HeatWave User Guide
Abstract

This is the HeatWave User Guide. This document provides information and procedures about loading data into a HeatWave cluster 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: 2021-03-18 (revision: 68967)
# Table of Contents

Preface and Legal Notices ........................................................................................................... v  
1 Introduction ................................................................................................................................. 1  
2 Before You Begin ......................................................................................................................... 3  
3 Preparing Data ............................................................................................................................ 5  
   3.1 Identifying Tables to Load .................................................................................................... 5  
   3.2 Excluding Table Columns ................................................................................................. 5  
   3.3 Encoding String Columns ................................................................................................. 6  
   3.4 Defining Data Placement Keys ......................................................................................... 7  
   3.5 Defining the Secondary Engine ....................................................................................... 8  
4 Loading Data ............................................................................................................................... 9  
5 Unloading Tables ......................................................................................................................... 13  
6 Running Queries ........................................................................................................................ 15  
7 Table Load and Query Example ............................................................................................... 19  
8 Best Practices ............................................................................................................................. 21  
9 Troubleshooting ......................................................................................................................... 29  
10 Reference .................................................................................................................................. 31  
   10.1 Supported Data Types ...................................................................................................... 31  
   10.2 Supported Functions and Operators ............................................................................... 32  
      10.2.1 Aggregate Functions ................................................................................................. 32  
      10.2.2 Arithmetic Operators .............................................................................................. 33  
      10.2.3 Cast Functions and Operators ............................................................................... 33  
      10.2.4 Comparison Functions and Operators .................................................................. 34  
      10.2.5 Control Flow Functions and Operators ............................................................... 34  
      10.2.6 Date and Time Functions ....................................................................................... 35  
      10.2.7 Logical Operators .................................................................................................. 36  
      10.2.8 Mathematical Functions ....................................................................................... 36  
      10.2.9 String Functions and Operators .......................................................................... 37  
   10.3 Supported SQL Modes ...................................................................................................... 38  
   10.4 String Column Encoding Reference ............................................................................... 39  
      10.4.1 Variable-length Encoding ....................................................................................... 39  
      10.4.2 Dictionary Encoding ............................................................................................... 41  
   10.5 Metadata Queries .............................................................................................................. 41  
   10.6 Limitations ......................................................................................................................... 44  
   10.7 System Variables .............................................................................................................. 48  
   10.8 Secondary Engine Variables ............................................................................................ 50  
   10.9 Status Variables ............................................................................................................... 51  
   10.10 Performance Schema Tables ......................................................................................... 52  
      10.10.1 The rpd_exec_stats Table .................................................................................. 52  
      10.10.2 The rpd_nodes Table .......................................................................................... 53  
      10.10.3 The rpd_table_id Table ..................................................................................... 54  
      10.10.4 The rpd_tables Table .......................................................................................... 54  
      10.10.5 The rpd_column_id Table ................................................................................. 54  
      10.10.6 The rpd_columns Table ....................................................................................... 56  
      10.10.7 The rpd_query_stats Table ............................................................................... 56  
   10.11 Generating tpch Sample Data .......................................................................................... 57
Preface and Legal Notices

This is the HeatWave User Guide. This document provides information and procedures about loading data into a HeatWave cluster 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

Copyright © 1997, 2021, Oracle and/or its affiliates.

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

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

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

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

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

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

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

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

This documentation is NOT distributed under a GPL license. Use of this documentation is subject to the following terms:

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

Documentation Accessibility

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

Access to Oracle Support for Accessibility

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

This guide is intended for users of HeatWave. HeatWave is a distributed, scalable, shared-nothing, in-memory, columnar, query processing engine designed for fast execution of analytic queries. It is enabled when you add a HeatWave cluster to a MySQL DB System.

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

The number of HeatWave nodes required depends on the size of your data and the amount of compression that is achieved when loading the data into the HeatWave cluster. A HeatWave cluster supports up to 24 nodes.

Figure 1.1 HeatWave Architecture

Queries are issued from a MySQL client or application that interacts with the HeatWave cluster by connecting to the MySQL DB System node. Clients and applications do not connect to the HeatWave cluster directly. Queries that meet certain prerequisites are automatically offloaded from the MySQL DB System to the HeatWave cluster 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 a HeatWave cluster requires preparing tables on the MySQL DB System and executing table load operations. Preparing tables involves modifying table definitions to exclude certain columns, define string column encodings, add data placement keys, and specify HeatWave (RAPID) as the secondary engine for the table. (InnoDB is the primary engine.) Loading a table into a HeatWave cluster requires executing an ALTER TABLE operation with the SECONDARY_LOAD keyword. For more information, see Chapter 3, Preparing Data, and Chapter 4, Loading Data.
When a table is loaded, data is sliced horizontally and distributed among the HeatWave nodes. After a table is loaded, changes to a table’s data on the MySQL DB System node are automatically propagated to the HeatWave nodes. No user action is required to keep data synchronized. For more information, see Change Propagation.
Chapter 2 Before You Begin

Before you begin using HeatWave, the following is assumed:

• You have an operational MySQL Database Service and are able to connect to it using MySQL Shell or 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-analytics` 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

3.1 Identifying Tables to Load .................................................................................................................. 5
3.2 Excluding Table Columns .................................................................................................................. 5
3.3 Encoding String Columns .................................................................................................................. 6
3.4 Defining Data Placement Keys ......................................................................................................... 7
3.5 Defining the Secondary Engine ........................................................................................................ 8

This section describes how to prepare data for loading into a HeatWave cluster. Data is prepared on the
MySQL DB System before it is loaded into the HeatWave cluster. 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”.


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 8, Best Practices.

3.1 Identifying Tables to Load

The tables accessed by the queries you intend to run must be loaded into the HeatWave cluster. If a query
accesses a table that is not loaded, the query is not offloaded to the HeatWave cluster 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, load the tables belonging to that schema.

3.2 Excluding Table Columns

Before loading a table into the HeatWave cluster, 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 10.1, “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 the HeatWave cluster when executing a table load operation.
mysql> CREATE TABLE orders (id INT, desc BLOB NOT SECONDARY);
mysql> ALTER TABLE orders MODIFY desc BLOB NOT SECONDARY;

By default, queries that access columns defined with the `NOT SECONDARY` attribute are executed on the MySQL DB system.

### 3.3 Encoding String Columns

Encoding string columns helps accelerate 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 the HeatWave cluster, variable-length encoding is applied to `CHAR`, `VARCHAR`, and `TEXT`-type columns by default. No action is required. However, if you want to use dictionary encoding, you must define the encoding type explicitly for individual string columns. See [Defining Dictionary Encoding](#).

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

Variable-length encoding requires a small percentage of additional space for column values on the HeatWave nodes. Dictionary encoding requires space on the MySQL node for dictionaries.

