- `Mysqlx_cursor_close`

  The number of cursor-close messages received

- `Mysqlx_cursor_fetch`

  The number of cursor-fetch messages received

- `Mysqlx_cursor_open`

  The number of cursor-open messages received

- `Mysqlx_errors_sent`

  The number of errors sent to clients.

- `Mysqlx_errors_unknown_message_type`

  The number of unknown message types that have been received.

- `Mysqlx_expect_close`

  The number of expectation blocks closed.

- `Mysqlx_expect_open`

  The number of expectation blocks opened.

- `Mysqlx_init_error`

  The number of errors during initialisation.

- `Mysqlx_messages_sent`

  The total number of messages of all types sent to clients.

- `Mysqlx_notice_global_sent`

  The number of global notifications sent to clients.

- `Mysqlx_notice_other_sent`

  The number of other types of notices sent back to clients.

- `Mysqlx_notice_warning_sent`

  The number of warning notices sent back to clients.

- `Mysqlx_notified_by_group_replication`

  Number of Group Replication notifications sent to clients.

- `Mysqlx_port`

  The TCP port which X Plugin is listening to. If a network bind has failed, or if the `skip_networking` system variable is enabled, the value shows `UNDEFINED`.

- `Mysqlx_prep_deallocate`

  The number of prepared-statement-deallocate messages received

- `Mysqlx_prep_execute`

  The number of prepared-statement-execute messages received

- `Mysqlx_prep_prepare`

  The number of prepared-statement messages received

- `Mysqlx_rows_sent`

  The number of rows sent back to clients.

- `Mysqlx_sessions`

  The number of sessions that have been opened.

- `Mysqlx_sessions_accepted`

  The number of session attempts which have been accepted.

- `Mysqlx_sessions_closed`

  The number of sessions that have been closed.

- `Mysqlx_sessions_fatal_error`

  The number of sessions that have closed with a fatal error.

- `Mysqlx_sessions_killed`

  The number of sessions which have been killed.

- `Mysqlx_sessions_rejected`

  The number of session attempts which have been rejected.

- `Mysqlx_socket`

  The Unix socket which X Plugin is listening to.

- `Mysqlx_ssl_accept_renegotiates`

  The number of negotiations needed to establish the connection.

- `Mysqlx_ssl_accepts`

  The number of accepted SSL connections.

- `Mysqlx_ssl_active`

  If SSL is active.

- `Mysqlx_ssl_cipher`

  The current SSL cipher (empty for non-SSL connections).

- `Mysqlx_ssl_cipher_list`

  A list of possible SSL ciphers (empty for non-SSL connections).

- `Mysqlx_ssl_ctx_verify_depth`

  The certificate verification depth limit currently set in ctx.

- `Mysqlx_ssl_ctx_verify_mode`

  The certificate verification mode currently set in ctx.

- `Mysqlx_ssl_finished_accepts`

The number of successful SSL connections to the server.

- `Mysqlx_ssl_server_not_after`

  The last date for which the SSL certificate is valid.

- `Mysqlx_ssl_server_not_before`

  The first date for which the SSL certificate is valid.

- `Mysqlx_ssl_verify_depth`

  The certificate verification depth for SSL connections.

- `Mysqlx_ssl_verify_mode`

  The certificate verification mode for SSL connections.

- `Mysqlx_ssl_version`

  The name of the protocol used for SSL connections.

- `Mysqlx_stmt_create_collection`

  The number of create collection statements received.

- `Mysqlx_stmt_create_collection_index`

  The number of create collection index statements received.

- `Mysqlx_stmt_disable_notices`

  The number of disable notice statements received.

- `Mysqlx_stmt_drop_collection`

  The number of drop collection statements received.

- `Mysqlx_stmt_drop_collection_index`

  The number of drop collection index statements received.

- `Mysqlx_stmt_enable_notices`

  The number of enable notice statements received.

- `Mysqlx_stmt_ensure_collection`

  The number of ensure collection statements received.

- `Mysqlx_stmt_execute_mysqlx`

  The number of StmtExecute messages received with namespace set to `mysqlx`.

- `Mysqlx_stmt_execute_sql`

  The number of StmtExecute requests received for the SQL namespace.

- `Mysqlx_stmt_execute_xplugin`

  The number of StmtExecute requests received for the `xplugin` namespace. From MySQL 8.0.19, the `xplugin` namespace has been removed so this status variable is no longer used.

- `Mysqlx_stmt_get_collection_options`

The number of get collection object statements received.

- `Mysqlx_stmt_kill_client`

  The number of kill client statements received.

- `Mysqlx_stmt_list_clients`

  The number of list client statements received.

- `Mysqlx_stmt_list_notices`

  The number of list notice statements received.

- `Mysqlx_stmt_list_objects`

  The number of list object statements received.

- `Mysqlx_stmt_modify_collection_options`

  The number of modify collection options statements received.

- `Mysqlx_stmt_ping`

  The number of ping statements received.

- `Mysqlx_worker_threads`

  The number of worker threads available.

- `Mysqlx_worker_threads_active`

  The number of worker threads currently used.

20.5.7 Monitoring X Plugin

For general X Plugin monitoring, use the status variables that it exposes. See Section 20.5.6.3, “X Plugin Status Variables”. For information specifically about monitoring the effects of message compression, see Monitoring Connection Compression for X Plugin.

Monitoring SQL Generated by X Plugin

This section describes how to monitor the SQL statements which X Plugin generates when you run X DevAPI operations. When you execute a CRUD statement, it is translated into SQL and executed against the server. To be able to monitor the generated SQL, the Performance Schema tables must be enabled. The SQL is registered under the `performance_schema.events_statements_current`, `performance_schema.events_statements_history`, and `performance_schema.events_statements_history_long` tables. The following example uses the `world_x` schema, imported as part of the quickstart tutorials in this section. We use MySQL Shell in Python mode, and the `\sql` command which enables you to issue SQL statements without changing to SQL mode. This is important, because if you instead try to switch to SQL mode, the procedure shows the result of this operation rather than the X DevAPI operation. The `\sql` command is used in the same way if you are using MySQL Shell in JavaScript mode.

1. Check if the `events_statements_history` consumer is enabled. Issue:

```
mysql-py> \sql SELECT enabled FROM performance_schema.setup_consumers WHERE NAME = 'events_statements_h
+---------+
| enabled |
+---------+
| YES     |
+---------+
```

2. Check if all instruments report data to the consumer. Issue:

```
mysql-py> \sql SELECT NAME, ENABLED, TIMED FROM performance_schema.setup_instruments WHERE NAME LIKE
```

   If this statement reports at least one row, you need to enable the instruments. See Section 27.4, “Performance Schema Runtime Configuration”.

3. Get the thread ID of the current connection. Issue:

```
mysql-py> \sql SELECT thread_id INTO @id FROM performance_schema.threads WHERE processlist_id=connec
```

4. Execute the X DevAPI CRUD operation for which you want to see the generated SQL. For example, issue:

```
mysql-py> db.CountryInfo.find("Name = :country").bind("country", "Italy")
```

   You must not issue any further operations for the next step to show the correct result.

5. Show the last SQL query made by this thread ID. Issue:

```
mysql-py> \sql SELECT THREAD_ID, MYSQL_ERRNO,SQL_TEXT FROM performance_schema.events_statements_hist
+-----------+-------------+-------------------------------------------------------------------------
| THREAD_ID | MYSQL_ERRNO | SQL_TEXT
+-----------+-------------+-------------------------------------------------------------------------
|        29 |           0 | SELECT doc FROM `world_x`.`CountryInfo` WHERE (JSON_EXTRACT(doc,'$.Name'
+-----------+-------------+-------------------------------------------------------------------------
```

   The result shows the SQL generated by X Plugin based on the most recent statement, in this case the X DevAPI CRUD operation from the previous step.

Chapter 21 InnoDB Cluster

This chapter introduces MySQL InnoDB Cluster, which combines MySQL technologies to enable you to deploy and administer a complete integrated high availability solution for MySQL. This content is a high-level overview of InnoDB Cluster, for full documentation, see MySQL InnoDB Cluster.

Important

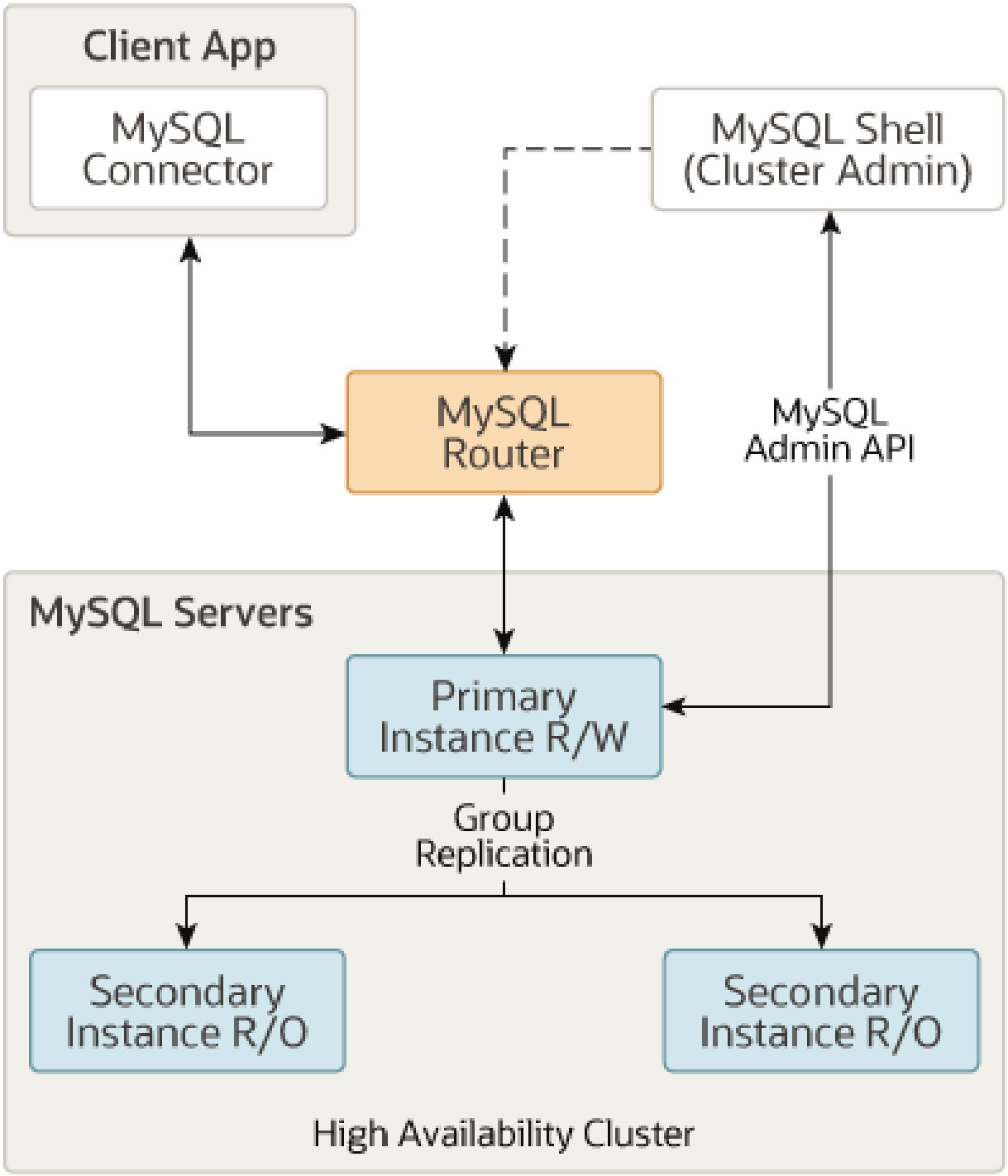
InnoDB Cluster does not provide support for MySQL NDB Cluster. For more information about MySQL NDB Cluster, see Chapter 23, *MySQL NDB Cluster 8.0* and Section 23.2.6, “MySQL Server Using InnoDB Compared with NDB Cluster”.

An InnoDB Cluster consists of at least three MySQL Server instances, and it provides high-availability and scaling features. InnoDB Cluster uses the following MySQL technologies:

- MySQL Shell, which is an advanced client and code editor for MySQL.
- MySQL Server, and Group Replication, which enables a set of MySQL instances to provide high-availability. InnoDB Cluster provides an alternative, easy to use programmatic way to work with Group Replication.
- MySQL Router, a lightweight middleware that provides transparent routing between your application and InnoDB Cluster.

The following diagram shows an overview of how these technologies work together:

Figure 21.1 InnoDB Cluster overview



Being built on MySQL Group Replication, provides features such as automatic membership management, fault tolerance, automatic failover, and so on. An InnoDB Cluster usually runs in a single-primary mode, with one primary instance (read-write) and multiple secondary instances (read-only). Advanced users can also take advantage of a multi-primary mode, where all instances are primaries. You can even change the topology of the cluster while InnoDB Cluster is online, to ensure the highest possible availability.

You work with InnoDB Cluster using the AdminAPI, provided as part of MySQL Shell. AdminAPI is available in JavaScript and Python, and is well suited to scripting and automation of deployments of MySQL to achieve high-availability and scalability. By using MySQL Shell's AdminAPI, you can avoid the need to configure many instances manually. Instead, AdminAPI provides an effective modern

interface to sets of MySQL instances and enables you to provision, administer, and monitor your deployment from one central tool.

To get started with InnoDB Cluster you need to download and install MySQL Shell. You need some hosts with MySQL Server instances installed, and you can also install MySQL Router.

InnoDB Cluster supports MySQL Clone, which enables you to provision instances simply. In the past, to provision a new instance before it joins a set of MySQL instances you would need to somehow manually transfer the transactions to the joining instance. This could involve making file copies, manually copying them, and so on. Using InnoDB Cluster, you can simply add an instance to the cluster and it is automatically provisioned.

Similarly, InnoDB Cluster is tightly integrated with MySQL Router, and you can use AdminAPI to work with them together. MySQL Router can automatically configure itself based on an InnoDB Cluster, in a process called bootstrapping, which removes the need for you to configure routing manually. MySQL Router then transparently connects client applications to the InnoDB Cluster, providing routing and load-balancing for client connections. This integration also enables you to administer some aspects of a MySQL Router bootstrapped against an InnoDB Cluster using AdminAPI. InnoDB Cluster status information includes details about MySQL Routers bootstrapped against the cluster. Operations enable you to create MySQL Router users at the cluster level, to work with the MySQL Routers bootstrapped against the cluster, and so on.

For more information on these technologies, see the user documentation linked in the descriptions. In addition to this user documentation, there is developer documentation for all AdminAPI methods in the MySQL Shell JavaScript API Reference or MySQL Shell Python API Reference, available from Connectors and APIs.

Chapter 22 InnoDB ReplicaSet

This chapter introduces MySQL InnoDB ReplicaSet, which combines MySQL technologies to enable you to deploy and administer Chapter 17, *Replication*. This content is a high-level overview of InnoDB ReplicaSet, for full documentation, see MySQL InnoDB ReplicaSet.

An InnoDB ReplicaSet consists of at least two MySQL Server instances, and it provides all of the MySQL Replication features you are familiar with, such as read scale-out and data security. InnoDB ReplicaSet uses the following MySQL technologies:

- MySQL Shell, which is an advanced client and code editor for MySQL.
- MySQL Server, and Chapter 17, *Replication*, which enables a set of MySQL instances to provide availability and asynchronous read scale-out. InnoDB ReplicaSet provides an alternative, easy to use programmatic way to work with Replication.
- MySQL Router, a lightweight middleware that provides transparent routing between your application and InnoDB ReplicaSet.

The interface to an InnoDB ReplicaSet is similar to MySQL InnoDB Cluster, you use MySQL Shell to work with MySQL Server instances as a ReplicaSet, and MySQL Router is also tightly integrated in the same way as InnoDB Cluster.

Being based on MySQL Replication, an InnoDB ReplicaSet has a single primary, which replicates to one or more secondary instances. An InnoDB ReplicaSet does not provide all of the features which InnoDB Cluster provides, such as automatic failover, or multi-primary mode. But, it does support features such as configuring, adding, and removing instances in a similar way. You can manually switch over or fail over to a secondary instance, for example in the event of a failure. You can even adopt an existing Replication deployment and then administer it as an InnoDB ReplicaSet.

You work with InnoDB ReplicaSet using the AdminAPI, provided as part of MySQL Shell. AdminAPI is available in JavaScript and Python, and is well suited to scripting and automation of deployments of MySQL to achieve high-availability and scalability. By using MySQL Shell's AdminAPI, you can avoid the need to configure many instances manually. Instead, AdminAPI provides an effective modern interface to sets of MySQL instances and enables you to provision, administer, and monitor your deployment from one central tool.

To get started with InnoDB ReplicaSet you need to download and install MySQL Shell. You need some hosts with MySQL Server instances installed, and you can also install MySQL Router.

InnoDB ReplicaSet supports MySQL Clone, which enables you to provision instances simply. In the past, to provision a new instance before it joined a MySQL Replication deployment, you would need to somehow manually transfer the transactions to the joining instance. This could involve making file copies, manually copying them, and so on. You can simply add an instance to the replica set and it is automatically provisioned.

Similarly, InnoDB ReplicaSet is tightly integrated with MySQL Router, and you can use AdminAPI to work with them together. MySQL Router can automatically configure itself based on an InnoDB ReplicaSet, in a process called bootstrapping, which removes the need for you to configure routing manually. MySQL Router then transparently connects client applications to the InnoDB ReplicaSet, providing routing and load-balancing for client connections. This integration also enables you to administer some aspects of a MySQL Router bootstrapped against an InnoDB ReplicaSet using AdminAPI. InnoDB ReplicaSet status information includes details about MySQL Routers bootstrapped against the ReplicaSet. Operations enable you to create MySQL Router users at the ReplicaSet level, to work with the MySQL Routers bootstrapped against the ReplicaSet, and so on.

For more information on these technologies, see the user documentation linked in the descriptions. In addition to this user documentation, there is developer documentation for all AdminAPI methods in the MySQL Shell JavaScript API Reference or MySQL Shell Python API Reference, available from Connectors and APIs.

Chapter 23 MySQL NDB Cluster 8.0

Table of Contents

This chapter provides information about MySQL *NDB Cluster*, a high-availability, high-redundancy version of MySQL adapted for the distributed computing environment. The most recent NDB Cluster release series uses version 8 of the `NDB` storage engine (also known as `NDBCLUSTER`) to enable running several computers with MySQL servers and other software in a cluster. NDB Cluster 8.0, now available as a General Availability (GA) release (beginning with version 8.0.19), incorporates version 8.0 of the `NDB` storage engine. NDB Cluster 7.6 and NDB Cluster 7.5, still available as GA releases, use versions 7.6 and 7.5 of `NDB`, respectively. Previous GA releases NDB Cluster 7.4 and NDB Cluster 7.3 incorporated `NDB` versions 7.4 and 7.3, respectively. *NDB 7.4 and older release series are no longer supported or maintained*.

This chapter contains information about NDB Cluster 8.0 releases through 8.0.34. NDB Cluster 8.0 is now available (beginning with NDB 8.0.19) as a General Availability release, and recommended for new deployments; the latest available release is NDB 8.0.32. NDB Cluster 7.6 and 7.5 are previous GA releases still supported in production; for information about NDB Cluster 7.6, see What is New in NDB Cluster 7.6. For similar information about NDB Cluster 7.5, see What is New in NDB Cluster 7.5. NDB Cluster 7.4 and 7.3 are previous GA releases which are no longer maintained. We recommend that new deployments for production use MySQL NDB Cluster 8.0.

23.1 General Information

MySQL NDB Cluster uses the MySQL server with the `NDB` storage engine. Support for the `NDB` storage engine is not included in standard MySQL Server 8.0 binaries built by Oracle. Instead, users of NDB Cluster binaries from Oracle should upgrade to the most recent binary release of NDB Cluster for supported platforms—these include RPMs that should work with most Linux distributions. NDB Cluster 8.0 users who build from source should use the sources provided for MySQL 8.0 and build with the options required to provide NDB support. (Locations where the sources can be obtained are listed later in this section.)

Important

MySQL NDB Cluster does not support InnoDB Cluster, which must be deployed using MySQL Server 8.0 with the `InnoDB` storage engine as well as additional applications that are not included in the NDB Cluster distribution. MySQL Server 8.0 binaries cannot be used with MySQL NDB Cluster. For more information about deploying and using InnoDB Cluster, see MySQL AdminAPI. Section 23.2.6, “MySQL Server Using InnoDB Compared with NDB Cluster”, discusses differences between the `NDB` and `InnoDB` storage engines.

Supported Platforms. NDB Cluster is currently available and supported on a number of platforms. For exact levels of support available for on specific combinations of operating system versions, operating system distributions, and hardware platforms, please refer to https://www.mysql.com/support/supportedplatforms/cluster.html.

Availability. NDB Cluster binary and source packages are available for supported platforms from https://dev.mysql.com/downloads/cluster/.

NDB Cluster release numbers. NDB 8.0 follows the same release pattern as the MySQL Server 8.0 series of releases, beginning with MySQL 8.0.13 and MySQL NDB Cluster 8.0.13. In this *Manual* and other MySQL documentation, we identify these and later NDB Cluster releases employing a version number that begins with “NDB”. This version number is that of the `NDBCLUSTER` storage engine used in the NDB 8.0 release, and is the same as the MySQL 8.0 server version on which the NDB Cluster 8.0 release is based.

Version strings used in NDB Cluster software. The version string displayed by the `mysql` client supplied with the MySQL NDB Cluster distribution uses this format:

```
mysql-mysql_server_version-cluster
```

`mysql_server_version` represents the version of the MySQL Server on which the NDB Cluster release is based. For all NDB Cluster 8.0 releases, this is `8.0.n`, where *`n`* is the release number. Building from source using `-DWITH_NDB` or the equivalent adds the `-cluster` suffix to the version string. (See Section 23.3.1.4, “Building NDB Cluster from Source on Linux”, and Section 23.3.2.2, “Compiling and Installing NDB Cluster from Source on Windows”.) You can see this format used in the `mysql` client, as shown here:

```
$> mysql
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 2
Server version: 8.0.34-cluster Source distribution

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> SELECT VERSION()\G
*************************** 1. row ***************************
VERSION(): 8.0.34-cluster
1 row in set (0.00 sec)
```

The first General Availability release of NDB Cluster using MySQL 8.0 is NDB 8.0.19, using MySQL 8.0.19.

The version string displayed by other NDB Cluster programs not normally included with the MySQL 8.0 distribution uses this format:

```
mysql-mysql_server_version ndb-ndb_engine_version
```

mysql_server_version represents the version of the MySQL Server on which the NDB Cluster release is based. For all NDB Cluster 8.0 releases, this is `8.0.n`, where *n* is the release number. *ndb_engine_version* is the version of the `NDB` storage engine used by this release of the NDB Cluster software. For all NDB 8.0 releases, this number is the same as the MySQL Server version. You can see this format used in the output of the `SHOW` command in the `ndb_mgm` client, like this:

```
ndb_mgm> SHOW
Connected to Management Server at: localhost:1186
Cluster Configuration
---------------------
[ndbd(NDB)]     2 node(s)
id=1    @10.0.10.6  (mysql-8.0.34 ndb-8.0.34, Nodegroup: 0, *)
id=2    @10.0.10.8  (mysql-8.0.34 ndb-8.0.34, Nodegroup: 0)

[ndb_mgmd(MGM)] 1 node(s)
id=3    @10.0.10.2  (mysql-8.0.34 ndb-8.0.34)

[mysqld(API)]   2 node(s)
id=4    @10.0.10.10  (mysql-8.0.34 ndb-8.0.34)
id=5 (not connected, accepting connect from any host)
```

Compatibility with standard MySQL 8.0 releases. While many standard MySQL schemas and applications can work using NDB Cluster, it is also true that unmodified applications and database schemas may be slightly incompatible or have suboptimal performance when run using NDB Cluster (see Section 23.2.7, “Known Limitations of NDB Cluster”). Most of these issues can be overcome, but this also means that you are very unlikely to be able to switch an existing application datastore—that currently uses, for example, `MyISAM` or `InnoDB`—to use the `NDB` storage engine without allowing for the possibility of changes in schemas, queries, and applications. A `mysqld` compiled without `NDB` support (that is, built without `-DWITH_NDB` or `-DWITH_NDBCLUSTER_STORAGE_ENGINE`) cannot function as a drop-in replacement for a `mysqld` that is built with it.

NDB Cluster development source trees. NDB Cluster development trees can also be accessed from https://github.com/mysql/mysql-server.

The NDB Cluster development sources maintained at https://github.com/mysql/mysql-server are licensed under the GPL. For information about obtaining MySQL sources using Git and building them yourself, see Section 2.8.5, “Installing MySQL Using a Development Source Tree”.

Note

As with MySQL Server 8.0, NDB Cluster 8.0 releases are built using `CMake`.

NDB Cluster 8.0 is available beginning with NDB 8.0.19 as a General Availability release, and is recommended for new deployments. NDB Cluster 7.6 and 7.5 are previous GA releases still supported in production; for information about NDB Cluster 7.6, see What is New in NDB Cluster 7.6. For similar information about NDB Cluster 7.5, see What is New in NDB Cluster 7.5. NDB Cluster 7.4 and 7.3 are previous GA releases which are no longer maintained. We recommend that new deployments for production use MySQL NDB Cluster 8.0.

The contents of this chapter are subject to revision as NDB Cluster continues to evolve. Additional information regarding NDB Cluster can be found on the MySQL website at http://www.mysql.com/products/cluster/.

Additional Resources. More information about NDB Cluster can be found in the following places:

- For answers to some commonly asked questions about NDB Cluster, see Section A.10, “MySQL 8.0 FAQ: NDB Cluster”.
- The NDB Cluster Forum: https://forums.mysql.com/list.php?25.

- Many NDB Cluster users and developers blog about their experiences with NDB Cluster, and make feeds of these available through PlanetMySQL.

23.2 NDB Cluster Overview

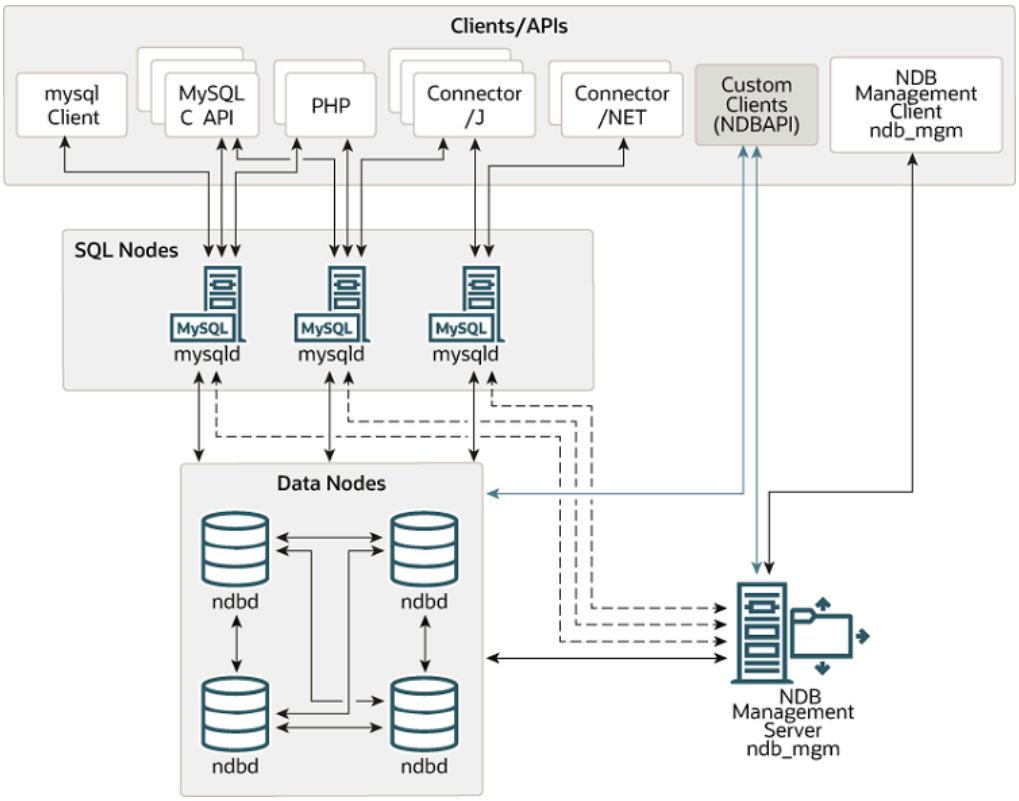
NDB Cluster is a technology that enables clustering of in-memory databases in a shared-nothing system. The shared-nothing architecture enables the system to work with very inexpensive hardware, and with a minimum of specific requirements for hardware or software.

NDB Cluster is designed not to have any single point of failure. In a shared-nothing system, each component is expected to have its own memory and disk, and the use of shared storage mechanisms such as network shares, network file systems, and SANs is not recommended or supported.

NDB Cluster integrates the standard MySQL server with an in-memory clustered storage engine called `NDB` (which stands for "*N*etwork *D*ata*B*ase"). In our documentation, the term `NDB` refers to the part of the setup that is specific to the storage engine, whereas "MySQL NDB Cluster" refers to the combination of one or more MySQL servers with the `NDB` storage engine.

An NDB Cluster consists of a set of computers, known as *hosts*, each running one or more processes. These processes, known as *nodes*, may include MySQL servers (for access to NDB data), data nodes (for storage of the data), one or more management servers, and possibly other specialized data access programs. The relationship of these components in an NDB Cluster is shown here:

Figure 23.1 NDB Cluster Components



All these programs work together to form an NDB Cluster (see Section 23.5, "NDB Cluster Programs". When data is stored by the `NDB` storage engine, the tables (and table data) are stored in the data nodes. Such tables are directly accessible from all other MySQL servers (SQL nodes) in the cluster. Thus, in a payroll application storing data in a cluster, if one application updates the salary of an employee, all other MySQL servers that query this data can see this change immediately.

Although an NDB Cluster SQL node uses the `mysqld` server daemon, it differs in a number of critical respects from the `mysqld` binary supplied with the MySQL 8.0 distributions, and the two versions of `mysqld` are not interchangeable.

In addition, a MySQL server that is not connected to an NDB Cluster cannot use the `NDB` storage engine and cannot access any NDB Cluster data.

The data stored in the data nodes for NDB Cluster can be mirrored; the cluster can handle failures of individual data nodes with no other impact than that a small number of transactions are aborted due to losing the transaction state. Because transactional applications are expected to handle transaction failure, this should not be a source of problems.

Individual nodes can be stopped and restarted, and can then rejoin the system (cluster). Rolling restarts (in which all nodes are restarted in turn) are used in making configuration changes and software upgrades (see Section 23.6.5, “Performing a Rolling Restart of an NDB Cluster”). Rolling restarts are also used as part of the process of adding new data nodes online (see Section 23.6.7, “Adding NDB Cluster Data Nodes Online”). For more information about data nodes, how they are organized in an NDB Cluster, and how they handle and store NDB Cluster data, see Section 23.2.2, “NDB Cluster Nodes, Node Groups, Fragment Replicas, and Partitions”.

Backing up and restoring NDB Cluster databases can be done using the `NDB`-native functionality found in the NDB Cluster management client and the `ndb_restore` program included in the NDB Cluster distribution. For more information, see Section 23.6.8, “Online Backup of NDB Cluster”, and Section 23.5.23, “ndb_restore — Restore an NDB Cluster Backup”. You can also use the standard MySQL functionality provided for this purpose in `mysqldump` and the MySQL server. See Section 4.5.4, “mysqldump — A Database Backup Program”, for more information.

NDB Cluster nodes can employ different transport mechanisms for inter-node communications; TCP/IP over standard 100 Mbps or faster Ethernet hardware is used in most real-world deployments.

23.2.1 NDB Cluster Core Concepts

`NDBCLUSTER` (also known as `NDB`) is an in-memory storage engine offering high-availability and data-persistence features.

The `NDBCLUSTER` storage engine can be configured with a range of failover and load-balancing options, but it is easiest to start with the storage engine at the cluster level. NDB Cluster's `NDB` storage engine contains a complete set of data, dependent only on other data within the cluster itself.

The “Cluster” portion of NDB Cluster is configured independently of the MySQL servers. In an NDB Cluster, each part of the cluster is considered to be a *node*.

Note

In many contexts, the term “node” is used to indicate a computer, but when discussing NDB Cluster it means a *process*. It is possible to run multiple nodes on a single computer; for a computer on which one or more cluster nodes are being run we use the term *cluster host*.

There are three types of cluster nodes, and in a minimal NDB Cluster configuration, there are at least three nodes, one of each of these types:

- *Management node*: The role of this type of node is to manage the other nodes within the NDB Cluster, performing such functions as providing configuration data, starting and stopping nodes, and running backups. Because this node type manages the configuration of the other nodes, a node of this type should be started first, before any other node. A management node is started with the command `ndb_mgmd`.

- *Data node*: This type of node stores cluster data. There are as many data nodes as there are fragment replicas, times the number of fragments (see Section 23.2.2, “NDB Cluster Nodes, Node Groups, Fragment Replicas, and Partitions”). For example, with two fragment replicas, each having two fragments, you need four data nodes. One fragment replica is sufficient for data storage, but provides no redundancy; therefore, it is recommended to have two (or more) fragment replicas to

provide redundancy, and thus high availability. A data node is started with the command `ndbd` (see Section 23.5.1, “ndbd — The NDB Cluster Data Node Daemon”) or `ndbmtd` (see Section 23.5.3, “ndbmtd — The NDB Cluster Data Node Daemon (Multi-Threaded)”).

NDB Cluster tables are normally stored completely in memory rather than on disk (this is why we refer to NDB Cluster as an *in-memory* database). However, some NDB Cluster data can be stored on disk; see Section 23.6.11, “NDB Cluster Disk Data Tables”, for more information.

- *SQL node*: This is a node that accesses the cluster data. In the case of NDB Cluster, an SQL node is a traditional MySQL server that uses the `NDBCLUSTER` storage engine. An SQL node is a `mysqld` process started with the `--ndbcluster` and `--ndb-connectstring` options, which are explained elsewhere in this chapter, possibly with additional MySQL server options as well.

  An SQL node is actually just a specialized type of *API node*, which designates any application which accesses NDB Cluster data. Another example of an API node is the `ndb_restore` utility that is used to restore a cluster backup. It is possible to write such applications using the NDB API. For basic information about the NDB API, see Getting Started with the NDB API.

Important

It is not realistic to expect to employ a three-node setup in a production environment. Such a configuration provides no redundancy; to benefit from NDB Cluster's high-availability features, you must use multiple data and SQL nodes. The use of multiple management nodes is also highly recommended.

For a brief introduction to the relationships between nodes, node groups, fragment replicas, and partitions in NDB Cluster, see Section 23.2.2, “NDB Cluster Nodes, Node Groups, Fragment Replicas, and Partitions”.

Configuration of a cluster involves configuring each individual node in the cluster and setting up individual communication links between nodes. NDB Cluster is currently designed with the intention that data nodes are homogeneous in terms of processor power, memory space, and bandwidth. In addition, to provide a single point of configuration, all configuration data for the cluster as a whole is located in one configuration file.

The management server manages the cluster configuration file and the cluster log. Each node in the cluster retrieves the configuration data from the management server, and so requires a way to determine where the management server resides. When interesting events occur in the data nodes, the nodes transfer information about these events to the management server, which then writes the information to the cluster log.

In addition, there can be any number of cluster client processes or applications. These include standard MySQL clients, `NDB`-specific API programs, and management clients. These are described in the next few paragraphs.

