructions, see Adding a HeatWave Cluster, in the MySQL Database Service User Guide.

Chapter 3 Preparing Data

Table of Contents

This section describes how to prepare data for loading into HeatWave. Data is prepared on the MySQL DB System before it is loaded into HeatWave. For information about importing data into a MySQL DB System, refer to the MySQL Database Service User Guide.

Preparing data involves:

1. Identifying the tables you want to load. See Section 3.1, “Identifying Tables to Load”.
2. Excluding table columns that are not required or have unsupported data types. See Section 3.2, “Excluding Table Columns”.
3. Encoding string columns. See Section 3.3, “Encoding String Columns”.
4. Defining Data Placement Keys. See Section 3.4, “Defining Data Placement Keys”.
5. Defining the secondary engine for tables you want to load. See Section 3.5, “Defining the Secondary Engine”.

For related best practices, see Chapter 9, *Best Practices*.

> **Tip**
>
> Instead of preparing and loading tables into HeatWave manually, consider using the Auto Parallel Load utility, which prepares and loads tables for you using an optimized number of parallel load threads. See Section 4.1, “Auto Parallel Load”.

3.1 Identifying Tables to Load

The tables you intend to query must be loaded in HeatWave. If a query accesses a table that is not loaded, the query is not offloaded to HeatWave for processing.

Before loading data, take time to identify the tables that your queries access. For example, if your queries access data in a particular schema exclusively, load the tables belonging to that schema, or, if your queries access a few particular tables, load those tables into HeatWave.

If using the Auto Parallel Load utility, you can specify the particular schemas and tables you want to load. See Section 4.1, “Auto Parallel Load”.

3.2 Excluding Table Columns

Before loading a table into HeatWave, identify table columns to exclude. Columns to exclude are:

- Columns with unsupported data types. *It is required that these columns are excluded*; otherwise, the table cannot be loaded. For a list of data types that HeatWave supports, see Section 11.1, “Supported Data Types”.

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

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

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

The Auto Parallel Load utility excludes columns with unsupported data types automatically and permits excluding irrelevant columns explicitly. For more information, see Section 4.1, “Auto Parallel Load”.

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

3.3 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:

```
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 Chapter 8, *Workload Optimization using Advisor*.

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 3.1 Column Encoding Type Characteristics

Encoding Type	Expression, Filter, Function, and Operator Support	Best Suited To	Space Required On
`Variable-length (VARLEN)`	Supports `JOIN` operations, string functions and operators, and `LIKE` predicates. See Section 11.4.1, “Variable-length Encoding”.	Columns with a high number of distinct values	HeatWave nodes
`Dictionary (SORTED)`	Does not support `JOIN` operations, string functions and operators, or `LIKE` predicates.	Columns with a low number of distinct values	MySQL DB System node

For additional information about string column encoding, see Section 11.4, “String Column Encoding Reference”.

3.4 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 Chapter 8, *Workload Optimization using Advisor*.

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

```
RAPID_COLUMN=DATA_PLACEMENT_KEY=N
```

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

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

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

```
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';
```

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.

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

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

> **Note**
>
> Other information is permitted in column comments. For example, it is permitted to specify a column description alongside a data placement keyword string:
>
> ```
> COMMENT 'column_description RAPID_COLUMN=DATA_PLACEMENT_KEY=1'
> ```

Usage notes:

- `JOIN` and `GROUP BY` query optimizations are only applied if at least one of the `JOIN` or `GROUP BY` relations has a key that matches the defined data placement key.
- If a `JOIN` operation can be executed with or without the `JOIN` and `GROUP BY` query optimization, a compilation-time cost model determines how the query is executed. The cost model uses estimated statistics.
- A data placement key cannot be defined on a dictionary-encoded string column but are permitted on variable-length encoded columns. HeatWave applies variable-length encoding to string columns by default. See Section 3.3, “Encoding String Columns”.
- A data placement key can only be defined on a column with a supported data type. See Section 11.1, “Supported Data Types”.
- A data placement key column cannot be defined as a `NOT SECONDARY` column. See Section 3.2, “Excluding Table Columns”.
- For related metadata queries, see Section 11.5, “Metadata Queries”.

3.5 Defining the Secondary Engine

Tables on a MySQL DB System are defined with `InnoDB` as the primary storage engine. For each table that you want to load into HeatWave, you must define the HeatWave query processing engine (`RAPID`) as the secondary engine.

To define `RAPID` as the secondary engine for a table, specify the `SECONDARY_ENGINE` table option in a `CREATE TABLE` or `ALTER TABLE` statement:

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

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

> **Tip**
>
> Instead of defining the secondary storage engine for each table manually, consider using the Auto Parallel Load utility to prepare and load tables. For more information, see Section 4.1, “Auto Parallel Load”.

Chapter 4 Loading Data

Table of Contents

This section describes how to load data into HeatWave. Before attempting to load data, ensure that you have:

- Loaded the data into the MySQL DB System. Data is loaded into HeatWave from the MySQL DB System. For information about importing data into a MySQL DB System, refer to the MySQL Database Service User Guide.
- Defined a primary key on each table you intend to load.
- Excluded columns with unsupported data types. See Section 3.2, “Excluding Table Columns”.
- Defined `RAPID` as the secondary engine. See Section 3.5, “Defining the Secondary Engine”.

Otherwise, the table load operation is not permitted.

For related best practices, see Chapter 9, *Best Practices*.

> **Tip**
>
> Instead of preparing and loading tables into HeatWave manually, consider using the Auto Parallel Load utility. See Section 4.1, “Auto Parallel Load”.

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

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

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

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

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

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

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

```
| tpch.orders                  | AVAIL_RPDGSTABSTATE |
| tpch.lineitem                | AVAIL_RPDGSTABSTATE |
| tpch.customer                | AVAIL_RPDGSTABSTATE |
| tpch.nation                  | AVAIL_RPDGSTABSTATE |
| tpch.region                  | AVAIL_RPDGSTABSTATE |
| tpch.part                    | AVAIL_RPDGSTABSTATE |
+------------------------------+---------------------+
```

The `AVAIL_RPDGSTABSTATE` status indicates that the table is loaded. For information about load statuses, see Section 11.10.4, “The rpd_tables Table”.

When loading a table into HeatWave, data is read from `InnoDB` using batched, multi-threaded reads. Data 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 3.4, “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 4.2, “Change Propagation”.

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.

The following limitations apply when loading tables:

- Loading a table is not permitted if the primary key is absent. Primary key columns defined with column prefixes are not supported.
- HeatWave supports a maximum of 470 columns per table.
- 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.
- 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 Chapter 10, *Troubleshooting*.

4.1 Auto Parallel Load

Loading data into HeatWave involves several manual steps. The time required to perform these steps typically depends on the number of schemas, tables, and columns. Auto Parallel Load facilitates the process by automating many of the steps involved, including:

- Excluding schemas, tables, and columns that cannot be loaded.
- Generating a load script with DDL statements for preparing and loading each table.
- Verifying that there is sufficient memory available for the data.
- Applying options and settings affecting the data load process.
- Optimizing load parallelism based on machine-learning models.

- Loading data into HeatWave by executing the generated load script.

Auto Parallel Load 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 a list of schemas and options as arguments.

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

You can write an Auto Parallel Load `CALL` statement manually using the syntax described below, or use the Auto Parallel Load load command that is generated when performing a node count estimate in the OCI console. See Generating a Node Count Estimate in the *MySQL DB System User Guide*.

Auto Parallel Load can be run from any MySQL client or connector.

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

- Auto Parallel Load Syntax
- Auto Parallel Load Requirements
- Running Auto Parallel Load
- Dictionary Size Estimation
- The Auto Parallel Load Report Table
- Auto Parallel Load Command-Line Help
- Auto Parallel Load Examples

Auto Parallel Load Syntax

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

db_list: {
     JSON_ARRAY({""|"schema_name"["schema_name"] ...)
}

options: {
     JSON_OBJECT("key","value"["key","value"] ...)
          "key","value": {
          ["mode",{"normal"|"dryrun"}]
          ["output",{"normal"|"compact"|"silent"|"help"}]
          ["sql_mode","sql_mode"]
          ["policy",{"disable_unsupported_columns"|"not_disable_unsupported_columns"}]
          ["exclude_list",JSON_ARRAY("db_object"["db_object"] ...)]
          ["set_load_parallelism",{TRUE|FALSE}]
          ["auto_enc",JSON_OBJECT("mode",{"off"|"check"}]
}
```

`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.)
  - `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 Section 3.2, “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("db0","db1","db2","db3"),
JSON_OBJECT("exclude_list", JSON_ARRAY("db0.t1","db0.t2", "db0.t3.c1")));
```

  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 on the MySQL node for dictionary-encoded columns. 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. Dictionary-encoded columns require memory on the MySQL node for dictionaries. 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 Dictionary Size Estimation.

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.

Running Auto Parallel Load

Run Auto Parallel Load in `"dryrun"` mode first to check for errors and warnings and to inspect the generated load script. A simple Auto Parallel Load call in `"dryrun"` mode that specifies a list of schemas `("db0","db1","db2","db3")` is written as follows:

```
CALL sys.heatwave_load(JSON_ARRAY("db0","db1","db2","db3"),
JSON_OBJECT("mode","dryrun"));
```

An Auto Parallel Load call in `"dryrun"` mode with additional options specified appears as follows:

```
CALL sys.heatwave_load(JSON_ARRAY("db0","db1","db2","db3"),
JSON_OBJECT("mode","dryrun", "exclude_list", JSON_ARRAY("db0.t1","db0.t2", "db0.t3"),
"policy","disable_unsupported_columns",
"auto_enc",JSON_OBJECT("mode","check")));
```

For information about Auto Parallel Load options, see Auto Parallel Load Syntax.

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:

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

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

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

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

```
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:

```
CALL sys.heatwave_load(JSON_ARRAY("db0","db1","db2","db3"),
JSON_OBJECT("mode","normal","exclude_list",JSON_ARRAY("db0.t1","db0.t2", "db0.t3"),
"policy","disable_unsupported_columns",
"auto_enc",JSON_OBJECT("mode","check")));
```

> **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:
>
> ```
> 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.

Dictionary Size Estimation

The `auto_enc` option is run in `check` mode by default to ensure that there is enough memory on the MySQL node for dictionary-encoded string columns.

The following example uses the `auto_enc` option in `check` mode, which is useful if you have dictionary-encoded columns and want to ensure that there is enough memory on the MySQL node for the associated dictionaries before attempting a load operation. Insufficient memory on the MySQL node can cause a load failure.

```
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:

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

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

```
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:

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

- View the number of load commands generated:

```
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:

```
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.)

```
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:

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

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

Auto Parallel Load Examples

- Load the tables belonging to a single schema. No options are specified, which means that the default options are used.

```
CALL sys.heatwave_load(JSON_ARRAY("db0"),NULL);
```

- Run Auto Parallel Load in `dryrun` mode to determine if there are any warnings.

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

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

```
CALL sys.heatwave_load(JSON_ARRAY("db0","db1","db2","db3"),NULL);
```

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

```
CALL sys.heatwave_load(JSON_ARRAY("db0","db1","db2","db3"),
JSON_OBJECT("policy","not_disable_unsupported_columns"));
```

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

```
CALL sys.heatwave_load(JSON_ARRAY("db0","db1","db2","db3"),
JSON_OBJECT("exclude_list",JSON_ARRAY("db0.t1","db0.t2","db0.t3.c1")));
```

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

```
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.

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

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

- Call Auto Parallel Load from a stored procedure:

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

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

  -- CUSTOM OUTPUT
  SELECT log as 'Unsupported objects' FROM sys.heatwave_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();
```

4.2 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.

Changes accumulate on the MySQL DB System node and are propagated to HeatWave in batch transactions. Change propagation occurs every 200 milliseconds or when pending changes reach 64 MBs in size. Changes are applied using the `READ COMMITTED` isolation level.

A change propagation failure can cause table data in HeatWaveto become stale. Queries that access stale table data are not offloaded to HeatWave for processing. To check if change propagation is enabled globally, query the `rapid_change_propagation_status` variable:

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

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

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

If change propagation is disabled for a particular table, you must unload and reload the table. See Chapter 5, *Unloading Tables*, and Chapter 4, *Loading Data*.

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

Change propagation is aborted if dictionary-encoded string column updates cause a dictionary overflow, which occurs if the number of new unique values exceeds dictionary capacity.

Chapter 5 Unloading Tables

Unloading a table from HeatWave may be necessary to replace an existing table, to reload a table after a change propagation failure has caused data become stale, 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.

Chapter 6 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 MySQL DB System-connected MySQL client or application. (Clients and applications do not connect to HeatWave directly.) For information about connecting to a MySQL DB System, refer to the MySQL Database Service User Guide. After HeatWave processes a query, results are sent back to the MySQL DB System and to the client or application that issued the query.

Running queries is described under the following topics in this section:

- Query Offload Prerequisites
- Running a Query
- Query Runtimes and Estimates
- Tracking Scanned Data
- CREATE TABLE ... SELECT Statements
- INSERT ... SELECT Statements
- Auto Scheduling
- Auto Query Plan Improvement
- Debugging Queries

For related best practices, see Chapter 9, *Best Practices*.

Query Offload Prerequisites

The following prerequisites apply for offloading queries:

- 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 Section 3.5, “Defining the Secondary Engine”.
- All tables accessed by the query must be loaded in HeatWave. See Chapter 4, *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 11.2, “Supported Functions and Operators”.
- Queries must avoid known limitations. See Section 11.6, “Limitations”.

If any prerequisite is not satisfied, the query is not offloaded and falls back to the MySQL DB System for execution by default. This behavior is controlled by the `use_secondary_engine` variable, which is set on `ON` by default. A `use_secondary_engine=OFF` setting forces a query to execute on the MySQL DB System. A `use_secondary_engine=FORCED` setting forces a query to execute on HeatWave or fail if that is not possible.

Running a Query

Before running a query, use `EXPLAIN` to determine if the query can be offloaded. If so, the `Extra` column of `EXPLAIN` output shows: "`Using secondary engine RAPID`". If that information does not appear, the query cannot 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\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: 14862970
     filtered: 33.33
        Extra: Using where; Using temporary; Using filesort; Using secondary
       engine RAPID
```

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

> **Note**
>
> For information about obtaining HeatWave query runtime estimates before running a query, see Query Runtimes and Estimates.

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

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

```
mysql> SET SESSION use_secondary_engine=OFF;

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

```
| 1-URGENT        |     2017573 |
| 2-HIGH          |     2015859 |
| 3-MEDIUM        |     2013174 |
| 4-NOT SPECIFIED |     2014476 |
| 5-LOW           |     2013674 |
+-----------------+-------------+
5 rows in set (8.91 sec)
```

If a query does not offload and you cannot determine why, refer to Chapter 10, *Troubleshooting*, or try debugging the query using the procedure described in Debugging Queries.

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

Query Runtimes and Estimates

HeatWave query runtimes and runtime estimates can be viewed 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 200 executed queries.

Using Query Insights

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

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

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

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

For additional information about using *Query Insights*, see Section 8.3, “Query Insights”.

Using the rpd_query_stats Table

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

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

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

```
mysql> SELECT query_id,
```

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

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

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

Tracking Scanned Data

HeatWave tracks the amount of data scanned by all queries and by individual queries.

Data Scanned By All Queries

To view a cumulative total of data scanned (in MBs) by all successfully executed HeatWave queries from the time the HeatWave cluster was last started, query the `hw_data_scanned` global status variable. For example:

```
mysql> SHOW GLOBAL STATUS LIKE 'hw_data_scanned';
+-----------------+-------+
| Variable_name   | Value |
+-----------------+-------+
| hw_data_scanned | 66    |
+-----------------+-------+
```

The cumulative total does not include data scanned by failed queries, queries that were not offloaded to HeatWave, or `EXPLAIN` queries.

The `hw_data_scanned` value is reset to 0 only when the HeatWave cluster is restarted.

If a subset of HeatWave nodes go offline, HeatWave retains the cumulative total of scanned data as long as the HeatWave cluster remains in an active state. When the HeatWave cluster becomes fully operational and starts processing queries again, HeatWave resumes tracking the amount of data scanned, adding to the cumulative total.

Data Scanned By Individual Queries

To view the amount of data scanned by an individual HeatWave query or to view an estimate for the amount of data that would be scanned by a query run with `EXPLAIN`, run the query and query the `totalBaseDataScanned` field in the `QKRN_TEXT` column of the `performance_schema.rpd_query_stats` table:

```
mysql> SELECT query_id,
       JSON_EXTRACT(JSON_UNQUOTE(qkrn_text->'$**.sessionId'),'$[0]') AS session_id,
       JSON_EXTRACT(JSON_UNQUOTE(qkrn_text->'$**.totalBaseDataScanned'), '$[0]') AS data_scanned,
       JSON_EXTRACT(JSON_UNQUOTE(qexec_text->'$**.error'),'$[0]') AS error_message
       FROM performance_schema.rpd_query_stats;
+----------+------------+--------------+---------------+
| query_id | session_id | data_scanned | error_message |
+----------+------------+--------------+---------------+
|        1 | 8          | 66           | ""            |
+----------+------------+--------------+---------------+
```

The example above retrieves any error message associated with the query ID. If a query fails or was interrupted, the number of bytes scanned by the failed or interrupted query and the associated error message are returned, as shown in the following examples:

```
mysql> SELECT query_id,
```

```
      JSON_EXTRACT(JSON_UNQUOTE(qkrn_text->'$**.sessionId'),'$[0]') AS session_id,
      JSON_EXTRACT(JSON_UNQUOTE(qkrn_text->'$**.totalBaseDataScanned'), '$[0]') AS data_scanned,
      JSON_EXTRACT(JSON_UNQUOTE(qexec_text->'$**.error'),'$[0]') AS error_message
      FROM performance_schema.rpd_query_stats;
+----------+------------+--------------+----------------------------------------+
| query_id | session_id | data_scanned | error_message                          |
+----------+------------+--------------+----------------------------------------+
|        1 | 8          | 461          | "Operation was interrupted by the user." |
+----------+------------+--------------+----------------------------------------+
```

```
mysql> SELECT query_id,
      JSON_EXTRACT(JSON_UNQUOTE(qkrn_text->'$**.sessionId'),'$[0]') AS session_id,
      JSON_EXTRACT(JSON_UNQUOTE(qkrn_text->'$**.totalBaseDataScanned'), '$[0]') AS data_scanned,
      JSON_EXTRACT(JSON_UNQUOTE(qexec_text->'$**.error'),'$[0]') AS error_message
      FROM performance_schema.rpd_query_stats;
+----------+------------+--------------+-----------------------------------------------+
| query_id | session_id | data_scanned | error_message                                 |
+----------+------------+--------------+-----------------------------------------------+
|        1 | 8          | 987          | "Out of memory error during query execution in|
|          |            |              |  RAPID."                                      |
+----------+------------+--------------+-----------------------------------------------+
```

CREATE TABLE ... SELECT Statements

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

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

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

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

INSERT ... SELECT Statements

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

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

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

Usage notes:

- The `SELECT` portion of the `INSERT ... SELECT` statement is subject to the same HeatWave requirements and limitations as regular `SELECT` queries.
- Functions, operators, and attributes deprecated by MySQL Server are not supported in the `SELECT` query.

- The `ON DUPLICATE KEY UPDATE` clause is not supported.
- `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.

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.

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 11.5, “Metadata Queries”.

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 actual statistics collected from previously executed queries are used instead of estimated statistics, which improves query execution plans, cost estimations, execution times, and memory efficiency.

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

The statics 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 OCI-managed `rapid_stats_cache_max_entries` setting.

Debugging Queries

This section describes how to enable query tracing, and how to query the `INFORMATION_SCHEMA.OPTIMIZER_TRACE` table for information about why a query is not offloaded to HeatWave for processing.

In the following example, optimizer trace data is retrieved for a query that uses the `LAST_DAY()` function, which is currently not supported.

1. Enable tracing by setting the `optimizer_trace` and `optimizer_trace_offset` variables:

```
mysql> SET SESSION optimizer_trace="enabled=on";
```

```
mysql> SET optimizer_trace_offset=-2;
```

2. Issue the query with `EXPLAIN`. If the `Extra` column does not show "`Using secondary engine RAPID`", the query cannot be offloaded. For example:

```
mysql> EXPLAIN SELECT LAST_DAY(O_ORDERDATE) 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: 1488913
     filtered: 100.00
        Extra: NULL
1 row in set, 1 warning (0.00 sec)
```

3. Query the `INFORMATION_SCHEMA.OPTIMIZER_TRACE` table for offload failure information:

```
mysql> SELECT QUERY, TRACE->'$**.Rapid_Offload_Fails' FROM INFORMATION_SCHEMA.OPTIMIZER_TRACE;
+-----------------------------------------------+-----------------------------------------+
|QUERY                                          |TRACE->'$**.Rapid_Offload_Fails'         |
+-----------------------------------------------+-----------------------------------------+
|EXPLAIN SELECT LAST_DAY(O_ORDERDATE) FROM orders |[{"Reason": "ConstructQkrnExprTree():     |
|                                               |  "Function lastday is not yet           |
|                                               |   supported"}], {"Reason": "Currently   |
|                                               |   unsupported RAPID query compilation   |
|                                               |   scenario from physical MySQL plan"}]  |
+-----------------------------------------------+-----------------------------------------+
```

The reason reported for the offload failure depends on the limitation encountered. For the most 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 small queries of little cost from being offloaded to HeatWave. For information about the query cost threshold, see Chapter 10, *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 Section 3.2, "Excluding Table Columns".

Chapter 7 Table Load and Query Example

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

> **Tip**
>
> Instead of preparing and loading tables into HeatWave manually, consider using the Auto Parallel Load utility. For more information, see Section 4.1, “Auto Parallel Load”.

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\G
*************************** 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\G
*************************** 1. row ***************************
       Table: orders
Create Table: CREATE TABLE `orders` (
  `O_ORDERKEY` int NOT NULL,
```

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

# 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\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: 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;
+-----------------+-------------+
| O_ORDERPRIORITY | ORDER_COUNT |
+-----------------+-------------+
| 1-URGENT        |     2017573 |
| 2-HIGH          |     2015859 |
| 3-MEDIUM        |     2013174 |
| 4-NOT SPECIFIED |     2014476 |
| 5-LOW           |     2013674 |
+-----------------+-------------+
5 rows in set (0.04 sec)

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

mysql> SET SESSION use_secondary_engine=OFF;

mysql> SELECT O_ORDERPRIORITY, COUNT(*) AS ORDER_COUNT FROM orders
       WHERE O_ORDERDATE >= DATE '1994-03-01' GROUP BY O_ORDERPRIORITY
       ORDER BY O_ORDERPRIORITY;
+-----------------+-------------+
| O_ORDERPRIORITY | ORDER_COUNT |
+-----------------+-------------+
| 1-URGENT        |     2017573 |
```

```
| 2-HIGH          |     2015859 |
| 3-MEDIUM        |     2013174 |
| 4-NOT SPECIFIED |     2014476 |
| 5-LOW           |     2013674 |
+-----------------+-------------+
5 rows in set (8.91 sec)
```

Chapter 8 Workload Optimization using Advisor

Table of Contents

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 8.1, “Auto Encoding”.

- *Auto Data Placement*

  Recommends data placement keys for optimizing `JOIN` and `GROUP BY` query performance. See Section 8.2, “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 8.3, “Query Insights”.

Advisor is workload-aware and provides recommendations based on machine learning models, data analysis, and HeatWave query history. Advisor analyzes the last 200 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.

8.1 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 3.3, “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"["schema_name"]}]
        ["exclude_query",JSON_ARRAY("query_id"["query_id"] ...)]
        ["query_session_id",JSON_ARRAY("query_session_id"["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 8.4, “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 8.5, “Advisor Report Table”.)
  - `silent`: Sends output to the `heatwave_advisor_report` table only. (See Section 8.5, “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 8.4, “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 8.4, “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 8.4, “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 11.4.2, “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

```
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 3.3, “Encoding String Columns”.)

```
CALL sys.heatwave_advisor(JSON_OBJECT("auto_enc",JSON_OBJECT("mode","recommend","fixed_enc",
JSON_OBJECT("tpch.CUSTOMER.C_ADDRESS","varlen"))));
```