The following table provides an overview of encoding type characteristics:

<table>
<thead>
<tr>
<th>Encoding Type</th>
<th>Expression, Filter, Function, and Operator Support</th>
<th>Best Suited To</th>
<th>Space Required On</th>
</tr>
</thead>
<tbody>
<tr>
<td>VARLEN</td>
<td>Supports <code>JOIN</code> operations, string functions and operators, and <code>LIKE</code> predicates. See Section 10.4.1, “Variable-length Encoding”.</td>
<td>Columns with a high number of distinct values</td>
<td>HeatWave nodes</td>
</tr>
<tr>
<td>SORTED</td>
<td>Does not support <code>JOIN</code> operations, string functions and operators, or <code>LIKE</code> predicates.</td>
<td>Columns with a low number of distinct values</td>
<td>MySQL node</td>
</tr>
</tbody>
</table>

For additional information about string column encoding, see Section 10.4, “String Column Encoding Reference”.  

6
Defining Dictionary Encoding

To apply dictionary encoding to a string column, define the `RAPID_COLUMN=ENCODING=SORTED` keyword string in a column comment before loading the table. The keyword string must be uppercase; otherwise, it is ignored.

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

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

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

**Note**

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

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:

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

3.4 Defining Data Placement Keys

By default, when table data is loaded into a HeatWave cluster, it is sliced horizontally and partitioned by the table’s primary key for distribution among the HeatWave nodes. To optimize `JOIN` and `GROUP BY` query performance, you can partition the data by `JOIN` and `GROUP BY` keys instead of the primary key. Partitioning data in this way can improve `JOIN` and `GROUP BY` query performance by avoiding costs associated with redistributing data among the HeatWave nodes at query execution time.

Generally, data placement keys should defined only if partitioning data by the primary key (the default) does not provide suitable performance.

Data placement keys should also be reserved 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.

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

```sql
RAPID_COLUMN=DATA_PLACEMENT_KEY=N
```

where `N` is an index value that defines the 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 data placement key columns with index values of 1 and 2.

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

```sql
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';
```
Defining the Secondary Engine

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.

```sql
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 for a column description to be specified alongside a data placement keyword string:

```sql
COMMENT 'column_description RAPID_COLUMN=DATA_PLACEMENT_KEY=1'
```

Usage notes:

- The **JOIN** and **GROUP BY** query optimization is only applied if at least one of the **JOIN** or **GROUP BY** relations has a key that matches the 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.

- **VARLEN**-encoded columns are not considered for **JOIN** or **GROUP BY** query optimization and are therefore not recommended for use as data placement keys. String columns are encoded as variable-length columns by default. See Section 10.4.1, “Variable-length Encoding”.

- A data placement key cannot be defined on a dictionary-encoded string column.

- A data placement key can only be defined on a column with a supported data type. See Section 10.1, “Supported Data Types”.

- A data placement key column cannot be defined as **NOT SECONDARY**.

### 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 a HeatWave cluster, you must define HeatWave (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:

```sql
mysql> CREATE TABLE orders (id INT) SECONDARY_ENGINE = RAPID;
mysql> ALTER TABLE orders SECONDARY_ENGINE = RAPID;
```
Chapter 4 Loading Data

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

- Load the data into the MySQL DB System. Data is loaded into the HeatWave cluster from the MySQL DB System. For information about importing data into a MySQL DB System, refer to the MySQL Database Service User Guide.

- Define a primary key on the table.

- Exclude columns with unsupported data types. See Section 3.2, “Excluding Table Columns”.

- Define 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 8, Best Practices.

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

```sql
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.

```sql
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:

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

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

The AVAIL_RPDGSTABSTATE status indicates that the table is loaded. For more information about load statuses, see Section 10.10.4, “The rpd_tables Table”.

9
Change Propagation

When loading a table into the HeatWave cluster, 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. Data is distributed among HeatWave nodes by slicing tables horizontally. Data is partitioned by 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 the HeatWave cluster. For more information, see 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 into the HeatWave cluster. 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 9, Troubleshooting.

Change Propagation

After tables are loaded into the HeatWave cluster, 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 the HeatWave cluster in a batch transaction. Change propagation is triggered 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 the HeatWave cluster to become stale. Queries that access stale table data are not offloaded to the HeatWave cluster 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.

```sql
mysql> SELECT NAME, POOL_TYPE FROM rpd_tables, rpd_table_id
WHERE rpd_tables.ID = rpd_table_id.ID AND SCHEMA_NAME LIKE 'tpch';
```

<table>
<thead>
<tr>
<th>NAME</th>
<th>POOL_TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>tpch.orders</td>
<td>RAPID_LOAD_POOL_TRANSACTIONAL</td>
</tr>
<tr>
<td>tpch.region</td>
<td>RAPID_LOAD_POOL_TRANSACTIONAL</td>
</tr>
<tr>
<td>tpch.lineitem</td>
<td>RAPID_LOAD_POOL_TRANSACTIONAL</td>
</tr>
<tr>
<td>tpch.supplier</td>
<td>RAPID_LOAD_POOL_TRANSACTIONAL</td>
</tr>
<tr>
<td>tpch.partsupp</td>
<td>RAPID_LOAD_POOL_TRANSACTIONAL</td>
</tr>
<tr>
<td>tpch.part</td>
<td>RAPID_LOAD_POOL_TRANSACTIONAL</td>
</tr>
<tr>
<td>tpch.customer</td>
<td>RAPID_LOAD_POOL_TRANSACTIONAL</td>
</tr>
</tbody>
</table>

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 a HeatWave cluster 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 the HeatWave cluster, specify the `SECONDARY_UNLOAD` clause in an `ALTER TABLE` statement:

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

Data is removed from the HeatWave cluster only. The table contents on the MySQL DB System are not affected.
Chapter 6 Running Queries

When a HeatWave cluster is enabled and the data you want to query is loaded, queries that qualify are automatically offloaded from the MySQL DB System to the attached HeatWave cluster 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 the HeatWave cluster directly.) For information about connecting to a MySQL DB System, refer to the MySQL Database Service User Guide. After the HeatWave cluster processes a query, results are sent back to the MySQL DB System and to the client or application that issued the query.

For related best practices, see Chapter 8, Best Practices.

Query Offload Prerequisites

The following prerequisites apply for offloading queries:

• The query must be a SELECT statement.

• 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 the HeatWave cluster. 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:

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

• Queries must only use supported functions and operators. See Section 10.2, “Supported Functions and Operators”.

• Queries must avoid known limitations. See Section 10.6, “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 the HeatWave cluster, 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.

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

```
*************************** 1. row ***************************
id: 1
select_type: SIMPLE
table: orders
```
After using `EXPLAIN` to verify that the query can be offloaded, run the query and note the execution time.

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