Standard MySQL clients. NDB Cluster can be used with existing MySQL applications written in PHP, Perl, C, C++, Java, Python, and so on. Such client applications send SQL statements to and receive responses from MySQL servers acting as NDB Cluster SQL nodes in much the same way that they interact with standalone MySQL servers.

MySQL clients using an NDB Cluster as a data source can be modified to take advantage of the ability to connect with multiple MySQL servers to achieve load balancing and failover. For example, Java clients using Connector/J 5.0.6 and later can use `jdbc:mysql:loadbalance://` URLs (improved in Connector/J 5.1.7) to achieve load balancing transparently; for more information about using Connector/J with NDB Cluster, see Using Connector/J with NDB Cluster.

NDB client programs. Client programs can be written that access NDB Cluster data directly from the `NDBCLUSTER` storage engine, bypassing any MySQL Servers that may be connected to the cluster, using the *NDB API*, a high-level C++ API. Such applications may be useful for specialized purposes where an SQL interface to the data is not needed. For more information, see The NDB API.

NDB-specific Java applications can also be written for NDB Cluster using the *NDB Cluster Connector for Java*. This NDB Cluster Connector includes *ClusterJ*, a high-level database API similar to object-relational mapping persistence frameworks such as Hibernate and JPA that connect directly to NDBCLUSTER, and so does not require access to a MySQL Server. See Java and NDB Cluster, and The ClusterJ API and Data Object Model, for more information.

NDB Cluster also supports applications written in JavaScript using Node.js. The MySQL Connector for JavaScript includes adapters for direct access to the NDB storage engine and as well as for the MySQL Server. Applications using this Connector are typically event-driven and use a domain object model similar in many ways to that employed by ClusterJ. For more information, see MySQL NoSQL Connector for JavaScript.

Management clients. These clients connect to the management server and provide commands for starting and stopping nodes gracefully, starting and stopping message tracing (debug versions only), showing node versions and status, starting and stopping backups, and so on. An example of this type of program is the ndb_mgm management client supplied with NDB Cluster (see Section 23.5.5, “ndb_mgm — The NDB Cluster Management Client”). Such applications can be written using the *MGM API*, a C-language API that communicates directly with one or more NDB Cluster management servers. For more information, see The MGM API.

Oracle also makes available MySQL Cluster Manager, which provides an advanced command-line interface simplifying many complex NDB Cluster management tasks, such restarting an NDB Cluster with a large number of nodes. The MySQL Cluster Manager client also supports commands for getting and setting the values of most node configuration parameters as well as mysqld server options and variables relating to NDB Cluster. MySQL Cluster Manager 1.4.8 provides experimental support for NDB 8.0. See MySQL Cluster Manager 1.4.8 User Manual, for more information.

Event logs. NDB Cluster logs events by category (startup, shutdown, errors, checkpoints, and so on), priority, and severity. A complete listing of all reportable events may be found in Section 23.6.3, “Event Reports Generated in NDB Cluster”. Event logs are of the two types listed here:

- *Cluster log*: Keeps a record of all desired reportable events for the cluster as a whole.
- *Node log*: A separate log which is also kept for each individual node.

Note

Under normal circumstances, it is necessary and sufficient to keep and examine only the cluster log. The node logs need be consulted only for application development and debugging purposes.

Checkpoint. Generally speaking, when data is saved to disk, it is said that a *checkpoint* has been reached. More specific to NDB Cluster, a checkpoint is a point in time where all committed transactions are stored on disk. With regard to the NDB storage engine, there are two types of checkpoints which work together to ensure that a consistent view of the cluster's data is maintained. These are shown in the following list:

- *Local Checkpoint (LCP)*: This is a checkpoint that is specific to a single node; however, LCPs take place for all nodes in the cluster more or less concurrently. An LCP usually occurs every few minutes; the precise interval varies, and depends upon the amount of data stored by the node, the level of cluster activity, and other factors.

  NDB 8.0 supports partial LCPs, which can significantly improve performance under some conditions. See the descriptions of the EnablePartialLcp and RecoveryWork configuration parameters which enable partial LCPs and control the amount of storage they use.

- *Global Checkpoint (GCP)*: A GCP occurs every few seconds, when transactions for all nodes are synchronized and the redo-log is flushed to disk.

For more information about the files and directories created by local checkpoints and global checkpoints, see NDB Cluster Data Node File System Directory.

23.2.2 NDB Cluster Nodes, Node Groups, Fragment Replicas, and Partitions

This section discusses the manner in which NDB Cluster divides and duplicates data for storage.

A number of concepts central to an understanding of this topic are discussed in the next few paragraphs.

Data node. An `ndbd` or `ndbmtd` process, which stores one or more *fragment replicas*—that is, copies of the *partitions* (discussed later in this section) assigned to the node group of which the node is a member.

Each data node should be located on a separate computer. While it is also possible to host multiple data node processes on a single computer, such a configuration is not usually recommended.

It is common for the terms “node” and “data node” to be used interchangeably when referring to an `ndbd` or `ndbmtd` process; where mentioned, management nodes (`ndb_mgmd` processes) and SQL nodes (`mysqld` processes) are specified as such in this discussion.

Node group. A node group consists of one or more nodes, and stores partitions, or sets of *fragment replicas* (see next item).

The number of node groups in an NDB Cluster is not directly configurable; it is a function of the number of data nodes and of the number of fragment replicas (`NoOfReplicas` configuration parameter), as shown here:

```
[# of node groups] = [# of data nodes] / NoOfReplicas
```

Thus, an NDB Cluster with 4 data nodes has 4 node groups if `NoOfReplicas` is set to 1 in the `config.ini` file, 2 node groups if `NoOfReplicas` is set to 2, and 1 node group if `NoOfReplicas` is set to 4. Fragment replicas are discussed later in this section; for more information about `NoOfReplicas`, see Section 23.4.3.6, “Defining NDB Cluster Data Nodes”.

Note

All node groups in an NDB Cluster must have the same number of data nodes.

You can add new node groups (and thus new data nodes) online, to a running NDB Cluster; see Section 23.6.7, “Adding NDB Cluster Data Nodes Online”, for more information.

Partition. This is a portion of the data stored by the cluster. Each node is responsible for keeping at least one copy of any partitions assigned to it (that is, at least one fragment replica) available to the cluster.

The number of partitions used by default by NDB Cluster depends on the number of data nodes and the number of LDM threads in use by the data nodes, as shown here:

```
[# of partitions] = [# of data nodes] * [# of LDM threads]
```

When using data nodes running `ndbmtd`, the number of LDM threads is controlled by the setting for `MaxNoOfExecutionThreads`. When using `ndbd` there is a single LDM thread, which means that there are as many cluster partitions as nodes participating in the cluster. This is also the case when using `ndbmtd` with `MaxNoOfExecutionThreads` set to 3 or less. (You should be aware that the number of LDM threads increases with the value of this parameter, but not in a strictly linear fashion, and that there are additional constraints on setting it; see the description of `MaxNoOfExecutionThreads` for more information.)

NDB and user-defined partitioning. NDB Cluster normally partitions `NDBCLUSTER` tables automatically. However, it is also possible to employ user-defined partitioning with `NDBCLUSTER` tables. This is subject to the following limitations:

1. Only the `KEY` and `LINEAR KEY` partitioning schemes are supported in production with `NDB` tables.

2. The maximum number of partitions that may be defined explicitly for any `NDB` table is `8 * [number of LDM threads] * [number of node groups]`, the number of node groups in an NDB Cluster being determined as discussed previously in this section. When running `ndbd` for data node processes, setting the number of LDM threads has no effect (since `ThreadConfig` applies only to `ndbmtd`); in such cases, this value can be treated as though it were equal to 1 for purposes of performing this calculation.

   See Section 23.5.3, “ndbmtd — The NDB Cluster Data Node Daemon (Multi-Threaded)”, for more information.

For more information relating to NDB Cluster and user-defined partitioning, see Section 23.2.7, “Known Limitations of NDB Cluster”, and Section 24.6.2, “Partitioning Limitations Relating to Storage Engines”.

Fragment replica. This is a copy of a cluster partition. Each node in a node group stores a fragment replica. Also sometimes known as a *partition replica*. The number of fragment replicas is equal to the number of nodes per node group.

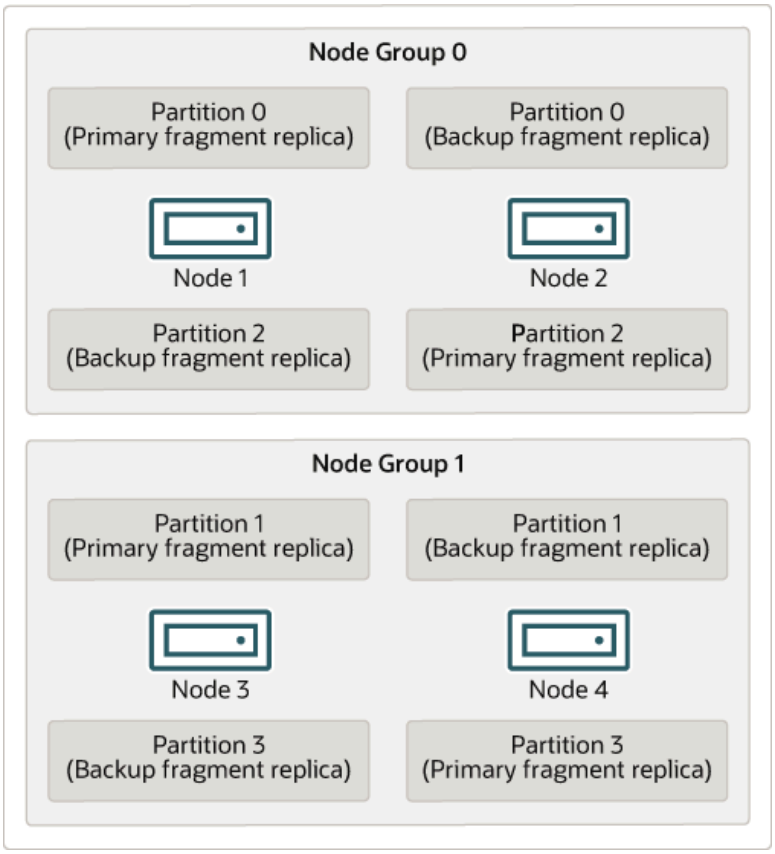
A fragment replica belongs entirely to a single node; a node can (and usually does) store several fragment replicas.

The following diagram illustrates an NDB Cluster with four data nodes running `ndbd`, arranged in two node groups of two nodes each; nodes 1 and 2 belong to node group 0, and nodes 3 and 4 belong to node group 1.

Note

Only data nodes are shown here; although a working NDB Cluster requires an `ndb_mgmd` process for cluster management and at least one SQL node to access the data stored by the cluster, these have been omitted from the figure for clarity.

Figure 23.2 NDB Cluster with Two Node Groups

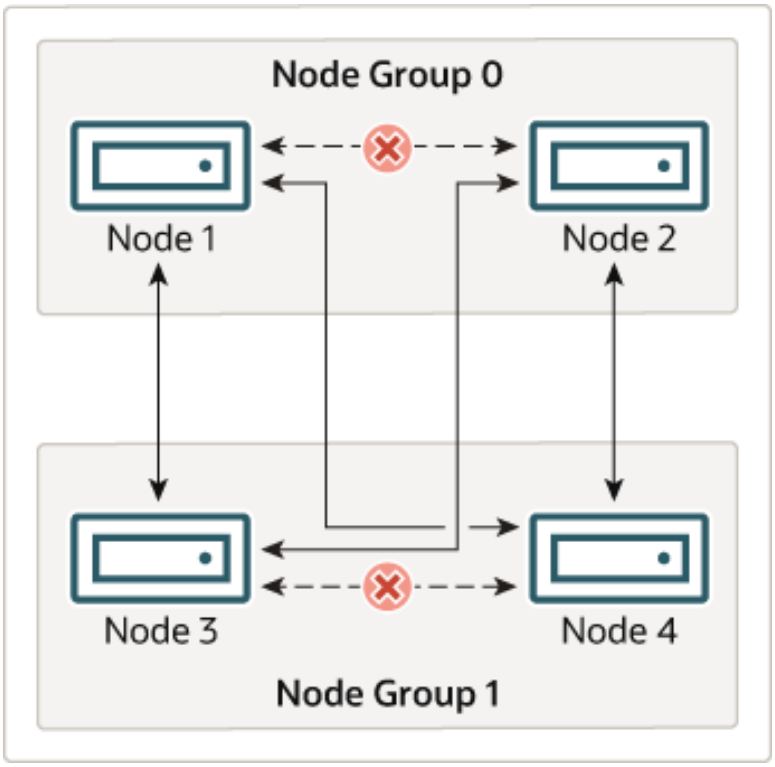


The data stored by the cluster is divided into four partitions, numbered 0, 1, 2, and 3. Each partition is stored—in multiple copies—on the same node group. Partitions are stored on alternate node groups as follows:

- Partition 0 is stored on node group 0; a *primary fragment replica* (primary copy) is stored on node 1, and a *backup fragment replica* (backup copy of the partition) is stored on node 2.
- Partition 1 is stored on the other node group (node group 1); this partition's primary fragment replica is on node 3, and its backup fragment replica is on node 4.
- Partition 2 is stored on node group 0. However, the placing of its two fragment replicas is reversed from that of Partition 0; for Partition 2, the primary fragment replica is stored on node 2, and the backup on node 1.
- Partition 3 is stored on node group 1, and the placement of its two fragment replicas are reversed from those of partition 1. That is, its primary fragment replica is located on node 4, with the backup on node 3.

What this means regarding the continued operation of an NDB Cluster is this: so long as each node group participating in the cluster has at least one node operating, the cluster has a complete copy of all data and remains viable. This is illustrated in the next diagram.

Figure 23.3 Nodes Required for a 2x2 NDB Cluster



In this example, the cluster consists of two node groups each consisting of two data nodes. Each data node is running an instance of `ndbd`. Any combination of at least one node from node group 0 and at least one node from node group 1 is sufficient to keep the cluster “alive”. However, if both nodes from a single node group fail, the combination consisting of the remaining two nodes in the other node group is not sufficient. In this situation, the cluster has lost an entire partition and so can no longer provide access to a complete set of all NDB Cluster data.

The maximum number of node groups supported for a single NDB Cluster instance is 48.

23.2.3 NDB Cluster Hardware, Software, and Networking Requirements

One of the strengths of NDB Cluster is that it can be run on commodity hardware and has no unusual requirements in this regard, other than for large amounts of RAM, due to the fact that all live data storage is done in memory. (It is possible to reduce this requirement using Disk Data tables—see Section 23.6.11, “NDB Cluster Disk Data Tables”, for more information about these.) Naturally, multiple and faster CPUs can enhance performance. Memory requirements for other NDB Cluster processes are relatively small.

The software requirements for NDB Cluster are also modest. Host operating systems do not require any unusual modules, services, applications, or configuration to support NDB Cluster. For supported operating systems, a standard installation should be sufficient. The MySQL software requirements are simple: all that is needed is a production release of NDB Cluster. It is not strictly necessary to compile MySQL yourself merely to be able to use NDB Cluster. We assume that you are using the binaries appropriate to your platform, available from the NDB Cluster software downloads page at https://dev.mysql.com/downloads/cluster/.

For communication between nodes, NDB Cluster supports TCP/IP networking in any standard topology, and the minimum expected for each host is a standard 100 Mbps Ethernet card, plus a switch, hub, or router to provide network connectivity for the cluster as a whole. We strongly recommend that an NDB Cluster be run on its own subnet which is not shared with machines not forming part of the cluster for the following reasons:

- **Security.** Communications between NDB Cluster nodes are not encrypted or shielded in any way. The only means of protecting transmissions within an NDB Cluster is to run your NDB Cluster on a protected network. If you intend to use NDB Cluster for Web applications, the cluster should definitely reside behind your firewall and not in your network's De-Militarized Zone (DMZ) or elsewhere.

  See Section 23.6.20.1, “NDB Cluster Security and Networking Issues”, for more information.

- **Efficiency.** Setting up an NDB Cluster on a private or protected network enables the cluster to make exclusive use of bandwidth between cluster hosts. Using a separate switch for your NDB Cluster not only helps protect against unauthorized access to NDB Cluster data, it also ensures that NDB Cluster nodes are shielded from interference caused by transmissions between other computers on the network. For enhanced reliability, you can use dual switches and dual cards to remove the network as a single point of failure; many device drivers support failover for such communication links.

Network communication and latency. NDB Cluster requires communication between data nodes and API nodes (including SQL nodes), as well as between data nodes and other data nodes, to execute queries and updates. Communication latency between these processes can directly affect the observed performance and latency of user queries. In addition, to maintain consistency and service despite the silent failure of nodes, NDB Cluster uses heartbeating and timeout mechanisms which treat an extended loss of communication from a node as node failure. This can lead to reduced redundancy. Recall that, to maintain data consistency, an NDB Cluster shuts down when the last node in a node group fails. Thus, to avoid increasing the risk of a forced shutdown, breaks in communication between nodes should be avoided wherever possible.

The failure of a data or API node results in the abort of all uncommitted transactions involving the failed node. Data node recovery requires synchronization of the failed node's data from a surviving data node, and re-establishment of disk-based redo and checkpoint logs, before the data node returns to service. This recovery can take some time, during which the Cluster operates with reduced redundancy.

Heartbeating relies on timely generation of heartbeat signals by all nodes. This may not be possible if the node is overloaded, has insufficient machine CPU due to sharing with other programs, or is experiencing delays due to swapping. If heartbeat generation is sufficiently delayed, other nodes treat the node that is slow to respond as failed.

This treatment of a slow node as a failed one may or may not be desirable in some circumstances, depending on the impact of the node's slowed operation on the rest of the cluster. When setting timeout

values such as `HeartbeatIntervalDbDb` and `HeartbeatIntervalDbApi` for NDB Cluster, care must be taken care to achieve quick detection, failover, and return to service, while avoiding potentially expensive false positives.

Where communication latencies between data nodes are expected to be higher than would be expected in a LAN environment (on the order of 100 µs), timeout parameters must be increased to ensure that any allowed periods of latency periods are well within configured timeouts. Increasing timeouts in this way has a corresponding effect on the worst-case time to detect failure and therefore time to service recovery.

LAN environments can typically be configured with stable low latency, and such that they can provide redundancy with fast failover. Individual link failures can be recovered from with minimal and controlled latency visible at the TCP level (where NDB Cluster normally operates). WAN environments may offer a range of latencies, as well as redundancy with slower failover times. Individual link failures may require route changes to propagate before end-to-end connectivity is restored. At the TCP level this can appear as large latencies on individual channels. The worst-case observed TCP latency in these scenarios is related to the worst-case time for the IP layer to reroute around the failures.

23.2.4 What is New in MySQL NDB Cluster

The following sections describe changes in the implementation of NMySQL DB Cluster in NDB Cluster 8.0 through 8.0.34, as compared to earlier release series. NDB Cluster 8.0 is available as a General Availability (GA) release, beginning with NDB 8.0.19. NDB Cluster 7.6 and 7.5 are previous GA releases still supported in production; for information about NDB Cluster 7.6, see What is New in NDB Cluster 7.6. For similar information about NDB Cluster 7.5, see What is New in NDB Cluster 7.5. NDB Cluster 7.4 and 7.3 were previous GA releases which have reached their end of life, and which are no longer supported or maintained. We recommend that new deployments for production use MySQL NDB Cluster 8.0.

What is New in NDB Cluster 8.0

Major changes and new features in NDB Cluster 8.0 which are likely to be of interest are shown in the following list:

- **Compatibility enhancements.** The following changes reduce longstanding nonessential differences in `NDB` behavior as compared to that of other MySQL storage engines:

  - **Development in parallel with MySQL server.** Beginning with this release, MySQL NDB Cluster is being developed in parallel with the standard MySQL 8.0 server under a new unified release model with the following features:

    - NDB 8.0 is developed in, built from, and released with the MySQL 8.0 source code tree.
    - The numbering scheme for NDB Cluster 8.0 releases follows the scheme for MySQL 8.0.
    - Building the source with `NDB` support appends `-cluster` to the version string returned by `mysql -V`, as shown here:

      ```
      $> mysql -V
      mysql  Ver 8.0.34-cluster for Linux on x86_64 (Source distribution)
      ```

      `NDB` binaries continue to display both the MySQL Server version and the `NDB` engine version, like this:

      ```
      $> ndb_mgm -V
      MySQL distrib mysql-8.0.34 ndb-8.0.34, for Linux (x86_64)
      ```

      In MySQL Cluster NDB 8.0, these two version numbers are always the same.

  To build the MySQL source with NDB Cluster support, use the CMake option `-DWITH_NDB` (NDB 8.0.31 and later; for earlier releases, use `-DWITH_NDBCLUSTER` instead).

- **Platform support notes.** NDB 8.0 makes the following changes in platform support:
  - `NDBCLUSTER` no longer supports 32-bit platforms. Beginning with NDB 8.0.21, the NDB build process checks the system architecture and aborts if it is not a 64-bit platform.
  - It is now possible to build `NDB` from source for 64-bit `ARM` CPUs. Currently, this support is source-only, and we do not provide any precompiled binaries for this platform.
- **Database and table names.** NDB 8.0 removes the previous 63-byte limit on identifiers for databases and tables. These identifiers can now use up to 64 bytes, as for such objects using other MySQL storage engines. See Section 23.2.7.11, “Previous NDB Cluster Issues Resolved in NDB Cluster 8.0”.
- **Generated names for foreign keys.** `NDB` now uses the pattern *`tbl_name`*`_fk_`*`N`* for naming internally generated foreign keys. This is similar to the pattern used by `InnoDB`.

- **Schema and metadata distribution and synchronization.** NDB 8.0 makes use of the MySQL data dictionary to distribute schema information to SQL nodes joining a cluster and to synchronize new schema changes between existing SQL nodes. The following list describes individual enhancements relating to this integration work:
  - **Schema distribution enhancements.** The `NDB` schema distribution coordinator, which handles schema operations and tracks their progress, has been extended in NDB 8.0 to ensure that resources used during a schema operation are released at its conclusion. Previously, some of this work was done by the schema distribution client; this has been changed due to the fact that the client did not always have all needed state information, which could lead to resource leaks when the client decided to abandon the schema operation prior to completion and without informing the coordinator.

    To help fix this issue, schema operation timeout detection has been moved from the schema distribution client to the coordinator, providing the coordinator with an opportunity to clean up any resources used during the schema operation. The coordinator now checks ongoing schema operations for timeout at regular intervals, and marks participants that have not yet completed a given schema operation as failed when detecting timeout. It also provides suitable warnings whenever a schema operation timeout occurs. (It should be noted that, after such a timeout is detected, the schema operation itself continues.) Additional reporting is done by printing a list of active schema operations at regular intervals whenever one or more of these operations is ongoing.

    As an additional part of this work, a new `mysqld` option `--ndb-schema-dist-timeout` makes it possible to set the length of time to wait until a schema operation is marked as having timed out.
  - **Disk data file distribution.** NDB Cluster 8.0.14, uses the MySQL data dictionary to make sure that disk data files and related constructs such as tablespaces and log file groups are correctly distributed between all connected SQL nodes.
  - **Schema synchronization of tablespace objects.** When a MySQL Server connects as an SQL node to an NDB cluster, it checks its data dictionary against the information found in the `NDB` dictionary.

    Previously, the only `NDB` objects synchronized on connection of a new SQL node were databases and tables; MySQL NDB Cluster 8.0 also implements schema synchronization of disk data objects including tablespaces and log file groups. Among other benefits, this eliminates the possibility of a mismatch between the MySQL data dictionary and the `NDB` dictionary following a native backup and restore, in which tablespaces and log file groups were restored to the `NDB` dictionary, but not to the MySQL Server's data dictionary.

    It is also no longer possible to issue a `CREATE TABLE` statement that refers to a nonexistent tablespace. Such a statement now fails with an error.

- **Database DDL synchronization enhancements.** Work done for NDB 8.0 insures that synchronization of databases by newly joined (or rejoined) SQL nodes with those on existing SQL nodes now makes proper use of the data dictionary so that any database-level operations (`CREATE DATABASE`, `ALTER DATABASE`, or `DROP DATABASE`) that may have been missed by this SQL node are now correctly duplicated on it when it connects (or reconnects) to the cluster.

  As part of the schema synchronization procedure performed when starting, an SQL node now compares all databases on the cluster's data nodes with those in its own data dictionary, and if any of these is found to be missing from the SQL node's data dictionary, the SQL Node installs it locally by executing a `CREATE DATABASE` statement. A database thus created uses the default MySQL Server database properties (such as those as determined by `character_set_database` and `collation_database`) that are in effect on this SQL node at the time the statement is executed.

- **NDB metadata change detection and synchronization.** NDB 8.0 implements a new mechanism for detection of updates to metadata for data objects such as tables, tablespaces, and log file groups with the MySQL data dictionary. This is done using a thread, the `NDB` metadata change monitor thread, which runs in the background and checks periodically for inconsistencies between the `NDB` dictionary and the MySQL data dictionary.

  The monitor performs metadata checks every 60 seconds by default. The polling interval can be adjusted by setting the value of the `ndb_metadata_check_interval` system variable; polling can be disabled altogether by setting the `ndb_metadata_check` system variable to `OFF`. The status variable `Ndb_metadata_detected_count` shows the number of times since `mysqld` was last started that inconsistencies have been detected.

  `NDB` ensures that `NDB` database, table, log file group, and tablespace objects submitted by the metadata change monitor thread during operations following startup are automatically checked for mismatches and synchronized by the `NDB` binlog thread.

  NDB 8.0 adds two status variables relating to automatic synchronization: `Ndb_metadata_synced_count` shows the number of objects synchronized automatically; `Ndb_metadata_excluded_count` indicates the number of objects for which synchronization has failed (prior to NDB 8.0.22, this variable was named `Ndb_metadata_blacklist_size`). In addition, you can see which objects have been synchronized by inspecting the cluster log.

  Setting the `ndb_metadata_sync` system variable to `true` overrides any settings that have been made for `ndb_metadata_check_interval` and `ndb_metadata_check`, causing the change monitor thread to begin continuous metadata change detection.

  In NDB 8.0.22 and later, setting `ndb_metadata_sync` to `true` clears the list of objects for which synchronization has failed previously, which means it is no longer necessary to discover individual tables or to re-trigger synchronization by reconnecting the SQL node to the cluster. In addition, setting this variable to `false` clears the list of objects waiting to be retried.

  Beginning with NDB 8.0.21, more detailed information about the current state of automatic synchronization than can be obtained from log messages or status variables is provided by two new tables added to the MySQL Performance Schema. The tables are listed here:

  - `ndb_sync_pending_objects`: Contains information about database objects for which mismatches have been detected between the `NDB` dictionary and the MySQL data dictionary (and which have not been excluded from automatic synchronization).

  - `ndb_sync_excluded_objects`: Contains information about `NDB` database objects which have been excluded because they cannot be synchronized between the `NDB` dictionary and the MySQL data dictionary, and thus require manual intervention.

  A row in one of these tables provides the database object's parent schema, name, and type. Types of objects include schemas, tablespaces, log file groups, and tables. (If the

object is a log file group or tablespace, the parent schema is `NULL`.) In addition, the `ndb_sync_excluded_objects` table shows the reason for which the object has been excluded.

These tables are present only if `NDBCLUSTER` storage engine support is enabled. For more information about these tables, see Section 27.12.12, “Performance Schema NDB Cluster Tables”.

- **Changes in NDB table extra metadata.** The extra metadata property of an `NDB` table is used for storing serialized metadata from the MySQL data dictionary, rather than storing the binary representation of the table as in previous versions. (This was a `.frm` file, no longer used by the MySQL Server—see Chapter 14, *MySQL Data Dictionary*.) As part of the work to support this change, the available size of the table's extra metadata has been increased. This means that `NDB` tables created in NDB Cluster 8.0 are not compatible with previous NDB Cluster releases. Tables created in previous releases can be used with NDB 8.0, but cannot be opened afterwards by an earlier version.

  This metadata is accessible using the NDB API methods `getExtraMetadata()` and `setExtraMetadata()`.

  For more information, see Section 23.3.7, “Upgrading and Downgrading NDB Cluster”.

- **On-the-fly upgrades of tables using .frm files.** A table created in NDB 7.6 and earlier contains metadata in the form of a compressed `.frm` file, which is no longer supported in MySQL 8.0. To facilitate online upgrades to NDB 8.0, `NDB` performs on-the-fly translation of this metadata and writes it into the MySQL Server's data dictionary, which enables the `mysqld` in NDB Cluster 8.0 to work with the table without preventing subsequent use of the table by a previous version of the `NDB` software.

  Important

  Once a table's structure has been modified in NDB 8.0, its metadata is stored using the data dictionary, and it can no longer be accessed by NDB 7.6 and earlier.

  This enhancement also makes it possible to restore an `NDB` backup made using an earlier version to a cluster running NDB 8.0 (or later).

- **Metadata consistency check error logging.** As part of work previously done in NDB 8.0, the metadata check performed as part of auto-synchronization between the representation of an `NDB` table in the NDB dictionary and its counterpart in the MySQL data dictionary includes the table's name, storage engine, and internal ID. Beginning with NDB 8.0.23, the range of properties checked is expanded to include properties of the following data objects:

  - Columns
  - Indexes
  - Foreign keys

  In addition, details of any mismatches in metadata properties are now written to the MySQL server error log. The formats used for the error log messages differ slightly depending on whether the discrepancy is found on the table level or on the level of a column, index, or foreign key. The format for a log error resulting from a table-level property mismatch is shown here, where *`property`* is the property name, *`ndb_value`* is the property value as stored in the

NDB dictionary, and *`mysqld_value`* is the value of the property as stored in the MySQL data dictionary:

```
Diff in 'property' detected, 'ndb_value' != 'mysqld_value'
```

For mismatches in properties of columns, indexes, and foreign keys, the format is as follows, where *`obj_type`* is one of `column`, `index`, or `foreign key`, and *`obj_name`* is the name of the object:

```
Diff in obj_type 'obj_name.property' detected, 'ndb_value' != 'mysqld_value'
```

Metadata checks are performed during automatic synchronization of `NDB` tables when they are installed in the data dictionary of any `mysqld` acting as an SQL node in an NDB Cluster. If the `mysqld` is debug-compiled, checks are also made whenever a `CREATE TABLE` statement is executed, and whenever an `NDB` table is opened.

- **Synchronization of user privileges with NDB_STORED_USER.** A new mechanism for sharing and synchronizing users, roles, and privileges between SQL nodes is available in NDB 8.0, using the `NDB_STORED_USER` privilege. Distributed privileges as implemented in NDB 7.6 and earlier (see Distributed Privileges Using Shared Grant Tables) are no longer supported.

  Once a user account is created on an SQL node, the user and its privileges can be stored in `NDB` and thus shared between all SQL nodes in the cluster by issuing a `GRANT` statement such as this one:

  ```
  GRANT NDB_STORED_USER ON *.* TO 'jon'@'localhost';
  ```

  `NDB_STORED_USER` always has global scope and must be granted using `ON *.*`. System reserved accounts such as `mysql.session@localhost` or `mysql.infoschema@localhost` cannot be assigned this privilege.

  Roles can also be shared between SQL nodes by issuing the appropriate `GRANT NDB_STORED_USER` statement. Assigning such a role to a user does not cause the user to be shared; the `NDB_STORED_USER` privilege must be granted to each user explicitly.

  A user or role having `NDB_STORED_USER`, along with its privileges, is shared with all SQL nodes as soon as they join a given NDB Cluster. It is possible to make such changes from any connected SQL node, but recommended practice is to do so from a designated SQL node only, since the order of execution of statements affecting privileges from different SQL nodes cannot be guaranteed to be the same on all SQL nodes.

  Prior to NDB 8.0.27, changes to the privileges of a user or role were synchronized immediately with all connected SQL nodes. Beginning with MySQL 8.0.27, an SQL node takes a global read lock when updating privileges, which keeps concurrent changes executed by multiple SQL nodes from causing a deadlock.

  Implications for upgrades. Due to changes in the MySQL server's privilege system (see Section 6.2.3, “Grant Tables”), privilege tables using the `NDB` storage engine do not function correctly in NDB 8.0. It is safe but not necessary to retain such privilege tables created in NDB 7.6 or earlier, but they are no longer used for access control. In NDB 8.0, a `mysqld` acting as an SQL node and detecting such tables in `NDB` writes a warning to the MySQL server log, and creates `InnoDB` shadow tables local to itself; such shadow tables are created on each MySQL server connected to the cluster. When performing an upgrade from NDB 7.6 or earlier, the privilege tables using `NDB` can be removed safely using `ndb_drop_table` once all MySQL servers acting as SQL nodes have been upgraded (see Section 23.3.7, “Upgrading and Downgrading NDB Cluster”).

  The `ndb_restore` utility's `--restore-privilege-tables` option is deprecated but continues to be honored in NDB 8.0, and can still be used to restore distributed privilege tables present in a

backup taken from a previous release of NDB Cluster to a cluster running NDB 8.0. These tables are handled as described in the preceding paragraph.

Shared users and grants are stored in the `ndb_sql_metadata` table, which `ndb_restore` by default does not restore in NDB 8.0; you can specify the `--include-stored-grants` option to cause it to do so.

See Section 23.6.13, "Privilege Synchronization and NDB_STORED_USER", for more information.

- **INFORMATION_SCHEMA changes.** The following changes are made in the display of information regarding Disk Data files in the Information Schema `FILES` table:

  - Tablespaces and log file groups are no longer represented in the `FILES` table. (These constructs are not actually files.)

  - Each data file is now represented by a single row in the `FILES` table. Each undo log file is also now represented in this table by one row only. (Previously, a row was displayed for each copy of each of these files on each data node.)

  In addition, `INFORMATION_SCHEMA` tables are now populated with tablespace statistics for MySQL Cluster tables. (Bug #27167728)

- **Error information with ndb_perror.** The deprecated `--ndb` option for `perror` has been removed. Instead, use `ndb_perror` to obtain error message information from `NDB` error codes. (Bug #81704, Bug #81705, Bug #23523926, Bug #23523957)

- **Condition pushdown enhancements.** Previously, condition pushdown was limited to predicate terms referring to column values from the same table to which the condition was being pushed. In NDB 8.0, this restriction is removed such that column values from tables earlier in the query plan can also be referred to from pushed conditions. NDB 8.0 supports joins comparing column expressions, as well as comparisons between columns in the same table. Columns and column expressions to be compared must be of exactly the same type; this means they must also be of the same signedness, length, character set, precision, and scale, whenever these attributes apply. Conditions being pushed could not be part of pushed joins prior to NDB 8.0.27, when this restriction is lifted.

  Pushing down larger parts of a condition allows more rows to be filtered out by the data nodes, thereby reducing the number of rows which `mysqld` must handle during join processing. Another benefit of these enhancements is that filtering can be performed in parallel in the LDM threads, rather than in a single mysqld process on an SQL node; this has the potential to improve query performance significantly.

  Existing rules for type compatibility between column values being compared continue to apply (see Section 8.2.1.5, "Engine Condition Pushdown Optimization").

  Pushdown of outer joins and semijoins. Work done in NDB 8.0.20 allows many outer joins and semijoins, and not only those using a primary key or unique key lookup, to be pushed down to the data nodes (see Section 8.2.1.5, "Engine Condition Pushdown Optimization").

  Outer joins using scans which can now be pushed include those which meet the following conditions:

  - There are no unpushed conditions on the table

  - There are no unpushed conditions on other tables in the same join nest, or in upper join nests on which it depends

- All other tables in the same join nest, or in upper join nests on which it depends, are also pushed

A semijoin that uses an index scan can now be pushed if it meets the conditions just noted for a pushed outer join, and it uses the `firstMatch` strategy (see Section 8.2.2.1, “Optimizing IN and EXISTS Subquery Predicates with Semijoin Transformations”).