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

```
mysql> CALL sys.heatwave_advisor(JSON_OBJECT("target_schema",JSON_ARRAY("tpch_1024"),
       "auto_enc",JSON_OBJECT("mode","recommend")));
+------------------------------+
| INITIALIZING HEATWAVE ADVISOR |
+------------------------------+
```

```
| Version: 1.12                |
|                              |
| Output Mode: normal          |
| Excluded Queries: 0          |
| Target Schemas: 1            |
|                              |
+------------------------------+
6 rows in set (0.01 sec)

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

+-------------------------------------------------------------------------------------------------+
| ENCODING SUGGESTIONS                                                                            |
+-------------------------------------------------------------------------------------------------+
| Total Auto Encoding suggestions produced for 21 columns                                         |
| Queries executed: 200                                                                           |
|   Total query execution time: 28.82 min                                                         |
|   Most recent query executed on: Tuesday 8th June 2021 14:42:13                                 |
|   Oldest query executed on: Tuesday 8th June 2021 14:11:45                                      |
|                                                                                                 |
|                                                      CURRENT           SUGGESTED            |
| COLUMN                                                COLUMN              COLUMN            |
| NAME                                                 ENCODING           ENCODING            |
| ------                                               --------           ---------           |
| `tpch_1024`.`CUSTOMER`.`C_ADDRESS`                     VARLEN          DICTIONARY           |
| `tpch_1024`.`CUSTOMER`.`C_COMMENT`                     VARLEN          DICTIONARY           |
| `tpch_1024`.`CUSTOMER`.`C_MKTSEGMENT`                  VARLEN          DICTIONARY           |
| `tpch_1024`.`CUSTOMER`.`C_NAME`                        VARLEN          DICTIONARY           |
| `tpch_1024`.`LINEITEM`.`L_COMMENT`                     VARLEN          DICTIONARY           |
| `tpch_1024`.`LINEITEM`.`L_SHIPINSTRUCT`                VARLEN          DICTIONARY           |
| `tpch_1024`.`LINEITEM`.`L_SHIPMODE`                    VARLEN          DICTIONARY           |
| `tpch_1024`.`NATION`.`N_COMMENT`                       VARLEN          DICTIONARY           |
| `tpch_1024`.`NATION`.`N_NAME`                          VARLEN          DICTIONARY           |
| `tpch_1024`.`ORDERS`.`O_CLERK`                         VARLEN          DICTIONARY           |
| `tpch_1024`.`ORDERS`.`O_ORDERPRIORITY`                 VARLEN          DICTIONARY           |
| `tpch_1024`.`PART`.`P_BRAND`                           VARLEN          DICTIONARY           |
| `tpch_1024`.`PART`.`P_COMMENT`                         VARLEN          DICTIONARY           |
| `tpch_1024`.`PART`.`P_CONTAINER`                       VARLEN          DICTIONARY           |
| `tpch_1024`.`PART`.`P_MFGR`                            VARLEN          DICTIONARY           |
| `tpch_1024`.`PARTSUPP`.`PS_COMMENT`                    VARLEN          DICTIONARY           |
| `tpch_1024`.`REGION`.`R_COMMENT`                       VARLEN          DICTIONARY           |
| `tpch_1024`.`REGION`.`R_NAME`                          VARLEN          DICTIONARY           |
| `tpch_1024`.`SUPPLIER`.`S_ADDRESS`                     VARLEN          DICTIONARY           |
| `tpch_1024`.`SUPPLIER`.`S_NAME`                        VARLEN          DICTIONARY           |
| `tpch_1024`.`SUPPLIER`.`S_PHONE`                       VARLEN          DICTIONARY           |
|                                                                                                 |
| Applying the suggested encodings might improve query performance and cluster memory usage.      |
|   Estimated HeatWave cluster memory savings: 355.60 GiB                                         |
|                                                                                                 |
+-------------------------------------------------------------------------------------------------+
35 rows in set (18.18 sec)

+-------------------------------------------------------------------------------------------------+
| SCRIPT GENERATION                                                                               |
+-------------------------------------------------------------------------------------------------+
```

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

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

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

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

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

Usage Notes:

- Auto Encoding analyzes string columns (`CHAR`, `VARCHAR`, and `TEXT`-type columns) of tables that are loaded in HeatWave. Automatically or manually excluded columns, columns greater than 8000 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 11.4.2, “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.

8.2 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 3.4, “Defining Data Placement Keys”.

Auto Data Placement 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"["schema_name"]}]
        ["exclude_query",JSON_ARRAY("query_id"["query_id"] ...)]
        ["query_session_id",JSON_ARRAY("query_session_id"["query_session_id"] ...)]
        ["auto_dp",JSON_OBJECT(auto_dp_option)]
}

  auto_dp_option: {
                 ["benefit_threshold",N]
                 ["max_combinations",N]
 }
```

For syntax examples, see Section 8.4, “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 8.5, “Advisor Report Table”.)
  - `silent`: Sends output to the `heatwave_advisor_report` table only. (See Section 8.5, “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 8.4, “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 8.4, “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 8.4, “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 |
+------------------------------+
| Version: 1.12                |
|                              |
| Output Mode: normal          |
| Excluded Queries: 0          |
| Target Schemas: 1            |
|                              |
+------------------------------+
```

```
6 rows in set (0.01 sec)

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

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

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

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

```
+-----------------------------------------------------------------------------------------+
9 rows in set (16.43 sec)


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

Usage Notes:

- If a table already has data placement keys or columns are customized 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.

8.3 Query Insights

Query Insights provides:

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

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

Query Insights 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"["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 8.4, “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 8.5, “Advisor Report Table”.)
  - `silent`: Sends output to the `heatwave_advisor_report` table only. (See Section 8.5, “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 8.4, “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 8.4, “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 8.4, “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 200 queries, which is the HeatWave query history limit. To view the current HeatWave query history, query the `performance_schema.rpd_query_stats` table:

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

The following example shows how to retrieve runtime data for the entire query history using 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.

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

+-----------------------------------------------------+
| ANALYZING LOADED DATA                               |
+-----------------------------------------------------+
```

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

+-----------------------------------------------------------------------------------------+
| QUERY INSIGHTS                                                                          |
+-----------------------------------------------------------------------------------------+
| Queries executed on Heatwave: 4                                                         |
| Session IDs (as filter): None                                                           |
|                                                                                         |
| QUERY-ID  SESSION-ID  QUERY-STRING             EXEC-RUNTIME  COMMENT                    |
| --------  ----------  ------------             ------------  -------                    |
|        1          32  SELECT COUNT(*)                                                   |
|                       FROM tpch128.LINEITEM    0.628                                    |
|        2          32  SELECT COUNT(*)                                                   |
|                       FROM tpch128.ORDERS      0.114 (est.)  Explain.                   |
|        3          32  SELECT COUNT(*)                                                   |
|                       FROM tpch128.ORDERS,                                              |
|                       tpch128.LINEITEM         5.207 (est.)  Out of memory              |
|                                                              error during               |
|                                                              query execution            |
|                                                              in RAPID.                  |
|        4          32  SELECT COUNT(*)                                                   |
|                       FROM tpch128.SUPPLIER,                                            |
|                       tpch128.LINEITEM         3.478 (est.)  Operation was              |
|                                                              interrupted by             |
|                                                              the user.                  |
| TOTAL ESTIMATED:   3   EXEC-RUNTIME:       8.798 sec                                    |
| TOTAL EXECUTED:    1   EXEC-RUNTIME:       0.628 sec                                    |
|                                                                                         |
|                                                                                         |
| Retrieve detailed query statistics using the query below:                               |
|     SELECT log FROM sys.heatwave_advisor_report WHERE stage = "QUERY_INSIGHTS" AND      |
|     type = "info";                                                                      |
|                                                                                         |
+-----------------------------------------------------------------------------------------+

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

For Query Insights `CALL` statements examples that filter results by schema and session ID, see Section 8.4, “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.

Query Insights data can be retrieved in machine readable format for use in scripts; see Section 8.4, “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 8.5, “Advisor Report Table”.

8.4 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

- 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:

```
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:

```
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.

```
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:

```
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:

```
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:

```
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.

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

8.5 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:

```
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.)

```
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:

```
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;
```

Chapter 9 Best Practices

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

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

Provisioning

To determine the appropriate HeatWave cluster size for a workload, generate a node count estimate in the console. Node count 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, see Generating a Node Count Estimate, in the MySQL Database Service User Guide.

Generate a node count 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 node count 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 Database Service; useful for all types of exports and imports. MySQL Shell supports export to, and import from, Object Storage. The minimum supported source version of MySQL is 5.7.9. For more information, see Importing and Exporting Databases, in the MySQL Database Service User 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.

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 4.1, “Auto Parallel Load”.

  > **Tip**
  >
  > An Auto Parallel Load command is generated in the console when performing a node count estimate. For more information, see Generating a Node Count Estimate.

- To minimize the number of HeatWave nodes required for your data, exclude table columns that are not accessed by your queries. For information about excluding columns, see Section 3.2, “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 8.1, “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 8.2, “Auto Data Placement”.

Loading Data

Instead of preparing and loading tables into HeatWave manually, consider using the Auto Parallel Load utility. See Section 4.1, “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 after a failure or intended stoppage. 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 Chapter 5, *Unloading Tables*.
  2. Importing data into the table on the MySQL DB System node. See Importing and Exporting Databases in the MySQL Database Service User Guide.
  3. Performing a `SECONDARY_LOAD` operation to reload the table into HeatWave. See Chapter 4, *Loading Data*.

  Depending on the amount of data, an incremental bulk load may be a more expedient 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 4.2, “Change Propagation”.

Use the following strategies to improve load performance:

- *Increase the number of read threads*

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

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

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

> **Tip**
>
> The Auto Parallel Load utility automatically optimizes the number of parallel read threads for each table. See Section 4.1, “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:
mysql> ALTER TABLE supplier SECONDARY_LOAD;

Session 2:
mysql> ALTER TABLE parts SECONDARY_LOAD;

Session 3:
mysql> ALTER TABLE region SECONDARY_LOAD;

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

- *Avoid or reduce conflicting operations*

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

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 Chapter 8, *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:

```
SET optimizer_trace_max_mem_size=1000000;
```

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

```
mysql> EXPLAIN SELECT COUNT(*) FROM orders o WHERE o_totalprice> (SELECT AVG(o_totalprice)
       FROM orders WHERE o_custkey=o.o_custkey)\G
*************************** 1. row ***************************
           id: 1
  select_type: PRIMARY
        table: o
   partitions: NULL
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 14862970
     filtered: 100.00
        Extra: Using where
*************************** 2. row ***************************
           id: 2
  select_type: DEPENDENT SUBQUERY
        table: orders
   partitions: NULL
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 14862970
     filtered: 10.00
        Extra: Using where
2 rows in set, 2 warnings (0.00 sec)
```

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

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

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

  - Using a SET statement before running queries:

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

  - Using a SET_VAR optimizer hint when issuing a query:

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

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

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

