```
| Ndb_api_trans_abort_count_replica             | 0         |
| Ndb_api_trans_close_count_replica             | 0         |
| Ndb_api_pk_op_count_replica                   | 0         |
| Ndb_api_uk_op_count_replica                   | 0         |
| Ndb_api_table_scan_count_replica              | 0         |
| Ndb_api_range_scan_count_replica              | 0         |
| Ndb_api_pruned_scan_count_replica             | 0         |
| Ndb_api_scan_batch_count_replica              | 0         |
| Ndb_api_read_row_count_replica                | 0         |
| Ndb_api_trans_local_read_row_count_replica    | 0         |
| Ndb_api_adaptive_send_forced_count_replica    | 0         |
| Ndb_api_adaptive_send_unforced_count_replica | 0         |
| Ndb_api_adaptive_send_deferred_count_replica | 0         |
| Ndb_api_event_data_count_injector             | 0         |
| Ndb_api_event_nondata_count_injector          | 0         |
| Ndb_api_event_bytes_count_injector            | 0         |
| Ndb_api_wait_exec_complete_count_session      | 0         |
| Ndb_api_wait_scan_result_count_session        | 0         |
| Ndb_api_wait_meta_request_count_session       | 0         |
| Ndb_api_wait_nanos_count_session              | 0         |
| Ndb_api_bytes_sent_count_session              | 0         |
| Ndb_api_bytes_received_count_session          | 0         |
| Ndb_api_trans_start_count_session             | 0         |
| Ndb_api_trans_commit_count_session            | 0         |
| Ndb_api_trans_abort_count_session             | 0         |
| Ndb_api_trans_close_count_session             | 0         |
| Ndb_api_pk_op_count_session                   | 0         |
| Ndb_api_uk_op_count_session                   | 0         |
| Ndb_api_table_scan_count_session              | 0         |
| Ndb_api_range_scan_count_session              | 0         |
| Ndb_api_pruned_scan_count_session             | 0         |
| Ndb_api_scan_batch_count_session              | 0         |
| Ndb_api_read_row_count_session                | 0         |
| Ndb_api_trans_local_read_row_count_session    | 0         |
| Ndb_api_adaptive_send_forced_count_session    | 0         |
| Ndb_api_adaptive_send_unforced_count_session | 0         |
| Ndb_api_adaptive_send_deferred_count_session | 0         |
+-----------------------------------------------+-----------+
90 rows in set (0.01 sec)
```

These status variables are also available from the Performance Schema `session_status` and `global_status` tables, as shown here:

```
mysql> SELECT * FROM performance_schema.session_status
    ->   WHERE VARIABLE_NAME LIKE 'ndb_api%';
+-----------------------------------------------+----------------+
| VARIABLE_NAME                                 | VARIABLE_VALUE |
+-----------------------------------------------+----------------+
| Ndb_api_wait_exec_complete_count              | 617            |
| Ndb_api_wait_scan_result_count                | 0              |
| Ndb_api_wait_meta_request_count               | 649            |
| Ndb_api_wait_nanos_count                      | 335663491      |
| Ndb_api_bytes_sent_count                      | 65764          |
| Ndb_api_bytes_received_count                  | 86940          |
| Ndb_api_trans_start_count                     | 308            |
| Ndb_api_trans_commit_count                    | 308            |
| Ndb_api_trans_abort_count                     | 0              |
| Ndb_api_trans_close_count                     | 308            |
| Ndb_api_pk_op_count                           | 311            |
| Ndb_api_uk_op_count                           | 0              |
| Ndb_api_table_scan_count                      | 0              |
| Ndb_api_range_scan_count                      | 0              |
| Ndb_api_pruned_scan_count                     | 0              |
| Ndb_api_scan_batch_count                      | 0              |
| Ndb_api_read_row_count                        | 307            |
| Ndb_api_trans_local_read_row_count            | 77             |
| Ndb_api_adaptive_send_forced_count            | 3              |
| Ndb_api_adaptive_send_unforced_count          | 614            |
| Ndb_api_adaptive_send_deferred_count          | 0              |
| Ndb_api_event_data_count                      | 0              |
| Ndb_api_event_nondata_count                   | 0              |
| Ndb_api_event_bytes_count                     | 0              |
```

```
| Ndb_api_wait_exec_complete_count_slave       | 0               |
| Ndb_api_wait_scan_result_count_slave         | 0               |
| Ndb_api_wait_meta_request_count_slave        | 0               |
| Ndb_api_wait_nanos_count_slave               | 0               |
| Ndb_api_bytes_sent_count_slave               | 0               |
| Ndb_api_bytes_received_count_slave           | 0               |
| Ndb_api_trans_start_count_slave              | 0               |
| Ndb_api_trans_commit_count_slave             | 0               |
| Ndb_api_trans_abort_count_slave              | 0               |
| Ndb_api_trans_close_count_slave              | 0               |
| Ndb_api_pk_op_count_slave                    | 0               |
| Ndb_api_uk_op_count_slave                    | 0               |
| Ndb_api_table_scan_count_slave               | 0               |
| Ndb_api_range_scan_count_slave               | 0               |
| Ndb_api_pruned_scan_count_slave              | 0               |
| Ndb_api_scan_batch_count_slave               | 0               |
| Ndb_api_read_row_count_slave                 | 0               |
| Ndb_api_trans_local_read_row_count_slave     | 0               |
| Ndb_api_adaptive_send_forced_count_slave     | 0               |
| Ndb_api_adaptive_send_unforced_count_slave   | 0               |
| Ndb_api_adaptive_send_deferred_count_slave   | 0               |
| Ndb_api_wait_exec_complete_count_replica     | 0               |
| Ndb_api_wait_scan_result_count_replica       | 0               |
| Ndb_api_wait_meta_request_count_replica      | 0               |
| Ndb_api_wait_nanos_count_replica             | 0               |
| Ndb_api_bytes_sent_count_replica             | 0               |
| Ndb_api_bytes_received_count_replica         | 0               |
| Ndb_api_trans_start_count_replica            | 0               |
| Ndb_api_trans_commit_count_replica           | 0               |
| Ndb_api_trans_abort_count_replica            | 0               |
| Ndb_api_trans_close_count_replica            | 0               |
| Ndb_api_pk_op_count_replica                  | 0               |
| Ndb_api_uk_op_count_replica                  | 0               |
| Ndb_api_table_scan_count_replica             | 0               |
| Ndb_api_range_scan_count_replica             | 0               |
| Ndb_api_pruned_scan_count_replica            | 0               |
| Ndb_api_scan_batch_count_replica             | 0               |
| Ndb_api_read_row_count_replica               | 0               |
| Ndb_api_trans_local_read_row_count_replica   | 0               |
| Ndb_api_adaptive_send_forced_count_replica   | 0               |
| Ndb_api_adaptive_send_unforced_count_replica | 0               |
| Ndb_api_adaptive_send_deferred_count_replica | 0               |
| Ndb_api_event_data_count_injector            | 0               |
| Ndb_api_event_nondata_count_injector         | 0               |
| Ndb_api_event_bytes_count_injector           | 0               |
| Ndb_api_wait_exec_complete_count_session     | 0               |
| Ndb_api_wait_scan_result_count_session       | 0               |
| Ndb_api_wait_meta_request_count_session      | 0               |
| Ndb_api_wait_nanos_count_session             | 0               |
| Ndb_api_bytes_sent_count_session             | 0               |
| Ndb_api_bytes_received_count_session         | 0               |
| Ndb_api_trans_start_count_session            | 0               |
| Ndb_api_trans_commit_count_session           | 0               |
| Ndb_api_trans_abort_count_session            | 0               |
| Ndb_api_trans_close_count_session            | 0               |
| Ndb_api_pk_op_count_session                  | 0               |
| Ndb_api_uk_op_count_session                  | 0               |
| Ndb_api_table_scan_count_session             | 0               |
| Ndb_api_range_scan_count_session             | 0               |
| Ndb_api_pruned_scan_count_session            | 0               |
| Ndb_api_scan_batch_count_session             | 0               |
| Ndb_api_read_row_count_session               | 0               |
| Ndb_api_trans_local_read_row_count_session   | 0               |
| Ndb_api_adaptive_send_forced_count_session   | 0               |
| Ndb_api_adaptive_send_unforced_count_session | 0               |
| Ndb_api_adaptive_send_deferred_count_session | 0               |
+----------------------------------------------+-----------------+
90 rows in set (0.01 sec)

mysql> SELECT * FROM performance_schema.global_status
    ->     WHERE VARIABLE_NAME LIKE 'ndb_api%';
+----------------------------------------------+-----------------+
```

```
| VARIABLE_NAME                                | VARIABLE_VALUE |
+----------------------------------------------+----------------+
| Ndb_api_wait_exec_complete_count             | 741            |
| Ndb_api_wait_scan_result_count               | 0              |
| Ndb_api_wait_meta_request_count              | 777            |
| Ndb_api_wait_nanos_count                     | 373888309      |
| Ndb_api_bytes_sent_count                     | 78124          |
| Ndb_api_bytes_received_count                 | 94988          |
| Ndb_api_trans_start_count                    | 370            |
| Ndb_api_trans_commit_count                   | 370            |
| Ndb_api_trans_abort_count                    | 0              |
| Ndb_api_trans_close_count                    | 370            |
| Ndb_api_pk_op_count                          | 373            |
| Ndb_api_uk_op_count                          | 0              |
| Ndb_api_table_scan_count                     | 0              |
| Ndb_api_range_scan_count                     | 0              |
| Ndb_api_pruned_scan_count                    | 0              |
| Ndb_api_scan_batch_count                     | 0              |
| Ndb_api_read_row_count                       | 369            |
| Ndb_api_trans_local_read_row_count           | 93             |
| Ndb_api_adaptive_send_forced_count           | 3              |
| Ndb_api_adaptive_send_unforced_count         | 738            |
| Ndb_api_adaptive_send_deferred_count         | 0              |
| Ndb_api_event_data_count                     | 0              |
| Ndb_api_event_nondata_count                  | 0              |
| Ndb_api_event_bytes_count                    | 0              |
| Ndb_api_wait_exec_complete_count_slave       | 0              |
| Ndb_api_wait_scan_result_count_slave         | 0              |
| Ndb_api_wait_meta_request_count_slave        | 0              |
| Ndb_api_wait_nanos_count_slave               | 0              |
| Ndb_api_bytes_sent_count_slave               | 0              |
| Ndb_api_bytes_received_count_slave           | 0              |
| Ndb_api_trans_start_count_slave              | 0              |
| Ndb_api_trans_commit_count_slave             | 0              |
| Ndb_api_trans_abort_count_slave              | 0              |
| Ndb_api_trans_close_count_slave              | 0              |
| Ndb_api_pk_op_count_slave                    | 0              |
| Ndb_api_uk_op_count_slave                    | 0              |
| Ndb_api_table_scan_count_slave               | 0              |
| Ndb_api_range_scan_count_slave               | 0              |
| Ndb_api_pruned_scan_count_slave              | 0              |
| Ndb_api_scan_batch_count_slave               | 0              |
| Ndb_api_read_row_count_slave                 | 0              |
| Ndb_api_trans_local_read_row_count_slave     | 0              |
| Ndb_api_adaptive_send_forced_count_slave     | 0              |
| Ndb_api_adaptive_send_unforced_count_slave   | 0              |
| Ndb_api_adaptive_send_deferred_count_slave   | 0              |
| Ndb_api_wait_exec_complete_count_replica     | 0              |
| Ndb_api_wait_scan_result_count_replica       | 0              |
| Ndb_api_wait_meta_request_count_replica      | 0              |
| Ndb_api_wait_nanos_count_replica             | 0              |
| Ndb_api_bytes_sent_count_replica             | 0              |
| Ndb_api_bytes_received_count_replica         | 0              |
| Ndb_api_trans_start_count_replica            | 0              |
| Ndb_api_trans_commit_count_replica           | 0              |
| Ndb_api_trans_abort_count_replica            | 0              |
| Ndb_api_trans_close_count_replica            | 0              |
| Ndb_api_pk_op_count_replica                  | 0              |
| Ndb_api_uk_op_count_replica                  | 0              |
| Ndb_api_table_scan_count_replica             | 0              |
| Ndb_api_range_scan_count_replica             | 0              |
| Ndb_api_pruned_scan_count_replica            | 0              |
| Ndb_api_scan_batch_count_replica             | 0              |
| Ndb_api_read_row_count_replica               | 0              |
| Ndb_api_trans_local_read_row_count_replica   | 0              |
| Ndb_api_adaptive_send_forced_count_replica   | 0              |
| Ndb_api_adaptive_send_unforced_count_replica | 0              |
| Ndb_api_adaptive_send_deferred_count_replica | 0              |
| Ndb_api_event_data_count_injector            | 0              |
| Ndb_api_event_nondata_count_injector         | 0              |
| Ndb_api_event_bytes_count_injector           | 0              |
| Ndb_api_wait_exec_complete_count_session     | 0              |
```

```
| Ndb_api_wait_scan_result_count_session       | 0              |
| Ndb_api_wait_meta_request_count_session      | 0              |
| Ndb_api_wait_nanos_count_session             | 0              |
| Ndb_api_bytes_sent_count_session             | 0              |
| Ndb_api_bytes_received_count_session         | 0              |
| Ndb_api_trans_start_count_session            | 0              |
| Ndb_api_trans_commit_count_session           | 0              |
| Ndb_api_trans_abort_count_session            | 0              |
| Ndb_api_trans_close_count_session            | 0              |
| Ndb_api_pk_op_count_session                  | 0              |
| Ndb_api_uk_op_count_session                  | 0              |
| Ndb_api_table_scan_count_session             | 0              |
| Ndb_api_range_scan_count_session             | 0              |
| Ndb_api_pruned_scan_count_session            | 0              |
| Ndb_api_scan_batch_count_session             | 0              |
| Ndb_api_read_row_count_session               | 0              |
| Ndb_api_trans_local_read_row_count_session   | 0              |
| Ndb_api_adaptive_send_forced_count_session   | 0              |
| Ndb_api_adaptive_send_unforced_count_session | 0              |
| Ndb_api_adaptive_send_deferred_count_session | 0              |
+----------------------------------------------+----------------+
90 rows in set (0.01 sec)
```

Each `Ndb` object has its own counters. NDB API applications can read the values of the counters for use in optimization or monitoring. For multithreaded clients which use more than one `Ndb` object concurrently, it is also possible to obtain a summed view of counters from all `Ndb` objects belonging to a given `Ndb_cluster_connection`.

Four sets of these counters are exposed. One set applies to the current session only; the other 3 are global. *This is in spite of the fact that their values can be obtained as either session or global status variables in the `mysql` client*. This means that specifying the `SESSION` or `GLOBAL` keyword with `SHOW STATUS` has no effect on the values reported for NDB API statistics status variables, and the value for each of these variables is the same whether the value is obtained from the equivalent column of the `session_status` or the `global_status` table.

- *Session counters (session specific)*

  Session counters relate to the `Ndb` objects in use by (only) the current session. Use of such objects by other MySQL clients does not influence these counts.

  In order to minimize confusion with standard MySQL session variables, we refer to the variables that correspond to these NDB API session counters as "`_session` variables", with a leading underscore.

- *Replica counters (global)*

  This set of counters relates to the `Ndb` objects used by the replica SQL thread, if any. If this `mysqld` does not act as a replica, or does not use `NDB` tables, then all of these counts are 0.

  We refer to the related status variables as "`_slave` variables" (with a leading underscore).

- *Injector counters (global)*

  Injector counters relate to the `Ndb` object used to listen to cluster events by the binary log injector thread. Even when not writing a binary log, `mysqld` processes attached to an NDB Cluster continue to listen for some events, such as schema changes.

  We refer to the status variables that correspond to NDB API injector counters as "`_injector` variables" (with a leading underscore).

- *Server (Global) counters (global)*

  This set of counters relates to all `Ndb` objects currently used by this `mysqld`. This includes all MySQL client applications, the replica SQL thread (if any), the binary log injector, and the `NDB` utility thread.

We refer to the status variables that correspond to these counters as “global variables” or “`mysqld`-level variables”.

You can obtain values for a particular set of variables by additionally filtering for the substring `session`, `slave`, or `injector` in the variable name (along with the common prefix `Ndb_api`). For `_session` variables, this can be done as shown here:

```
mysql> SHOW STATUS LIKE 'ndb_api%session';
+-----------------------------------------+---------+
| Variable_name                           | Value   |
+-----------------------------------------+---------+
| Ndb_api_wait_exec_complete_count_session   | 2       |
| Ndb_api_wait_scan_result_count_session     | 0       |
| Ndb_api_wait_meta_request_count_session    | 1       |
| Ndb_api_wait_nanos_count_session           | 8144375 |
| Ndb_api_bytes_sent_count_session           | 68      |
| Ndb_api_bytes_received_count_session       | 84      |
| Ndb_api_trans_start_count_session          | 1       |
| Ndb_api_trans_commit_count_session         | 1       |
| Ndb_api_trans_abort_count_session          | 0       |
| Ndb_api_trans_close_count_session          | 1       |
| Ndb_api_pk_op_count_session                | 1       |
| Ndb_api_uk_op_count_session                | 0       |
| Ndb_api_table_scan_count_session           | 0       |
| Ndb_api_range_scan_count_session           | 0       |
| Ndb_api_pruned_scan_count_session          | 0       |
| Ndb_api_scan_batch_count_session           | 0       |
| Ndb_api_read_row_count_session             | 1       |
| Ndb_api_trans_local_read_row_count_session | 1       |
+-----------------------------------------+---------+
18 rows in set (0.50 sec)
```

To obtain a listing of the NDB API `mysqld`-level status variables, filter for variable names beginning with `ndb_api` and ending in `_count`, like this:

```
mysql> SELECT * FROM performance_schema.session_status
    ->     WHERE VARIABLE_NAME LIKE 'ndb_api%count';
+-----------------------------------+----------------+
| VARIABLE_NAME                     | VARIABLE_VALUE |
+-----------------------------------+----------------+
| NDB_API_WAIT_EXEC_COMPLETE_COUNT  | 4              |
| NDB_API_WAIT_SCAN_RESULT_COUNT    | 3              |
| NDB_API_WAIT_META_REQUEST_COUNT   | 28             |
| NDB_API_WAIT_NANOS_COUNT          | 53756398       |
| NDB_API_BYTES_SENT_COUNT          | 1060           |
| NDB_API_BYTES_RECEIVED_COUNT      | 9724           |
| NDB_API_TRANS_START_COUNT         | 3              |
| NDB_API_TRANS_COMMIT_COUNT        | 2              |
| NDB_API_TRANS_ABORT_COUNT         | 0              |
| NDB_API_TRANS_CLOSE_COUNT         | 3              |
| NDB_API_PK_OP_COUNT               | 2              |
| NDB_API_UK_OP_COUNT               | 0              |
| NDB_API_TABLE_SCAN_COUNT          | 1              |
| NDB_API_RANGE_SCAN_COUNT          | 0              |
| NDB_API_PRUNED_SCAN_COUNT         | 0              |
| NDB_API_SCAN_BATCH_COUNT          | 0              |
| NDB_API_READ_ROW_COUNT            | 2              |
| NDB_API_TRANS_LOCAL_READ_ROW_COUNT | 2              |
| NDB_API_EVENT_DATA_COUNT          | 0              |
| NDB_API_EVENT_NONDATA_COUNT       | 0              |
| NDB_API_EVENT_BYTES_COUNT         | 0              |
+-----------------------------------+----------------+
21 rows in set (0.09 sec)
```

Not all counters are reflected in all 4 sets of status variables. For the event counters `DataEventsRecvdCount`, `NondataEventsRecvdCount`, and `EventBytesRecvdCount`, only `_injector` and `mysqld`-level NDB API status variables are available:

```
mysql> SHOW STATUS LIKE 'ndb_api%event%';
```

```
+-------------------------------------+-------+
| Variable_name                       | Value |
+-------------------------------------+-------+
| Ndb_api_event_data_count_injector   | 0     |
| Ndb_api_event_nondata_count_injector | 0     |
| Ndb_api_event_bytes_count_injector  | 0     |
| Ndb_api_event_data_count            | 0     |
| Ndb_api_event_nondata_count         | 0     |
| Ndb_api_event_bytes_count           | 0     |
+-------------------------------------+-------+
6 rows in set (0.00 sec)
```

`_injector` status variables are not implemented for any other NDB API counters, as shown here:

```
mysql> SHOW STATUS LIKE 'ndb_api%injector%';
+-------------------------------------+-------+
| Variable_name                       | Value |
+-------------------------------------+-------+
| Ndb_api_event_data_count_injector   | 0     |
| Ndb_api_event_nondata_count_injector | 0     |
| Ndb_api_event_bytes_count_injector  | 0     |
+-------------------------------------+-------+
3 rows in set (0.00 sec)
```

The names of the status variables can easily be associated with the names of the corresponding counters. Each NDB API statistics counter is listed in the following table with a description as well as the names of any MySQL server status variables corresponding to this counter.

Table 23.67 NDB API statistics counters

Counter Name	Description	Status Variables (by statistic type): • **Session** • **Slave (replica)** • **Injector** • **Server**
`WaitExecCompleteCount`	Number of times thread has been blocked while waiting for execution of an operation to complete. Includes all `execute()` calls as well as implicit executes for blob operations and auto-increment not visible to clients.	• `Ndb_api_wait_exec_complete_count` • `Ndb_api_wait_exec_complete_count` • [none] • `Ndb_api_wait_exec_complete_count`
`WaitScanResultCount`	Number of times thread has been blocked while waiting for a scan-based signal, such waiting for additional results, or for a scan to close.	• `Ndb_api_wait_scan_result_count_se` • `Ndb_api_wait_scan_result_count_s` • [none] • `Ndb_api_wait_scan_result_count`
`WaitMetaRequestCount`	Number of times thread has been blocked waiting for a metadata-based signal; this can occur when waiting for a DDL operation or for an epoch to be started (or ended).	• `Ndb_api_wait_meta_request_count_s` • `Ndb_api_wait_meta_request_count_s` • [none] • `Ndb_api_wait_meta_request_count`

Counter Name	Description	Status Variables (by statistic type): • **Session** • **Slave (replica)** • **Injector** • **Server**
WaitNanosCount	Total time (in nanoseconds) spent waiting for some type of signal from the data nodes.	• Ndb_api_wait_nanos_count_sessi • Ndb_api_wait_nanos_count_slave • [none] • Ndb_api_wait_nanos_count
BytesSentCount	Amount of data (in bytes) sent to the data nodes	• Ndb_api_bytes_sent_count_sessi • Ndb_api_bytes_sent_count_slave • [none] • Ndb_api_bytes_sent_count
BytesRecvdCount	Amount of data (in bytes) received from the data nodes	• Ndb_api_bytes_received_count_s • Ndb_api_bytes_received_count_s • [none] • Ndb_api_bytes_received_count
TransStartCount	Number of transactions started.	• Ndb_api_trans_start_count_sess • Ndb_api_trans_start_count_slav • [none] • Ndb_api_trans_start_count
TransCommitCount	Number of transactions committed.	• Ndb_api_trans_commit_count_ses • Ndb_api_trans_commit_count_sla • [none] • Ndb_api_trans_commit_count
TransAbortCount	Number of transactions aborted.	• Ndb_api_trans_abort_count_sess • Ndb_api_trans_abort_count_slav • [none] • Ndb_api_trans_abort_count
TransCloseCount	Number of transactions aborted. (This value may be greater than the sum of TransCommitCount and TransAbortCount.)	• Ndb_api_trans_close_count_sess • Ndb_api_trans_close_count_slav • [none] • Ndb_api_trans_close_count

Counter Name	Description	Status Variables (by statistic type): • **Session** • **Slave (replica)** • **Injector** • **Server**
PkOpCount	Number of operations based on or using primary keys. This count includes blob-part table operations, implicit unlocking operations, and auto-increment operations, as well as primary key operations normally visible to MySQL clients.	• Ndb_api_pk_op_count_session • Ndb_api_pk_op_count_slave • [none] • Ndb_api_pk_op_count
UkOpCount	Number of operations based on or using unique keys.	• Ndb_api_uk_op_count_session • Ndb_api_uk_op_count_slave • [none] • Ndb_api_uk_op_count
TableScanCount	Number of table scans that have been started. This includes scans of internal tables.	• Ndb_api_table_scan_count_session • Ndb_api_table_scan_count_slave • [none] • Ndb_api_table_scan_count
RangeScanCount	Number of range scans that have been started.	• Ndb_api_range_scan_count_session • Ndb_api_range_scan_count_slave • [none] • Ndb_api_range_scan_count
PrunedScanCount	Number of scans that have been pruned to a single partition.	• Ndb_api_pruned_scan_count_session • Ndb_api_pruned_scan_count_slave • [none] • Ndb_api_pruned_scan_count
ScanBatchCount	Number of batches of rows received. (A *batch* in this context is a set of scan results from a single fragment.)	• Ndb_api_scan_batch_count_session • Ndb_api_scan_batch_count_slave • [none] • Ndb_api_scan_batch_count
ReadRowCount	Total number of rows that have been read. Includes rows read using primary key, unique key, and scan operations.	• Ndb_api_read_row_count_session • Ndb_api_read_row_count_slave • [none]

Counter Name	Description	Status Variables (by statistic type): • **Session** • **Slave (replica)** • **Injector** • **Server**
		• `Ndb_api_read_row_count`
`TransLocalReadRowCount`	Number of rows read from the data same node on which the transaction was being run.	• `Ndb_api_trans_local_read_row_c` • `Ndb_api_trans_local_read_row_c` • [none] • `Ndb_api_trans_local_read_row_c`
`DataEventsRecvdCount`	Number of row change events received.	• [none] • [none] • `Ndb_api_event_data_count_injec` • `Ndb_api_event_data_count`
`NondataEventsRecvdCount`	Number of events received, other than row change events.	• [none] • [none] • `Ndb_api_event_nondata_count_in` • `Ndb_api_event_nondata_count`
`EventBytesRecvdCount`	Number of bytes of events received.	• [none] • [none] • `Ndb_api_event_bytes_count_inje` • `Ndb_api_event_bytes_count`

To see all counts of committed transactions—that is, all `TransCommitCount` counter status variables—you can filter the results of `SHOW STATUS` for the substring `trans_commit_count`, like this:

```
mysql> SHOW STATUS LIKE '%trans_commit_count%';
+-----------------------------------+-------+
| Variable_name                     | Value |
+-----------------------------------+-------+
| Ndb_api_trans_commit_count_session | 1     |
| Ndb_api_trans_commit_count_slave   | 0     |
| Ndb_api_trans_commit_count         | 2     |
+-----------------------------------+-------+
3 rows in set (0.00 sec)
```

From this you can determine that 1 transaction has been committed in the current `mysql` client session, and 2 transactions have been committed on this `mysqld` since it was last restarted.

You can see how various NDB API counters are incremented by a given SQL statement by comparing the values of the corresponding `_session` status variables immediately before and after performing the statement. In this example, after getting the initial values from `SHOW STATUS`, we create in the `test` database an `NDB` table, named `t`, that has a single column:

```
mysql> SHOW STATUS LIKE 'ndb_api%session%';
```

```
+---------------------------------------------+--------+
| Variable_name                               | Value  |
+---------------------------------------------+--------+
| Ndb_api_wait_exec_complete_count_session    | 2      |
| Ndb_api_wait_scan_result_count_session      | 0      |
| Ndb_api_wait_meta_request_count_session     | 3      |
| Ndb_api_wait_nanos_count_session            | 820705 |
| Ndb_api_bytes_sent_count_session            | 132    |
| Ndb_api_bytes_received_count_session        | 372    |
| Ndb_api_trans_start_count_session           | 1      |
| Ndb_api_trans_commit_count_session          | 1      |
| Ndb_api_trans_abort_count_session           | 0      |
| Ndb_api_trans_close_count_session           | 1      |
| Ndb_api_pk_op_count_session                 | 1      |
| Ndb_api_uk_op_count_session                 | 0      |
| Ndb_api_table_scan_count_session            | 0      |
| Ndb_api_range_scan_count_session            | 0      |
| Ndb_api_pruned_scan_count_session           | 0      |
| Ndb_api_scan_batch_count_session            | 0      |
| Ndb_api_read_row_count_session              | 1      |
| Ndb_api_trans_local_read_row_count_session  | 1      |
+---------------------------------------------+--------+
18 rows in set (0.00 sec)

mysql> USE test;
Database changed
mysql> CREATE TABLE t (c INT) ENGINE NDBCLUSTER;
Query OK, 0 rows affected (0.85 sec)
```

Now you can execute a new `SHOW STATUS` statement and observe the changes, as shown here (with the changed rows highlighted in the output):

```
mysql> SHOW STATUS LIKE 'ndb_api%session%';
+---------------------------------------------+-----------+
| Variable_name                               | Value     |
+---------------------------------------------+-----------+
| Ndb_api_wait_exec_complete_count_session    | 8         |
| Ndb_api_wait_scan_result_count_session      | 0         |
| Ndb_api_wait_meta_request_count_session     | 17        |
| Ndb_api_wait_nanos_count_session            | 706871709 |
| Ndb_api_bytes_sent_count_session            | 2376      |
| Ndb_api_bytes_received_count_session        | 3844      |
| Ndb_api_trans_start_count_session           | 4         |
| Ndb_api_trans_commit_count_session          | 4         |
| Ndb_api_trans_abort_count_session           | 0         |
| Ndb_api_trans_close_count_session           | 4         |
| Ndb_api_pk_op_count_session                 | 6         |
| Ndb_api_uk_op_count_session                 | 0         |
| Ndb_api_table_scan_count_session            | 0         |
| Ndb_api_range_scan_count_session            | 0         |
| Ndb_api_pruned_scan_count_session           | 0         |
| Ndb_api_scan_batch_count_session            | 0         |
| Ndb_api_read_row_count_session              | 2         |
| Ndb_api_trans_local_read_row_count_session  | 1         |
+---------------------------------------------+-----------+
18 rows in set (0.00 sec)
```

Similarly, you can see the changes in the NDB API statistics counters caused by inserting a row into `t`: Insert the row, then run the same `SHOW STATUS` statement used in the previous example, as shown here:

```
mysql> INSERT INTO t VALUES (100);
Query OK, 1 row affected (0.00 sec)

mysql> SHOW STATUS LIKE 'ndb_api%session%';
+---------------------------------------------+-----------+
| Variable_name                               | Value     |
+---------------------------------------------+-----------+
| Ndb_api_wait_exec_complete_count_session    | 11        |
| Ndb_api_wait_scan_result_count_session      | 6         |
| Ndb_api_wait_meta_request_count_session     | 20        |
```

```
| Ndb_api_wait_nanos_count_session           | 707370418 |
| Ndb_api_bytes_sent_count_session           | 2724      |
| Ndb_api_bytes_received_count_session       | 4116      |
| Ndb_api_trans_start_count_session          | 7         |
| Ndb_api_trans_commit_count_session         | 6         |
| Ndb_api_trans_abort_count_session          | 0         |
| Ndb_api_trans_close_count_session          | 7         |
| Ndb_api_pk_op_count_session                | 8         |
| Ndb_api_uk_op_count_session                | 0         |
| Ndb_api_table_scan_count_session           | 1         |
| Ndb_api_range_scan_count_session           | 0         |
| Ndb_api_pruned_scan_count_session          | 0         |
| Ndb_api_scan_batch_count_session           | 0         |
| Ndb_api_read_row_count_session             | 3         |
| Ndb_api_trans_local_read_row_count_session | 2         |
+--------------------------------------------+-----------+
18 rows in set (0.00 sec)
```

We can make a number of observations from these results:

- Although we created `t` with no explicit primary key, 5 primary key operations were performed in doing so (the difference in the “before” and “after” values of `Ndb_api_pk_op_count_session`, or 6 minus 1). This reflects the creation of the hidden primary key that is a feature of all tables using the `NDB` storage engine.
- By comparing successive values for `Ndb_api_wait_nanos_count_session`, we can see that the NDB API operations implementing the `CREATE TABLE` statement waited much longer (706871709 - 820705 = 706051004 nanoseconds, or approximately 0.7 second) for responses from the data nodes than those executed by the `INSERT` (707370418 - 706871709 = 498709 ns or roughly .0005 second). The execution times reported for these statements in the `mysql` client correlate roughly with these figures.

  On platforms without sufficient (nanosecond) time resolution, small changes in the value of the `WaitNanosCount` NDB API counter due to SQL statements that execute very quickly may not always be visible in the values of `Ndb_api_wait_nanos_count_session`, `Ndb_api_wait_nanos_count_slave`, or `Ndb_api_wait_nanos_count`.
- The `INSERT` statement incremented both the `ReadRowCount` and `TransLocalReadRowCount` NDB API statistics counters, as reflected by the increased values of `Ndb_api_read_row_count_session` and `Ndb_api_trans_local_read_row_count_session`.

23.6.16 ndbinfo: The NDB Cluster Information Database

`ndbinfo` is a database containing information specific to NDB Cluster.

This database contains a number of tables, each providing a different sort of data about NDB Cluster node status, resource usage, and operations. You can find more detailed information about each of these tables in the next several sections.

`ndbinfo` is included with NDB Cluster support in the MySQL Server; no special compilation or configuration steps are required; the tables are created by the MySQL Server when it connects to the cluster. You can verify that `ndbinfo` support is active in a given MySQL Server instance using `SHOW PLUGINS`; if `ndbinfo` support is enabled, you should see a row containing `ndbinfo` in the `Name` column and `ACTIVE` in the `Status` column, as shown here (emphasized text):

```
mysql> SHOW PLUGINS;
+---------------------------------+--------+--------------------+---------+---------+
| Name                            | Status | Type               | Library | License |
+---------------------------------+--------+--------------------+---------+---------+
| binlog                          | ACTIVE | STORAGE ENGINE     | NULL    | GPL     |
| mysql_native_password           | ACTIVE | AUTHENTICATION     | NULL    | GPL     |
| sha256_password                 | ACTIVE | AUTHENTICATION     | NULL    | GPL     |
| caching_sha2_password           | ACTIVE | AUTHENTICATION     | NULL    | GPL     |
| sha2_cache_cleaner              | ACTIVE | AUDIT              | NULL    | GPL     |
| daemon_keyring_proxy_plugin     | ACTIVE | DAEMON             | NULL    | GPL     |
```

```
| CSV                                | ACTIVE | STORAGE ENGINE     | NULL    | GPL     |
| MEMORY                             | ACTIVE | STORAGE ENGINE     | NULL    | GPL     |
| InnoDB                             | ACTIVE | STORAGE ENGINE     | NULL    | GPL     |
| INNODB_TRX                         | ACTIVE | INFORMATION SCHEMA | NULL    | GPL     |
| INNODB_CMP                         | ACTIVE | INFORMATION SCHEMA | NULL    | GPL     |
| INNODB_CMP_RESET                   | ACTIVE | INFORMATION SCHEMA | NULL    | GPL     |
| INNODB_CMPMEM                      | ACTIVE | INFORMATION SCHEMA | NULL    | GPL     |
| INNODB_CMPMEM_RESET                | ACTIVE | INFORMATION SCHEMA | NULL    | GPL     |
| INNODB_CMP_PER_INDEX               | ACTIVE | INFORMATION SCHEMA | NULL    | GPL     |
| INNODB_CMP_PER_INDEX_RESET         | ACTIVE | INFORMATION SCHEMA | NULL    | GPL     |
| INNODB_BUFFER_PAGE                 | ACTIVE | INFORMATION SCHEMA | NULL    | GPL     |
| INNODB_BUFFER_PAGE_LRU             | ACTIVE | INFORMATION SCHEMA | NULL    | GPL     |
| INNODB_BUFFER_POOL_STATS           | ACTIVE | INFORMATION SCHEMA | NULL    | GPL     |
| INNODB_TEMP_TABLE_INFO             | ACTIVE | INFORMATION SCHEMA | NULL    | GPL     |
| INNODB_METRICS                     | ACTIVE | INFORMATION SCHEMA | NULL    | GPL     |
| INNODB_FT_DEFAULT_STOPWORD         | ACTIVE | INFORMATION SCHEMA | NULL    | GPL     |
| INNODB_FT_DELETED                  | ACTIVE | INFORMATION SCHEMA | NULL    | GPL     |
| INNODB_FT_BEING_DELETED            | ACTIVE | INFORMATION SCHEMA | NULL    | GPL     |
| INNODB_FT_CONFIG                   | ACTIVE | INFORMATION SCHEMA | NULL    | GPL     |
| INNODB_FT_INDEX_CACHE              | ACTIVE | INFORMATION SCHEMA | NULL    | GPL     |
| INNODB_FT_INDEX_TABLE              | ACTIVE | INFORMATION SCHEMA | NULL    | GPL     |
| INNODB_TABLES                      | ACTIVE | INFORMATION SCHEMA | NULL    | GPL     |
| INNODB_TABLESTATS                  | ACTIVE | INFORMATION SCHEMA | NULL    | GPL     |
| INNODB_INDEXES                     | ACTIVE | INFORMATION SCHEMA | NULL    | GPL     |
| INNODB_TABLESPACES                 | ACTIVE | INFORMATION SCHEMA | NULL    | GPL     |
| INNODB_COLUMNS                     | ACTIVE | INFORMATION SCHEMA | NULL    | GPL     |
| INNODB_VIRTUAL                     | ACTIVE | INFORMATION SCHEMA | NULL    | GPL     |
| INNODB_CACHED_INDEXES              | ACTIVE | INFORMATION SCHEMA | NULL    | GPL     |
| INNODB_SESSION_TEMP_TABLESPACES    | ACTIVE | INFORMATION SCHEMA | NULL    | GPL     |
| MyISAM                             | ACTIVE | STORAGE ENGINE     | NULL    | GPL     |
| MRG_MYISAM                         | ACTIVE | STORAGE ENGINE     | NULL    | GPL     |
| PERFORMANCE_SCHEMA                 | ACTIVE | STORAGE ENGINE     | NULL    | GPL     |
| TempTable                          | ACTIVE | STORAGE ENGINE     | NULL    | GPL     |
| ARCHIVE                            | ACTIVE | STORAGE ENGINE     | NULL    | GPL     |
| BLACKHOLE                          | ACTIVE | STORAGE ENGINE     | NULL    | GPL     |
| ndbcluster                         | ACTIVE | STORAGE ENGINE     | NULL    | GPL     |
| ndbinfo                            | ACTIVE | STORAGE ENGINE     | NULL    | GPL     |
| ndb_transid_mysql_connection_map   | ACTIVE | INFORMATION SCHEMA | NULL    | GPL     |
| ngram                              | ACTIVE | FTPARSER           | NULL    | GPL     |
| mysqlx_cache_cleaner               | ACTIVE | AUDIT              | NULL    | GPL     |
| mysqlx                             | ACTIVE | DAEMON             | NULL    | GPL     |
+------------------------------------+--------+--------------------+---------+---------+
47 rows in set (0.00 sec)
```

You can also do this by checking the output of `SHOW ENGINES` for a line including `ndbinfo` in the `Engine` column and `YES` in the `Support` column, as shown here (emphasized text):

```
mysql> SHOW ENGINES\G
*************************** 1. row ***************************
      Engine: ndbcluster
     Support: YES
     Comment: Clustered, fault-tolerant tables
Transactions: YES
          XA: NO
  Savepoints: NO
*************************** 2. row ***************************
      Engine: CSV
     Support: YES
     Comment: CSV storage engine
Transactions: NO
          XA: NO
  Savepoints: NO
*************************** 3. row ***************************
      Engine: InnoDB
     Support: DEFAULT
     Comment: Supports transactions, row-level locking, and foreign keys
Transactions: YES
          XA: YES
  Savepoints: YES
*************************** 4. row ***************************
      Engine: BLACKHOLE
     Support: YES
```

```
     Comment: /dev/null storage engine (anything you write to it disappears)
Transactions: NO
          XA: NO
  Savepoints: NO
*************************** 5. row ***************************
      Engine: MyISAM
     Support: YES
     Comment: MyISAM storage engine
Transactions: NO
          XA: NO
  Savepoints: NO
*************************** 6. row ***************************
      Engine: MRG_MYISAM
     Support: YES
     Comment: Collection of identical MyISAM tables
Transactions: NO
          XA: NO
  Savepoints: NO
*************************** 7. row ***************************
      Engine: ARCHIVE
     Support: YES
     Comment: Archive storage engine
Transactions: NO
          XA: NO
  Savepoints: NO
*************************** 8. row ***************************
      Engine: ndbinfo
     Support: YES
     Comment: NDB Cluster system information storage engine
Transactions: NO
          XA: NO
  Savepoints: NO
*************************** 9. row ***************************
      Engine: PERFORMANCE_SCHEMA
     Support: YES
     Comment: Performance Schema
Transactions: NO
          XA: NO
  Savepoints: NO
*************************** 10. row ***************************
      Engine: MEMORY
     Support: YES
     Comment: Hash based, stored in memory, useful for temporary tables
Transactions: NO
          XA: NO
  Savepoints: NO
10 rows in set (0.00 sec)
```

If `ndbinfo` support is enabled, then you can access `ndbinfo` using SQL statements in `mysql` or another MySQL client. For example, you can see `ndbinfo` listed in the output of `SHOW DATABASES`, as shown here (emphasized text):

```
mysql> SHOW DATABASES;
+--------------------+
| Database           |
+--------------------+
| information_schema |
| mysql              |
| ndbinfo            |
| performance_schema |
| sys                |
+--------------------+
5 rows in set (0.04 sec)
```

If the `mysqld` process was not started with the `--ndbcluster` option, `ndbinfo` is not available and is not displayed by `SHOW DATABASES`. If `mysqld` was formerly connected to an NDB Cluster but the cluster becomes unavailable (due to events such as cluster shutdown, loss of network connectivity, and so forth), `ndbinfo` and its tables remain visible, but an attempt to access any tables (other than `blocks` or `config_params`) fails with `Got error 157 'Connection to NDB failed' from NDBINFO`.

With the exception of the `blocks` and `config_params` tables, what we refer to as `ndbinfo` "tables" are actually views generated from internal `NDB` tables not normally visible to the MySQL Server. You can make these tables visible by setting the `ndbinfo_show_hidden` system variable to `ON` (or `1`), but this is normally not necessary.

All `ndbinfo` tables are read-only, and are generated on demand when queried. Because many of them are generated in parallel by the data nodes while other are specific to a given SQL node, they are not guaranteed to provide a consistent snapshot.

In addition, pushing down of joins is not supported on `ndbinfo` tables; so joining large `ndbinfo` tables can require transfer of a large amount of data to the requesting API node, even when the query makes use of a `WHERE` clause.

`ndbinfo` tables are not included in the query cache. (Bug #59831)

You can select the `ndbinfo` database with a `USE` statement, and then issue a `SHOW TABLES` statement to obtain a list of tables, just as for any other database, like this:

```
mysql> USE ndbinfo;
Database changed

mysql> SHOW TABLES;
+---------------------------------+
| Tables_in_ndbinfo               |
+---------------------------------+
| arbitrator_validity_detail      |
| arbitrator_validity_summary     |
| backup_id                       |
| blobs                           |
| blocks                          |
| cluster_locks                   |
| cluster_operations              |
| cluster_transactions            |
| config_nodes                    |
| config_params                   |
| config_values                   |
| counters                        |
| cpudata                         |
| cpudata_1sec                    |
| cpudata_20sec                   |
| cpudata_50ms                    |
| cpuinfo                         |
| cpustat                         |
| cpustat_1sec                    |
| cpustat_20sec                   |
| cpustat_50ms                    |
| dict_obj_info                   |
| dict_obj_tree                   |
| dict_obj_types                  |
| dictionary_columns              |
| dictionary_tables               |
| disk_write_speed_aggregate      |
| disk_write_speed_aggregate_node |
| disk_write_speed_base           |
| diskpagebuffer                  |
| diskstat                        |
| diskstats_1sec                  |
| error_messages                  |
| events                          |
| files                           |
| foreign_keys                    |
| hash_maps                       |
| hwinfo                          |
| index_columns                   |
| index_stats                     |
| locks_per_fragment              |
| logbuffers                      |
| logspaces                       |
| membership                      |
```

```
| memory_per_fragment             |
| memoryusage                     |
| nodes                           |
| operations_per_fragment         |
| pgman_time_track_stats          |
| processes                       |
| resources                       |
| restart_info                    |
| server_locks                    |
| server_operations               |
| server_transactions             |
| table_distribution_status       |
| table_fragments                 |
| table_info                      |
| table_replicas                  |
| tc_time_track_stats             |
| threadblocks                    |
| threads                         |
| threadstat                      |
| transporters                    |
+---------------------------------+
64 rows in set (0.00 sec)
```

In NDB 8.0, all `ndbinfo` tables use the `NDB` storage engine; however, an `ndbinfo` entry still appears in the output of `SHOW ENGINES` and `SHOW PLUGINS` as described previously.

You can execute `SELECT` statements against these tables, just as you would normally expect:

```
mysql> SELECT * FROM memoryusage;
+---------+--------------------+--------+------------+------------+-------------+
| node_id | memory_type        | used   | used_pages | total      | total_pages |
+---------+--------------------+--------+------------+------------+-------------+
|       5 | Data memory        | 425984 |         13 | 2147483648 |       65536 |
|       5 | Long message buffer | 393216 |       1536 |   67108864 |      262144 |
|       6 | Data memory        | 425984 |         13 | 2147483648 |       65536 |
|       6 | Long message buffer | 393216 |       1536 |   67108864 |      262144 |
|       7 | Data memory        | 425984 |         13 | 2147483648 |       65536 |
|       7 | Long message buffer | 393216 |       1536 |   67108864 |      262144 |
|       8 | Data memory        | 425984 |         13 | 2147483648 |       65536 |
|       8 | Long message buffer | 393216 |       1536 |   67108864 |      262144 |
+---------+--------------------+--------+------------+------------+-------------+
8 rows in set (0.09 sec)
```

More complex queries, such as the two following `SELECT` statements using the `memoryusage` table, are possible:

```
mysql> SELECT SUM(used) as 'Data Memory Used, All Nodes'
     >     FROM memoryusage
     >     WHERE memory_type = 'Data memory';
+-----------------------------+
| Data Memory Used, All Nodes |
+-----------------------------+
|                        6460 |
+-----------------------------+
1 row in set (0.09 sec)

mysql> SELECT SUM(used) as 'Long Message Buffer, All Nodes'
     >     FROM memoryusage
     >     WHERE memory_type = 'Long message buffer';
+-----------------------------------+
| Long Message Buffer Used, All Nodes |
+-----------------------------------+
|                             1179648 |
+-----------------------------------+
1 row in set (0.08 sec)
```

`ndbinfo` table and column names are case-sensitive (as is the name of the `ndbinfo` database itself). These identifiers are in lowercase. Trying to use the wrong lettercase results in an error, as shown in this example:

```
mysql> SELECT * FROM nodes;
```

```
+---------+--------+---------+-------------+-------------------+
| node_id | uptime | status  | start_phase | config_generation |
+---------+--------+---------+-------------+-------------------+
|       5 |  17707 | STARTED |           0 |                 1 |
|       6 |  17706 | STARTED |           0 |                 1 |
|       7 |  17705 | STARTED |           0 |                 1 |
|       8 |  17704 | STARTED |           0 |                 1 |
+---------+--------+---------+-------------+-------------------+
4 rows in set (0.06 sec)

mysql> SELECT * FROM Nodes;
ERROR 1146 (42S02): Table 'ndbinfo.Nodes' doesn't exist
```

`mysqldump` ignores the `ndbinfo` database entirely, and excludes it from any output. This is true even when using the `--databases` or `--all-databases` option.

NDB Cluster also maintains tables in the `INFORMATION_SCHEMA` information database, including the `FILES` table which contains information about files used for NDB Cluster Disk Data storage, and the `ndb_transid_mysql_connection_map` table, which shows the relationships between transactions, transaction coordinators, and NDB Cluster API nodes. For more information, see the descriptions of the tables or Section 23.6.17, “INFORMATION_SCHEMA Tables for NDB Cluster”.

23.6.16.1 The ndbinfo arbitrator_validity_detail Table

The `arbitrator_validity_detail` table shows the view that each data node in the cluster has of the arbitrator. It is a subset of the `membership` table.

The `arbitrator_validity_detail` table contains the following columns:

- `node_id`

  This node's node ID

- `arbitrator`

  Node ID of arbitrator

- `arb_ticket`

  Internal identifier used to track arbitration

- `arb_connected`

  Whether this node is connected to the arbitrator; either of `Yes` or `No`

- `arb_state`

  Arbitration state

Notes

The node ID is the same as that reported by `ndb_mgm -e "SHOW"`.

All nodes should show the same `arbitrator` and `arb_ticket` values as well as the same `arb_state` value. Possible `arb_state` values are `ARBIT_NULL`, `ARBIT_INIT`, `ARBIT_FIND`, `ARBIT_PREP1`, `ARBIT_PREP2`, `ARBIT_START`, `ARBIT_RUN`, `ARBIT_CHOOSE`, `ARBIT_CRASH`, and `UNKNOWN`.

`arb_connected` shows whether the current node is connected to the `arbitrator`.

23.6.16.2 The ndbinfo arbitrator_validity_summary Table

The `arbitrator_validity_summary` table provides a composite view of the arbitrator with regard to the cluster's data nodes.

The `arbitrator_validity_summary` table contains the following columns:

- `arbitrator`

  Node ID of arbitrator

- `arb_ticket`

  Internal identifier used to track arbitration

- `arb_connected`

  Whether this arbitrator is connected to the cluster

- `consensus_count`

  Number of data nodes that see this node as arbitrator; either of `Yes` or `No`

Notes

In normal operations, this table should have only 1 row for any appreciable length of time. If it has more than 1 row for longer than a few moments, then either not all nodes are connected to the arbitrator, or all nodes are connected, but do not agree on the same arbitrator.

The `arbitrator` column shows the arbitrator's node ID.

`arb_ticket` is the internal identifier used by this arbitrator.

`arb_connected` shows whether this node is connected to the cluster as an arbitrator.

23.6.16.3 The ndbinfo backup_id Table

This table provides a way to find the ID of the backup started most recently for this cluster.

The `backup_id` table contains a single column `id`, which corresponds to a backup ID taken using the `ndb_mgm` client `START BACKUP` command. This table contains a single row.

Example: Assume the following sequence of `START BACKUP` commands issued in the NDB management client, with no other backups taken since the cluster was first started:

```
ndb_mgm> START BACKUP
Waiting for completed, this may take several minutes
Node 5: Backup 1 started from node 50
Node 5: Backup 1 started from node 50 completed
 StartGCP: 27894 StopGCP: 27897
 #Records: 2057 #LogRecords: 0
 Data: 51580 bytes Log: 0 bytes
ndb_mgm> START BACKUP 5
Waiting for completed, this may take several minutes
Node 5: Backup 5 started from node 50
Node 5: Backup 5 started from node 50 completed
 StartGCP: 27905 StopGCP: 27908
 #Records: 2057 #LogRecords: 0
 Data: 51580 bytes Log: 0 bytes
ndb_mgm> START BACKUP
Waiting for completed, this may take several minutes
Node 5: Backup 6 started from node 50
Node 5: Backup 6 started from node 50 completed
 StartGCP: 27912 StopGCP: 27915
 #Records: 2057 #LogRecords: 0
 Data: 51580 bytes Log: 0 bytes
ndb_mgm> START BACKUP 3
Connected to Management Server at: localhost:1186
Waiting for completed, this may take several minutes
Node 5: Backup 3 started from node 50
Node 5: Backup 3 started from node 50 completed
 StartGCP: 28149 StopGCP: 28152
```

```
 #Records: 2057 #LogRecords: 0
 Data: 51580 bytes Log: 0 bytes
ndb_mgm>
```

After this, the `backup_id` table contains the single row shown here, using the `mysql` client:

```
mysql> USE ndbinfo;

Database changed
mysql> SELECT * FROM backup_id;
+------+
| id   |
+------+
|    3 |
+------+
1 row in set (0.00 sec)
```

If no backups can be found, the table contains a single row with `0` as the `id` value.

The `backup_id` table was added in NDB 8.0.24.

23.6.16.4 The ndbinfo blobs Table

This table provides about blob values stored in `NDB`. The `blobs` table has the columns listed here:

- `table_id`

  Unique ID of the table containing the column

- `database_name`

  Name of the database in which this table resides

- `table_name`

  Name of the table

- `column_id`

  The column's unique ID within the table

- `column_name`

  Name of the column

- `inline_size`

  Inline size of the column

- `part_size`

  Part size of the column

- `stripe_size`

  Stripe size of the column

- `blob_table_name`

  Name of the blob table containing this column's blob data, if any

Rows exist in this table for those `NDB` table columns that store `BLOB`, `TEXT` values taking up more than 255 bytes and thus require the use of a blob table. Parts of `JSON` values exceeding 4000 bytes in size are also stored in this table. For more information about how NDB Cluster stores columns of such types, see String Type Storage Requirements.

The part and (NDB 8.0.30 and later) inline sizes of `NDB` blob columns can be set using `CREATE TABLE` and `ALTER TABLE` statements containing `NDB` table column comments (see NDB_COLUMN Options); this can also be done in NDB API applications (see `Column::setPartSize()` and `setInlineSize()`).

The `blobs` table was added in NDB 8.0.29.

23.6.16.5 The ndbinfo blocks Table

The `blocks` table is a static table which simply contains the names and internal IDs of all NDB kernel blocks (see NDB Kernel Blocks). It is for use by the other `ndbinfo` tables (most of which are actually views) in mapping block numbers to block names for producing human-readable output.

The `blocks` table contains the following columns:

- `block_number`

  Block number

- `block_name`

  Block name

Notes

To obtain a list of all block names, simply execute `SELECT block_name FROM ndbinfo.blocks`. Although this is a static table, its content can vary between different NDB Cluster releases.

23.6.16.6 The ndbinfo cluster_locks Table

The `cluster_locks` table provides information about current lock requests holding and waiting for locks on `NDB` tables in an NDB Cluster, and is intended as a companion table to `cluster_operations`. Information obtain from the `cluster_locks` table may be useful in investigating stalls and deadlocks.

The `cluster_locks` table contains the following columns:

- `node_id`

  ID of reporting node

- `block_instance`

  ID of reporting LDM instance

- `tableid`

  ID of table containing this row

- `fragmentid`

  ID of fragment containing locked row

- `rowid`

  ID of locked row

- `transid`

  Transaction ID

- `mode`

  Lock request mode

- `state`

  Lock state

- `detail`

  Whether this is first holding lock in row lock queue

- `op`

  Operation type

- `duration_millis`

  Milliseconds spent waiting or holding lock

- `lock_num`

  ID of lock object

- `waiting_for`

  Waiting for lock with this ID

Notes

The table ID (`tableid` column) is assigned internally, and is the same as that used in other `ndbinfo` tables. It is also shown in the output of `ndb_show_tables`.

The transaction ID (`transid` column) is the identifier generated by the NDB API for the transaction requesting or holding the current lock.

The `mode` column shows the lock mode; this is always one of `S` (indicating a shared lock) or `X` (an exclusive lock). If a transaction holds an exclusive lock on a given row, all other locks on that row have the same transaction ID.

The `state` column shows the lock state. Its value is always one of `H` (holding) or `W` (waiting). A waiting lock request waits for a lock held by a different transaction.

When the `detail` column contains a `*` (asterisk character), this means that this lock is the first holding lock in the affected row's lock queue; otherwise, this column is empty. This information can be used to help identify the unique entries in a list of lock requests.

The `op` column shows the type of operation requesting the lock. This is always one of the values `READ`, `INSERT`, `UPDATE`, `DELETE`, `SCAN`, or `REFRESH`.

The `duration_millis` column shows the number of milliseconds for which this lock request has been waiting or holding the lock. This is reset to 0 when a lock is granted for a waiting request.

The lock ID (`lockid` column) is unique to this node and block instance.

The lock state is shown in the `lock_state` column; if this is `W`, the lock is waiting to be granted, and the `waiting_for` column shows the lock ID of the lock object this request is waiting for. Otherwise, the `waiting_for` column is empty. `waiting_for` can refer only to locks on the same row, as identified by `node_id`, `block_instance`, `tableid`, `fragmentid`, and `rowid`.

23.6.16.7 The ndbinfo cluster_operations Table

The `cluster_operations` table provides a per-operation (stateful primary key op) view of all activity in the NDB Cluster from the point of view of the local data management (LQH) blocks (see The DBLQH Block).

The `cluster_operations` table contains the following columns:

- `node_id`

  Node ID of reporting LQH block

- `block_instance`

  LQH block instance

- `transid`

  Transaction ID

- `operation_type`

  Operation type (see text for possible values)

- `state`

  Operation state (see text for possible values)

- `tableid`

  Table ID

- `fragmentid`

  Fragment ID

- `client_node_id`

  Client node ID

- `client_block_ref`

  Client block reference

- `tc_node_id`

  Transaction coordinator node ID

- `tc_block_no`

  Transaction coordinator block number

- `tc_block_instance`

  Transaction coordinator block instance

Notes

The transaction ID is a unique 64-bit number which can be obtained using the NDB API's `getTransactionId()` method. (Currently, the MySQL Server does not expose the NDB API transaction ID of an ongoing transaction.)

The `operation_type` column can take any one of the values `READ`, `READ-SH`, `READ-EX`, `INSERT`, `UPDATE`, `DELETE`, `WRITE`, `UNLOCK`, `REFRESH`, `SCAN`, `SCAN-SH`, `SCAN-EX`, or `<unknown>`.

The `state` column can have any one of the values `ABORT_QUEUED`, `ABORT_STOPPED`, `COMMITTED`, `COMMIT_QUEUED`, `COMMIT_STOPPED`, `COPY_CLOSE_STOPPED`, `COPY_FIRST_STOPPED`, `COPY_STOPPED`, `COPY_TUPKEY`, `IDLE`, `LOG_ABORT_QUEUED`, `LOG_COMMIT_QUEUED`, `LOG_COMMIT_QUEUED_WAIT_SIGNAL`, `LOG_COMMIT_WRITTEN`, `LOG_COMMIT_WRITTEN_WAIT_SIGNAL`, `LOG_QUEUED`, `PREPARED`, `PREPARED_RECEIVED_COMMIT`, `SCAN_CHECK_STOPPED`, `SCAN_CLOSE_STOPPED`, `SCAN_FIRST_STOPPED`,

SCAN_RELEASE_STOPPED, SCAN_STATE_USED, SCAN_STOPPED, SCAN_TUPKEY, STOPPED, TC_NOT_CONNECTED, WAIT_ACC, WAIT_ACC_ABORT, WAIT_AI_AFTER_ABORT, WAIT_ATTR, WAIT_SCAN_AI, WAIT_TUP, WAIT_TUPKEYINFO, WAIT_TUP_COMMIT, or WAIT_TUP_TO_ABORT. (If the MySQL Server is running with ndbinfo_show_hidden enabled, you can view this list of states by selecting from the ndb$dblqh_tcconnect_state table, which is normally hidden.)

You can obtain the name of an NDB table from its table ID by checking the output of ndb_show_tables.

The fragid is the same as the partition number seen in the output of ndb_desc --extra-partition-info (short form -p).

In client_node_id and client_block_ref, client refers to an NDB Cluster API or SQL node (that is, an NDB API client or a MySQL Server attached to the cluster).

The block_instance and tc_block_instance column provide, respectively, the DBLQH and DBTC block instance numbers. You can use these along with the block names to obtain information about specific threads from the threadblocks table.

23.6.16.8 The ndbinfo cluster_transactions Table

The cluster_transactions table shows information about all ongoing transactions in an NDB Cluster.

The cluster_transactions table contains the following columns:

- node_id

  Node ID of transaction coordinator

- block_instance

  TC block instance

- transid

  Transaction ID

- state

  Operation state (see text for possible values)

- count_operations

  Number of stateful primary key operations in transaction (includes reads with locks, as well as DML operations)

- outstanding_operations

  Operations still being executed in local data management blocks

- inactive_seconds

  Time spent waiting for API

- client_node_id

  Client node ID

- client_block_ref

  Client block reference

Notes

The transaction ID is a unique 64-bit number which can be obtained using the NDB API's `getTransactionId()` method. (Currently, the MySQL Server does not expose the NDB API transaction ID of an ongoing transaction.)

`block_instance` refers to an instance of a kernel block. Together with the block name, this number can be used to look up a given instance in the `threadblocks` table.

The `state` column can have any one of the values `CS_ABORTING`, `CS_COMMITTING`, `CS_COMMIT_SENT`, `CS_COMPLETE_SENT`, `CS_COMPLETING`, `CS_CONNECTED`, `CS_DISCONNECTED`, `CS_FAIL_ABORTED`, `CS_FAIL_ABORTING`, `CS_FAIL_COMMITTED`, `CS_FAIL_COMMITTING`, `CS_FAIL_COMPLETED`, `CS_FAIL_PREPARED`, `CS_PREPARE_TO_COMMIT`, `CS_RECEIVING`, `CS_REC_COMMITTING`, `CS_RESTART`, `CS_SEND_FIRE_TRIG_REQ`, `CS_STARTED`, `CS_START_COMMITTING`, `CS_START_SCAN`, `CS_WAIT_ABORT_CONF`, `CS_WAIT_COMMIT_CONF`, `CS_WAIT_COMPLETE_CONF`, `CS_WAIT_FIRE_TRIG_REQ`. (If the MySQL Server is running with `ndbinfo_show_hidden` enabled, you can view this list of states by selecting from the `ndb$dbtc_apiconnect_state` table, which is normally hidden.)

In `client_node_id` and `client_block_ref`, `client` refers to an NDB Cluster API or SQL node (that is, an NDB API client or a MySQL Server attached to the cluster).

The `tc_block_instance` column provides the `DBTC` block instance number. You can use this along with the block name to obtain information about specific threads from the `threadblocks` table.

23.6.16.9 The ndbinfo config_nodes Table

The `config_nodes` table shows nodes configured in an NDB Cluster `config.ini` file. For each node, the table displays a row containing the node ID, the type of node (management node, data node, or API node), and the name or IP address of the host on which the node is configured to run.