These additional improvements are made in NDB 8.0.21:

- Antijoins produced by the MySQL Optimizer through the transformation of `NOT EXISTS` and `NOT IN` queries (see Section 8.2.2.1, “Optimizing IN and EXISTS Subquery Predicates with Semijoin Transformations”) can be pushed down to the data nodes by `NDB`.

  This can be done when there is no unpushed condition on the table, and the query fulfills any other conditions which must be met for an outer join to be pushed down.

- `NDB` attempts to identify and evaluate a non-dependent scalar subquery before trying to retrieve any rows from the table to which it is attached. When it can do so, the value obtained is used as part of a pushed condition, instead of using the subquery which provided the value.

Beginning with NDB 8.0.27, conditions pushed as part of a pushed query can now refer to columns from ancestor tables within the same pushed query, subject to the following conditions:

- Pushed conditions may include any of the comparison operators `<`, `<=`, `>`, `>=`, `=`, and `<>`.
- Values being compared must be of the same type, including length, precision, and scale.
- `NULL` handling is performed according to the comparison semantics specified by the ISO SQL standard; any comparison with `NULL` returns `NULL`.

Consider the table created using the statement shown here:

```
CREATE TABLE t (
    x INT PRIMARY KEY,
    y INT
) ENGINE=NDB;
```

A query such as `SELECT * FROM t AS m JOIN t AS n ON m.x >= n.y` can now use the engine condition pushdown optimization to push down the condition column `y`.

When a join cannot be pushed, `EXPLAIN` should provide the reason or reasons.

See Section 8.2.1.5, “Engine Condition Pushdown Optimization”, for more information.

The NDB API methods `branch_col_eq_param()`, `branch_col_ne_param()`, `branch_col_lt_param()`, `branch_col_le_param()`, `branch_col_gt_param()`, and `branch_col_ge_param()` were added in NDB 8.0.27 as part of this work. These `NdbInterpretedCode` can be used to compare column values with values of parameters.

In addition, `NdbScanFilter::cmp_param()`, also added in NDB 8.0.27, makes it possible to define comparisons between column values and parameter values for use in performing scans.

- **Increase in maximum row size.** NDB 8.0 increases the maximum number of bytes that can be stored in an `NDBCLUSTER` table from 14000 to 30000 bytes.

  A `BLOB` or `TEXT` column continues to use 264 bytes of this total, as before.

  The maximum offset for a fixed-width column of an `NDB` table is 8188 bytes; this is also unchanged from previous releases.

  See Section 23.2.7.5, “Limits Associated with Database Objects in NDB Cluster”, for more information.

- **ndb_mgm SHOW command and single user mode.** In NDB 8.0, when the cluster in single user mode, the output of the management client `SHOW` command indicates which API or SQL node has exclusive access while this mode is in effect.

- **Online column renames.** Columns of `NDB` tables can now be renamed online, using `ALGORITHM=INPLACE`. See Section 23.6.12, "Online Operations with ALTER TABLE in NDB Cluster", for more information.

- **Improved ndb_mgmd startup times.** Start times for management nodes daemon have been significantly improved in NDB 8.0, in the following ways:

  - Due to replacing the list data structure formerly used by `ndb_mgmd` for handling node properties from configuration data with a hash table, overall startup times for the management server have been decreased by a factor of 6 or more.

  - In addition, in cases where data and SQL node host names not present in the management server's `hosts` file are used in the cluster configuration file, `ndb_mgmd` start times can be up to 20 times shorter than was previously the case.

- **NDB API enhancements.** `NdbScanFilter::cmp()` and several comparison methods of `NdbInterpretedCode` can now be used to compare table column values with each other. The affected `NdbInterpretedCode` methods are listed here:

  - `branch_col_eq()`
  - `branch_col_ge()`
  - `branch_col_gt()`
  - `branch_col_le()`
  - `branch_col_lt()`
  - `branch_col_ne()`

  For all of the methods just listed, table column values to be compared much be of exactly matching types, including with respect to length, precision, signedness, scale, character set, and collation, as applicable.

  See the descriptions of the individual API methods for more information.

- **Offline multithreaded index builds.** It is now possible to specify a set of cores to be used for I/O threads performing offline multithreaded builds of ordered indexes, as opposed to normal I/O duties such as file I/O , compression , or decompression. "Offline" in this context refers to building of ordered indexes performed when the parent table is not being written to; such building takes place when an `NDB` cluster performs a node or system restart, or as part of restoring a cluster from backup using `ndb_restore --rebuild-indexes`.

  In addition, the default behavior for offline index build work is modified to use all cores available to `ndbmtd`, rather limiting itself to the core reserved for the I/O thread. Doing so can improve restart and restore times and performance, availability, and the user experience.

  This enhancement is implemented as follows:

  1. The default value for `BuildIndexThreads` is changed from 0 to 128. This means that offline ordered index builds are now multithreaded by default.

  2. The default value for `TwoPassInitialNodeRestartCopy` is changed from `false` to `true`. This means that an initial node restart first copies all data from a "live" node to one that is starting —without creating any indexes—builds ordered indexes offline, and then again synchronizes its data with the live node, that is, synchronizing twice and building indexes offline between the two

synchronizations. This causes an initial node restart to behave more like the normal restart of a node, and reduces the time required for building indexes.

3. A new thread type (`idxbld`) is defined for the `ThreadConfig` configuration parameter, to allow locking of offline index build threads to specific CPUs.

In addition, `NDB` now distinguishes the thread types that are accessible to `ThreadConfig` by these two criteria:

1. Whether the thread is an execution thread. Threads of types `main`, `ldm`, `recv`, `rep`, `tc`, and `send` are execution threads; thread types `io`, `watchdog`, and `idxbld` are not.

2. Whether the allocation of the thread to a given task is permanent or temporary. Currently all thread types except `idxbld` are permanent.

For additional information, see the descriptions of the indicated parameters in the Manual. (Bug #25835748, Bug #26928111)

- **logbuffers table backup process information.** When performing an NDB backup, the `ndbinfo.logbuffers` table now displays information regarding buffer usage by the backup process on each data node. This is implemented as rows reflecting two new log types in addition to `REDO` and `DD-UNDO`. 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. Rows having these two log types are present in the table only while an NDB backup is currently in progress. (Bug #25822988)

- **ndbinfo.processes table on Windows.** The process ID of the monitor process used on Windows platforms by `RESTART` to spawn and restart a `mysqld` is now shown in the `processes` table as an `angel_pid`.

- **String hashing improvements.** Prior to NDB 8.0, all string hashing was based on first transforming the string into a normalized form, then MD5-hashing the resulting binary image. This could give rise to some performance problems, for the following reasons:

  - The normalized string is always space padded to its full length. For a `VARCHAR`, this often involved adding more spaces than there were characters in the original string.

  - The string libraries were not optimized for this space padding, which added considerable overhead in some use cases.

  - The padding semantics varied between character sets, some of which were not padded to their full length.

  - The transformed string could become quite large, even without space padding; some Unicode 9.0 collations can transform a single code point into 100 bytes or more of character data.

  - Subsequent MD5 hashing consisted mainly of padding with spaces, and was not particularly efficient, possibly causing additional performance penalties by flushing significant portions of the L1 cache.

  A collation provides its own hash function, which hashes the string directly without first creating a normalized string. In addition, for a Unicode 9.0 collation, the hash is computed without padding. `NDB` now takes advantage of this built-in function whenever hashing a string identified as using a Unicode 9.0 collation.

  Since, for other collations, there are existing databases which are hash partitioned on the transformed string, `NDB` continues to employ the previous method for hashing strings that use these, to maintain compatibility. (Bug #89590, Bug #89604, Bug #89609, Bug #27515000, Bug #27523758, Bug #27522732)

- **RESET MASTER changes.** Because the MySQL Server now executes `RESET MASTER` with a global read lock, the behavior of this statement when used with NDB Cluster has changed in the following two respects:

  - It is no longer guaranteed to be synchronous; that is, it is now possible that a read coming immediately before `RESET MASTER` is issued may not be logged until after the binary log has been rotated.

  - It now behaves in exactly the same fashion, whether the statement is issued on the same SQL node that is writing the binary log, or on a different SQL node in the same cluster.

  Note

  `SHOW BINLOG EVENTS`, `FLUSH LOGS`, and most data definition statements continue, as they did in previous `NDB` versions, to operate in a synchronous fashion.

- **ndb_restore option usage.** The `--nodeid` and `--backupid` options are now both required when invoking `ndb_restore`.

- **ndb_log_bin default.** NDB 8.0 changes the default value of the `ndb_log_bin` system variable from `TRUE` to `FALSE`.

- **Dynamic transactional resource allocation.** Allocation of resources in the transaction coordinator is now performed using dynamic memory pools. This means that resource allocation determined by data node configuration parameters such as `MaxDMLOperationsPerTransaction`, `MaxNoOfConcurrentIndexOperations`, `MaxNoOfConcurrentOperations`, `MaxNoOfConcurrentScans`, `MaxNoOfConcurrentTransactions`, `MaxNoOfFiredTriggers`, `MaxNoOfLocalScans`, and `TransactionBufferMemory` is now done in such a way that, if the load represented by each of these parameters is within the target load for all such resources, others of these resources can be limited so as not to exceed the total resources available.

  As part of this work, several new data node parameters controlling transactional resources in `DBTC`, listed here, have been added:

  - `ReservedConcurrentIndexOperations`
  - `ReservedConcurrentOperations`
  - `ReservedConcurrentScans`
  - `ReservedConcurrentTransactions`
  - `ReservedFiredTriggers`
  - `ReservedLocalScans`
  - `ReservedTransactionBufferMemory`.

  See the descriptions of the parameters just listed for further information.

- **Backups using multiple LDMs per data node.** `NDB` backups can now be performed in a parallel fashion on individual data nodes using multiple local data managers (LDMs). (Previously, backups were done in parallel across data nodes, but were always serial within data node processes.) No special syntax is required for the `START BACKUP` command in the `ndb_mgm` client to enable this feature, but all data nodes must be using multiple LDMs. This means that data nodes must be running `ndbmtd` (`ndbd` is single-threaded and thus always has only one LDM) and they must be configured to use multiple LDMs before taking the backup; you can do this by

choosing an appropriate setting for one of the multi-threaded data node configuration parameters `MaxNoOfExecutionThreads` or `ThreadConfig`.

Backups using multiple LDMs create subdirectories, one per LDM, under the `BACKUP/BACKUP-backup_id/` directory. `ndb_restore` now detects these subdirectories automatically, and if they exist, attempts to restore the backup in parallel; see Section 23.5.23.3, “Restoring from a backup taken in parallel”, for details. (Single-threaded backups are restored as in previous versions of `NDB`.) It is also possible to restore backups taken in parallel using an `ndb_restore` binary from a previous version of NDB Cluster by modifying the usual restore procedure; Restoring a parallel backup serially, provides information on how to do this.

You can force the creation of single-threaded backups by setting the `EnableMultithreadedBackup` data node parameter to 0 for all data nodes in the `[ndbd default]` section of the cluster's global configuration file (`config.ini`).

- **Binary configuration file enhancements.** NDB 8.0 uses a new format for the management server's binary configuration file. Previously, a maximum of 16381 sections could appear in the cluster configuration file; now the maximum number of sections is 4G. This is intended to support larger numbers of nodes in a cluster than was possible before this change.

  Upgrades to the new format are relatively seamless, and should seldom if ever require manual intervention, as the management server continues to be able to read the old format without issue. A downgrade from NDB 8.0 to an older version of the NDB Cluster software requires manual removal of any binary configuration files or, alternatively, starting the older management server binary with the `--initial` option.

  For more information, see Section 23.3.7, “Upgrading and Downgrading NDB Cluster”.

- **Increased number of data nodes.** NDB 8.0 increases the maximum number of data nodes supported per cluster to 144 (previously, this was 48). Data nodes can now use node IDs in the range 1 to 144, inclusive.

  Previously, the recommended node IDs for management nodes were 49 and 50. These are still supported for management nodes, but using them as such limits the maximum number of data nodes to 142; for this reason, it is now recommended that node IDs 145 and 146 are used for management nodes.

  As part of this work, the format used for the data node `sysfile` has been updated to version 2. This file records information such as the last global checkpoint index, restart status, and node group membership of each node (see NDB Cluster Data Node File System Directory).

- **RedoOverCommitCounter and RedoOverCommitLimit changes.** Due to ambiguities in the semantics for setting them to 0, the minimum value for each of the data node configuration parameters `RedoOverCommitCounter` and `RedoOverCommitLimit` has been increased to 1.

- **ndb_autoincrement_prefetch_sz changes.** The default value of the `ndb_autoincrement_prefetch_sz` server system variable is increased to 512.

- **Changes in parameter maximums and defaults.** NDB 8.0 makes the following changes in configuration parameter maximum and default values:

  - The maximum for `DataMemory` is increased to 16 terabytes.
  - The maximum for `DiskPageBufferMemory` is also increased to 16 terabytes.
  - The default value for `StringMemory` is increased to 25%.
  - The default for `LcpScanProgressTimeout` is increased to 180 seconds.

- **Disk Data checkpointing improvements.** NDB Cluster 8.0 provides a number of new enhancements which help to reduce the latency of checkpoints of Disk Data tables and tablespaces

when using non-volatile memory devices such as solid-state drives and the NVMe specification for such devices. These improvements include those in the following list:

- Avoiding bursts of checkpoint disk writes
- Speeding up checkpoints for disk data tablespaces when the redo log or the undo log becomes full
- Balancing checkpoints to disk and in-memory checkpoints against one other, when necessary
- Protecting disk devices from overload to help ensure low latency under high loads

As part of this work, two data node configuration parameters have been added. `MaxDiskDataLatency` places a ceiling on the degree of latency permitted for disk access and causes transactions taking longer than this length of time to be aborted. `DiskDataUsingSameDisk` makes it possible to take advantage of housing Disk Data tablespaces on separate disks by increasing the rate at which checkpoints of such tablespaces can be performed.

In addition, three new tables in the `ndbinfo` database provide information about Disk Data performance:

- The `diskstat` table reports on writes to Disk Data tablespaces during the past second
- The `diskstats_1sec` table reports on writes to Disk Data tablespaces for each of the last 20 seconds
- The `pgman_time_track_stats` table reports on the latency of disk operations relating to Disk Data tablespaces

- **Memory allocation and TransactionMemory.** A new `TransactionMemory` parameter simplifies allocation of data node memory for transactions as part of the work done to pool transactional and Local Data Manager (LDM) memory. This parameter is intended to replace several older transactional memory parameters which have been deprecated.

  Transaction memory can now be set in any of the three ways listed here:

  - Several configuration parameters are incompatible with `TransactionMemory`. If any of these are set, `TransactionMemory` cannot be set (see Parameters incompatible with TransactionMemory), and the data node's transaction memory is determined as it was previous to NDB 8.0.

    Note

    Attempting to set `TransactionMemory` and any of these parameters concurrently in the `config.ini` file prevents the management server from starting.

  - If `TransactionMemory` is set, this value is used for determining transaction memory. `TransactionMemory` cannot be set if any of the incompatible parameters mentioned in the previous item have also been set.
  - If none of the incompatible parameters are set and `TransactionMemory` is also not set, transaction memory is set by `NDB`.

  For more information, see the description of `TransactionMemory`, as well as Section 23.4.3.13, “Data Node Memory Management”.

- **Support for additional fragment replicas.** NDB 8.0 increases the maximum number of fragment replicas supported in production from two to four. (Previously, it was possible to set `NoOfReplicas` to 3 or 4, but this was not officially supported or verified in testing.)

- **Restoring by slices.** Beginning with NDB 8.0.20, it is possible to divide a backup into roughly equal portions (slices) and to restore these slices in parallel using two new options implemented for `ndb_restore`:

  - `--num-slices` determines the number of slices into which the backup should be divided.
  - `--slice-id` provides the ID of the slice to be restored by the current instance of `ndb_restore`.

  This makes it possible to employ multiple instances of `ndb_restore` to restore subsets of the backup in parallel, potentially reducing the amount of time required to perform the restore operation.

  For more information, see the description of the `ndb_restore --num-slices` option.

- **Read from any fragment replica enabled.** Read from any fragment replica is enabled by default for all `NDB` tables. This means that the default value for the `ndb_read_backup` system variable is now ON, and that the value of the `NDB_TABLE` comment option `READ_BACKUP` is 1 when creating a new `NDB` table. Enabling read from any fragment replica significantly improves performance for reads from `NDB` tables, with minimal impact on writes.

  For more information, see the description of the `ndb_read_backup` system variable, and Section 13.1.20.12, “Setting NDB Comment Options”.

- **ndb_blob_tool enhancements.** Beginning with NDB 8.0.20, the `ndb_blob_tool` utility can detect missing blob parts for which inline parts exist and replace these with placeholder blob parts (consisting of space characters) of the correct length. To check whether there are missing blob parts, use the `--check-missing` option with this program. To replace any missing blob parts with placeholders, use the `--add-missing` option.

  For more information, see Section 23.5.6, “ndb_blob_tool — Check and Repair BLOB and TEXT columns of NDB Cluster Tables”.

- **ndbinfo versioning.** `NDB` 8.0.20 and later supports versioning for `ndbinfo` tables, and maintains the current definitions for its tables internally. At startup, `NDB` compares its supported `ndbinfo` version with the version stored in the data dictionary. If the versions differ, `NDB` drops any old `ndbinfo` tables and recreates them using the current definitions.

- **Support for Fedora Linux.** Beginning with NDB 8.0.20, Fedora Linux is a supported platform for NDB Cluster Community releases and can be installed using the RPMs supplied for this purpose by Oracle. These can be obtained from the NDB Cluster downloads page.

- **NDB programs—NDBT dependency removal.** The dependency of a number of `NDB` utility programs on the `NDBT` library has been removed. This library is used internally for development, and is not required for normal use; its inclusion in these programs could lead to unwanted issues when testing.

  Affected programs are listed here, along with the `NDB` versions in which the dependency was removed:

  - `ndb_restore`
  - `ndb_delete_all`
  - `ndb_show_tables` (NDB 8.0.20)
  - `ndb_waiter` (NDB 8.0.20)

  The principal effect of this change for users is that these programs no longer print `NDBT_ProgramExit - status` following completion of a run. Applications that depend upon such behavior should be updated to reflect the change when upgrading to the indicated versions.

- **Foreign keys and lettercasing.** `NDB` stores the names of foreign keys using the case with which they were defined. Formerly, when the value of the `lower_case_table_names` system

variable was set to 0, it performed case-sensitive comparisons of foreign key names as used in `SELECT` and other SQL statements with the names as stored. Beginning with NDB 8.0.20, such comparisons are now always performed in a case-insensitive fashion, regardless of the value of `lower_case_table_names`.

- **Multiple transporters.** NDB 8.0.20 introduces support for multiple transporters to handle node-to-node communication between pairs of data nodes. This facilitates higher rates of update operations for each node group in the cluster, and helps avoid constraints imposed by system or other limitations on inter-node communications using a single socket.

  By default, `NDB` now uses a number of transporters based on the number of local data management (LDM) threads or the number of transaction coordinator (TC) threads, whichever is greater. By default, the number of transporters is equal to half of this number. While the default should perform well for most workloads, it is possible to adjust the number of transporters employed by each node group by setting the `NodeGroupTransporters` data node configuration parameter (also introduced in NDB 8.0.20), up a maximum of the greater of the number of LDM threads or the number of TC threads. Setting it to 0 causes the number of transporters to be the same as the number of LDM threads.

- **ndb_restore: primary key schema changes.** NDB 8.0.21 (and later) supports different primary key definitions for source and target tables when restoring an `NDB` native backup with `ndb_restore` when it is run with the `--allow-pk-changes` option. Both increasing and decreasing the number of columns making up the original primary key are supported.

  When the primary key is extended with an additional column or columns, any columns added must be defined as `NOT NULL`, and no values in any such columns may be changed during the time that the backup is being taken. Because some applications set all column values in a row when updating it, whether or not all values are actually changed, this can cause a restore operation to fail even if no values in the column to be added to the primary key have changed. You can override this behavior using the `--ignore-extended-pk-updates` option also added in NDB 8.0.21; in this case, you must ensure that no such values are changed.

  A column can be removed from the table's primary key whether or not this column remains part of the table.

  For more information, see the description of the `--allow-pk-changes` option for `ndb_restore`.

- **Merging backups with ndb_restore.** In some cases, it may be desirable to consolidate data originally stored in different instances of NDB Cluster (all using the same schema) into a single target NDB Cluster. This is now supported when using backups created in the `ndb_mgm` client (see Section 23.6.8.2, “Using The NDB Cluster Management Client to Create a Backup”) and restoring them with `ndb_restore`, using the `--remap-column` option added in NDB 8.0.21 along with `--restore-data` (and possibly additional compatible options as needed or desired). `--remap-column` can be employed to handle cases in which primary and unique key values are overlapping between source clusters, and it is necessary that they do not overlap in the target cluster, as well as to preserve other relationships between tables such as foreign keys.

  `--remap-column` takes as its argument a string having the format *`db.tbl.col:fn:args`*, where *`db`*, *`tbl`*, and *`col`* are, respectively, the names of the database, table, and column, *`fn`* is the name of a remapping function, and *`args`* is one or more arguments to *`fn`*. There is no default value. Only `offset` is supported as the function name, with *`args`* as the integer offset to be applied to the value of the column when inserting it into the target table from the backup. This column must be one of `INT` or `BIGINT`; the allowed range of the offset value is the same as the signed version of that type (this allows the offset to be negative if desired).

  The new option can be used multiple times in the same invocation of `ndb_restore`, so that you can remap to new values multiple columns of the same table, different tables, or both. The offset value does not have to be the same for all instances of the option.

  In addition, two new options are provided for `ndb_desc`, also beginning in NDB 8.0.21:

- `--auto-inc` (short form `-a`): Includes the next auto-increment value in the output, if the table has an `AUTO_INCREMENT` column.
- `--context` (short form `-x`): Provides extra information about the table, including the schema, database name, table name, and internal ID.

For more information and examples, see the description of the `--remap-column` option.

- **Send thread improvements.** As of NDB 8.0.20, each send thread now handles sends to a subset of transporters, and each block thread now assists only one send thread, resulting in more send threads, and thus better performance and data node scalability.
- **Adaptive spin control using SpinMethod.** A simple interface for setting up adaptive CPU spin on platforms supporting it, using the `SpinMethod` data node parameter. This parameter (added in NDB 8.0.20, functional beginning with NDB 8.0.24) has four settings, one each for static spinning, cost-based adaptive spinning, latency-optimized adaptive spinning, and adaptive spinning optimized for database machines on which each thread has its own CPU. Each of these settings causes the data node to use a set of predetermined values for one or more spin parameters which enable adaptive spinning, set spin timing, and set spin overhead, as appropriate to a given scenario, thus obviating the need to set these directly for common use cases.

  For fine-tuning spin behavior, it is also possible to set these and additional spin parameters directly, using the existing `SchedulerSpinTimer` data node configuration parameter as well as the following `DUMP` commands in the `ndb_mgm` client:

  - `DUMP 104000 (SetSchedulerSpinTimerAll)`: Sets spin time for all threads
  - `DUMP 104001 (SetSchedulerSpinTimerThread)`: Sets spin time for a specified thread
  - `DUMP 104002 (SetAllowedSpinOverhead)`: Sets spin overhead as the number of units of CPU time allowed to gain 1 unit of latency
  - `DUMP 104003 (SetSpintimePerCall)`: Sets the time for a call to spin
  - `DUMP 104004 (EnableAdaptiveSpinning)`: Enables or disables adaptive spinning

  NDB 8.0.20 also adds a new TCP configuration parameter `TcpSpinTime` which sets the time to spin for a given TCP connection.

  The `ndb_top` tool is also enhanced to provide spin time information per thread.

  For additional information, see the description of the `SpinMethod` parameter, the listed `DUMP` commands, and Section 23.5.29, “ndb_top — View CPU usage information for NDB threads”.
- **Disk Data and cluster restarts.** Beginning with NDB 8.0.21, an initial restart of the cluster forces the removal of all Disk Data objects such as tablespaces and log file groups, including any data files and undo log files associated with these objects.

  See Section 23.6.11, “NDB Cluster Disk Data Tables”, for more information.
- **Disk Data extent allocation.** Beginning with NDB 8.0.20, allocation of extents in data files is done in a round-robin fashion among all data files used by a given tablespace. This is expected to improve distribution of data in cases where multiple storage devices are used for Disk Data storage.

  For more information, see Section 23.6.11.1, “NDB Cluster Disk Data Objects”.
- **--ndb-log-fail-terminate option.** Beginning with NDB 8.0.21, you can cause the SQL node to terminate whenever it is unable to log all row events fully. This can be done by starting `mysqld` with the `--ndb-log-fail-terminate` option.

- **AllowUnresolvedHostNames parameter.** By default, a management node refuses to start when it cannot resolve a host name present in the global configuration file, which can be problematic in some environments such as Kubernetes. Beginning with NDB 8.0.22, it is possible to override this behavior by setting `AllowUnresolvedHostNames` to `true` in the `[tcp default]` section of the cluster global configuration file (`config.ini` file). Doing so causes such errors to be treated as warnings instead, and to permit `ndb_mgmd` to continue starting

- **Blob write performance enhancements.** NDB 8.0.22 implements a number of improvements which allow more efficient batching when modifying multiple blob columns in the same row, or when modifying multiple rows containing blob columns in the same statement, by reducing the number of round trips required between an SQL or other API node and the data nodes when applying these modifications. The performance of many `INSERT`, `UPDATE`, and `DELETE` statements can thus be improved. Examples of such statements are listed here, where *`table`* is an `NDB` table containing one or more Blob columns:

  - `INSERT INTO table VALUES ROW(1, blob_value1, blob_value2, ...)`, that is, insertion of a row containing one or more Blob columns

  - `INSERT INTO table VALUES ROW(1, blob_value1), ROW(2, blob_value2), ROW(3, blob_value3), ...`, that is, insertion of multiple rows containing one or more Blob columns

  - `UPDATE table SET blob_column1 = blob_value1, blob_column2 = blob_value2, ...`

  - `UPDATE table SET blob_column = blob_value WHERE primary_key_column in (value_list)`, where the primary key column is not a Blob type

  - `DELETE FROM table WHERE primary_key_column = value`, where the primary key column is not a Blob type

  - `DELETE FROM table WHERE primary_key_column IN (value_list)`, where the primary key column is not a Blob type

  Other SQL statements may benefit from these improvements as well. These include `LOAD DATA INFILE` and `CREATE TABLE ... SELECT ...`. In addition, `ALTER TABLE table ENGINE = NDB`, where *`table`* uses a storage engine other than `NDB` prior to execution of the statement, may also execute more efficiently.

  This enhancement applies to statements affecting columns of MySQL type `BLOB`, `MEDIUMBLOB`, `LONGBLOB`, `TEXT`, `MEDIUMTEXT`, and `LONGTEXT`. Statements which update `TINYBLOB` or `TINYTEXT` columns (or both types) only are not affected by this work, and no changes in their performance should be expected.

  The performance of some SQL statements is not noticeably improved by this enhancement, due to the fact that they require scans of table Blob columns, which breaks up batching. Such statements include those of the types listed here:

  - `SELECT FROM table [WHERE key_column IN (blob_value_list)]`, where rows are selected by matching on a primary key or unique key column which uses a Blob type

  - `UPDATE table SET blob_column = blob_value WHERE condition`, using a *`condition`* which does not depend on a unique value

  - `DELETE FROM table WHERE condition` to delete rows containing one or more Blob columns, using a *`condition`* which does not depend on a unique value

- A copying `ALTER TABLE` statement on a table which already used the `NDB` storage engine prior to executing the statement, and whose rows contain one or more Blob columns before or after the statement is executed (or both)

To take advantage of this improvement to its fullest extent, you may wish to increase the values used for the `--ndb-batch-size` and `--ndb-blob-write-batch-bytes` options for `mysqld`, to minimize the number of round trips required to modify blobs. For replication, it is also recommended that you enable the `slave_allow_batching` system variable, which minimizes the number of round trips required by the replica cluster to apply epoch transactions.

Note

Beginning with NDB 8.0.30, you should also use `ndb_replica_batch_size` instead of `--ndb-batch-size`, and `ndb_replica_blob_write_batch_bytes` rather than `--ndb-blob-write-batch-bytes`. See the descriptions of these variables, as well as Section 23.7.5, “Preparing the NDB Cluster for Replication”, for more information.

- **Node.js update.** Beginning with NDB 8.0.22, the `NDB` adapter for Node.js is built using version 12.18.3, and only that version (or a later version of Node.js) is now supported.

- **Encrypted backups.** NDB 8.0.22 adds support for backup files encrypted using AES-256-CBC; this is intended to protect against recovery of data from backups that have been accessed by unauthorized parties. When encrypted, backup data is protected by a user-supplied password. The password can be any string consisting of up to 256 characters from the range of printable ASCII characters other than `!`, `'`, `"`, `$`, `%`, `\`, and `^`. Retention of the password used to encrypt any given NDB Cluster backup must be performed by the user or application; `NDB` does not save the password. The password can be empty, although this is not recommended.

  When taking an NDB Cluster backup, you can encrypt it by using `ENCRYPT PASSWORD=`*`password`* with the management client `START BACKUP` command. Users of the MGM API can also initiate an encrypted backup by calling `ndb_mgm_start_backup4()`.

  You can encrypt existing backup files using the `ndbxfrm` utility which is added to the NDB Cluster distribution in the 8.0.22 release; this program can also be employed for decrypting encrypted backup files. In addition, `ndbxfrm` can compress backup files and decompress compressed backup files using the same method that is employed by NDB Cluster for creating backups when the `CompressedBackup` configuration parameter is set to 1.

  To restore from an encrypted backup, use `ndb_restore` with the options `--decrypt` and `--backup-password`. Both options are required, along with any others that would be needed to restore the same backup if it were not encrypted. `ndb_print_backup_file` and `ndbxfrm` can also read encrypted files using, respectively, `-P` *`password`* and `--decrypt-password=`*`password`*.

  In all cases in which a password is supplied together with an option for encryption or decryption, the password must be quoted; you can use either single or double quotation marks to delimit the password.

  Beginning with NDB 8.0.24, several `NDB` programs, listed here, also support input of the password from standard input, similarly to how this is done when logging in interactively with the `mysql` client using the `--password` option (without including the password on the command line):

  - For `ndb_restore` and `ndb_print_backup_file`, the `--backup-password-from-stdin` option enables input of the password in a secure fashion, similar to how it is done by the `mysql` client' `--password` option. For `ndb_restore`, use the option together with the `--decrypt` option; for `ndb_print_backup_file`, use the option in place of the `-P` option.

- For `ndb_mgm` the option `--backup-password-from-stdin`, is supported together with `--execute "START BACKUP [options]"` for starting a cluster backup from the system shell.
- Two `ndbxfrm` options, `--encrypt-password-from-stdin` and `--decrypt-password-from-stdin`, cause similar behavior when using that program to encrypt or to decrypt a backup file.

See the descriptions of the programs just listed for more information.

It is also possible, beginning with NDB 8.0.22, to enforce encryption of backups by setting `RequireEncryptedBackup=1` in the `[ndbd default]` section of the cluster global configuration file. When this is done, the `ndb_mgm` client rejects any attempt to perform a backup that is not encrypted.

Beginning with NDB 8.0.24, you can cause `ndb_mgm` to use encryption whenever it creates a backup by starting it with `--encrypt-backup`. In this case, the user is prompted for a password when invoking `START BACKUP` if none is supplied.

- **IPv6 support.** Beginning with NDB 8.0.22, IPv6 addressing is supported for connections to management and data nodes; this includes connections between management and data nodes with SQL nodes. When configuring a cluster, you can use numeric IPv6 addresses, host names which resolve to IPv6 addresses or both.

  For IPv6 addressing to work, the operating platform and network on which the cluster is deployed must support IPv6. As when using IPv4 addressing, hostname resolution to IPv6 addresses must be provided by the operating platform.

  A known issue on Linux platforms when running NDB 8.0.22 and later is that the OS kernel must provide IPv6 support, even if the cluster does not use any IPv6 addresses. If you do not need and wish to disable systemwide support for IPv6, do so only after booting the system, like this:

  ```
  $> sysctl -w net.ipv6.conf.all.disable_ipv6=1
  $> sysctl -w net.ipv6.conf.default.disable_ipv6=1
  ```

  Alternatively, you can add the corresponding lines to `/etc/sysctl.conf`.

  IPv4 addressing continues to be supported by `NDB`. Using IPv4 and IPv6 addresses concurrently is not recommended, but can be made to work in the following cases:

  - When the management node is configured with IPv6 and data nodes are configured with IPv4 addresses in the `config.ini` file: This works if `--bind-address` is not used with `mgmd`, and data nodes are started with `--ndb-connectstring` set to the IPv4 address of the management nodes.
  - When the management node is configured with IPv4 and data nodes are configured with IPv6 addresses in `config.ini`: Similarly to the other case, this works if `--bind-address` is not passed to `mgmd` and data nodes are started with `--ndb-connectstring` set to the IPv6 address of the management node.

  These cases work because `ndb_mgmd` does not bind to any IP address by default.

  To perform an upgrade from a version of `NDB` that does not support IPv6 addressing to one that does, provided that the network supports IPv4 and IPv6, first perform the software upgrade; after this has been done, you can update IPv4 addresses used in the `config.ini` file with IPv6 addresses. After this, to cause the configuration changes to take effect and to make the cluster start using the IPv6 addresses, it is necessary to perform a system restart of the cluster.

- **Auto-Installer deprecation and removal.** The MySQL NDB Cluster Auto-Installer web-based installation tool (`ndb_setup.py`) is deprecated in NDB 8.0.22, and is removed in NDB 8.0.23 and later. It is no longer supported.

- **ndbmemcache deprecation and removal.** `ndbmemcache` is no longer supported. `ndbmemcache` was deprecated in NDB 8.0.22, and removed in NDB 8.0.23.

- **ndbinfo backup_id table.** NDB 8.0.24 adds a `backup_id` table to the `ndbinfo` information database. This is intended to serve as a replacement for obtaining this information by using `ndb_select_all` to dump the contents of the internal `SYSTAB_0` table, which is error-prone and takes an excessively long time to perform.

  This table has a single column and row containing the ID of the most recent backup of the cluster taken using the `START BACKUP` management client command. In the event that no backup of this cluster can be found, the table contains a single row whose column value is `0`.

- **Table partitioning enhancements.** NDB 8.0.23 introduces a new method for handling table partitions and fragments, which can determine the number of local data managers (LDMs) for a given data node independently of the number of redo log parts. This means that the number of LDMs can now be highly variable. `NDB` can employ this method when the `ClassicFragmentation` data node configuration parameter, also implemented in NDB 8.0.23, is set to `false`; when this is the case, the number of LDMs is no longer used to determine how many partitions to create for a table per data node, and the value of the `PartitionsPerNode` parameter (also introduced in NDB 8.0.23) determines this number instead, which is also used for calculating the number of fragments used for a table.

  When `ClassicFragmentation` has its default value `true`, then the traditional method of using the number of LDMs is used to determine the number of fragments that a table should have.

  For more information, see the descriptions of the new parameters referenced previously, in Multi-Threading Configuration Parameters (ndbmtd).

- **Terminology updates.** To align with work begun in MySQL 8.0.21 and NDB 8.0.21, NDB 8.0.23 implements a number of changes in terminology, listed here:

  - The system variable `ndb_slave_conflict_role` is now deprecated. It is replaced by `ndb_conflict_role`.