```
mysql> SELECT s_nationkey, s_suppkey, l_comment FROM lineitem, supplier, nation
       WHERE s_suppkey = l_suppkey LIMIT 10;
ERROR 3015 (HY000): Out of memory in storage engine 'Failure detected in RAPID; query
```

```
execution cannot proceed'.
```

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

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

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

```
mysql> SELECT o_orderkey, c_custkey, l_shipdate, s_nationkey, s_suppkey, l_comment
       FROM lineitem, supplier, nation, customer, orders
       WHERE c_custkey = o_custkey AND o_orderkey = l_orderkey
       AND c_nationkey = s_nationkey AND c_nationkey = n_nationkey AND c_custkey < 3000000
       LIMIT 10;
ERROR 3015 (HY000): Out of memory in storage engine 'Failure detected in RAPID;
query execution cannot proceed'.
```

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

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

```
     filtered: 5.00
        Extra: Using where; Using join buffer (hash join); Using secondary engine RAPID
*************************** 4. row ***************************
           id: 1
  select_type: SIMPLE
        table: orders
   partitions: NULL
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 14862970
     filtered: 10.00
        Extra: Using where; Using join buffer (hash join); Using secondary engine RAPID
*************************** 5. row ***************************
           id: 1
  select_type: SIMPLE
        table: lineitem
   partitions: NULL
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 56834662
     filtered: 10.00
        Extra: Using where; Using join buffer (hash join); Using secondary engine RAPID
```

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

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

```
        table: nation
   partitions: NULL
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 25
     filtered: 10.00
        Extra: Using where; Using join buffer (hash join); Using secondary engine RAPID
*************************** 4. row ***************************
           id: 1
  select_type: SIMPLE
        table: orders
   partitions: NULL
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 14862970
     filtered: 10.00
        Extra: Using where; Using join buffer (hash join); Using secondary engine RAPID
*************************** 5. row ***************************
           id: 1
  select_type: SIMPLE
        table: lineitem
   partitions: NULL
         type: ALL
possible_keys: NULL
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 56834662
     filtered: 10.00
        Extra: Using where; Using join buffer (hash join); Using secondary engine RAPID
```

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

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

5. Check the size of your data by performing a node count estimate. If your data has grown substantially, the HeatWave cluster may require additional nodes. For instructions, see Generating a Node Count Estimate, in the MySQL Database Service User Guide.

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

   - Using a `SET` statement before running queries:

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

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

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

   - Unloading tables that are not used. These tables consume memory on HeatWave nodes unnecessarily. See Chapter 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 Section 3.2, “Excluding Table Columns”.

7. After running queries, consider using HeatWave Advisor for encoding and data placement recommendations. See Chapter 8, *Workload Optimization using Advisor*.

Monitoring

The following monitoring practices are recommended:

- Monitor operating system memory usage. Use the console to set 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, table data becomes stale and queries that access tables with stale data are not offloaded. For instructions, see Section 4.2, “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 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.

  - 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 Chapter 4, *Loading Data*.

> **Tip**
>
> Instead of loading data into HeatWave manually, consider using the Auto Parallel Load utility, which prepares and loads tables for you using an optimized number of parallel load threads. See Section 4.1, “Auto Parallel Load”.

- When the HeatWave cluster is restarted. Data in the HeatWave cluster is lost in this case, requiring reload.

Chapter 10 Troubleshooting

- *Problem:* Queries are not offloaded.

  - *Solution A:* Your query contains an unsupported predicate, function, operator, or has encountered some other limitation. See Chapter 6, *Running Queries*.

  - *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.

    2. Run the query using `EXPLAIN`.

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

  - *Solution C:* The table you are querying is not loaded. You can check the load status of a table in HeatWave by querying `LOAD_STATUS` data from HeatWave Performance Schema tables. For example:

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

    For information about load statuses, see Section 11.10.4, “The rpd_tables Table”.

    Alternatively, run the following statement:

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

    The following error is reported if the table is already loaded:

```
ERROR 13331 (HY000): Table is already loaded.
```

- *Solution D:* The HeatWave cluster has failed. To determine the status of the HeatWave cluster, run the following statement:

```
mysql> SHOW GLOBAL STATUS LIKE 'rapid_plugin_bootstrapped';
+--------------------------------+------------+
| Variable_name                  | Value      |
+--------------------------------+------------+
| rapid_plugin_bootstrapped      | YES        |
+--------------------------------+------------+
```

  See for Section 11.7, “System Variables” for `rapid_plugin_bootstrapped` status values.

  If the HeatWave cluster has failed, restart it and reload your data. For restart instructions, refer to the MySQL Database Service User Guide.

- *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;
```

- *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 node count estimate. If your data has grown substantially, you may require additional HeatWave nodes. For node count estimate instructions, refer to the MySQL Database Service User 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 8000 bytes are rejected during `SECONDARY_LOAD` operations. Reduce the size of `TEXT`-type values to less than 8000 bytes or exclude the column before loading the table. See Section 3.2, “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 8000 bytes.”

  Solution: `TEXT`-type values larger than 8000 bytes are rejected during change propagation. Reduce the size of `TEXT`-type values to less than 8000 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 4.2, “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:

```
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.

Chapter 11 Reference

Table of Contents

11.1 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 Section 3.2, “Excluding Table Columns”.

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

  For `ENUM` limitations, see Section 11.6, “Limitations”.

11.2 Supported Functions and Operators

This section describes functions and operators supported by HeatWave.

11.2.1 Aggregate Functions

The following table shows supported aggregate functions. The *VARLEN Support* column identifies functions that support variable-length encoded string columns. See Section 3.3, “Encoding String Columns”.

Table 11.1 Aggregate (GROUP BY) Functions

Name	VARLEN Support	Description
`AVG()`		Return the average value of the argument
`COUNT()`	Yes	Return a count of the number of rows returned
`COUNT(DISTINCT)`		Return the count of a number of different values
`MAX()`	Yes	Return the maximum value
`MIN()`	Yes	Return the minimum value
`STD()`		Return the population standard deviation

Name	VARLEN Support	Description
STDDEV()		Return the population standard deviation
STDDEV_POP()		Return the population standard deviation
STDDEV_SAMP()		Return the sample standard deviation
SUM()		Return the sum
VAR_POP()		Return the population standard variance
VAR_SAMP()		Return the sample variance
VARIANCE()		Return the population standard variance

11.2.2 Arithmetic Operators

The following table shows supported arithmetic operators. Arithmetic operators are not supported with variable-length encoded string columns. See Section 3.3, “Encoding String Columns”.

Table 11.2 Arithmetic Operators

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

11.2.3 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:

```
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`.

11.2.4 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 3.3, "Encoding String Columns".

Table 11.3 Comparison Functions and Operators

Name	VARLEN Support	Description
`BETWEEN ... AND ...`	Yes	Check whether a value is within a range of values
`COALESCE()`	Yes	Return the first non-NULL argument. Not supported as a `JOIN` predicate.
`=`	Yes	Equal operator
`<=>`		NULL-safe equal to operator
`>`	Yes	Greater than operator
`>=`	Yes	Greater than or equal operator
`GREATEST()`	Yes	Return the largest argument.
`IN()`	Yes	Check whether a value is within a set of values
`IS`		Test a value against a boolean
`IS NOT`		Test a value against a boolean
`IS NOT NULL`	Yes	NOT NULL value test
`IS NULL`	Yes	NULL value test
`ISNULL()`		Test whether the argument is NULL
`LEAST()`	Yes	Return the smallest argument.
`<`	Yes	Less than operator
`<=`	Yes	Less than or equal operator
`LIKE`	Yes	Simple pattern matching
`NOT BETWEEN ... AND ...`	Yes	Check whether a value is not within a range of values
`!=`, `<>`	Yes	Not equal operator
`NOT IN()`	Yes	Check whether a value is not within a set of values
`NOT LIKE`	Yes	Negation of simple pattern matching
`STRCMP()`	Yes	Compare two strings.

11.2.5 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 3.3, "Encoding String Columns".

Table 11.4 Control Flow Functions and Operators

Name	VARLEN Support	Description
`CASE`	Yes	Case operator

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

11.2.6 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 3.3, “Encoding String Columns”.