This table does not indicate whether a given node is actually running, or whether it is currently connected to the cluster. Information about nodes connected to an NDB Cluster can be obtained from the `nodes` and `processes` table.

The `config_nodes` table contains the following columns:

- `node_id`

  The node's ID

- `node_type`

  The type of node

- `node_hostname`

  The name or IP address of the host on which the node resides

Notes

The `node_id` column shows the node ID used in the `config.ini` file for this node; if none is specified, the node ID that would be assigned automatically to this node is displayed.

The `node_type` column displays one of the following three values:

- `MGM`: Management node.
- `NDB`: Data node.
- `API`: API node; this includes SQL nodes.

The `node_hostname` column shows the node host as specified in the `config.ini` file. This can be empty for an API node, if `HostName` has not been set in the cluster configuration file. If `HostName` has not been set for a data node in the configuration file, `localhost` is used here. `localhost` is also used if `HostName` has not been specified for a management node.

23.6.16.10 The ndbinfo config_params Table

The `config_params` table is a static table which provides the names and internal ID numbers of and other information about NDB Cluster configuration parameters. This table can also be used in conjunction with the `config_values` table for obtaining realtime information about node configuration parameters.

The `config_params` table contains the following columns:

- `param_number`

  The parameter's internal ID number

- `param_name`

  The name of the parameter

- `param_description`

  A brief description of the parameter

- `param_type`

  The parameter's data type

- `param_default`

  The parameter's default value, if any

- `param_min`

  The parameter's maximum value, if any

- `param_max`

  The parameter's minimum value, if any

- `param_mandatory`

  This is 1 if the parameter is required, otherwise 0

- `param_status`

  Currently unused

Notes

This table is read-only.

Although this is a static table, its content can vary between NDB Cluster installations, since supported parameters can vary due to differences between software releases, cluster hardware configurations, and other factors.

23.6.16.11 The ndbinfo config_values Table

The `config_values` table provides information about the current state of node configuration parameter values. Each row in the table corresponds to the current value of a parameter on a given node.

The `config_values` table contains the following columns:

- `node_id`

  ID of the node in the cluster

- `config_param`

  The parameter's internal ID number

- `config_value`

  Current value of the parameter

Notes

This table's `config_param` column and the `config_params` table's `param_number` column use the same parameter identifiers. By joining the two tables on these columns, you can obtain detailed information about desired node configuration parameters. The query shown here provides the current values for all parameters on each data node in the cluster, ordered by node ID and parameter name:

```
SELECT    v.node_id AS 'Node Id',
          p.param_name AS 'Parameter',
          v.config_value AS 'Value'
FROM      config_values v
JOIN      config_params p
ON        v.config_param=p.param_number
WHERE     p.param_name NOT LIKE '\_\_%'
ORDER BY  v.node_id, p.param_name;
```

Partial output from the previous query when run on a small example cluster used for simple testing:

```
+---------+----------------------------------------+----------------+
| Node Id | Parameter                              | Value          |
+---------+----------------------------------------+----------------+
|       2 | Arbitration                            | 1              |
|       2 | ArbitrationTimeout                     | 7500           |
|       2 | BackupDataBufferSize                   | 16777216       |
|       2 | BackupDataDir                          | /home/jon/data |
|       2 | BackupDiskWriteSpeedPct                | 50             |
|       2 | BackupLogBufferSize                    | 16777216       |

...

|       3 | TotalSendBufferMemory                  | 0              |
|       3 | TransactionBufferMemory                | 1048576        |
|       3 | TransactionDeadlockDetectionTimeout    | 1200           |
|       3 | TransactionInactiveTimeout             | 4294967039     |
|       3 | TwoPassInitialNodeRestartCopy          | 0              |
|       3 | UndoDataBuffer                         | 16777216       |
|       3 | UndoIndexBuffer                        | 2097152        |
+---------+----------------------------------------+----------------+
248 rows in set (0.02 sec)
```

The `WHERE` clause filters out parameters whose names begin with a double underscore (`__`); these parameters are reserved for testing and other internal uses by the NDB developers, and are not intended for use in a production NDB Cluster.

You can obtain output that is more specific, more detailed, or both by issuing the proper queries. This example provides all types of available information about the `NodeId`, `NoOfReplicas`, `HostName`, `DataMemory`, `IndexMemory`, and `TotalSendBufferMemory` parameters as currently set for all data nodes in the cluster:

```
SELECT  p.param_name AS Name,
        v.node_id AS Node,
        p.param_type AS Type,
```

```
        p.param_default AS 'Default',
        p.param_min AS Minimum,
        p.param_max AS Maximum,
        CASE p.param_mandatory WHEN 1 THEN 'Y' ELSE 'N' END AS 'Required',
        v.config_value AS Current
FROM    config_params p
JOIN    config_values v
ON      p.param_number = v.config_param
WHERE   p. param_name
  IN ('NodeId', 'NoOfReplicas', 'HostName',
      'DataMemory', 'IndexMemory', 'TotalSendBufferMemory')\G
```

The output from this query when run on a small NDB Cluster with 2 data nodes used for simple testing is shown here:

```
*************************** 1. row ***************************
    Name: NodeId
    Node: 2
    Type: unsigned
 Default:
 Minimum: 1
 Maximum: 144
Required: Y
 Current: 2
*************************** 2. row ***************************
    Name: HostName
    Node: 2
    Type: string
 Default: localhost
 Minimum:
 Maximum:
Required: N
 Current: 127.0.0.1
*************************** 3. row ***************************
    Name: TotalSendBufferMemory
    Node: 2
    Type: unsigned
 Default: 0
 Minimum: 262144
 Maximum: 4294967039
Required: N
 Current: 0
*************************** 4. row ***************************
    Name: NoOfReplicas
    Node: 2
    Type: unsigned
 Default: 2
 Minimum: 1
 Maximum: 4
Required: N
 Current: 2
*************************** 5. row ***************************
    Name: DataMemory
    Node: 2
    Type: unsigned
 Default: 102760448
 Minimum: 1048576
 Maximum: 1099511627776
Required: N
 Current: 524288000
*************************** 6. row ***************************
    Name: NodeId
    Node: 3
    Type: unsigned
 Default:
 Minimum: 1
 Maximum: 144
Required: Y
 Current: 3
*************************** 7. row ***************************
    Name: HostName
    Node: 3
```

```
    Type: string
 Default: localhost
 Minimum:
 Maximum:
Required: N
 Current: 127.0.0.1
*************************** 8. row ***************************
    Name: TotalSendBufferMemory
    Node: 3
    Type: unsigned
 Default: 0
 Minimum: 262144
 Maximum: 4294967039
Required: N
 Current: 0
*************************** 9. row ***************************
    Name: NoOfReplicas
    Node: 3
    Type: unsigned
 Default: 2
 Minimum: 1
 Maximum: 4
Required: N
 Current: 2
*************************** 10. row ***************************
    Name: DataMemory
    Node: 3
    Type: unsigned
 Default: 102760448
 Minimum: 1048576
 Maximum: 1099511627776
Required: N
 Current: 524288000
10 rows in set (0.01 sec)
```

23.6.16.12 The ndbinfo counters Table

The `counters` table provides running totals of events such as reads and writes for specific kernel blocks and data nodes. Counts are kept from the most recent node start or restart; a node start or restart resets all counters on that node. Not all kernel blocks have all types of counters.

The `counters` table contains the following columns:

- `node_id`

  The data node ID

- `block_name`

  Name of the associated NDB kernel block (see NDB Kernel Blocks).

- `block_instance`

  Block instance

- `counter_id`

  The counter's internal ID number; normally an integer between 1 and 10, inclusive.

- `counter_name`

  The name of the counter. See text for names of individual counters and the NDB kernel block with which each counter is associated.

- `val`

  The counter's value

Notes

Each counter is associated with a particular NDB kernel block.

The `OPERATIONS` counter is associated with the `DBLQH` (local query handler) kernel block. A primary-key read counts as one operation, as does a primary-key update. For reads, there is one operation in `DBLQH` per operation in `DBTC`. For writes, there is one operation counted per fragment replica.

The `ATTRINFO`, `TRANSACTIONS`, `COMMITS`, `READS`, `LOCAL_READS`, `SIMPLE_READS`, `WRITES`, `LOCAL_WRITES`, `ABORTS`, `TABLE_SCANS`, and `RANGE_SCANS` counters are associated with the `DBTC` (transaction co-ordinator) kernel block.

`LOCAL_WRITES` and `LOCAL_READS` are primary-key operations using a transaction coordinator in a node that also holds the primary fragment replica of the record.

The `READS` counter includes all reads. `LOCAL_READS` includes only those reads of the primary fragment replica on the same node as this transaction coordinator. `SIMPLE_READS` includes only those reads in which the read operation is the beginning and ending operation for a given transaction. Simple reads do not hold locks but are part of a transaction, in that they observe uncommitted changes made by the transaction containing them but not of any other uncommitted transactions. Such reads are “simple” from the point of view of the TC block; since they hold no locks they are not durable, and once `DBTC` has routed them to the relevant LQH block, it holds no state for them.

`ATTRINFO` keeps a count of the number of times an interpreted program is sent to the data node. See NDB Protocol Messages, for more information about `ATTRINFO` messages in the `NDB` kernel.

The `LOCAL_TABLE_SCANS_SENT`, `READS_RECEIVED`, `PRUNED_RANGE_SCANS_RECEIVED`, `RANGE_SCANS_RECEIVED`, `LOCAL_READS_SENT`, `CONST_PRUNED_RANGE_SCANS_RECEIVED`, `LOCAL_RANGE_SCANS_SENT`, `REMOTE_READS_SENT`, `REMOTE_RANGE_SCANS_SENT`, `READS_NOT_FOUND`, `SCAN_BATCHES_RETURNED`, `TABLE_SCANS_RECEIVED`, and `SCAN_ROWS_RETURNED` counters are associated with the `DBSPJ` (select push-down join) kernel block.

The `block_name` and `block_instance` columns provide, respectively, the applicable NDB kernel block name and instance number. You can use these to obtain information about specific threads from the `threadblocks` table.

A number of counters provide information about transporter overload and send buffer sizing when troubleshooting such issues. For each LQH instance, there is one instance of each counter in the following list:

- `LQHKEY_OVERLOAD`: Number of primary key requests rejected at the LQH block instance due to transporter overload
- `LQHKEY_OVERLOAD_TC`: Count of instances of `LQHKEY_OVERLOAD` where the TC node transporter was overloaded
- `LQHKEY_OVERLOAD_READER`: Count of instances of `LQHKEY_OVERLOAD` where the API reader (reads only) node was overloaded.
- `LQHKEY_OVERLOAD_NODE_PEER`: Count of instances of `LQHKEY_OVERLOAD` where the next backup data node (writes only) was overloaded
- `LQHKEY_OVERLOAD_SUBSCRIBER`: Count of instances of `LQHKEY_OVERLOAD` where a event subscriber (writes only) was overloaded.
- `LQHSCAN_SLOWDOWNS`: Count of instances where a fragment scan batch size was reduced due to scanning API transporter overload.

23.6.16.13 The ndbinfo cpudata Table

The `cpudata` table provides data about CPU usage during the last second.

The `cpustat` table contains the following columns:

- `node_id`

  Node ID

- `cpu_no`

  CPU ID

- `cpu_online`

  1 if the CPU is currently online, otherwise 0

- `cpu_userspace_time`

  CPU time spent in userspace

- `cpu_idle_time`

  CPU time spent idle

- `cpu_system_time`

  CPU time spent in system time

- `cpu_interrupt_time`

  CPU time spent handling interrupts (hardware and software)

- `cpu_exec_vm_time`

  CPU time spent in virtual machine execution

Notes

The `cpudata` table is available only on Linux and Solaris operating systems.

This table was added in NDB 8.0.23.

23.6.16.14 The ndbinfo cpudata_1sec Table

The `cpudata_1sec` table provides data about CPU usage per second over the last 20 seconds.

The `cpustat` table contains the following columns:

- `node_id`

  Node ID

- `measurement_id`

  Measurement sequence ID; later measurements have lower IDs

- `cpu_no`

  CPU ID

- `cpu_online`

  1 if the CPU is currently online, otherwise 0

- `cpu_userspace_time`

  CPU time spent in userspace

- `cpu_idle_time`

  CPU time spent idle

- `cpu_system_time`

  CPU time spent in system time

- `cpu_interrupt_time`

  CPU time spent handling interrupts (hardware and software)

- `cpu_exec_vm_time`

  CPU time spent in virtual machine execution

- `elapsed_time`

  Time in microseconds used for this measurement

Notes

The `cpudata_1sec` table is available only on Linux and Solaris operating systems.

This table was added in NDB 8.0.23.

23.6.16.15 The ndbinfo cpudata_20sec Table

The `cpudata_20sec` table provides data about CPU usage per 20-second interval over the last 400 seconds.

The `cpustat` table contains the following columns:

- `node_id`

  Node ID

- `measurement_id`

  Measurement sequence ID; later measurements have lower IDs

- `cpu_no`

  CPU ID

- `cpu_online`

  1 if the CPU is currently online, otherwise 0

- `cpu_userspace_time`

  CPU time spent in userspace

- `cpu_idle_time`

  CPU time spent idle

- `cpu_system_time`

  CPU time spent in system time

- `cpu_interrupt_time`

  CPU time spent handling interrupts (hardware and software)

- `cpu_exec_vm_time`

  CPU time spent in virtual machine execution

- `elapsed_time`

  Time in microseconds used for this measurement

Notes

The `cpudata_20sec` table is available only on Linux and Solaris operating systems.

This table was added in NDB 8.0.23.

23.6.16.16 The ndbinfo cpudata_50ms Table

The `cpudata_50ms` table provides data about CPU usage per 50-millisecond interval over the last second.

The `cpustat` table contains the following columns:

- `node_id`

  Node ID

- `measurement_id`

  Measurement sequence ID; later measurements have lower IDs

- `cpu_no`

  CPU ID

- `cpu_online`

  1 if the CPU is currently online, otherwise 0

- `cpu_userspace_time`

  CPU time spent in userspace

- `cpu_idle_time`

  CPU time spent idle

- `cpu_system_time`

  CPU time spent in system time

- `cpu_interrupt_time`

  CPU time spent handling interrupts (hardware and software)

- `cpu_exec_vm_time`

  CPU time spent in virtual machine execution

- `elapsed_time`

  Time in microseconds used for this measurement

Notes

The `cpudata_50ms` table is available only on Linux and Solaris operating systems.

This table was added in NDB 8.0.23.

23.6.16.17 The ndbinfo cpuinfo Table

The `cpuinfo` table provides information about the CPU on which a given data node executes.

The `cpuinfo` table contains the following columns:

- `node_id`

  Node ID

- `cpu_no`

  CPU ID

- `cpu_online`

  1 if the CPU is online, otherwise 0

- `core_id`

  CPU core ID

- `socket_id`

  CPU socket ID

Notes

The `cpuinfo` table is available on all operating systems supported by `NDB`, with the exception of MacOS and FreeBSD.

This table was added in NDB 8.0.23.

23.6.16.18 The ndbinfo cpustat Table

The `cpustat` table provides per-thread CPU statistics gathered each second, for each thread running in the `NDB` kernel.

The `cpustat` table contains the following columns:

- `node_id`

  ID of the node where the thread is running

- `thr_no`

  Thread ID (specific to this node)

- `OS_user`

  OS user time

- `OS_system`

  OS system time

- `OS_idle`

  OS idle time

- `thread_exec`

  Thread execution time

- `thread_sleeping`

  Thread sleep time

- `thread_spinning`

  Thread spin time

- `thread_send`

  Thread send time

- `thread_buffer_full`

  Thread buffer full time

- `elapsed_time`

  Elapsed time

23.6.16.19 The ndbinfo cpustat_50ms Table

The `cpustat_50ms` table provides raw, per-thread CPU data obtained each 50 milliseconds for each thread running in the `NDB` kernel.

Like `cpustat_1sec` and `cpustat_20sec`, this table shows 20 measurement sets per thread, each referencing a period of the named duration. Thus, `cpsustat_50ms` provides 1 second of history.

The `cpustat_50ms` table contains the following columns:

- `node_id`

  ID of the node where the thread is running

- `thr_no`

  Thread ID (specific to this node)

- `OS_user_time`

  OS user time

- `OS_system_time`

  OS system time

- `OS_idle_time`

  OS idle time

- `exec_time`

  Thread execution time

- `sleep_time`

  Thread sleep time

- `spin_time`

  Thread spin time

- `send_time`

Thread send time

- `buffer_full_time`

  Thread buffer full time

- `elapsed_time`

  Elapsed time

23.6.16.20 The ndbinfo cpustat_1sec Table

The `cpustat-1sec` table provides raw, per-thread CPU data obtained each second for each thread running in the `NDB` kernel.

Like `cpustat_50ms` and `cpustat_20sec`, this table shows 20 measurement sets per thread, each referencing a period of the named duration. Thus, `cpsustat_1sec` provides 20 seconds of history.

The `cpustat_1sec` table contains the following columns:

- `node_id`

  ID of the node where the thread is running

- `thr_no`

  Thread ID (specific to this node)

- `OS_user_time`

  OS user time

- `OS_system_time`

  OS system time

- `OS_idle_time`

  OS idle time

- `exec_time`

  Thread execution time

- `sleep_time`

  Thread sleep time

- `spin_time`

  Thread spin time

- `send_time`

  Thread send time

- `buffer_full_time`

  Thread buffer full time

- `elapsed_time`

  Elapsed time

23.6.16.21 The ndbinfo cpustat_20sec Table

The `cpustat_20sec` table provides raw, per-thread CPU data obtained each 20 seconds, for each thread running in the `NDB` kernel.

Like `cpustat_50ms` and `cpustat_1sec`, this table shows 20 measurement sets per thread, each referencing a period of the named duration. Thus, `cpsustat_20sec` provides 400 seconds of history.

The `cpustat_20sec` table contains the following columns:

- `node_id`

  ID of the node where the thread is running

- `thr_no`

  Thread ID (specific to this node)

- `OS_user_time`

  OS user time

- `OS_system_time`

  OS system time

- `OS_idle_time`

  OS idle time

- `exec_time`

  Thread execution time

- `sleep_time`

  Thread sleep time

- `spin_time`

  Thread spin time

- `send_time`

  Thread send time

- `buffer_full_time`

  Thread buffer full time

- `elapsed_time`

  Elapsed time

23.6.16.22 The ndbinfo dictionary_columns Table

The table provides `NDB` dictionary information about columns of `NDB` tables. `dictionary_columns` has the columns listed here (with brief descriptions):

- `table_id`

  ID of the table containing the column

- `column_id`

The column's unique ID

- `name`

  Name of the column

- `column_type`

  Data type of the column from the NDB API; see Column::Type, for possible values

- `default_value`

  The column's default value, if any

- `nullable`

  Either of `NULL` or `NOT NULL`

- `array_type`

  The column's internal attribute storage format; one of `FIXED`, `SHORT_VAR`, or `MEDIUM_VAR`; for more information, see Column::ArrayType, in the NDB API documentation

- `storage_type`

  Type of storage used by the table; either of `MEMORY` or `DISK`

- `primary_key`

  `1` if this is a primary key column, otherwise `0`

- `partition_key`

  `1` if this is a partitioning key column, otherwise `0`

- `dynamic`

  `1` if the column is dynamic, otherwise `0`

- `auto_inc`

  `1` if this is an `AUTO_INCREMENT` column, otherwise `0`

You can obtain information about all of the columns in a given table by joining `dictionary_columns` with the `dictionary_tables` table, like this:

```
SELECT dc.*
  FROM dictionary_columns dc
JOIN dictionary_tables dt
  ON dc.table_id=dt.table_id
WHERE dt.table_name='t1'
  AND dt.database_name='mydb';
```

The `dictionary_columns` table was added in NDB 8.0.29.

Note

Blob columns are not shown in this table. This is a known issue.

23.6.16.23 The ndbinfo dictionary_tables Table

This table provides `NDB` dictionary information for `NDB` tables. `dictionary_tables` contains the columns listed here:

- `table_id`

  The table' unique ID

- `database_name`

  Name of the database containing the table

- `table_name`

  Name of the table

- `status`

  The table status; one of `New`, `Changed`, `Retrieved`, `Invalid`, or `Altered`. (See Object::Status, for more information about object status values.)

- `attributes`

  Number of table attributes

- `primary_key_cols`

  Number of columns in the table's primary key

- `primary_key`

  A comma-separated list of the columns in the table's primary key

- `storage`

  Type of storage used by the table; one of `memory`, `disk`, or `default`

- `logging`

  Whether logging is enabled for this table

- `dynamic`

  `1` if the table is dynamic, otherwise `0`; the table is considered dynamic if *`table->getForceVarPart()`* is true, or if at least one table column is dynamic

- `read_backup`

  `1` if read from any replica (`READ_BACKUP` option is enabled for this table, otherwise `0`; see Section 13.1.20.12, “Setting NDB Comment Options”)

- `fully_replicated`

  `1` if `FULLY_REPLICATED` is enabled for this table (each data node in the cluster has a complete copy of the table), `0` if not; see Section 13.1.20.12, “Setting NDB Comment Options”

- `checksum`

  If this table uses a checksum, the value in this column is `1`; if not, it is `0`

- `row_size`

  The amount of data, in bytes that can be stored in one row, not including any blob data stored separately in blob tables; see Table::getRowSizeInBytes(), in the API documentation, for more information

- `min_rows`

Minimum number of rows, as used for calculating partitions; see Table::getMinRows(), in the API documentation, for more information

- `max_rows`

  Maximum number of rows, as used for calculating partitions; see Table::getMaxRows(), in the API documentation, for more information

- `tablespace`

  ID of the tablespace to which the table belongs, if any; this is `0`, if the table does not use data on disk

- `fragment_type`

  The table's fragment type; one of `Single`, `AllSmall`, `AllMedium`, `AllLarge`, `DistrKeyHash`, `DistrKeyLin`, `UserDefined`, `unused`, or `HashMapPartition`; for more information, see Object::FragmentType, in the NDB API documentation

- `hash_map`

  The hash map used by the table

- `fragments`

  Number of table fragments

- `partitions`

  Number of partitions used by the table

- `partition_balance`

  Type of partition balance used, if any; one of `FOR_RP_BY_NODE`, `FOR_RA_BY_NODE`, `FOR_RP_BY_LDM`, `FOR_RA_BY_LDM`, `FOR_RA_BY_LDM_X_2`, `FOR_RA_BY_LDM_X_3`, or `FOR_RA_BY_LDM_X_4`; see Section 13.1.20.12, “Setting NDB Comment Options”

- `contains_GCI`

  `1` if the table includes a global checkpoint index, otherwise `0`

- `single_user_mode`

  Type of access allowed to the table when single user mode is in effect; one of `locked`, `read_only`, or `read_write`; these are equivalent to the values `SingleUserModeLocked`, `SingleUserModeReadOnly`, and `SingleUserModeReadWrite`, respectively, of the `Table::SingleUserMode` type in the NDB API

- `force_var_part`

  This is `1` if *`table`*`->getForceVarPart()` is true for this table, and `0` if it is not

- `GCI_bits`

  Used in testing

- `author_bits`

  Used in testing

The `dictionary_tables` table was added in NDB 8.0.29.

23.6.16.24 The ndbinfo dict_obj_info Table

The dict_obj_info table provides information about NDB data dictionary (DICT) objects such as tables and indexes. (The dict_obj_types table can be queried for a list of all the types.) This information includes the object's type, state, parent object (if any), and fully qualified name.

The dict_obj_info table contains the following columns:

- type

  Type of DICT object; join on dict_obj_types to obtain the name

- id

  Object identifier; for Disk Data undo log files and data files, this is the same as the value shown in the LOGFILE_GROUP_NUMBER column of the Information Schema FILES table; for undo log files, it also the same as the value shown for the log_id column in the ndbinfo logbuffers and logspaces tables

- version

  Object version

- state

  Object state; see Object::State for values and descriptions.

- parent_obj_type

  Parent object's type (a dict_obj_types type ID); 0 indicates that the object has no parent

- parent_obj_id

  Parent object ID (such as a base table); 0 indicates that the object has no parent

- fq_name

  Fully qualified object name; for a table, this has the form *database_name*/def/*table_name*, for a primary key, the form is sys/def/*table_id*/PRIMARY, and for a unique key it is sys/def/*table_id*/*uk_name*$unique

23.6.16.25 The ndbinfo dict_obj_tree Table

The dict_obj_tree table provides a tree-based view of table information from the dict_obj_info table. This is intended primarily for use in testing, but can be useful in visualizing hierarchies of NDB database objects.

The dict_obj_tree table contains the following columns:

- type

  Type of DICT object; join on dict_obj_types to obtain the name of the object type

- id

  Object identifier; same as the id column in dict_obj_info

  For Disk Data undo log files and data files, this is the same as the value shown in the LOGFILE_GROUP_NUMBER column of the Information Schema FILES table; for undo log files, it also the same as the value shown for the log_id column in the ndbinfo logbuffers and logspaces tables

- name

  The fully qualified name of the object; the same as the fq_name column in dict_obj_info

For a table, this is *database_name*/def/*table_name* (the same as its *parent_name*); for an index of any type, this takes the form NDB$INDEX_*index_id*_CUSTOM

- parent_type

  The *DICT* object type of this object's parent object; join on dict_obj_types to obtain the name of the object type

- parent_id

  Identifier for this object's parent object; the same as the dict_obj_info table's id column

- parent_name

  Fully qualified name of this object's parent object; the same as the dict_obj_info table's fq_name column

  For a table, this has the form *database_name*/def/*table_name*. For an index, the name is sys/def/*table_id*/*index_name*. For a primary key, it is sys/def/*table_id*/PRIMARY, and for a unique key it is sys/def/*table_id*/*uk_name*$unique

- root_type

  The *DICT* object type of the root object; join on dict_obj_types to obtain the name of the object type

- root_id

  Identifier for the root object; the same as the dict_obj_info table's id column

- root_name

  Fully qualified name of the root object; the same as the dict_obj_info table's fq_name column

- level

  Level of the object in the hierarchy

- path

  Complete path to the object in the *NDB* object hierarchy; objects are separated by a right arrow (represented as ->), starting with the root object on the left

- indented_name

  The name prefixed with a right arrow (represented as ->) with a number of spaces preceding it that correspond to the object's depth in the hierarchy

The path column is useful for obtaining a complete path to a given NDB database object in a single line, whereas the indented_name column can be used to obtain a tree-like layout of complete hierarchy information for a desired object.

Example: Assuming the existence of a test database and no existing table named t1 in this database, execute the following SQL statement:

```
CREATE TABLE test.t1 (
    a INT PRIMARY KEY,
    b INT,
    UNIQUE KEY(b)
)   ENGINE = NDB;
```

You can obtain the path to the table just created using the query shown here:

```
mysql> SELECT path FROM ndbinfo.dict_obj_tree
```

```
    -> WHERE name LIKE 'test%t1';
+-------------+
| path        |
+-------------+
| test/def/t1 |
+-------------+
1 row in set (0.14 sec)
```

You can see the paths to all dependent objects of this table using the path to the table as the root name in a query like this one:

```
mysql> SELECT path FROM ndbinfo.dict_obj_tree
    -> WHERE root_name = 'test/def/t1';
+-----------------------------------------------------------+
| path                                                      |
+-----------------------------------------------------------+
| test/def/t1                                               |
| test/def/t1 -> sys/def/13/b                               |
| test/def/t1 -> sys/def/13/b -> NDB$INDEX_15_CUSTOM        |
| test/def/t1 -> sys/def/13/b$unique                        |
| test/def/t1 -> sys/def/13/b$unique -> NDB$INDEX_16_UI     |
| test/def/t1 -> sys/def/13/PRIMARY                         |
| test/def/t1 -> sys/def/13/PRIMARY -> NDB$INDEX_14_CUSTOM  |
+-----------------------------------------------------------+
7 rows in set (0.16 sec)
```

To obtain a hierarchical view of the `t1` table with all its dependent objects, execute a query similar to this one which selects the indented name of each object having `test/def/t1` as the name of its root object:

```
mysql> SELECT indented_name FROM ndbinfo.dict_obj_tree
    -> WHERE root_name = 'test/def/t1';
+----------------------------+
| indented_name              |
+----------------------------+
| test/def/t1                |
|   -> sys/def/13/b          |
|     -> NDB$INDEX_15_CUSTOM |
|   -> sys/def/13/b$unique   |
|     -> NDB$INDEX_16_UI     |
|   -> sys/def/13/PRIMARY    |
|     -> NDB$INDEX_14_CUSTOM |
+----------------------------+
7 rows in set (0.15 sec)
```

When working with Disk Data tables, note that, in this context, a tablespace or log file group is considered a root object. This means that you must know the name of any tablespace or log file group associated with a given table, or obtain this information from `SHOW CREATE TABLE` and then querying `INFORMATION_SCHEMA.FILES`, or similar means as shown here:

```
mysql> SHOW CREATE TABLE test.dt_1\G
*************************** 1. row ***************************
       Table: dt_1
Create Table: CREATE TABLE `dt_1` (
  `member_id` int unsigned NOT NULL AUTO_INCREMENT,
  `last_name` varchar(50) NOT NULL,
  `first_name` varchar(50) NOT NULL,
  `dob` date NOT NULL,
  `joined` date NOT NULL,
  PRIMARY KEY (`member_id`),
  KEY `last_name` (`last_name`,`first_name`)
) /*!50100 TABLESPACE `ts_1` STORAGE DISK */ ENGINE=ndbcluster DEFAULT CHARSET=utf8mb4 COLLATE=utf8mb4_
1 row in set (0.00 sec)

mysql> SELECT DISTINCT TABLESPACE_NAME, LOGFILE_GROUP_NAME
    -> FROM INFORMATION_SCHEMA.FILES WHERE TABLESPACE_NAME='ts_1';
+-----------------+--------------------+
| TABLESPACE_NAME | LOGFILE_GROUP_NAME |
+-----------------+--------------------+
| ts_1            | lg_1               |
```

```
+-----------------+--------------------+
1 row in set (0.00 sec)
```

Now you can obtain hierarchical information for the table, tablespace, and log file group like this:

```
mysql> SELECT indented_name FROM ndbinfo.dict_obj_tree
    -> WHERE root_name = 'test/def/dt_1';
+----------------------------+
| indented_name              |
+----------------------------+
| test/def/dt_1              |
|   -> sys/def/23/last_name  |
|     -> NDB$INDEX_25_CUSTOM |
|   -> sys/def/23/PRIMARY    |
|     -> NDB$INDEX_24_CUSTOM |
+----------------------------+
5 rows in set (0.15 sec)

mysql> SELECT indented_name FROM ndbinfo.dict_obj_tree
    -> WHERE root_name = 'ts_1';
+-----------------+
| indented_name   |
+-----------------+
| ts_1            |
|   -> data_1.dat |
|   -> data_2.dat |
+-----------------+
3 rows in set (0.17 sec)

mysql> SELECT indented_name FROM ndbinfo.dict_obj_tree
    -> WHERE root_name LIKE 'lg_1';
+-----------------+
| indented_name   |
+-----------------+
| lg_1            |
|   -> undo_1.log |
|   -> undo_2.log |
+-----------------+
3 rows in set (0.16 sec)
```

The `dict_obj_tree` table was added in NDB 8.0.24.

23.6.16.26 The ndbinfo dict_obj_types Table

The `dict_obj_types` table is a static table listing possible dictionary object types used in the NDB kernel. These are the same types defined by `Object::Type` in the NDB API.

The `dict_obj_types` table contains the following columns:

- `type_id`

  The type ID for this type

- `type_name`

  The name of this type

23.6.16.27 The ndbinfo disk_write_speed_base Table

The `disk_write_speed_base` table provides base information about the speed of disk writes during LCP, backup, and restore operations.

The `disk_write_speed_base` table contains the following columns:

- `node_id`

  Node ID of this node

- `thr_no`

Thread ID of this LDM thread

- `millis_ago`

  Milliseconds since this reporting period ended

- `millis_passed`

  Milliseconds elapsed in this reporting period

- `backup_lcp_bytes_written`

  Number of bytes written to disk by local checkpoints and backup processes during this period

- `redo_bytes_written`

  Number of bytes written to REDO log during this period

- `target_disk_write_speed`

  Actual speed of disk writes per LDM thread (base data)

23.6.16.28 The ndbinfo disk_write_speed_aggregate Table

The `disk_write_speed_aggregate` table provides aggregated information about the speed of disk writes during LCP, backup, and restore operations.

The `disk_write_speed_aggregate` table contains the following columns:

- `node_id`

  Node ID of this node

- `thr_no`

  Thread ID of this LDM thread

- `backup_lcp_speed_last_sec`

  Number of bytes written to disk by backup and LCP processes in the last second

- `redo_speed_last_sec`

  Number of bytes written to REDO log in the last second

- `backup_lcp_speed_last_10sec`

  Number of bytes written to disk by backup and LCP processes per second, averaged over the last 10 seconds

- `redo_speed_last_10sec`

  Number of bytes written to REDO log per second, averaged over the last 10 seconds

- `std_dev_backup_lcp_speed_last_10sec`

  Standard deviation in number of bytes written to disk by backup and LCP processes per second, averaged over the last 10 seconds

- `std_dev_redo_speed_last_10sec`

  Standard deviation in number of bytes written to REDO log per second, averaged over the last 10 seconds

- `backup_lcp_speed_last_60sec`

  Number of bytes written to disk by backup and LCP processes per second, averaged over the last 60 seconds

- `redo_speed_last_60sec`

  Number of bytes written to REDO log per second, averaged over the last 10 seconds

- `std_dev_backup_lcp_speed_last_60sec`

  Standard deviation in number of bytes written to disk by backup and LCP processes per second, averaged over the last 60 seconds

- `std_dev_redo_speed_last_60sec`

  Standard deviation in number of bytes written to REDO log per second, averaged over the last 60 seconds

- `slowdowns_due_to_io_lag`

  Number of seconds since last node start that disk writes were slowed due to REDO log I/O lag

- `slowdowns_due_to_high_cpu`

  Number of seconds since last node start that disk writes were slowed due to high CPU usage

- `disk_write_speed_set_to_min`

  Number of seconds since last node start that disk write speed was set to minimum

- `current_target_disk_write_speed`

  Actual speed of disk writes per LDM thread (aggregated)

23.6.16.29 The ndbinfo disk_write_speed_aggregate_node Table

The `disk_write_speed_aggregate_node` table provides aggregated information per node about the speed of disk writes during LCP, backup, and restore operations.

The `disk_write_speed_aggregate_node` table contains the following columns:

- `node_id`

  Node ID of this node

- `backup_lcp_speed_last_sec`

  Number of bytes written to disk by backup and LCP processes in the last second

- `redo_speed_last_sec`

  Number of bytes written to the redo log in the last second

- `backup_lcp_speed_last_10sec`

  Number of bytes written to disk by backup and LCP processes per second, averaged over the last 10 seconds

- `redo_speed_last_10sec`

  Number of bytes written to the redo log each second, averaged over the last 10 seconds

- `backup_lcp_speed_last_60sec`

Number of bytes written to disk by backup and LCP processes per second, averaged over the last 60 seconds

- `redo_speed_last_60sec`

  Number of bytes written to the redo log each second, averaged over the last 60 seconds

23.6.16.30 The ndbinfo diskpagebuffer Table

The `diskpagebuffer` table provides statistics about disk page buffer usage by NDB Cluster Disk Data tables.

The `diskpagebuffer` table contains the following columns:

- `node_id`

  The data node ID

- `block_instance`

  Block instance

- `pages_written`

  Number of pages written to disk.

- `pages_written_lcp`

  Number of pages written by local checkpoints.

- `pages_read`

  Number of pages read from disk

- `log_waits`

  Number of page writes waiting for log to be written to disk

- `page_requests_direct_return`

  Number of requests for pages that were available in buffer

- `page_requests_wait_queue`

  Number of requests that had to wait for pages to become available in buffer

- `page_requests_wait_io`

  Number of requests that had to be read from pages on disk (pages were unavailable in buffer)

Notes

You can use this table with NDB Cluster Disk Data tables to determine whether `DiskPageBufferMemory` is sufficiently large to allow data to be read from the buffer rather from disk; minimizing disk seeks can help improve performance of such tables.

You can determine the proportion of reads from `DiskPageBufferMemory` to the total number of reads using a query such as this one, which obtains this ratio as a percentage:

```
SELECT
  node_id,
  100 * page_requests_direct_return /
    (page_requests_direct_return + page_requests_wait_io)
      AS hit_ratio
```

```
FROM ndbinfo.diskpagebuffer;
```

The result from this query should be similar to what is shown here, with one row for each data node in the cluster (in this example, the cluster has 4 data nodes):

```
+---------+-----------+
| node_id | hit_ratio |
+---------+-----------+
|       5 |   97.6744 |
|       6 |   97.6879 |
|       7 |   98.1776 |
|       8 |   98.1343 |
+---------+-----------+
4 rows in set (0.00 sec)
```

`hit_ratio` values approaching 100% indicate that only a very small number of reads are being made from disk rather than from the buffer, which means that Disk Data read performance is approaching an optimum level. If any of these values are less than 95%, this is a strong indicator that the setting for `DiskPageBufferMemory` needs to be increased in the `config.ini` file.

Note

A change in `DiskPageBufferMemory` requires a rolling restart of all of the cluster's data nodes before it takes effect.

`block_instance` refers to an instance of a kernel block. Together with the block name, this number can be used to look up a given instance in the `threadblocks` table. Using this information, you can obtain information about disk page buffer metrics relating to individual threads; an example query using `LIMIT 1` to limit the output to a single thread is shown here:

```
mysql> SELECT
     >   node_id, thr_no, block_name, thread_name, pages_written,
     >   pages_written_lcp, pages_read, log_waits,
     >   page_requests_direct_return, page_requests_wait_queue,
     >   page_requests_wait_io
     > FROM ndbinfo.diskpagebuffer
     >   INNER JOIN ndbinfo.threadblocks USING (node_id, block_instance)
     >   INNER JOIN ndbinfo.threads USING (node_id, thr_no)
     > WHERE block_name = 'PGMAN' LIMIT 1\G
*************************** 1. row ***************************
                    node_id: 1
                     thr_no: 1
                 block_name: PGMAN
                thread_name: rep
              pages_written: 0
          pages_written_lcp: 0
                 pages_read: 1
                  log_waits: 0
page_requests_direct_return: 4
   page_requests_wait_queue: 0
      page_requests_wait_io: 1
1 row in set (0.01 sec)
```

23.6.16.31 The ndbinfo diskstat Table

The `diskstat` table provides information about writes to Disk Data tablespaces during the past 1 second.

The `diskstat` table contains the following columns:

- `node_id`

  Node ID of this node

- `block_instance`

  ID of reporting instance of `PGMAN`

- `pages_made_dirty`

  Number of pages made dirty during the past second

- `reads_issued`

  Reads issued during the past second

- `reads_completed`

  Reads completed during the past second

- `writes_issued`

  Writes issued during the past second

- `writes_completed`

  Writes completed during the past second

- `log_writes_issued`

  Number of times a page write has required a log write during the past second

- `log_writes_completed`

  Number of log writes completed during the last second

- `get_page_calls_issued`

  Number of `get_page()` calls issued during the past second

- `get_page_reqs_issued`

  Number of times that a `get_page()` call has resulted in a wait for I/O or completion of I/O already begun during the past second

- `get_page_reqs_completed`

  Number of `get_page()` calls waiting for I/O or I/O completion that have completed during the past second

Notes

Each row in this table corresponds to an instance of `PGMAN`; there is one such instance per LDM thread plus an additional instance for each data node.

23.6.16.32 The ndbinfo diskstats_1sec Table

The `diskstats_1sec` table provides information about writes to Disk Data tablespaces over the past 20 seconds.

The `diskstat` table contains the following columns:

- `node_id`

  Node ID of this node

- `block_instance`

  ID of reporting instance of `PGMAN`

- `pages_made_dirty`

Pages made dirty during the designated 1-second interval

- `reads_issued`

  Reads issued during the designated 1-second interval

- `reads_completed`

  Reads completed during the designated 1-second interval

- `writes_issued`

  Writes issued during the designated 1-second interval

- `writes_completed`

  Writes completed during the designated 1-second interval

- `log_writes_issued`

  Number of times a page write has required a log write during the designated 1-second interval

- `log_writes_completed`

  Number of log writes completed during the designated 1-second interval

- `get_page_calls_issued`

  Number of `get_page()` calls issued during the designated 1-second interval

- `get_page_reqs_issued`

  Number of times that a `get_page()` call has resulted in a wait for I/O or completion of I/O already begun during the designated 1-second interval

- `get_page_reqs_completed`

  Number of `get_page()` calls waiting for I/O or I/O completion that have completed during the designated 1-second interval

- `seconds_ago`

  Number of 1-second intervals in the past of the interval to which this row applies

Notes

Each row in this table corresponds to an instance of `PGMAN` during a 1-second interval occurring from 0 to 19 seconds ago; there is one such instance per LDM thread plus an additional instance for each data node.

23.6.16.33 The ndbinfo error_messages Table

The `error_messages` table provides information about

The `error_messages` table contains the following columns:

- `error_code`

  Numeric error code

- `error_description`

  Description of error

- `error_status`

  Error status code

- `error_classification`

  Error classification code

Notes

`error_code` is a numeric NDB error code. This is the same error code that can be supplied to `ndb_perror`.

`error_description` provides a basic description of the condition causing the error.

The `error_status` column provides status information relating to the error. Possible values for this column are listed here:

- `No error`
- `Illegal connect string`
- `Illegal server handle`
- `Illegal reply from server`
- `Illegal number of nodes`
- `Illegal node status`
- `Out of memory`
- `Management server not connected`
- `Could not connect to socket`
- `Start failed`
- `Stop failed`
- `Restart failed`
- `Could not start backup`
- `Could not abort backup`
- `Could not enter single user mode`
- `Could not exit single user mode`
- `Failed to complete configuration change`
- `Failed to get configuration`
- `Usage error`
- `Success`
- `Permanent error`
- `Temporary error`
- `Unknown result`

- `Temporary error, restart node`
- `Permanent error, external action needed`
- `Ndbd file system error, restart node initial`
- `Unknown`

The error_classification column shows the error classification. See NDB Error Classifications, for information about classification codes and their meanings.

23.6.16.34 The ndbinfo events Table

This table provides information about event subscriptions in `NDB`. The columns of the `events` table are listed here, with short descriptions of each:

- `event_id`

  The event ID

- `name`

  The name of the event

- `table_id`

  The ID of the table on which the event occurred

- `reporting`

  One of `updated`, `all`, `subscribe`, or `DDL`

- `columns`

  A comma-separated list of columns affected by the event

- `table_event`

  One or more of `INSERT`, `DELETE`, `UPDATE`, `SCAN`, `DROP`, `ALTER`, `CREATE`, `GCP_COMPLETE`, `CLUSTER_FAILURE`, `STOP`, `NODE_FAILURE`, `SUBSCRIBE`, `UNSUBSCRIBE`, and `ALL` (defined by `Event::TableEvent` in the NDB API)

The `events` table was added in NDB 8.0.29.

23.6.16.35 The ndbinfo files Table

The `files` tables provides information about files and other objects used by `NDB` disk data tables, and contains the columns listed here:

- `id`

  Object ID

- `type`

  The type of object; one of `Log file group`, `Tablespace`, `Undo file`, or `Data file`

- `name`

  The name of the object

- `parent`

  ID of the parent object

- `parent_name`

  Name of the parent object

- `free_extents`

  Number of free extents

- `total_extents`

  Total number of extents

- `extent_size`

  Extent size (MB)

- `initial_size`

  Initial size (bytes)

- `maximum_size`

  Maximum size (bytes)

- `autoextend_size`

  Autoextend size (bytes)

For log file groups and tablespaces, `parent` is always `0`, and the `parent_name`, `free_extents`, `total_extents`, `extent_size`, `initial_size`, `maximum_size`, and `autoentend_size` columns are all `NULL`.

The `files` table is empty if no disk data objects have been created in `NDB`. See Section 23.6.11.1, “NDB Cluster Disk Data Objects”, for more information.

The `files` table was added in NDB 8.0.29.

See also Section 26.3.15, “The INFORMATION_SCHEMA FILES Table”.

23.6.16.36 The ndbinfo foreign_keys Table

The `foreign_keys` table provides information about foreign keys on `NDB` tables. This table has the following columns:

- `object_id`

  The foreign key's object ID

- `name`

  Name of the foreign key

- `parent_table`

  The name of the foreign key's parent table

- `parent_columns`

  A comma-delimited list of parent columns

- `child_table`

  The name of the child table

- `child_columns`

  A comma-separated list of child columns

- `parent_index`

  Name of the parent index

- `child_index`

  Name of the child index

- `on_update_action`

  The `ON UPDATE` action specified for the foreign key; one of `No Action`, `Restrict`, `Cascade`, `Set Null`, or `Set Default`

- `on_delete_action`

  The `ON DELETE` action specified for the foreign key; one of `No Action`, `Restrict`, `Cascade`, `Set Null`, or `Set Default`

The `foreign_keys` table was added in NDB 8.0.29.

23.6.16.37 The ndbinfo hash_maps Table

- `id`

  The hash map's unique ID

- `version`

  Hash map version (integer)

- `state`

  Hash map state; see Object::State for values and descriptions.

- `fq_name`

  The hash map's fully qualified name

The `hash_maps` table is actually a view consisting of the four columns having the same names of the `dict_obj_info` table, as shown here:

```
CREATE VIEW hash_maps AS
  SELECT id, version, state, fq_name
  FROM dict_obj_info
  WHERE type=24;  # Hash map; defined in dict_obj_types
```

See the description of `dict_obj_info` for more information.

The `hash_maps` table was added in NDB 8.0.29.

23.6.16.38 The ndbinfo hwinfo Table

The `hwinfo` table provides information about the hardware on which a given data node executes.

The `hwinfo` table contains the following columns:

- `node_id`

  Node ID

- `cpu_cnt_max`

  Number of processors on this host

- `cpu_cnt`

  Number of processors available to this node

- `num_cpu_cores`

  Number of CPU cores on this host

- `num_cpu_sockets`

  Number of CPU sockets on this host

- `HW_memory_size`

  Amount of memory available on this host

- `model_name`

  CPU model name

Notes

The `hwinfo` table is available on all operating systems supported by `NDB`.

This table was added in NDB 8.0.23.

23.6.16.39 The ndbinfo index_columns Table

This table provides information about indexes on `NDB` tables. The columns of the `index_columns` table are listed here, along with brief descriptions:

- `table_id`

  Unique ID of the `NDB` table for which the index is defined

- Name of the database containing this table

  `varchar(64)`

- `table_name`

  Name of the table

- `index_object_id`

  Object ID of this index

- `index_name`

  Name of the index; if the index is not named, the name of the first column in the index is used

- `index_type`

  Type of index; normally this is 3 (unique hash index) or 6 (ordered index); the values are the same as those in the `type_id` column of the `dict_obj_types` table

- `status`

  One of `new`, `changed`, `retrieved`, `invalid`, or `altered`

- `columns`

  A comma-delimited list of columns making up the index

The `index_columns` table was added in NDB 8.0.29.

23.6.16.40 The ndbinfo index_stats Table

The `index_stats` table provides basic information about `NDB` index statistics.

More complete index statistics information can be obtained using the `ndb_index_stat` utility.

The `index_stats` table contains the following columns:

- `index_id`

  Index ID

- `index_version`

  Index version

- `sample_version`

  Sample version

Notes

This table was added in NDB 8.0.28.

23.6.16.41 The ndbinfo locks_per_fragment Table

The `locks_per_fragment` table provides information about counts of lock claim requests, and the outcomes of these requests on a per-fragment basis, serving as a companion table to `operations_per_fragment` and `memory_per_fragment`. This table also shows the total time spent waiting for locks successfully and unsuccessfully since fragment or table creation, or since the most recent restart.

The `locks_per_fragment` table contains the following columns:

- `fq_name`

  Fully qualified table name

- `parent_fq_name`

  Fully qualified name of parent object

- `type`

  Table type; see text for possible values

- `table_id`

  Table ID

- `node_id`

  Reporting node ID

- `block_instance`

  LDM instance ID

- `fragment_num`

  Fragment identifier

- `ex_req`

  Exclusive lock requests started

- `ex_imm_ok`

  Exclusive lock requests immediately granted

- `ex_wait_ok`

  Exclusive lock requests granted following wait

- `ex_wait_fail`

  Exclusive lock requests not granted

- `sh_req`

  Shared lock requests started

- `sh_imm_ok`

  Shared lock requests immediately granted

- `sh_wait_ok`

  Shared lock requests granted following wait

- `sh_wait_fail`

  Shared lock requests not granted

- `wait_ok_millis`

  Time spent waiting for lock requests that were granted, in milliseconds

- `wait_fail_millis`

  Time spent waiting for lock requests that failed, in milliseconds

Notes

`block_instance` refers to an instance of a kernel block. Together with the block name, this number can be used to look up a given instance in the `threadblocks` table.

`fq_name` is a fully qualified database object name in *`database`*/*`schema`*/*`name`* format, such as `test/def/t1` or `sys/def/10/b$unique`.

`parent_fq_name` is the fully qualified name of this object's parent object (table).

`table_id` is the table's internal ID generated by `NDB`. This is the same internal table ID shown in other `ndbinfo` tables; it is also visible in the output of `ndb_show_tables`.

The `type` column shows the type of table. This is always one of `System table`, `User table`, `Unique hash index`, `Hash index`, `Unique ordered index`, `Ordered index`, `Hash index trigger`, `Subscription trigger`, `Read only constraint`, `Index trigger`, `Reorganize trigger`, `Tablespace`, `Log file group`, `Data file`, `Undo file`, `Hash map`, `Foreign key definition`, `Foreign key parent trigger`, `Foreign key child trigger`, or `Schema transaction`.

The values shown in all of the columns `ex_req`, `ex_req_imm_ok`, `ex_wait_ok`, `ex_wait_fail`, `sh_req`, `sh_req_imm_ok`, `sh_wait_ok`, and `sh_wait_fail` represent cumulative numbers of requests since the table or fragment was created, or since the last restart of this node, whichever of these occurred later. This is also true for the time values shown in the `wait_ok_millis` and `wait_fail_millis` columns.

Every lock request is considered either to be in progress, or to have completed in some way (that is, to have succeeded or failed). This means that the following relationships are true:

```
ex_req >= (ex_req_imm_ok + ex_wait_ok + ex_wait_fail)

sh_req >= (sh_req_imm_ok + sh_wait_ok + sh_wait_fail)
```

The number of requests currently in progress is the current number of incomplete requests, which can be found as shown here:

```
[exclusive lock requests in progress] =
    ex_req - (ex_req_imm_ok + ex_wait_ok + ex_wait_fail)

[shared lock requests in progress] =
    sh_req - (sh_req_imm_ok + sh_wait_ok + sh_wait_fail)
```

A failed wait indicates an aborted transaction, but the abort may or may not be caused by a lock wait timeout. You can obtain the total number of aborts while waiting for locks as shown here:

```
[aborts while waiting for locks] = ex_wait_fail + sh_wait_fail
```

23.6.16.42 The ndbinfo logbuffers Table

The `logbuffer` table provides information on NDB Cluster log buffer usage.

The `logbuffers` table contains the following columns:

- `node_id`

  The ID of this data node.

- `log_type`

  Type of log. One of: `REDO`, `DD-UNDO`, `BACKUP-DATA`, or `BACKUP-LOG`.

- `log_id`

  The log ID; for Disk Data undo log files, this is the same as the value shown in the `LOGFILE_GROUP_NUMBER` column of the Information Schema `FILES` table as well as the value shown for the `log_id` column of the `ndbinfo logspaces` table

- `log_part`

  The log part number

- `total`

  Total space available for this log

- `used`

  Space used by this log

Notes

`logbuffers` table rows reflecting two additional log types are available when performing an NDB backup. One of these rows has the log type `BACKUP-DATA`, which shows the amount of data buffer used during backup to copy fragments to backup files. The other row has the log type `BACKUP-LOG`, which displays the amount of log buffer used during the backup to record changes made after

the backup has started. One each of these `log_type` rows is shown in the `logbuffers` table for each data node in the cluster. These rows are not present unless an NDB backup is currently being performed.

23.6.16.43 The ndbinfo logspaces Table

This table provides information about NDB Cluster log space usage.

The `logspaces` table contains the following columns:

- `node_id`

  The ID of this data node.

- `log_type`

  Type of log; one of: `REDO` or `DD-UNDO`.

- `node_id`

  The log ID; for Disk Data undo log files, this is the same as the value shown in the `LOGFILE_GROUP_NUMBER` column of the Information Schema `FILES` table, as well as the value shown for the `log_id` column of the `ndbinfo logbuffers` table

- `log_part`

  The log part number.

- `total`

  Total space available for this log.

- `used`

  Space used by this log.

23.6.16.44 The ndbinfo membership Table

The `membership` table describes the view that each data node has of all the others in the cluster, including node group membership, president node, arbitrator, arbitrator successor, arbitrator connection states, and other information.

The `membership` table contains the following columns:

- `node_id`

  This node's node ID

- `group_id`

  Node group to which this node belongs

- `left node`

  Node ID of the previous node

- `right_node`

  Node ID of the next node

- `president`

  President's node ID

- `successor`

  Node ID of successor to president

- `succession_order`

  Order in which this node succeeds to presidency

- `Conf_HB_order`

  -

- `arbitrator`

  Node ID of arbitrator

- `arb_ticket`

  Internal identifier used to track arbitration

- `arb_state`

  Arbitration state

- `arb_connected`

  Whether this node is connected to the arbitrator; either of `Yes` or `No`

- `connected_rank1_arbs`

  Connected arbitrators of rank 1

- `connected_rank2_arbs`

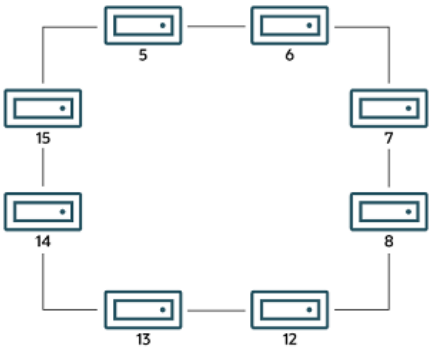
  Connected arbitrators of rank 1

Notes

The node ID and node group ID are the same as reported by `ndb_mgm -e "SHOW"`.

`left_node` and `right_node` are defined in terms of a model that connects all data nodes in a circle, in order of their node IDs, similar to the ordering of the numbers on a clock dial, as shown here:

Figure 23.6 Circular Arrangement of NDB Cluster Nodes



In this example, we have 8 data nodes, numbered 5, 6, 7, 8, 12, 13, 14, and 15, ordered clockwise in a circle. We determine “left” and “right” from the interior of the circle. The node to the left of node 5 is node 15, and the node to the right of node 5 is node 6. You can see all these relationships by running the following query and observing the output:

```
mysql> SELECT node_id,left_node,right_node
    -> FROM ndbinfo.membership;
```

```
+---------+-----------+------------+
| node_id | left_node | right_node |
+---------+-----------+------------+
|       5 |        15 |          6 |
|       6 |         5 |          7 |
|       7 |         6 |          8 |
|       8 |         7 |         12 |
|      12 |         8 |         13 |
|      13 |        12 |         14 |
|      14 |        13 |         15 |
|      15 |        14 |          5 |
+---------+-----------+------------+
8 rows in set (0.00 sec)
```

The designations “left” and “right” are used in the event log in the same way.

The `president` node is the node viewed by the current node as responsible for setting an arbitrator (see NDB Cluster Start Phases). If the president fails or becomes disconnected, the current node expects the node whose ID is shown in the `successor` column to become the new president. The `succession_order` column shows the place in the succession queue that the current node views itself as having.

In a normal NDB Cluster, all data nodes should see the same node as `president`, and the same node (other than the president) as its `successor`. In addition, the current president should see itself as `1` in the order of succession, the `successor` node should see itself as `2`, and so on.

All nodes should show the same `arb_ticket` values as well as the same `arb_state` values. Possible `arb_state` values are `ARBIT_NULL`, `ARBIT_INIT`, `ARBIT_FIND`, `ARBIT_PREP1`, `ARBIT_PREP2`, `ARBIT_START`, `ARBIT_RUN`, `ARBIT_CHOOSE`, `ARBIT_CRASH`, and `UNKNOWN`.

`arb_connected` shows whether this node is connected to the node shown as this node's `arbitrator`.

The `connected_rank1_arbs` and `connected_rank2_arbs` columns each display a list of 0 or more arbitrators having an `ArbitrationRank` equal to 1, or to 2, respectively.

Note

Both management nodes and API nodes are eligible to become arbitrators.

23.6.16.45 The ndbinfo memoryusage Table

Querying this table provides information similar to that provided by the `ALL REPORT MemoryUsage` command in the `ndb_mgm` client, or logged by `ALL DUMP 1000`.

The `memoryusage` table contains the following columns:

- `node_id`

  The node ID of this data node.

- `memory_type`

  One of `Data memory`, `Index memory`, or `Long message buffer`.

- `used`

  Number of bytes currently used for data memory or index memory by this data node.

- `used_pages`

  Number of pages currently used for data memory or index memory by this data node; see text.

- `total`

Total number of bytes of data memory or index memory available for this data node; see text.

- `total_pages`

  Total number of memory pages available for data memory or index memory on this data node; see text.

Notes

The `total` column represents the total amount of memory in bytes available for the given resource (data memory or index memory) on a particular data node. This number should be approximately equal to the setting of the corresponding configuration parameter in the `config.ini` file.

Suppose that the cluster has 2 data nodes having node IDs `5` and `6`, and the `config.ini` file contains the following:

```
[ndbd default]
DataMemory = 1G
IndexMemory = 1G
```

Suppose also that the value of the `LongMessageBuffer` configuration parameter is allowed to assume its default (64 MB).

The following query shows approximately the same values:

```
mysql> SELECT node_id, memory_type, total
     > FROM ndbinfo.memoryusage;
+---------+--------------------+------------+
| node_id | memory_type        | total      |
+---------+--------------------+------------+
|       5 | Data memory        | 1073741824 |
|       5 | Index memory       | 1074003968 |
|       5 | Long message buffer |   67108864 |
|       6 | Data memory        | 1073741824 |
|       6 | Index memory       | 1074003968 |
|       6 | Long message buffer |   67108864 |
+---------+--------------------+------------+
6 rows in set (0.00 sec)
```

In this case, the `total` column values for index memory are slightly higher than the value set of `IndexMemory` due to internal rounding.

For the `used_pages` and `total_pages` columns, resources are measured in pages, which are 32K in size for `DataMemory` and 8K for `IndexMemory`. For long message buffer memory, the page size is 256 bytes.

23.6.16.46 The ndbinfo memory_per_fragment Table

The `memory_per_fragment` table provides information about the usage of memory by individual fragments.

The `memory_per_fragment` table contains the following columns:

- `fq_name`

  Name of this fragment

- `parent_fq_name`

  Name of this fragment's parent

- `type`

  Type of object; see text for possible values

- `table_id`

  Table ID for this table

- `node_id`

  Node ID for this node

- `block_instance`

  Kernel block instance ID

- `fragment_num`

  Fragment ID (number)

- `fixed_elem_alloc_bytes`

  Number of bytes allocated for fixed-sized elements

- `fixed_elem_free_bytes`

  Free bytes remaining in pages allocated to fixed-size elements

- `fixed_elem_size_bytes`

  Length of each fixed-size element in bytes

- `fixed_elem_count`

  Number of fixed-size elements

- `fixed_elem_free_count`

  Number of free rows for fixed-size elements

- `var_elem_alloc_bytes`

  Number of bytes allocated for variable-size elements

- `var_elem_free_bytes`

  Free bytes remaining in pages allocated to variable-size elements

- `var_elem_count`

  Number of variable-size elements

- `hash_index_alloc_bytes`

  Number of bytes allocated to hash indexes

Notes

The `type` column from this table shows the dictionary object type used for this fragment (`Object::Type`, in the NDB API), and can take any one of the values shown in the following list:

- System table
- User table
- Unique hash index
- Hash index

- Unique ordered index
- Ordered index
- Hash index trigger
- Subscription trigger
- Read only constraint
- Index trigger
- Reorganize trigger
- Tablespace
- Log file group
- Data file
- Undo file
- Hash map
- Foreign key definition
- Foreign key parent trigger
- Foreign key child trigger
- Schema transaction

You can also obtain this list by executing `SELECT * FROM ndbinfo.dict_obj_types` in the `mysql` client.

The `block_instance` column provides the NDB kernel block instance number. You can use this to obtain information about specific threads from the `threadblocks` table.

23.6.16.47 The ndbinfo nodes Table

This table contains information on the status of data nodes. For each data node that is running in the cluster, a corresponding row in this table provides the node's node ID, status, and uptime. For nodes that are starting, it also shows the current start phase.

The `nodes` table contains the following columns:

- `node_id`

  The data node's unique node ID in the cluster.

- `uptime`

  Time since the node was last started, in seconds.

- `status`

  Current status of the data node; see text for possible values.

- `start_phase`

  If the data node is starting, the current start phase.

- `config_generation`

  The version of the cluster configuration file in use on this data node.

Notes

The `uptime` column shows the time in seconds that this node has been running since it was last started or restarted. This is a `BIGINT` value. This figure includes the time actually needed to start the node; in other words, this counter starts running the moment that `ndbd` or `ndbmtd` is first invoked; thus, even for a node that has not yet finished starting, `uptime` may show a nonzero value.

The `status` column shows the node's current status. This is one of: `NOTHING`, `CMVMI`, `STARTING`, `STARTED`, `SINGLEUSER`, `STOPPING_1`, `STOPPING_2`, `STOPPING_3`, or `STOPPING_4`. When the status is `STARTING`, you can see the current start phase in the `start_phase` column (see later in this section). `SINGLEUSER` is displayed in the `status` column for all data nodes when the cluster is in single user mode (see Section 23.6.6, “NDB Cluster Single User Mode”). Seeing one of the `STOPPING` states does not necessarily mean that the node is shutting down but can mean rather that it is entering a new state. For example, if you put the cluster in single user mode, you can sometimes see data nodes report their state briefly as `STOPPING_2` before the status changes to `SINGLEUSER`.