  - Many `NDB` status variables are deprecated. These variables, and their replacements, are shown in the following table:

    Table 23.1 Deprecated NDB status variables and their replacements

Deprecated variable	Replacement
`Ndb_api_adaptive_send_deferred_count_slave`	`Ndb_api_adaptive_send_deferred_count_repli`
`Ndb_api_adaptive_send_forced_count_slave`	`Ndb_api_adaptive_send_forced_count_replica`
`Ndb_api_adaptive_send_unforced_count_slave`	`Ndb_api_adaptive_send_unforced_count_repli`
`Ndb_api_bytes_received_count_slave`	`Ndb_api_bytes_received_count_replica`
`Ndb_api_bytes_sent_count_slave`	`Ndb_api_bytes_sent_count_replica`
`Ndb_api_pk_op_count_slave`	`Ndb_api_pk_op_count_replica`
`Ndb_api_pruned_scan_count_slave`	`Ndb_api_pruned_scan_count_replica`
`Ndb_api_range_scan_count_slave`	`Ndb_api_range_scan_count_replica`
`Ndb_api_read_row_count_slave`	`Ndb_api_read_row_count_replica`
`Ndb_api_scan_batch_count_slave`	`Ndb_api_scan_batch_count_replica`
`Ndb_api_table_scan_count_slave`	`Ndb_api_table_scan_count_replica`
`Ndb_api_trans_abort_count_slave`	`Ndb_api_trans_abort_count_replica`
`Ndb_api_trans_close_count_slave`	`Ndb_api_trans_close_count_replica`
`Ndb_api_trans_commit_count_slave`	`Ndb_api_trans_commit_count_replica`

Deprecated variable	Replacement
Ndb_api_trans_local_read_row_count_slave	Ndb_api_trans_local_read_row_count_replica
Ndb_api_trans_start_count_slave	Ndb_api_trans_start_count_replica
Ndb_api_uk_op_count_slave	Ndb_api_uk_op_count_replica
Ndb_api_wait_exec_complete_count_slave	Ndb_api_wait_exec_complete_count_replica
Ndb_api_wait_meta_request_count_slave	Ndb_api_wait_meta_request_count_replica
Ndb_api_wait_nanos_count_slave	Ndb_api_wait_nanos_count_replica
Ndb_api_wait_scan_result_count_slave	Ndb_api_wait_scan_result_count_replica
Ndb_slave_max_replicated_epoch	Ndb_replica_max_replicated_epoch

The deprecated status variables continue to be shown in the output of SHOW STATUS, but applications should be updated as soon as possible not to rely upon them any longer, since their availability in future release series is not guaranteed.

- The values ADD_TABLE_MASTER and ADD_TABLE_SLAVE previously shown in the tab_copy_status column of the ndbinfo ndbinfo.table_distribution_status table are deprecated. These are replaced by, respectively, the values ADD_TABLE_COORDINATOR and ADD_TABLE_PARTICIPANT.

- The --help output of some NDB client and utility programs such as ndb_restore has been modified.

- **ThreadConfig enhancements.** As of NDB 8.0.23, the configurability of the ThreadConfig parameter has been extended with two new thread types, listed here:

  - query: A query thread works (only) on READ COMMITTED queries. A query thread also acts as a recovery thread. The number of query threads must be 0, 1, 2, or 3 times the number of LDM threads. 0 (the default, unless using ThreadConfig, or AutomaticThreadConfig is enabled) causes LDMs to behave as they did prior to NDB 8.0.23.

- `recover`: A recovery thread retrieves data from a local checkpoint. A recovery thread specified as such never acts as a query thread.

It is also possible to combine the existing `main` and `rep` threads in either of two ways:

- Into a single thread by setting either one of these arguments to 0. When this is done, the resulting combined thread is shown with the name `main_rep` in the `ndbinfo.threads` table.
- Together with the `recv` thread by setting both `ldm` and `tc` to 0, and setting `recv` to 1. In this case, the combined thread is named `main_rep_recv`.

In addition, the maximum numbers of a number of existing thread types have been increased. The new maximums, including those for query threads and recovery threads, are listed here:

- LDM: 332
- Query: 332
- Recovery: 332
- TC: 128
- Receive: 64
- Send: 64
- Main: 2

Maximums for other thread types remain unchanged.

Also, as the result of work done relating to this task, `NDB` now employs mutexes to protect job buffers when using more than 32 block threads. While this can cause a slight decrease in performance (1 to 2 percent in most cases), it also significantly reduces the amount of memory required by very large configurations. For example, a setup with 64 threads which used 2 GB of job buffer memory prior to NDB 8.0.23 should require only about 1 GB instead in NDB 8.0.23 and later. In our testing this has resulted in an overall improvement on the order of 5 percent in the execution of very complex queries.

For further information, see the descriptions of the `ThreadConfig` parameter and the `ndbinfo.threads` table.

- **ThreadConfig thread count changes.** As the result of work done in NDB 8.0.30, setting the value of `ThreadConfig` requires including `main`, `rep`, `recv`, and `ldm` in the `ThreadConfig` value string explicitly, in this and subsequent NDB Cluster releases. In addition, `count=0` must be set explicitly for each thread type (of `main`, `rep`, or `ldm`) that is not to be used, and setting `count=1` for replication threads (`rep`) requires also setting `count=1` for `main`.

  These changes can have a significant impact on upgrades of NDB clusters where this parameter is in use; see Section 23.3.7, “Upgrading and Downgrading NDB Cluster”, for more information.

- **ndbmtd Thread Auto-Configuration.** Beginning with NDB 8.0.23, it is possible to employ automatic configuration of threads for multi-threaded data nodes using the `ndbmtd` configuration parameter `AutomaticThreadConfig`. When this parameter is set to 1, `NDB` sets up thread assignments automatically, based on the number of processors available to applications, for all thread supported thread types, including the new `query` and `recover` thread types described in the previous item. If the system does not limit the number of processors, you can do so if

desired by setting `NumCPUs` (also added in NDB 8.0.23). Otherwise, automatic thread configuration accommodates up to 1024 CPUs.

Automatic thread configuration occurs regardless of any values set for `ThreadConfig` or `MaxNoOfExecutionThreads` in `config.ini`; this means that it is not necessary to set either of these parameters.

In addition, NDB 8.0.23 implements a number of new `ndbinfo` information database tables providing information about hardware and CPU availability, as well as CPU usage by `NDB` data nodes. These tables are listed here:

- `cpudata`
- `cpudata_1sec`
- `cpudata_20sec`
- `cpudata_50ms`
- `cpuinfo`
- `hwinfo`

Some of these tables are not available on every platform supported by NDB Cluster; see the individual descriptions of them for more information.

- **Hierarchical views of NDB database objects.** The `dict_obj_tree` table, added to the `ndbinfo` information database in NDB 8.0.24, can provide hierarchical and tree-like views of many `NDB` database objects, including the following:

  - Tables and associated indexes
  - Tablespaces and associated data files
  - Logfile groups and associated undo log files

  For more information and examples, see Section 23.6.16.25, “The ndbinfo dict_obj_tree Table”.

- **Index statistics enhancements.** NDB 8.0.24 implements the following improvements in calculation of index statistics:

  - Index statistics were previously collected from one fragment only; this is changed such that this extrapolation is extended to additional fragments.
  - The algorithm used for very small tables, such as those having very few rows where results are discarded, has been improved, so that estimates for such tables should be more accurate than previously.

  As of NDB 8.0.27, the index statistics tables are created and updated automatically by default, `IndexStatAutoCreate` and `IndexStatAutoUpdate` both default to `1` (enabled) rather than `0` (disabled), and it is no longer necessary to run `ANALYZE TABLE` to update the statistics.

  For additional information, see Section 23.6.15, “NDB API Statistics Counters and Variables”.

- **Conversion between NULL and NOT NULL during restore operations.** Beginning with NDB 8.0.26, `ndb_restore` can support restoring of `NULL` columns as `NOT NULL` and the reverse, using the options listed here:

  - To restore a `NULL` column as `NOT NULL`, use the `--lossy-conversions` option.

    The column originally declared as `NULL` must not contain any `NULL` rows; if it does, `ndb_restore` exits with an error.

- To restore a `NOT NULL` column as `NULL`, use the `--promote-attributes` option.

For more information, see the descriptions of the indicated `ndb_restore` options.

- **SQL-compliant NULL comparison mode for NdbScanFilter.** Traditionally, when making comparisons involving `NULL`, `NdbScanFilter` treats `NULL` as equal to `NULL` (and thus considers `NULL == NULL` to be `TRUE`). This is not the same as specified by the SQL Standard, which requires that any comparison with `NULL` return `NULL`, including `NULL == NULL`.

  Previously, it was not possible for an NDB API application to override this behavior; beginning with NDB 8.0.26, you can do so by calling `NdbScanFilter::setSqlCmpSemantics()` prior to creating a scan filter. (Thus, this method is always invoked as a class method and not as an instance method.) Doing so causes the next `NdbScanFilter` object to be created to employ SQL-compliant `NULL` comparison for all comparison operations performed over the lifetime of the instance. You must invoke the method for each `NdbScanFilter` object that should use SQL-compliant comparisons.

  For more information, see NdbScanFilter::setSqlCmpSemantics().

- **Deprecation of NDB API .FRM file methods.** MySQL 8.0 and NDB 8.0 no longer use `.FRM` files for storing table metadata. For this reason, the NDB API methods `getFrmData()`, `getFrmLength()`, and `setFrm()` are deprecated as of NDB 8.0.27, and subject to removal in a future release. For reading and writing table metadata, use `getExtraMetadata()` and `setExtraMetadata()` instead.

- **Preference for IPv4 or IPv6 addressing.** NDB 8.0.26 adds the `PreferIPVersion` configuration parameter, which controls the addressing preference for DNS resolution. IPv4 (`PreferIPVersion=4`) is the default. Because configuration retrieval in NDB requires that this preference be the same for all TCP connections, you should set it only in the `[tcp default]` section of the cluster global configuration (`config.ini`) file.

  See Section 23.4.3.10, “NDB Cluster TCP/IP Connections”, for more information.

- **Logging enhancements.** Previously, analysis of NDB Cluster data node and management node logs could be hampered by the fact that different log messages used different formats, and that not all log messages included timestamps. Such issues were due in part to the fact that logging was performed by a number of different mechanisms, such as the functions `printf`, `fprintf`, `ndbout`, and `ndbout_c`, overloading of the `<<` operator, and so on.

  We fix these problems by standardizing on the `EventLogger` mechanism, which is already present in `NDB`, and which begins each log message with a timestamp in `YYYY-MM-DD HH:MM:SS` format.

  See Section 23.6.3, “Event Reports Generated in NDB Cluster”, for more information about NDB Cluster event logs and the `EventLogger` log message format.

- **Copying ALTER TABLE improvements.** Beginning with NDB 8.0.27, a copying `ALTER TABLE` on an `NDB` table compares the fragment commit counts for the source table before and after performing the copy. This allows the SQL node executing this statement to determine whether there has been any concurrent write activity to the table being altered; if so, the SQL node can then terminate the operation.

  When concurrent writes are detected being made to the table being altered, the `ALTER TABLE` statement is rejected with the error `Detected change to data in source table during copying ALTER TABLE. Alter aborted to avoid inconsistency` (`ER_TABLE_DEF_CHANGED`). Stopping the alter operation, rather than allowing it to proceed with concurrent writes taking place, can help prevent silent data loss or corruption.

- **ndbinfo index_stats table.** NDB 8.0.28 adds the `index_stats` table, which provides basic information about NDB index statistics. It is intended primarily for internal testing, but may be useful as a supplement to `ndb_index_stat`.

- **ndb_import --table option.** Prior to NDB 8.0.28, `ndb_import` always imported the data read from a CSV file into a table whose name was derived from the name of the file being read. NDB 8.0.28 adds a `--table` option (short form: `-t`) for this program to specify the name of the target table directly, and override the previous behavior.

  The default behavior for `ndb_import` remains to use the base name of the input file as the name of the target table.

- **ndb_import --missing-ai-column option.** Beginning with NDB 8.0.29, `ndb_import` can import data from a CSV file that contains empty values for an `AUTO_INCREMENT` column, using the `--missing-ai-column` option introduced in that release. The option can be used with one or more tables containing such a column.

  In order for this option to work, the `AUTO_INCREMENT` column in the CSV file must not contain any values. Otherwise, the import operation cannot proceed.

- **ndb_import and empty lines.** `ndb_import` has always rejected any empty lines encountered in an incoming CSV file. NDB 8.0.30 adds support for importing empty lines into a single column, provided that it is possible to convert the empty value into a column value.

- **ndb_restore --with-apply-status option.** Beginning with NDB 8.0.29, it is possible to restore the `ndb_apply_status` table from an `NDB` backup, using `ndb_restore` with the `--with-apply-status` option added in that release. To use this option, you must also use `--restore-data` when invoking `ndb_restore`.

  `--with-apply-status` restores all rows of the `ndb_apply_status` table except for the row having `server_id = 0`; to restore this row, use `--restore-epoch`. For more information, see ndb_apply_status Table, as the description of the `--with-apply-status` option.

- **SQL access to tables with missing indexes.** Prior to NDB 8.0.29, when a user query attempted to open an `NDB` table with a missing or broken index, the MySQL server raised `NDB` error `4243` (`Index not found`). This situation could arise when constraint violations or missing data make it impossible to restore an index on an `NDB` table, and `ndb_restore --disable-indexes` was used to restore the data without the index.

  Beginning with NDB 8.0.29, an SQL query against an `NDB` table which has missing indexes succeeds if the query does not use any of the missing indexes. Otherwise, the query is rejected with `ER_NOT_KEYFILE`. In this case, you can use `ALTER TABLE ... ALTER INDEX ... INVISIBLE` to keep the MySQL Optimizer from trying to use the index, or drop the index (and then possibly re-create it) using the appropriate SQL statements.

- **NDB API List::clear() method.** The NDB API `Dictionary` methods `listEvents()`, `listIndexes()`, and `listObjects()` each require a reference to a `List` object which is empty. Previously, reusing an existing `List` with any of these methods was problematic for this reason. NDB 8.0.29 makes this easier by implementing a `clear()` method which removes all data from the list.

  As part of this work, the `List` class destructor now calls `List::clear()` before removing any elements or attributes from the list.

- **NDB dictionary tables in ndbinfo.** NDB 8.0.29 introduces several new tables in the `ndbinfo` database providing information from `NdbDictionary` that previously required the use of `ndb_desc`, `ndb_select_all`, and other `NDB` utility programs.

  Two of these tables are actually views. The `hash_maps` table provides information about hash maps used by `NDB`; the `files` table shows information regarding files used for storing data on disk (see Section 23.6.11, “NDB Cluster Disk Data Tables”).

  The remaining six `ndbinfo` tables added in NDB 8.0.29 are base tables. These tables are not hidden and are not named using the prefix `ndb$`. These tables are listed here, with descriptions of the objects represented in each table:

- `blobs`: Blob tables used to store the variable-size parts of `BLOB` and `TEXT` columns
- `dictionary_columns`: Columns of `NDB` tables
- `dictionary_tables`: `NDB` tables
- `events`: Event subscriptions in the NDB API
- `foreign_keys`: Foreign keys on `NDB` tables
- `index_columns`: Indexes on `NDB` tables

NDB 8.0.29 also makes changes in the `ndbinfo` storage engine's implementation of primary keys to improve compatibility with `NdbDictionary`.

- **ndbcluster plugin and Performance Schema.** As of NDB 8.0.29, `ndbcluster` plugin threads are shown in the Performance Schema `threads` and `setup_threads` tables, making it possible to obtain information about the performance of these threads. The three threads exposed in `performance_schema` tables are listed here:

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

  See ndbcluster Plugin Threads, for more information and examples.

  In NDB 8.0.30 and later, transaction batching memory usage is visible as `memory/ndbcluster/Thd_ndb::batch_mem_root` in the Performance Schema `memory_summary_by_thread_by_event_name` and `setup_instruments` tables. You can use this information to see how much memory is being used by transactions. For additional information, see Transaction Memory Usage.

- **Configurable blob inline size.** Beginning with NDB 8.0.30, it is possible to set a blob column's inline size as part of `CREATE TABLE` or `ALTER TABLE`. The maximum inline size supported by NDB Cluster is 29980 bytes.

  For additional information and examples, see NDB_COLUMN Options, as well as String Type Storage Requirements.

- **replica_allow_batching enabled by default.** Replica write batching improves NDB Cluster Replication performance greatly, especially when replicating blob-type columns (`TEXT`, `BLOB`, and `JSON`), and so generally should be enabled whenever using replication with NDB Cluster. For this reason, beginning with NDB 8.0.30, the `replica_allow_batching` system variable is enabled by default, and setting it to `OFF` raises a warning.

- **Conflict resolution insert operation support.** Prior to NDB 8.0.30, there were only two strategies available for resolving primary key conflicts for update and delete operations, implemented as the functions `NDB$MAX()` and `NDB$MAX_DELETE_WIN()`. Neither of these has any effect on write operations, other than that a write operation with the same primary key as a previous write is always rejected, and accepted and applied only if no operation having the same primary key already exists. NDB 8.0.30 introduces two new conflict resolution functions `NDB$MAX_INS()` and `NDB$MAX_DEL_WIN_INS()` that handle primary key conflicts between insert operations. These functions handle conflicting writes 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.

For conflicting update and delete operations, `NDB$MAX_INS()` behaves as `NDB$MAX()` does, and `NDB$MAX_DEL_WIN_INS()` behaves in the same way as `NDB$MAX_DELETE_WIN()`.

This enhancement provides support for configuring conflict detection when handling conflicting replicated write operations, so that a replicated `INSERT` with a higher timestamp column value is applied idempotently, while a replicated `INSERT` with a lower timestamp column value is rejected.

As with the other conflict resolution functions, rejected operations can optionally be logged in an exceptions table; rejected operations increment a counter (status variables `Ndb_conflict_fn_max` for “greatest timestamp wins” and `Ndb_conflict_fn_old` for “same timestamp wins”).

For more information, see the descriptions of the new conflict resolution functions, and as well as Section 23.7.12, “NDB Cluster Replication Conflict Resolution”.

- **Replication applier batch size control.** Previously, the size of batches used when writing to a replica NDB Cluster was controlled by `--ndb-batch-size`, and the batch size used for writing blob data to the replica was determined by `ndb-blob-write-batch-bytes`. One problem with this arrangement was that the replica used the global values of these variables which meant that changing either of them for the replica also affected the value used by all other sessions. In addition, it was not possible to set different defaults for these values exclusive to the replica, which should preferably have a higher default value than other sessions.

  NDB 8.0.30 adds two new system variables which are specific to the replica applier. `ndb_replica_batch_size` now controls the batch size used for the replica applier, and `ndb_replica_blob_write_batch_bytes` variable now determines the blob write batch size used to perform batch blob writes on the replica.

  This change should improve the behavior of MySQL NDB Cluster Replication using default settings, and lets the user fine tune NDB replication performance without affecting user threads, such as those performing processing of SQL queries.

  For more information, see the descriptions of the new variables. See also Section 23.7.5, “Preparing the NDB Cluster for Replication”.

- **Binary Log Transaction Compression.** NDB 8.0.31 adds support for binary logs using compressed transactions with `ZSTD` compression. To enable this feature, set the `ndb_log_transaction_compression` system variable introduced in this release to `ON`. The level of compression used can be controlled using the `ndb_log_transaction_compression_level_zstd` system variable, which is also added in that release; the default compression level is 3.

  Although the `binlog_transaction_compression` and `binlog_transaction_compression_level_zstd` server system variables have no effect on binary logging of `NDB` tables, starting `mysqld` with `--binlog-transaction-compression=ON` causes `ndb_log_transaction_compression` to be enabled automatically. You can disable it in a MySQL client session using `SET @@global.ndb_log_transaction_compression=OFF` after server startup has completed.

  See the description of `ndb_log_transaction_compression` as well as Section 5.4.4.5, “Binary Log Transaction Compression”, for more information.

- **NDB Replication: Multithreaded Applier.** As of NDB 8.0.33, NDB Cluster replication supports the MySQL multithreaded applier (MTA) on replica servers (and nonzero values of

`replica_parallel_workers`), which enables the application of binary log transactions in parallel on the replica and thereby increasing throughput. (For more information about the multithreaded applier in the MySQL server, see Section 17.2.3, “Replication Threads”.)

Enabling this feature on the replica requires that the source be started with `--ndb-log-transaction-dependency` set to `ON` (this option is also implemented in NDB 8.0.33). It is also necessary on the source to set `binlog_transaction_dependency_tracking` to `WRITESET`. In addition, you must ensure that `replica_parallel_workers` has a value greater than 1 on the replica, and thus, that the replica uses multiple worker threads.

For additional information and requirements, see Section 23.7.11, “NDB Cluster Replication Using the Multithreaded Applier”.

- **Changes in build options.** NDB 8.0.31 makes the following changes in CMake options used for building MySQL Cluster.

  - The `WITH_NDBCLUSTER` option is deprecated, and `WITH_PLUGIN_NDBCLUSTER` is removed.
  - To build MySQL Cluster from source, use the newly-added `WITH_NDB` option.
  - `WITH_NDBCLUSTER_STORAGE_ENGINE` continues to be supported, but is no longer needed for most builds.

  See CMake Options for Compiling NDB Cluster, for more information.

- **File system encryption.** Transparent Data Encryption (TDE) provides protection by encryption of `NDB` data at rest, that is, of all `NDB` table data and log files which are persisted to disk. This is intended to protect against recovering data after obtaining unauthorized access to NDB Cluster data files such as tablespace files or logs.

  Encryption is implemented transparently by the NDB file system layer (`NDBFS`) on the data nodes; data is encrypted and decrypted as it is read from and written to the file, and `NDBFS` internal client blocks operate on files as normal.

  `NDBFS` can transparently encrypt a file directly from a user provided password, but decoupling the encryption and decryption of individual files from the user provided password can be advantageous for reasons of efficiency, usability, security, and flexibility. See Section 23.6.14.2, “NDB File System Encryption Implementation”.

  TDE uses two types of keys. A secret key is used to encrypt the actual data and log files stored on disk (including LCP, redo, undo, and tablespace files). A master key is then used to encrypt the secret key.

  The `EncryptedFileSystem` data node configuration parameter, available beginning with NDB 8.0.29, when set to `1`, enforces encryption on files storing table data. This includes LCP data files, redo log files, tablespace files, and undo log files.

  It is also necessary to provide a password to each data node when starting or restarting it, using one of the options `--filesystem-password` or `--filesystem-password-from-stdin`. See Section 23.6.14.1, “NDB File System Encryption Setup and Usage”. This password uses the same format and is subject to the same constraints as the password used for an encrypted `NDB` backup (see the description of the `ndb_restore --backup-password` option for details).

  Only tables using the `NDB` storage engine are subject to encryption by this feature; see Section 23.6.14.3, “NDB File System Encryption Limitations”. Other tables, such as those used for `NDB` schema distribution, replication, and binary logging, typically use `InnoDB`; see Section 15.13,

"InnoDB Data-at-Rest Encryption". For information about encryption of binary log files, see Section 17.3.2, "Encrypting Binary Log Files and Relay Log Files".

Files generated or used by `NDB` processes, such as operating system logs, crash logs, and core dumps, are not encrypted. Files used by `NDB` but not containing any user table data are also not encrypted; these include LCP control files, schema files, and system files (see NDB Cluster Data Node File System). The management server configuration cache is also not encrypted.

In addition, NDB 8.0.31 adds a new utility `ndb_secretsfile_reader` for extracting key information from a secrets file (`S0.sysfile`).

This enhancement builds on work done in NDB 8.0.22 to implement encrypted `NDB` backups. For more information about encrypted backups, see the description of the `RequireEncryptedBackup` configuration parameter, as well as Section 23.6.8.2, "Using The NDB Cluster Management Client to Create a Backup".

- **Removal of unneeded program options.** A number of "junk" command-line options for NDB utility and other programs which had never been implemented were removed in NDB Cluster 8.0.31. The options and the programs from which they have been dropped are listed here:

  - `--ndb-optimized-node-selection`:

    `ndbd`, `ndbmtd`, `ndb_mgm`, `ndb_delete_all`, `ndb_desc`, `ndb_drop_index`, `ndb_drop_table`, `ndb_show_table`, `ndb_blob_tool`, `ndb_config`, `ndb_index_stat`, `ndb_move_data`, `ndbinfo_select_all`, `ndb_select_count`

  - `--character-sets-dir`:

    `ndb_mgm`, `ndb_mgmd`, `ndb_config`, `ndb_delete_all`, `ndb_desc`, `ndb_drop_index`, `ndb_drop_table`, `ndb_show_table`, `ndb_blob_tool`, `ndb_config`, `ndb_index_stat`, `ndb_move_data`, `ndbinfo_select_all`, `ndb_select_count`, `ndb_waiter`

  - `--core-file`:

    `ndb_mgm`, `ndb_mgmd`, `ndb_config`, `ndb_delete_all`, `ndb_desc`, `ndb_drop_index`, `ndb_drop_table`, `ndb_show_table`, `ndb_blob_tool`, `ndb_config`, `ndb_index_stat`, `ndb_move_data`, `ndbinfo_select_all`, `ndb_select_count`, `ndb_waiter`

  - `--connect-retries` and `--connect-retry-delay`:

    `ndb_mgmd`

  - `--ndb-nodeid`:

    `ndb_config`

  For more information, see the relevant program and option descriptions in Section 23.5, "NDB Cluster Programs".

- **Reading Configuration Cache Files.** Beginning with NDB 8.0.32, it is possible to read binary configuration cache files created by `ndb_mgmd` using the `ndb_config` option `--config-binary-file` introduced in that release. This can simplify the process of determining whether the settings in a given configuration file have been applied to the cluster, or of recovery of settings from the binary cache after the `config.ini` file has somehow been damaged or lost.

  For more information and examples, see the description of this option in Section 23.5.7, "ndb_config — Extract NDB Cluster Configuration Information".

MySQL Cluster Manager 1.4.8 also provides experimental support for NDB Cluster 8.0. MySQL Cluster Manager has an advanced command-line interface that can simplify many complex NDB Cluster management tasks. See MySQL Cluster Manager 1.4.8 User Manual, for more information.

23.2.5 Options, Variables, and Parameters Added, Deprecated or Removed in NDB 8.0

The next few sections contain information about `NDB` node configuration parameters and NDB-specific `mysqld` options and variables that have been added to, deprecated in, or removed from NDB 8.0.

Parameters Introduced in NDB 8.0

The following node configuration parameters have been added in NDB 8.0.

- `AllowUnresolvedHostNames`: When false (default), failure by management node to resolve host name results in fatal error; when true, unresolved host names are reported as warnings only. Added in NDB 8.0.22.
- `AutomaticThreadConfig`: Use automatic thread configuration; overrides any settings for ThreadConfig and MaxNoOfExecutionThreads, and disables ClassicFragmentation. Added in NDB 8.0.23.
- `ClassicFragmentation`: When true, use traditional table fragmentation; set false to enable flexible distribution of fragments among LDMs. Disabled by AutomaticThreadConfig. Added in NDB 8.0.23.
- `DiskDataUsingSameDisk`: Set to false if Disk Data tablespaces are located on separate physical disks. Added in NDB 8.0.19.
- `EnableMultithreadedBackup`: Enable multi-threaded backup. Added in NDB 8.0.16.
- `EncryptedFileSystem`: Encrypt local checkpoint and tablespace files. EXPERIMENTAL; NOT SUPPORTED IN PRODUCTION. Added in NDB 8.0.29.
- `KeepAliveSendInterval`: Time between keep-alive signals on links between data nodes, in milliseconds. Set to 0 to disable. Added in NDB 8.0.27.
- `MaxDiskDataLatency`: Maximum allowed mean latency of disk access (ms) before starting to abort transactions. Added in NDB 8.0.19.
- `NodeGroupTransporters`: Number of transporters to use between nodes in same node group. Added in NDB 8.0.20.
- `NumCPUs`: Specify number of CPUs to use with AutomaticThreadConfig. Added in NDB 8.0.23.
- `PartitionsPerNode`: Determines the number of table partitions created on each data node; not used if ClassicFragmentation is enabled. Added in NDB 8.0.23.
- `PreferIPVersion`: Indicate DNS resolver preference for IP version 4 or 6. Added in NDB 8.0.26.
- `RequireEncryptedBackup`: Whether backups must be encrypted (1 = encryption required, otherwise 0). Added in NDB 8.0.22.
- `ReservedConcurrentIndexOperations`: Number of simultaneous index operations having dedicated resources on one data node. Added in NDB 8.0.16.

- `ReservedConcurrentOperations`: Number of simultaneous operations having dedicated resources in transaction coordinators on one data node. Added in NDB 8.0.16.
- `ReservedConcurrentScans`: Number of simultaneous scans having dedicated resources on one data node. Added in NDB 8.0.16.
- `ReservedConcurrentTransactions`: Number of simultaneous transactions having dedicated resources on one data node. Added in NDB 8.0.16.
- `ReservedFiredTriggers`: Number of triggers having dedicated resources on one data node. Added in NDB 8.0.16.
- `ReservedLocalScans`: Number of simultaneous fragment scans having dedicated resources on one data node. Added in NDB 8.0.16.
- `ReservedTransactionBufferMemory`: Dynamic buffer space (in bytes) for key and attribute data allocated to each data node. Added in NDB 8.0.16.
- `SpinMethod`: Determines spin method used by data node; see documentation for details. Added in NDB 8.0.20.
- `TcpSpinTime`: Time to spin before going to sleep when receiving. Added in NDB 8.0.20.
- `TransactionMemory`: Memory allocated for transactions on each data node. Added in NDB 8.0.19.

Parameters Deprecated in NDB 8.0

The following node configuration parameters have been deprecated in NDB 8.0.

- `BatchSizePerLocalScan`: Used to calculate number of lock records for scan with hold lock. Deprecated in NDB 8.0.19.
- `MaxAllocate`: No longer used; has no effect. Deprecated in NDB 8.0.27.
- `MaxNoOfConcurrentIndexOperations`: Total number of index operations that can execute simultaneously on one data node. Deprecated in NDB 8.0.19.
- `MaxNoOfConcurrentTransactions`: Maximum number of transactions executing concurrently on this data node, total number of transactions that can be executed concurrently is this value times number of data nodes in cluster. Deprecated in NDB 8.0.19.
- `MaxNoOfFiredTriggers`: Total number of triggers that can fire simultaneously on one data node. Deprecated in NDB 8.0.19.
- `MaxNoOfLocalOperations`: Maximum number of operation records defined on this data node. Deprecated in NDB 8.0.19.
- `MaxNoOfLocalScans`: Maximum number of fragment scans in parallel on this data node. Deprecated in NDB 8.0.19.
- `ReservedTransactionBufferMemory`: Dynamic buffer space (in bytes) for key and attribute data allocated to each data node. Deprecated in NDB 8.0.19.
- `UndoDataBuffer`: Unused; has no effect. Deprecated in NDB 8.0.27.
- `UndoIndexBuffer`: Unused; has no effect. Deprecated in NDB 8.0.27.

Parameters Removed in NDB 8.0

No node configuration parameters have been removed in NDB 8.0.

Options and Variables Introduced in NDB 8.0

The following system variables, status variables, and server options have been added in NDB 8.0.

- `Ndb_api_adaptive_send_deferred_count_replica`: Number of adaptive send calls not actually sent by this replica. Added in NDB 8.0.23.
- `Ndb_api_adaptive_send_forced_count_replica`: Number of adaptive sends with forced-send set sent by this replica. Added in NDB 8.0.23.
- `Ndb_api_adaptive_send_unforced_count_replica`: Number of adaptive sends without forced-send sent by this replica. Added in NDB 8.0.23.
- `Ndb_api_bytes_received_count_replica`: Quantity of data (in bytes) received from data nodes by this replica. Added in NDB 8.0.23.
- `Ndb_api_bytes_sent_count_replica`: Qunatity of data (in bytes) sent to data nodes by this replica. Added in NDB 8.0.23.
- `Ndb_api_pk_op_count_replica`: Number of operations based on or using primary keys by this replica. Added in NDB 8.0.23.
- `Ndb_api_pruned_scan_count_replica`: Number of scans that have been pruned to one partition by this replica. Added in NDB 8.0.23.
- `Ndb_api_range_scan_count_replica`: Number of range scans that have been started by this replica. Added in NDB 8.0.23.
- `Ndb_api_read_row_count_replica`: Total number of rows that have been read by this replica. Added in NDB 8.0.23.
- `Ndb_api_scan_batch_count_replica`: Number of batches of rows received by this replica. Added in NDB 8.0.23.
- `Ndb_api_table_scan_count_replica`: Number of table scans that have been started, including scans of internal tables, by this replica. Added in NDB 8.0.23.
- `Ndb_api_trans_abort_count_replica`: Number of transactions aborted by this replica. Added in NDB 8.0.23.
- `Ndb_api_trans_close_count_replica`: Number of transactions aborted (may be greater than sum of TransCommitCount and TransAbortCount) by this replica. Added in NDB 8.0.23.
- `Ndb_api_trans_commit_count_replica`: Number of transactions committed by this replica. Added in NDB 8.0.23.
- `Ndb_api_trans_local_read_row_count_replica`: Total number of rows that have been read by this replica. Added in NDB 8.0.23.
- `Ndb_api_trans_start_count_replica`: Number of transactions started by this replica. Added in NDB 8.0.23.
- `Ndb_api_uk_op_count_replica`: Number of operations based on or using unique keys by this replica. Added in NDB 8.0.23.
- `Ndb_api_wait_exec_complete_count_replica`: Number of times thread has been blocked while waiting for operation execution to complete by this replica. Added in NDB 8.0.23.
- `Ndb_api_wait_meta_request_count_replica`: Number of times thread has been blocked waiting for metadata-based signal by this replica. Added in NDB 8.0.23.
- `Ndb_api_wait_nanos_count_replica`: Total time (in nanoseconds) spent waiting for some type of signal from data nodes by this replica. Added in NDB 8.0.23.
- `Ndb_api_wait_scan_result_count_replica`: Number of times thread has been blocked while waiting for scan-based signal by this replica. Added in NDB 8.0.23.