Table 11.5 Date and Time Functions

Name	VARLEN Support	Description
ADDDATE()		Add time values (intervals) to a date value
ADDTIME()	Yes	Add time
CURDATE()		Return the current date
CURRENT_DATE(), CURRENT_DATE		Synonyms for CURDATE()
CURRENT_TIME(), CURRENT_TIME		Synonyms for CURTIME()
CURRENT_TIMESTAMP(), CURRENT_TIMESTAMP		Synonyms for NOW()
CURTIME()		Return the current time
DATE()	Yes	Extract the date part of a date or datetime expression
DATE_ADD()	Yes	Add time values (intervals) to a date value
DATE_FORMAT()	Yes	Format date as specified
DATE_SUB()		Subtract a time value (interval) from a date
DATEDIFF()		Subtract two dates
DAY()	Yes	Synonym for DAYOFMONTH()
DAYNAME()	Yes	Return the name of the weekday
DAYOFMONTH()	Yes	Return the day of the month (0-31)
DAYOFWEEK()		Return the weekday index of the argument
DAYOFYEAR()	Yes	Return the day of the year (1-366)
EXTRACT()		Extract part of a date
FROM_UNIXTIME()		Format Unix timestamp as a date
HOUR()	Yes	Extract the hour
LOCALTIME(), LOCALTIME		Synonym for NOW()
LOCALTIMESTAMP, LOCALTIMESTAMP()		Synonym for NOW()
MAKEDATE()		Create a date from the year and day of year. Supports FLOAT, DOUBLE, INTEGER, and YEAR type values.

Name	VARLEN Support	Description
MICROSECOND()	Yes	Return the microseconds from argument
MINUTE()	Yes	Return the minute from the argument
MONTH()	Yes	Return the month from the date passed
MONTHNAME()	Yes	Return the name of the month
NOW()		Return the current date and time
QUARTER()	Yes	Return the quarter from a date argument
SECOND()		Return the second (0-59)
STR_TO_DATE()	Yes	Convert a string to a date
SUBTIME()	Yes	Subtract times
TIME()	Yes	Extract the time portion of the expression passed
TIME_FORMAT()	Yes	Format as time.
TIME_TO_SEC()		Return the argument converted to seconds
TIMESTAMP()	Yes	With a single argument, this function returns the date or datetime expression; with two arguments, the sum of the arguments
TIMESTAMPDIFF()	Yes	Subtract an interval from a datetime expression
TO_DAYS()	Yes	Return the date argument converted to days
TO_SECONDS()	Yes	Return the date or datetime argument converted to seconds since Year 0
UNIX_TIMESTAMP()		Return a Unix timestamp
WEEK()	Yes	Return the week number. Restrictions apply. See Section 11.6, “Limitations”
WEEKDAY()		Return the weekday index
WEEKOFYEAR()		Return the calendar week of the date (1-53)
YEAR()	Yes	Return the year
YEARWEEK()		Return the year and week

11.2.7 Logical Operators

The following table shows supported logical operators.

Table 11.6 Logical Operators

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

11.2.8 Mathematical Functions

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

Table 11.7 Mathematical Functions

Name	Description
ABS()	Return the absolute value
ACOS()	Return the arc cosine
ASIN()	Return the arc sine
ATAN()	Return the arc tangent
CEIL()	Return the smallest integer value not less than the argument. The function is not applied to BIGINT values. The input value is returned. CEIL() is a synonym for CEILING().
CEILING()	Return the smallest integer value not less than the argument. The function is not applied to BIGINT values. The input value is returned. CEILING() is a synonym for CEIL().
COS()	Return the cosine
COT()	Return the cotangent
DEGREES()	Convert radians to degrees
EXP()	Raise to the power of
FLOOR()	Return the largest integer value not greater than the argument. The function is not applied to BIGINT values. The input value is returned.
LN()	Return the natural logarithm of the argument
LOG()	Return the natural logarithm of the first argument
LOG10()	Return the base-10 logarithm of the argument
MOD()	Return the remainder
RADIANS()	Return argument converted to radians
ROUND()	Round the argument
SIN()	Return the sine of the argument
SQRT()	Return the square root of the argument
TAN()	Return the tangent of the argument
TRUNCATE()	Truncate to specified number of decimal places

11.2.9 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 11.8 String Functions and Operators

Name	Description
ASCII()	Return numeric value of left-most character
BIT_LENGTH()	Return length of argument in bits
CHAR_LENGTH()	Return number of characters in argument
CONCAT()	Return concatenated string
CONCAT_WS()	Return concatenated with separator

Name	Description
FIND_IN_SET()	Index (position) of first argument within second argument
FORMAT()	Return a number formatted to specified number of decimal places. *Does not support variable-length-encoded columns.*
FROM_BASE64()	Decode base64 encoded string and return result
GREATEST()	Return the largest argument. Not supported with temporal columns.
HEX()	Hexadecimal representation of decimal or string value
INSERT()	Return the index of the first occurrence of substring
INSTR()	Return the index of the first occurrence of substring
LEAST()	Return the smallest argument. Not supported with temporal columns.
LEFT()	Return the leftmost number of characters as specified
LENGTH()	Return the length of a string in bytes
LIKE	Simple pattern matching
LOCATE()	Return the position of the first occurrence of substring
LOWER()	Return the argument in lowercase
LPAD()	Return the string argument, left-padded with the specified string
LTRIM()	Remove leading spaces
NOT LIKE	Negation of simple pattern matching
OCTET_LENGTH()	Synonym for LENGTH()
ORD()	Return character code for leftmost character of the argument
POSITION()	Synonym for LOCATE()
REPEAT()	Repeat a string the specified number of times
QUOTE()	Escape the argument for use in an SQL statement
REGEXP	Whether string matches regular expression
REGEXP_LIKE()	Whether string matches regular expression
REGEXP_REPLACE()	Replace substrings matching regular expression. Supports up to three arguments.
REGEXP_SUBSTR()	Return substring matching regular expression. Supports up to three arguments.
REPLACE()	Replace occurrences of a specified string
REVERSE()	Reverse the characters in a string
RIGHT()	Return the specified rightmost number of characters
RLIKE	Whether string matches regular expression
RPAD()	Append string the specified number of times
RTRIM()	Remove trailing spaces
STRCMP()	Compare two strings
SUBSTR()	Return the substring as specified

Name	Description
SUBSTRING()	Return the substring as specified
SUBSTRING_INDEX()	Return a substring from a string before the specified number of occurrences of the delimiter
TO_BASE64()	Return the argument converted to a base-64 string
TRIM()	Remove leading and trailing spaces
UNHEX()	Return a string containing hex representation of a number
UPPER()	Convert to uppercase

11.2.10 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()
  - MIN()
  - MAX()

11.3 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

11.4 String Column Encoding Reference

HeatWave supports two string column encoding types:

- Section 11.4.1, “Variable-length Encoding”
- Section 11.4.2, “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 3.3, “Encoding String Columns”.

11.4.1 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.

11.4.1.1 VARLEN Supported Expressions, Filters, Functions, and Operators

For supported functions and operators, refer to Section 11.2, “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`

11.4.1.2 VARLEN Encoding Limits

- The maximum size of VARLEN-encoded columns for base tables is 8000 bytes. For example, if using a 4-byte character set, a VARCHAR column is limited to 2000 characters (VARCHAR(2000)).

  TEXT-type values larger than 8000 bytes are rejected by table load and change propagation operations. Both operations fail with an error when encountering a TEXT-type value larger than 8000 bytes. (This limit is not enforced for VARLEN-encoded VARCHAR columns.)

- The maximum size of VARLEN-encoded columns for final and intermediate results generated by HeatWave is 16382 bytes.

- When a query includes VARLEN-encoded columns, the maximum number of columns produced by any physical operator is 128. However, the actual maximum number of columns depends on factors such as MySQL limits, protocol limits, the total number of columns, column types, and column widths (the string length of supported string-type columns). For example, for any physical operator, the maximum number of 8000-byte VARLEN-encoded columns is 31 if the query only uses VARLEN-encoded columns. The maximum number of 16382-byte VARLEN-encoded columns is 15. On the other hand, HeatWave can only produce a maximum of 128 VARLEN-encoded columns that are 1 byte in size if the query includes only VARLEN-encoded columns. If a query includes non-VARLEN-encoded columns, the column number limits are likely to be lower.

- Only expressions with non-boolean types are supported.

11.4.1.3 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.

11.4.1.4 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.

11.4.2 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.

11.5 Metadata Queries

This section provides metadata queries that you can use to retrieve information about data, queries, and HeatWave.