The `start_phase` column uses the same range of values as those used in the output of the `ndb_mgm` client *`node_id`* `STATUS` command (see Section 23.6.1, “Commands in the NDB Cluster Management Client”). If the node is not currently starting, then this column shows `0`. For a listing of NDB Cluster start phases with descriptions, see Section 23.6.4, “Summary of NDB Cluster Start Phases”.

The `config_generation` column shows which version of the cluster configuration is in effect on each data node. This can be useful when performing a rolling restart of the cluster in order to make changes in configuration parameters. For example, from the output of the following `SELECT` statement, you can see that node 3 is not yet using the latest version of the cluster configuration (`6`) although nodes 1, 2, and 4 are doing so:

```
mysql> USE ndbinfo;
Database changed
mysql> SELECT * FROM nodes;
+---------+--------+---------+-------------+-------------------+
| node_id | uptime | status  | start_phase | config_generation |
+---------+--------+---------+-------------+-------------------+
|       1 |  10462 | STARTED |           0 |                 6 |
|       2 |  10460 | STARTED |           0 |                 6 |
|       3 |  10457 | STARTED |           0 |                 5 |
|       4 |  10455 | STARTED |           0 |                 6 |
+---------+--------+---------+-------------+-------------------+
2 rows in set (0.04 sec)
```

Therefore, for the case just shown, you should restart node 3 to complete the rolling restart of the cluster.

Nodes that are stopped are not accounted for in this table. Suppose that you have an NDB Cluster with 4 data nodes (node IDs 1, 2, 3 and 4), and all nodes are running normally, then this table contains 4 rows, 1 for each data node:

```
mysql> USE ndbinfo;
Database changed
mysql> SELECT * FROM nodes;
+---------+--------+---------+-------------+-------------------+
| node_id | uptime | status  | start_phase | config_generation |
+---------+--------+---------+-------------+-------------------+
|       1 |  11776 | STARTED |           0 |                 6 |
|       2 |  11774 | STARTED |           0 |                 6 |
|       3 |  11771 | STARTED |           0 |                 6 |
|       4 |  11769 | STARTED |           0 |                 6 |
+---------+--------+---------+-------------+-------------------+
4 rows in set (0.04 sec)
```

If you shut down one of the nodes, only the nodes that are still running are represented in the output of this `SELECT` statement, as shown here:

```
ndb_mgm> 2 STOP
Node 2: Node shutdown initiated
```

```
Node 2: Node shutdown completed.
Node 2 has shutdown.
```

```
mysql> SELECT * FROM nodes;
+---------+--------+---------+-------------+-------------------+
| node_id | uptime | status  | start_phase | config_generation |
+---------+--------+---------+-------------+-------------------+
|       1 |  11807 | STARTED |           0 |                 6 |
|       3 |  11802 | STARTED |           0 |                 6 |
|       4 |  11800 | STARTED |           0 |                 6 |
+---------+--------+---------+-------------+-------------------+
3 rows in set (0.02 sec)
```

23.6.16.48 The ndbinfo operations_per_fragment Table

The `operations_per_fragment` table provides information about the operations performed on individual fragments and fragment replicas, as well as about some of the results from these operations.

The `operations_per_fragment` table contains the following columns:

- `fq_name`

  Name of this fragment

- `parent_fq_name`

  Name of this fragment's parent

- `type`

  Type of object; see text for possible values

- `table_id`

  Table ID for this table

- `node_id`

  Node ID for this node

- `block_instance`

  Kernel block instance ID

- `fragment_num`

  Fragment ID (number)

- `tot_key_reads`

  Total number of key reads for this fragment replica

- `tot_key_inserts`

  Total number of key inserts for this fragment replica

- `tot_key_updates`

  total number of key updates for this fragment replica

- `tot_key_writes`

  Total number of key writes for this fragment replica

- `tot_key_deletes`

Total number of key deletes for this fragment replica

- `tot_key_refs`

  Number of key operations refused

- `tot_key_attrinfo_bytes`

  Total size of all `attrinfo` attributes

- `tot_key_keyinfo_bytes`

  Total size of all `keyinfo` attributes

- `tot_key_prog_bytes`

  Total size of all interpreted programs carried by `attrinfo` attributes

- `tot_key_inst_exec`

  Total number of instructions executed by interpreted programs for key operations

- `tot_key_bytes_returned`

  Total size of all data and metadata returned from key read operations

- `tot_frag_scans`

  Total number of scans performed on this fragment replica

- `tot_scan_rows_examined`

  Total number of rows examined by scans

- `tot_scan_rows_returned`

  Total number of rows returned to client

- `tot_scan_bytes_returned`

  Total size of data and metadata returned to the client

- `tot_scan_prog_bytes`

  Total size of interpreted programs for scan operations

- `tot_scan_bound_bytes`

  Total size of all bounds used in ordered index scans

- `tot_scan_inst_exec`

  Total number of instructions executed for scans

- `tot_qd_frag_scans`

  Number of times that scans of this fragment replica have been queued

- `conc_frag_scans`

  Number of scans currently active on this fragment replica (excluding queued scans)

- `conc_qd_frag_scans`

Number of scans currently queued for this fragment replica

- tot_commits

  Total number of row changes committed to this fragment replica

Notes

The `fq_name` contains the fully qualified name of the schema object to which this fragment replica belongs. This currently has the following formats:

- Base table: *`DbName`*`/def/`*`TblName`*
- `BLOB` table: *`DbName`*`/def/NDB$BLOB_`*`BaseTblId_ColNo`*
- Ordered index: `sys/def/`*`BaseTblId/IndexName`*
- Unique index: `sys/def/`*`BaseTblId/IndexName`*`$unique`

The `$unique` suffix shown for unique indexes is added by `mysqld`; for an index created by a different NDB API client application, this may differ, or not be present.

The syntax just shown for fully qualified object names is an internal interface which is subject to change in future releases.

Consider a table `t1` created and modified by the following SQL statements:

```
CREATE DATABASE mydb;

USE mydb;

CREATE TABLE t1 (
  a INT NOT NULL,
  b INT NOT NULL,
  t TEXT NOT NULL,
  PRIMARY KEY (b)
) ENGINE=ndbcluster;

CREATE UNIQUE INDEX ix1 ON t1(b) USING HASH;
```

If `t1` is assigned table ID 11, this yields the `fq_name` values shown here:

- Base table: `mydb/def/t1`
- `BLOB` table: `mydb/def/NDB$BLOB_11_2`
- Ordered index (primary key): `sys/def/11/PRIMARY`
- Unique index: `sys/def/11/ix1$unique`

For indexes or `BLOB` tables, the `parent_fq_name` column contains the `fq_name` of the corresponding base table. For base tables, this column is always `NULL`.

The `type` column shows the schema object type used for this fragment, which can take any one of the values `System table`, `User table`, `Unique hash index`, or `Ordered index`. `BLOB` tables are shown as `User table`.

The `table_id` column value is unique at any given time, but can be reused if the corresponding object has been deleted. The same ID can be seen using the `ndb_show_tables` utility.

The `block_instance` column shows which LDM instance this fragment replica belongs to. You can use this to obtain information about specific threads from the `threadblocks` table. The first such instance is always numbered 0.

Since there are typically two fragment replicas, and assuming that this is so, each `fragment_num` value should appear twice in the table, on two different data nodes from the same node group.

Since `NDB` does not use single-key access for ordered indexes, the counts for `tot_key_reads`, `tot_key_inserts`, `tot_key_updates`, `tot_key_writes`, and `tot_key_deletes` are not incremented by ordered index operations.

Note

When using `tot_key_writes`, you should keep in mind that a write operation in this context updates the row if the key exists, and inserts a new row otherwise. (One use of this is in the `NDB` implementation of the `REPLACE` SQL statement.)

The `tot_key_refs` column shows the number of key operations refused by the LDM. Generally, such a refusal is due to duplicate keys (inserts), `Key not found` errors (updates, deletes, and reads), or the operation was rejected by an interpreted program used as a predicate on the row matching the key.

The `attrinfo` and `keyinfo` attributes counted by the `tot_key_attrinfo_bytes` and `tot_key_keyinfo_bytes` columns are attributes of an `LQHKEYREQ` signal (see The NDB Communication Protocol) used to initiate a key operation by the LDM. An `attrinfo` typically contains tuple field values (inserts and updates) or projection specifications (for reads); `keyinfo` contains the primary or unique key needed to locate a given tuple in this schema object.

The value shown by `tot_frag_scans` includes both full scans (that examine every row) and scans of subsets. Unique indexes and `BLOB` tables are never scanned, so this value, like other scan-related counts, is 0 for fragment replicas of these.

`tot_scan_rows_examined` may display less than the total number of rows in a given fragment replica, since ordered index scans can limited by bounds. In addition, a client may choose to end a scan before all potentially matching rows have been examined; this occurs when using an SQL statement containing a `LIMIT` or `EXISTS` clause, for example. `tot_scan_rows_returned` is always less than or equal to `tot_scan_rows_examined`.

`tot_scan_bytes_returned` includes, in the case of pushed joins, projections returned to the `DBSPJ` block in the NDB kernel.

`tot_qd_frag_scans` can be effected by the setting for the `MaxParallelScansPerFragment` data node configuration parameter, which limits the number of scans that may execute concurrently on a single fragment replica.

23.6.16.49 The ndbinfo pgman_time_track_stats Table

This table provides information regarding the latency of disk operations for NDB Cluster Disk Data tablespaces.

The `pgman_time_track_stats` table contains the following columns:

- `node_id`

  Unique node ID of this node in the cluster

- `block_number`

  Block number (from `blocks` table)

- `block_instance`

  Block instance number

- `upper_bound`

  Upper bound

- `page_reads`

  Page read latency (ms)

- `page_writes`

  Page write latency (ms)

- `log_waits`

  Log wait latency (ms)

- `get_page`

  Latency of `get_page()` calls (ms)

Notes

The read latency (`page_reads` column) measures the time from when the read request is sent to the file system thread until the read is complete and has been reported back to the execution thread. The write latency (`page_writes`) is calculated in a similar fashion. The size of the page read to or written from a Disk Data tablespace is always 32 KB.

Log wait latency (`log_waits` column) is the length of time a page write must wait for the undo log to be flushed, which must be done prior to each page write.

23.6.16.50 The ndbinfo processes Table

This table contains information about NDB Cluster node processes; each node is represented by the row in the table. Only nodes that are connected to the cluster are shown in this table. You can obtain information about nodes that are configured but not connected to the cluster from the `nodes` and `config_nodes` tables.

The `processes` table contains the following columns:

- `node_id`

  The node's unique node ID in the cluster

- `node_type`

  Type of node (management, data, or API node; see text)

- `node_version`

  Version of the `NDB` software program running on this node.

- `process_id`

  This node's process ID

- `angel_process_id`

  Process ID of this node's angel process

- `process_name`

  Name of the executable

- `service_URI`

  Service URI of this node (see text)

Notes

`node_id` is the ID assigned to this node in the cluster.

The `node_type` column displays one of the following three values:

- `MGM`: Management node.
- `NDB`: Data node.
- `API`: API or SQL node.

For an executable shipped with the NDB Cluster distribution, `node_version` shows the software Cluster version string, such as `8.0.34-ndb-8.0.34`.

`process_id` is the node executable's process ID as shown by the host operating system using a process display application such as `top` on Linux, or the Task Manager on Windows platforms.

`angel_process_id` is the system process ID for the node's angel process, which ensures that a data node or SQL is automatically restarted in cases of failures. For management nodes and API nodes other than SQL nodes, the value of this column is `NULL`.

The `process_name` column shows the name of the running executable. For management nodes, this is `ndb_mgmd`. For data nodes, this is `ndbd` (single-threaded) or `ndbmtd` (multithreaded). For SQL nodes, this is `mysqld`. For other types of API nodes, it is the name of the executable program connected to the cluster; NDB API applications can set a custom value for this using `Ndb_cluster_connection::set_name()`.

`service_URI` shows the service network address. For management nodes and data nodes, the scheme used is `ndb://`. For SQL nodes, this is `mysql://`. By default, API nodes other than SQL nodes use `ndb://` for the scheme; NDB API applications can set this to a custom value using `Ndb_cluster_connection::set_service_uri()`. regardless of the node type, the scheme is followed by the IP address used by the NDB transporter for the node in question. For management nodes and SQL nodes, this address includes the port number (usually 1186 for management nodes and 3306 for SQL nodes). If the SQL node was started with the `bind_address` system variable set, this address is used instead of the transporter address, unless the bind address is set to `*`, `0.0.0.0`, or `::`.

Additional path information may be included in the `service_URI` value for an SQL node reflecting various configuration options. For example, `mysql://198.51.100.3/tmp/mysql.sock` indicates that the SQL node was started with the `skip_networking` system variable enabled, and `mysql://198.51.100.3:3306/?server-id=1` shows that replication is enabled for this SQL node.

23.6.16.51 The ndbinfo resources Table

This table provides information about data node resource availability and usage.

These resources are sometimes known as *super-pools*.

The `resources` table contains the following columns:

- `node_id`

  The unique node ID of this data node.

- `resource_name`

  Name of the resource; see text.

- `reserved`

  The amount reserved for this resource, as a number of 32KB pages.

- `used`

  The amount actually used by this resource, as a number of 32KB pages.

- `max`

  The maximum amount (number of 32KB pages) of this resource used, since the node was last started.

Notes

The `resource_name` can be any one of the names shown in the following table:

- `RESERVED`: Reserved by the system; cannot be overridden.
- `TRANSACTION_MEMORY`: Memory allocated for transactions on this data node. In NDB 8.0.19 and later this can be controlled using the `TransactionMemory` configuration parameter.
- `DISK_OPERATIONS`: If a log file group is allocated, the size of the undo log buffer is used to set the size of this resource. This resource is used only to allocate the undo log buffer for an undo log file group; there can only be one such group. Overallocation occurs as needed by `CREATE LOGFILE GROUP`.
- `DISK_RECORDS`: Records allocated for Disk Data operations.
- `DATA_MEMORY`: Used for main memory tuples, indexes, and hash indexes. Sum of DataMemory and IndexMemory, plus 8 pages of 32 KB each if IndexMemory has been set. Cannot be overallocated.
- `JOBBUFFER`: Used for allocating job buffers by the NDB scheduler; cannot be overallocated. This is approximately 2 MB per thread plus a 1 MB buffer in both directions for all threads that can communicate. For large configurations this consume several GB.
- `FILE_BUFFERS`: Used by the redo log handler in the `DBLQH` kernel block; cannot be overallocated. Size is `NoOfFragmentLogParts` * `RedoBuffer`, plus 1 MB per log file part.
- `TRANSPORTER_BUFFERS`: Used for send buffers by `ndbmtd`; the sum of `TotalSendBufferMemory` and `ExtraSendBufferMemory`. This resource that can be overallocated by up to 25 percent. `TotalSendBufferMemory` is calculated by summing the send buffer memory per node, the default value of which is 2 MB. Thus, in a system having four data nodes and eight API nodes, the data nodes have 12 * 2 MB send buffer memory. `ExtraSendBufferMemory` is used by `ndbmtd` and amounts to 2 MB extra memory per thread. Thus, with 4 LDM threads, 2 TC threads, 1 main thread, 1 replication thread, and 2 receive threads, `ExtraSendBufferMemory` is 10 * 2 MB. Overallocation of this resource can be performed by setting the `SharedGlobalMemory` data node configuration parameter.
- `DISK_PAGE_BUFFER`: Used for the disk page buffer; determined by the `DiskPageBufferMemory` configuration parameter. Cannot be overallocated.
- `QUERY_MEMORY`: Used by the `DBSPJ` kernel block.
- `SCHEMA_TRANS_MEMORY`: Minimum is 2 MB; can be overallocated to use any remaining available memory.

23.6.16.52 The ndbinfo restart_info Table

The `restart_info` table contains information about node restart operations. Each entry in the table corresponds to a node restart status report in real time from a data node with the given node ID. Only the most recent report for any given node is shown.

The `restart_info` table contains the following columns:

- `node_id`

Node ID in the cluster

- `node_restart_status`

  Node status; see text for values. Each of these corresponds to a possible value of `node_restart_status_int`.

- `node_restart_status_int`

  Node status code; see text for values.

- `secs_to_complete_node_failure`

  Time in seconds to complete node failure handling

- `secs_to_allocate_node_id`

  Time in seconds from node failure completion to allocation of node ID

- `secs_to_include_in_heartbeat_protocol`

  Time in seconds from allocation of node ID to inclusion in heartbeat protocol

- `secs_until_wait_for_ndbcntr_master`

  Time in seconds from being included in heartbeat protocol until waiting for `NDBCNTR` master began

- `secs_wait_for_ndbcntr_master`

  Time in seconds spent waiting to be accepted by `NDBCNTR` master for starting

- `secs_to_get_start_permitted`

  Time in seconds elapsed from receiving of permission for start from master until all nodes have accepted start of this node

- `secs_to_wait_for_lcp_for_copy_meta_data`

  Time in seconds spent waiting for LCP completion before copying metadata

- `secs_to_copy_meta_data`

  Time in seconds required to copy metadata from master to newly starting node

- `secs_to_include_node`

  Time in seconds waited for GCP and inclusion of all nodes into protocols

- `secs_starting_node_to_request_local_recovery`

  Time in seconds that the node just starting spent waiting to request local recovery

- `secs_for_local_recovery`

  Time in seconds required for local recovery by node just starting

- `secs_restore_fragments`

  Time in seconds required to restore fragments from LCP files

- `secs_undo_disk_data`

  Time in seconds required to execute undo log on disk data part of records

- `secs_exec_redo_log`

  Time in seconds required to execute redo log on all restored fragments

- `secs_index_rebuild`

  Time in seconds required to rebuild indexes on restored fragments

- `secs_to_synchronize_starting_node`

  Time in seconds required to synchronize starting node from live nodes

- `secs_wait_lcp_for_restart`

  Time in seconds required for LCP start and completion before restart was completed

- `secs_wait_subscription_handover`

  Time in seconds spent waiting for handover of replication subscriptions

- `total_restart_secs`

  Total number of seconds from node failure until node is started again

Notes

The following list contains values defined for the `node_restart_status_int` column with their internal status names (in parentheses), and the corresponding messages shown in the `node_restart_status` column:

- `0` (`ALLOCATED_NODE_ID`)

  `Allocated node id`

- `1` (`INCLUDED_IN_HB_PROTOCOL`)

  `Included in heartbeat protocol`

- `2` (`NDBCNTR_START_WAIT`)

  `Wait for NDBCNTR master to permit us to start`

- `3` (`NDBCNTR_STARTED`)

  `NDBCNTR master permitted us to start`

- `4` (`START_PERMITTED`)

  `All nodes permitted us to start`

- `5` (`WAIT_LCP_TO_COPY_DICT`)

  `Wait for LCP completion to start copying metadata`

- `6` (`COPY_DICT_TO_STARTING_NODE`)

  `Copying metadata to starting node`

- `7` (`INCLUDE_NODE_IN_LCP_AND_GCP`)

  `Include node in LCP and GCP protocols`

- `8` (`LOCAL_RECOVERY_STARTED`)

  `Restore fragments ongoing`

- 9 (COPY_FRAGMENTS_STARTED)

  Synchronizing starting node with live nodes

- 10 (WAIT_LCP_FOR_RESTART)

  Wait for LCP to ensure durability

- 11 (WAIT_SUMA_HANDOVER)

  Wait for handover of subscriptions

- 12 (RESTART_COMPLETED)

  Restart completed

- 13 (NODE_FAILED)

  Node failed, failure handling in progress

- 14 (NODE_FAILURE_COMPLETED)

  Node failure handling completed

- 15 (NODE_GETTING_PERMIT)

  All nodes permitted us to start

- 16 (NODE_GETTING_INCLUDED)

  Include node in LCP and GCP protocols

- 17 (NODE_GETTING_SYNCHED)

  Synchronizing starting node with live nodes

- 18 (NODE_GETTING_LCP_WAITED)

  [none]

- 19 (NODE_ACTIVE)

  Restart completed

- 20 (NOT_DEFINED_IN_CLUSTER)

  [none]

- 21 (NODE_NOT_RESTARTED_YET)

  Initial state

Status numbers 0 through 12 apply on master nodes only; the remainder of those shown in the table apply to all restarting data nodes. Status numbers 13 and 14 define node failure states; 20 and 21 occur when no information about the restart of a given node is available.

See also Section 23.6.4, “Summary of NDB Cluster Start Phases”.

23.6.16.53 The ndbinfo server_locks Table

The server_locks table is similar in structure to the cluster_locks table, and provides a subset of the information found in the latter table, but which is specific to the SQL node (MySQL server) where it resides. (The cluster_locks table provides information about all locks in the cluster.) More precisely, server_locks contains information about locks requested by threads belonging to the

current `mysqld` instance, and serves as a companion table to `server_operations`. This may be useful for correlating locking patterns with specific MySQL user sessions, queries, or use cases.

The `server_locks` table contains the following columns:

- `mysql_connection_id`

  MySQL connection ID

- `node_id`

  ID of reporting node

- `block_instance`

  ID of reporting LDM instance

- `tableid`

  ID of table containing this row

- `fragmentid`

  ID of fragment containing locked row

- `rowid`

  ID of locked row

- `transid`

  Transaction ID

- `mode`

  Lock request mode

- `state`

  Lock state

- `detail`

  Whether this is first holding lock in row lock queue

- `op`

  Operation type

- `duration_millis`

  Milliseconds spent waiting or holding lock

- `lock_num`

  ID of lock object

- `waiting_for`

  Waiting for lock with this ID

Notes

The `mysql_connection_id` column shows the MySQL connection or thread ID as shown by `SHOW PROCESSLIST`.

`block_instance` refers to an instance of a kernel block. Together with the block name, this number can be used to look up a given instance in the `threadblocks` table.

The `tableid` is assigned to the table by `NDB`; the same ID is used for this table in other `ndbinfo` tables, as well as in the output of `ndb_show_tables`.

The transaction ID shown in the `transid` column is the identifier generated by the NDB API for the transaction requesting or holding the current lock.

The `mode` column shows the lock mode, which is always one of `S` (shared lock) or `X` (exclusive lock). If a transaction has an exclusive lock on a given row, all other locks on that row have the same transaction ID.

The `state` column shows the lock state. Its value is always one of `H` (holding) or `W` (waiting). A waiting lock request waits for a lock held by a different transaction.

The `detail` column indicates whether this lock is the first holding lock in the affected row's lock queue, in which case it contains a `*` (asterisk character); otherwise, this column is empty. This information can be used to help identify the unique entries in a list of lock requests.

The `op` column shows the type of operation requesting the lock. This is always one of the values `READ`, `INSERT`, `UPDATE`, `DELETE`, `SCAN`, or `REFRESH`.

The `duration_millis` column shows the number of milliseconds for which this lock request has been waiting or holding the lock. This is reset to 0 when a lock is granted for a waiting request.

The lock ID (`lockid` column) is unique to this node and block instance.

If the `lock_state` column's value is `W`, this lock is waiting to be granted, and the `waiting_for` column shows the lock ID of the lock object this request is waiting for. Otherwise, `waiting_for` is empty. `waiting_for` can refer only to locks on the same row (as identified by `node_id`, `block_instance`, `tableid`, `fragmentid`, and `rowid`).

23.6.16.54 The ndbinfo server_operations Table

The `server_operations` table contains entries for all ongoing `NDB` operations that the current SQL node (MySQL Server) is currently involved in. It effectively is a subset of the `cluster_operations` table, in which operations for other SQL and API nodes are not shown.

The `server_operations` table contains the following columns:

- `mysql_connection_id`

  MySQL Server connection ID

- `node_id`

  Node ID

- `block_instance`

  Block instance

- `transid`

  Transaction ID

- `operation_type`

  Operation type (see text for possible values)

- `state`

  Operation state (see text for possible values)

- `tableid`

  Table ID

- `fragmentid`

  Fragment ID

- `client_node_id`

  Client node ID

- `client_block_ref`

  Client block reference

- `tc_node_id`

  Transaction coordinator node ID

- `tc_block_no`

  Transaction coordinator block number

- `tc_block_instance`

  Transaction coordinator block instance

Notes

The `mysql_connection_id` is the same as the connection or session ID shown in the output of `SHOW PROCESSLIST`. It is obtained from the `INFORMATION_SCHEMA` table `NDB_TRANSID_MYSQL_CONNECTION_MAP`.

`block_instance` refers to an instance of a kernel block. Together with the block name, this number can be used to look up a given instance in the `threadblocks` table.

The transaction ID (`transid`) is a unique 64-bit number which can be obtained using the NDB API's `getTransactionId()` method. (Currently, the MySQL Server does not expose the NDB API transaction ID of an ongoing transaction.)

The `operation_type` column can take any one of the values `READ`, `READ-SH`, `READ-EX`, `INSERT`, `UPDATE`, `DELETE`, `WRITE`, `UNLOCK`, `REFRESH`, `SCAN`, `SCAN-SH`, `SCAN-EX`, or `<unknown>`.

The `state` column can have any one of the values `ABORT_QUEUED`, `ABORT_STOPPED`, `COMMITTED`, `COMMIT_QUEUED`, `COMMIT_STOPPED`, `COPY_CLOSE_STOPPED`, `COPY_FIRST_STOPPED`, `COPY_STOPPED`, `COPY_TUPKEY`, `IDLE`, `LOG_ABORT_QUEUED`, `LOG_COMMIT_QUEUED`, `LOG_COMMIT_QUEUED_WAIT_SIGNAL`, `LOG_COMMIT_WRITTEN`, `LOG_COMMIT_WRITTEN_WAIT_SIGNAL`, `LOG_QUEUED`, `PREPARED`, `PREPARED_RECEIVED_COMMIT`, `SCAN_CHECK_STOPPED`, `SCAN_CLOSE_STOPPED`, `SCAN_FIRST_STOPPED`, `SCAN_RELEASE_STOPPED`, `SCAN_STATE_USED`, `SCAN_STOPPED`, `SCAN_TUPKEY`, `STOPPED`, `TC_NOT_CONNECTED`, `WAIT_ACC`, `WAIT_ACC_ABORT`, `WAIT_AI_AFTER_ABORT`, `WAIT_ATTR`, `WAIT_SCAN_AI`, `WAIT_TUP`, `WAIT_TUPKEYINFO`, `WAIT_TUP_COMMIT`, or `WAIT_TUP_TO_ABORT`. (If the MySQL Server is running with `ndbinfo_show_hidden` enabled, you can view this list of states by selecting from the `ndb$dblqh_tcconnect_state` table, which is normally hidden.)

You can obtain the name of an `NDB` table from its table ID by checking the output of `ndb_show_tables`.

The `fragid` is the same as the partition number seen in the output of `ndb_desc --extra-partition-info` (short form `-p`).

In `client_node_id` and `client_block_ref`, `client` refers to an NDB Cluster API or SQL node (that is, an NDB API client or a MySQL Server attached to the cluster).

The `block_instance` and `tc_block_instance` column provide NDB kernel block instance numbers. You can use these to obtain information about specific threads from the `threadblocks` table.

23.6.16.55 The ndbinfo server_transactions Table

The `server_transactions` table is subset of the `cluster_transactions` table, but includes only those transactions in which the current SQL node (MySQL Server) is a participant, while including the relevant connection IDs.

The `server_transactions` table contains the following columns:

- `mysql_connection_id`

  MySQL Server connection ID

- `node_id`

  Transaction coordinator node ID

- `block_instance`

  Transaction coordinator block instance

- `transid`

  Transaction ID

- `state`

  Operation state (see text for possible values)

- `count_operations`

  Number of stateful operations in the transaction

- `outstanding_operations`

  Operations still being executed by local data management layer (LQH blocks)

- `inactive_seconds`

  Time spent waiting for API

- `client_node_id`

  Client node ID

- `client_block_ref`

  Client block reference

Notes

The `mysql_connection_id` is the same as the connection or session ID shown in the output of `SHOW PROCESSLIST`. It is obtained from the `INFORMATION_SCHEMA` table `NDB_TRANSID_MYSQL_CONNECTION_MAP`.

`block_instance` refers to an instance of a kernel block. Together with the block name, this number can be used to look up a given instance in the `threadblocks` table.

The transaction ID (`transid`) is a unique 64-bit number which can be obtained using the NDB API's `getTransactionId()` method. (Currently, the MySQL Server does not expose the NDB API transaction ID of an ongoing transaction.)

The `state` column can have any one of the values `CS_ABORTING`, `CS_COMMITTING`, `CS_COMMIT_SENT`, `CS_COMPLETE_SENT`, `CS_COMPLETING`, `CS_CONNECTED`, `CS_DISCONNECTED`, `CS_FAIL_ABORTED`, `CS_FAIL_ABORTING`, `CS_FAIL_COMMITTED`, `CS_FAIL_COMMITTING`, `CS_FAIL_COMPLETED`, `CS_FAIL_PREPARED`, `CS_PREPARE_TO_COMMIT`, `CS_RECEIVING`, `CS_REC_COMMITTING`, `CS_RESTART`, `CS_SEND_FIRE_TRIG_REQ`, `CS_STARTED`, `CS_START_COMMITTING`, `CS_START_SCAN`, `CS_WAIT_ABORT_CONF`, `CS_WAIT_COMMIT_CONF`, `CS_WAIT_COMPLETE_CONF`, `CS_WAIT_FIRE_TRIG_REQ`. (If the MySQL Server is running with `ndbinfo_show_hidden` enabled, you can view this list of states by selecting from the `ndb$dbtc_apiconnect_state` table, which is normally hidden.)

In `client_node_id` and `client_block_ref`, `client` refers to an NDB Cluster API or SQL node (that is, an NDB API client or a MySQL Server attached to the cluster).

The `block_instance` column provides the `DBTC` kernel block instance number. You can use this to obtain information about specific threads from the `threadblocks` table.

23.6.16.56 The ndbinfo table_distribution_status Table

The `table_distribution_status` table provides information about the progress of table distribution for `NDB` tables.

The `table_distribution_status` table contains the following columns:

- `node_id`

  Node id

- `table_id`

  Table ID

- `tab_copy_status`

  Status of copying of table distribution data to disk; one of `IDLE`, `SR_PHASE1_READ_PAGES`, `SR_PHASE2_READ_TABLE`, `SR_PHASE3_COPY_TABLE`, `REMOVE_NODE`, `LCP_READ_TABLE`, `COPY_TAB_REQ`, `COPY_NODE_STATE`, `ADD_TABLE_COORDINATOR` (*prior to NDB 8.0.23*: `ADD_TABLE_MASTER`), `ADD_TABLE_PARTICIPANT` (*prior to NDB 8.0.23*: `ADD_TABLE_SLAVE`), `INVALIDATE_NODE_LCP`, `ALTER_TABLE`, `COPY_TO_SAVE`, or `GET_TABINFO`

- `tab_update_status`

  Status of updating of table distribution data; one of `IDLE`, `LOCAL_CHECKPOINT`, `LOCAL_CHECKPOINT_QUEUED`, `REMOVE_NODE`, `COPY_TAB_REQ`, `ADD_TABLE_MASTER`, `ADD_TABLE_SLAVE`, `INVALIDATE_NODE_LCP`, or `CALLBACK`

- `tab_lcp_status`

  Status of table LCP; one of `ACTIVE` (waiting for local checkpoint to be performed), `WRITING_TO_FILE` (checkpoint performed but not yet written to disk), or `COMPLETED` (checkpoint performed and persisted to disk)

- `tab_status`

  Table internal status; one of `ACTIVE` (table exists), `CREATING` (table is being created), or `DROPPING` (table is being dropped)

- `tab_storage`

Table recoverability; one of `NORMAL` (fully recoverable with redo logging and checkpointing), `NOLOGGING` (recoverable from node crash, empty following cluster crash), or `TEMPORARY` (not recoverable)

- `tab_partitions`

  Number of partitions in table

- `tab_fragments`

  Number of fragments in table; normally same as `tab_partitions`; for fully replicated tables equal to `tab_partitions * [number of node groups]`

- `current_scan_count`

  Current number of active scans

- `scan_count_wait`

  Current number of scans waiting to be performed before `ALTER TABLE` can complete.

- `is_reorg_ongoing`

  Whether the table is currently being reorganized (1 if true)

23.6.16.57 The ndbinfo table_fragments Table

The `table_fragments` table provides information about the fragmentation, partitioning, distribution, and (internal) replication of `NDB` tables.

The `table_fragments` table contains the following columns:

- `node_id`

  Node ID (`DIH` master)

- `table_id`

  Table ID

- `partition_id`

  Partition ID

- `fragment_id`

  Fragment ID (same as partition ID unless table is fully replicated)

- `partition_order`

  Order of fragment in partition

- `log_part_id`

  Log part ID of fragment

- `no_of_replicas`

  Number of fragment replicas

- `current_primary`

  Current primary node ID

- `preferred_primary`

  Preferred primary node ID

- `current_first_backup`

  Current first backup node ID

- `current_second_backup`

  Current second backup node ID

- `current_third_backup`

  Current third backup node ID

- `num_alive_replicas`

  Current number of live fragment replicas

- `num_dead_replicas`

  Current number of dead fragment replicas

- `num_lcp_replicas`

  Number of fragment replicas remaining to be checkpointed

23.6.16.58 The ndbinfo table_info Table

The `table_info` table provides information about logging, checkpointing, distribution, and storage options in effect for individual `NDB` tables.

The `table_info` table contains the following columns:

- `table_id`

  Table ID

- `logged_table`

  Whether table is logged (1) or not (0)

- `row_contains_gci`

  Whether table rows contain GCI (1 true, 0 false)

- `row_contains_checksum`

  Whether table rows contain checksum (1 true, 0 false)

- `read_backup`

  If backup fragment replicas are read this is 1, otherwise 0

- `fully_replicated`

  If table is fully replicated this is 1, otherwise 0

- `storage_type`

  Table storage type; one of `MEMORY` or `DISK`

- `hashmap_id`

Hashmap ID

- `partition_balance`

  Partition balance (fragment count type) used for table; one of `FOR_RP_BY_NODE`, `FOR_RA_BY_NODE`, `FOR_RP_BY_LDM`, or `FOR_RA_BY_LDM`

- `create_gci`

  GCI in which table was created

23.6.16.59 The ndbinfo table_replicas Table

The `table_replicas` table provides information about the copying, distribution, and checkpointing of `NDB` table fragments and fragment replicas.

The `table_replicas` table contains the following columns:

- `node_id`

  ID of the node from which data is fetched (`DIH` master)

- `table_id`

  Table ID

- `fragment_id`

  Fragment ID

- `initial_gci`

  Initial GCI for table

- `replica_node_id`

  ID of node where fragment replica is stored

- `is_lcp_ongoing`

  Is 1 if LCP is ongoing on this fragment, 0 otherwise

- `num_crashed_replicas`

  Number of crashed fragment replica instances

- `last_max_gci_started`

  Highest GCI started in most recent LCP

- `last_max_gci_completed`

  Highest GCI completed in most recent LCP

- `last_lcp_id`

  ID of most recent LCP

- `prev_lcp_id`

  ID of previous LCP

- `prev_max_gci_started`

Highest GCI started in previous LCP

- `prev_max_gci_completed`

  Highest GCI completed in previous LCP

- `last_create_gci`

  Last Create GCI of last crashed fragment replica instance

- `last_replica_gci`

  Last GCI of last crashed fragment replica instance

- `is_replica_alive`

  1 if this fragment replica is alive, 0 otherwise

23.6.16.60 The ndbinfo tc_time_track_stats Table

The `tc_time_track_stats` table provides time-tracking information obtained from the `DBTC` block (TC) instances in the data nodes, through API nodes access `NDB`. Each TC instance tracks latencies for a set of activities it undertakes on behalf of API nodes or other data nodes; these activities include transactions, transaction errors, key reads, key writes, unique index operations, failed key operations of any type, scans, failed scans, fragment scans, and failed fragment scans.

A set of counters is maintained for each activity, each counter covering a range of latencies less than or equal to an upper bound. At the conclusion of each activity, its latency is determined and the appropriate counter incremented. `tc_time_track_stats` presents this information as rows, with a row for each instance of the following:

- Data node, using its ID
- TC block instance
- Other communicating data node or API node, using its ID
- Upper bound value

Each row contains a value for each activity type. This is the number of times that this activity occurred with a latency within the range specified by the row (that is, where the latency does not exceed the upper bound).

The `tc_time_track_stats` table contains the following columns:

- `node_id`

  Requesting node ID

- `block_number`

  TC block number

- `block_instance`

  TC block instance number

- `comm_node_id`

  Node ID of communicating API or data node

- `upper_bound`

Upper bound of interval (in microseconds)

- `scans`

  Based on duration of successful scans from opening to closing, tracked against the API or data nodes requesting them.

- `scan_errors`

  Based on duration of failed scans from opening to closing, tracked against the API or data nodes requesting them.

- `scan_fragments`

  Based on duration of successful fragment scans from opening to closing, tracked against the data nodes executing them

- `scan_fragment_errors`

  Based on duration of failed fragment scans from opening to closing, tracked against the data nodes executing them

- `transactions`

  Based on duration of successful transactions from beginning until sending of commit `ACK`, tracked against the API or data nodes requesting them. Stateless transactions are not included.

- `transaction_errors`

  Based on duration of failing transactions from start to point of failure, tracked against the API or data nodes requesting them.

- `read_key_ops`

  Based on duration of successful primary key reads with locks. Tracked against both the API or data node requesting them and the data node executing them.

- `write_key_ops`

  Based on duration of successful primary key writes, tracked against both the API or data node requesting them and the data node executing them.

- `index_key_ops`

  Based on duration of successful unique index key operations, tracked against both the API or data node requesting them and the data node executing reads of base tables.

- `key_op_errors`

  Based on duration of all unsuccessful key read or write operations, tracked against both the API or data node requesting them and the data node executing them.

Notes

The `block_instance` column provides the `DBTC` kernel block instance number. You can use this together with the block name to obtain information about specific threads from the `threadblocks` table.

23.6.16.61 The ndbinfo threadblocks Table

The `threadblocks` table associates data nodes, threads, and instances of `NDB` kernel blocks.

The `threadblocks` table contains the following columns:

- `node_id`

  Node ID

- `thr_no`

  Thread ID

- `block_name`

  Block name

- `block_instance`

  Block instance number

Notes

The value of the `block_name` in this table is one of the values found in the `block_name` column when selecting from the `ndbinfo.blocks` table. Although the list of possible values is static for a given NDB Cluster release, the list may vary between releases.

The `block_instance` column provides the kernel block instance number.

23.6.16.62 The ndbinfo threads Table

The `threads` table provides information about threads running in the `NDB` kernel.

The `threads` table contains the following columns:

- `node_id`

  ID of the node where the thread is running

- `thr_no`

  Thread ID (specific to this node)

- `thread_name`

  Thread name (type of thread)

- `thread_description`

  Thread (type) description

Notes

Sample output from a 2-node example cluster, including thread descriptions, is shown here:

```
mysql> SELECT * FROM threads;
+---------+--------+-------------+-------------------------------------------------------------------+
| node_id | thr_no | thread_name | thread_description                                                |
+---------+--------+-------------+-------------------------------------------------------------------+
|       5 |      0 | main        | main thread, schema and distribution handling                     |
|       5 |      1 | rep         | rep thread, asynch replication and proxy block handling           |
|       5 |      2 | ldm         | ldm thread, handling a set of data partitions                     |
|       5 |      3 | recv        | receive thread, performing receive and polling for new receives   |
|       6 |      0 | main        | main thread, schema and distribution handling                     |
|       6 |      1 | rep         | rep thread, asynch replication and proxy block handling           |
|       6 |      2 | ldm         | ldm thread, handling a set of data partitions                     |
|       6 |      3 | recv        | receive thread, performing receive and polling for new receives   |
+---------+--------+-------------+-------------------------------------------------------------------+
8 rows in set (0.01 sec)
```

NDB 8.0.23 introduces the possibility to set either of the `ThreadConfig` arguments `main` or `rep` to 0 while keeping the other at 1, in which case the thread name is `main_rep` and its description is `main and rep thread, schema, distribution, proxy block and asynch replication handling`. It is also possible beginning with NDB 8.0.23 to set both `main` and `rep` to 0, in which case the name of the resulting thread is shown in this table as `main_rep_recv`, and its description is `main, rep and recv thread, schema, distribution, proxy block and asynch replication handling and handling receive and polling for new receives`.

23.6.16.63 The ndbinfo threadstat Table

The `threadstat` table provides a rough snapshot of statistics for threads running in the `NDB` kernel.

The `threadstat` table contains the following columns:

- `node_id`

  Node ID

- `thr_no`

  Thread ID

- `thr_nm`

  Thread name

- `c_loop`

  Number of loops in main loop

- `c_exec`

  Number of signals executed

- `c_wait`

  Number of times waiting for additional input

- `c_l_sent_prioa`

  Number of priority A signals sent to own node

- `c_l_sent_priob`

  Number of priority B signals sent to own node

- `c_r_sent_prioa`

  Number of priority A signals sent to remote node

- `c_r_sent_priob`

  Number of priority B signals sent to remote node

- `os_tid`

  OS thread ID

- `os_now`

  OS time (ms)

- `os_ru_utime`

OS user CPU time (µs)

- `os_ru_stime`

  OS system CPU time (µs)

- `os_ru_minflt`

  OS page reclaims (soft page faults)

- `os_ru_majflt`

  OS page faults (hard page faults)

- `os_ru_nvcsw`

  OS voluntary context switches

- `os_ru_nivcsw`

  OS involuntary context switches

Notes

`os_time` uses the system `gettimeofday()` call.

The values of the `os_ru_utime`, `os_ru_stime`, `os_ru_minflt`, `os_ru_majflt`, `os_ru_nvcsw`, and `os_ru_nivcsw` columns are obtained using the system `getrusage()` call, or the equivalent.

Since this table contains counts taken at a given point in time, for best results it is necessary to query this table periodically and store the results in an intermediate table or tables. The MySQL Server's Event Scheduler can be employed to automate such monitoring. For more information, see Section 25.4, “Using the Event Scheduler”.

23.6.16.64 The ndbinfo transporters Table

This table contains information about NDB transporters.

The `transporters` table contains the following columns:

- `node_id`

  This data node's unique node ID in the cluster

- `remote_node_id`

  The remote data node's node ID

- `status`

  Status of the connection

- `remote_address`

  Name or IP address of the remote host

- `bytes_sent`

  Number of bytes sent using this connection

- `bytes_received`

  Number of bytes received using this connection

- `connect_count`

  Number of times connection established on this transporter

- `overloaded`

  1 if this transporter is currently overloaded, otherwise 0

- `overload_count`

  Number of times this transporter has entered overload state since connecting

- `slowdown`

  1 if this transporter is in slowdown state, otherwise 0

- `slowdown_count`

  Number of times this transporter has entered slowdown state since connecting

Notes

For each running data node in the cluster, the `transporters` table displays a row showing the status of each of that node's connections with all nodes in the cluster, *including itself*. This information is shown in the table's *status* column, which can have any one of the following values: `CONNECTING`, `CONNECTED`, `DISCONNECTING`, or `DISCONNECTED`.

Connections to API and management nodes which are configured but not currently connected to the cluster are shown with status `DISCONNECTED`. Rows where the `node_id` is that of a data node which is not currently connected are not shown in this table. (This is similar omission of disconnected nodes in the `ndbinfo.nodes` table.

The `remote_address` is the host name or address for the node whose ID is shown in the `remote_node_id` column. The `bytes_sent` from this node and `bytes_received` by this node are the numbers, respectively, of bytes sent and received by the node using this connection since it was established. For nodes whose status is `CONNECTING` or `DISCONNECTED`, these columns always display `0`.

Assume you have a 5-node cluster consisting of 2 data nodes, 2 SQL nodes, and 1 management node, as shown in the output of the `SHOW` command in the `ndb_mgm` client:

```
ndb_mgm> SHOW
Connected to Management Server at: localhost:1186
Cluster Configuration
---------------------
[ndbd(NDB)]     2 node(s)
id=1    @10.100.10.1  (8.0.34-ndb-8.0.34, Nodegroup: 0, *)
id=2    @10.100.10.2  (8.0.34-ndb-8.0.34, Nodegroup: 0)

[ndb_mgmd(MGM)] 1 node(s)
id=10   @10.100.10.10  (8.0.34-ndb-8.0.34)

[mysqld(API)]   2 node(s)
id=20   @10.100.10.20  (8.0.34-ndb-8.0.34)
id=21   @10.100.10.21  (8.0.34-ndb-8.0.34)
```

There are 10 rows in the `transporters` table—5 for the first data node, and 5 for the second—assuming that all data nodes are running, as shown here:

```
mysql> SELECT node_id, remote_node_id, status
    ->   FROM ndbinfo.transporters;
+---------+----------------+---------------+
| node_id | remote_node_id | status        |
+---------+----------------+---------------+
|       1 |              1 | DISCONNECTED  |
|       1 |              2 | CONNECTED     |
```

```
|       1 |             10 | CONNECTED     |
|       1 |             20 | CONNECTED     |
|       1 |             21 | CONNECTED     |
|       2 |              1 | CONNECTED     |
|       2 |              2 | DISCONNECTED  |
|       2 |             10 | CONNECTED     |
|       2 |             20 | CONNECTED     |
|       2 |             21 | CONNECTED     |
+---------+----------------+---------------+
10 rows in set (0.04 sec)
```

If you shut down one of the data nodes in this cluster using the command `2 STOP` in the `ndb_mgm` client, then repeat the previous query (again using the `mysql` client), this table now shows only 5 rows—1 row for each connection from the remaining management node to another node, including both itself and the data node that is currently offline—and displays `CONNECTING` for the status of each remaining connection to the data node that is currently offline, as shown here:

```
mysql> SELECT node_id, remote_node_id, status
    ->   FROM ndbinfo.transporters;
+---------+----------------+---------------+
| node_id | remote_node_id | status        |
+---------+----------------+---------------+
|       1 |              1 | DISCONNECTED  |
|       1 |              2 | CONNECTING    |
|       1 |             10 | CONNECTED     |
|       1 |             20 | CONNECTED     |
|       1 |             21 | CONNECTED     |
+---------+----------------+---------------+
5 rows in set (0.02 sec)
```

The `connect_count`, `overloaded`, `overload_count`, `slowdown`, and `slowdown_count` counters are reset on connection, and retain their values after the remote node disconnects. The `bytes_sent` and `bytes_received` counters are also reset on connection, and so retain their values following disconnection (until the next connection resets them).

The *overload* state referred to by the `overloaded` and `overload_count` columns occurs when this transporter's send buffer contains more than `OVerloadLimit` bytes (default is 80% of `SendBufferMemory`, that is, 0.8 * 2097152 = 1677721 bytes). When a given transporter is in a state of overload, any new transaction that tries to use this transporter fails with Error 1218 (`Send Buffers overloaded in NDB kernel`). This affects both scans and primary key operations.

The *slowdown* state referenced by the `slowdown` and `slowdown_count` columns of this table occurs when the transporter's send buffer contains more than 60% of the overload limit (equal to 0.6 * 2097152 = 1258291 bytes by default). In this state, any new scan using this transporter has its batch size reduced to minimize the load on the transporter.

Common causes of send buffer slowdown or overloading include the following:

- Data size, in particular the quantity of data stored in `TEXT` columns or `BLOB` columns (or both types of columns)
- Having a data node (ndbd or ndbmtd) on the same host as an SQL node that is engaged in binary logging
- Large number of rows per transaction or transaction batch
- Configuration issues such as insufficient `SendBufferMemory`
- Hardware issues such as insufficient RAM or poor network connectivity

See also Section 23.4.3.14, “Configuring NDB Cluster Send Buffer Parameters”.

23.6.17 INFORMATION_SCHEMA Tables for NDB Cluster

Two `INFORMATION_SCHEMA` tables provide information that is of particular use when managing an NDB Cluster. The `FILES` table provides information about NDB Cluster Disk Data files (see

Section 23.6.11.1, “NDB Cluster Disk Data Objects”). The `ndb_transid_mysql_connection_map` table provides a mapping between transactions, transaction coordinators, and API nodes.

Additional statistical and other data about NDB Cluster transactions, operations, threads, blocks, and other aspects of performance can be obtained from the tables in the `ndbinfo` database. For information about these tables, see Section 23.6.16, “ndbinfo: The NDB Cluster Information Database”.

23.6.18 NDB Cluster and the Performance Schema

NDB 8.0 provides information in the MySQL Performance Schema about threads and transaction memory usage; NDB 8.0.29 adds `ndbcluster` plugin threads, and NDB 8.0.30 adds instrumenting for transaction batch memory. These features are described in greater detail in the sections which follow.

ndbcluster Plugin Threads

Beginning with NDB 8.0.29, `ndbcluster` plugin threads are visible in the Performance Schema `threads` table, as shown in the following query:

```
mysql> SELECT name, type, thread_id, thread_os_id
    -> FROM performance_schema.threads
    -> WHERE name LIKE '%ndbcluster%'\G
+---------------------------------+------------+-----------+--------------+
| name                            | type       | thread_id | thread_os_id |
+---------------------------------+------------+-----------+--------------+
| thread/ndbcluster/ndb_binlog    | BACKGROUND |        30 |        11980 |
| thread/ndbcluster/ndb_index_stat | BACKGROUND |        31 |        11981 |
| thread/ndbcluster/ndb_metadata  | BACKGROUND |        32 |        11982 |
+---------------------------------+------------+-----------+--------------+
```

The `threads` table shows all three of the threads listed here:

- `ndb_binlog`: Binary logging thread
- `ndb_index_stat`: Index statistics thread
- `ndb_metadata`: Metadata thread

These threads are also shown by name in the `setup_threads` table.

Thread names are shown in the `name` column of the `threads` and `setup_threads` tables using the format *`prefix/plugin_name/thread_name`*. *`prefix`*, the object type as determined by the `performance_schema` engine, is `thread` for plugin threads (see Thread Instrument Elements). The *`plugin_name`* is `ndbcluster`. *`thread_name`* is the standalone name of the thread (`ndb_binlog`, `ndb_index_stat`, or `ndb_metadata`).

Using the thread ID or OS thread ID for a given thread in the `threads` or `setup_threads` table, it is possible to obtain considerable information from Performance Schema about plugin execution and resource usage. This example shows how to obtain the amount of memory allocated by the threads created by the `ndbcluster` plugin from the `mem_root` arena by joining the `threads` and `memory_summary_by_thread_by_event_name` tables:

```
mysql> SELECT
    ->   t.name,
    ->   m.sum_number_of_bytes_alloc,
    ->   IF(m.sum_number_of_bytes_alloc > 0, "true", "false") AS 'Has allocated memory'
    -> FROM performance_schema.memory_summary_by_thread_by_event_name m
    -> JOIN performance_schema.threads t
    -> ON m.thread_id = t.thread_id
    -> WHERE t.name LIKE '%ndbcluster%'
    ->   AND event_name LIKE '%THD::main_mem_root%';
+---------------------------------+-------------------------+----------------------+
| name                            | sum_number_of_bytes_alloc | Has allocated memory |
+---------------------------------+-------------------------+----------------------+
| thread/ndbcluster/ndb_binlog    |                   20576 | true                 |
| thread/ndbcluster/ndb_index_stat |                       0 | false                |
| thread/ndbcluster/ndb_metadata  |                    8240 | true                 |
```

```
+---------------------------------+--------------------------+---------------------+
```

Transaction Memory Usage

Starting with NDB 8.0.30, you can see the amount of memory used for transaction batching by querying the Performance Schema `memory_summary_by_thread_by_event_name` table, similar to what is shown here:

```
mysql> SELECT EVENT_NAME
    ->   FROM performance_schema.memory_summary_by_thread_by_event_name
    ->   WHERE THREAD_ID = PS_CURRENT_THREAD_ID()
    ->     AND EVENT_NAME LIKE 'memory/ndbcluster/%';
+----------------------------------------+
| EVENT_NAME                             |
+----------------------------------------+
| memory/ndbcluster/Thd_ndb::batch_mem_root |
+----------------------------------------+
1 row in set (0.01 sec)
```

The `ndbcluster` transaction memory instrument is also visible in the Performance Schema `setup_instruments` table, as shown here:

```
mysql> SELECT * from performance_schema.setup_instruments
    ->   WHERE NAME LIKE '%ndb%'\G
*************************** 1. row ***************************
         NAME: memory/ndbcluster/Thd_ndb::batch_mem_root
      ENABLED: YES
        TIMED: NULL
   PROPERTIES:
   VOLATILITY: 0
DOCUMENTATION: Memory used for transaction batching
1 row in set (0.01 sec)
```

23.6.19 Quick Reference: NDB Cluster SQL Statements

This section discusses several SQL statements that can prove useful in managing and monitoring a MySQL server that is connected to an NDB Cluster, and in some cases provide information about the cluster itself.

- `SHOW ENGINE NDB STATUS`, `SHOW ENGINE NDBCLUSTER STATUS`

  The output of this statement contains information about the server's connection to the cluster, creation and usage of NDB Cluster objects, and binary logging for NDB Cluster replication.

  See Section 13.7.7.15, “SHOW ENGINE Statement”, for a usage example and more detailed information.

- `SHOW ENGINES`

  This statement can be used to determine whether or not clustering support is enabled in the MySQL server, and if so, whether it is active.

  See Section 13.7.7.16, “SHOW ENGINES Statement”, for more detailed information.

  Note

  This statement does not support a `LIKE` clause. However, you can use `LIKE` to filter queries against the Information Schema `ENGINES` table, as discussed in the next item.

- `SELECT * FROM INFORMATION_SCHEMA.ENGINES [WHERE ENGINE LIKE 'NDB%']`

  This is the equivalent of `SHOW ENGINES`, but uses the `ENGINES` table of the `INFORMATION_SCHEMA` database. Unlike the case with the `SHOW ENGINES` statement, it is possible to filter the results using a `LIKE` clause, and to select specific columns to obtain information that may

be of use in scripts. For example, the following query shows whether the server was built with NDB support and, if so, whether it is enabled:

```
mysql> SELECT ENGINE, SUPPORT FROM INFORMATION_SCHEMA.ENGINES
    ->   WHERE ENGINE LIKE 'NDB%';
+------------+---------+
| ENGINE     | SUPPORT |
+------------+---------+
| ndbcluster | YES     |
| ndbinfo    | YES     |
+------------+---------+
```

If NDB support is not enabled, the preceding query returns an empty set. See Section 26.3.13, “The INFORMATION_SCHEMA ENGINES Table”, for more information.

- `SHOW VARIABLES LIKE 'NDB%'`

  This statement provides a list of most server system variables relating to the NDB storage engine, and their values, as shown here:

```
mysql> SHOW VARIABLES LIKE 'NDB%';
+-----------------------------------+----------------------------------------+
| Variable_name                     | Value                                  |
+-----------------------------------+----------------------------------------+
| ndb_allow_copying_alter_table     | ON                                     |
| ndb_autoincrement_prefetch_sz     | 512                                    |
| ndb_batch_size                    | 32768                                  |
| ndb_blob_read_batch_bytes         | 65536                                  |
| ndb_blob_write_batch_bytes        | 65536                                  |
| ndb_clear_apply_status            | ON                                     |
| ndb_cluster_connection_pool       | 1                                      |
| ndb_cluster_connection_pool_nodeids |                                      |
| ndb_connectstring                 | 127.0.0.1                              |
| ndb_data_node_neighbour           | 0                                      |
| ndb_default_column_format         | FIXED                                  |
| ndb_deferred_constraints          | 0                                      |
| ndb_distribution                  | KEYHASH                                |
| ndb_eventbuffer_free_percent      | 20                                     |
| ndb_eventbuffer_max_alloc         | 0                                      |
| ndb_extra_logging                 | 1                                      |
| ndb_force_send                    | ON                                     |
| ndb_fully_replicated              | OFF                                    |
| ndb_index_stat_enable             | ON                                     |
| ndb_index_stat_option             | loop_enable=1000ms,loop_idle=1000ms,
loop_busy=100ms,update_batch=1,read_batch=4,idle_batch=32,check_batch=8,
check_delay=10m,delete_batch=8,clean_delay=1m,error_batch=4,error_delay=1m,
evict_batch=8,evict_delay=1m,cache_limit=32M,cache_lowpct=90,zero_total=0 |
| ndb_join_pushdown                 | ON                                     |
| ndb_log_apply_status              | OFF                                    |
| ndb_log_bin                       | OFF                                    |
| ndb_log_binlog_index              | ON                                     |
| ndb_log_empty_epochs              | OFF                                    |
| ndb_log_empty_update              | OFF                                    |
| ndb_log_exclusive_reads           | OFF                                    |
| ndb_log_orig                      | OFF                                    |
| ndb_log_transaction_id            | OFF                                    |
| ndb_log_update_as_write           | ON                                     |
| ndb_log_update_minimal            | OFF                                    |
| ndb_log_updated_only              | ON                                     |
| ndb_metadata_check                | ON                                     |
| ndb_metadata_check_interval       | 60                                     |
| ndb_metadata_sync                 | OFF                                    |
| ndb_mgmd_host                     | 127.0.0.1                              |
| ndb_nodeid                        | 0                                      |
| ndb_optimization_delay            | 10                                     |
| ndb_optimized_node_selection      | 3                                      |
| ndb_read_backup                   | ON                                     |
| ndb_recv_thread_activation_threshold | 8                                   |
| ndb_recv_thread_cpu_mask          |                                        |
| ndb_report_thresh_binlog_epoch_slip | 10                                   |
| ndb_report_thresh_binlog_mem_usage | 10                                    |
```

```
| ndb_row_checksum                      | 1                                      |
| ndb_schema_dist_lock_wait_timeout     | 30                                     |
| ndb_schema_dist_timeout               | 120                                    |
| ndb_schema_dist_upgrade_allowed       | ON                                     |
| ndb_show_foreign_key_mock_tables      | OFF                                    |
| ndb_slave_conflict_role               | NONE                                   |
| ndb_table_no_logging                  | OFF                                    |
| ndb_table_temporary                   | OFF                                    |
| ndb_use_copying_alter_table           | OFF                                    |
| ndb_use_exact_count                   | OFF                                    |
| ndb_use_transactions                  | ON                                     |
| ndb_version                           | 524308                                 |
| ndb_version_string                    | ndb-8.0.34                             |
| ndb_wait_connected                    | 30                                     |
| ndb_wait_setup                        | 30                                     |
| ndbinfo_database                      | ndbinfo                                |
| ndbinfo_max_bytes                     | 0                                      |
| ndbinfo_max_rows                      | 10                                     |
| ndbinfo_offline                       | OFF                                    |
| ndbinfo_show_hidden                   | OFF                                    |
| ndbinfo_table_prefix                  | ndb$                                   |
| ndbinfo_version                       | 524308                                 |
+---------------------------------------+----------------------------------------+
```

See Section 5.1.8, “Server System Variables”, for more information.

- `SELECT * FROM performance_schema.global_variables WHERE VARIABLE_NAME LIKE 'NDB%'`

  This statement is the equivalent of the `SHOW VARIABLES` statement described in the previous item, and provides almost identical output, as shown here:

```
mysql> SELECT * FROM performance_schema.global_variables
    ->   WHERE VARIABLE_NAME LIKE 'NDB%';
+---------------------------------------+----------------------------------------+
| VARIABLE_NAME                         | VARIABLE_VALUE                         |
+---------------------------------------+----------------------------------------+
| ndb_allow_copying_alter_table         | ON                                     |
| ndb_autoincrement_prefetch_sz         | 512                                    |
| ndb_batch_size                        | 32768                                  |
| ndb_blob_read_batch_bytes             | 65536                                  |
| ndb_blob_write_batch_bytes            | 65536                                  |
| ndb_clear_apply_status                | ON                                     |
| ndb_cluster_connection_pool           | 1                                      |
| ndb_cluster_connection_pool_nodeids   |                                        |
| ndb_connectstring                     | 127.0.0.1                              |
| ndb_data_node_neighbour               | 0                                      |
| ndb_default_column_format             | FIXED                                  |
| ndb_deferred_constraints              | 0                                      |
| ndb_distribution                      | KEYHASH                                |
| ndb_eventbuffer_free_percent          | 20                                     |
| ndb_eventbuffer_max_alloc             | 0                                      |
| ndb_extra_logging                     | 1                                      |
| ndb_force_send                        | ON                                     |
| ndb_fully_replicated                  | OFF                                    |
| ndb_index_stat_enable                 | ON                                     |
| ndb_index_stat_option                 | loop_enable=1000ms,loop_idle=1000ms,
loop_busy=100ms,update_batch=1,read_batch=4,idle_batch=32,check_batch=8,
check_delay=10m,delete_batch=8,clean_delay=1m,error_batch=4,error_delay=1m,
evict_batch=8,evict_delay=1m,cache_limit=32M,cache_lowpct=90,zero_total=0      |
| ndb_join_pushdown                     | ON                                     |
| ndb_log_apply_status                  | OFF                                    |
| ndb_log_bin                           | OFF                                    |
| ndb_log_binlog_index                  | ON                                     |
| ndb_log_empty_epochs                  | OFF                                    |
| ndb_log_empty_update                  | OFF                                    |
| ndb_log_exclusive_reads               | OFF                                    |
| ndb_log_orig                          | OFF                                    |
| ndb_log_transaction_id                | OFF                                    |
| ndb_log_update_as_write               | ON                                     |
| ndb_log_update_minimal                | OFF                                    |
```

```
| ndb_log_updated_only                 | ON                                     |
| ndb_metadata_check                   | ON                                     |
| ndb_metadata_check_interval          | 60                                     |
| ndb_metadata_sync                    | OFF                                    |
| ndb_mgmd_host                        | 127.0.0.1                              |
| ndb_nodeid                           | 0                                      |
| ndb_optimization_delay               | 10                                     |
| ndb_optimized_node_selection         | 3                                      |
| ndb_read_backup                      | ON                                     |
| ndb_recv_thread_activation_threshold | 8                                      |
| ndb_recv_thread_cpu_mask             |                                        |
| ndb_report_thresh_binlog_epoch_slip  | 10                                     |
| ndb_report_thresh_binlog_mem_usage   | 10                                     |
| ndb_row_checksum                     | 1                                      |
| ndb_schema_dist_lock_wait_timeout    | 30                                     |
| ndb_schema_dist_timeout              | 120                                    |
| ndb_schema_dist_upgrade_allowed      | ON                                     |
| ndb_show_foreign_key_mock_tables     | OFF                                    |
| ndb_slave_conflict_role              | NONE                                   |
| ndb_table_no_logging                 | OFF                                    |
| ndb_table_temporary                  | OFF                                    |
| ndb_use_copying_alter_table          | OFF                                    |
| ndb_use_exact_count                  | OFF                                    |
| ndb_use_transactions                 | ON                                     |
| ndb_version                          | 524308                                 |
| ndb_version_string                   | ndb-8.0.34                             |
| ndb_wait_connected                   | 30                                     |
| ndb_wait_setup                       | 30                                     |
| ndbinfo_database                     | ndbinfo                                |
| ndbinfo_max_bytes                    | 0                                      |
| ndbinfo_max_rows                     | 10                                     |
| ndbinfo_offline                      | OFF                                    |
| ndbinfo_show_hidden                  | OFF                                    |
| ndbinfo_table_prefix                 | ndb$                                   |
| ndbinfo_version                      | 524308                                 |
+--------------------------------------+----------------------------------------+
```

Unlike the case with the `SHOW VARIABLES` statement, it is possible to select individual columns. For example:

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

A more useful query is shown here:

```
mysql> SELECT VARIABLE_NAME AS Name, VARIABLE_VALUE AS Value
     >   FROM performance_schema.global_variables
     >   WHERE VARIABLE_NAME
     >     IN ('version', 'ndb_version',
     >       'ndb_version_string', 'ndbinfo_version');

+--------------------+----------------+
| Name               | Value          |
+--------------------+----------------+
| ndb_version        | 524317         |
| ndb_version_string | ndb-8.0.29     |
| ndbinfo_version    | 524317         |
| version            | 8.0.29-cluster |
+--------------------+----------------+
4 rows in set (0.00 sec)
```

For more information, see Section 27.12.15, “Performance Schema Status Variable Tables”, and Section 5.1.8, “Server System Variables”.