```
+-----------------+-------------+
| O_ORDERPRIORITY | ORDER_COUNT |
|-----------------+-------------|
| 1-URGENT        | 2017573     |
| 2-HIGH          | 2015859     |
| 3-MEDIUM        | 2013174     |
| 4-NOT SPECIFIED | 2014476     |
| 5-LOW           | 2013674     |
```

```
5 rows in set (0.04 sec)
```

To compare the HeatWave cluster query execution time with MySQL DB System execution time, you can 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 a reason, refer to Chapter 9, Troubleshooting, or try debugging the query using the procedure described in Debugging Queries.

**Note**

A HeatWave cluster processes one query at a time. Concurrently issued queries are prioritized for execution. For information about query prioritization, see Query Scheduling.

**Query Scheduling**

HeatWave uses a priority-based scheduling mechanism based on query cost estimates to schedule concurrently issued queries for execution. The scheduling mechanism prioritizes short running queries over long running queries but also takes query age into account so that long running queries are eventually scheduled for execution. This query scheduling approach reduces overall query execution wait times.
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.

A query is prioritized for scheduling according to its estimated cost. A light-weight cost estimate is performed for each query at query compilation time. Based on estimated cost, a query is assigned a priority level, with the shortest queries given the highest priority and scheduled first.

A dynamic aging technique is used to ensure that long-running queries are not queued indefinitely. When a query finishes executing, the scheduler increases the priority level of queued long-running queries. In this way, long running queries eventually reach the highest priority level and are scheduled for execution.

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

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 the HeatWave cluster for processing.

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

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

```sql
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:

```sql
mysql> EXPLAIN SELECT YEARWEEK(O_ORDERDATE) FROM orders
```

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

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

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

```
+-----------------+-----------------------------------------------------------------
| QUERY           | ["Reason": "Function yearweek is not yet supported"]            |
```

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:

```json
{"Reason": "Currently unsupported RAPID query compilation scenario"}
```
For a query that does not meet the query cost threshold for secondary engine execution, this reason is reported:

```json
{"Reason": "The estimated query cost does not exceed secondary_engine_cost_threshold."}
```

For information about the query cost threshold, see Chapter 9, Troubleshooting.
Chapter 7 Table Load and Query Example

The following example demonstrates loading a table into a HeatWave cluster and executing a query. It is assumed that a HeatWave cluster 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 cluster query execution times.

```sql
 mysql> USE tpch;
 mysql> SHOW CREATE TABLE orders;
 Table: orders
 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

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

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

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

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

 mysql> ALTER TABLE orders SECONDARY_ENGINE RAPID;

 mysql> SHOW CREATE TABLE orders;
 Table: orders
 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
```
# Load the table into the HeatWave cluster

```sql
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.

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

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

# Execute the query and note the execution time

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

```
+-----------------+-------------+
| 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 cluster 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

```sql
mysql> SET SESSION use_secondary_engine=OFF;
```

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

```
+-----------------+-------------+
| 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 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
- Running Queries
- Monitoring
- Reloading Data

Provisioning

To determine the appropriate HeatWave cluster size for a workload, generate a node count estimate using the Auto-Provisioning feature in the console. The Auto-Provisioning feature uses Machine Learning techniques based on the node shape that you select and the data present in the MySQL DB System. 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 a HeatWave cluster:

- To minimize the number of HeatWave nodes required for your data, exclude table columns that are not accessed by your analytics 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 a column with a limited number of distinct values, such as a “country” column.
  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 the HeatWave cluster.

- 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.
Loading Data

Loading of data into a HeatWave cluster can be classified into three types: Initial Bulk Load, Incremental Bulk Load, and Change Propagation.

• **Initial Bulk Load**: Performed when loading data into a HeatWave cluster 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 the HeatWave cluster. An incremental bulk load involves these steps:

  1. Performing a `SECONDARY_UNLOAD` operation to unload a table from the HeatWave cluster. 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 the HeatWave cluster. See Chapter 4, Loading Data.

Depending on the amount of data, an incremental bulk load may be more expedient method of loading new data than waiting for change propagation to occur. It also provides more 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 the HeatWave cluster, data changes are automatically propagated from InnoDB tables on the MySQL DB System to their counterpart tables in the HeatWave cluster. See Change Propagation.

Use the following strategies to improve load performance:

• **Increase the number of read threads**

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

  ```sql
  mysql> SET SESSION innodb_parallel_read_thread = 32;
  
  If the MySQL DB System is not busy, you can increase the value to 64.
  
• **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**
Running Queries

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.

Running Queries

The following practices are recommended when running analytics 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, try increasing the optimizer trace buffer size. The `MISSING_BYTES_BEYOND_MAX_MEM_SIZE` column of the `INFORMATION_SCHEMA.OPTIMIZER_TRACE` table shows how many bytes are missing from a trace. If the column shows a non-zero value, increase the `optimizer_trace_max_mem_size` setting accordingly. For example:

```
SET optimizer_trace_max_mem_size=1000000;
```

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

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

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

```
mysql> EXPLAIN SELECT COUNT(*) FROM orders o, (SELECT o_custkey, AVG(o_totalprice) a_totalprice FROM orders WHERE o_custkey=o.o_custkey)a WHERE o.o_custkey=a.o_custkey AND o.o_totalprice>a.a_totalprice;
```
• By default, SELECT queries are offloaded to the HeatWave cluster for execution and fall back to the MySQL DB system if that is not possible. To force a query to execute on the HeatWave cluster 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:

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

- Using a SET_VAR optimizer hint when issuing a query:

  ```sql
  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.

     ```sql
     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.

     ```sql
     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 bad 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:

     ```sql
     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).

     ```sql
     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 = n_nationkey AND c_nationkey = s_nationkey
           AND c_nationkey = n_nationkey AND c_custkey < 3000000
           LIMIT 10;
     *************************** 1. row ***************************
     id: 1
     select_type: SIMPLE
     table: supplier
     partitions: NULL
     type: ALL
     possible_keys: NULL
     key: NULL
     key_len: NULL
     ref: NULL
     rows: 99626
     ```
To force a join order so that there are predicates associated with each pair of tables, add a `STRAIGHT_JOIN` hint. For example:

```sql
mysql> EXPLAIN SELECT o_orderkey, c_custkey, l_shipdate, s_nationkey, s_suppkey, l_comment
FROM SUPPLIER STRAIGHT_JOIN CUSTOMER STRAIGHT_JOIN NATION STRAIGHT_JOIN ORDERS
STRAIGHT_JOIN LINEITEM WHERE c_custkey = o_custkey AND o_orderkey = l_orderkey
AND c_nationkey = s_nationkey AND c_nationkey = n_nationkey AND c_custkey < 3000000 LIMIT 10\G
```

<table>
<thead>
<tr>
<th>id</th>
<th>select_type</th>
<th>table</th>
<th>partitions</th>
<th>type</th>
<th>possible_keys</th>
<th>key</th>
<th>key_len</th>
<th>ref</th>
<th>rows</th>
<th>filtered</th>
<th>Extra</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>SIMPLE</td>
<td>nation</td>
<td>NULL</td>
<td>ALL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>25</td>
<td>10.00</td>
<td>Using where; Using join buffer (hash join); Using secondary engine RAPID</td>
</tr>
<tr>
<td>1</td>
<td>SIMPLE</td>
<td>customer</td>
<td>NULL</td>
<td>ALL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>1382274</td>
<td>5.00</td>
<td>Using where; Using join buffer (hash join); Using secondary engine RAPID</td>
</tr>
<tr>
<td>1</td>
<td>SIMPLE</td>
<td>orders</td>
<td>NULL</td>
<td>ALL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>14862970</td>
<td>10.00</td>
<td>Using where; Using join buffer (hash join); Using secondary engine RAPID</td>
</tr>
<tr>
<td>1</td>
<td>SIMPLE</td>
<td>lineitem</td>
<td>NULL</td>
<td>ALL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>56834662</td>
<td>10.00</td>
<td>Using where; Using join buffer (hash join); Using secondary engine RAPID</td>
</tr>
<tr>
<td>1</td>
<td>SIMPLE</td>
<td>supplier</td>
<td>NULL</td>
<td>ALL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>26</td>
<td>100.00</td>
<td>Using secondary engine RAPID</td>
</tr>
</tbody>
</table>

To force a join order so that there are predicates associated with each pair of tables, add a `STRAIGHT_JOIN` hint. For example:
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 \texttt{GROUP BY} clause, or a \texttt{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 order of selectiveness.

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

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

   - Using a `SET` statement before running queries:
   ```sql
   mysql> SET SESSION rapid_execution_strategy = MIN_MEM_CONSUMPTION;
   ```
   - Using a `SET_VAR` optimizer hint when issuing a query:
   ```sql
   mysql> SELECT /*+ SET_VAR(rapid_execution_strategy = MIN_MEM_CONSUMPTION) */ ... FROM ...
   ```
   - Unloading tables that are not used. These tables consume memory on the HeatWave nodes unnecessarily. See Chapter 5, Unloading Tables.
   - Excluding table columns that are not accessed by your queries. These columns consume memory on the HeatWave nodes unnecessarily. This strategy requires reloading data. See Section 3.2, "Excluding Table Columns".

**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 Change Propagation.

**Reloading Data**

Reloading data is recommend 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 the HeatWave cluster. Try reloading tables three hours after the maintenance window. If the table load operation fails, try again at six hours after the maintenance window. For table load instructions, see Chapter 4, Loading Data.

- When the HeatWave cluster is restarted. Data in the HeatWave cluster is lost in this case, requiring reload.
Chapter 9 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 the HeatWave cluster, 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:

  ```sql
  mysql> USE performance_schema;
  mysql> SELECT NAME, LOAD_STATUS FROM rpd_tables,rpd_table_id
  WHERE rpd_tables.ID = rpd_table_id.ID;
  +------------------------------+---------------------+
  | NAME                         | LOAD_STATUS         |
  +------------------------------+---------------------+
  | tpch.supplier                | AVAIL_RPDGSTABSTATE |
  | tpch.partsupp                | AVAIL_RPDGSTABSTATE |
  | tpch.orders                  | AVAIL_RPDGSTABSTATE |
  | tpch.lineitem                | AVAIL_RPDGSTABSTATE |
  | tpch.customer                | AVAIL_RPDGSTABSTATE |
  | tpch.nation                  | AVAIL_RPDGSTABSTATE |
  | tpch.region                  | AVAIL_RPDGSTABSTATE |
  | tpch.part                    | AVAIL_RPDGSTABSTATE |
  +------------------------------+---------------------+
  
  For information about load statuses, see Section 10.10.4, “The rpd_tables Table”.

  Alternatively, run the following statement:

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

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

  ```sql
  ERROR 13331 (HY000): Table is already loaded.
  ```
• **Solution D:** The HeatWave cluster has failed. To determine the status of the HeatWave cluster, run the following statement:

```sql
mysql> SHOW GLOBAL STATUS LIKE 'rapid_plugin_bootstrapped';
<table>
<thead>
<tr>
<th>Variable_name</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>rapid_plugin_bootstrapped</td>
<td>YES</td>
</tr>
</tbody>
</table>
```

See for Section 10.7, “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:

```sql
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:

```sql
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.

• **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 Change Propagation.
Chapter 10 Reference

Table of Contents

10.1 Supported Data Types .......................................................... 31
10.2 Supported Functions and Operators ........................................... 32
  10.2.1 Aggregate Functions ...................................................... 32
  10.2.2 Arithmetic Operators .................................................. 33
  10.2.3 Cast Functions and Operators ...................................... 33
  10.2.4 Comparison Functions and Operators ............................ 34
  10.2.5 Control Flow Functions and Operators ......................... 34
  10.2.6 Date and Time Functions ........................................... 35
  10.2.7 Logical Operators .................................................. 36
  10.2.8 Mathematical Functions ............................................ 36
  10.2.9 String Functions and Operators .................................. 37
10.3 Supported SQL Modes .......................................................... 38
10.4 String Column Encoding Reference ....................................... 39
  10.4.1 Variable-length Encoding ........................................... 39
  10.4.2 Dictionary Encoding ................................................ 41
10.5 Metadata Queries ............................................................... 41
10.6 Limitations .......................................................................... 44
10.7 System Variables .................................................................. 48
10.8 Secondary Engine Variables .................................................. 50
10.9 Status Variables ................................................................... 51
10.10 Performance Schema Tables .................................................. 52
  10.10.1 The rpd_exec_stats Table ........................................... 52
  10.10.2 The rpd_nodes Table .................................................. 53
  10.10.3 The rpd_table_id Table .............................................. 54
  10.10.4 The rpd_tables Table ................................................ 54
  10.10.5 The rpd_column_id Table ........................................... 55
  10.10.6 The rpd_columns Table ............................................... 56
  10.10.7 The rpd_query_stats Table ....................................... 56
10.11 Generating tpch Sample Data ............................................... 57

10.1 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
Supported Functions and Operators

- 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 10.6, “Limitations”.

10.2 Supported Functions and Operators

This section describes functions and operators supported by the HeatWave.

10.2.1 Aggregate Functions

The following table shows supported aggregate functions.