- `Ndb_config_generation`: Generation number of the current configuration of the cluster. Added in NDB 8.0.24.
- `Ndb_conflict_fn_max_del_win_ins`: Number of times that NDB replication conflict resolution based on outcome of NDB$MAX_DEL_WIN_INS() has been applied to insert operations. Added in NDB 8.0.30.
- `Ndb_conflict_fn_max_ins`: Number of times that NDB replication conflict resolution based on "greater timestamp wins" has been applied to insert operations. Added in NDB 8.0.30.
- `Ndb_metadata_blacklist_size`: Number of NDB metadata objects that NDB binlog thread has failed to synchronize; renamed in NDB 8.0.22 as Ndb_metadata_excluded_count. Added in NDB 8.0.18.
- `Ndb_metadata_detected_count`: Number of times NDB metadata change monitor thread has detected changes. Added in NDB 8.0.16.
- `Ndb_metadata_excluded_count`: Number of NDB metadata objects that NDB binlog thread has failed to synchronize. Added in NDB 8.0.22.
- `Ndb_metadata_synced_count`: Number of NDB metadata objects which have been synchronized. Added in NDB 8.0.18.
- `Ndb_trans_hint_count_session`: Number of transactions using hints that have been started in this session. Added in NDB 8.0.17.
- `ndb-applier-allow-skip-epoch`: Lets replication applier skip epochs. Added in NDB 8.0.28.
- `ndb-log-fail-terminate`: Terminate mysqld process if complete logging of all found row events is not possible. Added in NDB 8.0.21.
- `ndb-log-transaction-dependency`: Make binary log thread calculate transaction dependencies for every transaction it writes to binary log. Added in NDB 8.0.33.
- `ndb-schema-dist-timeout`: How long to wait before detecting timeout during schema distribution. Added in NDB 8.0.17.
- `ndb_conflict_role`: Role for replica to play in conflict detection and resolution. Value is one of PRIMARY, SECONDARY, PASS, or NONE (default). Can be changed only when replication SQL thread is stopped. See documentation for further information. Added in NDB 8.0.23.
- `ndb_dbg_check_shares`: Check for any lingering shares (debug builds only). Added in NDB 8.0.13.
- `ndb_log_transaction_compression`: Whether to compress NDB binary log; can also be enabled on startup by enabling --binlog-transaction-compression option. Added in NDB 8.0.31.
- `ndb_log_transaction_compression_level_zstd`: The ZSTD compression level to use when writing compressed transactions to the NDB binary log. Added in NDB 8.0.31.
- `ndb_metadata_check`: Enable auto-detection of NDB metadata changes with respect to MySQL data dictionary; enabled by default. Added in NDB 8.0.16.
- `ndb_metadata_check_interval`: Interval in seconds to perform check for NDB metadata changes with respect to MySQL data dictionary. Added in NDB 8.0.16.
- `ndb_metadata_sync`: Triggers immediate synchronization of all changes between NDB dictionary and MySQL data dictionary; causes ndb_metadata_check and ndb_metadata_check_interval values to be ignored. Resets to false when synchronization is complete. Added in NDB 8.0.19.
- `ndb_replica_batch_size`: Batch size in bytes for replica applier. Added in NDB 8.0.30.
- `ndb_schema_dist_lock_wait_timeout`: Time during schema distribution to wait for lock before returning error. Added in NDB 8.0.18.

- `ndb_schema_dist_timeout`: Time to wait before detecting timeout during schema distribution. Added in NDB 8.0.16.
- `ndb_schema_dist_upgrade_allowed`: Allow schema distribution table upgrade when connecting to NDB. Added in NDB 8.0.17.
- `ndbinfo`: Enable ndbinfo plugin, if supported. Added in NDB 8.0.13.
- `replica_allow_batching`: Turns update batching on and off for replica. Added in NDB 8.0.26.

Options and Variables Deprecated in NDB 8.0

The following system variables, status variables, and options have been deprecated in NDB 8.0.

- `Ndb_api_adaptive_send_deferred_count_slave`: Number of adaptive send calls not actually sent by this replica. Deprecated in NDB 8.0.23.
- `Ndb_api_adaptive_send_forced_count_slave`: Number of adaptive sends with forced-send set sent by this replica. Deprecated in NDB 8.0.23.
- `Ndb_api_adaptive_send_unforced_count_slave`: Number of adaptive sends without forced-send sent by this replica. Deprecated in NDB 8.0.23.
- `Ndb_api_bytes_received_count_slave`: Quantity of data (in bytes) received from data nodes by this replica. Deprecated in NDB 8.0.23.
- `Ndb_api_bytes_sent_count_slave`: Qunatity of data (in bytes) sent to data nodes by this replica. Deprecated in NDB 8.0.23.
- `Ndb_api_pk_op_count_slave`: Number of operations based on or using primary keys by this replica. Deprecated in NDB 8.0.23.
- `Ndb_api_pruned_scan_count_slave`: Number of scans that have been pruned to one partition by this replica. Deprecated in NDB 8.0.23.
- `Ndb_api_range_scan_count_slave`: Number of range scans that have been started by this replica. Deprecated in NDB 8.0.23.
- `Ndb_api_read_row_count_slave`: Total number of rows that have been read by this replica. Deprecated in NDB 8.0.23.
- `Ndb_api_scan_batch_count_slave`: Number of batches of rows received by this replica. Deprecated in NDB 8.0.23.
- `Ndb_api_table_scan_count_slave`: Number of table scans that have been started, including scans of internal tables, by this replica. Deprecated in NDB 8.0.23.
- `Ndb_api_trans_abort_count_slave`: Number of transactions aborted by this replica. Deprecated in NDB 8.0.23.
- `Ndb_api_trans_close_count_slave`: Number of transactions aborted (may be greater than sum of TransCommitCount and TransAbortCount) by this replica. Deprecated in NDB 8.0.23.
- `Ndb_api_trans_commit_count_slave`: Number of transactions committed by this replica. Deprecated in NDB 8.0.23.
- `Ndb_api_trans_local_read_row_count_slave`: Total number of rows that have been read by this replica. Deprecated in NDB 8.0.23.
- `Ndb_api_trans_start_count_slave`: Number of transactions started by this replica. Deprecated in NDB 8.0.23.
- `Ndb_api_uk_op_count_slave`: Number of operations based on or using unique keys by this replica. Deprecated in NDB 8.0.23.

- `Ndb_api_wait_exec_complete_count_slave`: Number of times thread has been blocked while waiting for operation execution to complete by this replica. Deprecated in NDB 8.0.23.
- `Ndb_api_wait_meta_request_count_slave`: Number of times thread has been blocked waiting for metadata-based signal by this replica. Deprecated in NDB 8.0.23.
- `Ndb_api_wait_nanos_count_slave`: Total time (in nanoseconds) spent waiting for some type of signal from data nodes by this replica. Deprecated in NDB 8.0.23.
- `Ndb_api_wait_scan_result_count_slave`: Number of times thread has been blocked while waiting for scan-based signal by this replica. Deprecated in NDB 8.0.23.
- `Ndb_metadata_blacklist_size`: Number of NDB metadata objects that NDB binlog thread has failed to synchronize; renamed in NDB 8.0.22 as Ndb_metadata_excluded_count. Deprecated in NDB 8.0.21.
- `Ndb_replica_max_replicated_epoch`: Most recently committed NDB epoch on this replica. When this value is greater than or equal to Ndb_conflict_last_conflict_epoch, no conflicts have yet been detected. Deprecated in NDB 8.0.23.
- `ndb_slave_conflict_role`: Role for replica to play in conflict detection and resolution. Value is one of PRIMARY, SECONDARY, PASS, or NONE (default). Can be changed only when replication SQL thread is stopped. See documentation for further information. Deprecated in NDB 8.0.23.
- `slave_allow_batching`: Turns update batching on and off for replica. Deprecated in NDB 8.0.26.

Options and Variables Removed in NDB 8.0

The following system variables, status variables, and options have been removed in NDB 8.0.

- `Ndb_metadata_blacklist_size`: Number of NDB metadata objects that NDB binlog thread has failed to synchronize; renamed in NDB 8.0.22 as Ndb_metadata_excluded_count. Removed in NDB 8.0.22.

23.2.6 MySQL Server Using InnoDB Compared with NDB Cluster

MySQL Server offers a number of choices in storage engines. Since both `NDB` and `InnoDB` can serve as transactional MySQL storage engines, users of MySQL Server sometimes become interested in NDB Cluster. They see `NDB` as a possible alternative or upgrade to the default `InnoDB` storage engine in MySQL 8.0. While `NDB` and `InnoDB` share common characteristics, there are differences in architecture and implementation, so that some existing MySQL Server applications and usage scenarios can be a good fit for NDB Cluster, but not all of them.

In this section, we discuss and compare some characteristics of the `NDB` storage engine used by NDB 8.0 with `InnoDB` used in MySQL 8.0. The next few sections provide a technical comparison. In many instances, decisions about when and where to use NDB Cluster must be made on a case-by-case basis, taking all factors into consideration. While it is beyond the scope of this documentation to provide specifics for every conceivable usage scenario, we also attempt to offer some very general guidance on the relative suitability of some common types of applications for `NDB` as opposed to `InnoDB` back ends.

NDB Cluster 8.0 uses a `mysqld` based on MySQL 8.0, including support for `InnoDB` 1.1. While it is possible to use `InnoDB` tables with NDB Cluster, such tables are not clustered. It is also not possible to use programs or libraries from an NDB Cluster 8.0 distribution with MySQL Server 8.0, or the reverse.

While it is also true that some types of common business applications can be run either on NDB Cluster or on MySQL Server (most likely using the `InnoDB` storage engine), there are some important architectural and implementation differences. Section 23.2.6.1, “Differences Between the NDB and InnoDB Storage Engines”, provides a summary of the these differences. Due to the differences, some usage scenarios are clearly more suitable for one engine or the other; see Section 23.2.6.2, “NDB and InnoDB Workloads”. This in turn has an impact on the types of applications that better suited for

use with NDB or InnoDB. See Section 23.2.6.3, “NDB and InnoDB Feature Usage Summary”, for a comparison of the relative suitability of each for use in common types of database applications.

For information about the relative characteristics of the NDB and MEMORY storage engines, see When to Use MEMORY or NDB Cluster.

See Chapter 16, *Alternative Storage Engines*, for additional information about MySQL storage engines.

23.2.6.1 Differences Between the NDB and InnoDB Storage Engines

The NDB storage engine is implemented using a distributed, shared-nothing architecture, which causes it to behave differently from InnoDB in a number of ways. For those unaccustomed to working with NDB, unexpected behaviors can arise due to its distributed nature with regard to transactions, foreign keys, table limits, and other characteristics. These are shown in the following table:

Table 23.2 Differences between InnoDB and NDB storage engines

Feature	InnoDB (MySQL 8.0)	NDB 8.0
MySQL Server Version	8.0	8.0
InnoDB Version	InnoDB 8.0.32	InnoDB 8.0.32
NDB Cluster Version	N/A	NDB 8.0.34/8.0.34
Storage Limits	64TB	128TB
Foreign Keys	Yes	Yes
Transactions	All standard types	READ COMMITTED
MVCC	Yes	No
Data Compression	Yes	No (NDB checkpoint and backup files can be compressed)
Large Row Support (> 14K)	Supported for VARBINARY, VARCHAR, BLOB, and TEXT columns	Supported for BLOB and TEXT columns only (Using these types to store very large amounts of data can lower NDB performance)
Replication Support	Asynchronous and semisynchronous replication using MySQL Replication; MySQL Group Replication	Automatic synchronous replication within an NDB Cluster; asynchronous replication between NDB Clusters, using MySQL Replication (Semisynchronous replication is not supported)
Scaleout for Read Operations	Yes (MySQL Replication)	Yes (Automatic partitioning in NDB Cluster; NDB Cluster Replication)
Scaleout for Write Operations	Requires application-level partitioning (sharding)	Yes (Automatic partitioning in NDB Cluster is transparent to applications)
High Availability (HA)	Built-in, from InnoDB cluster	Yes (Designed for 99.999% uptime)
Node Failure Recovery and Failover	From MySQL Group Replication	Automatic (Key element in NDB architecture)
Time for Node Failure Recovery	30 seconds or longer	Typically < 1 second
Real-Time Performance	No	Yes
In-Memory Tables	No	Yes (Some data can optionally be stored on disk; both in-

Feature	InnoDB (MySQL 8.0)	NDB 8.0
		memory and disk data storage are durable)
NoSQL Access to Storage Engine	Yes	Yes (Multiple APIs, including Memcached, Node.js/JavaScript, Java, JPA, C++, and HTTP/REST)
Concurrent and Parallel Writes	Yes	Up to 48 writers, optimized for concurrent writes
Conflict Detection and Resolution (Multiple Sources)	Yes (MySQL Group Replication)	Yes
Hash Indexes	No	Yes
Online Addition of Nodes	Read/write replicas using MySQL Group Replication	Yes (all node types)
Online Upgrades	Yes (using replication)	Yes
Online Schema Modifications	Yes, as part of MySQL 8.0	Yes

23.2.6.2 NDB and InnoDB Workloads

NDB Cluster has a range of unique attributes that make it ideal to serve applications requiring high availability, fast failover, high throughput, and low latency. Due to its distributed architecture and multi-node implementation, NDB Cluster also has specific constraints that may keep some workloads from performing well. A number of major differences in behavior between the NDB and InnoDB storage engines with regard to some common types of database-driven application workloads are shown in the following table::

Table 23.3 Differences between InnoDB and NDB storage engines, common types of data-driven application workloads.

Workload	InnoDB	NDB Cluster (NDB)
High-Volume OLTP Applications	Yes	Yes
DSS Applications (data marts, analytics)	Yes	Limited (Join operations across OLTP datasets not exceeding 3TB in size)
Custom Applications	Yes	Yes
Packaged Applications	Yes	Limited (should be mostly primary key access); NDB Cluster 8.0 supports foreign keys
In-Network Telecoms Applications (HLR, HSS, SDP)	No	Yes
Session Management and Caching	Yes	Yes
E-Commerce Applications	Yes	Yes
User Profile Management, AAA Protocol	Yes	Yes

23.2.6.3 NDB and InnoDB Feature Usage Summary

When comparing application feature requirements to the capabilities of InnoDB with NDB, some are clearly more compatible with one storage engine than the other.

The following table lists supported application features according to the storage engine to which each feature is typically better suited.

Table 23.4 Supported application features according to the storage engine to which each feature is typically better suited

Preferred application requirements for `InnoDB`	Preferred application requirements for `NDB`
• Foreign keys  **Note** NDB Cluster 8.0 supports foreign keys • Full table scans • Very large databases, rows, or transactions • Transactions other than `READ COMMITTED`	• Write scaling • 99.999% uptime • Online addition of nodes and online schema operations • Multiple SQL and NoSQL APIs (see NDB Cluster APIs: Overview and Concepts) • Real-time performance • Limited use of `BLOB` columns • Foreign keys are supported, although their use may have an impact on performance at high throughput

23.2.7 Known Limitations of NDB Cluster

In the sections that follow, we discuss known limitations in current releases of NDB Cluster as compared with the features available when using the `MyISAM` and `InnoDB` storage engines. If you check the “Cluster” category in the MySQL bugs database at http://bugs.mysql.com, you can find known bugs in the following categories under “MySQL Server:” in the MySQL bugs database at http://bugs.mysql.com, which we intend to correct in upcoming releases of NDB Cluster:

- NDB Cluster
- Cluster Direct API (NDBAPI)
- Cluster Disk Data
- Cluster Replication
- ClusterJ

This information is intended to be complete with respect to the conditions just set forth. You can report any discrepancies that you encounter to the MySQL bugs database using the instructions given in Section 1.5, “How to Report Bugs or Problems”. Any problem which we do not plan to fix in NDB Cluster 8.0, is added to the list.

See Section 23.2.7.11, “Previous NDB Cluster Issues Resolved in NDB Cluster 8.0” for a list of issues in earlier releases that have been resolved in NDB Cluster 8.0.

Note

Limitations and other issues specific to NDB Cluster Replication are described in Section 23.7.3, “Known Issues in NDB Cluster Replication”.

23.2.7.1 Noncompliance with SQL Syntax in NDB Cluster

Some SQL statements relating to certain MySQL features produce errors when used with `NDB` tables, as described in the following list:

- **Temporary tables.** Temporary tables are not supported. Trying either to create a temporary table that uses the `NDB` storage engine or to alter an existing temporary table to use `NDB` fails with the error `Table storage engine 'ndbcluster' does not support the create option 'TEMPORARY'`.

- **Indexes and keys in NDB tables.** Keys and indexes on NDB Cluster tables are subject to the following limitations:

  - **Column width.** Attempting to create an index on an `NDB` table column whose width is greater than 3072 bytes succeeds, but only the first 3072 bytes are actually used for the index. In such cases, a warning `Specified key was too long; max key length is 3072 bytes` is issued, and a `SHOW CREATE TABLE` statement shows the length of the index as 3072.

  - **TEXT and BLOB columns.** You cannot create indexes on `NDB` table columns that use any of the `TEXT` or `BLOB` data types.

  - **FULLTEXT indexes.** The `NDB` storage engine does not support `FULLTEXT` indexes, which are possible for `MyISAM` and `InnoDB` tables only.

    However, you can create indexes on `VARCHAR` columns of `NDB` tables.

  - **USING HASH keys and NULL.** Using nullable columns in unique keys and primary keys means that queries using these columns are handled as full table scans. To work around this issue, make the column `NOT NULL`, or re-create the index without the `USING HASH` option.

  - **Prefixes.** There are no prefix indexes; only entire columns can be indexed. (The size of an `NDB` column index is always the same as the width of the column in bytes, up to and including 3072 bytes, as described earlier in this section. Also see Section 23.2.7.6, “Unsupported or Missing Features in NDB Cluster”, for additional information.)

  - **BIT columns.** A `BIT` column cannot be a primary key, unique key, or index, nor can it be part of a composite primary key, unique key, or index.

  - **AUTO_INCREMENT columns.** Like other MySQL storage engines, the `NDB` storage engine can handle a maximum of one `AUTO_INCREMENT` column per table, and this column must be indexed. However, in the case of an NDB table with no explicit primary key, an `AUTO_INCREMENT` column is automatically defined and used as a “hidden” primary key. For this reason, you cannot create an `NDB` table having an `AUTO_INCREMENT` column and no explicit primary key.

    The following `CREATE TABLE` statements do not work, as shown here:

```
# No index on AUTO_INCREMENT column; table has no primary key
# Raises ER_WRONG_AUTO_KEY
mysql> CREATE TABLE n (
    ->     a INT,
    ->     b INT AUTO_INCREMENT
    ->     )
    -> ENGINE=NDB;
ERROR 1075 (42000): Incorrect table definition; there can be only one auto
column and it must be defined as a key

# Index on AUTO_INCREMENT column; table has no primary key
# Raises NDB error 4335
mysql> CREATE TABLE n (
    ->     a INT,
    ->     b INT AUTO_INCREMENT,
    ->     KEY k (b)
    ->     )
    -> ENGINE=NDB;
ERROR 1296 (HY000): Got error 4335 'Only one autoincrement column allowed per
table. Having a table without primary key uses an autoincr' from NDBCLUSTER
```

    The following statement creates a table with a primary key, an `AUTO_INCREMENT` column, and an index on this column, and succeeds:

```
# Index on AUTO_INCREMENT column; table has a primary key
mysql> CREATE TABLE n (
    ->     a INT PRIMARY KEY,
    ->     b INT AUTO_INCREMENT,
    ->     KEY k (b)
```

```
    ->     )
    -> ENGINE=NDB;
Query OK, 0 rows affected (0.38 sec)
```

- **Restrictions on foreign keys.** Support for foreign key constraints in NDB 8.0 is comparable to that provided by `InnoDB`, subject to the following restrictions:

  - Every column referenced as a foreign key requires an explicit unique key, if it is not the table's primary key.
  - `ON UPDATE CASCADE` is not supported when the reference is to the parent table's primary key.

    This is because an update of a primary key is implemented as a delete of the old row (containing the old primary key) plus an insert of the new row (with a new primary key). This is not visible to the `NDB` kernel, which views these two rows as being the same, and thus has no way of knowing that this update should be cascaded.
  - `ON DELETE CASCADE` is also not supported where the child table contains one or more columns of any of the `TEXT` or `BLOB` types. (Bug #89511, Bug #27484882)
  - `SET DEFAULT` is not supported. (Also not supported by `InnoDB`.)
  - The `NO ACTION` keyword is accepted but treated as `RESTRICT`. `NO ACTION`, which is a standard SQL keyword, is the default in MySQL 8.0. (Also the same as with `InnoDB`.)
  - In earlier versions of NDB Cluster, when creating a table with foreign key referencing an index in another table, it sometimes appeared possible to create the foreign key even if the order of the columns in the indexes did not match, due to the fact that an appropriate error was not always returned internally. A partial fix for this issue improved the error used internally to work in most cases; however, it remains possible for this situation to occur in the event that the parent index is a unique index. (Bug #18094360)

  For more information, see Section 13.1.20.5, “FOREIGN KEY Constraints”, and Section 1.6.3.2, “FOREIGN KEY Constraints”.

- **NDB Cluster and geometry data types.**
  Geometry data types (`WKT` and `WKB`) are supported for `NDB` tables. However, spatial indexes are not supported.

- **Character sets and binary log files.** Currently, the `ndb_apply_status` and `ndb_binlog_index` tables are created using the `latin1` (ASCII) character set. Because names of binary logs are recorded in this table, binary log files named using non-Latin characters are not referenced correctly in these tables. This is a known issue, which we are working to fix. (Bug #50226)

  To work around this problem, use only Latin-1 characters when naming binary log files or setting any the `--basedir`, `--log-bin`, or `--log-bin-index` options.

- **Creating NDB tables with user-defined partitioning.** Support for user-defined partitioning in NDB Cluster is restricted to [`LINEAR`] `KEY` partitioning. Using any other partitioning type with `ENGINE=NDB` or `ENGINE=NDBCLUSTER` in a `CREATE TABLE` statement results in an error.

  It is possible to override this restriction, but doing so is not supported for use in production settings. For details, see User-defined partitioning and the NDB storage engine (NDB Cluster).

  Default partitioning scheme. All NDB Cluster tables are by default partitioned by `KEY` using the table's primary key as the partitioning key. If no primary key is explicitly set for the table, the

"hidden" primary key automatically created by the NDB storage engine is used instead. For additional discussion of these and related issues, see Section 24.2.5, "KEY Partitioning".

CREATE TABLE and ALTER TABLE statements that would cause a user-partitioned NDBCLUSTER table not to meet either or both of the following two requirements are not permitted, and fail with an error:

1. The table must have an explicit primary key.

2. All columns listed in the table's partitioning expression must be part of the primary key.

Exception. If a user-partitioned NDBCLUSTER table is created using an empty column-list (that is, using PARTITION BY [LINEAR] KEY()), then no explicit primary key is required.

Maximum number of partitions for NDBCLUSTER tables. The maximum number of partitions that can defined for a NDBCLUSTER table when employing user-defined partitioning is 8 per node group. (See Section 23.2.2, "NDB Cluster Nodes, Node Groups, Fragment Replicas, and Partitions", for more information about NDB Cluster node groups.

DROP PARTITION not supported. It is not possible to drop partitions from NDB tables using ALTER TABLE ... DROP PARTITION. The other partitioning extensions to ALTER TABLE—ADD PARTITION, REORGANIZE PARTITION, and COALESCE PARTITION—are supported for NDB tables, but use copying and so are not optimized. See Section 24.3.1, "Management of RANGE and LIST Partitions" and Section 13.1.9, "ALTER TABLE Statement".

Partition selection. Partition selection is not supported for NDB tables. See Section 24.5, "Partition Selection", for more information.

- **JSON data type.** The MySQL JSON data type is supported for NDB tables in the mysqld supplied with NDB 8.0.

  An NDB table can have a maximum of 3 JSON columns.

  The NDB API has no special provision for working with JSON data, which it views simply as BLOB data. Handling data as JSON must be performed by the application.

23.2.7.2 Limits and Differences of NDB Cluster from Standard MySQL Limits

In this section, we list limits found in NDB Cluster that either differ from limits found in, or that are not found in, standard MySQL.

Memory usage and recovery. Memory consumed when data is inserted into an NDB table is not automatically recovered when deleted, as it is with other storage engines. Instead, the following rules hold true:

- A DELETE statement on an NDB table makes the memory formerly used by the deleted rows available for re-use by inserts on the same table only. However, this memory can be made available for general re-use by performing OPTIMIZE TABLE.

  A rolling restart of the cluster also frees any memory used by deleted rows. See Section 23.6.5, "Performing a Rolling Restart of an NDB Cluster".

- A DROP TABLE or TRUNCATE TABLE operation on an NDB table frees the memory that was used by this table for re-use by any NDB table, either by the same table or by another NDB table.

  Note

  Recall that TRUNCATE TABLE drops and re-creates the table. See Section 13.1.37, "TRUNCATE TABLE Statement".

- **Limits imposed by the cluster's configuration.**

A number of hard limits exist which are configurable, but available main memory in the cluster sets limits. See the complete list of configuration parameters in Section 23.4.3, “NDB Cluster Configuration Files”. Most configuration parameters can be upgraded online. These hard limits include:

- Database memory size and index memory size (`DataMemory` and `IndexMemory`, respectively).

  `DataMemory` is allocated as 32KB pages. As each `DataMemory` page is used, it is assigned to a specific table; once allocated, this memory cannot be freed except by dropping the table.

  See Section 23.4.3.6, “Defining NDB Cluster Data Nodes”, for more information.

- The maximum number of operations that can be performed per transaction is set using the configuration parameters `MaxNoOfConcurrentOperations` and `MaxNoOfLocalOperations`.

  Note

  Bulk loading, `TRUNCATE TABLE`, and `ALTER TABLE` are handled as special cases by running multiple transactions, and so are not subject to this limitation.

- Different limits related to tables and indexes. For example, the maximum number of ordered indexes in the cluster is determined by `MaxNoOfOrderedIndexes`, and the maximum number of ordered indexes per table is 16.

- **Node and data object maximums.** The following limits apply to numbers of cluster nodes and metadata objects:

  - The maximum number of data nodes is 144. (In NDB 7.6 and earlier, this was 48.)

    A data node must have a node ID in the range of 1 to 144, inclusive.

    Management and API nodes may use node IDs in the range 1 to 255, inclusive.

  - The total maximum number of nodes in an NDB Cluster is 255. This number includes all SQL nodes (MySQL Servers), API nodes (applications accessing the cluster other than MySQL servers), data nodes, and management servers.

  - The maximum number of metadata objects in current versions of NDB Cluster is 20320. This limit is hard-coded.

  See Section 23.2.7.11, “Previous NDB Cluster Issues Resolved in NDB Cluster 8.0”, for more information.

23.2.7.3 Limits Relating to Transaction Handling in NDB Cluster

A number of limitations exist in NDB Cluster with regard to the handling of transactions. These include the following:

- **Transaction isolation level.** The `NDBCLUSTER` storage engine supports only the `READ COMMITTED` transaction isolation level. (`InnoDB`, for example, supports `READ COMMITTED`, `READ UNCOMMITTED`, `REPEATABLE READ`, and `SERIALIZABLE`.) You should keep in mind that `NDB` implements `READ COMMITTED` on a per-row basis; when a read request arrives at the data node storing the row, what is returned is the last committed version of the row at that time.

  Uncommitted data is never returned, but when a transaction modifying a number of rows commits concurrently with a transaction reading the same rows, the transaction performing the read can observe “before” values, “after” values, or both, for different rows among these, due to the fact that a given row read request can be processed either before or after the commit of the other transaction.

To ensure that a given transaction reads only before or after values, you can impose row locks using `SELECT ... LOCK IN SHARE MODE`. In such cases, the lock is held until the owning transaction is committed. Using row locks can also cause the following issues:

- Increased frequency of lock wait timeout errors, and reduced concurrency
- Increased transaction processing overhead due to reads requiring a commit phase
- Possibility of exhausting the available number of concurrent locks, which is limited by `MaxNoOfConcurrentOperations`

`NDB` uses `READ COMMITTED` for all reads unless a modifier such as `LOCK IN SHARE MODE` or `FOR UPDATE` is used. `LOCK IN SHARE MODE` causes shared row locks to be used; `FOR UPDATE` causes exclusive row locks to be used. Unique key reads have their locks upgraded automatically by `NDB` to ensure a self-consistent read; `BLOB` reads also employ extra locking for consistency.

See Section 23.6.8.4, “NDB Cluster Backup Troubleshooting”, for information on how NDB Cluster's implementation of transaction isolation level can affect backup and restoration of `NDB` databases.

- **Transactions and BLOB or TEXT columns.** `NDBCLUSTER` stores only part of a column value that uses any of MySQL's `BLOB` or `TEXT` data types in the table visible to MySQL; the remainder of the `BLOB` or `TEXT` is stored in a separate internal table that is not accessible to MySQL. This gives rise to two related issues of which you should be aware whenever executing `SELECT` statements on tables that contain columns of these types:

  1. For any `SELECT` from an NDB Cluster table: If the `SELECT` includes a `BLOB` or `TEXT` column, the `READ COMMITTED` transaction isolation level is converted to a read with read lock. This is done to guarantee consistency.
  2. For any `SELECT` which uses a unique key lookup to retrieve any columns that use any of the `BLOB` or `TEXT` data types and that is executed within a transaction, a shared read lock is held on the table for the duration of the transaction—that is, until the transaction is either committed or aborted.

     This issue does not occur for queries that use index or table scans, even against `NDB` tables having `BLOB` or `TEXT` columns.

     For example, consider the table `t` defined by the following `CREATE TABLE` statement:

     ```
     CREATE TABLE t (
         a INT NOT NULL AUTO_INCREMENT PRIMARY KEY,
         b INT NOT NULL,
         c INT NOT NULL,
         d TEXT,
         INDEX i(b),
         UNIQUE KEY u(c)
     ) ENGINE = NDB,
     ```

     The following query on `t` causes a shared read lock, because it uses a unique key lookup:

     ```
     SELECT * FROM t WHERE c = 1;
     ```

     However, none of the four queries shown here causes a shared read lock:

     ```
     SELECT * FROM t WHERE b = 1;

     SELECT * FROM t WHERE d = '1';

     SELECT * FROM t;
     ```

```
SELECT b,c WHERE a = 1;
```

This is because, of these four queries, the first uses an index scan, the second and third use table scans, and the fourth, while using a primary key lookup, does not retrieve the value of any `BLOB` or `TEXT` columns.

You can help minimize issues with shared read locks by avoiding queries that use unique key lookups that retrieve `BLOB` or `TEXT` columns, or, in cases where such queries are not avoidable, by committing transactions as soon as possible afterward.

- **Unique key lookups and transaction isolation.** Unique indexes are implemented in `NDB` using a hidden index table which is maintained internally. When a user-created `NDB` table is accessed using a unique index, the hidden index table is first read to find the primary key that is then used to read the user-created table. To avoid modification of the index during this double-read operation, the row found in the hidden index table is locked. When a row referenced by a unique index in the user-created `NDB` table is updated, the hidden index table is subject to an exclusive lock by the transaction in which the update is performed. This means that any read operation on the same (user-created) `NDB` table must wait for the update to complete. This is true even when the transaction level of the read operation is `READ COMMITTED`.

  One workaround which can be used to bypass potentially blocking reads is to force the SQL node to ignore the unique index when performing the read. This can be done by using the `IGNORE INDEX` index hint as part of the `SELECT` statement reading the table (see Section 8.9.4, “Index Hints”). Because the MySQL server creates a shadowing ordered index for every unique index created in `NDB`, this lets the ordered index be read instead, and avoids unique index access locking. The resulting read is as consistent as a committed read by primary key, returning the last committed value at the time the row is read.

  Reading via an ordered index makes less efficient use of resources in the cluster, and may have higher latency.

  It is also possible to avoid using the unique index for access by querying for ranges rather than for unique values.

- **Rollbacks.** There are no partial transactions, and no partial rollbacks of transactions. A duplicate key or similar error causes the entire transaction to be rolled back.

  This behavior differs from that of other transactional storage engines such as `InnoDB` that may roll back individual statements.

- **Transactions and memory usage.**
  As noted elsewhere in this chapter, NDB Cluster does not handle large transactions well; it is better to perform a number of small transactions with a few operations each than to attempt a single large transaction containing a great many operations. Among other considerations, large transactions require very large amounts of memory. Because of this, the transactional behavior of a number of MySQL statements is affected as described in the following list:

  - `TRUNCATE TABLE` is not transactional when used on `NDB` tables. If a `TRUNCATE TABLE` fails to empty the table, then it must be re-run until it is successful.
  - `DELETE FROM` (even with no `WHERE` clause) *is* transactional. For tables containing a great many rows, you may find that performance is improved by using several `DELETE FROM ... LIMIT ...` statements to “chunk” the delete operation. If your objective is to empty the table, then you may wish to use `TRUNCATE TABLE` instead.
  - **LOAD DATA statements.** `LOAD DATA` is not transactional when used on `NDB` tables.

    Important

    When executing a `LOAD DATA` statement, the `NDB` engine performs commits at irregular intervals that enable better utilization of the

communication network. It is not possible to know ahead of time when such commits take place.

  - **ALTER TABLE and transactions.** When copying an `NDB` table as part of an `ALTER TABLE`, the creation of the copy is nontransactional. (In any case, this operation is rolled back when the copy is deleted.)

- **Transactions and the COUNT() function.** When using NDB Cluster Replication, it is not possible to guarantee the transactional consistency of the `COUNT()` function on the replica. In other words, when performing on the source a series of statements (`INSERT`, `DELETE`, or both) that changes the number of rows in a table within a single transaction, executing `SELECT COUNT(*) FROM table` queries on the replica may yield intermediate results. This is due to the fact that `SELECT COUNT(...)` may perform dirty reads, and is not a bug in the `NDB` storage engine. (See Bug #31321 for more information.)

23.2.7.4 NDB Cluster Error Handling

Starting, stopping, or restarting a node may give rise to temporary errors causing some transactions to fail. These include the following cases:

- **Temporary errors.** When first starting a node, it is possible that you may see Error 1204 `Temporary failure, distribution changed` and similar temporary errors.

- **Errors due to node failure.** The stopping or failure of any data node can result in a number of different node failure errors. (However, there should be no aborted transactions when performing a planned shutdown of the cluster.)

In either of these cases, any errors that are generated must be handled within the application. This should be done by retrying the transaction.

See also Section 23.2.7.2, “Limits and Differences of NDB Cluster from Standard MySQL Limits”.

23.2.7.5 Limits Associated with Database Objects in NDB Cluster

Some database objects such as tables and indexes have different limitations when using the `NDBCLUSTER` storage engine:

- **Number of database objects.** The maximum number of *all* `NDB` database objects in a single NDB Cluster—including databases, tables, and indexes—is limited to 20320.

- **Attributes per table.** The maximum number of attributes (that is, columns and indexes) that can belong to a given table is 512.

- **Attributes per key.** The maximum number of attributes per key is 32.

- **Row size.** In NDB 8.0, the maximum permitted size of any one row is 30000 bytes (increased from 14000 bytes in previous releases).

  Each `BLOB` or `TEXT` column contributes 256 + 8 = 264 bytes to this total; this includes `JSON` columns. See String Type Storage Requirements, as well as JSON Storage Requirements, for more information relating to these types.

  In addition, the maximum offset for a fixed-width column of an `NDB` table is 8188 bytes; attempting to create a table that violates this limitation fails with NDB error 851 `Maximum offset for fixed-size columns exceeded`. For memory-based columns, you can work around this limitation by using a variable-width column type such as `VARCHAR` or defining the column as `COLUMN_FORMAT=DYNAMIC`; this does not work with columns stored on disk. For disk-based columns, you may be able to do so by reordering one or more of the table's disk-based columns such that the combined width of all but the disk-based column defined last in the `CREATE TABLE` statement used to create the table does not exceed 8188 bytes, less any possible rounding performed for some data types such as `CHAR` or `VARCHAR`; otherwise it is necessary to use memory-based storage for one or more of the offending column or columns instead.

- **BIT column storage per table.** The maximum combined width for all `BIT` columns used in a given `NDB` table is 4096.

- **FIXED column storage.** NDB Cluster 8.0 supports a maximum of 128 TB per fragment of data in `FIXED` columns.

23.2.7.6 Unsupported or Missing Features in NDB Cluster

A number of features supported by other storage engines are not supported for `NDB` tables. Trying to use any of these features in NDB Cluster does not cause errors in or of itself; however, errors may occur in applications that expects the features to be supported or enforced. Statements referencing such features, even if effectively ignored by `NDB`, must be syntactically and otherwise valid.

- **Index prefixes.** Prefixes on indexes are not supported for `NDB` tables. If a prefix is used as part of an index specification in a statement such as `CREATE TABLE`, `ALTER TABLE`, or `CREATE INDEX`, the prefix is not created by `NDB`.