- `SHOW STATUS LIKE 'NDB%'`

This statement shows at a glance whether or not the MySQL server is acting as a cluster SQL node, and if so, it provides the MySQL server's cluster node ID, the host name and port for the cluster management server to which it is connected, and the number of data nodes in the cluster, as shown here:

```
mysql> SHOW STATUS LIKE 'NDB%';
+-------------------------------------------+-------------------------------+
| Variable_name                             | Value                         |
+-------------------------------------------+-------------------------------+
| Ndb_metadata_detected_count               | 0                             |
| Ndb_cluster_node_id                       | 100                           |
| Ndb_config_from_host                      | 127.0.0.1                     |
| Ndb_config_from_port                      | 1186                          |
| Ndb_number_of_data_nodes                  | 2                             |
| Ndb_number_of_ready_data_nodes            | 2                             |
| Ndb_connect_count                         | 0                             |
| Ndb_execute_count                         | 0                             |
| Ndb_scan_count                            | 0                             |
| Ndb_pruned_scan_count                     | 0                             |
| Ndb_schema_locks_count                    | 0                             |
| Ndb_api_wait_exec_complete_count_session  | 0                             |
| Ndb_api_wait_scan_result_count_session    | 0                             |
| Ndb_api_wait_meta_request_count_session   | 1                             |
| Ndb_api_wait_nanos_count_session          | 163446                        |
| Ndb_api_bytes_sent_count_session          | 60                            |
| Ndb_api_bytes_received_count_session      | 28                            |
| Ndb_api_trans_start_count_session         | 0                             |
| Ndb_api_trans_commit_count_session        | 0                             |
| Ndb_api_trans_abort_count_session         | 0                             |
| Ndb_api_trans_close_count_session         | 0                             |
| Ndb_api_pk_op_count_session               | 0                             |
| Ndb_api_uk_op_count_session               | 0                             |
| Ndb_api_table_scan_count_session          | 0                             |
| Ndb_api_range_scan_count_session          | 0                             |
| Ndb_api_pruned_scan_count_session         | 0                             |
| Ndb_api_scan_batch_count_session          | 0                             |
| Ndb_api_read_row_count_session            | 0                             |
| Ndb_api_trans_local_read_row_count_session | 0                             |
| Ndb_api_adaptive_send_forced_count_session | 0                             |
| Ndb_api_adaptive_send_unforced_count_session | 0                             |
| Ndb_api_adaptive_send_deferred_count_session | 0                             |
| Ndb_trans_hint_count_session              | 0                             |
| Ndb_sorted_scan_count                     | 0                             |
| Ndb_pushed_queries_defined                | 0                             |
| Ndb_pushed_queries_dropped                | 0                             |
| Ndb_pushed_queries_executed               | 0                             |
| Ndb_pushed_reads                          | 0                             |
| Ndb_last_commit_epoch_server              | 37632503447571                |
| Ndb_last_commit_epoch_session             | 0                             |
| Ndb_system_name                           | MC_20191126162038             |
| Ndb_api_event_data_count_injector         | 0                             |
| Ndb_api_event_nondata_count_injector      | 0                             |
| Ndb_api_event_bytes_count_injector        | 0                             |
| Ndb_api_wait_exec_complete_count_slave    | 0                             |
| Ndb_api_wait_scan_result_count_slave      | 0                             |
| Ndb_api_wait_meta_request_count_slave     | 0                             |
| Ndb_api_wait_nanos_count_slave            | 0                             |
| Ndb_api_bytes_sent_count_slave            | 0                             |
| Ndb_api_bytes_received_count_slave        | 0                             |
| Ndb_api_trans_start_count_slave           | 0                             |
| Ndb_api_trans_commit_count_slave          | 0                             |
| Ndb_api_trans_abort_count_slave           | 0                             |
| Ndb_api_trans_close_count_slave           | 0                             |
| Ndb_api_pk_op_count_slave                 | 0                             |
| Ndb_api_uk_op_count_slave                 | 0                             |
| Ndb_api_table_scan_count_slave            | 0                             |
| Ndb_api_range_scan_count_slave            | 0                             |
| Ndb_api_pruned_scan_count_slave           | 0                             |
| Ndb_api_scan_batch_count_slave            | 0                             |
| Ndb_api_read_row_count_slave              | 0                             |
```

```
| Ndb_api_trans_local_read_row_count_slave     | 0                                |
| Ndb_api_adaptive_send_forced_count_slave     | 0                                |
| Ndb_api_adaptive_send_unforced_count_slave   | 0                                |
| Ndb_api_adaptive_send_deferred_count_slave   | 0                                |
| Ndb_slave_max_replicated_epoch               | 0                                |
| Ndb_api_wait_exec_complete_count             | 4                                |
| Ndb_api_wait_scan_result_count               | 7                                |
| Ndb_api_wait_meta_request_count              | 172                              |
| Ndb_api_wait_nanos_count                     | 1083548094028                    |
| Ndb_api_bytes_sent_count                     | 4640                             |
| Ndb_api_bytes_received_count                 | 109356                           |
| Ndb_api_trans_start_count                    | 4                                |
| Ndb_api_trans_commit_count                   | 1                                |
| Ndb_api_trans_abort_count                    | 1                                |
| Ndb_api_trans_close_count                    | 4                                |
| Ndb_api_pk_op_count                          | 2                                |
| Ndb_api_uk_op_count                          | 0                                |
| Ndb_api_table_scan_count                     | 1                                |
| Ndb_api_range_scan_count                     | 1                                |
| Ndb_api_pruned_scan_count                    | 0                                |
| Ndb_api_scan_batch_count                     | 1                                |
| Ndb_api_read_row_count                       | 3                                |
| Ndb_api_trans_local_read_row_count           | 2                                |
| Ndb_api_adaptive_send_forced_count           | 1                                |
| Ndb_api_adaptive_send_unforced_count         | 5                                |
| Ndb_api_adaptive_send_deferred_count         | 0                                |
| Ndb_api_event_data_count                     | 0                                |
| Ndb_api_event_nondata_count                  | 0                                |
| Ndb_api_event_bytes_count                    | 0                                |
| Ndb_metadata_excluded_count                  | 0                                |
| Ndb_metadata_synced_count                    | 0                                |
| Ndb_conflict_fn_max                          | 0                                |
| Ndb_conflict_fn_old                          | 0                                |
| Ndb_conflict_fn_max_del_win                  | 0                                |
| Ndb_conflict_fn_epoch                        | 0                                |
| Ndb_conflict_fn_epoch_trans                  | 0                                |
| Ndb_conflict_fn_epoch2                       | 0                                |
| Ndb_conflict_fn_epoch2_trans                 | 0                                |
| Ndb_conflict_trans_row_conflict_count        | 0                                |
| Ndb_conflict_trans_row_reject_count          | 0                                |
| Ndb_conflict_trans_reject_count              | 0                                |
| Ndb_conflict_trans_detect_iter_count         | 0                                |
| Ndb_conflict_trans_conflict_commit_count     | 0                                |
| Ndb_conflict_epoch_delete_delete_count       | 0                                |
| Ndb_conflict_reflected_op_prepare_count      | 0                                |
| Ndb_conflict_reflected_op_discard_count      | 0                                |
| Ndb_conflict_refresh_op_count                | 0                                |
| Ndb_conflict_last_conflict_epoch             | 0                                |
| Ndb_conflict_last_stable_epoch               | 0                                |
| Ndb_index_stat_status                        | allow:1,enable:1,busy:0,
loop:1000,list:(new:0,update:0,read:0,idle:0,check:0,delete:0,error:0,total:0),
analyze:(queue:0,wait:0),stats:(nostats:0,wait:0),total:(analyze:(all:0,error:0),
query:(all:0,nostats:0,error:0),event:(act:0,skip:0,miss:0),cache:(refresh:0,
clean:0,pinned:0,drop:0,evict:0)),cache:(query:0,clean:0,drop:0,evict:0,
usedpct:0.00,highpct:0.00)                                                      |
| Ndb_index_stat_cache_query                   | 0                                |
| Ndb_index_stat_cache_clean                   | 0                                |
+----------------------------------------------+----------------------------------+
```

If the MySQL server was built with `NDB` support, but it is not currently connected to a cluster, every row in the output of this statement contains a zero or an empty string for the `Value` column.

See also Section 13.7.7.37, “SHOW STATUS Statement”.

- `SELECT * FROM performance_schema.global_status WHERE VARIABLE_NAME LIKE 'NDB%'`

  This statement provides similar output to the `SHOW STATUS` statement discussed in the previous item. Unlike the case with `SHOW STATUS`, it is possible using `SELECT` statements to extract values in SQL for use in scripts for monitoring and automation purposes.

  For more information, see Section 27.12.15, “Performance Schema Status Variable Tables”.

- `SELECT * FROM INFORMATION_SCHEMA.PLUGINS WHERE PLUGIN_NAME LIKE 'NDB%'`

  This statement displays information from the Information Schema `PLUGINS` table about plugins associated with NDB Cluster, such as version, author, and license, as shown here:

```
mysql> SELECT * FROM INFORMATION_SCHEMA.PLUGINS
     >     WHERE PLUGIN_NAME LIKE 'NDB%'\G
*************************** 1. row ***************************
           PLUGIN_NAME: ndbcluster
        PLUGIN_VERSION: 1.0
         PLUGIN_STATUS: ACTIVE
           PLUGIN_TYPE: STORAGE ENGINE
   PLUGIN_TYPE_VERSION: 80032.0
        PLUGIN_LIBRARY: NULL
PLUGIN_LIBRARY_VERSION: NULL
         PLUGIN_AUTHOR: Oracle Corporation
    PLUGIN_DESCRIPTION: Clustered, fault-tolerant tables
        PLUGIN_LICENSE: GPL
           LOAD_OPTION: ON
*************************** 2. row ***************************
           PLUGIN_NAME: ndbinfo
        PLUGIN_VERSION: 0.1
         PLUGIN_STATUS: ACTIVE
           PLUGIN_TYPE: STORAGE ENGINE
   PLUGIN_TYPE_VERSION: 80032.0
        PLUGIN_LIBRARY: NULL
PLUGIN_LIBRARY_VERSION: NULL
         PLUGIN_AUTHOR: Oracle Corporation
    PLUGIN_DESCRIPTION: MySQL Cluster system information storage engine
        PLUGIN_LICENSE: GPL
           LOAD_OPTION: ON
*************************** 3. row ***************************
           PLUGIN_NAME: ndb_transid_mysql_connection_map
        PLUGIN_VERSION: 0.1
         PLUGIN_STATUS: ACTIVE
           PLUGIN_TYPE: INFORMATION SCHEMA
   PLUGIN_TYPE_VERSION: 80032.0
        PLUGIN_LIBRARY: NULL
PLUGIN_LIBRARY_VERSION: NULL
         PLUGIN_AUTHOR: Oracle Corporation
    PLUGIN_DESCRIPTION: Map between MySQL connection ID and NDB transaction ID
        PLUGIN_LICENSE: GPL
           LOAD_OPTION: ON
```

  You can also use the `SHOW PLUGINS` statement to display this information, but the output from that statement cannot easily be filtered. See also The MySQL Plugin API, which describes where and how the information in the `PLUGINS` table is obtained.

You can also query the tables in the `ndbinfo` information database for real-time data about many NDB Cluster operations. See Section 23.6.16, “ndbinfo: The NDB Cluster Information Database”.

23.6.20 NDB Cluster Security Issues

This section discusses security considerations to take into account when setting up and running NDB Cluster.

Topics covered in this section include the following:

- NDB Cluster and network security issues

- Configuration issues relating to running NDB Cluster securely
- NDB Cluster and the MySQL privilege system
- MySQL standard security procedures as applicable to NDB Cluster

23.6.20.1 NDB Cluster Security and Networking Issues

In this section, we discuss basic network security issues as they relate to NDB Cluster. It is extremely important to remember that NDB Cluster “out of the box” is not secure; you or your network administrator must take the proper steps to ensure that your cluster cannot be compromised over the network.

Cluster communication protocols are inherently insecure, and no encryption or similar security measures are used in communications between nodes in the cluster. Because network speed and latency have a direct impact on the cluster's efficiency, it is also not advisable to employ SSL or other encryption to network connections between nodes, as such schemes cause slow communications.

It is also true that no authentication is used for controlling API node access to an NDB Cluster. As with encryption, the overhead of imposing authentication requirements would have an adverse impact on Cluster performance.

In addition, there is no checking of the source IP address for either of the following when accessing the cluster:

- SQL or API nodes using “free slots” created by empty `[mysqld]` or `[api]` sections in the `config.ini` file

  This means that, if there are any empty `[mysqld]` or `[api]` sections in the `config.ini` file, then any API nodes (including SQL nodes) that know the management server's host name (or IP address) and port can connect to the cluster and access its data without restriction. (See Section 23.6.20.2, “NDB Cluster and MySQL Privileges”, for more information about this and related issues.)

  Note

  You can exercise some control over SQL and API node access to the cluster by specifying a `HostName` parameter for all `[mysqld]` and `[api]` sections in the `config.ini` file. However, this also means that, should you wish to connect an API node to the cluster from a previously unused host, you need to add an `[api]` section containing its host name to the `config.ini` file.

  More information is available elsewhere in this chapter about the `HostName` parameter. Also see Section 23.4.1, “Quick Test Setup of NDB Cluster”, for configuration examples using `HostName` with API nodes.

- Any `ndb_mgm` client

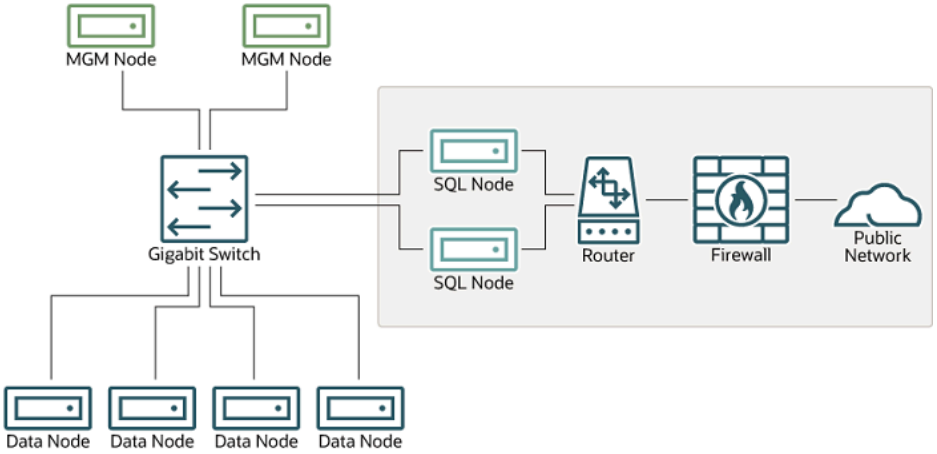
  This means that any cluster management client that is given the management server's host name (or IP address) and port (if not the standard port) can connect to the cluster and execute any management client command. This includes commands such as `ALL STOP` and `SHUTDOWN`.

For these reasons, it is necessary to protect the cluster on the network level. The safest network configuration for Cluster is one which isolates connections between Cluster nodes from any other network communications. This can be accomplished by any of the following methods:

1. Keeping Cluster nodes on a network that is physically separate from any public networks. This option is the most dependable; however, it is the most expensive to implement.

   We show an example of an NDB Cluster setup using such a physically segregated network here:

Figure 23.7 NDB Cluster with Hardware Firewall



This setup has two networks, one private (solid box) for the Cluster management servers and data nodes, and one public (dotted box) where the SQL nodes reside. (We show the management and data nodes connected using a gigabit switch since this provides the best performance.) Both networks are protected from the outside by a hardware firewall, sometimes also known as a *network-based firewall*.

This network setup is safest because no packets can reach the cluster's management or data nodes from outside the network—and none of the cluster's internal communications can reach the outside—without going through the SQL nodes, as long as the SQL nodes do not permit any packets to be forwarded. This means, of course, that all SQL nodes must be secured against hacking attempts.

Important

With regard to potential security vulnerabilities, an SQL node is no different from any other MySQL server. See Section 6.1.3, “Making MySQL Secure Against Attackers”, for a description of techniques you can use to secure MySQL servers.

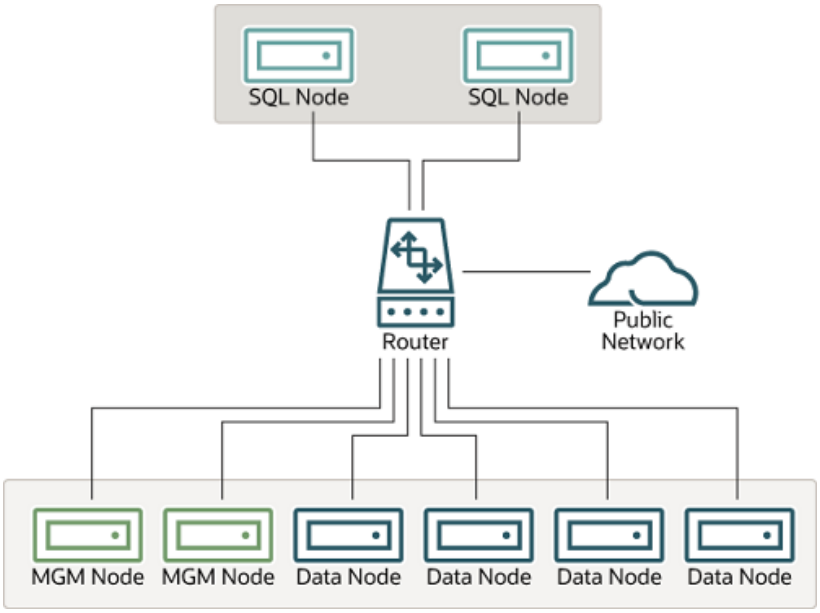
2. Using one or more software firewalls (also known as *host-based firewalls*) to control which packets pass through to the cluster from portions of the network that do not require access to it. In this type

of setup, a software firewall must be installed on every host in the cluster which might otherwise be accessible from outside the local network.

The host-based option is the least expensive to implement, but relies purely on software to provide protection and so is the most difficult to keep secure.

This type of network setup for NDB Cluster is illustrated here:

Figure 23.8 NDB Cluster with Software Firewalls



Using this type of network setup means that there are two zones of NDB Cluster hosts. Each cluster host must be able to communicate with all of the other machines in the cluster, but only those hosting SQL nodes (dotted box) can be permitted to have any contact with the outside, while those in the zone containing the data nodes and management nodes (solid box) must be isolated from any machines that are not part of the cluster. Applications using the cluster and user of those applications must *not* be permitted to have direct access to the management and data node hosts.

To accomplish this, you must set up software firewalls that limit the traffic to the type or types shown in the following table, according to the type of node that is running on each cluster host computer:

Table 23.68 Node types in a host-based firewall cluster configuration

Node Type	Permitted Traffic
SQL or API node	• It originates from the IP address of a management or data node (using any TCP or UDP port). • It originates from within the network in which the cluster resides and is on the port that your application is using.
Data node or Management node	• It originates from the IP address of a management or data node (using any TCP or UDP port).

Node Type	Permitted Traffic
	• It originates from the IP address of an SQL or API node.

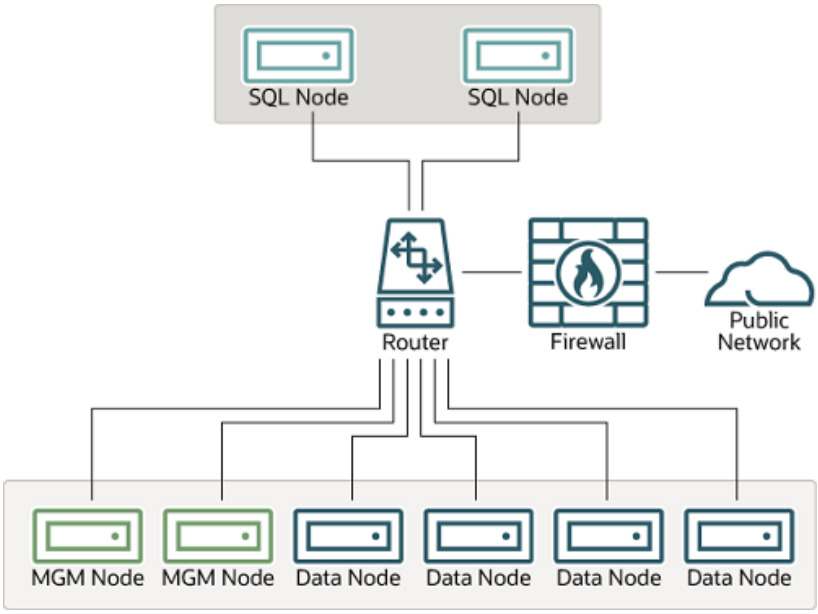
Any traffic other than that shown in the table for a given node type should be denied.

The specifics of configuring a firewall vary from firewall application to firewall application, and are beyond the scope of this Manual. `iptables` is a very common and reliable firewall application, which is often used with APF as a front end to make configuration easier. You can (and should) consult the documentation for the software firewall that you employ, should you choose to implement an NDB Cluster network setup of this type, or of a “mixed” type as discussed under the next item.

3. It is also possible to employ a combination of the first two methods, using both hardware and software to secure the cluster—that is, using both network-based and host-based firewalls. This is between the first two schemes in terms of both security level and cost. This type of network setup keeps the cluster behind the hardware firewall, but permits incoming packets to travel beyond the router connecting all cluster hosts to reach the SQL nodes.

   One possible network deployment of an NDB Cluster using hardware and software firewalls in combination is shown here:

   Figure 23.9 NDB Cluster with a Combination of Hardware and Software Firewalls



   In this case, you can set the rules in the hardware firewall to deny any external traffic except to SQL nodes and API nodes, and then permit traffic to them only on the ports required by your application.

Whatever network configuration you use, remember that your objective from the viewpoint of keeping the cluster secure remains the same—to prevent any unessential traffic from reaching the cluster while ensuring the most efficient communication between the nodes in the cluster.

Because NDB Cluster requires large numbers of ports to be open for communications between nodes, the recommended option is to use a segregated network. This represents the simplest way to prevent unwanted traffic from reaching the cluster.

Note

If you wish to administer an NDB Cluster remotely (that is, from outside the local network), the recommended way to do this is to use `ssh` or another secure

> login shell to access an SQL node host. From this host, you can then run the management client to access the management server safely, from within the cluster's own local network.
>
> Even though it is possible to do so in theory, it is *not* recommended to use `ndb_mgm` to manage a Cluster directly from outside the local network on which the Cluster is running. Since neither authentication nor encryption takes place between the management client and the management server, this represents an extremely insecure means of managing the cluster, and is almost certain to be compromised sooner or later.

23.6.20.2 NDB Cluster and MySQL Privileges

In this section, we discuss how the MySQL privilege system works in relation to NDB Cluster and the implications of this for keeping an NDB Cluster secure.

Standard MySQL privileges apply to NDB Cluster tables. This includes all MySQL privilege types (`SELECT` privilege, `UPDATE` privilege, `DELETE` privilege, and so on) granted on the database, table, and column level. As with any other MySQL Server, user and privilege information is stored in the `mysql` system database. The SQL statements used to grant and revoke privileges on `NDB` tables, databases containing such tables, and columns within such tables are identical in all respects with the `GRANT` and `REVOKE` statements used in connection with database objects involving any (other) MySQL storage engine. The same thing is true with respect to the `CREATE USER` and `DROP USER` statements.

It is important to keep in mind that, by default, the MySQL grant tables use the `InnoDB` storage engine. Because of this, those tables are not normally duplicated or shared among MySQL servers acting as SQL nodes in an NDB Cluster. In other words, changes in users and their privileges do not automatically propagate between SQL nodes by default. If you wish, you can enable synchronization of MySQL users and privileges across NDB Cluster SQL nodes; see Section 23.6.13, “Privilege Synchronization and NDB_STORED_USER”, for details.

Conversely, because there is no way in MySQL to deny privileges (privileges can either be revoked or not granted in the first place, but not denied as such), there is no special protection for `NDB` tables on one SQL node from users that have privileges on another SQL node; this is true even if you are not using automatic distribution of user privileges. The definitive example of this is the MySQL `root` account, which can perform any action on any database object. In combination with empty `[mysqld]` or `[api]` sections of the `config.ini` file, this account can be especially dangerous. To understand why, consider the following scenario:

- The `config.ini` file contains at least one empty `[mysqld]` or `[api]` section. This means that the NDB Cluster management server performs no checking of the host from which a MySQL Server (or other API node) accesses the NDB Cluster.
- There is no firewall, or the firewall fails to protect against access to the NDB Cluster from hosts external to the network.
- The host name or IP address of the NDB Cluster management server is known or can be determined from outside the network.

If these conditions are true, then anyone, anywhere can start a MySQL Server with `--ndbcluster --ndb-connectstring=`*`management_host`* and access this NDB Cluster. Using the MySQL `root` account, this person can then perform the following actions:

- Execute metadata statements such as `SHOW DATABASES` statement (to obtain a list of all `NDB` databases on the server) or `SHOW TABLES FROM` *`some_ndb_database`* statement to obtain a list of all `NDB` tables in a given database
- Run any legal MySQL statements on any of the discovered tables, such as:
  - `SELECT * FROM` *`some_table`* or `TABLE` *`some_table`* to read all the data from any table

- `DELETE FROM some_table` or TRUNCATE TABLE to delete all the data from a table
- `DESCRIBE some_table` or `SHOW CREATE TABLE some_table` to determine the table schema
- `UPDATE some_table SET column1 = some_value` to fill a table column with “garbage” data; this could actually cause much greater damage than simply deleting all the data

  More insidious variations might include statements like these:

  ```
  UPDATE some_table SET an_int_column = an_int_column + 1
  ```

  or

  ```
  UPDATE some_table SET a_varchar_column = REVERSE(a_varchar_column)
  ```

  Such malicious statements are limited only by the imagination of the attacker.

The only tables that would be safe from this sort of mayhem would be those tables that were created using storage engines other than `NDB`, and so not visible to a “rogue” SQL node.

A user who can log in as `root` can also access the `INFORMATION_SCHEMA` database and its tables, and so obtain information about databases, tables, stored routines, scheduled events, and any other database objects for which metadata is stored in `INFORMATION_SCHEMA`.

It is also a very good idea to use different passwords for the `root` accounts on different NDB Cluster SQL nodes unless you are using shared privileges.

In sum, you cannot have a safe NDB Cluster if it is directly accessible from outside your local network.

Important

Never leave the MySQL root account password empty. This is just as true when running MySQL as an NDB Cluster SQL node as it is when running it as a standalone (non-Cluster) MySQL Server, and should be done as part of the MySQL installation process before configuring the MySQL Server as an SQL node in an NDB Cluster.

If you need to synchronize `mysql` system tables between SQL nodes, you can use standard MySQL replication to do so, or employ a script to copy table entries between the MySQL servers. Users and their privileges can be shared and kept in synch using the `NDB_STORED_USER` privilege.

Summary. The most important points to remember regarding the MySQL privilege system with regard to NDB Cluster are listed here:

1. Users and privileges established on one SQL node do not automatically exist or take effect on other SQL nodes in the cluster. Conversely, removing a user or privilege on one SQL node in the cluster does not remove the user or privilege from any other SQL nodes.
2. You can share MySQL users and privileges among SQL nodes using `NDB_STORED_USER`.
3. Once a MySQL user is granted privileges on an `NDB` table from one SQL node in an NDB Cluster, that user can “see” any data in that table regardless of the SQL node from which the data originated, even if that user is not shared.

23.6.20.3 NDB Cluster and MySQL Security Procedures

In this section, we discuss MySQL standard security procedures as they apply to running NDB Cluster.

In general, any standard procedure for running MySQL securely also applies to running a MySQL Server as part of an NDB Cluster. First and foremost, you should always run a MySQL Server as the `mysql` operating system user; this is no different from running MySQL in a standard (non-Cluster)

environment. The `mysql` system account should be uniquely and clearly defined. Fortunately, this is the default behavior for a new MySQL installation. You can verify that the `mysqld` process is running as the `mysql` operating system user by using the system command such as the one shown here:

```
$> ps aux | grep mysql
root     10467  0.0  0.1   3616  1380 pts/3    S    11:53   0:00 \
  /bin/sh ./mysqld_safe --ndbcluster --ndb-connectstring=localhost:1186
mysql    10512  0.2  2.5  58528 26636 pts/3    Sl   11:53   0:00 \
  /usr/local/mysql/libexec/mysqld --basedir=/usr/local/mysql \
  --datadir=/usr/local/mysql/var --user=mysql --ndbcluster \
  --ndb-connectstring=localhost:1186 --pid-file=/usr/local/mysql/var/mothra.pid \
  --log-error=/usr/local/mysql/var/mothra.err
jon      10579  0.0  0.0   2736   688 pts/0    S+   11:54   0:00 grep mysql
```

If the `mysqld` process is running as any other user than `mysql`, you should immediately shut it down and restart it as the `mysql` user. If this user does not exist on the system, the `mysql` user account should be created, and this user should be part of the `mysql` user group; in this case, you should also make sure that the MySQL data directory on this system (as set using the `--datadir` option for `mysqld`) is owned by the `mysql` user, and that the SQL node's `my.cnf` file includes `user=mysql` in the `[mysqld]` section. Alternatively, you can start the MySQL server process with `--user=mysql` on the command line, but it is preferable to use the `my.cnf` option, since you might forget to use the command-line option and so have `mysqld` running as another user unintentionally. The `mysqld_safe` startup script forces MySQL to run as the `mysql` user.

Important

Never run `mysqld` as the system root user. Doing so means that potentially any file on the system can be read by MySQL, and thus—should MySQL be compromised—by an attacker.

As mentioned in the previous section (see Section 23.6.20.2, “NDB Cluster and MySQL Privileges”), you should always set a root password for the MySQL Server as soon as you have it running. You should also delete the anonymous user account that is installed by default. You can accomplish these tasks using the following statements:

```
$> mysql -u root

mysql> UPDATE mysql.user
    ->     SET Password=PASSWORD('secure_password')
    ->     WHERE User='root';

mysql> DELETE FROM mysql.user
    ->     WHERE User='';

mysql> FLUSH PRIVILEGES;
```

Be very careful when executing the `DELETE` statement not to omit the `WHERE` clause, or you risk deleting *all* MySQL users. Be sure to run the `FLUSH PRIVILEGES` statement as soon as you have modified the `mysql.user` table, so that the changes take immediate effect. Without `FLUSH PRIVILEGES`, the changes do not take effect until the next time that the server is restarted.

Note

Many of the NDB Cluster utilities such as `ndb_show_tables`, `ndb_desc`, and `ndb_select_all` also work without authentication and can reveal table names, schemas, and data. By default these are installed on Unix-style systems with the permissions `wxr-xr-x` (755), which means they can be executed by any user that can access the `mysql/bin` directory.

See Section 23.5, “NDB Cluster Programs”, for more information about these utilities.

23.7 NDB Cluster Replication

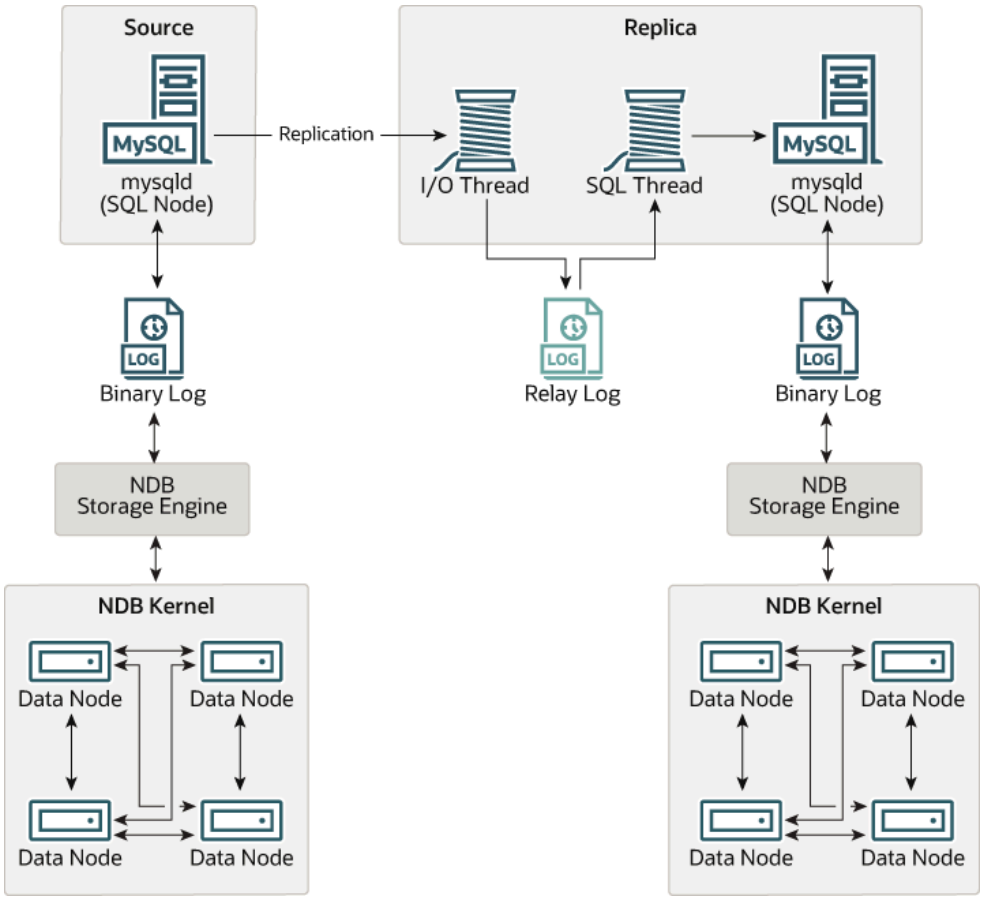
NDB Cluster supports *asynchronous replication*, more usually referred to simply as “replication”. This section explains how to set up and manage a configuration in which one group of computers operating as an NDB Cluster replicates to a second computer or group of computers. We assume some familiarity on the part of the reader with standard MySQL replication as discussed elsewhere in this Manual. (See Chapter 17, *Replication*).

Note

NDB Cluster does not support replication using GTIDs; semisynchronous replication and group replication are also not supported by the `NDB` storage engine.

Normal (non-clustered) replication involves a source server and a replica server, the source being so named because operations and data to be replicated originate with it, and the replica being the recipient of these. In NDB Cluster, replication is conceptually very similar but can be more complex in practice, as it may be extended to cover a number of different configurations including replicating between two complete clusters. Although an NDB Cluster itself depends on the `NDB` storage engine for clustering functionality, it is not necessary to use `NDB` as the storage engine for the replica's copies of the replicated tables (see Replication from NDB to other storage engines). However, for maximum availability, it is possible (and preferable) to replicate from one NDB Cluster to another, and it is this scenario that we discuss, as shown in the following figure:

Figure 23.10 NDB Cluster-to-Cluster Replication Layout



In this scenario, the replication process is one in which successive states of a source cluster are logged and saved to a replica cluster. This process is accomplished by a special thread known as the NDB binary log injector thread, which runs on each MySQL server and produces a binary log (`binlog`). This thread ensures that all changes in the cluster producing the binary log—and not just those changes that are effected through the MySQL Server—are inserted into the binary log with the correct serialization order. We refer to the MySQL source and replica servers as replication servers or replication nodes, and the data flow or line of communication between them as a *replication channel*.

For information about performing point-in-time recovery with NDB Cluster and NDB Cluster Replication, see Section 23.7.9.2, “Point-In-Time Recovery Using NDB Cluster Replication”.

NDB API replica status variables. NDB API counters can provide enhanced monitoring capabilities on replica clusters. These counters are implemented as NDB statistics `_slave` status variables, as seen in the output of `SHOW STATUS`, or in the results of queries against the Performance Schema `session_status` or `global_status` table in a `mysql` client session connected to a MySQL Server that is acting as a replica in NDB Cluster Replication. By comparing the values of these status variables before and after the execution of statements affecting replicated `NDB` tables, you can observe the corresponding actions taken on the NDB API level by the replica, which can be useful when monitoring or troubleshooting NDB Cluster Replication. Section 23.6.15, “NDB API Statistics Counters and Variables”, provides additional information.

Replication from NDB to non-NDB tables. It is possible to replicate `NDB` tables from an NDB Cluster acting as the replication source to tables using other MySQL storage engines such as `InnoDB` or `MyISAM` on a replica `mysqld`. This is subject to a number of conditions; see Replication from NDB to other storage engines, and Replication from NDB to a nontransactional storage engine, for more information.

23.7.1 NDB Cluster Replication: Abbreviations and Symbols

Throughout this section, we use the following abbreviations or symbols for referring to the source and replica clusters, and to processes and commands run on the clusters or cluster nodes:

Table 23.69 Abbreviations used throughout this section referring to source and replica clusters, and to processes and commands run on cluster nodes

Symbol or Abbreviation	Description (Refers to...)
S	The cluster serving as the (primary) replication source
R	The cluster acting as the (primary) replica
`shellS>`	Shell command to be issued on the source cluster
`mysqlS>`	MySQL client command issued on a single MySQL server running as an SQL node on the source cluster
`mysqlS*>`	MySQL client command to be issued on all SQL nodes participating in the replication source cluster
`shellR>`	Shell command to be issued on the replica cluster
`mysqlR>`	MySQL client command issued on a single MySQL server running as an SQL node on the replica cluster
`mysqlR*>`	MySQL client command to be issued on all SQL nodes participating in the replica cluster
C	Primary replication channel
C'	Secondary replication channel
S'	Secondary replication source
R'	Secondary replica

23.7.2 General Requirements for NDB Cluster Replication

A replication channel requires two MySQL servers acting as replication servers (one each for the source and replica). For example, this means that in the case of a replication setup with two replication channels (to provide an extra channel for redundancy), there should be a total of four replication nodes, two per cluster.

Replication of an NDB Cluster as described in this section and those following is dependent on row-based replication. This means that the replication source MySQL server must be running with `--binlog-format=ROW` or `--binlog-format=MIXED`, as described in Section 23.7.6, “Starting NDB Cluster Replication (Single Replication Channel)”. For general information about row-based replication, see Section 17.2.1, “Replication Formats”.

Important

If you attempt to use NDB Cluster Replication with `--binlog-format=STATEMENT`, replication fails to work properly because the `ndb_binlog_index` table on the source cluster and the `epoch` column of the `ndb_apply_status` table on the replica cluster are not updated (see Section 23.7.4, “NDB Cluster Replication Schema and Tables”). Instead, only updates on the MySQL server acting as the replication source propagate to the replica, and no updates from any other SQL nodes in the source cluster are replicated.

The default value for the `--binlog-format` option is `MIXED`.

Each MySQL server used for replication in either cluster must be uniquely identified among all the MySQL replication servers participating in either cluster (you cannot have replication servers on both the source and replica clusters sharing the same ID). This can be done by starting each SQL node using the `--server-id=id` option, where *`id`* is a unique integer. Although it is not strictly necessary, we assume for purposes of this discussion that all NDB Cluster binaries are of the same release version.

It is generally true in MySQL Replication that both MySQL servers (`mysqld` processes) involved must be compatible with one another with respect to both the version of the replication protocol used and the SQL feature sets which they support (see Section 17.5.2, “Replication Compatibility Between MySQL Versions”). It is due to such differences between the binaries in the NDB Cluster and MySQL Server 8.0 distributions that NDB Cluster Replication has the additional requirement that both `mysqld` binaries come from an NDB Cluster distribution. The simplest and easiest way to assure that the `mysqld` servers are compatible is to use the same NDB Cluster distribution for all source and replica `mysqld` binaries.

We assume that the replica server or cluster is dedicated to replication of the source cluster, and that no other data is being stored on it.

All `NDB` tables being replicated must be created using a MySQL server and client. Tables and other database objects created using the NDB API (with, for example, `Dictionary::createTable()`) are not visible to a MySQL server and so are not replicated. Updates by NDB API applications to existing tables that were created using a MySQL server can be replicated.

Note

It is possible to replicate an NDB Cluster using statement-based replication. However, in this case, the following restrictions apply:

- All updates to data rows on the cluster acting as the source must be directed to a single MySQL server.
- It is not possible to replicate a cluster using multiple simultaneous MySQL replication processes.
- Only changes made at the SQL level are replicated.

These are in addition to the other limitations of statement-based replication as opposed to row-based replication; see Section 17.2.1.1, “Advantages and Disadvantages of Statement-Based and Row-Based Replication”, for more specific information concerning the differences between the two replication formats.

23.7.3 Known Issues in NDB Cluster Replication

This section discusses known problems or issues when using replication with NDB Cluster.

Loss of connection between source and replica. A loss of connection can occur either between the source cluster SQL node and the replica cluster SQL node, or between the source SQL node and the data nodes of the source cluster. In the latter case, this can occur not only as a result of loss of physical connection (for example, a broken network cable), but due to the overflow of data node event buffers; if the SQL node is too slow to respond, it may be dropped by the cluster (this is controllable to some degree by adjusting the `MaxBufferedEpochs` and `TimeBetweenEpochs` configuration parameters). If this occurs, *it is entirely possible for new data to be inserted into the source cluster without being recorded in the source SQL node's binary log*. For this reason, to guarantee high availability, it is extremely important to maintain a backup replication channel, to monitor the primary channel, and to fail over to the secondary replication channel when necessary to keep the replica cluster synchronized with the source. NDB Cluster is not designed to perform such monitoring on its own; for this, an external application is required.

The source SQL node issues a “gap” event when connecting or reconnecting to the source cluster. (A gap event is a type of “incident event,” which indicates an incident that occurs that affects the contents of the database but that cannot easily be represented as a set of changes. Examples of incidents are server failures, database resynchronization, some software updates, and some hardware changes.) When the replica encounters a gap in the replication log, it stops with an error message. This message is available in the output of `SHOW REPLICA STATUS` (prior to NDB 8.0.22, use `SHOW SLAVE STATUS`), and indicates that the SQL thread has stopped due to an incident registered in the replication stream, and that manual intervention is required. See Section 23.7.8, “Implementing Failover with NDB Cluster Replication”, for more information about what to do in such circumstances.

Important

Because NDB Cluster is not designed on its own to monitor replication status or provide failover, if high availability is a requirement for the replica server or cluster, then you must set up multiple replication lines, monitor the source `mysqld` on the primary replication line, and be prepared fail over to a secondary line if and as necessary. This must be done manually, or possibly by means of a third-party application. For information about implementing this type of setup, see Section 23.7.7, “Using Two Replication Channels for NDB Cluster Replication”, and Section 23.7.8, “Implementing Failover with NDB Cluster Replication”.

If you are replicating from a standalone MySQL server to an NDB Cluster, one channel is usually sufficient.

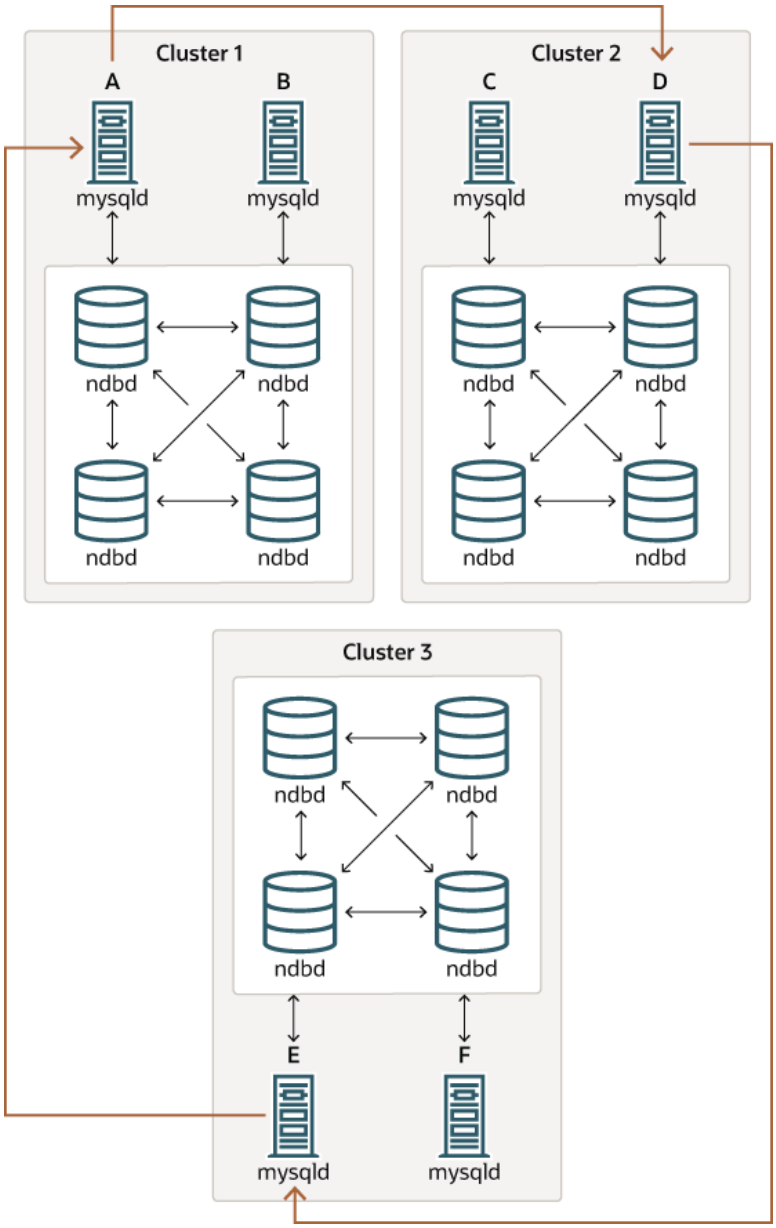
Circular replication. NDB Cluster Replication supports circular replication, as shown in the next example. The replication setup involves three NDB Clusters numbered 1, 2, and 3, in which Cluster 1 acts as the replication source for Cluster 2, Cluster 2 acts as the source for Cluster 3, and Cluster 3 acts as the source for Cluster 1, thus completing the circle. Each NDB Cluster has two SQL nodes, with SQL nodes A and B belonging to Cluster 1, SQL nodes C and D belonging to Cluster 2, and SQL nodes E and F belonging to Cluster 3.

Circular replication using these clusters is supported as long as the following conditions are met:

- The SQL nodes on all source and replica clusters are the same.
- All SQL nodes acting as sources and replicas are started with the system variable `log_replica_updates` (NDB 8.0.26 and later) or `log_slave_updates` (prior to NDB 8.0.26) enabled.

This type of circular replication setup is shown in the following diagram:

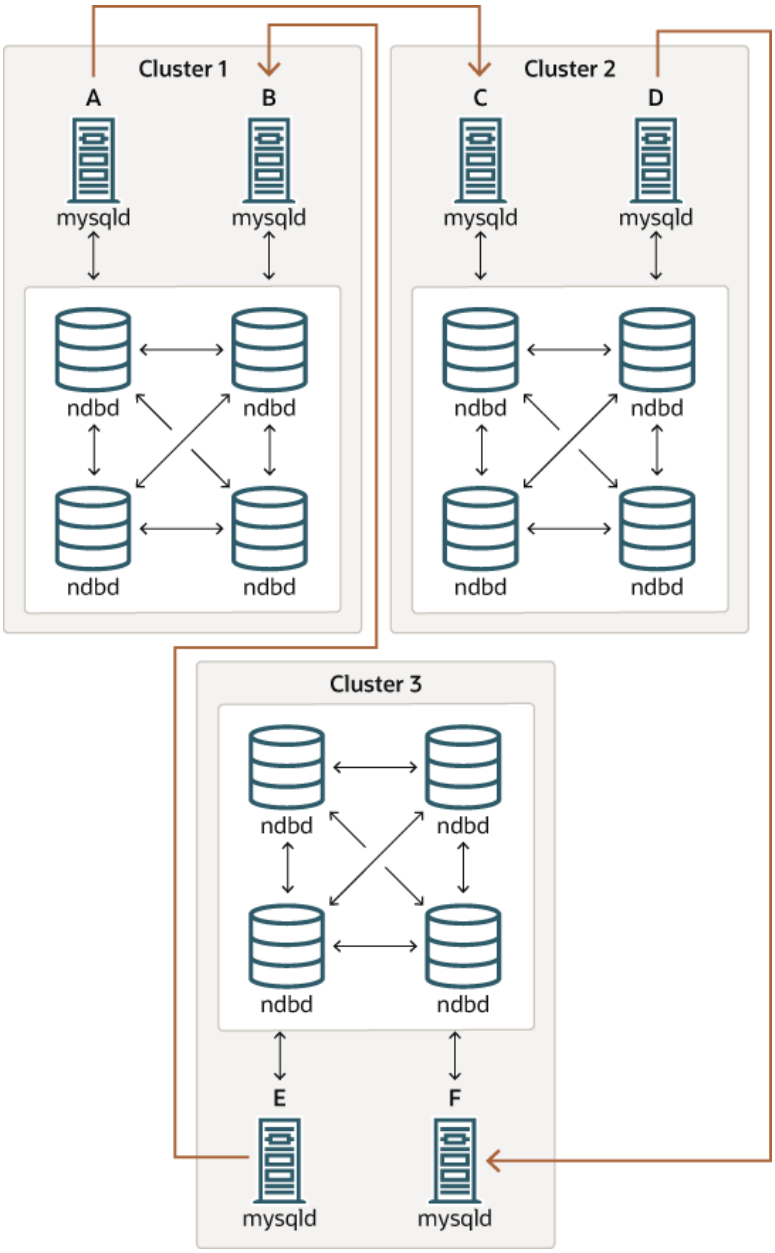
Figure 23.11 NDB Cluster Circular Replication With All Sources As Replicas



In this scenario, SQL node A in Cluster 1 replicates to SQL node C in Cluster 2; SQL node C replicates to SQL node E in Cluster 3; SQL node E replicates to SQL node A. In other words, the replication line (indicated by the curved arrows in the diagram) directly connects all SQL nodes used as sources and replicas.

It should also be possible to set up circular replication in which not all source SQL nodes are also replicas, as shown here:

Figure 23.12 NDB Cluster Circular Replication Where Not All Sources Are Replicas



In this case, different SQL nodes in each cluster are used as sources and replicas. However, you must *not* start any of the SQL nodes with the `log_replica_updates` or `log_slave_updates` system variable enabled. This type of circular replication scheme for NDB Cluster, in which the line of replication (again indicated by the curved arrows in the diagram) is discontinuous, should be possible, but it should be noted that it has not yet been thoroughly tested and must therefore still be considered experimental.

Note

The `NDB` storage engine uses *idempotent execution mode*, which suppresses duplicate-key and other errors that otherwise break circular replication of NDB Cluster. This is equivalent to setting the global value of the system variable `replica_exec_mode` or `slave_exec_mode` to `IDEMPOTENT`, although this is not necessary in NDB Cluster replication, since NDB Cluster sets this variable automatically and ignores any attempts to set it explicitly.

NDB Cluster replication and primary keys. In the event of a node failure, errors in replication of `NDB` tables without primary keys can still occur, due to the possibility of duplicate rows being inserted in such cases. For this reason, it is highly recommended that all `NDB` tables being replicated have explicit primary keys.

NDB Cluster Replication and Unique Keys. In older versions of NDB Cluster, operations that updated values of unique key columns of `NDB` tables could result in duplicate-key errors when replicated. This issue is solved for replication between `NDB` tables by deferring unique key checks until after all table row updates have been performed.

Deferring constraints in this way is currently supported only by `NDB`. Thus, updates of unique keys when replicating from `NDB` to a different storage engine such as `InnoDB` or `MyISAM` are still not supported.

The problem encountered when replicating without deferred checking of unique key updates can be illustrated using `NDB` table such as `t`, is created and populated on the source (and transmitted to a replica that does not support deferred unique key updates) as shown here:

```
CREATE TABLE t (
    p INT PRIMARY KEY,
    c INT,
    UNIQUE KEY u (c)
)   ENGINE NDB;

INSERT INTO t
    VALUES (1,1), (2,2), (3,3), (4,4), (5,5);
```

The following `UPDATE` statement on `t` succeeds on the source, since the rows affected are processed in the order determined by the `ORDER BY` option, performed over the entire table:

```
UPDATE t SET c = c - 1 ORDER BY p;
```

The same statement fails with a duplicate key error or other constraint violation on the replica, because the ordering of the row updates is performed for one partition at a time, rather than for the table as a whole.

Note

Every `NDB` table is implicitly partitioned by key when it is created. See Section 24.2.5, “KEY Partitioning”, for more information.

GTIDs not supported. Replication using global transaction IDs is not compatible with the `NDB` storage engine, and is not supported. Enabling GTIDs is likely to cause NDB Cluster Replication to fail.

Multithreaded replicas. NDB Cluster does not support multithreaded replicas. This is because the replica may not be able to separate transactions occurring in one database from those in another if they are written within the same epoch. In addition, every transaction handled by the `NDB` storage engine involves at least two databases—the target database and the `mysql` system database—due to the requirement for updating the `mysql.ndb_apply_status` table (see Section 23.7.4, “NDB Cluster Replication Schema and Tables”). This in turn breaks the requirement for multithreading that the transaction is specific to a given database.

Prior to NDB 8.0.26, setting any system variables relating to multithreaded replicas such as `replica_parallel_workers` or `slave_parallel_workers`, and `replica_checkpoint_group` or `slave_checkpoint_group` (or the equivalent `mysqld` startup options) was completely ignored, and had no effect.

In NDB 8.0.27 through NDB 8.0.32, `replica_parallel_workers` must be set to 0. In these versions, if this is set to any other value on startup, `NDB` changes it to 0, and writes a message to the `mysqld` server log file. This restriction is lifted in NDB 8.0.33.

Restarting with --initial. Restarting the cluster with the `--initial` option causes the sequence of GCI and epoch numbers to start over from `0`. (This is generally true of NDB Cluster and not

limited to replication scenarios involving Cluster.) The MySQL servers involved in replication should in this case be restarted. After this, you should use the `RESET MASTER` and `RESET REPLICA` (prior to NDB 8.0.22, use `RESET SLAVE`) statements to clear the invalid `ndb_binlog_index` and `ndb_apply_status` tables, respectively.

Replication from NDB to other storage engines. It is possible to replicate an `NDB` table on the source to a table using a different storage engine on the replica, taking into account the restrictions listed here:

- Multi-source and circular replication are not supported (tables on both the source and the replica must use the `NDB` storage engine for this to work).
- Using a storage engine which does not perform binary logging for tables on the replica requires special handling.
- Use of a nontransactional storage engine for tables on the replica also requires special handling.
- The source `mysqld` must be started with `--ndb-log-update-as-write=0` or `--ndb-log-update-as-write=OFF`.

The next few paragraphs provide additional information about each of the issues just described.

Multiple sources not supported when replicating NDB to other storage engines. For replication from `NDB` to a different storage engine, the relationship between the two databases must be one-to-one. This means that bidirectional or circular replication is not supported between NDB Cluster and other storage engines.

In addition, it is not possible to configure more than one replication channel when replicating between `NDB` and a different storage engine. (An NDB Cluster database *can* simultaneously replicate to multiple NDB Cluster databases.) If the source uses `NDB` tables, it is still possible to have more than one MySQL Server maintain a binary log of all changes, but for the replica to change sources (fail over), the new source-replica relationship must be explicitly defined on the replica.

Replicating NDB tables to a storage engine that does not perform binary logging. If you attempt to replicate from an NDB Cluster to a replica that uses a storage engine that does not handle its own binary logging, the replication process aborts with the error `Binary logging not possible ... Statement cannot be written atomically since more than one engine involved and at least one engine is self-logging` (Error `1595`). It is possible to work around this issue in one of the following ways:

- **Turn off binary logging on the replica.** This can be accomplished by setting `sql_log_bin = 0`.
- **Change the storage engine used for the mysql.ndb_apply_status table.** Causing this table to use an engine that does not handle its own binary logging can also eliminate the conflict. This can be done by issuing a statement such as `ALTER TABLE mysql.ndb_apply_status ENGINE=MyISAM` on the replica. It is safe to do this when using a storage engine other than `NDB` on the replica, since you do not need to worry about keeping multiple replicas synchronized.
- **Filter out changes to the mysql.ndb_apply_status table on the replica.** This can be done by starting the replica with `--replicate-ignore-table=mysql.ndb_apply_status`. If you need for other tables to be ignored by replication, you might wish to use an appropriate `--replicate-wild-ignore-table` option instead.

Important

You should *not* disable replication or binary logging of `mysql.ndb_apply_status` or change the storage engine used for this table when replicating from one NDB Cluster to another. See Replication and binary log filtering rules with replication between NDB Clusters, for details.

Replication from NDB to a nontransactional storage engine. When replicating from `NDB` to a nontransactional storage engine such as `MyISAM`, you may encounter unnecessary duplicate key errors when replicating `INSERT ... ON DUPLICATE KEY UPDATE` statements. You can suppress these by using `--ndb-log-update-as-write=0`, which forces updates to be logged as writes, rather than as updates.

Replication and binary log filtering rules with replication between NDB Clusters. If you are using any of the options `--replicate-do-*`, `--replicate-ignore-*`, `--binlog-do-db`, or `--binlog-ignore-db` to filter databases or tables being replicated, you must take care not to block replication or binary logging of the `mysql.ndb_apply_status`, which is required for replication between NDB Clusters to operate properly. In particular, you must keep in mind the following:

1. Using `--replicate-do-db=`*`db_name`* (and no other `--replicate-do-*` or `--replicate-ignore-*` options) means that *only* tables in database *`db_name`* are replicated. In this case, you should also use `--replicate-do-db=mysql`, `--binlog-do-db=mysql`, or `--replicate-do-table=mysql.ndb_apply_status` to ensure that `mysql.ndb_apply_status` is populated on replicas.

   Using `--binlog-do-db=`*`db_name`* (and no other `--binlog-do-db` options) means that changes *only* to tables in database *`db_name`* are written to the binary log. In this case, you should also use `--replicate-do-db=mysql`, `--binlog-do-db=mysql`, or `--replicate-do-table=mysql.ndb_apply_status` to ensure that `mysql.ndb_apply_status` is populated on replicas.

2. Using `--replicate-ignore-db=mysql` means that no tables in the `mysql` database are replicated. In this case, you should also use `--replicate-do-table=mysql.ndb_apply_status` to ensure that `mysql.ndb_apply_status` is replicated.

   Using `--binlog-ignore-db=mysql` means that no changes to tables in the `mysql` database are written to the binary log. In this case, you should also use `--replicate-do-table=mysql.ndb_apply_status` to ensure that `mysql.ndb_apply_status` is replicated.

You should also remember that each replication rule requires the following:

1. Its own `--replicate-do-*` or `--replicate-ignore-*` option, and that multiple rules cannot be expressed in a single replication filtering option. For information about these rules, see Section 17.1.6, “Replication and Binary Logging Options and Variables”.

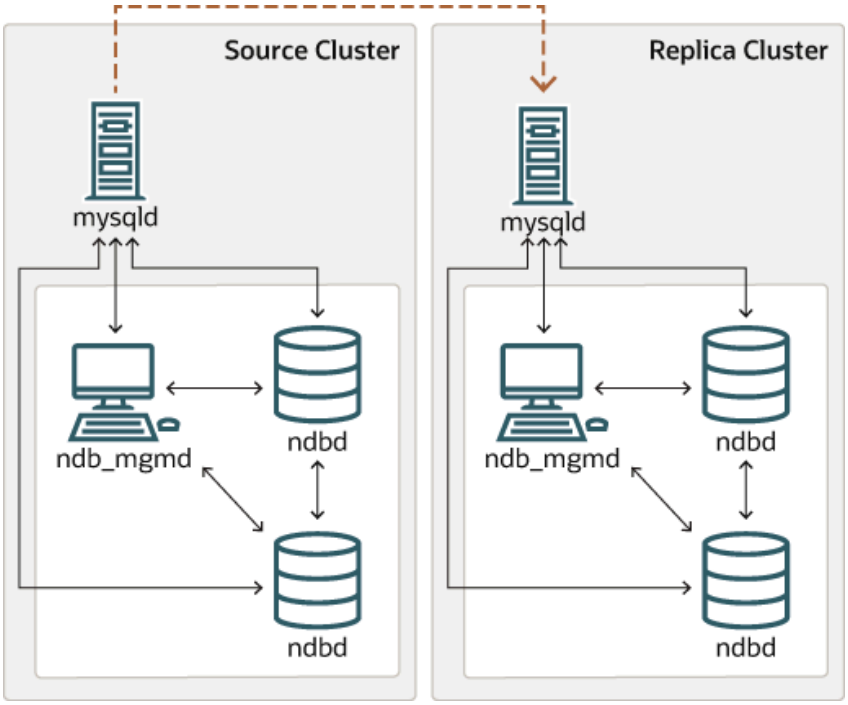
2. Its own `--binlog-do-db` or `--binlog-ignore-db` option, and that multiple rules cannot be expressed in a single binary log filtering option. For information about these rules, see Section 5.4.4, “The Binary Log”.