- 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%';
+-------------+---------------+
| COLUMN_NAME | EXTRA         |
+-------------+---------------+
| O_COMMENT   | NOT SECONDARY |
+-------------+---------------+
```

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

- To identify explicitly encoded string columns in tables on the MySQL DB System, query the `COLUMN_COMMENT` column of the `INFORMATION_SCHEMA.COLUMNS` table. For example:

```
mysql> SELECT COLUMN_NAME, COLUMN_COMMENT FROM INFORMATION_SCHEMA.COLUMNS
       WHERE TABLE_NAME LIKE 'orders' AND COLUMN_COMMENT LIKE '%ENCODING%';
+-----------------+-----------------------------+
| COLUMN_NAME     | COLUMN_COMMENT              |
+-----------------+-----------------------------+
| O_CLERK         | RAPID_COLUMN=ENCODING=SORTED |
| O_ORDERPRIORITY | RAPID_COLUMN=ENCODING=SORTED |
| O_ORDERSTATUS   | RAPID_COLUMN=ENCODING=SORTED |
| O_CLERK         | RAPID_COLUMN=ENCODING=SORTED |
| O_ORDERPRIORITY | RAPID_COLUMN=ENCODING=SORTED |
| O_ORDERSTATUS   | RAPID_COLUMN=ENCODING=SORTED |
+-----------------+-----------------------------+
```

  You can also view explicitly defined column encodings for an individual table using `SHOW CREATE TABLE`.

- To identify columns defined as data placement keys in tables on the MySQL DB System, query the `COLUMN_COMMENT` column of the `INFORMATION_SCHEMA.COLUMNS` table. For example:

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

```
| COLUMN_NAME     | COLUMN_COMMENT                    |
+-----------------+-----------------------------------+
| O_CUSTKEY       | RAPID_COLUMN=DATA_PLACEMENT_KEY=1 |
+-----------------+-----------------------------------+
```

You can also view data placement keys for an individual table using `SHOW CREATE TABLE`.

- To identify columns defined as data placement keys in tables that are loaded in HeatWave, query the `DATA_PLACEMENT_INDEX` column of the `performance_schema.rpd_columns` table for columns with a `DATA_PLACEMENT_INDEX` value greater than 0, which indicates that the column is defined as a data placement key. For example:

```
mysql> SELECT TABLE_NAME, COLUMN_NAME, DATA_PLACEMENT_INDEX
       FROM performance_schema.rpd_columns r1
       JOIN performance_schema.rpd_column_id r2 ON r1.COLUMN_ID = r2.ID
       WHERE r1.TABLE_ID = (SELECT ID FROM performance_schema.rpd_table_id
       WHERE TABLE_NAME = 'orders') AND r2.TABLE_NAME = 'orders'
       AND r1.DATA_PLACEMENT_INDEX > 0 ORDER BY r1.DATA_PLACEMENT_INDEX;
+------------+-------------+----------------------+
| TABLE_NAME | COLUMN_NAME | DATA_PLACEMENT_INDEX  |
+------------+-------------+----------------------+
| orders     | O_TOTALPRICE |                    1 |
| orders     | O_ORDERDATE  |                    2 |
| orders     | O_COMMENT    |                    3 |
+------------+-------------+----------------------+
```

For information about data placement key index values, see Section 3.4, “Defining Data Placement Keys”.

- To determine if data placement partitions were used by a `JOIN` or `GROUP BY` query, you can query the `QEP_TEXT` column of the `performance_schema.rpd_query_stats` table to view `prepart` data. (`prepart` is short for “pre-partitioning”.) The `prepart` data for a `GROUP BY` operation contains a single value; for example: `"prepart":#`, where `#` represents the number of HeatWave nodes. A value greater than 1 indicates that data placement partitions were used. The `prepart` data for a `JOIN` operation has two values that indicate the number of HeatWave nodes; one for each `JOIN` branch; for example: `"prepart":[#,#]`. A value greater than 1 for a `JOIN` branch indicates that the `JOIN` branch used data placement partitions. (A value of `"prepart":[1,1]` indicates that data placement partitions were not used by either `JOIN` branch.) `prepart` data is only generated if a `GROUP BY` or `JOIN` operation is executed. To query `QEP_TEXT` `prepart` data for the last executed query:

```
mysql> SELECT CONCAT( '"prepart":[', (JSON_EXTRACT(QEP_TEXT->>"$**.prepart", '$[0][0]')),
       "," ,(JSON_EXTRACT(QEP_TEXT->>"$**.prepart", '$[0][1]')) , ']' )
       FROM performance_schema.rpd_query_stats WHERE query_id = (select max(query_id)
       FROM performance_schema.rpd_query_stats);
+-------------------------------------------------------------------------------+
| concat( '"prepart":[', (JSON_EXTRACT(QEP_TEXT->>"$**.prepart", '$[0][0]')), |
|"," ,(JSON_EXTRACT(QEP_TEXT->>"$**.prepart", '$[0][1]')) , ']' )              |
+-------------------------------------------------------------------------------+
| "prepart":[2,2]                                                               |
+-------------------------------------------------------------------------------+
```

- 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.

```
mysql> SELECT TABLE_SCHEMA, TABLE_NAME, CREATE_OPTIONS FROM INFORMATION_SCHEMA.TABLES
       WHERE CREATE_OPTIONS LIKE '%SECONDARY_ENGINE%' AND TABLE_SCHEMA LIKE 'tpch';
+--------------+------------+--------------------------+
| TABLE_SCHEMA | TABLE_NAME | CREATE_OPTIONS           |
+--------------+------------+--------------------------+
| tpch         | customer   | SECONDARY_ENGINE="RAPID" |
| tpch         | lineitem   | SECONDARY_ENGINE="RAPID" |
```

```
| tpch         | nation     | SECONDARY_ENGINE="RAPID" |
| tpch         | orders     | SECONDARY_ENGINE="RAPID" |
| tpch         | part       | SECONDARY_ENGINE="RAPID" |
| tpch         | partsupp   | SECONDARY_ENGINE="RAPID" |
| tpch         | region     | SECONDARY_ENGINE="RAPID" |
| tpch         | supplier   | SECONDARY_ENGINE="RAPID" |
+--------------+------------+--------------------------+
```

You can also view create options for an individual table using `SHOW CREATE TABLE`.

> **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.

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

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

- To check the load status of a table in the HeatWave cluster, query the `LOAD_STATUS` data from HeatWave Performance Schema tables. For example:

```
mysql> USE performance_schema;
mysql> SELECT NAME, LOAD_STATUS FROM rpd_tables,rpd_table_id
       WHERE rpd_tables.ID = rpd_table_id.ID 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 11.10.4, “The rpd_tables Table”.

- To check whether change propagation is enabled or disabled for a particular table, query the `POOL_TYPE` data from the HeatWave Performance Schema tables. `RAPID_LOAD_POOL_TRANSACTIONAL` indicates that change propagation is enabled for the table. `RAPID_LOAD_POOL_SNAPSHOT` indicates that change propagation is disabled.

```
mysql> 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 |
+---------------+------------------------------+
```

To check the global change propagation status, query the `rapid_change_propagation_status` variable:

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

- To view the number of queries offloaded to the HeatWave cluster for execution:

```
mysql> SELECT VARIABLE_VALUE
       FROM performance_schema.global_status
       WHERE VARIABLE_NAME = 'rapid_query_offload_count';
+----------------+
| VARIABLE_VALUE |
+----------------+
| 62             |
+----------------+
```

- To view 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
```

```
    ) tmp;
```

The query returns the following data:

- `QUERY_ID`

  The ID assigned to the query by HeatWave. IDs are assigned in first in first out (FIFO) order.

- `CONNECTION_ID`

  The connection ID of the client that issued the query.

- `QUERY_START`

  The time the query was issued.

- `QUERY_END`

  The time the query finished executing.

- `QUEUE_WAIT`

  The amount of time the query waited in the scheduling queue.

- `EXEC_START`

  The time that HeatWave started executing the query.

11.6 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 11.1, “Supported Data Types”, Section 11.2, “Supported Functions and Operators”, and Section 11.3, “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 11.2.3, “Cast Functions and Operators”.
  - `COALESCE()` as a `JOIN` predicate.
  - 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 3.3, “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:

  ```
  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 11.1, “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.
- Variables:
  - `time_zone` and `timestamp` variable settings are not passed to HeatWave when queries are offloaded.
  - 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.
- 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 11.3, “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:

    ```
    SELECT AVG(c1) AS value FROM t1 WHERE c1 IS NULL;
    ```

  - Querying of `YEAR` type data using expressions and other functions. For example, the following queries are not offloaded:

    ```
    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 stored program.
- Queries that are part of a multi-query transaction.
- Views.

  `CREATE VIEW ... AS SELECT` queries. Setting `use_secondary_engine=FORCED` does not cause the statement to fail with an error. The statement is executed on the MySQL Database Service instance regardless of the `use_secondary_engine` setting.

- 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 Chapter 6, *Running Queries*.)
- `SET timezone = timezone`, with the *`timezone`* value specified as a 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.
- 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.

11.7 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_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`

	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).

- 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`
Scope	Session
Dynamic	No
`SET_VAR` Hint Applies	No
Type	Enumeration
Default Value	`MIN_RUNTIME`

Valid Values	`MIN_RUNTIME`
	
	`MIN_MEM_CONSUMPTION`

Specifies the query execution strategy to use. Minimum runtime (`MIN_RUNTIME`) or minimum memory consumption (`MIN_MEM_CONSUMPTION`).