**Table 10.1 Aggregate (GROUP BY) Functions**

<table>
<thead>
<tr>
<th>Name</th>
<th>VARLEN Support</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AVG()</td>
<td></td>
<td>Return the average value of the argument</td>
</tr>
<tr>
<td>COUNT()</td>
<td>Yes</td>
<td>Return a count of the number of rows returned</td>
</tr>
<tr>
<td>COUNT(DISTINCT)</td>
<td>Yes</td>
<td>Return the count of a number of different values</td>
</tr>
<tr>
<td>MAX()</td>
<td>Yes</td>
<td>Return the maximum value</td>
</tr>
<tr>
<td>MIN()</td>
<td>Yes</td>
<td>Return the minimum value</td>
</tr>
<tr>
<td>STD()</td>
<td></td>
<td>Return the population standard deviation</td>
</tr>
</tbody>
</table>
### Arithmetic Operators

The following table shows supported arithmetic operations. Arithmetic operators are not supported with **VARLEN**-encoded columns.

#### 10.2.2 Arithmetic Operators

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

#### 10.2.3 Cast Functions and Operators

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

- **CAST()** to **DECIMAL**.

- **CAST()** of **VARLEN** **DATE, DATETIME, and TIME** column values to **DOUBLE**.

- **CAST()** of **VARLEN** **DECIMAL** and **DOUBLE** column values to temporal types such as **DATE, DATETIME, and TIME**.

- **CAST()** of **TIME, DATETIME, TIMESTAMP, and DATE** values to **REAL, TIME, DATETIME, and DATE**.

- **CAST()** of values from **DATETIME, TIMESTAMP, DATE, and TIME** types to **DOUBLE**.

- Casting of temporal types to **VARCHAR**.

- **CAST()** of **DECIMAL** and **INTEGER** types to the **DECIMAL**. For example:

  ```sql
  CAST(c1 AS DECIMAL(5,2))
  ```

- **CAST()** of **INTEGER** values to **SIGNED** and **UNSIGNED**.

- **CAST()** of **ENUM** values to **CHAR**, and to **SIGNED** and **UNSIGNED** numeric values.
10.2.4 Comparison Functions and Operators

The following table shows supported comparison functions and operators.

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

10.2.5 Control Flow Functions and Operators

The following table shows supported control flow operators.

<table>
<thead>
<tr>
<th>Name</th>
<th>VARLEN Support</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CASE</td>
<td>Yes</td>
<td>Case operator</td>
</tr>
<tr>
<td>IF()</td>
<td>Yes</td>
<td>If/else construct</td>
</tr>
<tr>
<td>IFNULL()</td>
<td>Yes</td>
<td>Null if/else construct</td>
</tr>
<tr>
<td>NULLIF()</td>
<td>Yes</td>
<td>Return NULL if expr1 = expr2</td>
</tr>
</tbody>
</table>
10.2.6 Date and Time Functions

The following table shows supported date and time functions.

Table 10.5 Date and Time Functions

<table>
<thead>
<tr>
<th>Name</th>
<th>VARLEN Support</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDDATE()</td>
<td></td>
<td>Add time values (intervals) to a date value</td>
</tr>
<tr>
<td>ADDTIME()</td>
<td>Yes</td>
<td>Add time</td>
</tr>
<tr>
<td>CURDATE()</td>
<td></td>
<td>Return the current date</td>
</tr>
<tr>
<td>CURRENT_DATE(), CURRENT_DATE</td>
<td></td>
<td>Synonyms for CURDATE()</td>
</tr>
<tr>
<td>CURRENT_TIME(), CURRENT_TIME</td>
<td></td>
<td>Synonyms for CURTIME()</td>
</tr>
<tr>
<td>CURRENT_TIMESTAMP(), CURRENT_TIMESTAMP</td>
<td></td>
<td>Synonyms for NOW()</td>
</tr>
<tr>
<td>CURTIME()</td>
<td></td>
<td>Return the current time</td>
</tr>
<tr>
<td>DATE()</td>
<td>Yes</td>
<td>Extract the date part of a date or datetime expression</td>
</tr>
<tr>
<td>DATE_ADD()</td>
<td>Yes</td>
<td>Add time values (intervals) to a date value</td>
</tr>
<tr>
<td>DATE_FORMAT()</td>
<td>Yes</td>
<td>Format date as specified</td>
</tr>
<tr>
<td>DATE_SUB()</td>
<td></td>
<td>Subtract a time value (interval) from a date</td>
</tr>
<tr>
<td>DATEDIFF()</td>
<td></td>
<td>Subtract two dates</td>
</tr>
<tr>
<td>DAY()</td>
<td>Yes</td>
<td>Synonym for DAYOFMONTH()</td>
</tr>
<tr>
<td>DAYNAME()</td>
<td>Yes</td>
<td>Return the name of the weekday</td>
</tr>
<tr>
<td>DAYOFMONTH()</td>
<td>Yes</td>
<td>Return the day of the month (0-31)</td>
</tr>
<tr>
<td>DAYOFWEEK()</td>
<td></td>
<td>Return the weekday index of the argument</td>
</tr>
<tr>
<td>DAYOFYEAR()</td>
<td>Yes</td>
<td>Return the day of the year (1-366)</td>
</tr>
<tr>
<td>EXTRACT()</td>
<td>Yes</td>
<td>Extract part of a date</td>
</tr>
<tr>
<td>FROM_UNIXTIME()</td>
<td></td>
<td>Format Unix timestamp as a date</td>
</tr>
<tr>
<td>HOUR()</td>
<td>Yes</td>
<td>Extract the hour</td>
</tr>
<tr>
<td>LOCALTIME(), LOCALTIME</td>
<td></td>
<td>Synonym for NOW()</td>
</tr>
<tr>
<td>LOCALTIMESTAMP, LOCALTIMESTAMP</td>
<td></td>
<td>Synonym for NOW()</td>
</tr>
<tr>
<td>MICROSECOND()</td>
<td>Yes</td>
<td>Return the microseconds from argument</td>
</tr>
<tr>
<td>MINUTE()</td>
<td>Yes</td>
<td>Return the minute from the argument</td>
</tr>
<tr>
<td>MONTH()</td>
<td>Yes</td>
<td>Return the month from the date passed</td>
</tr>
<tr>
<td>MONTHNAME()</td>
<td>Yes</td>
<td>Return the name of the month</td>
</tr>
<tr>
<td>NOW()</td>
<td></td>
<td>Return the current date and time</td>
</tr>
<tr>
<td>QUARTER()</td>
<td>Yes</td>
<td>Return the quarter from a date argument</td>
</tr>
<tr>
<td>SECOND()</td>
<td></td>
<td>Return the second (0-59)</td>
</tr>
<tr>
<td>STR_TO_DATE()</td>
<td>Yes</td>
<td>Convert a string to a date</td>
</tr>
</tbody>
</table>
### Logical Operators

The following table shows supported logical operators.