If you are replicating an NDB Cluster to a replica that uses a storage engine other than `NDB`, the considerations just given previously may not apply, as discussed elsewhere in this section.

NDB Cluster Replication and IPv6. Beginning with NDB 8.0.22, all types of NDB Cluster nodes support IPv6; this includes management nodes, data nodes, and API or SQL nodes.

Prior to NDB 8.0.22, the NDB API and MGM API (and thus data nodes and management nodes) do not support IPv6, although MySQL Servers—including those acting as SQL nodes in an NDB Cluster—can use IPv6 to contact other MySQL Servers. In versions of NDB Cluster prior to 8.0.22, you can replicate between clusters using IPv6 to connect the SQL nodes acting as source and replica as shown by the dotted arrow in the following diagram:

Figure 23.13 Replication Between SQL Nodes Connected Using IPv6



Prior to NDB 8.0.22, all connections originating *within* the NDB Cluster —represented in the preceding diagram by solid arrows—must use IPv4. In other words, all NDB Cluster data nodes, management servers, and management clients must be accessible from one another using IPv4. In addition, SQL nodes must use IPv4 to communicate with the cluster. In NDB 8.0.22 and later, these restrictions no longer apply; in addition, any applications written using the NDB and MGM APIs can be written and deployed assuming an IPv6-only environment.

Attribute promotion and demotion. NDB Cluster Replication includes support for attribute promotion and demotion. The implementation of the latter distinguishes between lossy and non-lossy type conversions, and their use on the replica can be controlled by setting the global value of the system variable `replica_type_conversions` (NDB 8.0.26 and later) or `slave_type_conversions` (prior to NDB 8.0.26).

For more information about attribute promotion and demotion in NDB Cluster, see Row-based replication: attribute promotion and demotion.

`NDB`, unlike `InnoDB` or `MyISAM`, does not write changes to virtual columns to the binary log; however, this has no detrimental effects on NDB Cluster Replication or replication between `NDB` and other storage engines. Changes to stored generated columns are logged.

23.7.4 NDB Cluster Replication Schema and Tables

- ndb_apply_status Table
- ndb_binlog_index Table
- ndb_replication Table

Replication in NDB Cluster makes use of a number of dedicated tables in the `mysql` database on each MySQL Server instance acting as an SQL node in both the cluster being replicated and in the replica. This is true regardless of whether the replica is a single server or a cluster.

The `ndb_binlog_index` and `ndb_apply_status` tables are created in the `mysql` database. They should not be explicitly replicated by the user. User intervention is normally not required to create or maintain either of these tables, since both are maintained by the `NDB` binary log (binlog) injector thread.

This keeps the source `mysqld` process updated to changes performed by the `NDB` storage engine. The `NDB` *binlog injector thread* receives events directly from the `NDB` storage engine. The `NDB` injector is responsible for capturing all the data events within the cluster, and ensures that all events which change, insert, or delete data are recorded in the `ndb_binlog_index` table. The replica I/O (receiver) thread transfers the events from the source's binary log to the replica's relay log.

The `ndb_replication` table must be created manually. This table can be updated by the user to perform filtering by database or table. See ndb_replication Table, for more information. `ndb_replication` is also used in NDB Replication conflict detection and resolution for conflict resolution control; see Conflict Resolution Control.

Even though `ndb_binlog_index` and `ndb_apply_status` are created and maintained automatically, it is advisable to check for the existence and integrity of these tables as an initial step in preparing an NDB Cluster for replication. It is possible to view event data recorded in the binary log by querying the `mysql.ndb_binlog_index` table directly on the source. This can be also be accomplished using the `SHOW BINLOG EVENTS` statement on either the source or replica SQL node. (See Section 13.7.7.2, “SHOW BINLOG EVENTS Statement”.)

You can also obtain useful information from the output of `SHOW ENGINE NDB STATUS`.

Note

When performing schema changes on `NDB` tables, applications should wait until the `ALTER TABLE` statement has returned in the MySQL client connection that issued the statement before attempting to use the updated definition of the table.

ndb_apply_status Table

`ndb_apply_status` is used to keep a record of the operations that have been replicated from the source to the replica. If the `ndb_apply_status` table does not exist on the replica, `ndb_restore` re-creates it.

Unlike the case with `ndb_binlog_index`, the data in this table is not specific to any one SQL node in the (replica) cluster, and so `ndb_apply_status` can use the `NDBCLUSTER` storage engine, as shown here:

```
CREATE TABLE `ndb_apply_status` (
    `server_id`   INT(10) UNSIGNED NOT NULL,
    `epoch`       BIGINT(20) UNSIGNED NOT NULL,
    `log_name`    VARCHAR(255) CHARACTER SET latin1 COLLATE latin1_bin NOT NULL,
    `start_pos`   BIGINT(20) UNSIGNED NOT NULL,
    `end_pos`     BIGINT(20) UNSIGNED NOT NULL,
    PRIMARY KEY (`server_id`) USING HASH
) ENGINE=NDBCLUSTER   DEFAULT CHARSET=latin1;
```

The `ndb_apply_status` table is populated only on replicas, which means that, on the source, this table never contains any rows; thus, there is no need to allot any `DataMemory` to `ndb_apply_status` there.

Because this table is populated from data originating on the source, it should be allowed to replicate; any replication filtering or binary log filtering rules that inadvertently prevent the replica from updating `ndb_apply_status`, or that prevent the source from writing into the binary log may prevent replication between clusters from operating properly. For more information about potential problems arising from such filtering rules, see Replication and binary log filtering rules with replication between NDB Clusters.

It is possible to delete this table, but this is not recommended. Deleting it puts all SQL nodes in read-only mode; in NDB 8.0.24 and later, `NDB` detects that this table has been dropped, and re-creates it, after which it is possible once again to perform updates. Dropping and re-creating `ndb_apply_status` creates a gap event in the binary log; the gap event causes replica SQL nodes to

stop applying changes from the source until the replication channel is restarted. Prior to NDB 8.0.24, it was necessary in such cases to restart all SQL nodes to bring them out of read-only mode, and then to re-create `ndb_apply_status` manually.

`0` in the `epoch` column of this table indicates a transaction originating from a storage engine other than `NDB`.

`ndb_apply_status` is used to record which epoch transactions have been replicated and applied to a replica cluster from an upstream source. This information is captured in an `NDB` online backup, but (by design) it is not restored by `ndb_restore`. In some cases, it can be helpful to restore this information for use in new setups; beginning with NDB 8.0.29, you can do this by invoking `ndb_restore` with the `--with-apply-status` option. See the description of the option for more information.

ndb_binlog_index Table

NDB Cluster Replication uses the `ndb_binlog_index` table for storing the binary log's indexing data. Since this table is local to each MySQL server and does not participate in clustering, it uses the `InnoDB` storage engine. This means that it must be created separately on each `mysqld` participating in the source cluster. (The binary log itself contains updates from all MySQL servers in the cluster.) This table is defined as follows:

```
CREATE TABLE `ndb_binlog_index` (
    `Position` BIGINT(20) UNSIGNED NOT NULL,
    `File` VARCHAR(255) NOT NULL,
    `epoch` BIGINT(20) UNSIGNED NOT NULL,
    `inserts` INT(10) UNSIGNED NOT NULL,
    `updates` INT(10) UNSIGNED NOT NULL,
    `deletes` INT(10) UNSIGNED NOT NULL,
    `schemaops` INT(10) UNSIGNED NOT NULL,
    `orig_server_id` INT(10) UNSIGNED NOT NULL,
    `orig_epoch` BIGINT(20) UNSIGNED NOT NULL,
    `gci` INT(10) UNSIGNED NOT NULL,
    `next_position` bigint(20) unsigned NOT NULL,
    `next_file` varchar(255) NOT NULL,
    PRIMARY KEY (`epoch`,`orig_server_id`,`orig_epoch`)
) ENGINE=InnoDB DEFAULT CHARSET=latin1;
```

Note

If you are upgrading from an older release (prior to NDB 7.5.2), perform the MySQL upgrade procedure and ensure that the system tables are upgraded by starting the MySQL server with the `--upgrade=FORCE` option. The system table upgrade causes an `ALTER TABLE ... ENGINE=INNODB` statement to be executed for this table. Use of the `MyISAM` storage engine for this table continues to be supported for backward compatibility.

`ndb_binlog_index` may require additional disk space after being converted to `InnoDB`. If this becomes an issue, you may be able to conserve space by using an `InnoDB` tablespace for this table, changing its `ROW_FORMAT` to `COMPRESSED`, or both. For more information, see Section 13.1.21, “CREATE TABLESPACE Statement”, and Section 13.1.20, “CREATE TABLE Statement”, as well as Section 15.6.3, “Tablespaces”.

The size of the `ndb_binlog_index` table is dependent on the number of epochs per binary log file and the number of binary log files. The number of epochs per binary log file normally depends on the amount of binary log generated per epoch and the size of the binary log file, with smaller epochs resulting in more epochs per file. You should be aware that empty epochs produce inserts to the `ndb_binlog_index` table, even when the `--ndb-log-empty-epochs` option is `OFF`, meaning that the number of entries per file depends on the length of time that the file is in use; this relationship can be represented by the formula shown here:

```
[number of epochs per file] = [time spent per file] / TimeBetweenEpochs
```

A busy NDB Cluster writes to the binary log regularly and presumably rotates binary log files more quickly than a quiet one. This means that a “quiet” NDB Cluster with `--ndb-log-empty-epochs=ON` can actually have a much higher number of `ndb_binlog_index` rows per file than one with a great deal of activity.

When `mysqld` is started with the `--ndb-log-orig` option, the `orig_server_id` and `orig_epoch` columns store, respectively, the ID of the server on which the event originated and the epoch in which the event took place on the originating server, which is useful in NDB Cluster replication setups employing multiple sources. The `SELECT` statement used to find the closest binary log position to the highest applied epoch on the replica in a multi-source setup (see Section 23.7.10, “NDB Cluster Replication: Bidirectional and Circular Replication”) employs these two columns, which are not indexed. This can lead to performance issues when trying to fail over, since the query must perform a table scan, especially when the source has been running with `--ndb-log-empty-epochs=ON`. You can improve multi-source failover times by adding an index to these columns, as shown here:

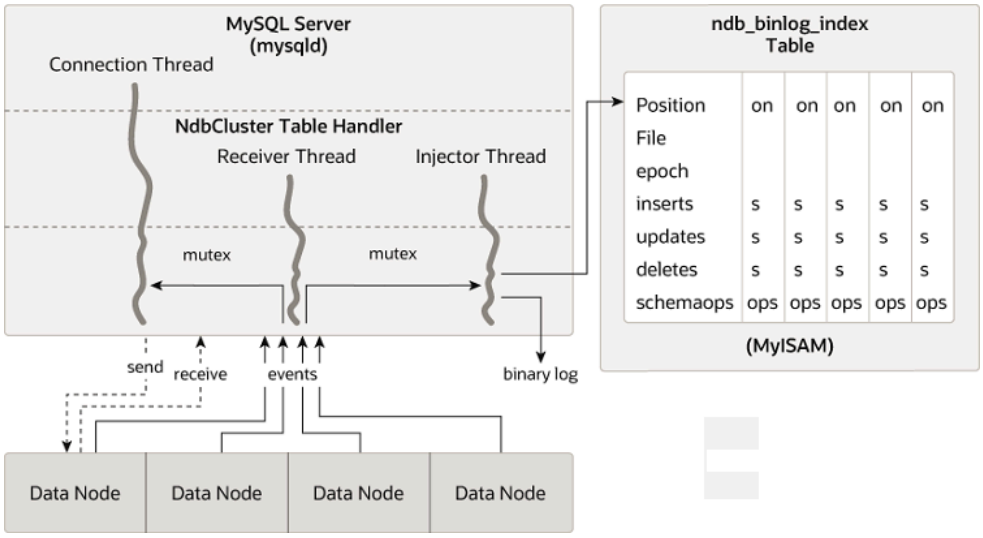
```
ALTER TABLE mysql.ndb_binlog_index
    ADD INDEX orig_lookup USING BTREE (orig_server_id, orig_epoch);
```

Adding this index provides no benefit when replicating from a single source to a single replica, since the query used to get the binary log position in such cases makes no use of `orig_server_id` or `orig_epoch`.

See Section 23.7.8, “Implementing Failover with NDB Cluster Replication”, for more information about using the `next_position` and `next_file` columns.

The following figure shows the relationship of the NDB Cluster replication source server, its binary log injector thread, and the `mysql.ndb_binlog_index` table.

Figure 23.14 The Replication Source Cluster



ndb_replication Table

The `ndb_replication` table is used to control binary logging and conflict resolution, and acts on a per-table basis. Each row in this table corresponds to a table being replicated, determines how to log changes to the table and, if a conflict resolution function is specified, and determines how to resolve conflicts for that table.

Unlike the `ndb_apply_status` and `ndb_replication` tables, the `ndb_replication` table must be created manually, using the SQL statement shown here:

```
CREATE TABLE mysql.ndb_replication  (
    db VARBINARY(63),
```

```
    table_name VARBINARY(63),
    server_id INT UNSIGNED,
    binlog_type INT UNSIGNED,
    conflict_fn VARBINARY(128),
    PRIMARY KEY USING HASH (db, table_name, server_id)
)   ENGINE=NDB
PARTITION BY KEY(db,table_name);
```

The columns of this table are listed here, with descriptions:

- `db` column

  The name of the database containing the table to be replicated.

  You may employ either or both of the wildcards `_` and `%` as part of the database name. (See Matching with wildcards, later in this section.)

- `table_name` column

  The name of the table to be replicated.

  The table name may include either or both of the wildcards `_` and `%`. See Matching with wildcards, later in this section.

- `server_id` column

  The unique server ID of the MySQL instance (SQL node) where the table resides.

  `0` in this column acts like a wildcard equivalent to `%`, and matches any server ID. (See Matching with wildcards, later in this section.)

- `binlog_type` column

  The type of binary logging to be employed. See text for values and descriptions.

- `conflict_fn` column

  The conflict resolution function to be applied; one of NDB$OLD(), NDB$MAX(), NDB$MAX_DELETE_WIN(), NDB$EPOCH(), NDB$EPOCH_TRANS(), NDB$EPOCH2(), NDB$EPOCH2_TRANS(); `NULL` indicates that conflict resolution is not used for this table. NDB 8.0.30 and later supports two additional conflict resolution functions NDB$MAX_INS() and NDB$MAX_DEL_WIN_INS().

  See Conflict Resolution Functions, for more information about these functions and their uses in NDB Replication conflict resolution.

  Some conflict resolution functions (`NDB$OLD()`, `NDB$EPOCH()`, `NDB$EPOCH_TRANS()`) require the use of one or more user-created exceptions tables. See Conflict Resolution Exceptions Table.

To enable conflict resolution with NDB Replication, it is necessary to create and populate this table with control information on the SQL node or nodes on which the conflict should be resolved. Depending on the conflict resolution type and method to be employed, this may be the source, the replica, or both servers. In a simple source-replica setup where data can also be changed locally on the replica this is typically the replica. In a more complex replication scheme, such as bidirectional replication, this is usually all of the sources involved. See Section 23.7.12, “NDB Cluster Replication Conflict Resolution”, for more information.

The `ndb_replication` table allows table-level control over binary logging outside the scope of conflict resolution, in which case `conflict_fn` is specified as `NULL`, while the remaining column values are used to control binary logging for a given table or set of tables matching a wildcard expression. By setting the proper value for the `binlog_type` column, you can make logging for a given table or tables use a desired binary log format, or disabling binary logging altogether. Possible values for this column, with values and descriptions, are shown in the following table:

Table 23.70 binlog_type values, with values and descriptions

Value	Description
0	Use server default
1	Do not log this table in the binary log (same effect as `sql_log_bin = 0`, but applies to one or more specified tables only)
2	Log updated attributes only; log these as `WRITE_ROW` events
3	Log full row, even if not updated (MySQL server default behavior)
6	Use updated attributes, even if values are unchanged
7	Log full row, even if no values are changed; log updates as `UPDATE_ROW` events
8	Log update as `UPDATE_ROW`; log only primary key columns in before image, and only updated columns in after image (same effect as `--ndb-log-update-minimal`, but applies to one or more specified tables only)
9	Log update as `UPDATE_ROW`; log only primary key columns in before image, and all columns other than primary key columns in after image

Note

`binlog_type` values 4 and 5 are not used, and so are omitted from the table just shown, as well as from the next table.

Several `binlog_type` values are equivalent to various combinations of the `mysqld` logging options `--ndb-log-updated-only`, `--ndb-log-update-as-write`, and `--ndb-log-update-minimal`, as shown in the following table:

Table 23.71 binlog_type values with equivalent combinations of NDB logging options

Value	`--ndb-log-updated-only` Value	`--ndb-log-update-as-write` Value	`--ndb-log-update-minimal` Value
0	--	--	--
1	--	--	--
2	ON	ON	OFF
3	OFF	ON	OFF
6	ON	OFF	OFF
7	OFF	OFF	OFF
8	ON	OFF	ON
9	OFF	OFF	ON

Binary logging can be set to different formats for different tables by inserting rows into the `ndb_replication` table using the appropriate `db`, `table_name`, and `binlog_type` column values. The internal integer value shown in the preceding table should be used when setting the binary logging format. The following two statements set binary logging to logging of full rows (value 3) for table `test.a`, and to logging of updates only (value 2) for table `test.b`:

```
# Table test.a: Log full rows
INSERT INTO mysql.ndb_replication VALUES("test", "a", 0, 3, NULL);
```

```
# Table test.b: log updates only
INSERT INTO mysql.ndb_replication VALUES("test", "b", 0, 2, NULL);
```

To disable logging for one or more tables, use 1 for `binlog_type`, as shown here:

```
# Disable binary logging for table test.t1
INSERT INTO mysql.ndb_replication VALUES("test", "t1", 0, 1, NULL);

# Disable binary logging for any table in 'test' whose name begins with 't'
INSERT INTO mysql.ndb_replication VALUES("test", "t%", 0, 1, NULL);
```

Disabling logging for a given table is the equivalent of setting `sql_log_bin = 0`, except that it applies to one or more tables individually. If an SQL node is not performing binary logging for a given table, it is not sent the row change events for those tables. This means that it is not receiving all changes and discarding some, but rather it is not subscribing to these changes.

Disabling logging can be useful for a number of reasons, including those listed here:

- Not sending changes across the network generally saves bandwidth, buffering, and CPU resources.
- Not logging changes to tables with very frequent updates but whose value is not great is a good fit for transient data (such as session data) that may be relatively unimportant in the event of a complete failure of the cluster.
- Using a session variable (or `sql_log_bin`) and application code, it is also possible to log (or not to log) certain SQL statements or types of SQL statements; for example, it may be desirable in some cases not to record DDL statements on one or more tables.
- Splitting replication streams into two (or more) binary logs can be done for reasons of performance, a need to replicate different databases to different places, use of different binary logging types for different databases, and so on.

Matching with wildcards. In order not to make it necessary to insert a row in the `ndb_replication` table for each and every combination of database, table, and SQL node in your replication setup, `NDB` supports wildcard matching on the this table's `db`, `table_name`, and `server_id` columns. Database and table names used in, respectively, `db` and `table_name` may contain either or both of the following wildcards:

- `_` (underscore character): matches zero or more characters
- `%` (percent sign): matches a single character

(These are the same wildcards as supported by the MySQL `LIKE` operator.)

The `server_id` column supports `0` as a wildcard equivalent to `_` (matches anything). This is used in the examples shown previously.

A given row in the `ndb_replication` table can use wildcards to match any of the database name, table name, and server ID in any combination. Where there are multiple potential matches in the table, the best match is chosen, according to the table shown here, where *W* represents a wildcard match, *E* an exact match, and the greater the value in the *Quality* column, the better the match:

Table 23.72 Weights of different combinations of wildcard and exact matches on columns in the mysql.ndb_replication table

db	table_name	server_id	Quality
W	W	W	1
W	W	E	2
W	E	W	3

db	table_name	server_id	Quality
W	E	E	4
E	W	W	5
E	W	E	6
E	E	W	7
E	E	E	8

Thus, an exact match on database name, table name, and server ID is considered best (strongest), while the weakest (worst) match is a wildcard match on all three columns. Only the strength of the match is considered when choosing which rule to apply; the order in which the rows occur in the table has no effect on this determination.

Logging Full or Partial Rows. There are two basic methods of logging rows, as determined by the setting of the `--ndb-log-updated-only` option for `mysqld`:

- Log complete rows (option set to `ON`)
- Log only column data that has been updated—that is, column data whose value has been set, regardless of whether or not this value was actually changed. This is the default behavior (option set to `OFF`).

It is usually sufficient—and more efficient—to log updated columns only; however, if you need to log full rows, you can do so by setting `--ndb-log-updated-only` to `0` or `OFF`.

Logging Changed Data as Updates. The setting of the MySQL Server's `--ndb-log-update-as-write` option determines whether logging is performed with or without the “before” image.

Because conflict resolution for updates and delete operations is done in the MySQL Server's update handler, it is necessary to control the logging performed by the replication source such that updates are updates and not writes; that is, such that updates are treated as changes in existing rows rather than the writing of new rows, even though these replace existing rows.

This option is turned on by default; in other words, updates are treated as writes. That is, updates are by default written as `write_row` events in the binary log, rather than as `update_row` events.

To disable the option, start the source `mysqld` with `--ndb-log-update-as-write=0` or `--ndb-log-update-as-write=OFF`. You must do this when replicating from NDB tables to tables using a different storage engine; see Replication from NDB to other storage engines, and Replication from NDB to a nontransactional storage engine, for more information.

Important

(*NDB 8.0.30 and later*:) For insert conflict resolution using `NDB$MAX_INS()` or `NDB$MAX_DEL_WIN_INS()`, an SQL node (that is, a `mysqld` process) can record row updates on the source cluster as `WRITE_ROW` events with the `--ndb-log-update-as-write` option enabled for idempotency and optimal size. This works for these algorithms since they both map a `WRITE_ROW` event to an insert or update depending on whether the row already exists, and the required metadata (the “after” image for the timestamp column) is present in the “WRITE_ROW” event.

23.7.5 Preparing the NDB Cluster for Replication

Preparing the NDB Cluster for replication consists of the following steps:

1. Check all MySQL servers for version compatibility (see Section 23.7.2, “General Requirements for NDB Cluster Replication”).

2. Create a replication account on the source Cluster with the appropriate privileges, using the following two SQL statements:

```
mysqlS> CREATE USER 'replica_user'@'replica_host'
     -> IDENTIFIED BY 'replica_password';

mysqlS> GRANT REPLICATION SLAVE ON *.*
     -> TO 'replica_user'@'replica_host';
```

In the previous statement, *`replica_user`* is the replication account user name, *`replica_host`* is the host name or IP address of the replica, and *`replica_password`* is the password to assign to this account.

For example, to create a replica user account with the name `myreplica`, logging in from the host named `replica-host`, and using the password `53cr37`, use the following `CREATE USER` and `GRANT` statements:

```
mysqlS> CREATE USER 'myreplica'@'replica-host'
     -> IDENTIFIED BY '53cr37';

mysqlS> GRANT REPLICATION SLAVE ON *.*
     -> TO 'myreplica'@'replica-host';
```

For security reasons, it is preferable to use a unique user account—not employed for any other purpose—for the replication account.

3. Set up the replica to use the source. Using the `mysql` client, this can be accomplished with the `CHANGE REPLICATION SOURCE TO` statement (beginning with NDB 8.0.23) or `CHANGE MASTER TO` statement (prior to NDB 8.0.23):

```
mysqlR> CHANGE MASTER TO
     -> MASTER_HOST='source_host',
     -> MASTER_PORT=source_port,
     -> MASTER_USER='replica_user',
     -> MASTER_PASSWORD='replica_password';
```

Beginning with NDB 8.0.23, you can also use the following statement:

```
mysqlR> CHANGE REPLICATION SOURCE TO
     -> SOURCE_HOST='source_host',
     -> SOURCE_PORT=source_port,
     -> SOURCE_USER='replica_user',
     -> SOURCE_PASSWORD='replica_password';
```

In the previous statement, *`source_host`* is the host name or IP address of the replication source, *`source_port`* is the port for the replica to use when connecting to the source, *`replica_user`* is the user name set up for the replica on the source, and *`replica_password`* is the password set for that user account in the previous step.

For example, to tell the replica to use the MySQL server whose host name is `rep-source` with the replication account created in the previous step, use the following statement:

```
mysqlR> CHANGE MASTER TO
     -> MASTER_HOST='rep-source',
     -> MASTER_PORT=3306,
     -> MASTER_USER='myreplica',
     -> MASTER_PASSWORD='53cr37';
```

Beginning with NDB 8.0.23, you can also use the following statement:

```
mysqlR> CHANGE REPLICATION SOURCE TO
     -> SOURCE_HOST='rep-source',
     -> SOURCE_PORT=3306,
     -> SOURCE_USER='myreplica',
     -> SOURCE_PASSWORD='53cr37';
```

For a complete list of options that can be used with this statement, see Section 13.4.2.1, “CHANGE MASTER TO Statement”.

To provide replication backup capability, you also need to add an `--ndb-connectstring` option to the replica's `my.cnf` file prior to starting the replication process. See Section 23.7.9, “NDB Cluster Backups With NDB Cluster Replication”, for details.

For additional options that can be set in `my.cnf` for replicas, see Section 17.1.6, “Replication and Binary Logging Options and Variables”.

4. If the source cluster is already in use, you can create a backup of the source and load this onto the replica to cut down on the amount of time required for the replica to synchronize itself with the source. If the replica is also running NDB Cluster, this can be accomplished using the backup and restore procedure described in Section 23.7.9, “NDB Cluster Backups With NDB Cluster Replication”.

   ```
   ndb-connectstring=management_host[:port]
   ```

   In the event that you are *not* using NDB Cluster on the replica, you can create a backup with this command on the source:

   ```
   shellS> mysqldump --master-data=1
   ```

   Then import the resulting data dump onto the replica by copying the dump file over to it. After this, you can use the `mysql` client to import the data from the dumpfile into the replica database as shown here, where *`dump_file`* is the name of the file that was generated using `mysqldump` on the source, and *`db_name`* is the name of the database to be replicated:

   ```
   shellR> mysql -u root -p db_name < dump_file
   ```

   For a complete list of options to use with `mysqldump`, see Section 4.5.4, “mysqldump — A Database Backup Program”.

   Note

   If you copy the data to the replica in this fashion, make sure that you stop the replica from trying to connect to the source to begin replicating before all the data has been loaded. You can do this by starting the replica with the `--skip-slave-start` option on the command line, by including `skip-slave-start` in the replica's `my.cnf` file, or beginning with NDB 8.0.24, by setting the `skip_slave_start` system variable. Beginning with NDB 8.0.26, use `--skip-replica-start` or `skip_replica_start` instead. Once the data loading has completed, follow the additional steps outlined in the next two sections.

5. Ensure that each MySQL server acting as a replication source is assigned a unique server ID, and has binary logging enabled, using the row-based format. (See Section 17.2.1, “Replication Formats”.) In addition, we strongly recommend enabling the `replica_allow_batching` system variable (NDB 8.0.26 and later; prior to NDB 8.0.26, use `slave_allow_batching`). Beginning with NDB 8.0.30, this is enabled by default.

   If you are using a release of NDB Cluster prior to NDB 8.0.30, you should also consider increasing the values used with the `--ndb-batch-size` and `--ndb-blob-write-batch-bytes` options as well. In NDB 8.0.30 and later, use `--ndb-replica-batch-size` to set the batch size used for writes on the replica instead of `--ndb-batch-size`, and `--ndb-replica-blob-write-batch-bytes` rather than `--ndb-blob-write-batch-bytes` to determine the batch size used by the replication applier for writing blob data. All of these options can be set either in the source server's `my.cnf` file, or on the command line when starting the source `mysqld` process. See Section 23.7.6, “Starting NDB Cluster Replication (Single Replication Channel)”, for more information.

23.7.6 Starting NDB Cluster Replication (Single Replication Channel)

This section outlines the procedure for starting NDB Cluster replication using a single replication channel.

1. Start the MySQL replication source server by issuing this command, where *`id`* is this server's unique ID (see Section 23.7.2, “General Requirements for NDB Cluster Replication”):

```
shellS> mysqld --ndbcluster --server-id=id \
        --log-bin --ndb-log-bin &
```

   This starts the server's `mysqld` process with binary logging enabled using the proper logging format. It is also necessary in NDB 8.0 to enable logging of updates to `NDB` tables explicitly, using the `--ndb-log-bin` option; this is a change from previous versions of NDB Cluster, in which this option was enabled by default.

   Note

   You can also start the source with `--binlog-format=MIXED`, in which case row-based replication is used automatically when replicating between clusters. Statement-based binary logging is not supported for NDB Cluster Replication (see Section 23.7.2, “General Requirements for NDB Cluster Replication”).

2. Start the MySQL replica server as shown here:

```
shellR> mysqld --ndbcluster --server-id=id &
```

   In the command just shown, *`id`* is the replica server's unique ID. It is not necessary to enable logging on the replica.

   Note

   Unless you want replication to begin immediately, delay the start of the replication threads until the appropriate `START REPLICA` statement has been issued, as explained in Step 4 below. You can do this by starting the replica with the `--skip-slave-start` option on the command line, by including `skip-slave-start` in the replica's `my.cnf` file, or in NDB 8.0.24 and later, by setting the `skip_slave_start` system variable. In NDB 8.0.26 and later, use `--skip-replica-start` and `skip_replica_start`.

3. It is necessary to synchronize the replica server with the source server's replication binary log. If binary logging has not previously been running on the source, run the following statement on the replica:

```
mysqlR> CHANGE MASTER TO
     -> MASTER_LOG_FILE='',
     -> MASTER_LOG_POS=4;
```

   Beginning with NDB 8.0.23, you can also use the following statement:

```
mysqlR> CHANGE REPLICATION SOURCE TO
     -> SOURCE_LOG_FILE='',
     -> SOURCE_LOG_POS=4;
```

   This instructs the replica to begin reading the source server's binary log from the log's starting point. Otherwise—that is, if you are loading data from the source using a backup—see Section 23.7.8, “Implementing Failover with NDB Cluster Replication”, for information on how to obtain the correct values to use for `SOURCE_LOG_FILE` | `MASTER_LOG_FILE` and `SOURCE_LOG_POS` | `MASTER_LOG_POS` in such cases.

4. Finally, instruct the replica to begin applying replication by issuing this command from the `mysql` client on the replica:

   ```
   mysqlR> START SLAVE;
   ```

   In NDB 8.0.22 and later, you can also use the following statement:

   ```
   mysqlR> START REPLICA;
   ```

   This also initiates the transmission of data and changes from the source to the replica.

It is also possible to use two replication channels, in a manner similar to the procedure described in the next section; the differences between this and using a single replication channel are covered in Section 23.7.7, “Using Two Replication Channels for NDB Cluster Replication”.

It is also possible to improve cluster replication performance by enabling *batched updates*. This can be accomplished by setting the system variable `replica_allow_batching` (NDB 8.0.26 and later) or `slave_allow_batching` (prior to NDB 8.0.26) on the replicas' `mysqld` processes. Normally, updates are applied as soon as they are received. However, the use of batching causes updates to be applied in batches of 32 KB each; this can result in higher throughput and less CPU usage, particularly where individual updates are relatively small.

Note

Batching works on a per-epoch basis; updates belonging to more than one transaction can be sent as part of the same batch.

All outstanding updates are applied when the end of an epoch is reached, even if the updates total less than 32 KB.

Batching can be turned on and off at runtime. To activate it at runtime, you can use either of these two statements:

```
SET GLOBAL slave_allow_batching = 1;
SET GLOBAL slave_allow_batching = ON;
```

Beginning with NDB 8.0.26, you can (and should) use one of the following statements:

```
SET GLOBAL replica_allow_batching = 1;
SET GLOBAL replica_allow_batching = ON;
```

If a particular batch causes problems (such as a statement whose effects do not appear to be replicated correctly), batching can be deactivated using either of the following statements:

```
SET GLOBAL slave_allow_batching = 0;
SET GLOBAL slave_allow_batching = OFF;
```

Beginning with NDB 8.0.26, you can (and should) use one of the following statements instead:

```
SET GLOBAL replica_allow_batching = 0;
SET GLOBAL replica_allow_batching = OFF;
```

You can check whether batching is currently being used by means of an appropriate `SHOW VARIABLES` statement, like this one:

```
mysql> SHOW VARIABLES LIKE 'slave%';
```

In ŃDB 8.0.26 and later, use the following statement:

```
mysql> SHOW VARIABLES LIKE 'replica%';
```

23.7.7 Using Two Replication Channels for NDB Cluster Replication

In a more complete example scenario, we envision two replication channels to provide redundancy and thereby guard against possible failure of a single replication channel. This requires a total of four replication servers, two source servers on the source cluster and two replica servers on the replica cluster. For purposes of the discussion that follows, we assume that unique identifiers are assigned as shown here:

Table 23.73 NDB Cluster replication servers described in the text

Server ID	Description
1	Source - primary replication channel (*S*)
2	Source - secondary replication channel (*S'*)
3	Replica - primary replication channel (*R*)
4	replica - secondary replication channel (*R'*)

Setting up replication with two channels is not radically different from setting up a single replication channel. First, the `mysqld` processes for the primary and secondary replication source servers must be started, followed by those for the primary and secondary replicas. The replication processes can be initiated by issuing the `START REPLICA` statement on each of the replicas. The commands and the order in which they need to be issued are shown here:

1. Start the primary replication source:

```
shellS> mysqld --ndbcluster --server-id=1 \
               --log-bin &
```

2. Start the secondary replication source:

```
shellS'> mysqld --ndbcluster --server-id=2 \
               --log-bin &
```

3. Start the primary replica server:

```
shellR> mysqld --ndbcluster --server-id=3 \
               --skip-slave-start &
```

4. Start the secondary replica server:

```
shellR'> mysqld --ndbcluster --server-id=4 \
                --skip-slave-start &
```

5. Finally, initiate replication on the primary channel by executing the `START REPLICA` statement on the primary replica as shown here:

```
mysqlR> START SLAVE;
```

   Beginning with NDB 8.0.22, you can also use the following statement:

```
mysqlR> START REPLICA;
```

Warning

Only the primary channel must be started at this point. The secondary replication channel needs to be started only in the event that the primary replication channel fails, as described in Section 23.7.8, “Implementing Failover with NDB Cluster Replication”. Running multiple replication channels simultaneously can result in unwanted duplicate records being created on the replicas.

As mentioned previously, it is not necessary to enable binary logging on the replicas.

23.7.8 Implementing Failover with NDB Cluster Replication

In the event that the primary Cluster replication process fails, it is possible to switch over to the secondary replication channel. The following procedure describes the steps required to accomplish this.

1. Obtain the time of the most recent global checkpoint (GCP). That is, you need to determine the most recent epoch from the `ndb_apply_status` table on the replica cluster, which can be found using the following query:

```
mysqlR'> SELECT @latest:=MAX(epoch)
      ->        FROM mysql.ndb_apply_status;
```

In a circular replication topology, with a source and a replica running on each host, when you are using `ndb_log_apply_status=1`, NDB Cluster epochs are written in the replicas' binary logs. This means that the `ndb_apply_status` table contains information for the replica on this host as well as for any other host which acts as a replica of the replication source server running on this host.

In this case, you need to determine the latest epoch on this replica to the exclusion of any epochs from any other replicas in this replica's binary log that were not listed in the `IGNORE_SERVER_IDS` options of the `CHANGE REPLICATION SOURCE TO` | `CHANGE MASTER TO` statement used to set up this replica. The reason for excluding such epochs is that rows in the `mysql.ndb_apply_status` table whose server IDs have a match in the `IGNORE_SERVER_IDS` list from the `CHANGE REPLICATION SOURCE TO` | `CHANGE MASTER TO` statement used to prepare this replicas's source are also considered to be from local servers, in addition to those having the replica's own server ID. You can retrieve this list as `Replicate_Ignore_Server_Ids` from the output of `SHOW REPLICA STATUS`. We assume that you have obtained this list and are substituting it for *`ignore_server_ids`* in the query shown here, which like the previous version of the query, selects the greatest epoch into a variable named `@latest`:

```
mysqlR'> SELECT @latest:=MAX(epoch)
      ->        FROM mysql.ndb_apply_status
      ->        WHERE server_id NOT IN (ignore_server_ids);
```

In some cases, it may be simpler or more efficient (or both) to use a list of the server IDs to be included and `server_id IN` *`server_id_list`* in the `WHERE` condition of the preceding query.

2. Using the information obtained from the query shown in Step 1, obtain the corresponding records from the `ndb_binlog_index` table on the source cluster.

You can use the following query to obtain the needed records from the `ndb_binlog_index` table on the source:

```
mysqlS'> SELECT
    ->     @file:=SUBSTRING_INDEX(next_file, '/', -1),
    ->     @pos:=next_position
    -> FROM mysql.ndb_binlog_index
    -> WHERE epoch = @latest;
```

These are the records saved on the source since the failure of the primary replication channel. We have employed a user variable `@latest` here to represent the value obtained in Step 1. Of course, it is not possible for one `mysqld` instance to access user variables set on another server instance directly. These values must be “plugged in” to the second query manually or by an application.

Important

You must ensure that the replica `mysqld` is started with `--slave-skip-errors=ddl_exist_errors` before executing `START REPLICA`. Otherwise, replication may stop with duplicate DDL errors.

3. Now it is possible to synchronize the secondary channel by running the following query on the secondary replica server:

```
mysqlR'> CHANGE MASTER TO
```

```
      ->     MASTER_LOG_FILE='@file',
      ->     MASTER_LOG_POS=@pos;
```

In NDB 8.0.23 and later, you can also use the statement shown here:

```
mysqlR'> CHANGE REPLICATION SOURCE TO
      ->     SOURCE_LOG_FILE='@file',
      ->     SOURCE_LOG_POS=@pos;
```

Again we have employed user variables (in this case `@file` and `@pos`) to represent the values obtained in Step 2 and applied in Step 3; in practice these values must be inserted manually or using an application that can access both of the servers involved.

Note

`@file` is a string value such as `'/var/log/mysql/replication-source-bin.00001'`, and so must be quoted when used in SQL or application code. However, the value represented by `@pos` must *not* be quoted. Although MySQL normally attempts to convert strings to numbers, this case is an exception.

4. You can now initiate replication on the secondary channel by issuing the appropriate command on the secondary replica `mysqld`:

```
mysqlR'> START SLAVE;
```

In NDB 8.0.22 or later, you can also use the following statement:

```
mysqlR'> START REPLICA;
```

Once the secondary replication channel is active, you can investigate the failure of the primary and effect repairs. The precise actions required to do this depend upon the reasons for which the primary channel failed.

Warning

The secondary replication channel is to be started only if and when the primary replication channel has failed. Running multiple replication channels simultaneously can result in unwanted duplicate records being created on the replicas.

If the failure is limited to a single server, it should in theory be possible to replicate from *S* to *R'*, or from *S'* to *R*.

23.7.9 NDB Cluster Backups With NDB Cluster Replication

This section discusses making backups and restoring from them using NDB Cluster replication. We assume that the replication servers have already been configured as covered previously (see Section 23.7.5, “Preparing the NDB Cluster for Replication”, and the sections immediately following). This having been done, the procedure for making a backup and then restoring from it is as follows:

1. There are two different methods by which the backup may be started.

   - **Method A.** This method requires that the cluster backup process was previously enabled on the source server, prior to starting the replication process. This can be done by including the following line in a `[mysql_cluster]` section in the `my.cnf file`, where *`management_host`* is the IP address or host name of the NDB management server for the source cluster, and *`port`* is the management server's port number:

     ```
     ndb-connectstring=management_host[:port]
     ```

Note

The port number needs to be specified only if the default port (1186) is not being used. See Section 23.3.3, “Initial Configuration of NDB Cluster”, for more information about ports and port allocation in NDB Cluster.

In this case, the backup can be started by executing this statement on the replication source:

```
shellS> ndb_mgm -e "START BACKUP"
```

- **Method B.** If the `my.cnf` file does not specify where to find the management host, you can start the backup process by passing this information to the `NDB` management client as part of the `START BACKUP` command. This can be done as shown here, where *`management_host`* and *`port`* are the host name and port number of the management server:

  ```
  shellS> ndb_mgm management_host:port -e "START BACKUP"
  ```

  In our scenario as outlined earlier (see Section 23.7.5, “Preparing the NDB Cluster for Replication”), this would be executed as follows:

  ```
  shellS> ndb_mgm rep-source:1186 -e "START BACKUP"
  ```

2. Copy the cluster backup files to the replica that is being brought on line. Each system running an `ndbd` process for the source cluster has cluster backup files located on it, and *all* of these files must be copied to the replica to ensure a successful restore. The backup files can be copied into any directory on the computer where the replica's management host resides, as long as the MySQL and NDB binaries have read permissions in that directory. In this case, we assume that these files have been copied into the directory `/var/BACKUPS/BACKUP-1`.

   While it is not necessary that the replica cluster have the same number of data nodes as the source, it is highly recommended this number be the same. It *is* necessary that the replication process is prevented from starting when the replica server starts. You can do this by starting the replica with the `--skip-slave-start` option on the command line, by including `skip-slave-start` in the replica's `my.cnf` file, or in NDB 8.0.24 or later, by setting the `skip_slave_start` system variable.

3. Create any databases on the replica cluster that are present on the source cluster and that are to be replicated.

   Important

   A `CREATE DATABASE` (or `CREATE SCHEMA`) statement corresponding to each database to be replicated must be executed on each SQL node in the replica cluster.

4. Reset the replica cluster using this statement in the `mysql` client:

   ```
   mysqlR> RESET SLAVE;
   ```

   In NDB 8.0.22 or later, you can also use this statement:

   ```
   mysqlR> RESET REPLICA;
   ```

5. You can now start the cluster restoration process on the replica using the `ndb_restore` command for each backup file in turn. For the first of these, it is necessary to include the `-m` option to restore the cluster metadata, as shown here:

   ```
   shellR> ndb_restore -c replica_host:port -n node-id \
   ```

```
        -b backup-id -m -r dir
```

`dir` is the path to the directory where the backup files have been placed on the replica. For the `ndb_restore` commands corresponding to the remaining backup files, the `-m` option should *not* be used.

For restoring from a source cluster with four data nodes (as shown in the figure in Section 23.7, “NDB Cluster Replication”) where the backup files have been copied to the directory `/var/BACKUPS/BACKUP-1`, the proper sequence of commands to be executed on the replica might look like this:

```
shellR> ndb_restore -c replica-host:1186 -n 2 -b 1 -m \
        -r ./var/BACKUPS/BACKUP-1
shellR> ndb_restore -c replica-host:1186 -n 3 -b 1 \
        -r ./var/BACKUPS/BACKUP-1
shellR> ndb_restore -c replica-host:1186 -n 4 -b 1 \
        -r ./var/BACKUPS/BACKUP-1
shellR> ndb_restore -c replica-host:1186 -n 5 -b 1 -e \
        -r ./var/BACKUPS/BACKUP-1
```

Important

The `-e` (or `--restore-epoch`) option in the final invocation of `ndb_restore` in this example is required to make sure that the epoch is written to the replica's `mysql.ndb_apply_status` table. Without this information, the replica cannot synchronize properly with the source. (See Section 23.5.23, “ndb_restore — Restore an NDB Cluster Backup”.)

6. Now you need to obtain the most recent epoch from the `ndb_apply_status` table on the replica (as discussed in Section 23.7.8, “Implementing Failover with NDB Cluster Replication”):

```
mysqlR> SELECT @latest:=MAX(epoch)
        FROM mysql.ndb_apply_status;
```

7. Using `@latest` as the epoch value obtained in the previous step, you can obtain the correct starting position `@pos` in the correct binary log file `@file` from the `mysql.ndb_binlog_index` table on the source. The query shown here gets these from the `Position` and `File` columns from the last epoch applied before the logical restore position:

```
mysqlS> SELECT
     ->     @file:=SUBSTRING_INDEX(File, '/', -1),
     ->     @pos:=Position
     -> FROM mysql.ndb_binlog_index
     -> WHERE epoch > @latest
     -> ORDER BY epoch ASC LIMIT 1;
```

In the event that there is currently no replication traffic, you can get similar information by running `SHOW MASTER STATUS` on the source and using the value shown in the `Position` column of the output for the file whose name has the suffix with the greatest value for all files shown in the `File` column. In this case, you must determine which file this is and supply the name in the next step manually or by parsing the output with a script.

8. Using the values obtained in the previous step, you can now issue the appropriate in the replica's `mysql` client. In NDB 8.0.23 and later, use the following `CHANGE REPLICATION SOURCE TO` statement:

```
mysqlR> CHANGE REPLICATION SOURCE TO
     ->     SOURCE_LOG_FILE='@file',
     ->     SOURCE_LOG_POS=@pos;
```

Prior to NDB 8.0.23, you can must use the `CHANGE MASTER TO` statement shown here:

```
mysqlR> CHANGE MASTER TO
     ->     MASTER_LOG_FILE='@file',
     ->     MASTER_LOG_POS=@pos;
```

9. Now that the replica knows from what point in which binary log file to start reading data from the source, you can cause the replica to begin replicating with this statement:

```
mysqlR> START SLAVE;
```

Beginning with NDB 8.0.22, you can also use the following statement:

```
mysqlR> START REPLICA;
```

To perform a backup and restore on a second replication channel, it is necessary only to repeat these steps, substituting the host names and IDs of the secondary source and replica for those of the primary source and replica servers where appropriate, and running the preceding statements on them.

For additional information on performing Cluster backups and restoring Cluster from backups, see Section 23.6.8, “Online Backup of NDB Cluster”.

23.7.9.1 NDB Cluster Replication: Automating Synchronization of the Replica to the Source Binary Log

It is possible to automate much of the process described in the previous section (see Section 23.7.9, “NDB Cluster Backups With NDB Cluster Replication”). The following Perl script `reset-replica.pl` serves as an example of how you can do this.

```
#!/user/bin/perl -w

#  file: reset-replica.pl

#  Copyright (c) 2005, 2020, Oracle and/or its affiliates. All rights reserved.

#  This program is free software; you can redistribute it and/or modify
#  it under the terms of the GNU General Public License as published by
#  the Free Software Foundation; either version 2 of the License, or
#  (at your option) any later version.

#  This program is distributed in the hope that it will be useful,
#  but WITHOUT ANY WARRANTY; without even the implied warranty of
#  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
#  GNU General Public License for more details.

#  You should have received a copy of the GNU General Public License
#  along with this program; if not, write to:
#  Free Software Foundation, Inc.
#  59 Temple Place, Suite 330
#  Boston, MA 02111-1307 USA
#
#  Version 1.1


######################## Includes #############################

use DBI;

######################## Globals #############################

my  $m_host='';
my  $m_port='';
my  $m_user='';
my  $m_pass='';
my  $s_host='';
my  $s_port='';
my  $s_user='';
my  $s_pass='';
my  $dbhM='';
my  $dbhS='';

######################## Sub Prototypes ######################

sub CollectCommandPromptInfo;
sub ConnectToDatabases;
```

```
sub DisconnectFromDatabases;
sub GetReplicaEpoch;
sub GetSourceInfo;
sub UpdateReplica;

######################## Program Main ############################

CollectCommandPromptInfo;
ConnectToDatabases;
GetReplicaEpoch;
GetSourceInfo;
UpdateReplica;
DisconnectFromDatabases;

################## Collect Command Prompt Info ##################

sub CollectCommandPromptInfo
{
  ### Check that user has supplied correct number of command line args
  die "Usage:\n
       reset-replica >source MySQL host< >source MySQL port< \n
                     >source user< >source pass< >replica MySQL host< \n
                     >replica MySQL port< >replica user< >replica pass< \n
       All 8 arguments must be passed. Use BLANK for NULL passwords\n"
       unless @ARGV == 8;

  $m_host  =  $ARGV[0];
  $m_port  =  $ARGV[1];
  $m_user  =  $ARGV[2];
  $m_pass  =  $ARGV[3];
  $s_host  =  $ARGV[4];
  $s_port  =  $ARGV[5];
  $s_user  =  $ARGV[6];
  $s_pass  =  $ARGV[7];

  if ($m_pass eq "BLANK") { $m_pass = '';}
  if ($s_pass eq "BLANK") { $s_pass = '';}
}

###############  Make connections to both databases #############

sub ConnectToDatabases
{
  ### Connect to both source and replica cluster databases

  ### Connect to source
  $dbhM
    = DBI->connect(
    "dbi:mysql:database=mysql;host=$m_host;port=$m_port",
    "$m_user", "$m_pass")
      or die "Can't connect to source cluster MySQL process!
              Error: $DBI::errstr\n";

  ### Connect to replica
  $dbhS
    = DBI->connect(
          "dbi:mysql:database=mysql;host=$s_host",
          "$s_user", "$s_pass")
    or die "Can't connect to replica cluster MySQL process!
            Error: $DBI::errstr\n";
}

################  Disconnect from both databases ################

sub DisconnectFromDatabases
{
  ### Disconnect from source

  $dbhM->disconnect
  or warn " Disconnection failed: $DBI::errstr\n";

  ### Disconnect from replica
```

```
  $dbhS->disconnect
  or warn " Disconnection failed: $DBI::errstr\n";
}

######################  Find the last good GCI ##################

sub GetReplicaEpoch
{
  $sth = $dbhS->prepare("SELECT MAX(epoch)
                         FROM mysql.ndb_apply_status;")
      or die "Error while preparing to select epoch from replica: ",
             $dbhS->errstr;

  $sth->execute
      or die "Selecting epoch from replica error: ", $sth->errstr;

  $sth->bind_col (1, \$epoch);
  $sth->fetch;
  print "\tReplica epoch =  $epoch\n";
  $sth->finish;
}

#######  Find the position of the last GCI in the binary log ########

sub GetSourceInfo
{
  $sth = $dbhM->prepare("SELECT
                           SUBSTRING_INDEX(File, '/', -1), Position
                         FROM mysql.ndb_binlog_index
                         WHERE epoch > $epoch
                         ORDER BY epoch ASC LIMIT 1;")
      or die "Prepare to select from source error: ", $dbhM->errstr;

  $sth->execute
      or die "Selecting from source error: ", $sth->errstr;

  $sth->bind_col (1, \$binlog);
  $sth->bind_col (2, \$binpos);
  $sth->fetch;
  print "\tSource binary log file =  $binlog\n";
  print "\tSource binary log position =  $binpos\n";
  $sth->finish;
}

##########  Set the replica to process from that location #########

sub UpdateReplica
{
  $sth = $dbhS->prepare("CHANGE MASTER TO
                         MASTER_LOG_FILE='$binlog',
                         MASTER_LOG_POS=$binpos;")
      or die "Prepare to CHANGE MASTER error: ", $dbhS->errstr;

  $sth->execute
       or die "CHANGE MASTER on replica error: ", $sth->errstr;
  $sth->finish;
  print "\tReplica has been updated. You may now start the replica.\n";
}

# end reset-replica.pl
```

23.7.9.2 Point-In-Time Recovery Using NDB Cluster Replication

Point-in-time recovery—that is, recovery of data changes made since a given point in time—is performed after restoring a full backup that returns the server to its state when the backup was made. Performing point-in-time recovery of NDB Cluster tables with NDB Cluster and NDB Cluster Replication can be accomplished using a native `NDB` data backup (taken by issuing `CREATE BACKUP` in the `ndb_mgm` client) and restoring the `ndb_binlog_index` table (from a dump made using `mysqldump`).

To perform point-in-time recovery of NDB Cluster, it is necessary to follow the steps shown here:

1. Back up all `NDB` databases in the cluster, using the `START BACKUP` command in the `ndb_mgm` client (see Section 23.6.8, “Online Backup of NDB Cluster”).

2. At some later point, prior to restoring the cluster, make a backup of the `mysql.ndb_binlog_index` table. It is probably simplest to use `mysqldump` for this task. Also back up the binary log files at this time.

   This backup should be updated regularly—perhaps even hourly—depending on your needs.

3. (*Catastrophic failure or error occurs*.)

4. Locate the last known good backup.

5. Clear the data node file systems (using `ndbd --initial` or `ndbmtd --initial`).

   Note

   Beginning with NDB 8.0.21, Disk Data tablespace and log files are removed by `--initial`. Previously, it was necessary to delete these manually.

6. Use `DROP TABLE` or `TRUNCATE TABLE` with the `mysql.ndb_binlog_index` table.

7. Execute `ndb_restore`, restoring all data. You must include the `--restore-epoch` option when you run `ndb_restore`, so that the `ndb_apply_status` table is populated correctly. (See Section 23.5.23, “ndb_restore — Restore an NDB Cluster Backup”, for more information.)

8. Restore the `ndb_binlog_index` table from the output of `mysqldump` and restore the binary log files from backup, if necessary.

9. Find the epoch applied most recently—that is, the maximum `epoch` column value in the `ndb_apply_status` table—as the user variable `@LATEST_EPOCH` (emphasized):

   ```
   SELECT @LATEST_EPOCH:=MAX(epoch)
       FROM mysql.ndb_apply_status;
   ```

10. Find the latest binary log file (`@FIRST_FILE`) and position (`Position` column value) within this file that correspond to `@LATEST_EPOCH` in the `ndb_binlog_index` table:

    ```
    SELECT Position, @FIRST_FILE:=File
        FROM mysql.ndb_binlog_index
        WHERE epoch > @LATEST_EPOCH ORDER BY epoch ASC LIMIT 1;
    ```

11. Using `mysqlbinlog`, replay the binary log events from the given file and position up to the point of the failure. (See Section 4.6.9, “mysqlbinlog — Utility for Processing Binary Log Files”.)

See also Section 7.5, “Point-in-Time (Incremental) Recovery”, for more information about the binary log, replication, and incremental recovery.

23.7.10 NDB Cluster Replication: Bidirectional and Circular Replication

It is possible to use NDB Cluster for bidirectional replication between two clusters, as well as for circular replication between any number of clusters.

Circular replication example. In the next few paragraphs we consider the example of a replication setup involving three NDB Clusters numbered 1, 2, and 3, in which Cluster 1 acts as the replication source for Cluster 2, Cluster 2 acts as the source for Cluster 3, and Cluster 3 acts as the source for Cluster 1. Each cluster has two SQL nodes, with SQL nodes A and B belonging to Cluster 1, SQL nodes C and D belonging to Cluster 2, and SQL nodes E and F belonging to Cluster 3.

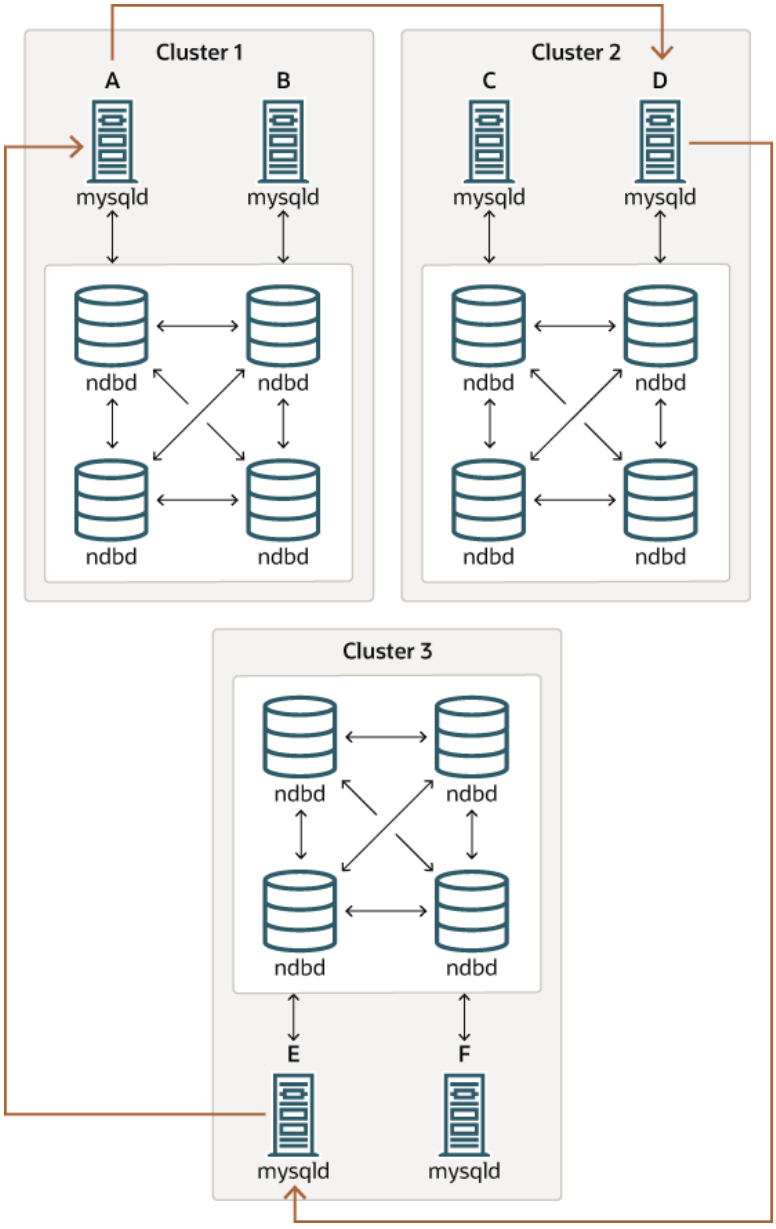
Circular replication using these clusters is supported as long as the following conditions are met:

- The SQL nodes on all sources and replicas are the same.

- All SQL nodes acting as sources and replicas are started with the system variable `log_replica_updates` (beginning with NDB 8.0.26) or `log_slave_updates` (NDB 8.0.26 and earlier) enabled.

This type of circular replication setup is shown in the following diagram:

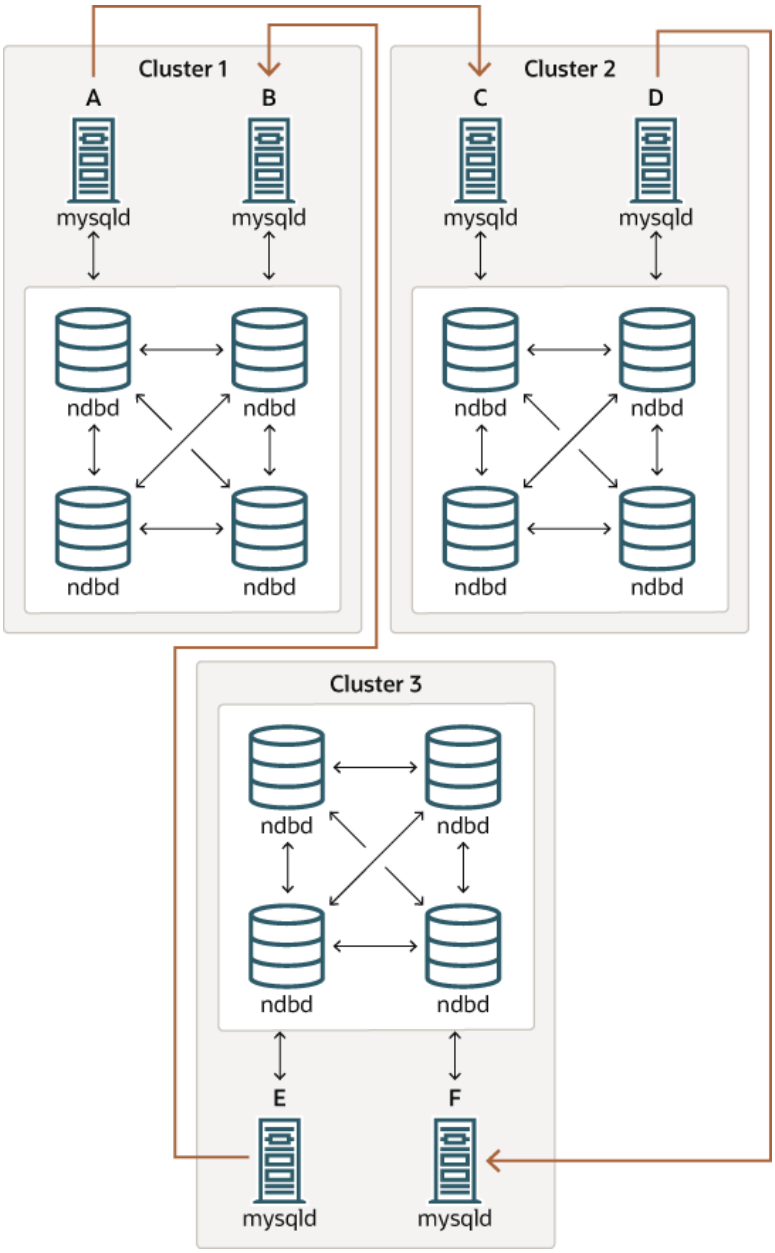
Figure 23.15 NDB Cluster Circular Replication with All Sources As Replicas



In this scenario, SQL node A in Cluster 1 replicates to SQL node C in Cluster 2; SQL node C replicates to SQL node E in Cluster 3; SQL node E replicates to SQL node A. In other words, the replication line (indicated by the curved arrows in the diagram) directly connects all SQL nodes used as replication sources and replicas.