HeatWave optimizes for network usage rather than memory. If you encounter out of memory errors when running a query, try running the query with the `MIN_MEM_CONSUMPTION` strategy by setting by setting `rapid_execution_strategy` prior to executing the query:

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

See Chapter 10, *Troubleshooting*.

- `rapid_stats_cache_max_entries`

Command-Line Format	`--rapid-stats-cache-max-entries=#`
Introduced	8.0.25
System Variable	`rapid_stats_cache_max_entries`
Scope	Global
Dynamic	Yes
`SET_VAR` Hint Applies	No
Type	Integer
Default Value	`65536`
Minimum Value	`0`

Maximum Value	`1048576`

The setting for this variable is managed by OCI and cannot be modified directly. Specifies the maximum number of entries in the statistics cache.

The number of entries permitted in the statistics cache by default is 65536, which is enough to store statistics for 4000 to 5000 unique queries of medium complexity.

For more information, see Auto Query Plan Improvement.

11.8 Secondary Engine Variables

This section describes MySQL DB System variables intended for use with HeatWave.

- `use_secondary_engine`

Introduced	8.0.13
System Variable	`use_secondary_engine`
Scope	Session
Dynamic	Yes
`SET_VAR` Hint Applies	Yes
Type	Enumeration
Default Value	`ON`

Valid Values	`OFF`
	`ON`
	`FORCED`

Whether to execute queries using the secondary engine. These values are permitted:

- `OFF`: Queries execute using the primary storage (InnoDB) on the MySQL DB System. Execution using the secondary engine (RAPID) is disabled.
- `ON`: Queries execute using the secondary engine (RAPID) when conditions warrant, falling back to the primary storage engine (InnoDB) otherwise. In the case of fallback to the primary engine, whenever that occurs during statement processing, the attempt to use the secondary engine is abandoned and execution is attempted using the primary engine.
- `FORCED`: Queries always execute using the secondary engine (RAPID) or fail if that is not possible. Under this mode, a query returns an error if it cannot be executed using the secondary engine, regardless of whether the tables that are accessed have a secondary engine defined.

- `show_create_table_skip_secondary_engine`

Command-Line Format	`--show-create-table-skip-secondary-engine[={OFF|ON}]`
Introduced	8.0.18
System Variable	`show_create_table_skip_secondary_engine`
Scope	Session
Dynamic	Yes

SET_VAR Hint Applies	Yes
Type	Boolean
Default Value	OFF

Whether to exclude the SECONDARY ENGINE clause from SHOW CREATE TABLE output, and from CREATE TABLE statements dumped by the mysqldump utility.

mysqldump provides the --show-create-skip-secondary-engine option. When specified, it enables the show_create_table_skip_secondary_engine system variable for the duration of the dump operation.

Attempting 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.

11.9 Status Variables

Several status variables provide operational information about HeatWave. You can retrieve status data using SHOW STATUS syntax. For example:

```
mysql> SHOW STATUS LIKE 'rapid%';
+---------------------------------+------------+
| Variable_name                   | Value      |
+---------------------------------+------------+
| hw_data_scanned                 | 0          |
| rapid_change_propagation_status | ON         |
| rapid_cluster_status            | ON         |
| rapid_core_count                | 64         |
| rapid_heap_usage                | 58720397   |
| rapid_load_progress             | 100.000000 |
| rapid_plugin_bootstrapped       | YES        |
| rapid_preload_stats_status      | Available  |
| rapid_query_offload_count       | 46         |
| rapid_service_status            | ONLINE     |
+---------------------------------+------------+
```

- hw_data_scanned

  Tracks the amount of data scanned by successfully executed HeatWave queries. Data is tracked in megabytes and is a cumulative total of data scanned since the HeatWave cluster was last started. The counter is reset to 0 when the HeatWave cluster is restarted (when the rapid_bootstrap state changes from OFF or IDLE to ON.)

- rapid_change_propagation_status

  The change propagation status.

  A status of ON indicates that change propagation is enabled globally, permitting changes to InnoDB tables on the MySQL DB System to be propagated to their counterpart tables in the HeatWave cluster.

- rapid_cluster_status

  The HeatWave cluster status.

- rapid_core_count

  The HeatWave node core count. The value remains at 0 until all HeatWave nodes are started.

- rapid_heap_usage

MySQL DB System node heap usage.

- `rapid_load_progress`

  A percentage value indicating the status of a table load operation.

- `rapid_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 when generating a node count estimate. You can generate a node count estimate when adding or modifying a HeatWave cluster. States include `Not started`, `In progress`, and `Statistics collected`.

- `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.

11.10 Performance Schema Tables

HeatWave Performance Schema tables provide information about HeatWave nodes, and about tables and columns that are currently loaded in HeatWave.

Information about HeatWave nodes is available only when `rapid_bootstrap` mode is `ON`. Information about tables and columns is available only after tables are loaded in the HeatWave cluster. See Chapter 4, *Loading Data*.

11.10.1 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 200 rows per node. Data is stored only for successfully executed 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.

11.10.2 The rpd_nodes Table

The `rpd_nodes` table provides information about HeatWave nodes.

The `rpd_nodes` table has these columns:

- `ID`

  A unique identifier for the HeatWave node.

- `CORES`

  The number of cores used by the HeatWave node.

- `MEMORY_TOTAL` (renamed from `DRAM` in MySQL 8.0.24)

  The total memory in bytes allocated to the HeatWave 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.

- `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. Introduced in MySQL 8.0.24.

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.

11.10.3 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.

- `ROWS`

  The total number of rows initially loaded. The reported value is not updated as changes are propagated to HeatWave. Introduced in MySQL 8.0.24.

The `rpd_table_id` table is read-only.

11.10.4 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.

- `POOL_TYPE`

  The load pool type of the table. Possible values are `RAPID_LOAD_POOL_SNAPSHOT` and `RAPID_LOAD_POOL_TRANSACTIONAL`.

- `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.

The `rpd_tables` table is read-only.

11.10.5 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 full column name including the schema name and table name.

- `SCHEMA_NAME`

  The schema name.

- `TABLE_NAME`

  The table name.

- `COLUMN_NAME`

The column name.

- `NDV`

  Number of distinct values in the column as initially loaded. The reported value is not updated as changes are propagated to HeatWave. Introduced in MySQL 8.0.24.

The `rpd_column_id` table is read-only.

11.10.6 The rpd_columns Table

The `rpd_columns` table provides column encoding information for columns of tables loaded in HeatWave.

The `rpd_columns` table has these columns:

- `TABLE_ID`

  A unique identifier for the table.

- `COLUMN_ID`

  A unique identifier for the table column.

- `ENCODING`

  The type of encoding used. Possible values include `VARLEN` and `SORTED`.

- `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 3.4, “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 11.5, “Metadata Queries”.

The `rpd_columns` table is read-only.

11.10.7 The rpd_query_stats Table

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

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 200 executed queries. Data is stored only for successfully executed queries.

The `rpd_query_stats` table has these columns:

- `QUERY_ID`

  The query ID.

- `QUERY_TEXT`

  The query.

- `QEXEC_TEXT`

  Query execution plan.

- `QKRN_TEXT`

  Logical query execution plan.

- `QEP_TEXT`

  Query execution log.

  Includes `prepart` data, which can be queried to determine if a `JOIN` or `GROUP BY` query used data placement partitions. See Section 11.5, “Metadata Queries”.

11.11 Generating tpch Sample Data

Examples in this guide and in the HeatWave 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 HeatWave Quickstart describes how to create the `tpch` schema and tables and load `tpch` sample data. The following instructions describe how to generate `tpch` sample data using the `dbgen` utility. The instructions assume you are on a Linux system that has `gcc` and `make` libraries installed.

To generate `tpch` sample data:

1. Download the TPC-H tools zip file from TPC Download Current.
2. Extract the zip file to a location on your system.
3. Change to the `dbgen` directory and make a copy of the makefile template.

   ```
   $ cd 2.18.0/dbgen
   $ cp makefile.suite makefile
   ```

4. Configure the following settings in the makefile:

   ```
   CC = gcc
   DATABASE= ORACLE
   MACHINE = LINUX
   WORKLOAD = TPCH
   ```

5. Run `make` to build the `dbgen` utility:

   ```
   $ make
   ```

6. Issue the following `dbgen` command to generate a 1GB set of data files for the `tpch` database:

   ```
   $ ./dbgen -s 1
   ```

   The operation may take a few minutes. When finished, the following data files appear in the working directory, one for each table in the `tpch` database:

   ```
   $ ls -1 *.tbl
   customer.tbl
   lineitem.tbl
   nation.tbl
   orders.tbl
   partsupp.tbl
   ```

```
part.tbl
region.tbl
supplier.tbl
```