**Table 10.6 Logical Operators**

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

### Mathematical Functions

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

**Table 10.7 Mathematical Functions**

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS()</td>
<td>Return the absolute value</td>
</tr>
<tr>
<td>ACOS()</td>
<td>Return the arc cosine</td>
</tr>
<tr>
<td>ASIN()</td>
<td>Return the arc sine</td>
</tr>
<tr>
<td>ATAN()</td>
<td>Return the arc tangent</td>
</tr>
<tr>
<td>CEIL()</td>
<td>Return the smallest integer value not less than the argument. The function</td>
</tr>
<tr>
<td></td>
<td>is not applied to BIGINT values. The input value is returned. CEIL() is</td>
</tr>
<tr>
<td></td>
<td>a synonym for CEILING().</td>
</tr>
<tr>
<td>UNIX_TIMESTAMP()</td>
<td>Return a Unix timestamp</td>
</tr>
<tr>
<td>WEEK()</td>
<td>Return the week number</td>
</tr>
<tr>
<td>WEEKDAY()</td>
<td>Return the weekday index</td>
</tr>
<tr>
<td>WEEKOFYEAR()</td>
<td>Return the calendar week of the date (1-53)</td>
</tr>
<tr>
<td>YEAR()</td>
<td>Return the year</td>
</tr>
<tr>
<td>TO_DAYS()</td>
<td>Return the date argument converted to days</td>
</tr>
<tr>
<td>TO_SECONDS()</td>
<td>Return the date or datetime argument converted to seconds since Year 0</td>
</tr>
<tr>
<td>TIMESTAMP()</td>
<td>With a single argument, this function returns the date or datetime expression; with two arguments, the sum of the arguments</td>
</tr>
<tr>
<td>TIMESTAMPDIFF()</td>
<td>Subtract an interval from a datetime expression</td>
</tr>
<tr>
<td>TIME()</td>
<td>Extract the time portion of the expression passed</td>
</tr>
<tr>
<td>TIME_FORMAT()</td>
<td>Format as time.</td>
</tr>
<tr>
<td>TIME_TO_SEC()</td>
<td>Return the argument converted to seconds</td>
</tr>
<tr>
<td>SUBTIME()</td>
<td>Subtract times</td>
</tr>
</tbody>
</table>
### String Functions and Operators

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

### 10.2.9 String Functions and Operators

The following table shows supported string functions and operators. The string functions and operators described in the following table are supported with variable-length encoded columns. They are not supported with dictionary encoded columns.

#### Table 10.8 String Functions and Operators

<table>
<thead>
<tr>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASCII()</td>
<td>Return numeric value of left-most character</td>
</tr>
<tr>
<td>BIT_LENGTH()</td>
<td>Return length of argument in bits</td>
</tr>
<tr>
<td>CHAR_LENGTH()</td>
<td>Return number of characters in argument</td>
</tr>
<tr>
<td>CONCAT()</td>
<td>Return concatenated string</td>
</tr>
<tr>
<td>CONCAT_WS()</td>
<td>Return concatenated with separator</td>
</tr>
<tr>
<td>FIND_IN_SET()</td>
<td>Index (position) of first argument within second argument</td>
</tr>
<tr>
<td>FROM_BASE64()</td>
<td>Decode base64 encoded string and return result</td>
</tr>
<tr>
<td>GREATEST()</td>
<td>Return the largest argument. Not supported with temporal columns</td>
</tr>
<tr>
<td>HEX()</td>
<td>Hexadecimal representation of decimal or string value</td>
</tr>
<tr>
<td>INSERT()</td>
<td>Return the index of the first occurrence of substring</td>
</tr>
<tr>
<td>INSTR()</td>
<td>Return the index of the first occurrence of substring</td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
</tr>
<tr>
<td>-----------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>LEAST()</td>
<td>Return the smallest argument. Not supported with temporal columns.</td>
</tr>
<tr>
<td>LEFT()</td>
<td>Return the leftmost number of characters as specified</td>
</tr>
<tr>
<td>LENGTH()</td>
<td>Return the length of a string in bytes</td>
</tr>
<tr>
<td>LIKE</td>
<td>Simple pattern matching</td>
</tr>
<tr>
<td>LOCATE()</td>
<td>Return the position of the first occurrence of substring</td>
</tr>
<tr>
<td>LOWER()</td>
<td>Return the argument in lowercase</td>
</tr>
<tr>
<td>LPAD()</td>
<td>Return the string argument, left-padded with the specified string</td>
</tr>
<tr>
<td>LTRIM()</td>
<td>Remove leading spaces</td>
</tr>
<tr>
<td>NOT LIKE</td>
<td>Negation of simple pattern matching</td>
</tr>
<tr>
<td>OCTET_LENGTH()</td>
<td>Synonym for LENGTH()</td>
</tr>
<tr>
<td>ORD()</td>
<td>Return character code for leftmost character of the argument</td>
</tr>
<tr>
<td>POSITION()</td>
<td>Synonym for LOCATE()</td>
</tr>
<tr>
<td>REPEAT()</td>
<td>Repeat a string the specified number of times</td>
</tr>
<tr>
<td>QUOTE()</td>
<td>Escape the argument for use in an SQL statement</td>
</tr>
<tr>
<td>REGEXP</td>
<td>Whether string matches regular expression</td>
</tr>
<tr>
<td>REGEXP_LIKE()</td>
<td>Whether string matches regular expression</td>
</tr>
<tr>
<td>REPLACE()</td>
<td>Replace occurrences of a specified string</td>
</tr>
<tr>
<td>REVERSE()</td>
<td>Reverse the characters in a string</td>
</tr>
<tr>
<td>RIGHT()</td>
<td>Return the specified rightmost number of characters</td>
</tr>
<tr>
<td>RLIKE</td>
<td>Whether string matches regular expression</td>
</tr>
<tr>
<td>RPAD()</td>
<td>Append string the specified number of times</td>
</tr>
<tr>
<td>RTRIM()</td>
<td>Remove trailing spaces</td>
</tr>
<tr>
<td>STRCMP()</td>
<td>Compare two strings</td>
</tr>
<tr>
<td>SUBSTR()</td>
<td>Return the substring as specified</td>
</tr>
<tr>
<td>SUBSTRING()</td>
<td>Return the substring as specified</td>
</tr>
<tr>
<td>SUBSTRING_INDEX()</td>
<td>Return a substring from a string before the specified number of occurrences of the delimiter</td>
</tr>
<tr>
<td>TO_BASE64()</td>
<td>Return the argument converted to a base-64 string</td>
</tr>
<tr>
<td>TRIM()</td>
<td>Remove leading and trailing spaces</td>
</tr>
<tr>
<td>UNHEX()</td>
<td>Return a string containing hex representation of a number</td>
</tr>
<tr>
<td>UPPER()</td>
<td>Convert to uppercase</td>
</tr>
</tbody>
</table>

### 10.3 Supported SQL Modes

The 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:
10.4 String Column Encoding Reference

HeatWave supports two string column encoding types:

- **Section 10.4.1, “Variable-length Encoding”**
- **Section 10.4.2, “Dictionary Encoding”**

String column encoding is automatically applied when tables are loaded into the HeatWave cluster. Variable-length encoding is the default.