  A statement containing an index prefix, and creating or modifying an `NDB` table, must still be syntactically valid. For example, the following statement always fails with Error 1089 `Incorrect prefix key; the used key part isn't a string, the used length is longer than the key part, or the storage engine doesn't support unique prefix keys`, regardless of storage engine:

  ```
  CREATE TABLE t1 (
      c1 INT NOT NULL,
      c2 VARCHAR(100),
      INDEX i1 (c2(500))
  );
  ```

  This happens on account of the SQL syntax rule that no index may have a prefix larger than itself.

- **Savepoints and rollbacks.** Savepoints and rollbacks to savepoints are ignored as in `MyISAM`.

- **Durability of commits.** There are no durable commits on disk. Commits are replicated, but there is no guarantee that logs are flushed to disk on commit.

- **Replication.** Statement-based replication is not supported. Use `--binlog-format=ROW` (or `--binlog-format=MIXED`) when setting up cluster replication. See Section 23.7, “NDB Cluster Replication”, for more information.

  Replication using global transaction identifiers (GTIDs) is not compatible with NDB Cluster, and is not supported in NDB Cluster 8.0. Do not enable GTIDs when using the `NDB` storage engine, as this is very likely to cause problems up to and including failure of NDB Cluster Replication.

  Semisynchronous replication is not supported in NDB Cluster.

- **Generated columns.** The `NDB` storage engine does not support indexes on virtual generated columns.

  As with other storage engines, you can create an index on a stored generated column, but you should bear in mind that `NDB` uses `DataMemory` for storage of the generated column as well as `IndexMemory` for the index. See JSON columns and indirect indexing in NDB Cluster, for an example.

  NDB Cluster writes changes in stored generated columns to the binary log, but does log not those made to virtual columns. This should not effect NDB Cluster Replication or replication between `NDB` and other MySQL storage engines.

Note

See Section 23.2.7.3, “Limits Relating to Transaction Handling in NDB Cluster”, for more information relating to limitations on transaction handling in `NDB`.

23.2.7.7 Limitations Relating to Performance in NDB Cluster

The following performance issues are specific to or especially pronounced in NDB Cluster:

- **Range scans.** There are query performance issues due to sequential access to the `NDB` storage engine; it is also relatively more expensive to do many range scans than it is with either `MyISAM` or `InnoDB`.
- **Reliability of Records in range.** The `Records in range` statistic is available but is not completely tested or officially supported. This may result in nonoptimal query plans in some cases. If necessary, you can employ `USE INDEX` or `FORCE INDEX` to alter the execution plan. See Section 8.9.4, “Index Hints”, for more information on how to do this.
- **Unique hash indexes.** Unique hash indexes created with `USING HASH` cannot be used for accessing a table if `NULL` is given as part of the key.

23.2.7.8 Issues Exclusive to NDB Cluster

The following are limitations specific to the `NDB` storage engine:

- **Machine architecture.** All machines used in the cluster must have the same architecture. That is, all machines hosting nodes must be either big-endian or little-endian, and you cannot use a mixture of both. For example, you cannot have a management node running on a PowerPC which directs a data node that is running on an x86 machine. This restriction does not apply to machines simply running `mysql` or other clients that may be accessing the cluster's SQL nodes.
- **Binary logging.**
  NDB Cluster has the following limitations or restrictions with regard to binary logging:
  - `sql_log_bin` has no effect on data operations; however, it is supported for schema operations.
  - NDB Cluster cannot produce a binary log for tables having `BLOB` columns but no primary key.
  - Only the following schema operations are logged in a cluster binary log which is *not* on the `mysqld` executing the statement:
    - `CREATE TABLE`
    - `ALTER TABLE`
    - `DROP TABLE`
    - `CREATE DATABASE` / `CREATE SCHEMA`
    - `DROP DATABASE` / `DROP SCHEMA`
    - `CREATE TABLESPACE`
    - `ALTER TABLESPACE`
    - `DROP TABLESPACE`
    - `CREATE LOGFILE GROUP`
    - `ALTER LOGFILE GROUP`
    - `DROP LOGFILE GROUP`
- **Schema operations.** Schema operations (DDL statements) are rejected while any data node restarts. Schema operations are also not supported while performing an online upgrade or downgrade.

- **Number of fragment replicas.** The number of fragment replicas, as determined by the `NoOfReplicas` data node configuration parameter, is the number of copies of all data stored by NDB Cluster. Setting this parameter to 1 means there is only a single copy; in this case, no redundancy is provided, and the loss of a data node entails loss of data. To guarantee redundancy, and thus preservation of data even if a data node fails, set this parameter to 2, which is the default and recommended value in production.

  Setting `NoOfReplicas` to a value greater than 2 is supported (to a maximum of 4) but unnecessary to guard against loss of data.

See also Section 23.2.7.10, “Limitations Relating to Multiple NDB Cluster Nodes”.

23.2.7.9 Limitations Relating to NDB Cluster Disk Data Storage

Disk Data object maximums and minimums. Disk data objects are subject to the following maximums and minimums:

- Maximum number of tablespaces: 2^{32} (4294967296)
- Maximum number of data files per tablespace: 2^{16} (65536)
- The minimum and maximum possible sizes of extents for tablespace data files are 32K and 2G, respectively. See Section 13.1.21, “CREATE TABLESPACE Statement”, for more information.

In addition, when working with NDB Disk Data tables, you should be aware of the following issues regarding data files and extents:

- Data files use `DataMemory`. Usage is the same as for in-memory data.
- Data files use file descriptors. It is important to keep in mind that data files are always open, which means the file descriptors are always in use and cannot be re-used for other system tasks.
- Extents require sufficient `DiskPageBufferMemory`; you must reserve enough for this parameter to account for all memory used by all extents (number of extents times size of extents).

Disk Data tables and diskless mode. Use of Disk Data tables is not supported when running the cluster in diskless mode.

23.2.7.10 Limitations Relating to Multiple NDB Cluster Nodes

Multiple SQL nodes.
The following are issues relating to the use of multiple MySQL servers as NDB Cluster SQL nodes, and are specific to the `NDBCLUSTER` storage engine:

- **Stored programs not distributed.** Stored procedures, stored functions, triggers, and scheduled events are all supported by tables using the `NDB` storage engine, but these do *not* propagate automatically between MySQL Servers acting as Cluster SQL nodes, and must be re-created separately on each SQL node. See Stored routines and triggers in NDB Cluster.
- **No distributed table locks.** A `LOCK TABLES` statement or `GET_LOCK()` call works only for the SQL node on which the lock is issued; no other SQL node in the cluster “sees” this lock. This is true for a lock issued by any statement that locks tables as part of its operations. (See next item for an example.)

  Implementing table locks in `NDBCLUSTER` can be done in an API application, and ensuring that all applications start by setting `LockMode` to `LM_Read` or `LM_Exclusive`. For more information about how to do this, see the description of `NdbOperation::getLockHandle()` in the *NDB Cluster API Guide*.
- **ALTER TABLE operations.** `ALTER TABLE` is not fully locking when running multiple MySQL servers (SQL nodes). (As discussed in the previous item, NDB Cluster does not support distributed table locks.)

Multiple management nodes.
When using multiple management servers:

- If any of the management servers are running on the same host, you must give nodes explicit IDs in connection strings because automatic allocation of node IDs does not work across multiple management servers on the same host. This is not required if every management server resides on a different host.
- When a management server starts, it first checks for any other management server in the same NDB Cluster, and upon successful connection to the other management server uses its configuration data. This means that the management server `--reload` and `--initial` startup options are ignored unless the management server is the only one running. It also means that, when performing a rolling restart of an NDB Cluster with multiple management nodes, the management server reads its own configuration file if (and only if) it is the only management server running in this NDB Cluster. See Section 23.6.5, “Performing a Rolling Restart of an NDB Cluster”, for more information.

Multiple network addresses. Multiple network addresses per data node are not supported. Use of these is liable to cause problems: In the event of a data node failure, an SQL node waits for confirmation that the data node went down but never receives it because another route to that data node remains open. This can effectively make the cluster inoperable.

Note

It is possible to use multiple network hardware *interfaces* (such as Ethernet cards) for a single data node, but these must be bound to the same address. This also means that it not possible to use more than one `[tcp]` section per connection in the `config.ini` file. See Section 23.4.3.10, “NDB Cluster TCP/IP Connections”, for more information.

23.2.7.11 Previous NDB Cluster Issues Resolved in NDB Cluster 8.0

A number of limitations and related issues that existed in earlier versions of NDB Cluster have been resolved in NDB 8.0. These are described briefly in the following list:

- **Database and table names.** In NDB 7.6 and earlier, when using the `NDB` storage engine, the maximum allowed length both for database names and for table names was 63 characters, and a statement using a database name or table name longer than this limit failed with an appropriate error. In NDB 8.0, this restriction is lifted; identifiers for `NDB` databases and tables may now use up to 64 bytes, as with other MySQL database and table names.
- **IPv6 support.** Prior to NDB 8.0.22, it was necessary for all network addresses used for connections between nodes within an NDB Cluster to use or to be resolvable to IPv4 addresses. Beginning with NDB 8.0.22, `NDB` supports IPv6 addresses for all types of cluster nodes, as well as for applications that use the NDB API or MGM API.

  For more information, see Known Issues When Upgrading or Downgrading NDB Cluster.

23.3 NDB Cluster Installation

This section describes the basics for planning, installing, configuring, and running an NDB Cluster. Whereas the examples in Section 23.4, “Configuration of NDB Cluster” provide more in-depth information on a variety of clustering options and configuration, the result of following the guidelines and procedures outlined here should be a usable NDB Cluster which meets the *minimum* requirements for availability and safeguarding of data.

For information about upgrading or downgrading an NDB Cluster between release versions, see Section 23.3.7, “Upgrading and Downgrading NDB Cluster”.

This section covers hardware and software requirements; networking issues; installation of NDB Cluster; basic configuration issues; starting, stopping, and restarting the cluster; loading of a sample database; and performing queries.

Assumptions. The following sections make a number of assumptions regarding the cluster's physical and network configuration. These assumptions are discussed in the next few paragraphs.

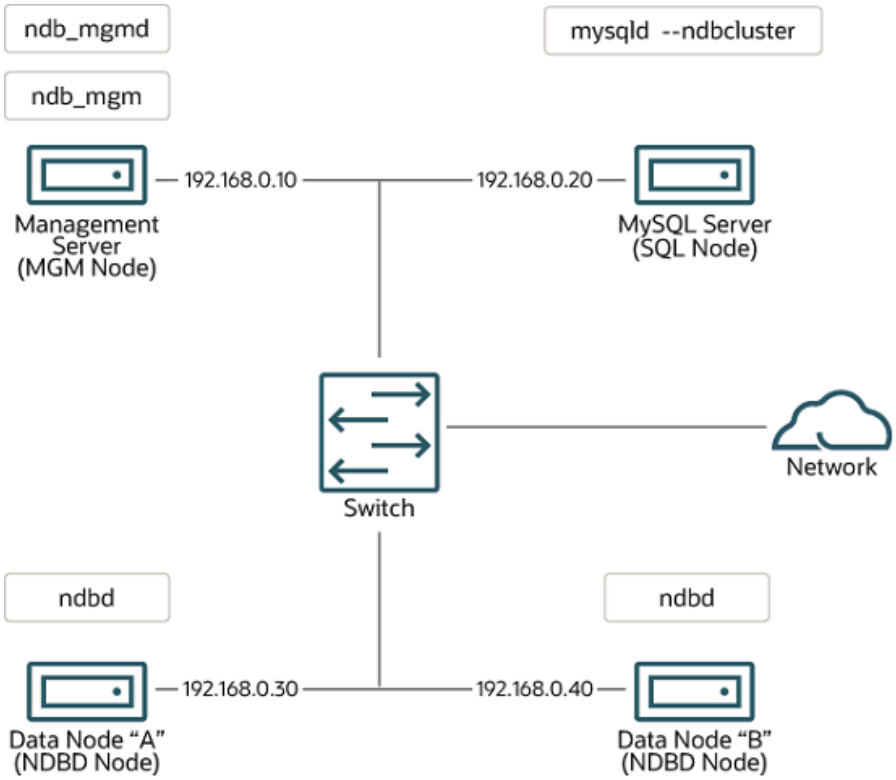
Cluster nodes and host computers. The cluster consists of four nodes, each on a separate host computer, and each with a fixed network address on a typical Ethernet network as shown here:

Table 23.5 Network addresses of nodes in example cluster

Node	IP Address
Management node (`mgmd`)	198.51.100.10
SQL node (`mysqld`)	198.51.100.20
Data node "A" (`ndbd`)	198.51.100.30
Data node "B" (`ndbd`)	198.51.100.40

This setup is also shown in the following diagram:

Figure 23.4 NDB Cluster Multi-Computer Setup



Network addressing. In the interest of simplicity (and reliability), this *How-To* uses only numeric IP addresses. However, if DNS resolution is available on your network, it is possible to use host names in lieu of IP addresses in configuring Cluster. Alternatively, you can use the `hosts` file (typically `/etc/hosts` for Linux and other Unix-like operating systems, `C:\WINDOWS\system32\drivers\etc\hosts` on Windows, or your operating system's equivalent) for providing a means to do host lookup if such is available.

As of NDB 8.0.22, NDB Cluster supports IPv6 for connections between all cluster nodes.

A known issue on Linux platforms when running NDB 8.0.22 and later is that the operating system kernel must provide IPv6 support, even if no IPv6 addresses are in use. If you wish to disable support for IPv6 on the system (because you do not plan to use any IPv6 addresses for NDB Cluster nodes), do so after booting the system, like this:

```
$> sysctl -w net.ipv6.conf.all.disable_ipv6=1
$> sysctl -w net.ipv6.conf.default.disable_ipv6=1
```

Alternatively, you can add the corresponding lines to `/etc/sysctl.conf`.

In NDB 8.0.21 and earlier releases, all network addresses used for connections to or from data and management nodes must use or be resolvable using IPv4, including addresses used by SQL nodes to contact the other nodes.

Potential hosts file issues. A common problem when trying to use host names for Cluster nodes arises because of the way in which some operating systems (including some Linux distributions) set up the system's own host name in the `/etc/hosts` during installation. Consider two machines with the host names `ndb1` and `ndb2`, both in the `cluster` network domain. Red Hat Linux (including some derivatives such as CentOS and Fedora) places the following entries in these machines' `/etc/hosts` files:

```
#  ndb1 /etc/hosts:
127.0.0.1   ndb1.cluster ndb1 localhost.localdomain localhost
```

```
#  ndb2 /etc/hosts:
127.0.0.1   ndb2.cluster ndb2 localhost.localdomain localhost
```

SUSE Linux (including OpenSUSE) places these entries in the machines' `/etc/hosts` files:

```
#  ndb1 /etc/hosts:
127.0.0.1       localhost
127.0.0.2       ndb1.cluster ndb1
```

```
#  ndb2 /etc/hosts:
127.0.0.1       localhost
127.0.0.2       ndb2.cluster ndb2
```

In both instances, `ndb1` routes `ndb1.cluster` to a loopback IP address, but gets a public IP address from DNS for `ndb2.cluster`, while `ndb2` routes `ndb2.cluster` to a loopback address and obtains a public address for `ndb1.cluster`. The result is that each data node connects to the management server, but cannot tell when any other data nodes have connected, and so the data nodes appear to hang while starting.

Caution

You cannot mix `localhost` and other host names or IP addresses in `config.ini`. For these reasons, the solution in such cases (other than to use IP addresses for *all* `config.ini` `HostName` entries) is to remove the fully qualified host names from `/etc/hosts` and use these in `config.ini` for all cluster hosts.

Host computer type. Each host computer in our installation scenario is an Intel-based desktop PC running a supported operating system installed to disk in a standard configuration, and running no unnecessary services. The core operating system with standard TCP/IP networking capabilities should be sufficient. Also for the sake of simplicity, we also assume that the file systems on all hosts are set up identically. In the event that they are not, you should adapt these instructions accordingly.

Network hardware. Standard 100 Mbps or 1 gigabit Ethernet cards are installed on each machine, along with the proper drivers for the cards, and that all four hosts are connected through a standard-issue Ethernet networking appliance such as a switch. (All machines should use network cards with the same throughput. That is, all four machines in the cluster should have 100 Mbps cards *or* all four machines should have 1 Gbps cards.) NDB Cluster works in a 100 Mbps network; however, gigabit Ethernet provides better performance.

Important

NDB Cluster is *not* intended for use in a network for which throughput is less than 100 Mbps or which experiences a high degree of latency. For this reason (among others), attempting to run an NDB Cluster over a wide area network such as the Internet is not likely to be successful, and is not supported in production.

Sample data. We use the `world` database which is available for download from the MySQL website (see https://dev.mysql.com/doc/index-other.html). We assume that each machine has sufficient memory for running the operating system, required NDB Cluster processes, and (on the data nodes) storing the database.

For general information about installing MySQL, see Chapter 2, *Installing and Upgrading MySQL*. For information about installation of NDB Cluster on Linux and other Unix-like operating systems, see Section 23.3.1, “Installation of NDB Cluster on Linux”. For information about installation of NDB Cluster on Windows operating systems, see Section 23.3.2, “Installing NDB Cluster on Windows”.

For general information about NDB Cluster hardware, software, and networking requirements, see Section 23.2.3, “NDB Cluster Hardware, Software, and Networking Requirements”.

23.3.1 Installation of NDB Cluster on Linux

This section covers installation methods for NDB Cluster on Linux and other Unix-like operating systems. While the next few sections refer to a Linux operating system, the instructions and procedures given there should be easily adaptable to other supported Unix-like platforms. For manual installation and setup instructions specific to Windows systems, see Section 23.3.2, “Installing NDB Cluster on Windows”.

Each NDB Cluster host computer must have the correct executable programs installed. A host running an SQL node must have installed on it a MySQL Server binary (`mysqld`). Management nodes require the management server daemon (`ndb_mgmd`); data nodes require the data node daemon (`ndbd` or `ndbmtd`). It is not necessary to install the MySQL Server binary on management node hosts and data node hosts. It is recommended that you also install the management client (`ndb_mgm`) on the management server host.

Installation of NDB Cluster on Linux can be done using precompiled binaries from Oracle (downloaded as a .tar.gz archive), with RPM packages (also available from Oracle), or from source code. All three of these installation methods are described in the section that follow.

Regardless of the method used, it is still necessary following installation of the NDB Cluster binaries to create configuration files for all cluster nodes, before you can start the cluster. See Section 23.3.3, “Initial Configuration of NDB Cluster”.

23.3.1.1 Installing an NDB Cluster Binary Release on Linux

This section covers the steps necessary to install the correct executables for each type of Cluster node from precompiled binaries supplied by Oracle.

For setting up a cluster using precompiled binaries, the first step in the installation process for each cluster host is to download the binary archive from the NDB Cluster downloads page. (For the most recent 64-bit NDB 8.0 release, this is `mysql-cluster-gpl-8.0.32-linux-glibc2.12-x86_64.tar.gz`.) We assume that you have placed this file in each machine's `/var/tmp` directory.

If you require a custom binary, see Section 2.8.5, “Installing MySQL Using a Development Source Tree”.

Note

After completing the installation, do not yet start any of the binaries. We show you how to do so following the configuration of the nodes (see Section 23.3.3, “Initial Configuration of NDB Cluster”).

SQL nodes. On each of the machines designated to host SQL nodes, perform the following steps as the system `root` user:

1. Check your `/etc/passwd` and `/etc/group` files (or use whatever tools are provided by your operating system for managing users and groups) to see whether there is already a `mysql` group

and `mysql` user on the system. Some OS distributions create these as part of the operating system installation process. If they are not already present, create a new `mysql` user group, and then add a `mysql` user to this group:

```
$> groupadd mysql
$> useradd -g mysql -s /bin/false mysql
```

The syntax for `useradd` and `groupadd` may differ slightly on different versions of Unix, or they may have different names such as `adduser` and `addgroup`.

2. Change location to the directory containing the downloaded file, unpack the archive, and create a symbolic link named `mysql` to the `mysql` directory.

Note

The actual file and directory names vary according to the NDB Cluster version number.

```
$> cd /var/tmp
$> tar -C /usr/local -xzvf mysql-cluster-gpl-8.0.32-linux-glibc2.12-x86_64.tar.gz
$> ln -s /usr/local/mysql-cluster-gpl-8.0.32-linux-glibc2.12-x86_64 /usr/local/mysql
```

3. Change location to the `mysql` directory and set up the system databases using `mysqld --initialize` as shown here:

```
$> cd mysql
$> mysqld --initialize
```

This generates a random password for the MySQL `root` account. If you do *not* want the random password to be generated, you can substitute the `--initialize-insecure` option for `--initialize`. In either case, you should review Section 2.9.1, “Initializing the Data Directory”, for additional information before performing this step. See also Section 4.4.2, “mysql_secure_installation — Improve MySQL Installation Security”.

4. Set the necessary permissions for the MySQL server and data directories:

```
$> chown -R root .
$> chown -R mysql data
$> chgrp -R mysql .
```

5. Copy the MySQL startup script to the appropriate directory, make it executable, and set it to start when the operating system is booted up:

```
$> cp support-files/mysql.server /etc/rc.d/init.d/
$> chmod +x /etc/rc.d/init.d/mysql.server
$> chkconfig --add mysql.server
```

(The startup scripts directory may vary depending on your operating system and version—for example, in some Linux distributions, it is `/etc/init.d`.)

Here we use Red Hat's `chkconfig` for creating links to the startup scripts; use whatever means is appropriate for this purpose on your platform, such as `update-rc.d` on Debian.

Remember that the preceding steps must be repeated on each machine where an SQL node is to reside.

Data nodes. Installation of the data nodes does not require the `mysqld` binary. Only the NDB Cluster data node executable `ndbd` (single-threaded) or `ndbmtd` (multithreaded) is required. These binaries can also be found in the `.tar.gz` archive. Again, we assume that you have placed this archive in `/var/tmp`.

As system `root` (that is, after using `sudo`, `su root`, or your system's equivalent for temporarily assuming the system administrator account's privileges), perform the following steps to install the data node binaries on the data node hosts:

1. Change location to the `/var/tmp` directory, and extract the `ndbd` and `ndbmtd` binaries from the archive into a suitable directory such as `/usr/local/bin`:

```
$> cd /var/tmp
$> tar -zxvf mysql-cluster-gpl-8.0.32-linux-glibc2.12-x86_64.tar.gz
$> cd mysql-cluster-gpl-8.0.32-linux-glibc2.12-x86_64
$> cp bin/ndbd /usr/local/bin/ndbd
$> cp bin/ndbmtd /usr/local/bin/ndbmtd
```

   (You can safely delete the directory created by unpacking the downloaded archive, and the files it contains, from `/var/tmp` once `ndb_mgm` and `ndb_mgmd` have been copied to the executables directory.)

2. Change location to the directory into which you copied the files, and then make both of them executable:

```
$> cd /usr/local/bin
$> chmod +x ndb*
```

The preceding steps should be repeated on each data node host.

Although only one of the data node executables is required to run an NDB Cluster data node, we have shown you how to install both `ndbd` and `ndbmtd` in the preceding instructions. We recommend that you do this when installing or upgrading NDB Cluster, even if you plan to use only one of them, since this saves time and trouble in the event that you later decide to change from one to the other.

Note

The data directory on each machine hosting a data node is `/usr/local/mysql/data`. This piece of information is essential when configuring the management node. (See Section 23.3.3, “Initial Configuration of NDB Cluster”.)

Management nodes. Installation of the management node does not require the `mysqld` binary. Only the NDB Cluster management server (`ndb_mgmd`) is required; you most likely want to install the management client (`ndb_mgm`) as well. Both of these binaries also be found in the `.tar.gz` archive. Again, we assume that you have placed this archive in `/var/tmp`.

As system `root`, perform the following steps to install `ndb_mgmd` and `ndb_mgm` on the management node host:

1. Change location to the `/var/tmp` directory, and extract the `ndb_mgm` and `ndb_mgmd` from the archive into a suitable directory such as `/usr/local/bin`:

```
$> cd /var/tmp
$> tar -zxvf mysql-cluster-gpl-8.0.32-linux-glibc2.12-x86_64.tar.gz
$> cd mysql-cluster-gpl-8.0.32-linux-glibc2.12-x86_64
$> cp bin/ndb_mgm* /usr/local/bin
```

   (You can safely delete the directory created by unpacking the downloaded archive, and the files it contains, from `/var/tmp` once `ndb_mgm` and `ndb_mgmd` have been copied to the executables directory.)

2. Change location to the directory into which you copied the files, and then make both of them executable:

```
$> cd /usr/local/bin
$> chmod +x ndb_mgm*
```

In Section 23.3.3, “Initial Configuration of NDB Cluster”, we create configuration files for all of the nodes in our example NDB Cluster.

23.3.1.2 Installing NDB Cluster from RPM

This section covers the steps necessary to install the correct executables for each type of NDB Cluster 8.0 node using RPM packages supplied by Oracle.

As an alternative to the method described in this section, Oracle provides MySQL Repositories for NDB Cluster that are compatible with many common Linux distributions. Two repositories, listed here, are available for RPM-based distributions:

- For distributions using `yum` or `dnf`, you can use the MySQL Yum Repository for NDB Cluster. See *Installing MySQL NDB Cluster Using the Yum Repository*, for instructions and additional information.
- For SLES, you can use the MySQL SLES Repository for NDB Cluster. See *Installing MySQL NDB Cluster Using the SLES Repository*, for instructions and additional information.

RPMs are available for both 32-bit and 64-bit Linux platforms. The filenames for these RPMs use the following pattern:

```
mysql-cluster-community-data-node-8.0.32-1.el7.x86_64.rpm

mysql-cluster-license-component-ver-rev.distro.arch.rpm

    license:= {commercial | community}

    component: {management-server | data-node | server | client | other—see text}

    ver: major.minor.release

    rev: major[.minor]

    distro: {el6 | el7 | sles12}

    arch: {i686 | x86_64}
```

`license` indicates whether the RPM is part of a Commercial or Community release of NDB Cluster. In the remainder of this section, we assume for the examples that you are installing a Community release.

Possible values for *`component`*, with descriptions, can be found in the following table:

Table 23.6 Components of the NDB Cluster RPM distribution

Component	Description
`auto-installer` (DEPRECATED)	NDB Cluster Auto Installer program; see Section 23.3.8, “The NDB Cluster Auto-Installer (NO LONGER SUPPORTED)”, for usage
`client`	MySQL and `NDB` client programs; includes `mysql` client, `ndb_mgm` client, and other client tools
`common`	Character set and error message information needed by the MySQL server
`data-node`	`ndbd` and `ndbmtd` data node binaries
`devel`	Headers and library files needed for MySQL client development
`embedded`	Embedded MySQL server
`embedded-compat`	Backwards-compatible embedded MySQL server
`embedded-devel`	Header and library files for developing applications for embedded MySQL
`java`	JAR files needed for support of ClusterJ applications
`libs`	MySQL client libraries
`libs-compat`	Backwards-compatible MySQL client libraries
`management-server`	The NDB Cluster management server (`ndb_mgmd`)
`memcached`	Files needed to support `ndbmemcache`

Component	Description
minimal-debuginfo	Debug information for package server-minimal; useful when developing applications that use this package or when debugging this package
ndbclient	NDB client library for running NDB API and MGM API applications (libndbclient)
ndbclient-devel	Header and other files needed for developing NDB API and MGM API applications
nodejs	Files needed to set up Node.JS support for NDB Cluster
server	The MySQL server (mysqld) with NDB storage engine support included, and associated MySQL server programs
server-minimal	Minimal installation of the MySQL server for NDB and related tools
test	mysqltest, other MySQL test programs, and support files

A single bundle (.tar file) of all NDB Cluster RPMs for a given platform and architecture is also available. The name of this file follows the pattern shown here:

```
mysql-cluster-license-ver-rev.distro.arch.rpm-bundle.tar
```

You can extract the individual RPM files from this file using tar or your preferred tool for extracting archives.

The components required to install the three major types of NDB Cluster nodes are given in the following list:

- *Management node*: management-server
- *Data node*: data-node
- *SQL node*: server and common

In addition, the client RPM should be installed to provide the ndb_mgm management client on at least one management node. You may also wish to install it on SQL nodes, to have mysql and other MySQL client programs available on these. We discuss installation of nodes by type later in this section.

ver represents the three-part NDB storage engine version number in 8.0.*x* format, shown as 8.0.32 in the examples. rev provides the RPM revision number in *major*.*minor* format. In the examples shown in this section, we use 1.1 for this value.

The *distro* (Linux distribution) is one of rhel5 (Oracle Linux 5, Red Hat Enterprise Linux 4 and 5), el6 (Oracle Linux 6, Red Hat Enterprise Linux 6), el7 (Oracle Linux 7, Red Hat Enterprise Linux 7), or sles12 (SUSE Enterprise Linux 12). For the examples in this section, we assume that the host runs Oracle Linux 7, Red Hat Enterprise Linux 7, or the equivalent (el7).

arch is i686 for 32-bit RPMs and x86_64 for 64-bit versions. In the examples shown here, we assume a 64-bit platform.

The NDB Cluster version number in the RPM file names (shown here as 8.0.32) can vary according to the version which you are actually using. *It is very important that all of the Cluster RPMs to be installed have the same version number*. The architecture should also be appropriate to the machine on which the RPM is to be installed; in particular, you should keep in mind that 64-bit RPMs (x86_64) cannot be used with 32-bit operating systems (use i686 for the latter).

Data nodes. On a computer that is to host an NDB Cluster data node it is necessary to install only the `data-node` RPM. To do so, copy this RPM to the data node host, and run the following command as the system root user, replacing the name shown for the RPM as necessary to match that of the RPM downloaded from the MySQL website:

```
$> rpm -Uhv mysql-cluster-community-data-node-8.0.32-1.el7.x86_64.rpm
```

This installs the `ndbd` and `ndbmtd` data node binaries in `/usr/sbin`. Either of these can be used to run a data node process on this host.

SQL nodes. Copy the `server` and `common` RPMs to each machine to be used for hosting an NDB Cluster SQL node (`server` requires `common`). Install the `server` RPM by executing the following command as the system root user, replacing the name shown for the RPM as necessary to match the name of the RPM downloaded from the MySQL website:

```
$> rpm -Uhv mysql-cluster-community-server-8.0.32-1.el7.x86_64.rpm
```

This installs the MySQL server binary (`mysqld`), with `NDB` storage engine support, in the `/usr/sbin` directory. It also installs all needed MySQL Server support files and useful MySQL server programs, including the `mysql.server` and `mysqld_safe` startup scripts (in `/usr/share/mysql` and `/usr/bin`, respectively). The RPM installer should take care of general configuration issues (such as creating the `mysql` user and group, if needed) automatically.

Important

You must use the versions of these RPMs released for NDB Cluster; those released for the standard MySQL server do not provide support for the `NDB` storage engine.

To administer the SQL node (MySQL server), you should also install the `client` RPM, as shown here:

```
$> rpm -Uhv mysql-cluster-community-client-8.0.32-1.el7.x86_64.rpm
```

This installs the `mysql` client and other MySQL client programs, such as `mysqladmin` and `mysqldump`, to `/usr/bin`.

Management nodes. To install the NDB Cluster management server, it is necessary only to use the `management-server` RPM. Copy this RPM to the computer intended to host the management node, and then install it by running the following command as the system root user (replace the name shown for the RPM as necessary to match that of the `management-server` RPM downloaded from the MySQL website):

```
$> rpm -Uhv mysql-cluster-community-management-server-8.0.32-1.el7.x86_64.rpm
```

This RPM installs the management server binary `ndb_mgmd` in the `/usr/sbin` directory. While this is the only program actually required for running a management node, it is also a good idea to have the `ndb_mgm` NDB Cluster management client available as well. You can obtain this program, as well as other `NDB` client programs such as `ndb_desc` and `ndb_config`, by installing the `client` RPM as described previously.

See Section 2.5.4, “Installing MySQL on Linux Using RPM Packages from Oracle”, for general information about installing MySQL using RPMs supplied by Oracle.

After installing from RPM, you still need to configure the cluster; see Section 23.3.3, “Initial Configuration of NDB Cluster”, for the relevant information.

It is very important that all of the Cluster RPMs to be installed have the same version number. The *`architecture`* designation should also be appropriate to the machine on which the RPM is to be installed; in particular, you should keep in mind that 64-bit RPMs cannot be used with 32-bit operating systems.

Data nodes. On a computer that is to host a cluster data node it is necessary to install only the `server` RPM. To do so, copy this RPM to the data node host, and run the following command as

the system root user, replacing the name shown for the RPM as necessary to match that of the RPM downloaded from the MySQL website:

```
$> rpm -Uhv MySQL-Cluster-server-gpl-8.0.32-1.sles11.i386.rpm
```

Although this installs all NDB Cluster binaries, only the program `ndbd` or `ndbmtd` (both in `/usr/sbin`) is actually needed to run an NDB Cluster data node.

SQL nodes. On each machine to be used for hosting a cluster SQL node, install the `server` RPM by executing the following command as the system root user, replacing the name shown for the RPM as necessary to match the name of the RPM downloaded from the MySQL website:

```
$> rpm -Uhv MySQL-Cluster-server-gpl-8.0.32-1.sles11.i386.rpm
```

This installs the MySQL server binary (`mysqld`) with `NDB` storage engine support in the `/usr/sbin` directory, as well as all needed MySQL Server support files. It also installs the `mysql.server` and `mysqld_safe` startup scripts (in `/usr/share/mysql` and `/usr/bin`, respectively). The RPM installer should take care of general configuration issues (such as creating the `mysql` user and group, if needed) automatically.

To administer the SQL node (MySQL server), you should also install the `client` RPM, as shown here:

```
$> rpm -Uhv MySQL-Cluster-client-gpl-8.0.32-1.sles11.i386.rpm
```

This installs the `mysql` client program.

Management nodes. To install the NDB Cluster management server, it is necessary only to use the `server` RPM. Copy this RPM to the computer intended to host the management node, and then install it by running the following command as the system root user (replace the name shown for the RPM as necessary to match that of the `server` RPM downloaded from the MySQL website):

```
$> rpm -Uhv MySQL-Cluster-server-gpl-8.0.32-1.sles11.i386.rpm
```

Although this RPM installs many other files, only the management server binary `ndb_mgmd` (in the `/usr/sbin` directory) is actually required for running a management node. The `server` RPM also installs `ndb_mgm`, the `NDB` management client.

See Section 2.5.4, “Installing MySQL on Linux Using RPM Packages from Oracle”, for general information about installing MySQL using RPMs supplied by Oracle. See Section 23.3.3, “Initial Configuration of NDB Cluster”, for information about required post-installation configuration.

23.3.1.3 Installing NDB Cluster Using .deb Files

The section provides information about installing NDB Cluster on Debian and related Linux distributions such Ubuntu using the `.deb` files supplied by Oracle for this purpose.

Oracle also provides an NDB Cluster APT repository for Debian and other distributions. See *Installing MySQL NDB Cluster Using the APT Repository*, for instructions and additional information.

Oracle provides `.deb` installer files for NDB Cluster for 32-bit and 64-bit platforms. For a Debian-based system, only a single installer file is necessary. This file is named using the pattern shown here, according to the applicable NDB Cluster version, Debian version, and architecture:

```
mysql-cluster-gpl-ndbver-debiandebianver-arch.deb
```

Here, *`ndbver`* is the 3-part `NDB` engine version number, *`debianver`* is the major version of Debian (`8` or `9`), and *`arch`* is one of `i686` or `x86_64`. In the examples that follow, we assume you wish to install NDB 8.0.32 on a 64-bit Debian 9 system; in this case, the installer file is named `mysql-cluster-gpl-8.0.32-debian9-x86_64.deb-bundle.tar`.