It is also possible to set up circular replication in such a way that not all source SQL nodes are also replicas, as shown here:

Figure 23.16 NDB Cluster Circular Replication Where Not All Sources Are Replicas

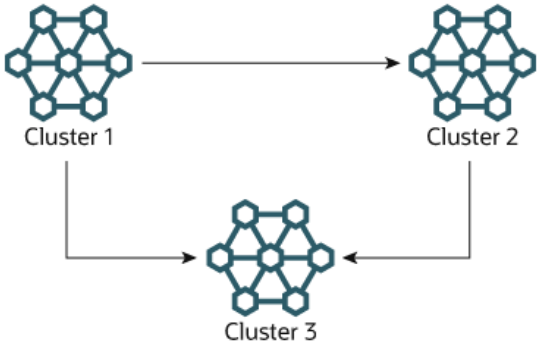


In this case, different SQL nodes in each cluster are used as replication sources and replicas. You must *not* start any of the SQL nodes with the system variable `log_replica_updates` (NDB 8.0.26 and later) or `log_slave_updates` (prior to NDB 8.0.26) enabled. This type of circular replication scheme for NDB Cluster, in which the line of replication (again indicated by the curved arrows in the diagram) is discontinuous, should be possible, but it should be noted that it has not yet been thoroughly tested and must therefore still be considered experimental.

Using NDB-native backup and restore to initialize a replica cluster. When setting up circular replication, it is possible to initialize the replica cluster by using the management client `START BACKUP` command on one NDB Cluster to create a backup and then applying this backup on another NDB Cluster using `ndb_restore`. This does not automatically create binary logs on the second NDB Cluster's SQL node acting as the replica; in order to cause the binary logs to be created, you must issue a `SHOW TABLES` statement on that SQL node; this should be done prior to running `START REPLICA`. This is a known issue.

Multi-source failover example. In this section, we discuss failover in a multi-source NDB Cluster replication setup with three NDB Clusters having server IDs 1, 2, and 3. In this scenario, Cluster 1 replicates to Clusters 2 and 3; Cluster 2 also replicates to Cluster 3. This relationship is shown here:

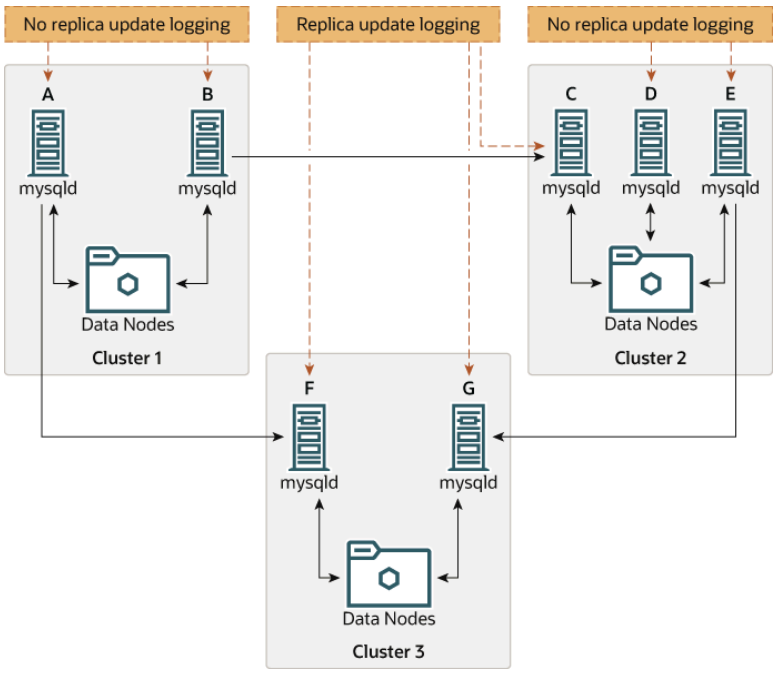
Figure 23.17 NDB Cluster Multi-Master Replication With 3 Sources



In other words, data replicates from Cluster 1 to Cluster 3 through 2 different routes: directly, and by way of Cluster 2.

Not all MySQL servers taking part in multi-source replication must act as both source and replica, and a given NDB Cluster might use different SQL nodes for different replication channels. Such a case is shown here:

Figure 23.18 NDB Cluster Multi-Source Replication, With MySQL Servers



MySQL servers acting as replicas must be run with the system variable `log_replica_updates` (beginning with NDB 8.0.26) or `log_slave_updates` (NDB 8.0.26 and earlier) enabled. Which `mysqld` processes require this option is also shown in the preceding diagram.

Note

Using the `log_replica_updates` or `log_slave_updates` system variable has no effect on servers not being run as replicas.

The need for failover arises when one of the replicating clusters goes down. In this example, we consider the case where Cluster 1 is lost to service, and so Cluster 3 loses 2 sources of updates from

Cluster 1. Because replication between NDB Clusters is asynchronous, there is no guarantee that Cluster 3's updates originating directly from Cluster 1 are more recent than those received through Cluster 2. You can handle this by ensuring that Cluster 3 catches up to Cluster 2 with regard to updates from Cluster 1. In terms of MySQL servers, this means that you need to replicate any outstanding updates from MySQL server C to server F.

On server C, perform the following queries:

```
mysqlC> SELECT @latest:=MAX(epoch)
     ->     FROM mysql.ndb_apply_status
     ->     WHERE server_id=1;

mysqlC> SELECT
     ->     @file:=SUBSTRING_INDEX(File, '/', -1),
     ->     @pos:=Position
     ->     FROM mysql.ndb_binlog_index
     ->     WHERE orig_epoch >= @latest
     ->     AND orig_server_id = 1
     ->     ORDER BY epoch ASC LIMIT 1;
```

Note

You can improve the performance of this query, and thus likely speed up failover times significantly, by adding the appropriate index to the `ndb_binlog_index` table. See Section 23.7.4, “NDB Cluster Replication Schema and Tables”, for more information.

Copy over the values for `@file` and `@pos` manually from server C to server F (or have your application perform the equivalent). Then, on server F, execute the following `CHANGE REPLICATION SOURCE TO` statement (NDB 8.0.23 and later) or `CHANGE MASTER TO` statement (prior to NDB 8.0.23):

```
mysqlF> CHANGE MASTER TO
     ->     MASTER_HOST = 'serverC'
     ->     MASTER_LOG_FILE='@file',
     ->     MASTER_LOG_POS=@pos;
```

Beginning with NDB 8.0.23, you can also use the following statement:

```
mysqlF> CHANGE REPLICATION SOURCE TO
     ->     SOURCE_HOST = 'serverC'
     ->     SOURCE_LOG_FILE='@file',
     ->     SOURCE_LOG_POS=@pos;
```

Once this has been done, you can issue a `START REPLICA` statement on MySQL server F; this causes any missing updates originating from server B to be replicated to server F.

The `CHANGE REPLICATION SOURCE TO` | `CHANGE MASTER TO` statement also supports an `IGNORE_SERVER_IDS` option which takes a comma-separated list of server IDs and causes events originating from the corresponding servers to be ignored. For more information, see Section 13.4.2.1, “CHANGE MASTER TO Statement”, and Section 13.7.7.36, “SHOW SLAVE | REPLICA STATUS Statement”. For information about how this option interacts with the `ndb_log_apply_status` variable, see Section 23.7.8, “Implementing Failover with NDB Cluster Replication”.

23.7.11 NDB Cluster Replication Using the Multithreaded Applier

- Requirements
- MTA Configuration: Source
- MTA Configuration: Replica
- Transaction Dependency and Writeset Handling
- Writeset Tracking Memory Usage

- Known Limitations

Beginning with NDB 8.0.33, NDB replication supports the use of the generic MySQL Server Multithreaded Applier mechanism (MTA), which allows independent binary log transactions to be applied in parallel on a replica, increasing peak replication throughput.

Requirements

The MySQL Server MTA implementation delegates the processing of separate binary log transactions to a pool of worker threads (whose size is configurable), and coordinates the worker threads to ensure that transaction dependencies encoded in the binary log are respected, and that commit ordering is maintained if required (see Section 17.2.3, “Replication Threads”). To use this functionality with NDB Cluster, it is necessary that the following three conditions be met:

1. *Binary log transaction dependencies are determined at the source*.

   For this to be true, the `binlog_transaction_dependency_tracking` server system variable must be set to `WRITESET` on the source. This is supported by NDB 8.0.33 and later. (The default is `COMMIT_ORDER`.)

   Writeset maintenance work in `NDB` is performed by the MySQL binary log injector thread as part of preparing and committing each epoch transaction to the binary log. This requires extra resources, and may reduce peak throughput.

2. *Transaction dependencies are encoded into the binary log*.

   NDB 8.0.33 and later supports the `--ndb-log-transaction-dependency` startup option for `mysqld`; set this option to `ON` to enable writing of `NDB` transaction dependencies into the binary log.

3. *The replica is configured to use multiple worker threads*.

   NDB 8.0.33 and later supports setting `replica_parallel_workers` to nonzero values to control the number of worker threads on the replica. The default is 4.

MTA Configuration: Source

Source `mysqld` configuration for the `NDB` MTA must include the following explicit settings:

- `binlog_transaction_dependency_tracking` must be set to `WRITESET`.
- The replication source mysqld must be started with `--ndb-log-transaction-dependency=ON`.

If set, `replica_parallel_type` must be `LOGICAL_CLOCK` (the default value; `DATABASE` is not supported).

In addition, it is recommended that you set the amount of memory used to track binary log transaction writesets on the source (`binlog_transaction_dependency_history_size`) to *`E * P`*, where *`E`* is the average epoch size (as the number of operations per epoch) and *`P`* is the maximum expected parallelism. See Writeset Tracking Memory Usage, for more information.

MTA Configuration: Replica

Replica `mysqld` configuration for the `NDB` MTA requires that `replica_parallel_workers` is greater than 1. The recommended starting value when first enabling MTA is 4, which is the default.

In addition, `replica_preserve_commit_order` must be `ON`. This is also the default value.

Transaction Dependency and Writeset Handling

Transaction dependencies are detected using analysis of each transaction's writeset, that is, the set of rows (table, key values) written by the transaction. Where two transactions modify the same

row they are considered to be dependent, and must be applied in order (in other words, serially) to avoid deadlocks or incorrect results. Where a table has secondary unique keys, these values are also added to the transaction's writeset to detect the case where there are transaction dependencies implied by different transactions affecting the same unique key value, and so requiring ordering. Where dependencies cannot be efficiently determined, `mysqld` falls back to considering transactions dependent for reasons of safety.

Transaction dependencies are encoded in the binary log by the source `mysqld`. Dependencies are encoded in an `ANONYMOUS_GTID` event using a scheme called 'Logical clock'. (See Section 17.1.4.1, “Replication Mode Concepts”.)

The writeset implementation employed by MySQL (and NDB Cluster) uses hash-based conflict detection based on matching 64-bit row hashes of relevant table and index values. This detects reliably when the same key is seen twice, but can also produce false positives if different table and index values hash to the same 64-bit value; this may result in artificial dependencies which can reduce the available parallelism.

Transaction dependencies are forced by any of the following:

- DDL statements
- Binary log rotation or encountering binary log file boundaries
- Writeset history size limitations
- Writes which reference parent foreign keys in the target table

  More specifically, transactions which perform inserts, updates, and deletes on foreign key *parent* tables are serialized relative to all preceding and following transactions, and not just to those transactions affecting tables involved in a constraint relationship. Conversely, transactions performing inserts, updates and deletes on foreign key *child* tables (referencing) are not especially serialized with regard to one another.

The MySQL MTA implementation attempts to apply independent binary log transactions in parallel. `NDB` records all changes occurring in all user transactions committing in an epoch (`TimeBetweenEpochs`, default 100 milliseconds), in one binary log transaction, referred to as an epoch transaction. Therefore, for two consecutive epoch transactions to be independent, and possible to apply in parallel, it is required that no row is modified in both epochs. If any single row is modified in both epochs, then they are dependent, and are applied serially, which can limit the expolitable parallelism available.

Epoch transactions are considered independent based on the set of rows modified on the source cluster in the epoch, but not including the generated `mysql.ndb_apply_status` `WRITE_ROW` events that convey epoch metadata. This avoids every epoch transaction being trivially dependent on the preceding epoch, but does require that the binlog is applied at the replica with the commit order preserved. This also implies that an NDB binary log with writeset dependencies is not suitable for use by a replica database using a different MySQL storage engine.

It may be possible or desirable to modify application transaction behavior to avoid patterns of repeated modifications to the same rows, in separate transactions over a short time period, to increase exploitable apply parallelism.

Writeset Tracking Memory Usage

The amount of memory used to track binary log transaction writesets can be set using the `binlog_transaction_dependency_history_size` server system variable, which defaults to 25000 row hashes.

If an average binary log transaction modifies *`N`* rows, then to be able to identify independent (parallelizable) transactions up to a parallelism level of *`P`*, we need `binlog_transaction_dependency_history_size` to be at least *`N * P`*. (The maximum is 1000000.)

The finite size of the history results in a finite maximum dependency length that can be reliably determined, giving a finite parallelism that can be expressed. Any row not found in the history may be dependent on the last transaction purged from the history.

Writeset history does not act like a sliding window over the last *N* transactions; rather, it is a finite buffer which is allowed to fill up completely, then its contents entirely discarded when it becomes full. This means that the history size follows a sawtooth pattern over time, and therefore the maximum detectable dependency length also follows a sawtooth pattern over time, such that independent transactions may still be marked as dependent if the writeset history buffer has been reset between their being processed.

In this scheme, each transaction in a binary log file is annotated with a `sequence_number` (1, 2, 3, ...), and as well as the sequence number of the most recent binary log transaction that it depends on, to which we refer as `last_committed`.

Within a given binary log file, the first transaction has `sequence_number` 1 and `last_committed` 0.

Where a binary log transaction depends on its immediate predecessor, its application is serialized. If the dependency is on an earlier transaction then it may be possible to apply the transaction in parallel with the preceding independent transactions.

The content of `ANONYMOUS_GTID` events, including `sequence_number` and `last_committed` (and thus the transaction dependencies), can be seen using `mysqlbinlog`.

The `ANONYMOUS_GTID` events generated on the source are handled separately from the compressed transaction payload with bulk `BEGIN`, `TABLE_MAP*`, `WRITE_ROW*`, `UPDATE_ROW*`, `DELETE_ROW*`, and `COMMIT` events, allowing dependencies to be determined prior to decompression. This means that the replica coordinator thread can delegate transaction payload decompression to a worker thread, providing automatic parallel decompression of independent transactions on the replica.

Known Limitations

Secondary unique columns. Tables with secondary unique columns (that is, unique keys other than the primary key) have all columns sent to the source so that unique-key related conflicts can be detected.

Where the current binary logging mode does not include all columns, but only changed columns (`--ndb-log-updated-only=OFF`, `--ndb-log-update-minimal=ON`, `--ndb-log-update-as-write=OFF`), this can increase the volume of data sent from data nodes to SQL nodes.

The impact depends on both the rate of modification (update or delete) of rows in such tables and the volume of data in columns which are not actually modified.

Replicating NDB to to InnoDB. `NDB` binary log injector transaction dependency tracking intentionally ignores the inter-transaction dependencies created by generated `mysql.ndb_apply_status` metadata events, which are handled separately as part of the commit of the epoch transaction on the replica applier. For replication to `InnoDB`, there is no special handling; this may result in reduced performance or other issues when using an `InnoDB` multithreaded applier to consume an `NDB` MTA binary log.

23.7.12 NDB Cluster Replication Conflict Resolution

- Requirements
- Source Column Control
- Conflict Resolution Control
- Conflict Resolution Functions
- Conflict Resolution Exceptions Table

- Conflict Detection Status Variables
- Examples

When using a replication setup involving multiple sources (including circular replication), it is possible that different sources may try to update the same row on the replica with different data. Conflict resolution in NDB Cluster Replication provides a means of resolving such conflicts by permitting a user-defined resolution column to be used to determine whether or not an update on a given source should be applied on the replica.

Some types of conflict resolution supported by NDB Cluster (`NDB$OLD()`, `NDB$MAX()`, and `NDB$MAX_DELETE_WIN()`; additionally, in NDB 8.0.30 and later, `NDB$MAX_INS()` and `NDB$MAX_DEL_WIN_INS()`) implement this user-defined column as a “timestamp” column (although its type cannot be `TIMESTAMP`, as explained later in this section). These types of conflict resolution are always applied a row-by-row basis rather than a transactional basis. The epoch-based conflict resolution functions `NDB$EPOCH()` and `NDB$EPOCH_TRANS()` compare the order in which epochs are replicated (and thus these functions are transactional). Different methods can be used to compare resolution column values on the replica when conflicts occur, as explained later in this section; the method used can be set to act on a single table, database, or server, or on a set of one or more tables using pattern matching. See Matching with wildcards, for information about using pattern matches in the `db`, `table_name`, and `server_id` columns of the `mysql.ndb_replication` table.

You should also keep in mind that it is the application's responsibility to ensure that the resolution column is correctly populated with relevant values, so that the resolution function can make the appropriate choice when determining whether to apply an update.

Requirements

Preparations for conflict resolution must be made on both the source and the replica. These tasks are described in the following list:

- On the source writing the binary logs, you must determine which columns are sent (all columns or only those that have been updated). This is done for the MySQL Server as a whole by applying the `mysqld` startup option `--ndb-log-updated-only` (described later in this section), or on one or more specific tables by placing the proper entries in the `mysql.ndb_replication` table (see ndb_replication Table).

  Note

  If you are replicating tables with very large columns (such as `TEXT` or `BLOB` columns), `--ndb-log-updated-only` can also be useful for reducing the size of the binary logs and avoiding possible replication failures due to exceeding `max_allowed_packet`.

  See Section 17.5.1.20, “Replication and max_allowed_packet”, for more information about this issue.

- On the replica, you must determine which type of conflict resolution to apply (“latest timestamp wins”, “same timestamp wins”, “primary wins”, “primary wins, complete transaction”, or none). This is done using the `mysql.ndb_replication` system table, and applies to one or more specific tables (see ndb_replication Table).
- NDB Cluster also supports read conflict detection, that is, detecting conflicts between reads of a given row in one cluster and updates or deletes of the same row in another cluster. This requires exclusive read locks obtained by setting `ndb_log_exclusive_reads` equal to 1 on the replica. All rows read by a conflicting read are logged in the exceptions table. For more information, see Read conflict detection and resolution.
- Prior to NDB 8.0.30, `NDB` applied `WRITE_ROW` events strictly as inserts, requiring that there was not already any such row; that is, an incoming write was always rejected if the row already existed.

(This is still the case when using any conflict resolution function other than `NDB$MAX_INS()` or `NDB$MAX_DEL_WIN_INS()`.)

Beginning with NDB 8.0.30, when using `NDB$MAX_INS()` or `NDB$MAX_DEL_WIN_INS()`, `NDB` can apply `WRITE_ROW` events idempotently, mapping such an event to an insert when the incoming row does not already exist, or to an update if it does.

When using the functions `NDB$OLD()`, `NDB$MAX()`, and `NDB$MAX_DELETE_WIN()` for timestamp-based conflict resolution (as well as `NDB$MAX_INS()` and `NDB$MAX_DEL_WIN_INS()`, beginning with NDB 8.0.30), we often refer to the column used for determining updates as a “timestamp” column. However, the data type of this column is never `TIMESTAMP`; instead, its data type should be `INT` (`INTEGER`) or `BIGINT`. The “timestamp” column should also be `UNSIGNED` and `NOT NULL`.

The `NDB$EPOCH()` and `NDB$EPOCH_TRANS()` functions discussed later in this section work by comparing the relative order of replication epochs applied on a primary and secondary NDB Cluster, and do not make use of timestamps.

Source Column Control

We can see update operations in terms of “before” and “after” images—that is, the states of the table before and after the update is applied. Normally, when updating a table with a primary key, the “before” image is not of great interest; however, when we need to determine on a per-update basis whether or not to use the updated values on a replica, we need to make sure that both images are written to the source's binary log. This is done with the `--ndb-log-update-as-write` option for `mysqld`, as described later in this section.

Important

Whether logging of complete rows or of updated columns only is done is decided when the MySQL server is started, and cannot be changed online; you must either restart `mysqld`, or start a new `mysqld` instance with different logging options.

Conflict Resolution Control

Conflict resolution is usually enabled on the server where conflicts can occur. Like logging method selection, it is enabled by entries in the `mysql.ndb_replication` table.

`NBT_UPDATED_ONLY_MINIMAL` and `NBT_UPDATED_FULL_MINIMAL` can be used with `NDB$EPOCH()`, `NDB$EPOCH2()`, and `NDB$EPOCH_TRANS()`, because these do not require “before” values of columns which are not primary keys. Conflict resolution algorithms requiring the old values, such as `NDB$MAX()` and `NDB$OLD()`, do not work correctly with these `binlog_type` values.

Conflict Resolution Functions

This section provides detailed information about the functions which can be used for conflict detection and resolution with NDB Replication.

- NDB$OLD()
- NDB$MAX()
- NDB$MAX_DELETE_WIN()
- NDB$MAX_INS()
- NDB$MAX_DEL_WIN_INS()
- NDB$EPOCH()
- NDB$EPOCH_TRANS()

- NDB$EPOCH2()
- NDB$EPOCH2_TRANS()

NDB$OLD()

If the value of *`column_name`* is the same on both the source and the replica, then the update is applied; otherwise, the update is not applied on the replica and an exception is written to the log. This is illustrated by the following pseudocode:

```
if (source_old_column_value == replica_current_column_value)
  apply_update();
else
  log_exception();
```

This function can be used for “same value wins” conflict resolution. This type of conflict resolution ensures that updates are not applied on the replica from the wrong source.

Important

The column value from the source's “before” image is used by this function.

NDB$MAX()

For an update or delete operation, if the “timestamp” column value for a given row coming from the source is higher than that on the replica, it is applied; otherwise it is not applied on the replica. This is illustrated by the following pseudocode:

```
if (source_new_column_value > replica_current_column_value)
  apply_update();
```

This function can be used for “greatest timestamp wins” conflict resolution. This type of conflict resolution ensures that, in the event of a conflict, the version of the row that was most recently updated is the version that persists.

This function has no effects on conflicts between write operations, other than that a write operation with the same primary key as a previous write is always rejected; it is accepted and applied only if no write operation using the same primary key already exists. Beginning with NDB 8.0.30, you can use `NDB$MAX_INS()` to handle conflict resolution between writes.

Important

The column value from the sources's “after” image is used by this function.

NDB$MAX_DELETE_WIN()

This is a variation on `NDB$MAX()`. Due to the fact that no timestamp is available for a delete operation, a delete using `NDB$MAX()` is in fact processed as `NDB$OLD`, but for some use cases, this is not optimal. For `NDB$MAX_DELETE_WIN()`, if the “timestamp” column value for a given row adding or updating an existing row coming from the source is higher than that on the replica, it is applied. However, delete operations are treated as always having the higher value. This is illustrated by the following pseudocode:

```
if ( (source_new_column_value > replica_current_column_value)
       ||
     operation.type == "delete")
  apply_update();
```

This function can be used for “greatest timestamp, delete wins” conflict resolution. This type of conflict resolution ensures that, in the event of a conflict, the version of the row that was deleted or (otherwise) most recently updated is the version that persists.

Note

As with `NDB$MAX()`, the column value from the source's “after” image is the value used by this function.

NDB$MAX_INS()

This function provides support for resolution of conflicting write operations. Such conflicts are handled by “NDB$MAX_INS()” as follows:

1. If there is no conflicting write, apply this one (this is the same as `NDB$MAX()`).
2. Otherwise, apply “greatest timestamp wins” conflict resolution, as follows:
   a. If the timestamp for the incoming write is greater than that of the conflicting write, apply the incoming operation.
   b. If the timestamp for the incoming write is *not* greater, reject the incoming write operation.

When handling an insert operation, `NDB$MAX_INS()` compares timestamps from the source and replica as illustrated by the following pseudocode:

```
if (source_new_column_value > replica_current_column_value)
  apply_insert();
else
  log_exception();
```

For an update operation, the updated timestamp column value from the source is compared with the replica's timestamp column value, as shown here:

```
if (source_new_column_value > replica_current_column_value)
  apply_update();
else
  log_exception();
```

This is the same as performed by `NDB$MAX()`.

For delete operations, the handling is also the same as that performed by `NDB$MAX()` (and thus the same as `NDB$OLD()`), and is done like this:

```
if (source_new_column_value == replica_current_column_value)
  apply_delete();
else
  log_exception();
```

`NDB$MAX_INS()` was added in NDB 8.0.30.

NDB$MAX_DEL_WIN_INS()

This function provides support for resolution of conflicting write operations, along with “delete wins” resolution like that of `NDB$MAX_DELETE_WIN()`. Write conflicts are handled by `NDB$MAX_DEL_WIN_INS()` as shown here:

1. If there is no conflicting write, apply this one (this is the same as `NDB$MAX_DELETE_WIN()`).
2. Otherwise, apply “greatest timestamp wins” conflict resolution, as follows:
   a. If the timestamp for the incoming write is greater than that of the conflicting write, apply the incoming operation.
   b. If the timestamp for the incoming write is *not* greater, reject the incoming write operation.

Handling of insert operations as performed by `NDB$MAX_DEL_WIN_INS()` can be represented in pseudocode as shown here:

```
if (source_new_column_value > replica_current_column_value)
```

```
  apply_insert();
else
  log_exception();
```

For update operations, the source's updated timestamp column value is compared with replica's timestamp column value, like this (again using pseudocode):

```
if (source_new_column_value > replica_current_column_value)
  apply_update();
else
  log_exception();
```

Deletes are handled using a “delete always wins” strategy (the same as `NDB$MAX_DELETE_WIN()`); a `DELETE` is always applied without any regard to any timestamp values, as illustrated by this pseudocode:

```
if (operation.type == "delete")
  apply_delete();
```

For conflicts between update and delete operations, this function behaves identically to `NDB$MAX_DELETE_WIN()`.

`NDB$MAX_DEL_WIN_INS()` was added in NDB 8.0.30.

NDB$EPOCH()

The `NDB$EPOCH()` function tracks the order in which replicated epochs are applied on a replica cluster relative to changes originating on the replica. This relative ordering is used to determine whether changes originating on the replica are concurrent with any changes that originate locally, and are therefore potentially in conflict.

Most of what follows in the description of `NDB$EPOCH()` also applies to `NDB$EPOCH_TRANS()`. Any exceptions are noted in the text.

`NDB$EPOCH()` is asymmetric, operating on one NDB Cluster in a bidirectional replication configuration (sometimes referred to as “active-active” replication). We refer here to cluster on which it operates as the primary, and the other as the secondary. The replica on the primary is responsible for detecting and handling conflicts, while the replica on the secondary is not involved in any conflict detection or handling.

When the replica on the primary detects conflicts, it injects events into its own binary log to compensate for these; this ensures that the secondary NDB Cluster eventually realigns itself with the primary and so keeps the primary and secondary from diverging. This compensation and realignment mechanism requires that the primary NDB Cluster always wins any conflicts with the secondary—that is, that the primary's changes are always used rather than those from the secondary in event of a conflict. This “primary always wins” rule has the following implications:

- Operations that change data, once committed on the primary, are fully persistent and are not undone or rolled back by conflict detection and resolution.
- Data read from the primary is fully consistent. Any changes committed on the Primary (locally or from the replica) are not reverted later.
- Operations that change data on the secondary may later be reverted if the primary determines that they are in conflict.
- Individual rows read on the secondary are self-consistent at all times, each row always reflecting either a state committed by the secondary, or one committed by the primary.
- Sets of rows read on the secondary may not necessarily be consistent at a given single point in time. For `NDB$EPOCH_TRANS()`, this is a transient state; for `NDB$EPOCH()`, it can be a persistent state.
- Assuming a period of sufficient length without any conflicts, all data on the secondary NDB Cluster (eventually) becomes consistent with the primary's data.

`NDB$EPOCH()` and `NDB$EPOCH_TRANS()` do not require any user schema modifications, or application changes to provide conflict detection. However, careful thought must be given to the schema used, and the access patterns used, to verify that the complete system behaves within specified limits.

Each of the `NDB$EPOCH()` and `NDB$EPOCH_TRANS()` functions can take an optional parameter; this is the number of bits to use to represent the lower 32 bits of the epoch, and should be set to no less than the value calculated as shown here:

```
CEIL( LOG2( TimeBetweenGlobalCheckpoints / TimeBetweenEpochs ), 1)
```

For the default values of these configuration parameters (2000 and 100 milliseconds, respectively), this gives a value of 5 bits, so the default value (6) should be sufficient, unless other values are used for `TimeBetweenGlobalCheckpoints`, `TimeBetweenEpochs`, or both. A value that is too small can result in false positives, while one that is too large could lead to excessive wasted space in the database.

Both `NDB$EPOCH()` and `NDB$EPOCH_TRANS()` insert entries for conflicting rows into the relevant exceptions tables, provided that these tables have been defined according to the same exceptions table schema rules as described elsewhere in this section (see NDB$OLD()). You must create any exceptions table before creating the data table with which it is to be used.

As with the other conflict detection functions discussed in this section, `NDB$EPOCH()` and `NDB$EPOCH_TRANS()` are activated by including relevant entries in the `mysql.ndb_replication` table (see ndb_replication Table). The roles of the primary and secondary NDB Clusters in this scenario are fully determined by `mysql.ndb_replication` table entries.

Because the conflict detection algorithms employed by `NDB$EPOCH()` and `NDB$EPOCH_TRANS()` are asymmetric, you must use different values for the `server_id` entries of the primary and secondary replicas.

A conflict between `DELETE` operations alone is not sufficient to trigger a conflict using `NDB$EPOCH()` or `NDB$EPOCH_TRANS()`, and the relative placement within epochs does not matter.

Limitations on NDB$EPOCH()

The following limitations currently apply when using `NDB$EPOCH()` to perform conflict detection:

- Conflicts are detected using NDB Cluster epoch boundaries, with granularity proportional to `TimeBetweenEpochs` (default: 100 milliseconds). The minimum conflict window is the minimum time during which concurrent updates to the same data on both clusters always report a conflict. This is always a nonzero length of time, and is roughly proportional to `2 * (latency + queueing + TimeBetweenEpochs)`. This implies that—assuming the default for `TimeBetweenEpochs` and ignoring any latency between clusters (as well as any queuing delays)—the minimum conflict window size is approximately 200 milliseconds. This minimum window should be considered when looking at expected application “race” patterns.

- Additional storage is required for tables using the `NDB$EPOCH()` and `NDB$EPOCH_TRANS()` functions; from 1 to 32 bits extra space per row is required, depending on the value passed to the function.

- Conflicts between delete operations may result in divergence between the primary and secondary. When a row is deleted on both clusters concurrently, the conflict can be detected, but is not recorded, since the row is deleted. This means that further conflicts during the propagation of any subsequent realignment operations are not detected, which can lead to divergence.

  Deletes should be externally serialized, or routed to one cluster only. Alternatively, a separate row should be updated transactionally with such deletes and any inserts that follow them, so that conflicts can be tracked across row deletes. This may require changes in applications.

- Only two NDB Clusters in a bidirectional “active-active” configuration are currently supported when using `NDB$EPOCH()` or `NDB$EPOCH_TRANS()` for conflict detection.

- Tables having `BLOB` or `TEXT` columns are not currently supported with `NDB$EPOCH()` or `NDB$EPOCH_TRANS()`.

NDB$EPOCH_TRANS()

`NDB$EPOCH_TRANS()` extends the `NDB$EPOCH()` function. Conflicts are detected and handled in the same way using the “primary wins all” rule (see NDB$EPOCH()) but with the extra condition that any other rows updated in the same transaction in which the conflict occurred are also regarded as being in conflict. In other words, where `NDB$EPOCH()` realigns individual conflicting rows on the secondary, `NDB$EPOCH_TRANS()` realigns conflicting transactions.

In addition, any transactions which are detectably dependent on a conflicting transaction are also regarded as being in conflict, these dependencies being determined by the contents of the secondary cluster's binary log. Since the binary log contains only data modification operations (inserts, updates, and deletes), only overlapping data modifications are used to determine dependencies between transactions.

`NDB$EPOCH_TRANS()` is subject to the same conditions and limitations as `NDB$EPOCH()`, and in addition requires that all transaction IDs are recorded in the secondary's binary log, using `--ndb-log-transaction-id` set to `ON`. This adds a variable amount of overhead (up to 13 bytes per row).

The deprecated `log_bin_use_v1_row_events` system variable, which defaults to `OFF`, must *not* be set to `ON` with `NDB$EPOCH_TRANS()`.

See NDB$EPOCH().

NDB$EPOCH2()

The `NDB$EPOCH2()` function is similar to `NDB$EPOCH()`, except that `NDB$EPOCH2()` provides for delete-delete handling with a bidirectional replication topology. In this scenario, primary and secondary roles are assigned to the two sources by setting the `ndb_slave_conflict_role` system variable to the appropriate value on each source (usually one each of `PRIMARY`, `SECONDARY`). When this is done, modifications made by the secondary are reflected by the primary back to the secondary which then conditionally applies them.

NDB$EPOCH2_TRANS()

`NDB$EPOCH2_TRANS()` extends the `NDB$EPOCH2()` function. Conflicts are detected and handled in the same way, and assigning primary and secondary roles to the replicating clusters, but with the extra condition that any other rows updated in the same transaction in which the conflict occurred are also regarded as being in conflict. That is, `NDB$EPOCH2()` realigns individual conflicting rows on the secondary, while `NDB$EPOCH_TRANS()` realigns conflicting transactions.

Where `NDB$EPOCH()` and `NDB$EPOCH_TRANS()` use metadata that is specified per row, per last modified epoch, to determine on the primary whether an incoming replicated row change from the secondary is concurrent with a locally committed change; concurrent changes are regarded as conflicting, with subsequent exceptions table updates and realignment of the secondary. A problem arises when a row is deleted on the primary so there is no longer any last-modified epoch available to determine whether any replicated operations conflict, which means that conflicting delete operations are not detected. This can result in divergence, an example being a delete on one cluster which is concurrent with a delete and insert on the other; this why delete operations can be routed to only one cluster when using `NDB$EPOCH()` and `NDB$EPOCH_TRANS()`.

`NDB$EPOCH2()` bypasses the issue just described—storing information about deleted rows on the PRIMARY—by ignoring any delete-delete conflict, and by avoiding any potential resultant divergence as well. This is accomplished by reflecting any operation successfully applied on and replicated from the secondary back to the secondary. On its return to the secondary, it can be used to reapply an operation on the secondary which was deleted by an operation originating from the primary.

When using `NDB$EPOCH2()`, you should keep in mind that the secondary applies the delete from the primary, removing the new row until it is restored by a reflected operation. In theory, the subsequent

insert or update on the secondary conflicts with the delete from the primary, but in this case, we choose to ignore this and allow the secondary to “win”, in the interest of preventing divergence between the clusters. In other words, after a delete, the primary does not detect conflicts, and instead adopts the secondary's following changes immediately. Because of this, the secondary's state can revisit multiple previous committed states as it progresses to a final (stable) state, and some of these may be visible.

You should also be aware that reflecting all operations from the secondary back to the primary increases the size of the primary's logbinary log, as well as demands on bandwidth, CPU usage, and disk I/O.

Application of reflected operations on the secondary depends on the state of the target row on the secondary. Whether or not reflected changes are applied on the secondary can be tracked by checking the `Ndb_conflict_reflected_op_prepare_count` and `Ndb_conflict_reflected_op_discard_count` status variables. The number of changes applied is simply the difference between these two values (note that `Ndb_conflict_reflected_op_prepare_count` is always greater than or equal to `Ndb_conflict_reflected_op_discard_count`).

Events are applied if and only if both of the following conditions are true:

- The existence of the row—that is, whether or not it exists—is in accordance with the type of event. For delete and update operations, the row must already exist. For insert operations, the row must *not* exist.
- The row was last modified by the primary. It is possible that the modification was accomplished through the execution of a reflected operation.

If both of these conditions are not met, the reflected operation is discarded by the secondary.

Conflict Resolution Exceptions Table

To use the `NDB$OLD()` conflict resolution function, it is also necessary to create an exceptions table corresponding to each `NDB` table for which this type of conflict resolution is to be employed. This is also true when using `NDB$EPOCH()` or `NDB$EPOCH_TRANS()`. The name of this table is that of the table for which conflict resolution is to be applied, with the string `$EX` appended. (For example, if the name of the original table is `mytable`, the name of the corresponding exceptions table name should be `mytable$EX`.) The syntax for creating the exceptions table is as shown here:

```
CREATE TABLE original_table$EX  (
    [NDB$]server_id INT UNSIGNED,
    [NDB$]source_server_id INT UNSIGNED,
    [NDB$]source_epoch BIGINT UNSIGNED,
    [NDB$]count INT UNSIGNED,

    [NDB$OP_TYPE ENUM('WRITE_ROW','UPDATE_ROW', 'DELETE_ROW',
      'REFRESH_ROW', 'READ_ROW') NOT NULL,]
    [NDB$CFT_CAUSE ENUM('ROW_DOES_NOT_EXIST', 'ROW_ALREADY_EXISTS',
      'DATA_IN_CONFLICT', 'TRANS_IN_CONFLICT') NOT NULL,]
    [NDB$ORIG_TRANSID BIGINT UNSIGNED NOT NULL,]

    original_table_pk_columns,

    [orig_table_column|orig_table_column$OLD|orig_table_column$NEW,]

    [additional_columns,]

    PRIMARY KEY([NDB$]server_id, [NDB$]source_server_id, [NDB$]source_epoch, [NDB$]count)
) ENGINE=NDB;
```

The first four columns are required. The names of the first four columns and the columns matching the original table's primary key columns are not critical; however, we suggest for reasons of clarity and consistency, that you use the names shown here for the `server_id`, `source_server_id`, `source_epoch`, and `count` columns, and that you use the same names as in the original table for the columns matching those in the original table's primary key.

If the exceptions table uses one or more of the optional columns NDBOP_TYPE, NDBCFT_CAUSE, or NDB$ORIG_TRANSID discussed later in this section, then each of the required columns must also be named using the prefix NDB$. If desired, you can use the NDB$ prefix to name the required columns even if you do not define any optional columns, but in this case, all four of the required columns must be named using the prefix.

Following these columns, the columns making up the original table's primary key should be copied in the order in which they are used to define the primary key of the original table. The data types for the columns duplicating the primary key columns of the original table should be the same as (or larger than) those of the original columns. A subset of the primary key columns may be used.

The exceptions table must use the NDB storage engine. (An example that uses NDB$OLD() with an exceptions table is shown later in this section.)

Additional columns may optionally be defined following the copied primary key columns, but not before any of them; any such extra columns cannot be NOT NULL. NDB Cluster supports three additional, predefined optional columns NDBOP_TYPE, NDBCFT_CAUSE, and NDB$ORIG_TRANSID, which are described in the next few paragraphs.

NDB$OP_TYPE: This column can be used to obtain the type of operation causing the conflict. If you use this column, define it as shown here:

```
NDB$OP_TYPE ENUM('WRITE_ROW', 'UPDATE_ROW', 'DELETE_ROW',
    'REFRESH_ROW', 'READ_ROW') NOT NULL
```

The WRITE_ROW, UPDATE_ROW, and DELETE_ROW operation types represent user-initiated operations. REFRESH_ROW operations are operations generated by conflict resolution in compensating transactions sent back to the originating cluster from the cluster that detected the conflict. READ_ROW operations are user-initiated read tracking operations defined with exclusive row locks.

NDB$CFT_CAUSE: You can define an optional column NDB$CFT_CAUSE which provides the cause of the registered conflict. This column, if used, is defined as shown here:

```
NDB$CFT_CAUSE ENUM('ROW_DOES_NOT_EXIST', 'ROW_ALREADY_EXISTS',
    'DATA_IN_CONFLICT', 'TRANS_IN_CONFLICT') NOT NULL
```

ROW_DOES_NOT_EXIST can be reported as the cause for UPDATE_ROW and WRITE_ROW operations; ROW_ALREADY_EXISTS can be reported for WRITE_ROW events. DATA_IN_CONFLICT is reported when a row-based conflict function detects a conflict; TRANS_IN_CONFLICT is reported when a transactional conflict function rejects all of the operations belonging to a complete transaction.

NDB$ORIG_TRANSID: The NDB$ORIG_TRANSID column, if used, contains the ID of the originating transaction. This column should be defined as follows:

```
NDB$ORIG_TRANSID BIGINT UNSIGNED NOT NULL
```

NDB$ORIG_TRANSID is a 64-bit value generated by NDB. This value can be used to correlate multiple exceptions table entries belonging to the same conflicting transaction from the same or different exceptions tables.

Additional reference columns which are not part of the original table's primary key can be named *colname*$OLD or *colname*$NEW. *colname*$OLD references old values in update and delete operations—that is, operations containing DELETE_ROW events. *colname*$NEW can be used to reference new values in insert and update operations—in other words, operations using WRITE_ROW events, UPDATE_ROW events, or both types of events. Where a conflicting operation does not supply a value for a given reference column that is not a primary key, the exceptions table row contains either NULL, or a defined default value for that column.

Important

The mysql.ndb_replication table is read when a data table is set up for replication, so the row corresponding to a table to be replicated must be

inserted into `mysql.ndb_replication` *before* the table to be replicated is created.

Conflict Detection Status Variables

Several status variables can be used to monitor conflict detection. You can see how many rows have been found in conflict by `NDB$EPOCH()` since this replica was last restarted from the current value of the `Ndb_conflict_fn_epoch` system status variable.

`Ndb_conflict_fn_epoch_trans` provides the number of rows that have been found directly in conflict by `NDB$EPOCH_TRANS()`. `Ndb_conflict_fn_epoch2` and `Ndb_conflict_fn_epoch2_trans` show the number of rows found in conflict by `NDB$EPOCH2()` and `NDB$EPOCH2_TRANS()`, respectively. The number of rows actually realigned, including those affected due to their membership in or dependency on the same transactions as other conflicting rows, is given by `Ndb_conflict_trans_row_reject_count`.

Another server status variable `Ndb_conflict_fn_max` provides a count of the number of times that a row was not applied on the current SQL node due to “greatest timestamp wins” conflict resolution since the last time that `mysqld` was started. `Ndb_conflict_fn_max_del_win` provides a count of the number of times that conflict resolution based on the outcome of `NDB$MAX_DELETE_WIN()` has been applied.

NDB 8.0.30 and later provides `Ndb_conflict_fn_max_ins` for tracking the number of times that “greater timestamp wins” handling has been applied to write operations (using `NDB$MAX_INS()`); a count of the number of times that “same timestamp wins” handling of writes has been applied (as implemented by `NDB$MAX_DEL_WIN_INS()`), is provided by the status variable `Ndb_conflict_fn_max_del_win_ins`.

The number of times that a row was not applied as the result of “same timestamp wins” conflict resolution on a given `mysqld` since the last time it was restarted is given by the global status variable `Ndb_conflict_fn_old`. In addition to incrementing `Ndb_conflict_fn_old`, the primary key of the row that was not used is inserted into an *exceptions table*, as explained elsewhere in this section.

See also NDB Cluster Status Variables.

Examples

The following examples assume that you have already a working NDB Cluster replication setup, as described in Section 23.7.5, “Preparing the NDB Cluster for Replication”, and Section 23.7.6, “Starting NDB Cluster Replication (Single Replication Channel)”.

NDB$MAX() example. Suppose you wish to enable “greatest timestamp wins” conflict resolution on table `test.t1`, using column `mycol` as the “timestamp”. This can be done using the following steps:

1. Make sure that you have started the source `mysqld` with `--ndb-log-update-as-write=OFF`.
2. On the source, perform this `INSERT` statement:

```
INSERT INTO mysql.ndb_replication
    VALUES ('test', 't1', 0, NULL, 'NDB$MAX(mycol)');
```

Note

If the `ndb_replication` table does not already exist, you must create it. See ndb_replication Table.

Inserting a 0 into the `server_id` column indicates that all SQL nodes accessing this table should use conflict resolution. If you want to use conflict resolution on a specific `mysqld` only, use the actual server ID.

Inserting `NULL` into the `binlog_type` column has the same effect as inserting 0 (`NBT_DEFAULT`); the server default is used.

3. Create the `test.t1` table:

```
CREATE TABLE test.t1 (
    columns
    mycol INT UNSIGNED,
    columns
) ENGINE=NDB;
```

   Now, when updates are performed on this table, conflict resolution is applied, and the version of the row having the greatest value for `mycol` is written to the replica.

Note

Other `binlog_type` options such as `NBT_UPDATED_ONLY_USE_UPDATE` (`6`) should be used to control logging on the source using the `ndb_replication` table rather than by using command-line options.

NDB$OLD() example. Suppose an `NDB` table such as the one defined here is being replicated, and you wish to enable “same timestamp wins” conflict resolution for updates to this table:

```
CREATE TABLE test.t2  (
    a INT UNSIGNED NOT NULL,
    b CHAR(25) NOT NULL,
    columns,
    mycol INT UNSIGNED NOT NULL,
    columns,
    PRIMARY KEY pk (a, b)
)   ENGINE=NDB;
```

The following steps are required, in the order shown:

1. First—and *prior* to creating `test.t2`—you must insert a row into the `mysql.ndb_replication` table, as shown here:

```
INSERT INTO mysql.ndb_replication
    VALUES ('test', 't2', 0, 0, 'NDB$OLD(mycol)');
```

   Possible values for the `binlog_type` column are shown earlier in this section; in this case, we use `0` to specify that the server default logging behavior be used. The value `'NDB$OLD(mycol)'` should be inserted into the `conflict_fn` column.

2. Create an appropriate exceptions table for `test.t2`. The table creation statement shown here includes all required columns; any additional columns must be declared following these columns, and before the definition of the table's primary key.

```
CREATE TABLE test.t2$EX  (
    server_id INT UNSIGNED,
    source_server_id INT UNSIGNED,
    source_epoch BIGINT UNSIGNED,
    count INT UNSIGNED,
    a INT UNSIGNED NOT NULL,
    b CHAR(25) NOT NULL,

    [additional_columns,]

    PRIMARY KEY(server_id, source_server_id, source_epoch, count)
)   ENGINE=NDB;
```

   We can include additional columns for information about the type, cause, and originating transaction ID for a given conflict. We are also not required to supply matching columns for all primary key columns in the original table. This means you can create the exceptions table like this:

```
CREATE TABLE test.t2$EX  (
    NDB$server_id INT UNSIGNED,
    NDB$source_server_id INT UNSIGNED,
    NDB$source_epoch BIGINT UNSIGNED,
    NDB$count INT UNSIGNED,
```

```
    a INT UNSIGNED NOT NULL,

    NDB$OP_TYPE ENUM('WRITE_ROW','UPDATE_ROW', 'DELETE_ROW',
      'REFRESH_ROW', 'READ_ROW') NOT NULL,
    NDB$CFT_CAUSE ENUM('ROW_DOES_NOT_EXIST', 'ROW_ALREADY_EXISTS',
      'DATA_IN_CONFLICT', 'TRANS_IN_CONFLICT') NOT NULL,
    NDB$ORIG_TRANSID BIGINT UNSIGNED NOT NULL,

    [additional_columns,]

    PRIMARY KEY(NDB$server_id, NDB$source_server_id, NDB$source_epoch, NDB$count)
)   ENGINE=NDB;
```

Note

The `NDB$` prefix is required for the four required columns since we included at least one of the columns `NDB$OP_TYPE`, `NDB$CFT_CAUSE`, or `NDB$ORIG_TRANSID` in the table definition.

3. Create the table `test.t2` as shown previously.

These steps must be followed for every table for which you wish to perform conflict resolution using `NDB$OLD()`. For each such table, there must be a corresponding row in `mysql.ndb_replication`, and there must be an exceptions table in the same database as the table being replicated.

Read conflict detection and resolution. NDB Cluster also supports tracking of read operations, which makes it possible in circular replication setups to manage conflicts between reads of a given row in one cluster and updates or deletes of the same row in another. This example uses `employee` and `department` tables to model a scenario in which an employee is moved from one department to another on the source cluster (which we refer to hereafter as cluster *A*) while the replica cluster (hereafter *B*) updates the employee count of the employee's former department in an interleaved transaction.

The data tables have been created using the following SQL statements:

```
# Employee table
CREATE TABLE employee (
    id INT PRIMARY KEY,
    name VARCHAR(2000),
    dept INT NOT NULL
)   ENGINE=NDB;

# Department table
CREATE TABLE department (
    id INT PRIMARY KEY,
    name VARCHAR(2000),
    members INT
)   ENGINE=NDB;
```

The contents of the two tables include the rows shown in the (partial) output of the following `SELECT` statements:

```
mysql> SELECT id, name, dept FROM employee;
+---------------+------+
| id   | name   | dept |
+------+--------+------+
...
| 998  |  Mike  | 3    |
| 999  |  Joe   | 3    |
| 1000 |  Mary  | 3    |
...
+------+--------+------+

mysql> SELECT id, name, members FROM department;
+-----+-------------+---------+
| id  | name        | members |
+-----+-------------+---------+
...
```

```
| 3   | Old project | 24      |
...
+-----+-------------+---------+
```

We assume that we are already using an exceptions table that includes the four required columns (and these are used for this table's primary key), the optional columns for operation type and cause, and the original table's primary key column, created using the SQL statement shown here:

```
CREATE TABLE employee$EX  (
    NDB$server_id INT UNSIGNED,
    NDB$source_server_id INT UNSIGNED,
    NDB$source_epoch BIGINT UNSIGNED,
    NDB$count INT UNSIGNED,

    NDB$OP_TYPE ENUM( 'WRITE_ROW','UPDATE_ROW', 'DELETE_ROW',
                      'REFRESH_ROW','READ_ROW') NOT NULL,
    NDB$CFT_CAUSE ENUM( 'ROW_DOES_NOT_EXIST',
                        'ROW_ALREADY_EXISTS',
                        'DATA_IN_CONFLICT',
                        'TRANS_IN_CONFLICT') NOT NULL,

    id INT NOT NULL,

    PRIMARY KEY(NDB$server_id, NDB$source_server_id, NDB$source_epoch, NDB$count)
)   ENGINE=NDB;
```

Suppose there occur the two simultaneous transactions on the two clusters. On cluster *A*, we create a new department, then move employee number 999 into that department, using the following SQL statements:

```
BEGIN;
  INSERT INTO department VALUES (4, "New project", 1);
  UPDATE employee SET dept = 4 WHERE id = 999;
COMMIT;
```

At the same time, on cluster *B*, another transaction reads from `employee`, as shown here:

```
BEGIN;
  SELECT name FROM employee WHERE id = 999;
  UPDATE department SET members = members - 1  WHERE id = 3;
commit;
```

The conflicting transactions are not normally detected by the conflict resolution mechanism, since the conflict is between a read (`SELECT`) and an update operation. You can circumvent this issue by executing `SET ndb_log_exclusive_reads = 1` on the replica cluster. Acquiring exclusive read locks in this way causes any rows read on the source to be flagged as needing conflict resolution on the replica cluster. If we enable exclusive reads in this way prior to the logging of these transactions, the read on cluster *B* is tracked and sent to cluster *A* for resolution; the conflict on the employee row is subsequently detected and the transaction on cluster *B* is aborted.

The conflict is registered in the exceptions table (on cluster *A*) as a `READ_ROW` operation (see Conflict Resolution Exceptions Table, for a description of operation types), as shown here:

```
mysql> SELECT id, NDB$OP_TYPE, NDB$CFT_CAUSE FROM employee$EX;
+-------+-------------+------------------+
| id    | NDB$OP_TYPE | NDB$CFT_CAUSE    |
+-------+-------------+------------------+
...
| 999   | READ_ROW    | TRANS_IN_CONFLICT |
+-------+-------------+------------------+
```

Any existing rows found in the read operation are flagged. This means that multiple rows resulting from the same conflict may be logged in the exception table, as shown by examining the effects a conflict between an update on cluster *A* and a read of multiple rows on cluster *B* from the same table in simultaneous transactions. The transaction executed on cluster *A* is shown here:

```
BEGIN;
```

```
  INSERT INTO department VALUES (4, "New project", 0);
  UPDATE employee SET dept = 4 WHERE dept = 3;
  SELECT COUNT(*) INTO @count FROM employee WHERE dept = 4;
  UPDATE department SET members = @count WHERE id = 4;
COMMIT;
```

Concurrently a transaction containing the statements shown here runs on cluster *B*:

```
SET ndb_log_exclusive_reads = 1;  # Must be set if not already enabled
...
BEGIN;
  SELECT COUNT(*) INTO @count FROM employee WHERE dept = 3 FOR UPDATE;
  UPDATE department SET members = @count WHERE id = 3;
COMMIT;
```

In this case, all three rows matching the `WHERE` condition in the second transaction's `SELECT` are read, and are thus flagged in the exceptions table, as shown here:

```
mysql> SELECT id, NDB$OP_TYPE, NDB$CFT_CAUSE FROM employee$EX;
+-------+-------------+-------------------+
| id    | NDB$OP_TYPE | NDB$CFT_CAUSE     |
+-------+-------------+-------------------+
...
| 998   | READ_ROW    | TRANS_IN_CONFLICT |
| 999   | READ_ROW    | TRANS_IN_CONFLICT |
| 1000  | READ_ROW    | TRANS_IN_CONFLICT |
...
+-------+-------------+-------------------+
```

Read tracking is performed on the basis of existing rows only. A read based on a given condition track conflicts only of any rows that are *found* and not of any rows that are inserted in an interleaved transaction. This is similar to how exclusive row locking is performed in a single instance of NDB Cluster.

Insert conflict detection and resolution example (NDB 8.0.30 and later). The following example illustrates the use of the insert conflict detection functions added in NDB 8.0.30. We assume that we are replicating two tables `t1` and `t2` in database `test`, and that we wish to use insert conflict detection with `NDB$MAX_INS()` for `t1` and `NDB$MAX_DEL_WIN_INS()` for `t2`. The two data tables are not created until later in the setup process.

Setting up insert conflict resolution is similar to setting up other conflict detection and resolution algorithms as shown in the previous examples. If the `mysql.ndb_replication` table used to configure binary logging and conflict resolution, does not already exist, it is first necessary to create it, as shown here:

```
CREATE TABLE mysql.ndb_replication (
    db VARBINARY(63),
    table_name VARBINARY(63),
    server_id INT UNSIGNED,
    binlog_type INT UNSIGNED,
    conflict_fn VARBINARY(128),
    PRIMARY KEY USING HASH (db, table_name, server_id)
) ENGINE=NDB
PARTITION BY KEY(db,table_name);
```

The `ndb_replication` table acts on a per-table basis; that is, we need to insert a row containing table information, a `binlog_type` value, the conflict resolution function to be employed, and the name of the timestamp column (`X`) for each table to be set up, like this:

```
INSERT INTO mysql.ndb_replication VALUES ("test", "t1", 0, 7, "NDB$MAX_INS(X)");
INSERT INTO mysql.ndb_replication VALUES ("test", "t2", 0, 7, "NDB$MAX_DEL_WIN_INS(X)");
```

Here we have set the binlog_type as `NBT_FULL_USE_UPDATE` (`7`) which means that full rows are always logged. See ndb_replication Table, for other possible values.

You can also create an exceptions table corresponding to each `NDB` table for which conflict resolution is to be employed. An exceptions table records all rows rejected by the conflict resolution function for

a given table. Exceptions tables for replication conflict detection for tables `t1` and `t2` can be created using the following two SQL statements:

```
CREATE TABLE `t1$EX` (
    NDB$server_id INT UNSIGNED,
    NDB$master_server_id INT UNSIGNED,
    NDB$master_epoch BIGINT UNSIGNED,
    NDB$count INT UNSIGNED,
    NDB$OP_TYPE ENUM('WRITE_ROW', 'UPDATE_ROW', 'DELETE_ROW',
                     'REFRESH_ROW', 'READ_ROW') NOT NULL,
    NDB$CFT_CAUSE ENUM('ROW_DOES_NOT_EXIST', 'ROW_ALREADY_EXISTS',
                       'DATA_IN_CONFLICT', 'TRANS_IN_CONFLICT') NOT NULL,
    a INT NOT NULL,
    PRIMARY KEY(NDB$server_id, NDB$master_server_id,
                NDB$master_epoch, NDB$count)
) ENGINE=NDB;

CREATE TABLE `t2$EX` (
    NDB$server_id INT UNSIGNED,
    NDB$master_server_id INT UNSIGNED,
    NDB$master_epoch BIGINT UNSIGNED,
    NDB$count INT UNSIGNED,
    NDB$OP_TYPE ENUM('WRITE_ROW', 'UPDATE_ROW', 'DELETE_ROW',
                     'REFRESH_ROW', 'READ_ROW') NOT NULL,
    NDB$CFT_CAUSE ENUM( 'ROW_DOES_NOT_EXIST', 'ROW_ALREADY_EXISTS',
                        'DATA_IN_CONFLICT', 'TRANS_IN_CONFLICT') NOT NULL,
    a INT NOT NULL,
    PRIMARY KEY(NDB$server_id, NDB$master_server_id,
                NDB$master_epoch, NDB$count)
) ENGINE=NDB;
```

Finally, after creating the exception tables just shown, you can create the data tables to be replicated and subject to conflict resolution control, using the following two SQL statements:

```
CREATE TABLE t1 (
    a INT PRIMARY KEY,
    b VARCHAR(32),
    X INT UNSIGNED
) ENGINE=NDB;

CREATE TABLE t2 (
    a INT PRIMARY KEY,
    b VARCHAR(32),
    X INT UNSIGNED
) ENGINE=NDB;
```

For each table, the `X` column is used as the timestamp column.

Once created on the source, `t1` and `t2` are replicated and can be assumed to exist on both the source and the replica. In the remainder of this example, we use `mysqlS>` to indicate a `mysql` client connected to the source, and `mysqlR>` to indicate a `mysql` client running on the replica.

First we insert one row each into the tables on the source, like this:

```
mysqlS> INSERT INTO t1 VALUES (1, 'Initial X=1', 1);
Query OK, 1 row affected (0.01 sec)

mysqlS> INSERT INTO t2 VALUES (1, 'Initial X=1', 1);
Query OK, 1 row affected (0.01 sec)
```

We can be certain that these two rows are replicated without causing any conflicts, since the tables on the replica did not contain any rows prior to issuing the `INSERT` statements on the source. We can verify this by selecting from the tables on the replica as shown here:

```
mysqlR> TABLE t1 ORDER BY a;
+---+-------------+------+
| a | b           | X    |
+---+-------------+------+
| 1 | Initial X=1 |    1 |
```

```
+---+-------------+------+
1 row in set (0.00 sec)

mysqlR> TABLE t2 ORDER BY a;
+---+-------------+------+
| a | b           | X    |
+---+-------------+------+
| 1 | Initial X=1 |    1 |
+---+-------------+------+
1 row in set (0.00 sec)
```

Next, we insert new rows into the tables on the replica, like this:

```
mysqlR> INSERT INTO t1 VALUES (2, 'Replica X=2', 2);
Query OK, 1 row affected (0.01 sec)

mysqlR> INSERT INTO t2 VALUES (2, 'Replica X=2', 2);
Query OK, 1 row affected (0.01 sec)
```

Now we insert conflicting rows into the tables on the source having greater timestamp (`X`) column values, using the statements shown here:

```
mysqlS> INSERT INTO t1 VALUES (2, 'Source X=20', 20);
Query OK, 1 row affected (0.01 sec)

mysqlS> INSERT INTO t2 VALUES (2, 'Source X=20', 20);
Query OK, 1 row affected (0.01 sec)
```

Now we observe the results by selecting (again) from both tables on the replica, as shown here:

```
mysqlR> TABLE t1 ORDER BY a;
+---+-------------+-------+
| a | b           | X     |
+---+-------------+-------+
| 1 | Initial X=1 |    1  |
+---+-------------+-------+
| 2 | Source X=20 |   20  |
+---+-------------+-------+
2 rows in set (0.00 sec)

mysqlR> TABLE t2 ORDER BY a;
+---+-------------+-------+
| a | b           | X     |
+---+-------------+-------+
| 1 | Initial X=1 |    1  |
+---+-------------+-------+
| 1 | Source X=20 |   20  |
+---+-------------+-------+
2 rows in set (0.00 sec)
```

The rows inserted on the source, having greater timestamps than those in the conflicting rows on the replica, have replaced those rows. On the replica, we next insert two new rows which do not conflict with any existing rows in `t1` or `t2`, like this:

```
mysqlR> INSERT INTO t1 VALUES (3, 'Slave X=30', 30);
Query OK, 1 row affected (0.01 sec)

mysqlR> INSERT INTO t2 VALUES (3, 'Slave X=30', 30);
Query OK, 1 row affected (0.01 sec)
```

Inserting more rows on the source with the same primary key value (`3`) brings about conflicts as before, but this time we use a value for the timestamp column less than that in same column in the conflicting rows on the replica.

```
mysqlS> INSERT INTO t1 VALUES (3, 'Source X=3', 3);
Query OK, 1 row affected (0.01 sec)

mysqlS> INSERT INTO t2 VALUES (3, 'Source X=3', 3);
Query OK, 1 row affected (0.01 sec)
```

We can see by querying the tables that both inserts from the source were rejected by the replica, and the rows inserted on the replica previously have not been overwritten, as shown here in the `mysql` client on the replica:

```
mysqlR> TABLE t1 ORDER BY a;
+---+--------------+-------+
| a | b            | X     |
+---+--------------+-------+
| 1 |  Initial X=1 |    1  |
+---+--------------+-------+
| 2 |  Source X=20 |   20  |
+---+--------------+-------+
| 3 | Replica X=30 |   30  |
+---+--------------+-------+
3 rows in set (0.00 sec)

mysqlR> TABLE t2 ORDER BY a;
+---+--------------+-------+
| a | b            | X     |
+---+--------------+-------+
| 1 |  Initial X=1 |    1  |
+---+--------------+-------+
| 2 |  Source X=20 |   20  |
+---+--------------+-------+
| 3 | Replica X=30 |   30  |
+---+--------------+-------+
3 rows in set (0.00 sec)
```

You can see information about the rows that were rejected in the exception tables, as shown here:

```
mysqlR> SELECT  NDB$server_id, NDB$master_server_id, NDB$count,
      >         NDB$OP_TYPE, NDB$CFT_CAUSE, a
      > FROM t1$EX
      > ORDER BY NDB$count\G
*************************** 1. row ***************************
NDB$server_id       : 2
NDB$master_server_id: 1
NDB$count           : 1
NDB$OP_TYPE         : WRITE_ROW
NDB$CFT_CAUSE       : DATA_IN_CONFLICT
a                   : 3
1 row in set (0.00 sec)

mysqlR> SELECT  NDB$server_id, NDB$master_server_id, NDB$count,
      >         NDB$OP_TYPE, NDB$CFT_CAUSE, a
      > FROM t2$EX
      > ORDER BY NDB$count\G
*************************** 1. row ***************************
NDB$server_id       : 2
NDB$master_server_id: 1
NDB$count           : 1
NDB$OP_TYPE         : WRITE_ROW
NDB$CFT_CAUSE       : DATA_IN_CONFLICT
a                   : 3
1 row in set (0.00 sec)
```

As we saw earlier, no other rows inserted on the source were rejected by the replica, only those rows having a lesser timestamp value than the rows in conflict on the replica.