To use dictionary encoding, you must define the encoding type explicitly for individual string columns. See Defining Dictionary Encoding.

10.4.1 Variable-length Encoding

Variable-length encoding (VARLEN) 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 that are 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 collations types supported by the MySQL DB System. User defined character sets are not supported.
- **VARLEN** columns can be declared as NULL.

10.4.1.1 VARLEN Supported Expressions, Filters, Functions, and Operators

For supported functions and operators, refer to Section 10.2, “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.
Variable-length Encoding

• 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

10.4.1.2 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.

10.4.1.3 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.

10.4.1.4 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:
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 the HeatWave cluster to the MySQL DB System is greater.

10.4.2 Dictionary Encoding

Dictionary encoding (SORTED) has the following characteristics:

- It is best suited for 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.
- It supports GROUP BY and ORDER BY operations on string columns.
- It 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.
- It does not support JOIN operations.
- It does not support operations that use string operators. Queries that use string operators on dictionary-encoded string columns are not offloaded.
- It does not support LIKE predicates.
- 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.

10.5 Metadata Queries

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

- 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:

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

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

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

- 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:

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

<table>
<thead>
<tr>
<th>COLUMN_NAME</th>
<th>COLUMN_COMMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>O_CLERK</td>
<td>RAPID_COLUMN=ENCODING=SORTED</td>
</tr>
</tbody>
</table>
### Metadata Queries

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

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

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

<table>
<thead>
<tr>
<th>COLUMN_NAME</th>
<th>COLUMN_COMMENT</th>
</tr>
</thead>
<tbody>
<tr>
<td>O_CUSTKEY</td>
<td>RAPID_COLUMN=DATA_PLACEMENT_KEY=1</td>
</tr>
<tr>
<td>O_ORDERPRIORITY</td>
<td>RAPID_COLUMN=ENCODING=SORTED</td>
</tr>
</tbody>
</table>

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

- To identify tables on the MySQL DB System that are defined with a secondary engine, query the `CREATE_OPTIONS` column of the `INFORMATION_SCHEMA.TABLES` table. The `CREATE_OPTIONS` column shows the `SECONDARY_ENGINE` clause, if defined.

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

<table>
<thead>
<tr>
<th>TABLE_SCHEMA</th>
<th>TABLE_NAME</th>
<th>CREATE_OPTIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>tpch</td>
<td>customer</td>
<td>SECONDARY_ENGINE=&quot;RAPID&quot;</td>
</tr>
<tr>
<td>tpch</td>
<td>lineitem</td>
<td>SECONDARY_ENGINE=&quot;RAPID&quot;</td>
</tr>
<tr>
<td>tpch</td>
<td>nation</td>
<td>SECONDARY_ENGINE=&quot;RAPID&quot;</td>
</tr>
<tr>
<td>tpch</td>
<td>orders</td>
<td>SECONDARY_ENGINE=&quot;RAPID&quot;</td>
</tr>
<tr>
<td>tpch</td>
<td>part</td>
<td>SECONDARY_ENGINE=&quot;RAPID&quot;</td>
</tr>
<tr>
<td>tpch</td>
<td>partsupp</td>
<td>SECONDARY_ENGINE=&quot;RAPID&quot;</td>
</tr>
<tr>
<td>tpch</td>
<td>region</td>
<td>SECONDARY_ENGINE=&quot;RAPID&quot;</td>
</tr>
<tr>
<td>tpch</td>
<td>supplier</td>
<td>SECONDARY_ENGINE=&quot;RAPID&quot;</td>
</tr>
</tbody>
</table>

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

#### Note

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

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

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

```
42
```
To check the load status of a table in the HeatWave cluster, query the `LOAD_STATUS` data from HeatWave Performance Schema tables. For example:

```sql
mysql> USE performance_schema;
mysql> SELECT NAME, LOAD_STATUS FROM rpd_tables,rpd_table_id
    WHERE rpd_tables.ID = rpd_table_id.ID 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 10.10.4, “The rpd_tables Table”.

To check whether change propagation is enabled or disabled for a particular table, query the `POOL_TYPE` data from 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 disabled.

```sql
mysql> SELECT NAME, POOL_TYPE FROM rpd_tables,rpd_table_id
    WHERE rpd_tables.ID = rpd_table_id.ID AND SCHEMA_NAME LIKE 'tpch';
```

```
+---------------+-------------------------------+
| NAME          | POOL_TYPE                     |
+---------------+-------------------------------+
| tpch.orders   | RAPID_LOAD_POOL_TRANSACTIONAL |
| tpch.region   | RAPID_LOAD_POOL_TRANSACTIONAL |
| tpch.lineitem | RAPID_LOAD_POOL_TRANSACTIONAL |
| tpch.supplier | RAPID_LOAD_POOL_TRANSACTIONAL |
| tpch.partsupp | RAPID_LOAD_POOL_TRANSACTIONAL |
| tpch.part     | RAPID_LOAD_POOL_TRANSACTIONAL |
| tpch.customer | RAPID_LOAD_POOL_TRANSACTIONAL |
+---------------+-------------------------------+
```

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

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

```
+----------------+
| VARIABLE_VALUE |
+----------------+
| ON             |
+----------------+
```

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

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