Once you have downloaded the appropriate `.deb` file, you can untar it, and then install it from the command line using `dpkg`, like this:

```
$> dpkg -i mysql-cluster-gpl-8.0.32-debian9-i686.deb
```

You can also remove it using `dpkg` as shown here:

```
$> dpkg -r mysql
```

The installer file should also be compatible with most graphical package managers that work with `.deb` files, such as `GDebi` for the Gnome desktop.

The `.deb` file installs NDB Cluster under `/opt/mysql/server-version/`, where *`version`* is the 2-part release series version for the included MySQL server. For NDB 8.0, this is always `5.7`. The directory layout is the same as that for the generic Linux binary distribution (see Table 2.3, "MySQL Installation Layout for Generic Unix/Linux Binary Package"), with the exception that startup scripts and configuration files are found in `support-files` instead of `share`. All NDB Cluster executables, such as `ndb_mgm`, `ndbd`, and `ndb_mgmd`, are placed in the `bin` directory.

23.3.1.4 Building NDB Cluster from Source on Linux

This section provides information about compiling NDB Cluster on Linux and other Unix-like platforms. Building NDB Cluster from source is similar to building the standard MySQL Server, although it differs in a few key respects discussed here. For general information about building MySQL from source, see Section 2.8, "Installing MySQL from Source". For information about compiling NDB Cluster on Windows platforms, see Section 23.3.2.2, "Compiling and Installing NDB Cluster from Source on Windows".

Building MySQL NDB Cluster 8.0 requires using the MySQL Server 8.0 sources. These are available from the MySQL downloads page at https://dev.mysql.com/downloads/. The archived source file should have a name similar to `mysql-8.0.32.tar.gz`. You can also obtain the sources from GitHub at https://github.com/mysql/mysql-server.

Note

In previous versions, building of NDB Cluster from standard MySQL Server sources was not supported. In MySQL 8.0 and NDB Cluster 8.0, this is no longer the case—*both products are now built from the same sources*.

The `WITH_NDB` option for `CMake` causes the binaries for the management nodes, data nodes, and other NDB Cluster programs to be built; it also causes `mysqld` to be compiled with `NDB` storage engine support. This option (or, prior to NDB 8.0.31, `WITH_NDBCLUSTER`) is required when building NDB Cluster.

Important

The `WITH_NDB_JAVA` option is enabled by default. This means that, by default, if `CMake` cannot find the location of Java on your system, the configuration process fails; if you do not wish to enable Java and ClusterJ support, you must indicate this explicitly by configuring the build using `-DWITH_NDB_JAVA=OFF`. Use `WITH_CLASSPATH` to provide the Java classpath if needed.

For more information about `CMake` options specific to building NDB Cluster, see CMake Options for Compiling NDB Cluster.

After you have run `make && make install` (or your system's equivalent), the result is similar to what is obtained by unpacking a precompiled binary to the same location.

Management nodes. When building from source and running the default `make install`, the management server and management client binaries (`ndb_mgmd` and `ndb_mgm`) can be found in `/usr/local/mysql/bin`. Only `ndb_mgmd` is required to be present on a management node host; however, it is also a good idea to have `ndb_mgm` present on the same host machine. Neither of these executables requires a specific location on the host machine's file system.

Data nodes. The only executable required on a data node host is the data node binary `ndbd` or `ndbmtd`. (`mysqld`, for example, does not have to be present on the host machine.) By default, when building from source, this file is placed in the directory `/usr/local/mysql/bin`. For installing on multiple data node hosts, only `ndbd` or `ndbmtd` need be copied to the other host machine or machines.

(This assumes that all data node hosts use the same architecture and operating system; otherwise you may need to compile separately for each different platform.) The data node binary need not be in any particular location on the host's file system, as long as the location is known.

When compiling NDB Cluster from source, no special options are required for building multithreaded data node binaries. Configuring the build with `NDB` storage engine support causes `ndbmtd` to be built automatically; `make install` places the `ndbmtd` binary in the installation `bin` directory along with `mysqld`, `ndbd`, and `ndb_mgm`.

SQL nodes. If you compile MySQL with clustering support, and perform the default installation (using `make install` as the system `root` user), `mysqld` is placed in `/usr/local/mysql/bin`. Follow the steps given in Section 2.8, “Installing MySQL from Source” to make `mysqld` ready for use. If you want to run multiple SQL nodes, you can use a copy of the same `mysqld` executable and its associated support files on several machines. The easiest way to do this is to copy the entire `/usr/local/mysql` directory and all directories and files contained within it to the other SQL node host or hosts, then repeat the steps from Section 2.8, “Installing MySQL from Source” on each machine. If you configure the build with a nondefault `PREFIX` option, you must adjust the directory accordingly.

In Section 23.3.3, “Initial Configuration of NDB Cluster”, we create configuration files for all of the nodes in our example NDB Cluster.

23.3.2 Installing NDB Cluster on Windows

This section describes installation procedures for NDB Cluster on Windows hosts. NDB Cluster 8.0 binaries for Windows can be obtained from https://dev.mysql.com/downloads/cluster/. For information about installing NDB Cluster on Windows from a binary release provided by Oracle, see Section 23.3.2.1, “Installing NDB Cluster on Windows from a Binary Release”.

It is also possible to compile and install NDB Cluster from source on Windows using Microsoft Visual Studio. For more information, see Section 23.3.2.2, “Compiling and Installing NDB Cluster from Source on Windows”.

23.3.2.1 Installing NDB Cluster on Windows from a Binary Release

This section describes a basic installation of NDB Cluster on Windows using a binary “no-install” NDB Cluster release provided by Oracle, using the same 4-node setup outlined in the beginning of this section (see Section 23.3, “NDB Cluster Installation”), as shown in the following table:

Table 23.7 Network addresses of nodes in example cluster

Node	IP Address
Management node (`mgmd`)	198.51.100.10
SQL node (`mysqld`)	198.51.100.20
Data node "A" (`ndbd`)	198.51.100.30
Data node "B" (`ndbd`)	198.51.100.40

As on other platforms, the NDB Cluster host computer running an SQL node must have installed on it a MySQL Server binary (`mysqld.exe`). You should also have the MySQL client (`mysql.exe`) on this host. For management nodes and data nodes, it is not necessary to install the MySQL Server binary; however, each management node requires the management server daemon (`ndb_mgmd.exe`); each data node requires the data node daemon (`ndbd.exe` or `ndbmtd.exe`). For this example, we refer to `ndbd.exe` as the data node executable, but you can install `ndbmtd.exe`, the multithreaded version of this program, instead, in exactly the same way. You should also install the management client (`ndb_mgm.exe`) on the management server host. This section covers the steps necessary to install the correct Windows binaries for each type of NDB Cluster node.

Note

As with other Windows programs, NDB Cluster executables are named with the `.exe` file extension. However, it is not necessary to include the `.exe`

> extension when invoking these programs from the command line. Therefore, we often simply refer to these programs in this documentation as `mysqld`, `mysql`, `ndb_mgmd`, and so on. You should understand that, whether we refer (for example) to `mysqld` or `mysqld.exe`, either name means the same thing (the MySQL Server program).

For setting up an NDB Cluster using Oracles's `no-install` binaries, the first step in the installation process is to download the latest NDB Cluster Windows ZIP binary archive from https://dev.mysql.com/downloads/cluster/. This archive has a filename of the `mysql-cluster-gpl-`*`ver`*`-win`*`arch`*`.zip`, where *`ver`* is the `NDB` storage engine version (such as `8.0.32`), and *`arch`* is the architecture (`32` for 32-bit binaries, and `64` for 64-bit binaries). For example, the NDB Cluster 8.0.32 archive for 64-bit Windows systems is named `mysql-cluster-gpl-8.0.32-win64.zip`.

You can run 32-bit NDB Cluster binaries on both 32-bit and 64-bit versions of Windows; however, 64-bit NDB Cluster binaries can be used only on 64-bit versions of Windows. If you are using a 32-bit version of Windows on a computer that has a 64-bit CPU, then you must use the 32-bit NDB Cluster binaries.

To minimize the number of files that need to be downloaded from the Internet or copied between machines, we start with the computer where you intend to run the SQL node.

SQL node. We assume that you have placed a copy of the archive in the directory `C:\Documents and Settings\`*`username`*`\My Documents\Downloads` on the computer having the IP address 198.51.100.20, where *`username`* is the name of the current user. (You can obtain this name using `ECHO %USERNAME%` on the command line.) To install and run NDB Cluster executables as Windows services, this user should be a member of the `Administrators` group.

Extract all the files from the archive. The Extraction Wizard integrated with Windows Explorer is adequate for this task. (If you use a different archive program, be sure that it extracts all files and directories from the archive, and that it preserves the archive's directory structure.) When you are asked for a destination directory, enter `C:\`, which causes the Extraction Wizard to extract the archive to the directory `C:\mysql-cluster-gpl-`*`ver`*`-win`*`arch`*. Rename this directory to `C:\mysql`.

It is possible to install the NDB Cluster binaries to directories other than `C:\mysql\bin`; however, if you do so, you must modify the paths shown in this procedure accordingly. In particular, if the MySQL Server (SQL node) binary is installed to a location other than `C:\mysql` or `C:\Program Files\MySQL\MySQL Server 8.0`, or if the SQL node's data directory is in a location other than `C:\mysql\data` or `C:\Program Files\MySQL\MySQL Server 8.0\data`, extra configuration options must be used on the command line or added to the `my.ini` or `my.cnf` file when starting the SQL node. For more information about configuring a MySQL Server to run in a nonstandard location, see Section 2.3.4, “Installing MySQL on Microsoft Windows Using a `noinstall` ZIP Archive”.

For a MySQL Server with NDB Cluster support to run as part of an NDB Cluster, it must be started with the options `--ndbcluster` and `--ndb-connectstring`. While you can specify these options on the command line, it is usually more convenient to place them in an option file. To do this, create a new text file in Notepad or another text editor. Enter the following configuration information into this file:

```
[mysqld]
# Options for mysqld process:
ndbcluster                      # run NDB storage engine
ndb-connectstring=198.51.100.10  # location of management server
```

You can add other options used by this MySQL Server if desired (see Section 2.3.4.2, “Creating an Option File”), but the file must contain the options shown, at a minimum. Save this file as `C:\mysql\my.ini`. This completes the installation and setup for the SQL node.

Data nodes. An NDB Cluster data node on a Windows host requires only a single executable, one of either `ndbd.exe` or `ndbmtd.exe`. For this example, we assume that you are using `ndbd.exe`, but the same instructions apply when using `ndbmtd.exe`. On each computer where you wish to run

a data node (the computers having the IP addresses 198.51.100.30 and 198.51.100.40), create the directories `C:\mysql`, `C:\mysql\bin`, and `C:\mysql\cluster-data`; then, on the computer where you downloaded and extracted the `no-install` archive, locate `ndbd.exe` in the `C:\mysql\bin` directory. Copy this file to the `C:\mysql\bin` directory on each of the two data node hosts.

To function as part of an NDB Cluster, each data node must be given the address or hostname of the management server. You can supply this information on the command line using the `--ndb-connectstring` or `-c` option when starting each data node process. However, it is usually preferable to put this information in an option file. To do this, create a new text file in Notepad or another text editor and enter the following text:

```
[mysql_cluster]
# Options for data node process:
ndb-connectstring=198.51.100.10  # location of management server
```

Save this file as `C:\mysql\my.ini` on the data node host. Create another text file containing the same information and save it on as `C:mysql\my.ini` on the other data node host, or copy the my.ini file from the first data node host to the second one, making sure to place the copy in the second data node's `C:\mysql` directory. Both data node hosts are now ready to be used in the NDB Cluster, which leaves only the management node to be installed and configured.

Management node. The only executable program required on a computer used for hosting an NDB Cluster management node is the management server program `ndb_mgmd.exe`. However, in order to administer the NDB Cluster once it has been started, you should also install the NDB Cluster management client program `ndb_mgm.exe` on the same machine as the management server. Locate these two programs on the machine where you downloaded and extracted the `no-install` archive; this should be the directory `C:\mysql\bin` on the SQL node host. Create the directory `C:\mysql\bin` on the computer having the IP address 198.51.100.10, then copy both programs to this directory.

You should now create two configuration files for use by `ndb_mgmd.exe`:

1. A local configuration file to supply configuration data specific to the management node itself. Typically, this file needs only to supply the location of the NDB Cluster global configuration file (see item 2).

   To create this file, start a new text file in Notepad or another text editor, and enter the following information:

   ```
   [mysql_cluster]
   # Options for management node process
   config-file=C:/mysql/bin/config.ini
   ```

   Save this file as the text file `C:\mysql\bin\my.ini`.

2. A global configuration file from which the management node can obtain configuration information governing the NDB Cluster as a whole. At a minimum, this file must contain a section for each node in the NDB Cluster, and the IP addresses or hostnames for the management node and all data nodes (`HostName` configuration parameter). It is also advisable to include the following additional information:

   - The IP address or hostname of any SQL nodes
   - The data memory and index memory allocated to each data node (`DataMemory` and `IndexMemory` configuration parameters)
   - The number of fragment replicas, using the `NoOfReplicas` configuration parameter (see Section 23.2.2, “NDB Cluster Nodes, Node Groups, Fragment Replicas, and Partitions”)
   - The directory where each data node stores it data and log file, and the directory where the management node keeps its log files (in both cases, the `DataDir` configuration parameter)

   Create a new text file using a text editor such as Notepad, and input the following information:

```
[ndbd default]
# Options affecting ndbd processes on all data nodes:
NoOfReplicas=2                      # Number of fragment replicas
DataDir=C:/mysql/cluster-data       # Directory for each data node's data files
                                    # Forward slashes used in directory path,
                                    # rather than backslashes. This is correct;
                                    # see Important note in text
DataMemory=80M    # Memory allocated to data storage
IndexMemory=18M   # Memory allocated to index storage
                  # For DataMemory and IndexMemory, we have used the
                  # default values. Since the "world" database takes up
                  # only about 500KB, this should be more than enough for
                  # this example Cluster setup.

[ndb_mgmd]
# Management process options:
HostName=198.51.100.10              # Hostname or IP address of management node
DataDir=C:/mysql/bin/cluster-logs   # Directory for management node log files

[ndbd]
# Options for data node "A":
                                # (one [ndbd] section per data node)
HostName=198.51.100.30          # Hostname or IP address

[ndbd]
# Options for data node "B":
HostName=198.51.100.40          # Hostname or IP address

[mysqld]
# SQL node options:
HostName=198.51.100.20          # Hostname or IP address
```

Save this file as the text file `C:\mysql\bin\config.ini`.

Important

A single backslash character (`\`) cannot be used when specifying directory paths in program options or configuration files used by NDB Cluster on Windows. Instead, you must either escape each backslash character with a second backslash (`\\`), or replace the backslash with a forward slash character (`/`). For example, the following line from the `[ndb_mgmd]` section of an NDB Cluster `config.ini` file does not work:

```
DataDir=C:\mysql\bin\cluster-logs
```

Instead, you may use either of the following:

```
DataDir=C:\\mysql\\bin\\cluster-logs  # Escaped backslashes
```

```
DataDir=C:/mysql/bin/cluster-logs     # Forward slashes
```

For reasons of brevity and legibility, we recommend that you use forward slashes in directory paths used in NDB Cluster program options and configuration files on Windows.

23.3.2.2 Compiling and Installing NDB Cluster from Source on Windows

Oracle provides precompiled NDB Cluster binaries for Windows which should be adequate for most users. However, if you wish, it is also possible to compile NDB Cluster for Windows from source code. The procedure for doing this is almost identical to the procedure used to compile the standard MySQL Server binaries for Windows, and uses the same tools. However, there are two major differences:

- Building MySQL NDB Cluster 8.0 requires using the MySQL Server 8.0 sources. These are available from the MySQL downloads page at https://dev.mysql.com/downloads/. The archived source file should have a name similar to `mysql-8.0.32.tar.gz`. You can also obtain the sources from GitHub at https://github.com/mysql/mysql-server.

- You must configure the build using the `WITH_NDB` option in addition to any other build options you wish to use with `CMake`. `WITH_NDBCLUSTER` is also supported for backwards compatibility, but is deprecated as of NDB 8.0.31.

Important

The `WITH_NDB_JAVA` option is enabled by default. This means that, by default, if `CMake` cannot find the location of Java on your system, the configuration process fails; if you do not wish to enable Java and ClusterJ support, you must indicate this explicitly by configuring the build using `-DWITH_NDB_JAVA=OFF`. (Bug #12379735) Use `WITH_CLASSPATH` to provide the Java classpath if needed.

For more information about `CMake` options specific to building NDB Cluster, see CMake Options for Compiling NDB Cluster.

Once the build process is complete, you can create a Zip archive containing the compiled binaries; Section 2.8.4, “Installing MySQL Using a Standard Source Distribution” provides the commands needed to perform this task on Windows systems. The NDB Cluster binaries can be found in the `bin` directory of the resulting archive, which is equivalent to the `no-install` archive, and which can be installed and configured in the same manner. For more information, see Section 23.3.2.1, “Installing NDB Cluster on Windows from a Binary Release”.

23.3.2.3 Initial Startup of NDB Cluster on Windows

Once the NDB Cluster executables and needed configuration files are in place, performing an initial start of the cluster is simply a matter of starting the NDB Cluster executables for all nodes in the cluster. Each cluster node process must be started separately, and on the host computer where it resides. The management node should be started first, followed by the data nodes, and then finally by any SQL nodes.

1. On the management node host, issue the following command from the command line to start the management node process. The output should appear similar to what is shown here:

```
C:\mysql\bin> ndb_mgmd
2010-06-23 07:53:34 [MgmtSrvr] INFO -- NDB Cluster Management Server. mysql-8.0.34-ndb-8.0.34
2010-06-23 07:53:34 [MgmtSrvr] INFO -- Reading cluster configuration from 'config.ini'
```

   The management node process continues to print logging output to the console. This is normal, because the management node is not running as a Windows service. (If you have used NDB Cluster on a Unix-like platform such as Linux, you may notice that the management node's default behavior in this regard on Windows is effectively the opposite of its behavior on Unix systems, where it runs by default as a Unix daemon process. This behavior is also true of NDB Cluster data node processes running on Windows.) For this reason, do not close the window in which `ndb_mgmd.exe` is running; doing so kills the management node process. (See Section 23.3.2.4, “Installing NDB Cluster Processes as Windows Services”, where we show how to install and run NDB Cluster processes as Windows services.)

   The required `-f` option tells the management node where to find the global configuration file (`config.ini`). The long form of this option is `--config-file`.

Important

An NDB Cluster management node caches the configuration data that it reads from `config.ini`; once it has created a configuration cache, it ignores the `config.ini` file on subsequent starts unless forced to do otherwise. This means that, if the management node fails to start due to an error in this file, you must make the management node re-read `config.ini` after you have corrected any errors in it. You can do this by starting `ndb_mgmd.exe` with the `--reload` or `--initial` option on the

command line. Either of these options works to refresh the configuration cache.

It is not necessary or advisable to use either of these options in the management node's `my.ini` file.

2. On each of the data node hosts, run the command shown here to start the data node processes:

```
C:\mysql\bin> ndbd
2010-06-23 07:53:46 [ndbd] INFO -- Configuration fetched from 'localhost:1186', generation: 1
```

In each case, the first line of output from the data node process should resemble what is shown in the preceding example, and is followed by additional lines of logging output. As with the management node process, this is normal, because the data node is not running as a Windows service. For this reason, do not close the console window in which the data node process is running; doing so kills `ndbd.exe`. (For more information, see Section 23.3.2.4, "Installing NDB Cluster Processes as Windows Services".)

3. Do not start the SQL node yet; it cannot connect to the cluster until the data nodes have finished starting, which may take some time. Instead, in a new console window on the management node host, start the NDB Cluster management client `ndb_mgm.exe`, which should be in `C:\mysql\bin` on the management node host. (Do not try to re-use the console window where `ndb_mgmd.exe` is running by typing **CTRL**+**C**, as this kills the management node.) The resulting output should look like this:

```
C:\mysql\bin> ndb_mgm
-- NDB Cluster -- Management Client --
ndb_mgm>
```

When the prompt `ndb_mgm>` appears, this indicates that the management client is ready to receive NDB Cluster management commands. You can observe the status of the data nodes as they start by entering `ALL STATUS` at the management client prompt. This command causes a running report of the data nodes's startup sequence, which should look something like this:

```
ndb_mgm> ALL STATUS
Connected to Management Server at: localhost:1186
Node 2: starting (Last completed phase 3) (mysql-8.0.34-ndb-8.0.34)
Node 3: starting (Last completed phase 3) (mysql-8.0.34-ndb-8.0.34)

Node 2: starting (Last completed phase 4) (mysql-8.0.34-ndb-8.0.34)
Node 3: starting (Last completed phase 4) (mysql-8.0.34-ndb-8.0.34)

Node 2: Started (version 8.0.34)
Node 3: Started (version 8.0.34)

ndb_mgm>
```

Note

Commands issued in the management client are not case-sensitive; we use uppercase as the canonical form of these commands, but you are not required to observe this convention when inputting them into the `ndb_mgm` client. For more information, see Section 23.6.1, "Commands in the NDB Cluster Management Client".

The output produced by `ALL STATUS` is likely to vary from what is shown here, according to the speed at which the data nodes are able to start, the release version number of the NDB Cluster software you are using, and other factors. What is significant is that, when you see that both data nodes have started, you are ready to start the SQL node.

You can leave `ndb_mgm.exe` running; it has no negative impact on the performance of the NDB Cluster, and we use it in the next step to verify that the SQL node is connected to the cluster after you have started it.

4. On the computer designated as the SQL node host, open a console window and navigate to the directory where you unpacked the NDB Cluster binaries (if you are following our example, this is `C:\mysql\bin`).

   Start the SQL node by invoking `mysqld.exe` from the command line, as shown here:

   ```
   C:\mysql\bin> mysqld --console
   ```

   The `--console` option causes logging information to be written to the console, which can be helpful in the event of problems. (Once you are satisfied that the SQL node is running in a satisfactory manner, you can stop it and restart it out without the `--console` option, so that logging is performed normally.)

   In the console window where the management client (`ndb_mgm.exe`) is running on the management node host, enter the `SHOW` command, which should produce output similar to what is shown here:

   ```
   ndb_mgm> SHOW
   Connected to Management Server at: localhost:1186
   Cluster Configuration
   ---------------------
   [ndbd(NDB)]     2 node(s)
   id=2    @198.51.100.30  (Version: 8.0.34-ndb-8.0.34, Nodegroup: 0, *)
   id=3    @198.51.100.40  (Version: 8.0.34-ndb-8.0.34, Nodegroup: 0)

   [ndb_mgmd(MGM)] 1 node(s)
   id=1    @198.51.100.10  (Version: 8.0.34-ndb-8.0.34)

   [mysqld(API)]   1 node(s)
   id=4    @198.51.100.20  (Version: 8.0.34-ndb-8.0.34)
   ```

   You can also verify that the SQL node is connected to the NDB Cluster in the `mysql` client (`mysql.exe`) using the `SHOW ENGINE NDB STATUS` statement.

You should now be ready to work with database objects and data using NDB Cluster 's `NDBCLUSTER` storage engine. See Section 23.3.5, “NDB Cluster Example with Tables and Data”, for more information and examples.

You can also install `ndb_mgmd.exe`, `ndbd.exe`, and `ndbmtd.exe` as Windows services. For information on how to do this, see Section 23.3.2.4, “Installing NDB Cluster Processes as Windows Services”).

23.3.2.4 Installing NDB Cluster Processes as Windows Services

Once you are satisfied that NDB Cluster is running as desired, you can install the management nodes and data nodes as Windows services, so that these processes are started and stopped automatically whenever Windows is started or stopped. This also makes it possible to control these processes from the command line with the appropriate `SC START` and `SC STOP` commands, or using the Windows graphical `Services` utility. `NET START` and `NET STOP` commands can also be used.

Installing programs as Windows services usually must be done using an account that has Administrator rights on the system.

To install the management node as a service on Windows, invoke `ndb_mgmd.exe` from the command line on the machine hosting the management node, using the `--install` option, as shown here:

```
C:\> C:\mysql\bin\ndb_mgmd.exe --install
Installing service 'NDB Cluster Management Server'
  as '"C:\mysql\bin\ndbd.exe" "--service=ndb_mgmd"'
Service successfully installed.
```

Important

When installing an NDB Cluster program as a Windows service, you should always specify the complete path; otherwise the service installation may fail with the error `The system cannot find the file specified`.

The `--install` option must be used first, ahead of any other options that might be specified for `ndb_mgmd.exe`. However, it is preferable to specify such options in an options file instead. If your options file is not in one of the default locations as shown in the output of `ndb_mgmd.exe --help`, you can specify the location using the `--config-file` option.

Now you should be able to start and stop the management server like this:

```
C:\> SC START ndb_mgmd

C:\> SC STOP ndb_mgmd
```

Note

If using `NET` commands, you can also start or stop the management server as a Windows service using the descriptive name, as shown here:

```
C:\> NET START 'NDB Cluster Management Server'
The NDB Cluster Management Server service is starting.
The NDB Cluster Management Server service was started successfully.

C:\> NET STOP  'NDB Cluster Management Server'
The NDB Cluster Management Server service is stopping..
The NDB Cluster Management Server service was stopped successfully.
```

It is usually simpler to specify a short service name or to permit the default service name to be used when installing the service, and then reference that name when starting or stopping the service. To specify a service name other than `ndb_mgmd`, append it to the `--install` option, as shown in this example:

```
C:\> C:\mysql\bin\ndb_mgmd.exe --install=mgmd1
Installing service 'NDB Cluster Management Server'
  as '"C:\mysql\bin\ndb_mgmd.exe" "--service=mgmd1"'
Service successfully installed.
```

Now you should be able to start or stop the service using the name you have specified, like this:

```
C:\> SC START mgmd1

C:\> SC STOP mgmd1
```

To remove the management node service, use `SC DELETE service_name`:

```
C:\> SC DELETE mgmd1
```

Alternatively, invoke `ndb_mgmd.exe` with the `--remove` option, as shown here:

```
C:\> C:\mysql\bin\ndb_mgmd.exe --remove
Removing service 'NDB Cluster Management Server'
Service successfully removed.
```

If you installed the service using a service name other than the default, pass the service name as the value of the `ndb_mgmd.exe --remove` option, like this:

```
C:\> C:\mysql\bin\ndb_mgmd.exe --remove=mgmd1
Removing service 'mgmd1'
Service successfully removed.
```

Installation of an NDB Cluster data node process as a Windows service can be done in a similar fashion, using the `--install` option for `ndbd.exe` (or `ndbmtd.exe`), as shown here:

```
C:\> C:\mysql\bin\ndbd.exe --install
Installing service 'NDB Cluster Data Node Daemon' as '"C:\mysql\bin\ndbd.exe" "--service=ndbd"'
Service successfully installed.
```

Now you can start or stop the data node as shown in the following example:

```
C:\> SC START ndbd

C:\> SC STOP ndbd
```

To remove the data node service, use `SC DELETE service_name`:

```
C:\> SC DELETE ndbd
```

Alternatively, invoke `ndbd.exe` with the `--remove` option, as shown here:

```
C:\> C:\mysql\bin\ndbd.exe --remove
Removing service 'NDB Cluster Data Node Daemon'
Service successfully removed.
```

As with `ndb_mgmd.exe` (and `mysqld.exe`), when installing `ndbd.exe` as a Windows service, you can also specify a name for the service as the value of `--install`, and then use it when starting or stopping the service, like this:

```
C:\> C:\mysql\bin\ndbd.exe --install=dnode1
Installing service 'dnode1' as '"C:\mysql\bin\ndbd.exe" "--service=dnode1"'
Service successfully installed.

C:\> SC START dnode1

C:\> SC STOP dnode1
```

If you specified a service name when installing the data node service, you can use this name when removing it as well, as shown here:

```
C:\> SC DELETE dnode1
```

Alternatively, you can pass the service name as the value of the `ndbd.exe --remove` option, as shown here:

```
C:\> C:\mysql\bin\ndbd.exe --remove=dnode1
Removing service 'dnode1'
Service successfully removed.
```

Installation of the SQL node as a Windows service, starting the service, stopping the service, and removing the service are done in a similar fashion, using `mysqld --install`, `SC START`, `SC STOP`, and `SC DELETE` (or `mysqld --remove`). `NET` commands can also be used to start or stop a service. For additional information, see Section 2.3.4.8, “Starting MySQL as a Windows Service”.

23.3.3 Initial Configuration of NDB Cluster

In this section, we discuss manual configuration of an installed NDB Cluster by creating and editing configuration files.

For our four-node, four-host NDB Cluster (see Cluster nodes and host computers), it is necessary to write four configuration files, one per node host.

- Each data node or SQL node requires a `my.cnf` file that provides two pieces of information: a *connection string* that tells the node where to find the management node, and a line telling the MySQL server on this host (the machine hosting the data node) to enable the `NDBCLUSTER` storage engine.

  For more information on connection strings, see Section 23.4.3.3, “NDB Cluster Connection Strings”.

- The management node needs a `config.ini` file telling it how many fragment replicas to maintain, how much memory to allocate for data and indexes on each data node, where to find the data nodes, where to save data to disk on each data node, and where to find any SQL nodes.

Configuring the data nodes and SQL nodes. The `my.cnf` file needed for the data nodes is fairly simple. The configuration file should be located in the `/etc` directory and can be edited using any text editor. (Create the file if it does not exist.) For example:

```
$> vi /etc/my.cnf
```

Note

We show `vi` being used here to create the file, but any text editor should work just as well.

For each data node and SQL node in our example setup, `my.cnf` should look like this:

```
[mysqld]
# Options for mysqld process:
ndbcluster                      # run NDB storage engine

[mysql_cluster]
# Options for NDB Cluster processes:
ndb-connectstring=198.51.100.10  # location of management server
```

After entering the preceding information, save this file and exit the text editor. Do this for the machines hosting data node “A”, data node “B”, and the SQL node.

Important

Once you have started a `mysqld` process with the `ndbcluster` and `ndb-connectstring` parameters in the `[mysqld]` and `[mysql_cluster]` sections of the `my.cnf` file as shown previously, you cannot execute any `CREATE TABLE` or `ALTER TABLE` statements without having actually started the cluster. Otherwise, these statements fail with an error. This is by design.

Configuring the management node. The first step in configuring the management node is to create the directory in which the configuration file can be found and then to create the file itself. For example (running as `root`):

```
$> mkdir /var/lib/mysql-cluster
$> cd /var/lib/mysql-cluster
$> vi config.ini
```

For our representative setup, the `config.ini` file should read as follows:

```
[ndbd default]
# Options affecting ndbd processes on all data nodes:
NoOfReplicas=2    # Number of fragment replicas
DataMemory=98M    # How much memory to allocate for data storage

[ndb_mgmd]
# Management process options:
HostName=198.51.100.10          # Hostname or IP address of management node
DataDir=/var/lib/mysql-cluster  # Directory for management node log files

[ndbd]
# Options for data node "A":
                                # (one [ndbd] section per data node)
HostName=198.51.100.30          # Hostname or IP address
NodeId=2                        # Node ID for this data node
DataDir=/usr/local/mysql/data   # Directory for this data node's data files

[ndbd]
# Options for data node "B":
```

```
HostName=198.51.100.40          # Hostname or IP address
NodeId=3                        # Node ID for this data node
DataDir=/usr/local/mysql/data   # Directory for this data node's data files

[mysqld]
# SQL node options:
HostName=198.51.100.20          # Hostname or IP address
                                # (additional mysqld connections can be
                                # specified for this node for various
                                # purposes such as running ndb_restore)
```

Note

The `world` database can be downloaded from https://dev.mysql.com/doc/index-other.html.

After all the configuration files have been created and these minimal options have been specified, you are ready to proceed with starting the cluster and verifying that all processes are running. We discuss how this is done in Section 23.3.4, “Initial Startup of NDB Cluster”.

For more detailed information about the available NDB Cluster configuration parameters and their uses, see Section 23.4.3, “NDB Cluster Configuration Files”, and Section 23.4, “Configuration of NDB Cluster”. For configuration of NDB Cluster as relates to making backups, see Section 23.6.8.3, “Configuration for NDB Cluster Backups”.

Note

The default port for Cluster management nodes is 1186; the default port for data nodes is 2202. However, the cluster can automatically allocate ports for data nodes from those that are already free.

23.3.4 Initial Startup of NDB Cluster

Starting the cluster is not very difficult after it has been configured. Each cluster node process must be started separately, and on the host where it resides. The management node should be started first, followed by the data nodes, and then finally by any SQL nodes:

1. On the management host, issue the following command from the system shell to start the management node process:

   ```
   $> ndb_mgmd --initial -f /var/lib/mysql-cluster/config.ini
   ```

   The first time that it is started, `ndb_mgmd` must be told where to find its configuration file, using the `-f` or `--config-file` option. This option requires that `--initial` or `--reload` also be specified; see Section 23.5.4, “ndb_mgmd — The NDB Cluster Management Server Daemon”, for details.

2. On each of the data node hosts, run this command to start the `ndbd` process:

   ```
   $> ndbd
   ```

3. If you used RPM files to install MySQL on the cluster host where the SQL node is to reside, you can (and should) use the supplied startup script to start the MySQL server process on the SQL node.

If all has gone well, and the cluster has been set up correctly, the cluster should now be operational. You can test this by invoking the `ndb_mgm` management node client. The output should look like that shown here, although you might see some slight differences in the output depending upon the exact version of MySQL that you are using:

```
$> ndb_mgm
-- NDB Cluster -- Management Client --
```

```
ndb_mgm> SHOW
Connected to Management Server at: localhost:1186
Cluster Configuration
---------------------
[ndbd(NDB)]     2 node(s)
id=2    @198.51.100.30  (Version: 8.0.34-ndb-8.0.34, Nodegroup: 0, *)
id=3    @198.51.100.40  (Version: 8.0.34-ndb-8.0.34, Nodegroup: 0)

[ndb_mgmd(MGM)] 1 node(s)
id=1    @198.51.100.10  (Version: 8.0.34-ndb-8.0.34)

[mysqld(API)]   1 node(s)
id=4    @198.51.100.20  (Version: 8.0.34-ndb-8.0.34)
```

The SQL node is referenced here as `[mysqld(API)]`, which reflects the fact that the `mysqld` process is acting as an NDB Cluster API node.

Note

The IP address shown for a given NDB Cluster SQL or other API node in the output of `SHOW` is the address used by the SQL or API node to connect to the cluster data nodes, and not to any management node.

You should now be ready to work with databases, tables, and data in NDB Cluster. See Section 23.3.5, “NDB Cluster Example with Tables and Data”, for a brief discussion.

23.3.5 NDB Cluster Example with Tables and Data

Note

The information in this section applies to NDB Cluster running on both Unix and Windows platforms.

Working with database tables and data in NDB Cluster is not much different from doing so in standard MySQL. There are two key points to keep in mind:

- For a table to be replicated in the cluster, it must use the `NDBCLUSTER` storage engine. To specify this, use the `ENGINE=NDBCLUSTER` or `ENGINE=NDB` option when creating the table:

  ```
  CREATE TABLE tbl_name (col_name column_definitions) ENGINE=NDBCLUSTER;
  ```

  Alternatively, for an existing table that uses a different storage engine, use `ALTER TABLE` to change the table to use `NDBCLUSTER`:

  ```
  ALTER TABLE tbl_name ENGINE=NDBCLUSTER;
  ```

- Every `NDBCLUSTER` table has a primary key. If no primary key is defined by the user when a table is created, the `NDBCLUSTER` storage engine automatically generates a hidden one. Such a key takes up space just as does any other table index. (It is not uncommon to encounter problems due to insufficient memory for accommodating these automatically created indexes.)