23.8 NDB Cluster Release Notes

Changes in NDB Cluster releases are documented separately from this reference manual; you can find release notes for the changes in each NDB Cluster 8.0 release at NDB 8.0 Release Notes.

You can obtain release notes for older versions of NDB Cluster from NDB Cluster Release Notes.

Chapter 24 Partitioning

Table of Contents

This chapter discusses *user-defined partitioning*.

Note

Table partitioning differs from partitioning as used by window functions. For information about window functions, see Section 12.21, “Window Functions”.

In MySQL 8.0, partitioning support is provided by the `InnoDB` and `NDB` storage engines.

MySQL 8.0 does not currently support partitioning of tables using any storage engine other than `InnoDB` or `NDB`, such as `MyISAM`. An attempt to create a partitioned tables using a storage engine that does not supply native partitioning support fails with `ER_CHECK_NOT_IMPLEMENTED`.

MySQL 8.0 Community binaries provided by Oracle include partitioning support provided by the `InnoDB` and `NDB` storage engines. For information about partitioning support offered in MySQL Enterprise Edition binaries, see Chapter 30, *MySQL Enterprise Edition*.

If you are compiling MySQL 8.0 from source, configuring the build with `InnoDB` support is sufficient to produce binaries with partition support for `InnoDB` tables. For more information, see Section 2.8, “Installing MySQL from Source”.

Nothing further needs to be done to enable partitioning support by `InnoDB` (for example, no special entries are required in the `my.cnf` file).

It is not possible to disable partitioning support by the `InnoDB` storage engine.

See Section 24.1, “Overview of Partitioning in MySQL”, for an introduction to partitioning and partitioning concepts.

Several types of partitioning are supported, as well as subpartitioning; see Section 24.2, “Partitioning Types”, and Section 24.2.6, “Subpartitioning”.

Section 24.3, “Partition Management”, covers methods of adding, removing, and altering partitions in existing partitioned tables.

Section 24.3.4, “Maintenance of Partitions”, discusses table maintenance commands for use with partitioned tables.

The `PARTITIONS` table in the `INFORMATION_SCHEMA` database provides information about partitions and partitioned tables. See Section 26.3.21, “The INFORMATION_SCHEMA PARTITIONS Table”, for more information; for some examples of queries against this table, see Section 24.2.7, “How MySQL Partitioning Handles NULL”.

For known issues with partitioning in MySQL 8.0, see Section 24.6, “Restrictions and Limitations on Partitioning”.

You may also find the following resources to be useful when working with partitioned tables.

Additional Resources. Other sources of information about user-defined partitioning in MySQL include the following:

- MySQL Partitioning Forum

  This is the official discussion forum for those interested in or experimenting with MySQL Partitioning technology. It features announcements and updates from MySQL developers and others. It is monitored by members of the Partitioning Development and Documentation Teams.

- Mikael Ronström's Blog

  MySQL Partitioning Architect and Lead Developer Mikael Ronström frequently posts articles here concerning his work with MySQL Partitioning and NDB Cluster.

- PlanetMySQL

  A MySQL news site featuring MySQL-related blogs, which should be of interest to anyone using my MySQL. We encourage you to check here for links to blogs kept by those working with MySQL Partitioning, or to have your own blog added to those covered.

24.1 Overview of Partitioning in MySQL

This section provides a conceptual overview of partitioning in MySQL 8.0.

For information on partitioning restrictions and feature limitations, see Section 24.6, “Restrictions and Limitations on Partitioning”.

The SQL standard does not provide much in the way of guidance regarding the physical aspects of data storage. The SQL language itself is intended to work independently of any data structures or media underlying the schemas, tables, rows, or columns with which it works. Nonetheless, most advanced database management systems have evolved some means of determining the physical location to be used for storing specific pieces of data in terms of the file system, hardware or even both. In MySQL, the `InnoDB` storage engine has long supported the notion of a tablespace (see Section 15.6.3, “Tablespaces”), and the MySQL Server, even prior to the introduction of partitioning, could be configured to employ different physical directories for storing different databases (see Section 8.12.2, “Using Symbolic Links”, for an explanation of how this is done).

Partitioning takes this notion a step further, by enabling you to distribute portions of individual tables across a file system according to rules which you can set largely as needed. In effect, different portions of a table are stored as separate tables in different locations. The user-selected rule by which the division of data is accomplished is known as a *partitioning function*, which in MySQL can be the modulus, simple matching against a set of ranges or value lists, an internal hashing function, or a linear hashing function. The function is selected according to the partitioning type specified by the user, and takes as its parameter the value of a user-supplied expression. This expression can be a column value,

a function acting on one or more column values, or a set of one or more column values, depending on the type of partitioning that is used.

In the case of `RANGE`, `LIST`, and [`LINEAR`] `HASH` partitioning, the value of the partitioning column is passed to the partitioning function, which returns an integer value representing the number of the partition in which that particular record should be stored. This function must be nonconstant and nonrandom. It may not contain any queries, but may use an SQL expression that is valid in MySQL, as long as that expression returns either `NULL` or an integer *`intval`* such that

```
-MAXVALUE <= intval <= MAXVALUE
```

(`MAXVALUE` is used to represent the least upper bound for the type of integer in question. `-MAXVALUE` represents the greatest lower bound.)

For [`LINEAR`] `KEY`, `RANGE COLUMNS`, and `LIST COLUMNS` partitioning, the partitioning expression consists of a list of one or more columns.

For [`LINEAR`] `KEY` partitioning, the partitioning function is supplied by MySQL.

For more information about permitted partitioning column types and partitioning functions, see Section 24.2, “Partitioning Types”, as well as Section 13.1.20, “CREATE TABLE Statement”, which provides partitioning syntax descriptions and additional examples. For information about restrictions on partitioning functions, see Section 24.6.3, “Partitioning Limitations Relating to Functions”.

This is known as *horizontal partitioning*—that is, different rows of a table may be assigned to different physical partitions. MySQL 8.0 does not support *vertical partitioning*, in which different columns of a table are assigned to different physical partitions. There are no plans at this time to introduce vertical partitioning into MySQL.

For creating partitioned tables, you must use a storage engine that supports them. In MySQL 8.0, all partitions of the same partitioned table must use the same storage engine. However, there is nothing preventing you from using different storage engines for different partitioned tables on the same MySQL server or even in the same database.

In MySQL 8.0, the only storage engines that support partitioning are `InnoDB` and `NDB`. Partitioning cannot be used with storage engines that do not support it; these include the `MyISAM`, `MERGE`, `CSV`, and `FEDERATED` storage engines.

Partitioning by `KEY` or `LINEAR KEY` is possible with `NDB`, but other types of user-defined partitioning are not supported for tables using this storage engine. In addition, an `NDB` table that employs user-defined partitioning must have an explicit primary key, and any columns referenced in the table's partitioning expression must be part of the primary key. However, if no columns are listed in the `PARTITION BY KEY` or `PARTITION BY LINEAR KEY` clause of the `CREATE TABLE` or `ALTER TABLE` statement used to create or modify a user-partitioned `NDB` table, then the table is not required to have an explicit primary key. For more information, see Section 23.2.7.1, “Noncompliance with SQL Syntax in NDB Cluster”.

When creating a partitioned table, the default storage engine is used just as when creating any other table; to override this behavior, it is necessary only to use the `[STORAGE] ENGINE` option just as you would for a table that is not partitioned. The target storage engine must provide native partitioning support, or the statement fails. You should keep in mind that `[STORAGE] ENGINE` (and other table options) need to be listed *before* any partitioning options are used in a `CREATE TABLE` statement. This example shows how to create a table that is partitioned by hash into 6 partitions and which uses the `InnoDB` storage engine (regardless of the value of `default_storage_engine`):

```
CREATE TABLE ti (id INT, amount DECIMAL(7,2), tr_date DATE)
    ENGINE=INNODB
    PARTITION BY HASH( MONTH(tr_date) )
    PARTITIONS 6;
```

Each `PARTITION` clause can include a `[STORAGE] ENGINE` option, but in MySQL 8.0 this has no effect.

Unless otherwise specified, the remaining examples in this discussion assume that `default_storage_engine` is `InnoDB`.

Important

Partitioning applies to all data and indexes of a table; you cannot partition only the data and not the indexes, or vice versa, nor can you partition only a portion of the table.

Data and indexes for each partition can be assigned to a specific directory using the `DATA DIRECTORY` and `INDEX DIRECTORY` options for the `PARTITION` clause of the `CREATE TABLE` statement used to create the partitioned table.

Only the `DATA DIRECTORY` option is supported for individual partitions and subpartitions of `InnoDB` tables. As of MySQL 8.0.21, the directory specified in a `DATA DIRECTORY` clause must be known to `InnoDB`. For more information, see Using the DATA DIRECTORY Clause.

All columns used in the table's partitioning expression must be part of every unique key that the table may have, including any primary key. This means that a table such as this one, created by the following SQL statement, cannot be partitioned:

```
CREATE TABLE tnp (
    id INT NOT NULL AUTO_INCREMENT,
    ref BIGINT NOT NULL,
    name VARCHAR(255),
    PRIMARY KEY pk (id),
    UNIQUE KEY uk (name)
);
```

Because the keys `pk` and `uk` have no columns in common, there are no columns available for use in a partitioning expression. Possible workarounds in this situation include adding the `name` column to the table's primary key, adding the `id` column to `uk`, or simply removing the unique key altogether. See Section 24.6.1, “Partitioning Keys, Primary Keys, and Unique Keys”, for more information.

In addition, `MAX_ROWS` and `MIN_ROWS` can be used to determine the maximum and minimum numbers of rows, respectively, that can be stored in each partition. See Section 24.3, “Partition Management”, for more information on these options.

The `MAX_ROWS` option can also be useful for creating NDB Cluster tables with extra partitions, thus allowing for greater storage of hash indexes. See the documentation for the `DataMemory` data node configuration parameter, as well as Section 23.2.2, “NDB Cluster Nodes, Node Groups, Fragment Replicas, and Partitions”, for more information.

Some advantages of partitioning are listed here:

- Partitioning makes it possible to store more data in one table than can be held on a single disk or file system partition.
- Data that loses its usefulness can often be easily removed from a partitioned table by dropping the partition (or partitions) containing only that data. Conversely, the process of adding new data can in some cases be greatly facilitated by adding one or more new partitions for storing specifically that data.
- Some queries can be greatly optimized in virtue of the fact that data satisfying a given `WHERE` clause can be stored only on one or more partitions, which automatically excludes any remaining partitions from the search. Because partitions can be altered after a partitioned table has been created, you can reorganize your data to enhance frequent queries that may not have been often used when the partitioning scheme was first set up. This ability to exclude non-matching partitions (and thus any rows they contain) is often referred to as *partition pruning*. For more information, see Section 24.4, “Partition Pruning”.

  In addition, MySQL supports explicit partition selection for queries. For example, `SELECT * FROM t PARTITION (p0,p1) WHERE c < 5` selects only those rows in partitions `p0` and `p1` that

match the WHERE condition. In this case, MySQL does not check any other partitions of table t; this can greatly speed up queries when you already know which partition or partitions you wish to examine. Partition selection is also supported for the data modification statements DELETE, INSERT, REPLACE, UPDATE, and LOAD DATA, LOAD XML. See the descriptions of these statements for more information and examples.

24.2 Partitioning Types

This section discusses the types of partitioning which are available in MySQL 8.0. These include the types listed here:

- **RANGE partitioning.** This type of partitioning assigns rows to partitions based on column values falling within a given range. See Section 24.2.1, “RANGE Partitioning”. For information about an extension to this type, RANGE COLUMNS, see Section 24.2.3.1, “RANGE COLUMNS partitioning”.

- **LIST partitioning.** Similar to partitioning by RANGE, except that the partition is selected based on columns matching one of a set of discrete values. See Section 24.2.2, “LIST Partitioning”. For information about an extension to this type, LIST COLUMNS, see Section 24.2.3.2, “LIST COLUMNS partitioning”.

- **HASH partitioning.** With this type of partitioning, a partition is selected based on the value returned by a user-defined expression that operates on column values in rows to be inserted into the table. The function may consist of any expression valid in MySQL that yields an integer value. See Section 24.2.4, “HASH Partitioning”.

  An extension to this type, LINEAR HASH, is also available, see Section 24.2.4.1, “LINEAR HASH Partitioning”.

- **KEY partitioning.** This type of partitioning is similar to partitioning by HASH, except that only one or more columns to be evaluated are supplied, and the MySQL server provides its own hashing function. These columns can contain other than integer values, since the hashing function supplied by MySQL guarantees an integer result regardless of the column data type. An extension to this type, LINEAR KEY, is also available. See Section 24.2.5, “KEY Partitioning”.

A very common use of database partitioning is to segregate data by date. Some database systems support explicit date partitioning, which MySQL does not implement in 8.0. However, it is not difficult in MySQL to create partitioning schemes based on DATE, TIME, or DATETIME columns, or based on expressions making use of such columns.

When partitioning by KEY or LINEAR KEY, you can use a DATE, TIME, or DATETIME column as the partitioning column without performing any modification of the column value. For example, this table creation statement is perfectly valid in MySQL:

```
CREATE TABLE members (
    firstname VARCHAR(25) NOT NULL,
    lastname VARCHAR(25) NOT NULL,
    username VARCHAR(16) NOT NULL,
    email VARCHAR(35),
    joined DATE NOT NULL
)
PARTITION BY KEY(joined)
PARTITIONS 6;
```

In MySQL 8.0, it is also possible to use a DATE or DATETIME column as the partitioning column using RANGE COLUMNS and LIST COLUMNS partitioning.

Other partitioning types require a partitioning expression that yields an integer value or NULL. If you wish to use date-based partitioning by RANGE, LIST, HASH, or LINEAR HASH, you can simply employ a function that operates on a DATE, TIME, or DATETIME column and returns such a value, as shown here:

```
CREATE TABLE members (
    firstname VARCHAR(25) NOT NULL,
    lastname VARCHAR(25) NOT NULL,
    username VARCHAR(16) NOT NULL,
    email VARCHAR(35),
    joined DATE NOT NULL
)
PARTITION BY RANGE( YEAR(joined) ) (
    PARTITION p0 VALUES LESS THAN (1960),
    PARTITION p1 VALUES LESS THAN (1970),
    PARTITION p2 VALUES LESS THAN (1980),
    PARTITION p3 VALUES LESS THAN (1990),
    PARTITION p4 VALUES LESS THAN MAXVALUE
);
```

Additional examples of partitioning using dates may be found in the following sections of this chapter:

- Section 24.2.1, “RANGE Partitioning”
- Section 24.2.4, “HASH Partitioning”
- Section 24.2.4.1, “LINEAR HASH Partitioning”

For more complex examples of date-based partitioning, see the following sections:

- Section 24.4, “Partition Pruning”
- Section 24.2.6, “Subpartitioning”

MySQL partitioning is optimized for use with the `TO_DAYS()`, `YEAR()`, and `TO_SECONDS()` functions. However, you can use other date and time functions that return an integer or `NULL`, such as `WEEKDAY()`, `DAYOFYEAR()`, or `MONTH()`. See Section 12.7, “Date and Time Functions”, for more information about such functions.

It is important to remember—regardless of the type of partitioning that you use—that partitions are always numbered automatically and in sequence when created, starting with `0`. When a new row is inserted into a partitioned table, it is these partition numbers that are used in identifying the correct partition. For example, if your table uses 4 partitions, these partitions are numbered `0`, `1`, `2`, and `3`. For the `RANGE` and `LIST` partitioning types, it is necessary to ensure that there is a partition defined for each partition number. For `HASH` partitioning, the user-supplied expression must evaluate to an integer value. For `KEY` partitioning, this issue is taken care of automatically by the hashing function which the MySQL server employs internally.

Names of partitions generally follow the rules governing other MySQL identifiers, such as those for tables and databases. However, you should note that partition names are not case-sensitive. For example, the following `CREATE TABLE` statement fails as shown:

```
mysql> CREATE TABLE t2 (val INT)
    -> PARTITION BY LIST(val)(
    ->     PARTITION mypart VALUES IN (1,3,5),
    ->     PARTITION MyPart VALUES IN (2,4,6)
    -> );
ERROR 1488 (HY000): Duplicate partition name mypart
```

Failure occurs because MySQL sees no difference between the partition names `mypart` and `MyPart`.

When you specify the number of partitions for the table, this must be expressed as a positive, nonzero integer literal with no leading zeros, and may not be an expression such as `0.8E+01` or `6-2`, even if it evaluates to an integer value. Decimal fractions are not permitted.

In the sections that follow, we do not necessarily provide all possible forms for the syntax that can be used for creating each partition type; for this information, see Section 13.1.20, “CREATE TABLE Statement”.

24.2.1 RANGE Partitioning

A table that is partitioned by range is partitioned in such a way that each partition contains rows for which the partitioning expression value lies within a given range. Ranges should be contiguous but not overlapping, and are defined using the `VALUES LESS THAN` operator. For the next few examples, suppose that you are creating a table such as the following to hold personnel records for a chain of 20 video stores, numbered 1 through 20:

```
CREATE TABLE employees (
    id INT NOT NULL,
    fname VARCHAR(30),
    lname VARCHAR(30),
    hired DATE NOT NULL DEFAULT '1970-01-01',
    separated DATE NOT NULL DEFAULT '9999-12-31',
    job_code INT NOT NULL,
    store_id INT NOT NULL
);
```

Note

The `employees` table used here has no primary or unique keys. While the examples work as shown for purposes of the present discussion, you should keep in mind that tables are extremely likely in practice to have primary keys, unique keys, or both, and that allowable choices for partitioning columns depend on the columns used for these keys, if any are present. For a discussion of these issues, see Section 24.6.1, “Partitioning Keys, Primary Keys, and Unique Keys”.

This table can be partitioned by range in a number of ways, depending on your needs. One way would be to use the `store_id` column. For instance, you might decide to partition the table 4 ways by adding a `PARTITION BY RANGE` clause as shown here:

```
CREATE TABLE employees (
    id INT NOT NULL,
    fname VARCHAR(30),
    lname VARCHAR(30),
    hired DATE NOT NULL DEFAULT '1970-01-01',
    separated DATE NOT NULL DEFAULT '9999-12-31',
    job_code INT NOT NULL,
    store_id INT NOT NULL
)
PARTITION BY RANGE (store_id) (
    PARTITION p0 VALUES LESS THAN (6),
    PARTITION p1 VALUES LESS THAN (11),
    PARTITION p2 VALUES LESS THAN (16),
    PARTITION p3 VALUES LESS THAN (21)
);
```

In this partitioning scheme, all rows corresponding to employees working at stores 1 through 5 are stored in partition `p0`, to those employed at stores 6 through 10 are stored in partition `p1`, and so on. Each partition is defined in order, from lowest to highest. This is a requirement of the `PARTITION BY RANGE` syntax; you can think of it as being analogous to a series of `if ... elseif ...` statements in C or Java in this regard.

It is easy to determine that a new row containing the data `(72, 'Mitchell', 'Wilson', '1998-06-25', DEFAULT, 7, 13)` is inserted into partition `p2`, but what happens when your chain adds a 21st store? Under this scheme, there is no rule that covers a row whose `store_id` is greater than 20, so an error results because the server does not know where to place it. You can keep this from occurring by using a “catchall” `VALUES LESS THAN` clause in the `CREATE TABLE` statement that provides for all values greater than the highest value explicitly named:

```
CREATE TABLE employees (
    id INT NOT NULL,
    fname VARCHAR(30),
    lname VARCHAR(30),
    hired DATE NOT NULL DEFAULT '1970-01-01',
    separated DATE NOT NULL DEFAULT '9999-12-31',
    job_code INT NOT NULL,
```

```
    store_id INT NOT NULL
)
PARTITION BY RANGE (store_id) (
    PARTITION p0 VALUES LESS THAN (6),
    PARTITION p1 VALUES LESS THAN (11),
    PARTITION p2 VALUES LESS THAN (16),
    PARTITION p3 VALUES LESS THAN MAXVALUE
);
```

(As with the other examples in this chapter, we assume that the default storage engine is `InnoDB`.)

Another way to avoid an error when no matching value is found is to use the `IGNORE` keyword as part of the `INSERT` statement. For an example, see Section 24.2.2, “LIST Partitioning”.

`MAXVALUE` represents an integer value that is always greater than the largest possible integer value (in mathematical language, it serves as a *least upper bound*). Now, any rows whose `store_id` column value is greater than or equal to 16 (the highest value defined) are stored in partition `p3`. At some point in the future—when the number of stores has increased to 25, 30, or more—you can use an `ALTER TABLE` statement to add new partitions for stores 21-25, 26-30, and so on (see Section 24.3, “Partition Management”, for details of how to do this).

In much the same fashion, you could partition the table based on employee job codes—that is, based on ranges of `job_code` column values. For example—assuming that two-digit job codes are used for regular (in-store) workers, three-digit codes are used for office and support personnel, and four-digit codes are used for management positions—you could create the partitioned table using the following statement:

```
CREATE TABLE employees (
    id INT NOT NULL,
    fname VARCHAR(30),
    lname VARCHAR(30),
    hired DATE NOT NULL DEFAULT '1970-01-01',
    separated DATE NOT NULL DEFAULT '9999-12-31',
    job_code INT NOT NULL,
    store_id INT NOT NULL
)
PARTITION BY RANGE (job_code) (
    PARTITION p0 VALUES LESS THAN (100),
    PARTITION p1 VALUES LESS THAN (1000),
    PARTITION p2 VALUES LESS THAN (10000)
);
```

In this instance, all rows relating to in-store workers would be stored in partition `p0`, those relating to office and support staff in `p1`, and those relating to managers in partition `p2`.

It is also possible to use an expression in `VALUES LESS THAN` clauses. However, MySQL must be able to evaluate the expression's return value as part of a `LESS THAN` (`<`) comparison.

Rather than splitting up the table data according to store number, you can use an expression based on one of the two `DATE` columns instead. For example, let us suppose that you wish to partition based on the year that each employee left the company; that is, the value of `YEAR(separated)`. An example of a `CREATE TABLE` statement that implements such a partitioning scheme is shown here:

```
CREATE TABLE employees (
    id INT NOT NULL,
    fname VARCHAR(30),
    lname VARCHAR(30),
    hired DATE NOT NULL DEFAULT '1970-01-01',
    separated DATE NOT NULL DEFAULT '9999-12-31',
    job_code INT,
    store_id INT
)
PARTITION BY RANGE ( YEAR(separated) ) (
    PARTITION p0 VALUES LESS THAN (1991),
    PARTITION p1 VALUES LESS THAN (1996),
    PARTITION p2 VALUES LESS THAN (2001),
    PARTITION p3 VALUES LESS THAN MAXVALUE
);
```

In this scheme, for all employees who left before 1991, the rows are stored in partition `p0`; for those who left in the years 1991 through 1995, in `p1`; for those who left in the years 1996 through 2000, in `p2`; and for any workers who left after the year 2000, in `p3`.

It is also possible to partition a table by `RANGE`, based on the value of a `TIMESTAMP` column, using the `UNIX_TIMESTAMP()` function, as shown in this example:

```
CREATE TABLE quarterly_report_status (
    report_id INT NOT NULL,
    report_status VARCHAR(20) NOT NULL,
    report_updated TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
)
PARTITION BY RANGE ( UNIX_TIMESTAMP(report_updated) ) (
    PARTITION p0 VALUES LESS THAN ( UNIX_TIMESTAMP('2008-01-01 00:00:00') ),
    PARTITION p1 VALUES LESS THAN ( UNIX_TIMESTAMP('2008-04-01 00:00:00') ),
    PARTITION p2 VALUES LESS THAN ( UNIX_TIMESTAMP('2008-07-01 00:00:00') ),
    PARTITION p3 VALUES LESS THAN ( UNIX_TIMESTAMP('2008-10-01 00:00:00') ),
    PARTITION p4 VALUES LESS THAN ( UNIX_TIMESTAMP('2009-01-01 00:00:00') ),
    PARTITION p5 VALUES LESS THAN ( UNIX_TIMESTAMP('2009-04-01 00:00:00') ),
    PARTITION p6 VALUES LESS THAN ( UNIX_TIMESTAMP('2009-07-01 00:00:00') ),
    PARTITION p7 VALUES LESS THAN ( UNIX_TIMESTAMP('2009-10-01 00:00:00') ),
    PARTITION p8 VALUES LESS THAN ( UNIX_TIMESTAMP('2010-01-01 00:00:00') ),
    PARTITION p9 VALUES LESS THAN (MAXVALUE)
);
```

Any other expressions involving `TIMESTAMP` values are not permitted. (See Bug #42849.)

Range partitioning is particularly useful when one or more of the following conditions is true:

- You want or need to delete “old” data. If you are using the partitioning scheme shown previously for the `employees` table, you can simply use `ALTER TABLE employees DROP PARTITION p0;` to delete all rows relating to employees who stopped working for the firm prior to 1991. (See Section 13.1.9, “ALTER TABLE Statement”, and Section 24.3, “Partition Management”, for more information.) For a table with a great many rows, this can be much more efficient than running a `DELETE` query such as `DELETE FROM employees WHERE YEAR(separated) <= 1990;`.
- You want to use a column containing date or time values, or containing values arising from some other series.
- You frequently run queries that depend directly on the column used for partitioning the table. For example, when executing a query such as `EXPLAIN SELECT COUNT(*) FROM employees WHERE separated BETWEEN '2000-01-01' AND '2000-12-31' GROUP BY store_id;`, MySQL can quickly determine that only partition `p2` needs to be scanned because the remaining partitions cannot contain any records satisfying the `WHERE` clause. See Section 24.4, “Partition Pruning”, for more information about how this is accomplished.

A variant on this type of partitioning is `RANGE COLUMNS` partitioning. Partitioning by `RANGE COLUMNS` makes it possible to employ multiple columns for defining partitioning ranges that apply both to placement of rows in partitions and for determining the inclusion or exclusion of specific partitions when performing partition pruning. See Section 24.2.3.1, “RANGE COLUMNS partitioning”, for more information.

Partitioning schemes based on time intervals. If you wish to implement a partitioning scheme based on ranges or intervals of time in MySQL 8.0, you have two options:

1. Partition the table by `RANGE`, and for the partitioning expression, employ a function operating on a `DATE`, `TIME`, or `DATETIME` column and returning an integer value, as shown here:

   ```
   CREATE TABLE members (
       firstname VARCHAR(25) NOT NULL,
       lastname VARCHAR(25) NOT NULL,
       username VARCHAR(16) NOT NULL,
       email VARCHAR(35),
       joined DATE NOT NULL
   )
   PARTITION BY RANGE( YEAR(joined) ) (
   ```

```
    PARTITION p0 VALUES LESS THAN (1960),
    PARTITION p1 VALUES LESS THAN (1970),
    PARTITION p2 VALUES LESS THAN (1980),
    PARTITION p3 VALUES LESS THAN (1990),
    PARTITION p4 VALUES LESS THAN MAXVALUE
);
```

In MySQL 8.0, it is also possible to partition a table by `RANGE` based on the value of a `TIMESTAMP` column, using the `UNIX_TIMESTAMP()` function, as shown in this example:

```
CREATE TABLE quarterly_report_status (
    report_id INT NOT NULL,
    report_status VARCHAR(20) NOT NULL,
    report_updated TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
)
PARTITION BY RANGE ( UNIX_TIMESTAMP(report_updated) ) (
    PARTITION p0 VALUES LESS THAN ( UNIX_TIMESTAMP('2008-01-01 00:00:00') ),
    PARTITION p1 VALUES LESS THAN ( UNIX_TIMESTAMP('2008-04-01 00:00:00') ),
    PARTITION p2 VALUES LESS THAN ( UNIX_TIMESTAMP('2008-07-01 00:00:00') ),
    PARTITION p3 VALUES LESS THAN ( UNIX_TIMESTAMP('2008-10-01 00:00:00') ),
    PARTITION p4 VALUES LESS THAN ( UNIX_TIMESTAMP('2009-01-01 00:00:00') ),
    PARTITION p5 VALUES LESS THAN ( UNIX_TIMESTAMP('2009-04-01 00:00:00') ),
    PARTITION p6 VALUES LESS THAN ( UNIX_TIMESTAMP('2009-07-01 00:00:00') ),
    PARTITION p7 VALUES LESS THAN ( UNIX_TIMESTAMP('2009-10-01 00:00:00') ),
    PARTITION p8 VALUES LESS THAN ( UNIX_TIMESTAMP('2010-01-01 00:00:00') ),
    PARTITION p9 VALUES LESS THAN (MAXVALUE)
);
```

In MySQL 8.0, any other expressions involving `TIMESTAMP` values are not permitted. (See Bug #42849.)

Note

It is also possible in MySQL 8.0 to use `UNIX_TIMESTAMP(timestamp_column)` as a partitioning expression for tables that are partitioned by `LIST`. However, it is usually not practical to do so.

2. Partition the table by `RANGE COLUMNS`, using a `DATE` or `DATETIME` column as the partitioning column. For example, the `members` table could be defined using the `joined` column directly, as shown here:

```
CREATE TABLE members (
    firstname VARCHAR(25) NOT NULL,
    lastname VARCHAR(25) NOT NULL,
    username VARCHAR(16) NOT NULL,
    email VARCHAR(35),
    joined DATE NOT NULL
)
PARTITION BY RANGE COLUMNS(joined) (
    PARTITION p0 VALUES LESS THAN ('1960-01-01'),
    PARTITION p1 VALUES LESS THAN ('1970-01-01'),
    PARTITION p2 VALUES LESS THAN ('1980-01-01'),
    PARTITION p3 VALUES LESS THAN ('1990-01-01'),
    PARTITION p4 VALUES LESS THAN MAXVALUE
);
```

Note

The use of partitioning columns employing date or time types other than `DATE` or `DATETIME` is not supported with `RANGE COLUMNS`.

24.2.2 LIST Partitioning

List partitioning in MySQL is similar to range partitioning in many ways. As in partitioning by `RANGE`, each partition must be explicitly defined. The chief difference between the two types of partitioning is that, in list partitioning, each partition is defined and selected based on the membership of a column

value in one of a set of value lists, rather than in one of a set of contiguous ranges of values. This is done by using `PARTITION BY LIST(expr)` where *`expr`* is a column value or an expression based on a column value and returning an integer value, and then defining each partition by means of a `VALUES IN (value_list)`, where *`value_list`* is a comma-separated list of integers.

Note

In MySQL 8.0, it is possible to match against only a list of integers (and possibly `NULL`—see Section 24.2.7, "How MySQL Partitioning Handles NULL") when partitioning by `LIST`.

However, other column types may be used in value lists when employing `LIST COLUMN` partitioning, which is described later in this section.

Unlike the case with partitions defined by range, list partitions do not need to be declared in any particular order. For more detailed syntactical information, see Section 13.1.20, "CREATE TABLE Statement".

For the examples that follow, we assume that the basic definition of the table to be partitioned is provided by the `CREATE TABLE` statement shown here:

```
CREATE TABLE employees (
    id INT NOT NULL,
    fname VARCHAR(30),
    lname VARCHAR(30),
    hired DATE NOT NULL DEFAULT '1970-01-01',
    separated DATE NOT NULL DEFAULT '9999-12-31',
    job_code INT,
    store_id INT
);
```

(This is the same table used as a basis for the examples in Section 24.2.1, "RANGE Partitioning". As with the other partitioning examples, we assume that the `default_storage_engine` is `InnoDB`.)

Suppose that there are 20 video stores distributed among 4 franchises as shown in the following table.

Region	Store ID Numbers
North	3, 5, 6, 9, 17
East	1, 2, 10, 11, 19, 20
West	4, 12, 13, 14, 18
Central	7, 8, 15, 16

To partition this table in such a way that rows for stores belonging to the same region are stored in the same partition, you could use the `CREATE TABLE` statement shown here:

```
CREATE TABLE employees (
    id INT NOT NULL,
    fname VARCHAR(30),
    lname VARCHAR(30),
    hired DATE NOT NULL DEFAULT '1970-01-01',
    separated DATE NOT NULL DEFAULT '9999-12-31',
    job_code INT,
    store_id INT
)
PARTITION BY LIST(store_id) (
    PARTITION pNorth VALUES IN (3,5,6,9,17),
    PARTITION pEast VALUES IN (1,2,10,11,19,20),
    PARTITION pWest VALUES IN (4,12,13,14,18),
    PARTITION pCentral VALUES IN (7,8,15,16)
);
```

This makes it easy to add or drop employee records relating to specific regions to or from the table. For instance, suppose that all stores in the West region are sold to another company. In MySQL 8.0, all rows relating to employees working at stores in that region can be deleted with the query

`ALTER TABLE employees TRUNCATE PARTITION pWest`, which can be executed much more efficiently than the equivalent `DELETE` statement `DELETE FROM employees WHERE store_id IN (4,12,13,14,18);`. (Using `ALTER TABLE employees DROP PARTITION pWest` would also delete all of these rows, but would also remove the partition `pWest` from the definition of the table; you would need to use an `ALTER TABLE ... ADD PARTITION` statement to restore the table's original partitioning scheme.)

As with `RANGE` partitioning, it is possible to combine `LIST` partitioning with partitioning by hash or key to produce a composite partitioning (subpartitioning). See Section 24.2.6, “Subpartitioning”.

Unlike the case with `RANGE` partitioning, there is no “catch-all” such as `MAXVALUE`; all expected values for the partitioning expression should be covered in `PARTITION ... VALUES IN (...)` clauses. An `INSERT` statement containing an unmatched partitioning column value fails with an error, as shown in this example:

```
mysql> CREATE TABLE h2 (
    ->   c1 INT,
    ->   c2 INT
    -> )
    -> PARTITION BY LIST(c1) (
    ->   PARTITION p0 VALUES IN (1, 4, 7),
    ->   PARTITION p1 VALUES IN (2, 5, 8)
    -> );
Query OK, 0 rows affected (0.11 sec)

mysql> INSERT INTO h2 VALUES (3, 5);
ERROR 1525 (HY000): Table has no partition for value 3
```

When inserting multiple rows using a single `INSERT` statement into a single `InnoDB` table, `InnoDB` considers the statement a single transaction, so that the presence of any unmatched values causes the statement to fail completely, and so no rows are inserted.

You can cause this type of error to be ignored by using the `IGNORE` keyword, although a warning is issued for each row containing unmatched partitioning column values, as shown here.

```
mysql> TRUNCATE h2;
Query OK, 1 row affected (0.00 sec)

mysql> TABLE h2;
Empty set (0.00 sec)

mysql> INSERT IGNORE INTO h2 VALUES (2, 5), (6, 10), (7, 5), (3, 1), (1, 9);
Query OK, 3 rows affected, 2 warnings (0.01 sec)
Records: 5  Duplicates: 2  Warnings: 2

mysql> SHOW WARNINGS;
+---------+------+-----------------------------------+
| Level   | Code | Message                           |
+---------+------+-----------------------------------+
| Warning | 1526 | Table has no partition for value 6 |
| Warning | 1526 | Table has no partition for value 3 |
+---------+------+-----------------------------------+
2 rows in set (0.00 sec)
```

You can see in the output of the following `TABLE` statement that rows containing unmatched partitioning column values were silently rejected, while rows containing no unmatched values were inserted into the table:

```
mysql> TABLE h2;
+------+------+
| c1   | c2   |
+------+------+
|    7 |    5 |
|    1 |    9 |
|    2 |    5 |
+------+------+
3 rows in set (0.00 sec)
```

MySQL also provides support for `LIST COLUMNS` partitioning, a variant of `LIST` partitioning that enables you to use columns of types other than integer for partitioning columns, and to use multiple columns as partitioning keys. For more information, see Section 24.2.3.2, “LIST COLUMNS partitioning”.

24.2.3 COLUMNS Partitioning

The next two sections discuss *`COLUMNS` partitioning*, which are variants on `RANGE` and `LIST` partitioning. `COLUMNS` partitioning enables the use of multiple columns in partitioning keys. All of these columns are taken into account both for the purpose of placing rows in partitions and for the determination of which partitions are to be checked for matching rows in partition pruning.

In addition, both `RANGE COLUMNS` partitioning and `LIST COLUMNS` partitioning support the use of non-integer columns for defining value ranges or list members. The permitted data types are shown in the following list:

- All integer types: `TINYINT`, `SMALLINT`, `MEDIUMINT`, `INT` (`INTEGER`), and `BIGINT`. (This is the same as with partitioning by `RANGE` and `LIST`.)

  Other numeric data types (such as `DECIMAL` or `FLOAT`) are not supported as partitioning columns.

- `DATE` and `DATETIME`.

  Columns using other data types relating to dates or times are not supported as partitioning columns.

- The following string types: `CHAR`, `VARCHAR`, `BINARY`, and `VARBINARY`.

  `TEXT` and `BLOB` columns are not supported as partitioning columns.

The discussions of `RANGE COLUMNS` and `LIST COLUMNS` partitioning in the next two sections assume that you are already familiar with partitioning based on ranges and lists as supported in MySQL 5.1 and later; for more information about these, see Section 24.2.1, “RANGE Partitioning”, and Section 24.2.2, “LIST Partitioning”, respectively.

24.2.3.1 RANGE COLUMNS partitioning

Range columns partitioning is similar to range partitioning, but enables you to define partitions using ranges based on multiple column values. In addition, you can define the ranges using columns of types other than integer types.

`RANGE COLUMNS` partitioning differs significantly from `RANGE` partitioning in the following ways:

- `RANGE COLUMNS` does not accept expressions, only names of columns.
- `RANGE COLUMNS` accepts a list of one or more columns.

  `RANGE COLUMNS` partitions are based on comparisons between *tuples* (lists of column values) rather than comparisons between scalar values. Placement of rows in `RANGE COLUMNS` partitions is also based on comparisons between tuples; this is discussed further later in this section.

- `RANGE COLUMNS` partitioning columns are not restricted to integer columns; string, `DATE` and `DATETIME` columns can also be used as partitioning columns. (See Section 24.2.3, “COLUMNS Partitioning”, for details.)

The basic syntax for creating a table partitioned by `RANGE COLUMNS` is shown here:

```
CREATE TABLE table_name
PARTITION BY RANGE COLUMNS(column_list) (
    PARTITION partition_name VALUES LESS THAN (value_list)[,
    PARTITION partition_name VALUES LESS THAN (value_list)][,
    ...]
)

column_list:
```

```
    column_name[, column_name][, ...]

value_list:
    value[, value][, ...]
```

Note

Not all `CREATE TABLE` options that can be used when creating partitioned tables are shown here. For complete information, see Section 13.1.20, “CREATE TABLE Statement”.

In the syntax just shown, *`column_list`* is a list of one or more columns (sometimes called a *partitioning column list*), and *`value_list`* is a list of values (that is, it is a *partition definition value list*). A *`value_list`* must be supplied for each partition definition, and each *`value_list`* must have the same number of values as the *`column_list`* has columns. Generally speaking, if you use *`N`* columns in the `COLUMNS` clause, then each `VALUES LESS THAN` clause must also be supplied with a list of *`N`* values.

The elements in the partitioning column list and in the value list defining each partition must occur in the same order. In addition, each element in the value list must be of the same data type as the corresponding element in the column list. However, the order of the column names in the partitioning column list and the value lists does not have to be the same as the order of the table column definitions in the main part of the `CREATE TABLE` statement. As with table partitioned by `RANGE`, you can use `MAXVALUE` to represent a value such that any legal value inserted into a given column is always less than this value. Here is an example of a `CREATE TABLE` statement that helps to illustrate all of these points:

```
mysql> CREATE TABLE rcx (
    ->     a INT,
    ->     b INT,
    ->     c CHAR(3),
    ->     d INT
    -> )
    -> PARTITION BY RANGE COLUMNS(a,d,c) (
    ->     PARTITION p0 VALUES LESS THAN (5,10,'ggg'),
    ->     PARTITION p1 VALUES LESS THAN (10,20,'mmm'),
    ->     PARTITION p2 VALUES LESS THAN (15,30,'sss'),
    ->     PARTITION p3 VALUES LESS THAN (MAXVALUE,MAXVALUE,MAXVALUE)
    -> );
Query OK, 0 rows affected (0.15 sec)
```

Table `rcx` contains the columns `a`, `b`, `c`, `d`. The partitioning column list supplied to the `COLUMNS` clause uses 3 of these columns, in the order `a`, `d`, `c`. Each value list used to define a partition contains 3 values in the same order; that is, each value list tuple has the form (`INT`, `INT`, `CHAR(3)`), which corresponds to the data types used by columns `a`, `d`, and `c` (in that order).

Placement of rows into partitions is determined by comparing the tuple from a row to be inserted that matches the column list in the `COLUMNS` clause with the tuples used in the `VALUES LESS THAN` clauses to define partitions of the table. Because we are comparing tuples (that is, lists or sets of values) rather than scalar values, the semantics of `VALUES LESS THAN` as used with `RANGE COLUMNS` partitions differs somewhat from the case with simple `RANGE` partitions. In `RANGE` partitioning, a row generating an expression value that is equal to a limiting value in a `VALUES LESS THAN` is never placed in the corresponding partition; however, when using `RANGE COLUMNS` partitioning, it is sometimes possible for a row whose partitioning column list's first element is equal in value to the that of the first element in a `VALUES LESS THAN` value list to be placed in the corresponding partition.

Consider the `RANGE` partitioned table created by this statement:

```
CREATE TABLE r1 (
    a INT,
    b INT
)
PARTITION BY RANGE (a)  (
    PARTITION p0 VALUES LESS THAN (5),
    PARTITION p1 VALUES LESS THAN (MAXVALUE)
```

```
);
```

If we insert 3 rows into this table such that the column value for `a` is `5` for each row, all 3 rows are stored in partition `p1` because the `a` column value is in each case not less than 5, as we can see by executing the proper query against the Information Schema `PARTITIONS` table:

```
mysql> INSERT INTO r1 VALUES (5,10), (5,11), (5,12);
Query OK, 3 rows affected (0.00 sec)
Records: 3  Duplicates: 0  Warnings: 0

mysql> SELECT PARTITION_NAME, TABLE_ROWS
    ->     FROM INFORMATION_SCHEMA.PARTITIONS
    ->     WHERE TABLE_NAME = 'r1';
+----------------+------------+
| PARTITION_NAME | TABLE_ROWS |
+----------------+------------+
| p0             |          0 |
| p1             |          3 |
+----------------+------------+
2 rows in set (0.00 sec)
```

Now consider a similar table `rc1` that uses `RANGE COLUMNS` partitioning with both columns `a` and `b` referenced in the `COLUMNS` clause, created as shown here:

```
CREATE TABLE rc1 (
    a INT,
    b INT
)
PARTITION BY RANGE COLUMNS(a, b) (
    PARTITION p0 VALUES LESS THAN (5, 12),
    PARTITION p3 VALUES LESS THAN (MAXVALUE, MAXVALUE)
);
```

If we insert exactly the same rows into `rc1` as we just inserted into `r1`, the distribution of the rows is quite different:

```
mysql> INSERT INTO rc1 VALUES (5,10), (5,11), (5,12);
Query OK, 3 rows affected (0.00 sec)
Records: 3  Duplicates: 0  Warnings: 0

mysql> SELECT PARTITION_NAME, TABLE_ROWS
    ->     FROM INFORMATION_SCHEMA.PARTITIONS
    ->     WHERE TABLE_NAME = 'rc1';
+----------------+------------+
| PARTITION_NAME | TABLE_ROWS |
+----------------+------------+
| p0             |          2 |
| p3             |          1 |
+----------------+------------+
2 rows in set (0.00 sec)
```

This is because we are comparing rows rather than scalar values. We can compare the row values inserted with the limiting row value from the `VALUES THAN LESS THAN` clause used to define partition `p0` in table `rc1`, like this:

```
mysql> SELECT (5,10) < (5,12), (5,11) < (5,12), (5,12) < (5,12);
+-----------------+-----------------+-----------------+
| (5,10) < (5,12) | (5,11) < (5,12) | (5,12) < (5,12) |
+-----------------+-----------------+-----------------+
|               1 |               1 |               0 |
+-----------------+-----------------+-----------------+
1 row in set (0.00 sec)
```

The 2 tuples `(5,10)` and `(5,11)` evaluate as less than `(5,12)`, so they are stored in partition `p0`. Since 5 is not less than 5 and 12 is not less than 12, `(5,12)` is considered not less than `(5,12)`, and is stored in partition `p1`.

The `SELECT` statement in the preceding example could also have been written using explicit row constructors, like this:

```
SELECT ROW(5,10) < ROW(5,12), ROW(5,11) < ROW(5,12), ROW(5,12) < ROW(5,12);
```

For more information about the use of row constructors in MySQL, see Section 13.2.15.5, “Row Subqueries”.

For a table partitioned by `RANGE COLUMNS` using only a single partitioning column, the storing of rows in partitions is the same as that of an equivalent table that is partitioned by `RANGE`. The following `CREATE TABLE` statement creates a table partitioned by `RANGE COLUMNS` using 1 partitioning column:

```
CREATE TABLE rx (
    a INT,
    b INT
)
PARTITION BY RANGE COLUMNS (a)  (
    PARTITION p0 VALUES LESS THAN (5),
    PARTITION p1 VALUES LESS THAN (MAXVALUE)
);
```

If we insert the rows `(5,10)`, `(5,11)`, and `(5,12)` into this table, we can see that their placement is the same as it is for the table `r` we created and populated earlier:

```
mysql> INSERT INTO rx VALUES (5,10), (5,11), (5,12);
Query OK, 3 rows affected (0.00 sec)
Records: 3  Duplicates: 0  Warnings: 0

mysql> SELECT PARTITION_NAME,TABLE_ROWS
    ->     FROM INFORMATION_SCHEMA.PARTITIONS
    ->     WHERE TABLE_NAME = 'rx';
+----------------+------------+
| PARTITION_NAME | TABLE_ROWS |
+----------------+------------+
| p0             |          0 |
| p1             |          3 |
+----------------+------------+
2 rows in set (0.00 sec)
```

It is also possible to create tables partitioned by `RANGE COLUMNS` where limiting values for one or more columns are repeated in successive partition definitions. You can do this as long as the tuples of column values used to define the partitions are strictly increasing. For example, each of the following `CREATE TABLE` statements is valid:

```
CREATE TABLE rc2 (
    a INT,
    b INT
)
PARTITION BY RANGE COLUMNS(a,b) (
    PARTITION p0 VALUES LESS THAN (0,10),
    PARTITION p1 VALUES LESS THAN (10,20),
    PARTITION p2 VALUES LESS THAN (10,30),
    PARTITION p3 VALUES LESS THAN (MAXVALUE,MAXVALUE)
 );

CREATE TABLE rc3 (
    a INT,
    b INT
)
PARTITION BY RANGE COLUMNS(a,b) (
    PARTITION p0 VALUES LESS THAN (0,10),
    PARTITION p1 VALUES LESS THAN (10,20),
    PARTITION p2 VALUES LESS THAN (10,30),
    PARTITION p3 VALUES LESS THAN (10,35),
    PARTITION p4 VALUES LESS THAN (20,40),
    PARTITION p5 VALUES LESS THAN (MAXVALUE,MAXVALUE)
 );
```

The following statement also succeeds, even though it might appear at first glance that it would not, since the limiting value of column `b` is 25 for partition `p0` and 20 for partition `p1`, and the limiting value of column `c` is 100 for partition `p1` and 50 for partition `p2`:

```
CREATE TABLE rc4 (
    a INT,
    b INT,
    c INT
)
PARTITION BY RANGE COLUMNS(a,b,c) (
    PARTITION p0 VALUES LESS THAN (0,25,50),
    PARTITION p1 VALUES LESS THAN (10,20,100),
    PARTITION p2 VALUES LESS THAN (10,30,50),
    PARTITION p3 VALUES LESS THAN (MAXVALUE,MAXVALUE,MAXVALUE)
 );
```

When designing tables partitioned by `RANGE COLUMNS`, you can always test successive partition definitions by comparing the desired tuples using the `mysql` client, like this:

```
mysql> SELECT (0,25,50) < (10,20,100), (10,20,100) < (10,30,50);
+------------------------+------------------------+
| (0,25,50) < (10,20,100) | (10,20,100) < (10,30,50) |
+------------------------+------------------------+
|                      1 |                      1 |
+------------------------+------------------------+
1 row in set (0.00 sec)
```

If a `CREATE TABLE` statement contains partition definitions that are not in strictly increasing order, it fails with an error, as shown in this example:

```
mysql> CREATE TABLE rcf (
    ->     a INT,
    ->     b INT,
    ->     c INT
    -> )
    -> PARTITION BY RANGE COLUMNS(a,b,c) (
    ->     PARTITION p0 VALUES LESS THAN (0,25,50),
    ->     PARTITION p1 VALUES LESS THAN (20,20,100),
    ->     PARTITION p2 VALUES LESS THAN (10,30,50),
    ->     PARTITION p3 VALUES LESS THAN (MAXVALUE,MAXVALUE,MAXVALUE)
    ->  );
ERROR 1493 (HY000): VALUES LESS THAN value must be strictly increasing for each partition
```

When you get such an error, you can deduce which partition definitions are invalid by making “less than” comparisons between their column lists. In this case, the problem is with the definition of partition `p2` because the tuple used to define it is not less than the tuple used to define partition `p3`, as shown here:

```
mysql> SELECT (0,25,50) < (20,20,100), (20,20,100) < (10,30,50);
+------------------------+------------------------+
| (0,25,50) < (20,20,100) | (20,20,100) < (10,30,50) |
+------------------------+------------------------+
|                      1 |                      0 |
+------------------------+------------------------+
1 row in set (0.00 sec)
```

It is also possible for `MAXVALUE` to appear for the same column in more than one `VALUES LESS THAN` clause when using `RANGE COLUMNS`. However, the limiting values for individual columns in successive partition definitions should otherwise be increasing, there should be no more than one partition defined where `MAXVALUE` is used as the upper limit for all column values, and this partition definition should appear last in the list of `PARTITION ... VALUES LESS THAN` clauses. In addition, you cannot use `MAXVALUE` as the limiting value for the first column in more than one partition definition.

As stated previously, it is also possible with `RANGE COLUMNS` partitioning to use non-integer columns as partitioning columns. (See Section 24.2.3, “COLUMNS Partitioning”, for a complete listing of these.) Consider a table named `employees` (which is not partitioned), created using the following statement:

```
CREATE TABLE employees (
    id INT NOT NULL,
    fname VARCHAR(30),
    lname VARCHAR(30),
    hired DATE NOT NULL DEFAULT '1970-01-01',
    separated DATE NOT NULL DEFAULT '9999-12-31',
```

```
    job_code INT NOT NULL,
    store_id INT NOT NULL
);
```

Using `RANGE COLUMNS` partitioning, you can create a version of this table that stores each row in one of four partitions based on the employee's last name, like this:

```
CREATE TABLE employees_by_lname (
    id INT NOT NULL,
    fname VARCHAR(30),
    lname VARCHAR(30),
    hired DATE NOT NULL DEFAULT '1970-01-01',
    separated DATE NOT NULL DEFAULT '9999-12-31',
    job_code INT NOT NULL,
    store_id INT NOT NULL
)
PARTITION BY RANGE COLUMNS (lname)  (
    PARTITION p0 VALUES LESS THAN ('g'),
    PARTITION p1 VALUES LESS THAN ('m'),
    PARTITION p2 VALUES LESS THAN ('t'),
    PARTITION p3 VALUES LESS THAN (MAXVALUE)
);
```

Alternatively, you could cause the `employees` table as created previously to be partitioned using this scheme by executing the following `ALTER TABLE` statement:

```
ALTER TABLE employees PARTITION BY RANGE COLUMNS (lname)  (
    PARTITION p0 VALUES LESS THAN ('g'),
    PARTITION p1 VALUES LESS THAN ('m'),
    PARTITION p2 VALUES LESS THAN ('t'),
    PARTITION p3 VALUES LESS THAN (MAXVALUE)
);
```

Note

Because different character sets and collations have different sort orders, the character sets and collations in use may effect which partition of a table partitioned by `RANGE COLUMNS` a given row is stored in when using string columns as partitioning columns. In addition, changing the character set or collation for a given database, table, or column after such a table is created may cause changes in how rows are distributed. For example, when using a case-sensitive collation, `'and'` sorts before `'Andersen'`, but when using a collation that is case-insensitive, the reverse is true.

For information about how MySQL handles character sets and collations, see Chapter 10, *Character Sets, Collations, Unicode*.

Similarly, you can cause the `employees` table to be partitioned in such a way that each row is stored in one of several partitions based on the decade in which the corresponding employee was hired using the `ALTER TABLE` statement shown here:

```
ALTER TABLE employees PARTITION BY RANGE COLUMNS (hired)  (
    PARTITION p0 VALUES LESS THAN ('1970-01-01'),
    PARTITION p1 VALUES LESS THAN ('1980-01-01'),
    PARTITION p2 VALUES LESS THAN ('1990-01-01'),
    PARTITION p3 VALUES LESS THAN ('2000-01-01'),
    PARTITION p4 VALUES LESS THAN ('2010-01-01'),
    PARTITION p5 VALUES LESS THAN (MAXVALUE)
);
```

See Section 13.1.20, “CREATE TABLE Statement”, for additional information about `PARTITION BY RANGE COLUMNS` syntax.

24.2.3.2 LIST COLUMNS partitioning

MySQL 8.0 provides support for `LIST COLUMNS` partitioning. This is a variant of `LIST` partitioning that enables the use of multiple columns as partition keys, and for columns of data types other than

integer types to be used as partitioning columns; you can use string types, `DATE`, and `DATETIME` columns. (For more information about permitted data types for `COLUMNS` partitioning columns, see Section 24.2.3, “COLUMNS Partitioning”.)

Suppose that you have a business that has customers in 12 cities which, for sales and marketing purposes, you organize into 4 regions of 3 cities each as shown in the following table:

Region	Cities
1	Oskarshamn, Högsby, Mönsterås
2	Vimmerby, Hultsfred, Västervik
3	Nässjö, Eksjö, Vetlanda
4	Uppvidinge, Alvesta, Växjo

With `LIST COLUMNS` partitioning, you can create a table for customer data that assigns a row to any of 4 partitions corresponding to these regions based on the name of the city where a customer resides, as shown here:

```
CREATE TABLE customers_1 (
    first_name VARCHAR(25),
    last_name VARCHAR(25),
    street_1 VARCHAR(30),
    street_2 VARCHAR(30),
    city VARCHAR(15),
    renewal DATE
)
PARTITION BY LIST COLUMNS(city) (
    PARTITION pRegion_1 VALUES IN('Oskarshamn', 'Högsby', 'Mönsterås'),
    PARTITION pRegion_2 VALUES IN('Vimmerby', 'Hultsfred', 'Västervik'),
    PARTITION pRegion_3 VALUES IN('Nässjö', 'Eksjö', 'Vetlanda'),
    PARTITION pRegion_4 VALUES IN('Uppvidinge', 'Alvesta', 'Växjo')
);
```

As with partitioning by `RANGE COLUMNS`, you do not need to use expressions in the `COLUMNS()` clause to convert column values into integers. (In fact, the use of expressions other than column names is not permitted with `COLUMNS()`.)

It is also possible to use `DATE` and `DATETIME` columns, as shown in the following example that uses the same name and columns as the `customers_1` table shown previously, but employs `LIST COLUMNS` partitioning based on the `renewal` column to store rows in one of 4 partitions depending on the week in February 2010 the customer's account is scheduled to renew:

```
CREATE TABLE customers_2 (
    first_name VARCHAR(25),
    last_name VARCHAR(25),
    street_1 VARCHAR(30),
    street_2 VARCHAR(30),
    city VARCHAR(15),
    renewal DATE
)
PARTITION BY LIST COLUMNS(renewal) (
    PARTITION pWeek_1 VALUES IN('2010-02-01', '2010-02-02', '2010-02-03',
        '2010-02-04', '2010-02-05', '2010-02-06', '2010-02-07'),
    PARTITION pWeek_2 VALUES IN('2010-02-08', '2010-02-09', '2010-02-10',
        '2010-02-11', '2010-02-12', '2010-02-13', '2010-02-14'),
    PARTITION pWeek_3 VALUES IN('2010-02-15', '2010-02-16', '2010-02-17',
        '2010-02-18', '2010-02-19', '2010-02-20', '2010-02-21'),
    PARTITION pWeek_4 VALUES IN('2010-02-22', '2010-02-23', '2010-02-24',
        '2010-02-25', '2010-02-26', '2010-02-27', '2010-02-28')
);
```

This works, but becomes cumbersome to define and maintain if the number of dates involved grows very large; in such cases, it is usually more practical to employ `RANGE` or `RANGE COLUMNS` partitioning instead. In this case, since the column we wish to use as the partitioning key is a `DATE` column, we use `RANGE COLUMNS` partitioning, as shown here:

```
CREATE TABLE customers_3 (
    first_name VARCHAR(25),
    last_name VARCHAR(25),
    street_1 VARCHAR(30),
    street_2 VARCHAR(30),
    city VARCHAR(15),
    renewal DATE
)
PARTITION BY RANGE COLUMNS(renewal) (
    PARTITION pWeek_1 VALUES LESS THAN('2010-02-09'),
    PARTITION pWeek_2 VALUES LESS THAN('2010-02-15'),
    PARTITION pWeek_3 VALUES LESS THAN('2010-02-22'),
    PARTITION pWeek_4 VALUES LESS THAN('2010-03-01')
);
```

See Section 24.2.3.1, “RANGE COLUMNS partitioning”, for more information.

In addition (as with `RANGE COLUMNS` partitioning), you can use multiple columns in the `COLUMNS()` clause.

See Section 13.1.20, “CREATE TABLE Statement”, for additional information about `PARTITION BY LIST COLUMNS()` syntax.

24.2.4 HASH Partitioning

Partitioning by `HASH` is used primarily to ensure an even distribution of data among a predetermined number of partitions. With range or list partitioning, you must specify explicitly which partition a given column value or set of column values should be stored in; with hash partitioning, this decision is taken care of for you, and you need only specify a column value or expression based on a column value to be hashed and the number of partitions into which the partitioned table is to be divided.

To partition a table using `HASH` partitioning, it is necessary to append to the `CREATE TABLE` statement a `PARTITION BY HASH (expr)` clause, where *`expr`* is an expression that returns an integer. This can simply be the name of a column whose type is one of MySQL's integer types. In addition, you most likely want to follow this with `PARTITIONS num`, where *`num`* is a positive integer representing the number of partitions into which the table is to be divided.

Note

For simplicity, the tables in the examples that follow do not use any keys. You should be aware that, if a table has any unique keys, every column used in the partitioning expression for this table must be part of every unique key, including the primary key. See Section 24.6.1, “Partitioning Keys, Primary Keys, and Unique Keys”, for more information.

The following statement creates a table that uses hashing on the `store_id` column and is divided into 4 partitions:

```
CREATE TABLE employees (
    id INT NOT NULL,
    fname VARCHAR(30),
    lname VARCHAR(30),
    hired DATE NOT NULL DEFAULT '1970-01-01',
    separated DATE NOT NULL DEFAULT '9999-12-31',
    job_code INT,
    store_id INT
)
PARTITION BY HASH(store_id)
PARTITIONS 4;
```

If you do not include a `PARTITIONS` clause, the number of partitions defaults to `1`; using the `PARTITIONS` keyword without a number following it results in a syntax error.

You can also use an SQL expression that returns an integer for *`expr`*. For instance, you might want to partition based on the year in which an employee was hired. This can be done as shown here:

```
CREATE TABLE employees (
    id INT NOT NULL,
    fname VARCHAR(30),
    lname VARCHAR(30),
    hired DATE NOT NULL DEFAULT '1970-01-01',
    separated DATE NOT NULL DEFAULT '9999-12-31',
    job_code INT,
    store_id INT
)
PARTITION BY HASH( YEAR(hired) )
PARTITIONS 4;
```

`expr` must return a nonconstant, nonrandom integer value (in other words, it should be varying but deterministic), and must not contain any prohibited constructs as described in Section 24.6, “Restrictions and Limitations on Partitioning”. You should also keep in mind that this expression is evaluated each time a row is inserted or updated (or possibly deleted); this means that very complex expressions may give rise to performance issues, particularly when performing operations (such as batch inserts) that affect a great many rows at one time.

The most efficient hashing function is one which operates upon a single table column and whose value increases or decreases consistently with the column value, as this allows for “pruning” on ranges of partitions. That is, the more closely that the expression varies with the value of the column on which it is based, the more efficiently MySQL can use the expression for hash partitioning.

For example, where `date_col` is a column of type `DATE`, then the expression `TO_DAYS(date_col)` is said to vary directly with the value of `date_col`, because for every change in the value of `date_col`, the value of the expression changes in a consistent manner. The variance of the expression `YEAR(date_col)` with respect to `date_col` is not quite as direct as that of `TO_DAYS(date_col)`, because not every possible change in `date_col` produces an equivalent change in `YEAR(date_col)`. Even so, `YEAR(date_col)` is a good candidate for a hashing function, because it varies directly with a portion of `date_col` and there is no possible change in `date_col` that produces a disproportionate change in `YEAR(date_col)`.