```
+----------------+
| VARIABLE_VALUE |
+----------------+
43
+----------------+
```
10.6 Limitations

This section lists functions, data types, variables, JOIN types, SQL modes, and other expressions and functionality not supported by HeatWave.

- Functions:
  - Bit functions and operators.
  - `COALESCE()` as a JOIN predicate.
  - Window functions.
  - Full-text search functions.
  - XML, JSON, Spatial, and other domain specific functions.
  - Encryption and compression functions.
  - User Defined Functions (UDFs).
  - `LAST_INSERT_ID()`
  - `FOUND_ROWS()`
  - 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.
  - The `EXTRACT()` function with a temporal interval unit.
  - 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 10.1, "Supported Data Types".

• Data types:
  • **BINARY**
  • **BIT**
  • **BLOB**
  • **JSON**
  • **VARBINARY**

• Spatial data types. See Spatial Data Types.

• The **DECIMAL** type with precision greater than 18 with expression operations.

• For the **ENUM** type, functions other than those listed below:
  • Comparison functions.
    • **IS [NOT] NULL, IS [NOT] TRUE, IS [NOT] FALSE**
    • **COALESCE(), LEAST(), GREATEST()**
    • **CONCAT(), CONCAT_WS()**
  • **CAST()** as **CHAR**, or as a **SIGNED** or **UNSIGNED** integer.

For **enum_col BETWEEN string AND string**, HeatWave results may not match MySQL results due differences in how string values are compared to **ENUM** values. Also, HeatWave does not recognize `' '` and `'_ '` substitution characters when comparing string values to **ENUM** values, which can cause HeatWave and MySQL result mismatches. For example, MySQL matches `%mall` to the **ENUM** value 'small', whereas HeatWave views '%mall' as a regular string.

• 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.
Limitations

- **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**. Therefore, the **VARCHAR** type cannot be joined only with **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 the HeatWave cluster.)

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 10.3, “Supported SQL Modes”.

• Other:
  • Multiple instances of \texttt{COUNT(DISTINCT value)}, \texttt{SUM(DISTINCT value)}, \texttt{AVG(DISTINCT value)} expressions in a query are permitted only if they specify the same \textit{value}.
  
  • \texttt{UNION ALL} queries with an \texttt{ORDER BY} or \texttt{LIMIT} clause, between different column types, between dictionary-encoded columns, or between \texttt{ENUM} columns.

  • \texttt{UNION} queries with or without an \texttt{ORDER BY} or \texttt{LIMIT} clause, between different column types, between dictionary-encoded columns, or between \texttt{ENUM} columns.

  • \texttt{UNION} and \texttt{UNION ALL} subqueries with or without an \texttt{ORDER BY} or \texttt{LIMIT} clause, between different column types, between dictionary-encoded columns, between \texttt{ENUM} columns, or specified in an \texttt{IN} or \texttt{EXISTS} clause.

  A mix of \texttt{UNION} and \texttt{UNION ALL} at the same level in a query.

  • Comparison predicates, \texttt{GROUP BY}, \texttt{JOIN}, and so on, if the key column is \texttt{DOUBLE PRECISION}.

  • Type conversion on relational data. For example, \texttt{SELECT CONCAT(2, L\_COMMENT) from \texttt{LINEITEM};} is not supported.

  • Queries with an impossible \texttt{WHERE} condition (queries known to have an empty result set). For example, the following query is not offloaded:

  \[
  \text{SELECT AVG(c1) AS value FROM t1 WHERE c1 IS NULL;}
  \]

  • Querying of \texttt{YEAR} type data using expressions and other functions. For example, the following queries are not offloaded:

  \[
  \text{SELECT YEAR(d) + 1 FROM t1;}
  \]
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.
- Concurrent processing of queries. A HeatWave cluster processes one query at a time. Concurrently issued queries are queued internally and processed in the order that they arrive.
- Partial query offload. Queries are offloaded to the HeatWave cluster entirely or not at all. 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.

- `SET timezone = timezone`, with the `timezone` value specified as an offset from UTC in the form of [+][-]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.

## 10.7 System Variables

HeatWave maintains several variables that configure its operation. Variables are set when the HeatWave cluster is enabled.

- **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                        |

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*

<table>
<thead>
<tr>
<th>Command-Line Format</th>
<th>--rapid-dmem-size=#</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduced</td>
<td>8.0.17</td>
</tr>
<tr>
<td>System Variable</td>
<td>rapid_dmem_size</td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Yes</td>
</tr>
<tr>
<td>SET_VAR Hint Applies</td>
<td>No</td>
</tr>
<tr>
<td>Type</td>
<td>Integer</td>
</tr>
<tr>
<td>Default Value</td>
<td>2048</td>
</tr>
<tr>
<td>Minimum Value</td>
<td>512</td>
</tr>
<tr>
<td>Maximum Value</td>
<td>2097152</td>
</tr>
</tbody>
</table>

Specifies the amount of DMEM available on each core of each node, in bytes.

### *rapid_memory_heap_size*

<table>
<thead>
<tr>
<th>Command-Line Format</th>
<th>--rapid-memory-heap-size=#</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduced</td>
<td>8.0.17</td>
</tr>
<tr>
<td>System Variable</td>
<td>rapid_memory_heap_size</td>
</tr>
<tr>
<td>Scope</td>
<td>Global</td>
</tr>
<tr>
<td>Dynamic</td>
<td>Yes</td>
</tr>
<tr>
<td>SET_VAR Hint Applies</td>
<td>No</td>
</tr>
<tr>
<td>Type</td>
<td>Integer</td>
</tr>
<tr>
<td>Default Value</td>
<td>unlimited</td>
</tr>
<tr>
<td>Minimum Value</td>
<td>67108864</td>
</tr>
<tr>
<td>Maximum Value</td>
<td>unlimited</td>
</tr>
</tbody>
</table>

The amount of memory available for the HeatWave plugin, in bytes. Ensures that HeatWave does not use more memory than is allocated to it. If not set, HeatWave is free to use any amount of memory available.

The minimum value is 67108864 (64MiB).
Secondary Engine Variables

- **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_CONSUMATION                              |

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

  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;
  ````

  **10.8 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 SELECT statements using the secondary engine. These values are permitted:

  - **OFF**: SELECT statements execute using the primary storage (InnoDB) on the MySQL DB System. Execution using the secondary engine (RAPID) is disabled.

  - **ON**: SELECT statements 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** SELECT statements always execute using the secondary engine (RAPID), or fail if that is not possible. Under this mode, a SELECT statement 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.

### 10.9 Status Variables

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

```sql
mysql> SHOW STATUS LIKE 'rapid%';
+---------------------------------+------------+
| Variable_name                   | Value      |
+---------------------------------+------------+
| rapid_change_propagation_status | ON         |
| rapid_cluster_status            | ON         |
| rapid_core_count                | 64         |
| rapid_heap_usage                | 58720397   |
| rapid_load_progress             | 100.00000  |
| rapid_plugin_bootstrapped       | YES        |
| rapid_preload_stats_status      | Available  |
| rapid_query_offload_count       | 46         |
| rapid_service_status            | ONLINE     |
```

- **rapid_change_propagation_status**

  The change propagation status.

  A status of ON indicates that change propagation is enabled, 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 to a MySQL DB System. 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 the cluster 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.

### 10.10 Performance Schema Tables

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

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*.

### 10.10.1 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.

10.10.2 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.
10.10.3 The rpd_table_id Table

The rpd_table_id table provides the ID, name, and schema of the tables loaded in the HeatWave cluster.

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. DDL, including TRUNCATE TABLE, is not permitted.

10.10.4 The rpd_tables Table

The rpd_tables table provides the system change number (SCN) and load pool type for tables loaded in the HeatWave cluster.

The rpd_tables table has these columns:

• **ID**
  A unique identifier for the table.
The `rpd_column_id` 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. DDL, including `TRUNCATE TABLE`, is not permitted.

10.10.5 The `rpd_column_id` Table

The `rpd_column_id` table provides information about columns of tables that are loaded in the HeatWave cluster.

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 rpd_columns Table

- **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_columns` table is read-only. DDL, including `TRUNCATE TABLE`, is not permitted.

### 10.10.6 The rpd_columns Table

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

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`.

The `rpd_columns` table is read-only. DDL, including `TRUNCATE TABLE`, is not permitted.

### 10.10.7 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.
10.11 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