By way of contrast, suppose that you have a column named `int_col` whose type is `INT`. Now consider the expression `POW(5-int_col,3) + 6`. This would be a poor choice for a hashing function because a change in the value of `int_col` is not guaranteed to produce a proportional change in the value of the expression. Changing the value of `int_col` by a given amount can produce widely differing changes in the value of the expression. For example, changing `int_col` from `5` to `6` produces a change of `-1` in the value of the expression, but changing the value of `int_col` from `6` to `7` produces a change of `-7` in the expression value.

In other words, the more closely the graph of the column value versus the value of the expression follows a straight line as traced by the equation `y=cx` where *`c`* is some nonzero constant, the better the expression is suited to hashing. This has to do with the fact that the more nonlinear an expression is, the more uneven the distribution of data among the partitions it tends to produce.

In theory, pruning is also possible for expressions involving more than one column value, but determining which of such expressions are suitable can be quite difficult and time-consuming. For this reason, the use of hashing expressions involving multiple columns is not particularly recommended.

When `PARTITION BY HASH` is used, the storage engine determines which partition of *`num`* partitions to use based on the modulus of the result of the expression. In other words, for a given expression *`expr`*, the partition in which the record is stored is partition number *`N`*, where *`N = MOD(expr, num)`*. Suppose that table `t1` is defined as follows, so that it has 4 partitions:

```
CREATE TABLE t1 (col1 INT, col2 CHAR(5), col3 DATE)
    PARTITION BY HASH( YEAR(col3) )
    PARTITIONS 4;
```

If you insert a record into `t1` whose `col3` value is `'2005-09-15'`, then the partition in which it is stored is determined as follows:

```
MOD(YEAR('2005-09-01'),4)
=  MOD(2005,4)
```

```
=  1
```

MySQL 8.0 also supports a variant of HASH partitioning known as *linear hashing* which employs a more complex algorithm for determining the placement of new rows inserted into the partitioned table. See Section 24.2.4.1, “LINEAR HASH Partitioning”, for a description of this algorithm.

The user-supplied expression is evaluated each time a record is inserted or updated. It may also—depending on the circumstances—be evaluated when records are deleted.

24.2.4.1 LINEAR HASH Partitioning

MySQL also supports linear hashing, which differs from regular hashing in that linear hashing utilizes a linear powers-of-two algorithm whereas regular hashing employs the modulus of the hashing function's value.

Syntactically, the only difference between linear-hash partitioning and regular hashing is the addition of the `LINEAR` keyword in the `PARTITION BY` clause, as shown here:

```
CREATE TABLE employees (
    id INT NOT NULL,
    fname VARCHAR(30),
    lname VARCHAR(30),
    hired DATE NOT NULL DEFAULT '1970-01-01',
    separated DATE NOT NULL DEFAULT '9999-12-31',
    job_code INT,
    store_id INT
)
PARTITION BY LINEAR HASH( YEAR(hired) )
PARTITIONS 4;
```

Given an expression *`expr`*, the partition in which the record is stored when linear hashing is used is partition number *`N`* from among *`num`* partitions, where *`N`* is derived according to the following algorithm:

1. Find the next power of 2 greater than *`num`*. We call this value *`V`*; it can be calculated as:

   ```
   V = POWER(2, CEILING(LOG(2, num)))
   ```

   (Suppose that *`num`* is 13. Then `LOG(2,13)` is 3.7004397181411. `CEILING(3.7004397181411)` is 4, and *`V`* = `POWER(2,4)`, which is 16.)

2. Set *`N`* = *`F`*(*`column_list`*) & (*`V`* - 1).

3. While *`N`* >= *`num`*:

   - Set *`V`* = *`V`* / 2
   - Set *`N`* = *`N`* & (*`V`* - 1)

Suppose that the table `t1`, using linear hash partitioning and having 6 partitions, is created using this statement:

```
CREATE TABLE t1 (col1 INT, col2 CHAR(5), col3 DATE)
    PARTITION BY LINEAR HASH( YEAR(col3) )
    PARTITIONS 6;
```

Now assume that you want to insert two records into `t1` having the `col3` column values `'2003-04-14'` and `'1998-10-19'`. The partition number for the first of these is determined as follows:

```
V = POWER(2, CEILING( LOG(2,6) )) = 8
N = YEAR('2003-04-14') & (8 - 1)
   = 2003 & 7
   = 3

(3 >= 6 is FALSE: record stored in partition #3)
```

The number of the partition where the second record is stored is calculated as shown here:

```
V = 8
N = YEAR('1998-10-19') & (8 - 1)
  = 1998 & 7
  = 6

(6 >= 6 is TRUE: additional step required)

N = 6 & ((8 / 2) - 1)
  = 6 & 3
  = 2

(2 >= 6 is FALSE: record stored in partition #2)
```

The advantage in partitioning by linear hash is that the adding, dropping, merging, and splitting of partitions is made much faster, which can be beneficial when dealing with tables containing extremely large amounts (terabytes) of data. The disadvantage is that data is less likely to be evenly distributed between partitions as compared with the distribution obtained using regular hash partitioning.

24.2.5 KEY Partitioning

Partitioning by key is similar to partitioning by hash, except that where hash partitioning employs a user-defined expression, the hashing function for key partitioning is supplied by the MySQL server. NDB Cluster uses `MD5()` for this purpose; for tables using other storage engines, the server employs its own internal hashing function.

The syntax rules for `CREATE TABLE ... PARTITION BY KEY` are similar to those for creating a table that is partitioned by hash. The major differences are listed here:

- `KEY` is used rather than `HASH`.
- `KEY` takes only a list of zero or more column names. Any columns used as the partitioning key must comprise part or all of the table's primary key, if the table has one. Where no column name is specified as the partitioning key, the table's primary key is used, if there is one. For example, the following `CREATE TABLE` statement is valid in MySQL 8.0:

  ```
  CREATE TABLE k1 (
      id INT NOT NULL PRIMARY KEY,
      name VARCHAR(20)
  )
  PARTITION BY KEY()
  PARTITIONS 2;
  ```

  If there is no primary key but there is a unique key, then the unique key is used for the partitioning key:

  ```
  CREATE TABLE k1 (
      id INT NOT NULL,
      name VARCHAR(20),
      UNIQUE KEY (id)
  )
  PARTITION BY KEY()
  PARTITIONS 2;
  ```

  However, if the unique key column were not defined as `NOT NULL`, then the previous statement would fail.

  In both of these cases, the partitioning key is the `id` column, even though it is not shown in the output of `SHOW CREATE TABLE` or in the `PARTITION_EXPRESSION` column of the Information Schema `PARTITIONS` table.

  Unlike the case with other partitioning types, columns used for partitioning by `KEY` are not restricted to integer or `NULL` values. For example, the following `CREATE TABLE` statement is valid:

  ```
  CREATE TABLE tm1 (
      s1 CHAR(32) PRIMARY KEY
  )
  ```

```
PARTITION BY KEY(s1)
PARTITIONS 10;
```

The preceding statement would *not* be valid, were a different partitioning type to be specified. (In this case, simply using `PARTITION BY KEY()` would also be valid and have the same effect as `PARTITION BY KEY(s1)`, since `s1` is the table's primary key.)

For additional information about this issue, see Section 24.6, “Restrictions and Limitations on Partitioning”.

Columns with index prefixes are not supported in partitioning keys. This means that `CHAR`, `VARCHAR`, `BINARY`, and `VARBINARY` columns can be used in a partitioning key, as long as they do not employ prefixes; because a prefix must be specified for `BLOB` and `TEXT` columns in index definitions, it is not possible to use columns of these two types in partitioning keys. Prior to MySQL 8.0.21, columns using prefixes were permitted when creating, altering, or upgrading a partitioned table, even though they were not included in the table's partitioning key; in MySQL 8.0.21 and later, this permissive behavior is deprecated, and the server displays appropriate warnings or errors when one or more such columns are used. See Column index prefixes not supported for key partitioning, for more information and examples.

Note

Tables using the `NDB` storage engine are implicitly partitioned by `KEY`, using the table's primary key as the partitioning key (as with other MySQL storage engines). In the event that the NDB Cluster table has no explicit primary key, the “hidden” primary key generated by the `NDB` storage engine for each NDB Cluster table is used as the partitioning key.

If you define an explicit partitioning scheme for an `NDB` table, the table must have an explicit primary key, and any columns used in the partitioning expression must be part of this key. However, if the table uses an “empty” partitioning expression—that is, `PARTITION BY KEY()` with no column references—then no explicit primary key is required.

You can observe this partitioning using the `ndb_desc` utility (with the `-p` option).

Important

For a key-partitioned table, you cannot execute an `ALTER TABLE DROP PRIMARY KEY`, as doing so generates the error `ERROR 1466 (HY000): Field in list of fields for partition function not found in table`. This is not an issue for NDB Cluster tables which are partitioned by `KEY`; in such cases, the table is reorganized using the “hidden” primary key as the table's new partitioning key. See Chapter 23, *MySQL NDB Cluster 8.0*.

It is also possible to partition a table by linear key. Here is a simple example:

```
CREATE TABLE tk (
    col1 INT NOT NULL,
    col2 CHAR(5),
    col3 DATE
)
PARTITION BY LINEAR KEY (col1)
PARTITIONS 3;
```

The `LINEAR` keyword has the same effect on `KEY` partitioning as it does on `HASH` partitioning, with the partition number being derived using a powers-of-two algorithm rather than modulo arithmetic. See Section 24.2.4.1, “LINEAR HASH Partitioning”, for a description of this algorithm and its implications.

24.2.6 Subpartitioning

Subpartitioning—also known as *composite partitioning*—is the further division of each partition in a partitioned table. Consider the following `CREATE TABLE` statement:

```
CREATE TABLE ts (id INT, purchased DATE)
    PARTITION BY RANGE( YEAR(purchased) )
    SUBPARTITION BY HASH( TO_DAYS(purchased) )
    SUBPARTITIONS 2 (
        PARTITION p0 VALUES LESS THAN (1990),
        PARTITION p1 VALUES LESS THAN (2000),
        PARTITION p2 VALUES LESS THAN MAXVALUE
    );
```

Table `ts` has 3 `RANGE` partitions. Each of these partitions—`p0`, `p1`, and `p2`—is further divided into 2 subpartitions. In effect, the entire table is divided into `3 * 2 = 6` partitions. However, due to the action of the `PARTITION BY RANGE` clause, the first 2 of these store only those records with a value less than 1990 in the `purchased` column.

It is possible to subpartition tables that are partitioned by `RANGE` or `LIST`. Subpartitions may use either `HASH` or `KEY` partitioning. This is also known as *composite partitioning*.

Note

`SUBPARTITION BY HASH` and `SUBPARTITION BY KEY` generally follow the same syntax rules as `PARTITION BY HASH` and `PARTITION BY KEY`, respectively. An exception to this is that `SUBPARTITION BY KEY` (unlike `PARTITION BY KEY`) does not currently support a default column, so the column used for this purpose must be specified, even if the table has an explicit primary key. This is a known issue which we are working to address; see Issues with subpartitions, for more information and an example.

It is also possible to define subpartitions explicitly using `SUBPARTITION` clauses to specify options for individual subpartitions. For example, a more verbose fashion of creating the same table `ts` as shown in the previous example would be:

```
CREATE TABLE ts (id INT, purchased DATE)
    PARTITION BY RANGE( YEAR(purchased) )
    SUBPARTITION BY HASH( TO_DAYS(purchased) ) (
        PARTITION p0 VALUES LESS THAN (1990) (
            SUBPARTITION s0,
            SUBPARTITION s1
        ),
        PARTITION p1 VALUES LESS THAN (2000) (
            SUBPARTITION s2,
            SUBPARTITION s3
        ),
        PARTITION p2 VALUES LESS THAN MAXVALUE (
            SUBPARTITION s4,
            SUBPARTITION s5
        )
    );
```

Some syntactical items of note are listed here:

- Each partition must have the same number of subpartitions.
- If you explicitly define any subpartitions using `SUBPARTITION` on any partition of a partitioned table, you must define them all. In other words, the following statement fails:

```
CREATE TABLE ts (id INT, purchased DATE)
    PARTITION BY RANGE( YEAR(purchased) )
    SUBPARTITION BY HASH( TO_DAYS(purchased) ) (
        PARTITION p0 VALUES LESS THAN (1990) (
            SUBPARTITION s0,
            SUBPARTITION s1
        ),
        PARTITION p1 VALUES LESS THAN (2000),
        PARTITION p2 VALUES LESS THAN MAXVALUE (
```

```
            SUBPARTITION s2,
            SUBPARTITION s3
        )
    );
```

This statement would still fail even if it used `SUBPARTITIONS 2`.

- Each `SUBPARTITION` clause must include (at a minimum) a name for the subpartition. Otherwise, you may set any desired option for the subpartition or allow it to assume its default setting for that option.
- Subpartition names must be unique across the entire table. For example, the following `CREATE TABLE` statement is valid:

```
CREATE TABLE ts (id INT, purchased DATE)
    PARTITION BY RANGE( YEAR(purchased) )
    SUBPARTITION BY HASH( TO_DAYS(purchased) ) (
        PARTITION p0 VALUES LESS THAN (1990) (
            SUBPARTITION s0,
            SUBPARTITION s1
        ),
        PARTITION p1 VALUES LESS THAN (2000) (
            SUBPARTITION s2,
            SUBPARTITION s3
        ),
        PARTITION p2 VALUES LESS THAN MAXVALUE (
            SUBPARTITION s4,
            SUBPARTITION s5
        )
    );
```

24.2.7 How MySQL Partitioning Handles NULL

Partitioning in MySQL does nothing to disallow `NULL` as the value of a partitioning expression, whether it is a column value or the value of a user-supplied expression. Even though it is permitted to use `NULL` as the value of an expression that must otherwise yield an integer, it is important to keep in mind that `NULL` is not a number. MySQL's partitioning implementation treats `NULL` as being less than any non-`NULL` value, just as `ORDER BY` does.

This means that treatment of `NULL` varies between partitioning of different types, and may produce behavior which you do not expect if you are not prepared for it. This being the case, we discuss in this section how each MySQL partitioning type handles `NULL` values when determining the partition in which a row should be stored, and provide examples for each.

Handling of NULL with RANGE partitioning. If you insert a row into a table partitioned by `RANGE` such that the column value used to determine the partition is `NULL`, the row is inserted into the lowest partition. Consider these two tables in a database named `p`, created as follows:

```
mysql> CREATE TABLE t1 (
    ->     c1 INT,
    ->     c2 VARCHAR(20)
    -> )
    -> PARTITION BY RANGE(c1) (
    ->     PARTITION p0 VALUES LESS THAN (0),
    ->     PARTITION p1 VALUES LESS THAN (10),
    ->     PARTITION p2 VALUES LESS THAN MAXVALUE
    -> );
Query OK, 0 rows affected (0.09 sec)

mysql> CREATE TABLE t2 (
    ->     c1 INT,
    ->     c2 VARCHAR(20)
    -> )
    -> PARTITION BY RANGE(c1) (
    ->     PARTITION p0 VALUES LESS THAN (-5),
    ->     PARTITION p1 VALUES LESS THAN (0),
    ->     PARTITION p2 VALUES LESS THAN (10),
    ->     PARTITION p3 VALUES LESS THAN MAXVALUE
```

```
    -> );
Query OK, 0 rows affected (0.09 sec)
```

You can see the partitions created by these two `CREATE TABLE` statements using the following query against the `PARTITIONS` table in the `INFORMATION_SCHEMA` database:

```
mysql> SELECT TABLE_NAME, PARTITION_NAME, TABLE_ROWS, AVG_ROW_LENGTH, DATA_LENGTH
    >   FROM INFORMATION_SCHEMA.PARTITIONS
    >   WHERE TABLE_SCHEMA = 'p' AND TABLE_NAME LIKE 't_';
+------------+----------------+------------+----------------+-------------+
| TABLE_NAME | PARTITION_NAME | TABLE_ROWS | AVG_ROW_LENGTH | DATA_LENGTH |
+------------+----------------+------------+----------------+-------------+
| t1         | p0             |          0 |              0 |           0 |
| t1         | p1             |          0 |              0 |           0 |
| t1         | p2             |          0 |              0 |           0 |
| t2         | p0             |          0 |              0 |           0 |
| t2         | p1             |          0 |              0 |           0 |
| t2         | p2             |          0 |              0 |           0 |
| t2         | p3             |          0 |              0 |           0 |
+------------+----------------+------------+----------------+-------------+
7 rows in set (0.00 sec)
```

(For more information about this table, see Section 26.3.21, “The INFORMATION_SCHEMA PARTITIONS Table”.) Now let us populate each of these tables with a single row containing a `NULL` in the column used as the partitioning key, and verify that the rows were inserted using a pair of `SELECT` statements:

```
mysql> INSERT INTO t1 VALUES (NULL, 'mothra');
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO t2 VALUES (NULL, 'mothra');
Query OK, 1 row affected (0.00 sec)

mysql> SELECT * FROM t1;
+------+--------+
| id   | name   |
+------+--------+
| NULL | mothra |
+------+--------+
1 row in set (0.00 sec)

mysql> SELECT * FROM t2;
+------+--------+
| id   | name   |
+------+--------+
| NULL | mothra |
+------+--------+
1 row in set (0.00 sec)
```

You can see which partitions are used to store the inserted rows by rerunning the previous query against `INFORMATION_SCHEMA.PARTITIONS` and inspecting the output:

```
mysql> SELECT TABLE_NAME, PARTITION_NAME, TABLE_ROWS, AVG_ROW_LENGTH, DATA_LENGTH
    >   FROM INFORMATION_SCHEMA.PARTITIONS
    >   WHERE TABLE_SCHEMA = 'p' AND TABLE_NAME LIKE 't_';
+------------+----------------+------------+----------------+-------------+
| TABLE_NAME | PARTITION_NAME | TABLE_ROWS | AVG_ROW_LENGTH | DATA_LENGTH |
+------------+----------------+------------+----------------+-------------+
| t1         | p0             |          1 |             20 |          20 |
| t1         | p1             |          0 |              0 |           0 |
| t1         | p2             |          0 |              0 |           0 |
| t2         | p0             |          1 |             20 |          20 |
| t2         | p1             |          0 |              0 |           0 |
| t2         | p2             |          0 |              0 |           0 |
| t2         | p3             |          0 |              0 |           0 |
+------------+----------------+------------+----------------+-------------+
7 rows in set (0.01 sec)
```

You can also demonstrate that these rows were stored in the lowest-numbered partition of each table by dropping these partitions, and then re-running the `SELECT` statements:

```
mysql> ALTER TABLE t1 DROP PARTITION p0;
Query OK, 0 rows affected (0.16 sec)

mysql> ALTER TABLE t2 DROP PARTITION p0;
Query OK, 0 rows affected (0.16 sec)

mysql> SELECT * FROM t1;
Empty set (0.00 sec)

mysql> SELECT * FROM t2;
Empty set (0.00 sec)
```

(For more information on `ALTER TABLE ... DROP PARTITION`, see Section 13.1.9, "ALTER TABLE Statement".)

`NULL` is also treated in this way for partitioning expressions that use SQL functions. Suppose that we define a table using a `CREATE TABLE` statement such as this one:

```
CREATE TABLE tndate (
    id INT,
    dt DATE
)
PARTITION BY RANGE( YEAR(dt) ) (
    PARTITION p0 VALUES LESS THAN (1990),
    PARTITION p1 VALUES LESS THAN (2000),
    PARTITION p2 VALUES LESS THAN MAXVALUE
);
```

As with other MySQL functions, `YEAR(NULL)` returns `NULL`. A row with a `dt` column value of `NULL` is treated as though the partitioning expression evaluated to a value less than any other value, and so is inserted into partition `p0`.

Handling of NULL with LIST partitioning. A table that is partitioned by `LIST` admits `NULL` values if and only if one of its partitions is defined using that value-list that contains `NULL`. The converse of this is that a table partitioned by `LIST` which does not explicitly use `NULL` in a value list rejects rows resulting in a `NULL` value for the partitioning expression, as shown in this example:

```
mysql> CREATE TABLE ts1 (
    ->     c1 INT,
    ->     c2 VARCHAR(20)
    -> )
    -> PARTITION BY LIST(c1) (
    ->     PARTITION p0 VALUES IN (0, 3, 6),
    ->     PARTITION p1 VALUES IN (1, 4, 7),
    ->     PARTITION p2 VALUES IN (2, 5, 8)
    -> );
Query OK, 0 rows affected (0.01 sec)

mysql> INSERT INTO ts1 VALUES (9, 'mothra');
ERROR 1504 (HY000): Table has no partition for value 9

mysql> INSERT INTO ts1 VALUES (NULL, 'mothra');
ERROR 1504 (HY000): Table has no partition for value NULL
```

Only rows having a `c1` value between `0` and `8` inclusive can be inserted into `ts1`. `NULL` falls outside this range, just like the number `9`. We can create tables `ts2` and `ts3` having value lists containing `NULL`, as shown here:

```
mysql> CREATE TABLE ts2 (
    ->     c1 INT,
    ->     c2 VARCHAR(20)
    -> )
    -> PARTITION BY LIST(c1) (
    ->     PARTITION p0 VALUES IN (0, 3, 6),
    ->     PARTITION p1 VALUES IN (1, 4, 7),
    ->     PARTITION p2 VALUES IN (2, 5, 8),
    ->     PARTITION p3 VALUES IN (NULL)
    -> );
Query OK, 0 rows affected (0.01 sec)
```

```
mysql> CREATE TABLE ts3 (
    ->     c1 INT,
    ->     c2 VARCHAR(20)
    -> )
    -> PARTITION BY LIST(c1) (
    ->     PARTITION p0 VALUES IN (0, 3, 6),
    ->     PARTITION p1 VALUES IN (1, 4, 7, NULL),
    ->     PARTITION p2 VALUES IN (2, 5, 8)
    -> );
Query OK, 0 rows affected (0.01 sec)
```

When defining value lists for partitioning, you can (and should) treat `NULL` just as you would any other value. For example, both `VALUES IN (NULL)` and `VALUES IN (1, 4, 7, NULL)` are valid, as are `VALUES IN (1, NULL, 4, 7)`, `VALUES IN (NULL, 1, 4, 7)`, and so on. You can insert a row having `NULL` for column `c1` into each of the tables `ts2` and `ts3`:

```
mysql> INSERT INTO ts2 VALUES (NULL, 'mothra');
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO ts3 VALUES (NULL, 'mothra');
Query OK, 1 row affected (0.00 sec)
```

By issuing the appropriate query against `INFORMATION_SCHEMA.PARTITIONS`, you can determine which partitions were used to store the rows just inserted (we assume, as in the previous examples, that the partitioned tables were created in the `p` database):

```
mysql> SELECT TABLE_NAME, PARTITION_NAME, TABLE_ROWS, AVG_ROW_LENGTH, DATA_LENGTH
     >   FROM INFORMATION_SCHEMA.PARTITIONS
     >   WHERE TABLE_SCHEMA = 'p' AND TABLE_NAME LIKE 'ts_';
+------------+----------------+------------+----------------+-------------+
| TABLE_NAME | PARTITION_NAME | TABLE_ROWS | AVG_ROW_LENGTH | DATA_LENGTH |
+------------+----------------+------------+----------------+-------------+
| ts2        | p0             |          0 |              0 |           0 |
| ts2        | p1             |          0 |              0 |           0 |
| ts2        | p2             |          0 |              0 |           0 |
| ts2        | p3             |          1 |             20 |          20 |
| ts3        | p0             |          0 |              0 |           0 |
| ts3        | p1             |          1 |             20 |          20 |
| ts3        | p2             |          0 |              0 |           0 |
+------------+----------------+------------+----------------+-------------+
7 rows in set (0.01 sec)
```

As shown earlier in this section, you can also verify which partitions were used for storing the rows by deleting these partitions and then performing a `SELECT`.

Handling of NULL with HASH and KEY partitioning. `NULL` is handled somewhat differently for tables partitioned by `HASH` or `KEY`. In these cases, any partition expression that yields a `NULL` value is treated as though its return value were zero. We can verify this behavior by examining the effects on the file system of creating a table partitioned by `HASH` and populating it with a record containing appropriate values. Suppose that you have a table `th` (also in the `p` database) created using the following statement:

```
mysql> CREATE TABLE th (
    ->     c1 INT,
    ->     c2 VARCHAR(20)
    -> )
    -> PARTITION BY HASH(c1)
    -> PARTITIONS 2;
Query OK, 0 rows affected (0.00 sec)
```

The partitions belonging to this table can be viewed using the query shown here:

```
mysql> SELECT TABLE_NAME,PARTITION_NAME,TABLE_ROWS,AVG_ROW_LENGTH,DATA_LENGTH
     >   FROM INFORMATION_SCHEMA.PARTITIONS
     >   WHERE TABLE_SCHEMA = 'p' AND TABLE_NAME ='th';
+------------+----------------+------------+----------------+-------------+
| TABLE_NAME | PARTITION_NAME | TABLE_ROWS | AVG_ROW_LENGTH | DATA_LENGTH |
+------------+----------------+------------+----------------+-------------+
```

```
| th         | p0             |          0 |              0 |           0 |
| th         | p1             |          0 |              0 |           0 |
+------------+----------------+------------+----------------+-------------+
2 rows in set (0.00 sec)
```

`TABLE_ROWS` for each partition is 0. Now insert two rows into `th` whose `c1` column values are `NULL` and 0, and verify that these rows were inserted, as shown here:

```
mysql> INSERT INTO th VALUES (NULL, 'mothra'), (0, 'gigan');
Query OK, 1 row affected (0.00 sec)

mysql> SELECT * FROM th;
+------+---------+
| c1   | c2      |
+------+---------+
| NULL | mothra  |
+------+---------+
|    0 | gigan   |
+------+---------+
2 rows in set (0.01 sec)
```

Recall that for any integer *`N`*, the value of `NULL MOD N` is always `NULL`. For tables that are partitioned by `HASH` or `KEY`, this result is treated for determining the correct partition as `0`. Checking the Information Schema `PARTITIONS` table once again, we can see that both rows were inserted into partition `p0`:

```
mysql> SELECT TABLE_NAME, PARTITION_NAME, TABLE_ROWS, AVG_ROW_LENGTH, DATA_LENGTH
    >   FROM INFORMATION_SCHEMA.PARTITIONS
    >   WHERE TABLE_SCHEMA = 'p' AND TABLE_NAME ='th';
+------------+----------------+------------+----------------+-------------+
| TABLE_NAME | PARTITION_NAME | TABLE_ROWS | AVG_ROW_LENGTH | DATA_LENGTH |
+------------+----------------+------------+----------------+-------------+
| th         | p0             |          2 |             20 |          20 |
| th         | p1             |          0 |              0 |           0 |
+------------+----------------+------------+----------------+-------------+
2 rows in set (0.00 sec)
```

By repeating the last example using `PARTITION BY KEY` in place of `PARTITION BY HASH` in the definition of the table, you can verify that `NULL` is also treated like 0 for this type of partitioning.

24.3 Partition Management

There are a number of ways using SQL statements to modify partitioned tables; it is possible to add, drop, redefine, merge, or split existing partitions using the partitioning extensions to the `ALTER TABLE` statement. There are also ways to obtain information about partitioned tables and partitions. We discuss these topics in the sections that follow.

- For information about partition management in tables partitioned by `RANGE` or `LIST`, see Section 24.3.1, “Management of RANGE and LIST Partitions”.
- For a discussion of managing `HASH` and `KEY` partitions, see Section 24.3.2, “Management of HASH and KEY Partitions”.
- See Section 24.3.5, “Obtaining Information About Partitions”, for a discussion of mechanisms provided in MySQL 8.0 for obtaining information about partitioned tables and partitions.
- For a discussion of performing maintenance operations on partitions, see Section 24.3.4, “Maintenance of Partitions”.

Note

All partitions of a partitioned table must have the same number of subpartitions; it is not possible to change the subpartitioning once the table has been created.

To change a table's partitioning scheme, it is necessary only to use the `ALTER TABLE` statement with a *`partition_options`* option, which has the same syntax as that as used with `CREATE TABLE`

for creating a partitioned table; this option (also) always begins with the keywords `PARTITION BY`. Suppose that the following `CREATE TABLE` statement was used to create a table that is partitioned by range:

```
CREATE TABLE trb3 (id INT, name VARCHAR(50), purchased DATE)
    PARTITION BY RANGE( YEAR(purchased) ) (
        PARTITION p0 VALUES LESS THAN (1990),
        PARTITION p1 VALUES LESS THAN (1995),
        PARTITION p2 VALUES LESS THAN (2000),
        PARTITION p3 VALUES LESS THAN (2005)
    );
```

To repartition this table so that it is partitioned by key into two partitions using the `id` column value as the basis for the key, you can use this statement:

```
ALTER TABLE trb3 PARTITION BY KEY(id) PARTITIONS 2;
```

This has the same effect on the structure of the table as dropping the table and re-creating it using `CREATE TABLE trb3 PARTITION BY KEY(id) PARTITIONS 2;`.

`ALTER TABLE ... ENGINE = ...` changes only the storage engine used by the table, and leaves the table's partitioning scheme intact. The statement succeeds only if the target storage engine provides partitioning support. You can use `ALTER TABLE ... REMOVE PARTITIONING` to remove a table's partitioning; see Section 13.1.9, “ALTER TABLE Statement”.

Important

Only a single `PARTITION BY`, `ADD PARTITION`, `DROP PARTITION`, `REORGANIZE PARTITION`, or `COALESCE PARTITION` clause can be used in a given `ALTER TABLE` statement. If you (for example) wish to drop a partition and reorganize a table's remaining partitions, you must do so in two separate `ALTER TABLE` statements (one using `DROP PARTITION` and then a second one using `REORGANIZE PARTITION`).

You can delete all rows from one or more selected partitions using `ALTER TABLE ... TRUNCATE PARTITION`.

24.3.1 Management of RANGE and LIST Partitions

Adding and dropping of range and list partitions are handled in a similar fashion, so we discuss the management of both sorts of partitioning in this section. For information about working with tables that are partitioned by hash or key, see Section 24.3.2, “Management of HASH and KEY Partitions”.

Dropping a partition from a table that is partitioned by either `RANGE` or by `LIST` can be accomplished using the `ALTER TABLE` statement with the `DROP PARTITION` option. Suppose that you have created a table that is partitioned by range and then populated with 10 records using the following `CREATE TABLE` and `INSERT` statements:

```
mysql> CREATE TABLE tr (id INT, name VARCHAR(50), purchased DATE)
    ->     PARTITION BY RANGE( YEAR(purchased) ) (
    ->         PARTITION p0 VALUES LESS THAN (1990),
    ->         PARTITION p1 VALUES LESS THAN (1995),
    ->         PARTITION p2 VALUES LESS THAN (2000),
    ->         PARTITION p3 VALUES LESS THAN (2005),
    ->         PARTITION p4 VALUES LESS THAN (2010),
    ->         PARTITION p5 VALUES LESS THAN (2015)
    ->     );
Query OK, 0 rows affected (0.28 sec)

mysql> INSERT INTO tr VALUES
    ->     (1, 'desk organiser', '2003-10-15'),
    ->     (2, 'alarm clock', '1997-11-05'),
    ->     (3, 'chair', '2009-03-10'),
    ->     (4, 'bookcase', '1989-01-10'),
    ->     (5, 'exercise bike', '2014-05-09'),
    ->     (6, 'sofa', '1987-06-05'),
```

```
    ->     (7, 'espresso maker', '2011-11-22'),
    ->     (8, 'aquarium', '1992-08-04'),
    ->     (9, 'study desk', '2006-09-16'),
    ->     (10, 'lava lamp', '1998-12-25');
Query OK, 10 rows affected (0.05 sec)
Records: 10  Duplicates: 0  Warnings: 0
```

You can see which items should have been inserted into partition `p2` as shown here:

```
mysql> SELECT * FROM tr
    ->     WHERE purchased BETWEEN '1995-01-01' AND '1999-12-31';
+------+------------+------------+
| id   | name       | purchased  |
+------+------------+------------+
|    2 | alarm clock | 1997-11-05 |
|   10 | lava lamp   | 1998-12-25 |
+------+------------+------------+
2 rows in set (0.00 sec)
```

You can also get this information using partition selection, as shown here:

```
mysql> SELECT * FROM tr PARTITION (p2);
+------+------------+------------+
| id   | name       | purchased  |
+------+------------+------------+
|    2 | alarm clock | 1997-11-05 |
|   10 | lava lamp   | 1998-12-25 |
+------+------------+------------+
2 rows in set (0.00 sec)
```

See Section 24.5, “Partition Selection”, for more information.

To drop the partition named `p2`, execute the following command:

```
mysql> ALTER TABLE tr DROP PARTITION p2;
Query OK, 0 rows affected (0.03 sec)
```

Note

The `NDBCLUSTER` storage engine does not support `ALTER TABLE ... DROP PARTITION`. It does, however, support the other partitioning-related extensions to `ALTER TABLE` that are described in this chapter.

It is very important to remember that, *when you drop a partition, you also delete all the data that was stored in that partition*. You can see that this is the case by re-running the previous `SELECT` query:

```
mysql> SELECT * FROM tr WHERE purchased
    -> BETWEEN '1995-01-01' AND '1999-12-31';
Empty set (0.00 sec)
```

Note

`DROP PARTITION` is supported by native partitioning in-place APIs and may be used with `ALGORITHM={COPY|INPLACE}`. `DROP PARTITION` with `ALGORITHM=INPLACE` deletes data stored in the partition and drops the partition. However, `DROP PARTITION` with `ALGORITHM=COPY` or `old_alter_table=ON` rebuilds the partitioned table and attempts to move data from the dropped partition to another partition with a compatible `PARTITION ... VALUES` definition. Data that cannot be moved to another partition is deleted.

Because of this, you must have the `DROP` privilege for a table before you can execute `ALTER TABLE ... DROP PARTITION` on that table.

If you wish to drop all data from all partitions while preserving the table definition and its partitioning scheme, use the `TRUNCATE TABLE` statement. (See Section 13.1.37, “TRUNCATE TABLE Statement”.)

If you intend to change the partitioning of a table *without* losing data, use `ALTER TABLE ... REORGANIZE PARTITION` instead. See below or in Section 13.1.9, “ALTER TABLE Statement”, for information about `REORGANIZE PARTITION`.

If you now execute a `SHOW CREATE TABLE` statement, you can see how the partitioning makeup of the table has been changed:

```
mysql> SHOW CREATE TABLE tr\G
*************************** 1. row ***************************
       Table: tr
Create Table: CREATE TABLE `tr` (
  `id` int(11) DEFAULT NULL,
  `name` varchar(50) DEFAULT NULL,
  `purchased` date DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=latin1
/*!50100 PARTITION BY RANGE ( YEAR(purchased))
(PARTITION p0 VALUES LESS THAN (1990) ENGINE = InnoDB,
 PARTITION p1 VALUES LESS THAN (1995) ENGINE = InnoDB,
 PARTITION p3 VALUES LESS THAN (2005) ENGINE = InnoDB,
 PARTITION p4 VALUES LESS THAN (2010) ENGINE = InnoDB,
 PARTITION p5 VALUES LESS THAN (2015) ENGINE = InnoDB) */
1 row in set (0.00 sec)
```

When you insert new rows into the changed table with `purchased` column values between `'1995-01-01'` and `'2004-12-31'` inclusive, those rows are stored in partition `p3`. You can verify this as follows:

```
mysql> INSERT INTO tr VALUES (11, 'pencil holder', '1995-07-12');
Query OK, 1 row affected (0.00 sec)

mysql> SELECT * FROM tr WHERE purchased
    -> BETWEEN '1995-01-01' AND '2004-12-31';
+------+---------------+------------+
| id   | name          | purchased  |
+------+---------------+------------+
|    1 | desk organiser | 2003-10-15 |
|   11 | pencil holder  | 1995-07-12 |
+------+---------------+------------+
2 rows in set (0.00 sec)

mysql> ALTER TABLE tr DROP PARTITION p3;
Query OK, 0 rows affected (0.03 sec)

mysql> SELECT * FROM tr WHERE purchased
    -> BETWEEN '1995-01-01' AND '2004-12-31';
Empty set (0.00 sec)
```

The number of rows dropped from the table as a result of `ALTER TABLE ... DROP PARTITION` is not reported by the server as it would be by the equivalent `DELETE` query.

Dropping `LIST` partitions uses exactly the same `ALTER TABLE ... DROP PARTITION` syntax as used for dropping `RANGE` partitions. However, there is one important difference in the effect this has on your use of the table afterward: You can no longer insert into the table any rows having any of the values that were included in the value list defining the deleted partition. (See Section 24.2.2, “LIST Partitioning”, for an example.)

To add a new range or list partition to a previously partitioned table, use the `ALTER TABLE ... ADD PARTITION` statement. For tables which are partitioned by `RANGE`, this can be used to add a new range to the end of the list of existing partitions. Suppose that you have a partitioned table containing membership data for your organization, which is defined as follows:

```
CREATE TABLE members (
    id INT,
    fname VARCHAR(25),
    lname VARCHAR(25),
    dob DATE
)
```

```
PARTITION BY RANGE( YEAR(dob) ) (
    PARTITION p0 VALUES LESS THAN (1980),
    PARTITION p1 VALUES LESS THAN (1990),
    PARTITION p2 VALUES LESS THAN (2000)
);
```

Suppose further that the minimum age for members is 16. As the calendar approaches the end of 2015, you realize that you must soon be prepared to admit members who were born in 2000 (and later). You can modify the `members` table to accommodate new members born in the years 2000 to 2010 as shown here:

```
ALTER TABLE members ADD PARTITION (PARTITION p3 VALUES LESS THAN (2010));
```

With tables that are partitioned by range, you can use `ADD PARTITION` to add new partitions to the high end of the partitions list only. Trying to add a new partition in this manner between or before existing partitions results in an error as shown here:

```
mysql> ALTER TABLE members
     >     ADD PARTITION (
     >     PARTITION n VALUES LESS THAN (1970));
ERROR 1463 (HY000): VALUES LESS THAN value must be strictly »
   increasing for each partition
```

You can work around this problem by reorganizing the first partition into two new ones that split the range between them, like this:

```
ALTER TABLE members
    REORGANIZE PARTITION p0 INTO (
        PARTITION n0 VALUES LESS THAN (1970),
        PARTITION n1 VALUES LESS THAN (1980)
);
```

Using `SHOW CREATE TABLE` you can see that the `ALTER TABLE` statement has had the desired effect:

```
mysql> SHOW CREATE TABLE members\G
*************************** 1. row ***************************
       Table: members
Create Table: CREATE TABLE `members` (
  `id` int(11) DEFAULT NULL,
  `fname` varchar(25) DEFAULT NULL,
  `lname` varchar(25) DEFAULT NULL,
  `dob` date DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=latin1
/*!50100 PARTITION BY RANGE ( YEAR(dob))
(PARTITION n0 VALUES LESS THAN (1970) ENGINE = InnoDB,
 PARTITION n1 VALUES LESS THAN (1980) ENGINE = InnoDB,
 PARTITION p1 VALUES LESS THAN (1990) ENGINE = InnoDB,
 PARTITION p2 VALUES LESS THAN (2000) ENGINE = InnoDB,
 PARTITION p3 VALUES LESS THAN (2010) ENGINE = InnoDB) */
1 row in set (0.00 sec)
```

See also Section 13.1.9.1, “ALTER TABLE Partition Operations”.

You can also use `ALTER TABLE ... ADD PARTITION` to add new partitions to a table that is partitioned by `LIST`. Suppose a table `tt` is defined using the following `CREATE TABLE` statement:

```
CREATE TABLE tt (
    id INT,
    data INT
)
PARTITION BY LIST(data) (
    PARTITION p0 VALUES IN (5, 10, 15),
    PARTITION p1 VALUES IN (6, 12, 18)
);
```

You can add a new partition in which to store rows having the `data` column values `7`, `14`, and `21` as shown:

```
ALTER TABLE tt ADD PARTITION (PARTITION p2 VALUES IN (7, 14, 21));
```

Keep in mind that you *cannot* add a new `LIST` partition encompassing any values that are already included in the value list of an existing partition. If you attempt to do so, an error results:

```
mysql> ALTER TABLE tt ADD PARTITION
    >     (PARTITION np VALUES IN (4, 8, 12));
ERROR 1465 (HY000): Multiple definition of same constant »
                    in list partitioning
```

Because any rows with the `data` column value `12` have already been assigned to partition `p1`, you cannot create a new partition on table `tt` that includes `12` in its value list. To accomplish this, you could drop `p1`, and add `np` and then a new `p1` with a modified definition. However, as discussed earlier, this would result in the loss of all data stored in `p1`—and it is often the case that this is not what you really want to do. Another solution might appear to be to make a copy of the table with the new partitioning and to copy the data into it using `CREATE TABLE ... SELECT ...`, then drop the old table and rename the new one, but this could be very time-consuming when dealing with a large amounts of data. This also might not be feasible in situations where high availability is a requirement.

You can add multiple partitions in a single `ALTER TABLE ... ADD PARTITION` statement as shown here:

```
CREATE TABLE employees (
  id INT NOT NULL,
  fname VARCHAR(50) NOT NULL,
  lname VARCHAR(50) NOT NULL,
  hired DATE NOT NULL
)
PARTITION BY RANGE( YEAR(hired) ) (
  PARTITION p1 VALUES LESS THAN (1991),
  PARTITION p2 VALUES LESS THAN (1996),
  PARTITION p3 VALUES LESS THAN (2001),
  PARTITION p4 VALUES LESS THAN (2005)
);

ALTER TABLE employees ADD PARTITION (
    PARTITION p5 VALUES LESS THAN (2010),
    PARTITION p6 VALUES LESS THAN MAXVALUE
);
```

Fortunately, MySQL's partitioning implementation provides ways to redefine partitions without losing data. Let us look first at a couple of simple examples involving `RANGE` partitioning. Recall the `members` table which is now defined as shown here:

```
mysql> SHOW CREATE TABLE members\G
*************************** 1. row ***************************
       Table: members
Create Table: CREATE TABLE `members` (
  `id` int(11) DEFAULT NULL,
  `fname` varchar(25) DEFAULT NULL,
  `lname` varchar(25) DEFAULT NULL,
  `dob` date DEFAULT NULL
) ENGINE=InnoDB DEFAULT CHARSET=latin1
/*!50100 PARTITION BY RANGE ( YEAR(dob))
(PARTITION n0 VALUES LESS THAN (1970) ENGINE = InnoDB,
 PARTITION n1 VALUES LESS THAN (1980) ENGINE = InnoDB,
 PARTITION p1 VALUES LESS THAN (1990) ENGINE = InnoDB,
 PARTITION p2 VALUES LESS THAN (2000) ENGINE = InnoDB,
 PARTITION p3 VALUES LESS THAN (2010) ENGINE = InnoDB) */
1 row in set (0.00 sec)
```

Suppose that you would like to move all rows representing members born before 1960 into a separate partition. As we have already seen, this cannot be done using `ALTER TABLE ... ADD PARTITION`. However, you can use another partition-related extension to `ALTER TABLE` to accomplish this:

```
ALTER TABLE members REORGANIZE PARTITION n0 INTO (
    PARTITION s0 VALUES LESS THAN (1960),
    PARTITION s1 VALUES LESS THAN (1970)
```

```
);
```

In effect, this command splits partition `p0` into two new partitions `s0` and `s1`. It also moves the data that was stored in `p0` into the new partitions according to the rules embodied in the two `PARTITION ... VALUES ...` clauses, so that `s0` contains only those records for which `YEAR(dob)` is less than 1960 and `s1` contains those rows in which `YEAR(dob)` is greater than or equal to 1960 but less than 1970.

A `REORGANIZE PARTITION` clause may also be used for merging adjacent partitions. You can reverse the effect of the previous statement on the `members` table as shown here:

```
ALTER TABLE members REORGANIZE PARTITION s0,s1 INTO (
    PARTITION p0 VALUES LESS THAN (1970)
);
```

No data is lost in splitting or merging partitions using `REORGANIZE PARTITION`. In executing the above statement, MySQL moves all of the records that were stored in partitions `s0` and `s1` into partition `p0`.

The general syntax for `REORGANIZE PARTITION` is shown here:

```
ALTER TABLE tbl_name
    REORGANIZE PARTITION partition_list
    INTO (partition_definitions);
```

Here, *`tbl_name`* is the name of the partitioned table, and *`partition_list`* is a comma-separated list of names of one or more existing partitions to be changed. *`partition_definitions`* is a comma-separated list of new partition definitions, which follow the same rules as for the *`partition_definitions`* list used in `CREATE TABLE`. You are not limited to merging several partitions into one, or to splitting one partition into many, when using `REORGANIZE PARTITION`. For example, you can reorganize all four partitions of the `members` table into two, like this:

```
ALTER TABLE members REORGANIZE PARTITION p0,p1,p2,p3 INTO (
    PARTITION m0 VALUES LESS THAN (1980),
    PARTITION m1 VALUES LESS THAN (2000)
);
```

You can also use `REORGANIZE PARTITION` with tables that are partitioned by `LIST`. Let us return to the problem of adding a new partition to the list-partitioned `tt` table and failing because the new partition had a value that was already present in the value-list of one of the existing partitions. We can handle this by adding a partition that contains only nonconflicting values, and then reorganizing the new partition and the existing one so that the value which was stored in the existing one is now moved to the new one:

```
ALTER TABLE tt ADD PARTITION (PARTITION np VALUES IN (4, 8));
ALTER TABLE tt REORGANIZE PARTITION p1,np INTO (
    PARTITION p1 VALUES IN (6, 18),
    PARTITION np VALUES in (4, 8, 12)
);
```

Here are some key points to keep in mind when using `ALTER TABLE ... REORGANIZE PARTITION` to repartition tables that are partitioned by `RANGE` or `LIST`:

- The `PARTITION` options used to determine the new partitioning scheme are subject to the same rules as those used with a `CREATE TABLE` statement.

  A new `RANGE` partitioning scheme cannot have any overlapping ranges; a new `LIST` partitioning scheme cannot have any overlapping sets of values.

- The combination of partitions in the *`partition_definitions`* list should account for the same range or set of values overall as the combined partitions named in the *`partition_list`*.

  For example, partitions `p1` and `p2` together cover the years 1980 through 1999 in the `members` table used as an example in this section. Any reorganization of these two partitions should cover the same range of years overall.

- For tables partitioned by RANGE, you can reorganize only adjacent partitions; you cannot skip range partitions.

  For instance, you could not reorganize the example members table using a statement beginning with ALTER TABLE members REORGANIZE PARTITION p0,p2 INTO ... because p0 covers the years prior to 1970 and p2 the years from 1990 through 1999 inclusive, so these are not adjacent partitions. (You cannot skip partition p1 in this case.)

- You cannot use REORGANIZE PARTITION to change the type of partitioning used by the table (for example, you cannot change RANGE partitions to HASH partitions or the reverse). You also cannot use this statement to change the partitioning expression or column. To accomplish either of these tasks without dropping and re-creating the table, you can use ALTER TABLE ... PARTITION BY ..., as shown here:

```
ALTER TABLE members
    PARTITION BY HASH( YEAR(dob) )
    PARTITIONS 8;
```

24.3.2 Management of HASH and KEY Partitions

Tables which are partitioned by hash or by key are very similar to one another with regard to making changes in a partitioning setup, and both differ in a number of ways from tables which have been partitioned by range or list. For that reason, this section addresses the modification of tables partitioned by hash or by key only. For a discussion of adding and dropping of partitions of tables that are partitioned by range or list, see Section 24.3.1, “Management of RANGE and LIST Partitions”.

You cannot drop partitions from tables that are partitioned by HASH or KEY in the same way that you can from tables that are partitioned by RANGE or LIST. However, you can merge HASH or KEY partitions using ALTER TABLE ... COALESCE PARTITION. Suppose that a clients table containing data about clients is divided into 12 partitions, created as shown here:

```
CREATE TABLE clients (
    id INT,
    fname VARCHAR(30),
    lname VARCHAR(30),
    signed DATE
)
PARTITION BY HASH( MONTH(signed) )
PARTITIONS 12;
```

To reduce the number of partitions from 12 to 8, execute the following ALTER TABLE statement:

```
mysql> ALTER TABLE clients COALESCE PARTITION 4;
Query OK, 0 rows affected (0.02 sec)
```

COALESCE works equally well with tables that are partitioned by HASH, KEY, LINEAR HASH, or LINEAR KEY. Here is an example similar to the previous one, differing only in that the table is partitioned by LINEAR KEY:

```
mysql> CREATE TABLE clients_lk (
    ->     id INT,
    ->     fname VARCHAR(30),
    ->     lname VARCHAR(30),
    ->     signed DATE
    -> )
    -> PARTITION BY LINEAR KEY(signed)
    -> PARTITIONS 12;
Query OK, 0 rows affected (0.03 sec)

mysql> ALTER TABLE clients_lk COALESCE PARTITION 4;
Query OK, 0 rows affected (0.06 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

The number following COALESCE PARTITION is the number of partitions to merge into the remainder —in other words, it is the number of partitions to remove from the table.

Attempting to remove more partitions than are in the table results in an error like this one:

```
mysql> ALTER TABLE clients COALESCE PARTITION 18;
ERROR 1478 (HY000): Cannot remove all partitions, use DROP TABLE instead
```

To increase the number of partitions for the `clients` table from 12 to 18, use `ALTER TABLE ... ADD PARTITION` as shown here:

```
ALTER TABLE clients ADD PARTITION PARTITIONS 6;
```

24.3.3 Exchanging Partitions and Subpartitions with Tables

In MySQL 8.0, it is possible to exchange a table partition or subpartition with a table using `ALTER TABLE pt EXCHANGE PARTITION p WITH TABLE nt`, where *`pt`* is the partitioned table and *`p`* is the partition or subpartition of *`pt`* to be exchanged with unpartitioned table *`nt`*, provided that the following statements are true:

1. Table *`nt`* is not itself partitioned.
2. Table *`nt`* is not a temporary table.
3. The structures of tables *`pt`* and *`nt`* are otherwise identical.
4. Table `nt` contains no foreign key references, and no other table has any foreign keys that refer to `nt`.
5. There are no rows in *`nt`* that lie outside the boundaries of the partition definition for *`p`*. This condition does not apply if `WITHOUT VALIDATION` is used.
6. Both tables must use the same character set and collation.
7. For `InnoDB` tables, both tables must use the same row format. To determine the row format of an `InnoDB` table, query `INFORMATION_SCHEMA.INNODB_TABLES`.
8. Any partition-level `MAX_ROWS` setting for `p` must be the same as the table-level `MAX_ROWS` value set for `nt`. The setting for any partition-level `MIN_ROWS` setting for `p` must also be the same any table-level `MIN_ROWS` value set for `nt`.

   This is true in either case whether not `pt` has an explicit table-level `MAX_ROWS` or `MIN_ROWS` option in effect.
9. The `AVG_ROW_LENGTH` cannot differ between the two tables `pt` and `nt`.
10. `pt` does not have any partitions that use the `DATA DIRECTORY` option. This restriction is lifted for `InnoDB` tables in MySQL 8.0.14 and later.
11. `INDEX DIRECTORY` cannot differ between the table and the partition to be exchanged with it.
12. No table or partition `TABLESPACE` options can be used in either of the tables.

In addition to the `ALTER`, `INSERT`, and `CREATE` privileges usually required for `ALTER TABLE` statements, you must have the `DROP` privilege to perform `ALTER TABLE ... EXCHANGE PARTITION`.

You should also be aware of the following effects of `ALTER TABLE ... EXCHANGE PARTITION`:

- Executing `ALTER TABLE ... EXCHANGE PARTITION` does not invoke any triggers on either the partitioned table or the table to be exchanged.
- Any `AUTO_INCREMENT` columns in the exchanged table are reset.
- The `IGNORE` keyword has no effect when used with `ALTER TABLE ... EXCHANGE PARTITION`.

The syntax for `ALTER TABLE ... EXCHANGE PARTITION` is shown here, where *`pt`* is the partitioned table, *`p`* is the partition (or subpartition) to be exchanged, and *`nt`* is the nonpartitioned table to be exchanged with *`p`*:

```
ALTER TABLE pt
    EXCHANGE PARTITION p
    WITH TABLE nt;
```

Optionally, you can append `WITH VALIDATION` or `WITHOUT VALIDATION`. When `WITHOUT VALIDATION` is specified, the `ALTER TABLE ... EXCHANGE PARTITION` operation does not perform any row-by-row validation when exchanging a partition a nonpartitioned table, allowing database administrators to assume responsibility for ensuring that rows are within the boundaries of the partition definition. `WITH VALIDATION` is the default.

One and only one partition or subpartition may be exchanged with one and only one nonpartitioned table in a single `ALTER TABLE EXCHANGE PARTITION` statement. To exchange multiple partitions or subpartitions, use multiple `ALTER TABLE EXCHANGE PARTITION` statements. `EXCHANGE PARTITION` may not be combined with other `ALTER TABLE` options. The partitioning and (if applicable) subpartitioning used by the partitioned table may be of any type or types supported in MySQL 8.0.

Exchanging a Partition with a Nonpartitioned Table

Suppose that a partitioned table `e` has been created and populated using the following SQL statements:

```
CREATE TABLE e (
    id INT NOT NULL,
    fname VARCHAR(30),
    lname VARCHAR(30)
)
    PARTITION BY RANGE (id) (
        PARTITION p0 VALUES LESS THAN (50),
        PARTITION p1 VALUES LESS THAN (100),
        PARTITION p2 VALUES LESS THAN (150),
        PARTITION p3 VALUES LESS THAN (MAXVALUE)
);

INSERT INTO e VALUES
    (1669, "Jim", "Smith"),
    (337, "Mary", "Jones"),
    (16, "Frank", "White"),
    (2005, "Linda", "Black");
```

Now we create a nonpartitioned copy of `e` named `e2`. This can be done using the `mysql` client as shown here:

```
mysql> CREATE TABLE e2 LIKE e;
Query OK, 0 rows affected (0.04 sec)

mysql> ALTER TABLE e2 REMOVE PARTITIONING;
Query OK, 0 rows affected (0.07 sec)
Records: 0  Duplicates: 0  Warnings: 0
```

You can see which partitions in table `e` contain rows by querying the Information Schema `PARTITIONS` table, like this:

```
mysql> SELECT PARTITION_NAME, TABLE_ROWS
           FROM INFORMATION_SCHEMA.PARTITIONS
           WHERE TABLE_NAME = 'e';
+----------------+------------+
| PARTITION_NAME | TABLE_ROWS |
+----------------+------------+
| p0             |          1 |
| p1             |          0 |
| p2             |          0 |
| p3             |          3 |
```

```
+---------------+------------+
2 rows in set (0.00 sec)
```

Note

For partitioned `InnoDB` tables, the row count given in the `TABLE_ROWS` column of the Information Schema `PARTITIONS` table is only an estimated value used in SQL optimization, and is not always exact.

To exchange partition `p0` in table `e` with table `e2`, you can use `ALTER TABLE`, as shown here:

```
mysql> ALTER TABLE e EXCHANGE PARTITION p0 WITH TABLE e2;
Query OK, 0 rows affected (0.04 sec)
```

More precisely, the statement just issued causes any rows found in the partition to be swapped with those found in the table. You can observe how this has happened by querying the Information Schema `PARTITIONS` table, as before. The table row that was previously found in partition `p0` is no longer present:

```
mysql> SELECT PARTITION_NAME, TABLE_ROWS
           FROM INFORMATION_SCHEMA.PARTITIONS
           WHERE TABLE_NAME = 'e';
+----------------+------------+
| PARTITION_NAME | TABLE_ROWS |
+----------------+------------+
| p0             |          0 |
| p1             |          0 |
| p2             |          0 |
| p3             |          3 |
+----------------+------------+
4 rows in set (0.00 sec)
```

If you query table `e2`, you can see that the “missing” row can now be found there:

```
mysql> SELECT * FROM e2;
+----+-------+-------+
| id | fname | lname |
+----+-------+-------+
| 16 | Frank | White |
+----+-------+-------+
1 row in set (0.00 sec)
```

The table to be exchanged with the partition does not necessarily have to be empty. To demonstrate this, we first insert a new row into table `e`, making sure that this row is stored in partition `p0` by choosing an `id` column value that is less than 50, and verifying this afterward by querying the `PARTITIONS` table:

```
mysql> INSERT INTO e VALUES (41, "Michael", "Green");
Query OK, 1 row affected (0.05 sec)

mysql> SELECT PARTITION_NAME, TABLE_ROWS
           FROM INFORMATION_SCHEMA.PARTITIONS
           WHERE TABLE_NAME = 'e';
+----------------+------------+
| PARTITION_NAME | TABLE_ROWS |
+----------------+------------+
| p0             |          1 |
| p1             |          0 |
| p2             |          0 |
| p3             |          3 |
+----------------+------------+
4 rows in set (0.00 sec)
```

Now we once again exchange partition `p0` with table `e2` using the same `ALTER TABLE` statement as previously:

```
mysql> ALTER TABLE e EXCHANGE PARTITION p0 WITH TABLE e2;
Query OK, 0 rows affected (0.28 sec)
```

The output of the following queries shows that the table row that was stored in partition `p0` and the table row that was stored in table `e2`, prior to issuing the `ALTER TABLE` statement, have now switched places:

```
mysql> SELECT * FROM e;
+------+-------+-------+
| id   | fname | lname |
+------+-------+-------+
|   16 | Frank | White |
| 1669 | Jim   | Smith |
|  337 | Mary  | Jones |
| 2005 | Linda | Black |
+------+-------+-------+
4 rows in set (0.00 sec)

mysql> SELECT PARTITION_NAME, TABLE_ROWS
           FROM INFORMATION_SCHEMA.PARTITIONS
           WHERE TABLE_NAME = 'e';
+----------------+------------+
| PARTITION_NAME | TABLE_ROWS |
+----------------+------------+
| p0             |          1 |
| p1             |          0 |
| p2             |          0 |
| p3             |          3 |
+----------------+------------+
4 rows in set (0.00 sec)

mysql> SELECT * FROM e2;
+----+---------+-------+
| id | fname   | lname |
+----+---------+-------+
| 41 | Michael | Green |
+----+---------+-------+
1 row in set (0.00 sec)
```

Nonmatching Rows

You should keep in mind that any rows found in the nonpartitioned table prior to issuing the `ALTER TABLE ... EXCHANGE PARTITION` statement must meet the conditions required for them to be stored in the target partition; otherwise, the statement fails. To see how this occurs, first insert a row into `e2` that is outside the boundaries of the partition definition for partition `p0` of table `e`. For example, insert a row with an `id` column value that is too large; then, try to exchange the table with the partition again:

```
mysql> INSERT INTO e2 VALUES (51, "Ellen", "McDonald");
Query OK, 1 row affected (0.08 sec)

mysql> ALTER TABLE e EXCHANGE PARTITION p0 WITH TABLE e2;
ERROR 1707 (HY000): Found row that does not match the partition
```

Only the `WITHOUT VALIDATION` option would permit this operation to succeed:

```
mysql> ALTER TABLE e EXCHANGE PARTITION p0 WITH TABLE e2 WITHOUT VALIDATION;
Query OK, 0 rows affected (0.02 sec)
```

When a partition is exchanged with a table that contains rows that do not match the partition definition, it is the responsibility of the database administrator to fix the non-matching rows, which can be performed using `REPAIR TABLE` or `ALTER TABLE ... REPAIR PARTITION`.

Exchanging Partitions Without Row-By-Row Validation

To avoid time consuming validation when exchanging a partition with a table that has many rows, it is possible to skip the row-by-row validation step by appending `WITHOUT VALIDATION` to the `ALTER TABLE ... EXCHANGE PARTITION` statement.

The following example compares the difference between execution times when exchanging a partition with a nonpartitioned table, with and without validation. The partitioned table (table `e`) contains two

partitions of 1 million rows each. The rows in p0 of table e are removed and p0 is exchanged with a nonpartitioned table of 1 million rows. The `WITH VALIDATION` operation takes 0.74 seconds. By comparison, the `WITHOUT VALIDATION` operation takes 0.01 seconds.

```
# Create a partitioned table with 1 million rows in each partition

CREATE TABLE e (
    id INT NOT NULL,
    fname VARCHAR(30),
    lname VARCHAR(30)
)
    PARTITION BY RANGE (id) (
        PARTITION p0 VALUES LESS THAN (1000001),
        PARTITION p1 VALUES LESS THAN (2000001),
);

mysql> SELECT COUNT(*) FROM e;
| COUNT(*) |
+----------+
|  2000000 |
+----------+
1 row in set (0.27 sec)

# View the rows in each partition

SELECT PARTITION_NAME, TABLE_ROWS FROM INFORMATION_SCHEMA.PARTITIONS WHERE TABLE_NAME = 'e';
+----------------+-------------+
| PARTITION_NAME | TABLE_ROWS  |
+----------------+-------------+
| p0             |     1000000 |
| p1             |     1000000 |
+----------------+-------------+
2 rows in set (0.00 sec)

# Create a nonpartitioned table of the same structure and populate it with 1 million rows

CREATE TABLE e2 (
    id INT NOT NULL,
    fname VARCHAR(30),
    lname VARCHAR(30)
);

mysql> SELECT COUNT(*) FROM e2;
+----------+
| COUNT(*) |
+----------+
|  1000000 |
+----------+
1 row in set (0.24 sec)

# Create another nonpartitioned table of the same structure and populate it with 1 million rows

CREATE TABLE e3 (
    id INT NOT NULL,
    fname VARCHAR(30),
    lname VARCHAR(30)
);

mysql> SELECT COUNT(*) FROM e3;
+----------+
| COUNT(*) |
+----------+
|  1000000 |
+----------+
1 row in set (0.25 sec)

# Drop the rows from p0 of table e

mysql> DELETE FROM e WHERE id < 1000001;
Query OK, 1000000 rows affected (5.55 sec)

# Confirm that there are no rows in partition p0
```