If you are importing tables from an existing database using the output of `mysqldump`, you can open the SQL script in a text editor and add the `ENGINE` option to any table creation statements, or replace any existing `ENGINE` options. Suppose that you have the `world` sample database on another MySQL server that does not support NDB Cluster, and you want to export the `City` table:

```
$> mysqldump --add-drop-table world City > city_table.sql
```

The resulting `city_table.sql` file contains this table creation statement (and the `INSERT` statements necessary to import the table data):

```
DROP TABLE IF EXISTS `City`;
CREATE TABLE `City` (
```

```
  `ID` int(11) NOT NULL auto_increment,
  `Name` char(35) NOT NULL default '',
  `CountryCode` char(3) NOT NULL default '',
  `District` char(20) NOT NULL default '',
  `Population` int(11) NOT NULL default '0',
  PRIMARY KEY  (`ID`)
) ENGINE=MyISAM DEFAULT CHARSET=latin1;

INSERT INTO `City` VALUES (1,'Kabul','AFG','Kabol',1780000);
INSERT INTO `City` VALUES (2,'Qandahar','AFG','Qandahar',237500);
INSERT INTO `City` VALUES (3,'Herat','AFG','Herat',186800);
(remaining INSERT statements omitted)
```

You need to make sure that MySQL uses the `NDBCLUSTER` storage engine for this table. There are two ways that this can be accomplished. One of these is to modify the table definition *before* importing it into the Cluster database. Using the `City` table as an example, modify the `ENGINE` option of the definition as follows:

```
DROP TABLE IF EXISTS `City`;
CREATE TABLE `City` (
  `ID` int(11) NOT NULL auto_increment,
  `Name` char(35) NOT NULL default '',
  `CountryCode` char(3) NOT NULL default '',
  `District` char(20) NOT NULL default '',
  `Population` int(11) NOT NULL default '0',
  PRIMARY KEY  (`ID`)
) ENGINE=NDBCLUSTER DEFAULT CHARSET=latin1;

INSERT INTO `City` VALUES (1,'Kabul','AFG','Kabol',1780000);
INSERT INTO `City` VALUES (2,'Qandahar','AFG','Qandahar',237500);
INSERT INTO `City` VALUES (3,'Herat','AFG','Herat',186800);
(remaining INSERT statements omitted)
```

This must be done for the definition of each table that is to be part of the clustered database. The easiest way to accomplish this is to do a search-and-replace on the file that contains the definitions and replace all instances of `TYPE=`*`engine_name`* or `ENGINE=`*`engine_name`* with `ENGINE=NDBCLUSTER`. If you do not want to modify the file, you can use the unmodified file to create the tables, and then use `ALTER TABLE` to change their storage engine. The particulars are given later in this section.

Assuming that you have already created a database named `world` on the SQL node of the cluster, you can then use the `mysql` command-line client to read `city_table.sql`, and create and populate the corresponding table in the usual manner:

```
$> mysql world < city_table.sql
```

It is very important to keep in mind that the preceding command must be executed on the host where the SQL node is running (in this case, on the machine with the IP address `198.51.100.20`).

To create a copy of the entire `world` database on the SQL node, use `mysqldump` on the noncluster server to export the database to a file named `world.sql` (for example, in the `/tmp` directory). Then modify the table definitions as just described and import the file into the SQL node of the cluster like this:

```
$> mysql world < /tmp/world.sql
```

If you save the file to a different location, adjust the preceding instructions accordingly.

Running `SELECT` queries on the SQL node is no different from running them on any other instance of a MySQL server. To run queries from the command line, you first need to log in to the MySQL Monitor in the usual way (specify the `root` password at the `Enter password:` prompt):

```
$> mysql -u root -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 1 to server version: 8.0.34-ndb-8.0.34
```

```
Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql>
```

We simply use the MySQL server's `root` account and assume that you have followed the standard security precautions for installing a MySQL server, including setting a strong `root` password. For more information, see Section 2.9.4, “Securing the Initial MySQL Account”.

It is worth taking into account that NDB Cluster nodes do *not* make use of the MySQL privilege system when accessing one another. Setting or changing MySQL user accounts (including the `root` account) effects only applications that access the SQL node, not interaction between nodes. See Section 23.6.20.2, “NDB Cluster and MySQL Privileges”, for more information.

If you did not modify the `ENGINE` clauses in the table definitions prior to importing the SQL script, you should run the following statements at this point:

```
mysql> USE world;
mysql> ALTER TABLE City ENGINE=NDBCLUSTER;
mysql> ALTER TABLE Country ENGINE=NDBCLUSTER;
mysql> ALTER TABLE CountryLanguage ENGINE=NDBCLUSTER;
```

Selecting a database and running a `SELECT` query against a table in that database is also accomplished in the usual manner, as is exiting the MySQL Monitor:

```
mysql> USE world;
mysql> SELECT Name, Population FROM City ORDER BY Population DESC LIMIT 5;
+-----------+------------+
| Name      | Population |
+-----------+------------+
| Bombay    |   10500000 |
| Seoul     |    9981619 |
| São Paulo |    9968485 |
| Shanghai  |    9696300 |
| Jakarta   |    9604900 |
+-----------+------------+
5 rows in set (0.34 sec)

mysql> \q
Bye

$>
```

Applications that use MySQL can employ standard APIs to access `NDB` tables. It is important to remember that your application must access the SQL node, and not the management or data nodes. This brief example shows how we might execute the `SELECT` statement just shown by using the PHP 5.X `mysqli` extension running on a Web server elsewhere on the network:

```
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
  "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
  <meta http-equiv="Content-Type"
          content="text/html; charset=iso-8859-1">
  <title>SIMPLE mysqli SELECT</title>
</head>
<body>
<?php
  # connect to SQL node:
  $link = new mysqli('198.51.100.20', 'root', 'root_password', 'world');
  # parameters for mysqli constructor are:
  #   host, user, password, database

  if( mysqli_connect_errno() )
    die("Connect failed: " . mysqli_connect_error());

  $query = "SELECT Name, Population
            FROM City
```

```
            ORDER BY Population DESC
            LIMIT 5";

  # if no errors...
  if( $result = $link->query($query) )
  {
?>
<table border="1" width="40%" cellpadding="4" cellspacing ="1">
  <tbody>
  <tr>
    <th width="10%">City</th>
    <th>Population</th>
  </tr>
<?
    # then display the results...
    while($row = $result->fetch_object())
      printf("<tr>\n  <td align=\"center\">%s</td><td>%d</td>\n</tr>\n",
              $row->Name, $row->Population);
?>
  </tbody
</table>
<?
  # ...and verify the number of rows that were retrieved
    printf("<p>Affected rows: %d</p>\n", $link->affected_rows);
  }
  else
    # otherwise, tell us what went wrong
    echo mysqli_error();

  # free the result set and the mysqli connection object
  $result->close();
  $link->close();
?>
</body>
</html>
```

We assume that the process running on the Web server can reach the IP address of the SQL node.

In a similar fashion, you can use the MySQL C API, Perl-DBI, Python-mysql, or MySQL Connectors to perform the tasks of data definition and manipulation just as you would normally with MySQL.

23.3.6 Safe Shutdown and Restart of NDB Cluster

To shut down the cluster, enter the following command in a shell on the machine hosting the management node:

```
$> ndb_mgm -e shutdown
```

The `-e` option here is used to pass a command to the `ndb_mgm` client from the shell. The command causes the `ndb_mgm`, `ndb_mgmd`, and any `ndbd` or `ndbmtd` processes to terminate gracefully. Any SQL nodes can be terminated using `mysqladmin shutdown` and other means. On Windows platforms, assuming that you have installed the SQL node as a Windows service, you can use `SC STOP service_name` or `NET STOP service_name`.

To restart the cluster on Unix platforms, run these commands:

- On the management host (`198.51.100.10` in our example setup):

  ```
  $> ndb_mgmd -f /var/lib/mysql-cluster/config.ini
  ```

- On each of the data node hosts (`198.51.100.30` and `198.51.100.40`):

  ```
  $> ndbd
  ```

- Use the `ndb_mgm` client to verify that both data nodes have started successfully.

- On the SQL host (`198.51.100.20`):

```
$> mysqld_safe &
```

On Windows platforms, assuming that you have installed all NDB Cluster processes as Windows services using the default service names (see Section 23.3.2.4, “Installing NDB Cluster Processes as Windows Services”), you can restart the cluster as follows:

- On the management host (`198.51.100.10` in our example setup), execute the following command:

```
C:\> SC START ndb_mgmd
```

- On each of the data node hosts (`198.51.100.30` and `198.51.100.40`), execute the following command:

```
C:\> SC START ndbd
```

- On the management node host, use the `ndb_mgm` client to verify that the management node and both data nodes have started successfully (see Section 23.3.2.3, “Initial Startup of NDB Cluster on Windows”).

- On the SQL node host (`198.51.100.20`), execute the following command:

```
C:\> SC START mysql
```

In a production setting, it is usually not desirable to shut down the cluster completely. In many cases, even when making configuration changes, or performing upgrades to the cluster hardware or software (or both), which require shutting down individual host machines, it is possible to do so without shutting down the cluster as a whole by performing a *rolling restart* of the cluster. For more information about doing this, see Section 23.6.5, “Performing a Rolling Restart of an NDB Cluster”.

23.3.7 Upgrading and Downgrading NDB Cluster

- Versions Supported for Upgrade to NDB 8.0
- Reverting an NDB Cluster 8.0 Upgrade
- Known Issues When Upgrading or Downgrading NDB Cluster

This section provides information about NDB Cluster software and compatibility between different NDB Cluster 8.0 releases with regard to performing upgrades and downgrades. You should already be familiar with installing and configuring NDB Cluster prior to attempting an upgrade or downgrade. See Section 23.4, “Configuration of NDB Cluster”.

Important

Online upgrades and downgrades between minor releases of the `NDB` storage engine are supported within NDB 8.0. In-place upgrades of the included MySQL Server (SQL node `mysqld`) are also supported; with multiple SQL nodes, it is possible to keep an SQL application online while individual `mysqld` processes are restarted. In-place downgrades of the included MySQL Server are *not* supported (see Section 2.11, “Downgrading MySQL”).

It may be possible in some cases to revert a recent upgrade from one NDB 8.0 minor release version to a later one, and to restore the needed states of any MySQL Server instances running as SQL nodes. Against the event that this becomes desirable or necessary, you are strongly advised to take a complete backup of each SQL node prior to upgrading NDB Cluster. For the same reason, you should also start the `mysqld` binaries from the new version with `--ndb-schema-dist-upgrade-allowed=0`, and not allow it to be set back to 1 until you are sure any likelihood of reverting to an older version is past. For more information, see Reverting an NDB Cluster 8.0 Upgrade.

For information about upgrades to NDB 8.0 from versions previous to 8.0, see Versions Supported for Upgrade to NDB 8.0.

For information about known issues and problems encountered when upgrading or downgrading NDB 8.0, see Known Issues When Upgrading or Downgrading NDB Cluster.

Versions Supported for Upgrade to NDB 8.0

The following versions of NDB Cluster are supported for upgrades to GA releases of NDB Cluster 8.0 (8.0.19 and later):

- NDB Cluster 7.6: NDB 7.6.4 and later
- NDB Cluster 7.5: NDB 7.5.4 and later
- NDB Cluster 7.4: NDB 7.4.6 and later

To upgrade from a release series previous to NDB 7.4, you must upgrade in stages, first to one of the versions just listed, and then from that version to the latest NDB 8.0 release. In such cases, upgrading to the latest NDB 7.6 release is recommended as the first step. For information about upgrades to NDB 7.6 from previous versions, see Upgrading and Downgrading NDB 7.6.

Reverting an NDB Cluster 8.0 Upgrade

Following a recent software upgrade of an NDB Cluster to an NDB 8.0 release, it is possible to revert the `NDB` software back to the earlier version, provided certain conditions are met before the upgrade, during the time the cluster is running the newer version, and after the NDB Cluster software is reverted to the earlier version. Specifics depend on local conditions; this section provides general information about what should be done at each of the points in the upgrade and rollback process just described.

In most cases, upgrading and downgrading the data nodes can be done without issue, as described elsewhere; see Section 23.6.5, “Performing a Rolling Restart of an NDB Cluster”. (Prior to performing an upgrade or downgrade, you should perform an `NDB` backup; see Section 23.6.8, “Online Backup of NDB Cluster”, for information about how to do this.) Downgrading SQL nodes online is not supported, due to the following issues:

- `mysqld` from a version 8.0 release cannot start if it detects a file system from a later version of MySQL.
- In many cases, `mysqld` cannot open tables that were created or modified by a later version of MySQL.
- In most if not all cases, `mysqld` cannot read binary log files that were created or modified in a later version of MySQL.

The procedure outlined next provides the basic steps necessary to upgrade a cluster from version *X* to version *Y* while allowing for a possible future rollback to *X*. (The procedure for reverting the upgraded cluster to version *X* follows later in this section.) For this purpose, version *X* is any NDB 8.0 GA release, or any previous NDB release supported for upgrade to NDB 8.0 (see Versions Supported for Upgrade to NDB 8.0), and version *Y* is an NDB 8.0 release which is later than *X*.

- *Prior to upgrade*: Take backups of NDB *X* SQL node states. This can be accomplished as one or more of the following:
  - A copy of the version *X* SQL node file system in a quiescent state using one or more system tools such as `cp`, `rsync`, `fwbackups`, Amanda, and so forth.

    A dump of any version *X* tables not stored in `NDB`. You can generate this dump using `mysqldump`.

    A backup created using MySQL Enterprise Backup; see Section 30.2, “MySQL Enterprise Backup Overview”, for more information.

Backing up the SQL nodes is recommended prior to any upgrade, whether or not you later intend to revert the cluster to the previous NDB version.

- *Upgrade to NDB Y*: All NDB *Y* `mysqld` binaries must be started with `--ndb-schema-dist-upgrade-allowed=0` to prevent any automatic schema upgrade. (Once any possibility of a downgrade is past, you can safely change the corresponding system variable `ndb_schema_dist_upgrade_allowed` back to 1, the default, in the `mysql` client.) When each NDB *Y* SQL node starts, it connects to the cluster and synchronizes its NDB table schemas. After this, you can restore MySQL table and state data from backup.

  To assure continuity of NDB replication, it is necessary to upgrade the cluster's SQL nodes in such a way that at least one `mysqld` is acting as the replication source at any given point in time during the upgrade. With two SQL nodes *A* and *B*, you can do so like this:

  1. While using SQL node *B* as the replication channel, upgrade SQL node *A* from NDB version *X* to version *Y*. This results in a gap in the binary log on *A* at epoch *E1*.
  2. After all replication appliers have consumed the binary log from SQL node *B* past epoch *E1*, switch the replication channel to use SQL node *A*.
  3. Upgrade SQL node *B* to NDB version *Y*. This results in a gap in the binary log on *B* at epoch *E2*.
  4. After all replication appliers have consumed the binary log from SQL node *A* past epoch *E2*, you can once again switch the replication channel to use either SQL node as desired.

  Do not use `ALTER TABLE` on any existing NDB tables; do not create any new NDB tables which cannot be safely dropped prior to downgrading.

The following procedure shows the basic steps needed to roll back (revert) an NDB Cluster from version *X* to version *Y* after an upgrade performed as just described. Here, version *X* is any NDB 8.0 GA release, or any previous NDB release supported for upgrade to NDB 8.0 (see Versions Supported for Upgrade to NDB 8.0); version *Y* is an NDB 8.0 release which is later than *X*.

- *Prior to rollback*: Gather any `mysqld` state information from the NDB *Y* SQL nodes that should be retained. In most cases, you can do this using `mysqldump`.

  After backing up the state data, drop all NDB tables which have been created or altered since the upgrade took place.

  Backing up the SQL nodes is always recommended prior to any NDB Cluster software version change.

  You must provide a file system compatible with MySQL *X* for each `mysqld` (SQL node). You can use either of the following two methods:

  - Create a new, compatible file system state by reinitializing the on-disk state of the version *X* SQL node. You can do this by removing the SQL node file system, then running `mysqld --initialize`.
  - Restore a file system that is compatible from a backup taken prior to the upgrade (see Section 7.4, “Using mysqldump for Backups”).

- *Following NDB downgrade*: After downgrading the data nodes to NDB *X*, start the version *X* SQL nodes (instances of `mysqld`). Restore or repair any other local state information needed on each SQL node. The MySQLD state can be aligned as necessary with some combination (0 or more) of the following actions:

  - Initialization commands such as `mysqld --initialize`.
  - Restore any desired or required state information captured from the version *X* SQL node.

- Restore any desired or required state information captured from the version *Y* SQL node.
- Perform cleanup such as deleting stale logs such as binary logs, or relay logs, and removing any time-dependent state which is no longer valid.

As when upgrading, it is necessary when downgrading to maintain continuity of NDB replication to downgrade the cluster's SQL nodes in such a way that at least one `mysqld` is acting as the replication source at any given point in time during the downgrade process. This can be done in a manner very similar to that described previously for upgrading the SQL nodes. With two SQL nodes *A* and *B*, you can maintain binary logging without any gaps during the downgrade like this:

1. With SQL node *B* acting as the replication channel, downgrade SQL node *A* from NDB version *Y* to version *X*. This results in a gap in the binary log on *A* at epoch *F1*.
2. After all replication appliers have consumed the binary log from SQL node *B* past epoch *F1*, switch the replication channel to use SQL node *A*.
3. Downgrade SQL node *B* to NDB version *X*. This results in a gap in the binary log on *B* at epoch *F2*.
4. After all replication appliers have consumed the binary log from SQL node *A* past epoch *F2*, redundancy of binary logging is restored, and you can again use either SQL node as the replication channel as desired.

See also Section 23.7.7, “Using Two Replication Channels for NDB Cluster Replication”.

Known Issues When Upgrading or Downgrading NDB Cluster

In this section, provide information about issues known to occur when upgrading or downgrading to, from, or between NDB 8.0 releases.

We recommend that you not attempt any schema changes during any NDB Cluster software upgrade or downgrade. Some of the reasons for this are listed here:

- DDL statements on `NDB` tables are not possible during some phases of data node startup.
- DDL statements on `NDB` tables may be rejected if any data nodes are stopped during execution; stopping each data node binary (so it can be replaced with a binary from the target version) is required as part of the upgrade or downgrade process.
- DDL statements on `NDB` tables are not allowed while there are data nodes in the same cluster running different release versions of the NDB Cluster software.

For additional information regarding the rolling restart procedure used to perform an online upgrade or downgrade of the data nodes, see Section 23.6.5, “Performing a Rolling Restart of an NDB Cluster”.

You should be aware of the issues in the following list when you perform an online upgrade between minor versions of NDB 8.0. These issues also apply when upgrading from a previous major version of NDB Cluster to any of the NDB 8.0 releases stated.

- NDB 8.0.22 adds support for IPv6 addressing for management nodes and data nodes in the `config.ini` file. To begin using IPv6 addresses as part of an upgrade, perform the following steps:

  1. Perform an upgrade of the cluster to version 8.0.22 or a later version of the NDB Cluster software in the usual manner.
  2. Change the addresses used in the `config.ini` file to IPv6 addresses.
  3. Perform a system restart of the cluster.

A known issue on Linux platforms when running NDB 8.0.22 and later is that the operating system kernel must provide IPv6 support, even if no IPv6 addresses are in use. If you wish to disable support for IPv6 on the system (because you do not plan to use any IPv6 addresses for NDB Cluster nodes), do so after booting the system, like this:

```
$> sysctl -w net.ipv6.conf.all.disable_ipv6=1
$> sysctl -w net.ipv6.conf.default.disable_ipv6=1
```

Alternatively, you can add the corresponding lines to `/etc/sysctl.conf`.

- Due to changes in the internal `mysql.ndb_schema` table, if you upgrade to an NDB 8.0 release prior to 8.0.24, then you are advised to use `--ndb-schema-dist-upgrade-allowed = 0` to avoid unexpected outages (Bug #30876990, Bug #31016905).

  In addition, if there is any possibility that you may revert to a previous version of NDB Cluster following an upgrade to a newer version, you must start all `mysqld` processes from the newer version with `--ndb-schema-dist-upgrade-allowed = 0` to prevent changes incompatible with the older version from being made to the `ndb_schema` table. See Reverting an NDB Cluster 8.0 Upgrade, for information about how to do this.

- The `EncryptedFileSystem` configuration parameter, introduced in NDB 8.0.29, could in some cases cause undo log files to be encrypted, even when set explicitly to `0`, which could lead to issues when using Disk Data tables and attempting to upgrade or downgrade to NDB 8.0.29. In such cases, you can work around the problem by performing initial restarts of the data nodes as part of the rolling restart process.

- If you are using multithreaded data nodes (`ndbmtd`) and the `ThreadConfig` configuration parameter, you may need to make changes in the value set for this in the `config.ini` file when upgrading from a previous release to NDB 8.0.30 or later. When upgrading from NDB 8.0.23 or earlier, any usage of `main`, `rep`, `recv`, or `ldm` threads that was implicit in the earlier version must be explicitly set. When upgrading from NDB 8.0.23 or later to NDB 8.0.30 or later, any usage of `recv` threads must be set explicitly in the `ThreadConfig` string. In addition, to avoid using `main`, `rep`, or `ldm` threads in NDB 8.0.30 or later, you must set the thread count for the given type to `0` explicitly.

  An example follows.

  NDB 8.0.22 and earlier:

  - `config.ini` file contains `ThreadConfig=ldm`.
  - This is interpreted by these versions of `NDB` as `ThreadConfig=main,ldm,recv,rep`.
  - Required in `config.ini` to match effect in NDB 8.0.30 or later: `ThreadConfig=main,ldm,recv,rep`.

  NDB 8.0.23—8.0.29:

  - `config.ini` file contains `ThreadConfig=ldm`.
  - This is interpreted by these versions of `NDB` as `ThreadConfig=ldm,recv`.
  - Required in `config.ini` to match effect in NDB 8.0.30 or later: `ThreadConfig=main={count=0},ldm,recv,rep={count=0}`.

  For more information, see the description of the `ThreadConfig` configuration parameter.

Upgrades from previous major versions of NDB Cluster (7.4, 7.5, 7.6) to NDB 8.0 are supported; see Versions Supported for Upgrade to NDB 8.0, for specific versions. Such upgrades are subject to the issues listed here:

- In NDB 8.0, the default values changed for `log_bin` (from 0 to 1) and `ndb_log_bin` (from 1 to 0). This means that you must now explicitly set `ndb_log_bin` to 1 to enable binary logging.

- Distributed privileges shared between MySQL servers as implemented in prior release series (see Distributed Privileges Using Shared Grant Tables) are not supported in NDB Cluster 8.0. When started, the `mysqld` supplied with NDB 8.0 and later checks for the existence of any grant tables which use the `NDB` storage engine; if it finds any, it creates local copies ("shadow tables") of these using `InnoDB`. This is true for each MySQL server connected to NDB Cluster. After this has been performed on all MySQL servers acting as NDB Cluster SQL nodes, the `NDB` grant tables may be safely removed using the `ndb_drop_table` utility supplied with the NDB Cluster distribution, like this:

  ```
  ndb_drop_table -d mysql user db columns_priv tables_priv proxies_priv procs_priv
  ```

  It is safe to retain the `NDB` grant tables, but they are not used for access control and are effectively ignored.

  For more information about the MySQL privileges system used in NDB 8.0, see Section 23.6.13, "Privilege Synchronization and NDB_STORED_USER", as well as Section 6.2.3, "Grant Tables".

- It is necessary to restart all data nodes with `--initial` when upgrading any release prior to NDB 7.6 to any NDB 8.0 release. This is due to the addition of support for increased numbers of nodes in NDB 8.0.

Issues encountered when trying to downgrade from NDB 8.0 to a previous major version can be found in the following list:

- Tables created in NDB 8.0 are not backwards compatible with NDB 7.6 and earlier releases due to a change in usage of the extra metadata property implemented by `NDB` tables to provide full support for the MySQL data dictionary. This means that it is necessary to take extra steps to preserve any desired state information from the cluster's SQL nodes prior to the downgrade, and then to restore it afterwards.

  More specifically, online downgrades of the `NDBCLUSTER` storage engine—that is, of the data nodes—are supported, but SQL nodes cannot be downgraded online. This is because a MySQL Server (`mysqld`) of a given MySQL 8.0 or earlier version cannot use system files from a (later) 8.0 version, and cannot open tables that were created in the later version. It may be possible to roll back a cluster that has recently been upgraded from a previous NDB release; see Reverting an NDB Cluster 8.0 Upgrade, for information regarding when and how this can be done.

  For additional information relating to these issues, see Changes in NDB table extra metadata; see also Chapter 14, *MySQL Data Dictionary*.

- In NDB 8.0, the binary configuration file format has been enhanced to provide support for greater numbers of nodes than in previous versions. The new format is not accessible to nodes running older versions of `NDB`, although newer management servers can detect older nodes and communicate with them using the appropriate format.

  While upgrades to NDB 8.0 should not be problematic in this regard, older management servers cannot read the newer binary configuration file format, so that some manual intervention is required when downgrading from NDB 8.0 to a previous major version. When performing such a downgrade, it is necessary to remove any cached binary configuration files prior to starting the management using the older `NDB` software version, and to have the plaintext configuration file available for the management server to read. Alternatively, you can start the older management server using the `--initial` option (again, it is necessary to have the `config.ini` available). If the cluster uses multiple management servers, one of these two things must be done for each management server binary.

  Also in connection with support for increased numbers of nodes, and due to incompatible changes implemented in NDB 8.0 in the data node LCP `Sysfile`, it is necessary, when performing an online

downgrade from NDB 8.0 to a prior major version, to restart all data nodes using the `--initial` option.

- Online downgrades of clusters running more than 48 data nodes, or with data nodes using node IDs greater than 48, to earlier NDB Cluster releases from NDB 8.0 are not supported. It is necessary in such cases to reduce the number of data nodes, to change the configurations for all data nodes such that they use node IDs less than or equal to 48, or both, as required not to exceed the old maximums.

- If you are downgrading from NDB 8.0 to NDB 7.5 or NDB 7.4, you must set an explicit value for `IndexMemory` in the cluster configuration file if none is already present. This is because NDB 8.0 does not use this parameter (which was removed in NDB 7.6) and sets it to 0 by default, whereas it is required in NDB 7.5 and NDB 7.4, in both of which the cluster refuses to start with `Invalid configuration received from Management Server...` if `IndexMemory` is not set to a nonzero value.

  Setting `IndexMemory` is *not* required for downgrades from NDB 8.0 to NDB 7.6.

23.3.8 The NDB Cluster Auto-Installer (NO LONGER SUPPORTED)

Note

This feature has been removed from NDB Cluster, and is no longer supported. See Section 23.2.4, “What is New in MySQL NDB Cluster”, for more information.

The web-based graphical configuration installer (Auto-Installer) was removed in NDB 8.0.23, and is no longer included as part of the NDB Cluster distribution.

23.4 Configuration of NDB Cluster

A MySQL server that is part of an NDB Cluster differs in one chief respect from a normal (nonclustered) MySQL server, in that it employs the `NDB` storage engine. This engine is also referred to sometimes as `NDBCLUSTER`, although `NDB` is preferred.

To avoid unnecessary allocation of resources, the server is configured by default with the `NDB` storage engine disabled. To enable `NDB`, you must modify the server's `my.cnf` configuration file, or start the server with the `--ndbcluster` option.

This MySQL server is a part of the cluster, so it also must know how to access a management node to obtain the cluster configuration data. The default behavior is to look for the management node on `localhost`. However, should you need to specify that its location is elsewhere, this can be done in `my.cnf`, or with the `mysql` client. Before the `NDB` storage engine can be used, at least one management node must be operational, as well as any desired data nodes.

For more information about `--ndbcluster` and other `mysqld` options specific to NDB Cluster, see MySQL Server Options for NDB Cluster.

For general information about installing NDB Cluster, see Section 23.3, “NDB Cluster Installation”.

23.4.1 Quick Test Setup of NDB Cluster

To familiarize you with the basics, we describe the simplest possible configuration for a functional NDB Cluster. After this, you should be able to design your desired setup from the information provided in the other relevant sections of this chapter.

First, you need to create a configuration directory such as `/var/lib/mysql-cluster`, by executing the following command as the system `root` user:

```
$> mkdir /var/lib/mysql-cluster
```

In this directory, create a file named `config.ini` that contains the following information. Substitute appropriate values for `HostName` and `DataDir` as necessary for your system.

```
# file "config.ini" - showing minimal setup consisting of 1 data node,
# 1 management server, and 3 MySQL servers.
# The empty default sections are not required, and are shown only for
# the sake of completeness.
# Data nodes must provide a hostname but MySQL Servers are not required
# to do so.
# If you don't know the hostname for your machine, use localhost.
# The DataDir parameter also has a default value, but it is recommended to
# set it explicitly.
# Note: [db], [api], and [mgm] are aliases for [ndbd], [mysqld], and [ndb_mgmd],
# respectively. [db] is deprecated and should not be used in new installations.

[ndbd default]
NoOfReplicas= 1

[mysqld  default]
[ndb_mgmd default]
[tcp default]

[ndb_mgmd]
HostName= myhost.example.com

[ndbd]
HostName= myhost.example.com
DataDir= /var/lib/mysql-cluster

[mysqld]
[mysqld]
[mysqld]
```

You can now start the `ndb_mgmd` management server. By default, it attempts to read the `config.ini` file in its current working directory, so change location into the directory where the file is located and then invoke `ndb_mgmd`:

```
$> cd /var/lib/mysql-cluster
$> ndb_mgmd
```

Then start a single data node by running `ndbd`:

```
$> ndbd
```

By default, `ndbd` looks for the management server at `localhost` on port 1186.

Note

If you have installed MySQL from a binary tarball, you must to specify the path of the `ndb_mgmd` and `ndbd` servers explicitly. (Normally, these can be found in `/usr/local/mysql/bin`.)

Finally, change location to the MySQL data directory (usually `/var/lib/mysql` or `/usr/local/mysql/data`), and make sure that the `my.cnf` file contains the option necessary to enable the NDB storage engine:

```
[mysqld]
ndbcluster
```

You can now start the MySQL server as usual:

```
$> mysqld_safe --user=mysql &
```

Wait a moment to make sure the MySQL server is running properly. If you see the notice `mysql ended`, check the server's `.err` file to find out what went wrong.

If all has gone well so far, you now can start using the cluster. Connect to the server and verify that the `NDBCLUSTER` storage engine is enabled:

```
$> mysql
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 1 to server version: 8.0.32

Type 'help;' or '\h' for help. Type '\c' to clear the buffer.

mysql> SHOW ENGINES\G
...
*************************** 12. row ***************************
Engine: NDBCLUSTER
Support: YES
Comment: Clustered, fault-tolerant, memory-based tables
*************************** 13. row ***************************
Engine: NDB
Support: YES
Comment: Alias for NDBCLUSTER
...
```

The row numbers shown in the preceding example output may be different from those shown on your system, depending upon how your server is configured.

Try to create an `NDBCLUSTER` table:

```
$> mysql
mysql> USE test;
Database changed

mysql> CREATE TABLE ctest (i INT) ENGINE=NDBCLUSTER;
Query OK, 0 rows affected (0.09 sec)

mysql> SHOW CREATE TABLE ctest \G
*************************** 1. row ***************************
       Table: ctest
Create Table: CREATE TABLE `ctest` (
  `i` int(11) default NULL
) ENGINE=ndbcluster DEFAULT CHARSET=latin1
1 row in set (0.00 sec)
```

To check that your nodes were set up properly, start the management client:

```
$> ndb_mgm
```

Use the `SHOW` command from within the management client to obtain a report on the cluster's status:

```
ndb_mgm> SHOW
Cluster Configuration
---------------------
[ndbd(NDB)]     1 node(s)
id=2    @127.0.0.1  (Version: 8.0.34-ndb-8.0.34, Nodegroup: 0, *)

[ndb_mgmd(MGM)] 1 node(s)
id=1    @127.0.0.1  (Version: 8.0.34-ndb-8.0.34)

[mysqld(API)]   3 node(s)
id=3    @127.0.0.1  (Version: 8.0.34-ndb-8.0.34)
id=4 (not connected, accepting connect from any host)
id=5 (not connected, accepting connect from any host)
```

At this point, you have successfully set up a working NDB Cluster . You can now store data in the cluster by using any table created with `ENGINE=NDBCLUSTER` or its alias `ENGINE=NDB`.

23.4.2 Overview of NDB Cluster Configuration Parameters, Options, and Variables

The next several sections provide summary tables of NDB Cluster node configuration parameters used in the `config.ini` file to govern various aspects of node behavior, as well as of options and variables read by `mysqld` from a `my.cnf` file or from the command line when run as an NDB Cluster process. Each of the node parameter tables lists the parameters for a given type (`ndbd`, `ndb_mgmd`, `mysqld`,

`computer`, `tcp`, or `shm`). All tables include the data type for the parameter, option, or variable, as well as its default, minimum, and maximum values as applicable.

Considerations when restarting nodes. For node parameters, these tables also indicate what type of restart is required (node restart or system restart)—and whether the restart must be done with `--initial`—to change the value of a given configuration parameter. When performing a node restart or an initial node restart, all of the cluster's data nodes must be restarted in turn (also referred to as a *rolling restart*). It is possible to update cluster configuration parameters marked as `node` online—that is, without shutting down the cluster—in this fashion. An initial node restart requires restarting each `ndbd` process with the `--initial` option.

A system restart requires a complete shutdown and restart of the entire cluster. An initial system restart requires taking a backup of the cluster, wiping the cluster file system after shutdown, and then restoring from the backup following the restart.

In any cluster restart, all of the cluster's management servers must be restarted for them to read the updated configuration parameter values.

Important

Values for numeric cluster parameters can generally be increased without any problems, although it is advisable to do so progressively, making such adjustments in relatively small increments. Many of these can be increased online, using a rolling restart.

However, decreasing the values of such parameters—whether this is done using a node restart, node initial restart, or even a complete system restart of the cluster—is not to be undertaken lightly; it is recommended that you do so only after careful planning and testing. This is especially true with regard to those parameters that relate to memory usage and disk space, such as `MaxNoOfTables`, `MaxNoOfOrderedIndexes`, and `MaxNoOfUniqueHashIndexes`. In addition, it is the generally the case that configuration parameters relating to memory and disk usage can be raised using a simple node restart, but they require an initial node restart to be lowered.

Because some of these parameters can be used for configuring more than one type of cluster node, they may appear in more than one of the tables.

Note

`4294967039` often appears as a maximum value in these tables. This value is defined in the `NDBCLUSTER` sources as `MAX_INT_RNIL` and is equal to `0xFFFFFEFF`, or $2^{32} - 2^8 - 1$.

23.4.2.1 NDB Cluster Data Node Configuration Parameters

The listings in this section provide information about parameters used in the `[ndbd]` or `[ndbd default]` sections of a `config.ini` file for configuring NDB Cluster data nodes. For detailed descriptions and other additional information about each of these parameters, see Section 23.4.3.6, “Defining NDB Cluster Data Nodes”.

These parameters also apply to `ndbmtd`, the multithreaded version of `ndbd`. A separate listing of parameters specific to `ndbmtd` follows.

- `Arbitration`: How arbitration should be performed to avoid split-brain issues in event of node failure.
- `ArbitrationTimeout`: Maximum time (milliseconds) database partition waits for arbitration signal.
- `BackupDataBufferSize`: Default size of databuffer for backup (in bytes).

- `BackupDataDir`: Path to where to store backups. Note that string '/BACKUP' is always appended to this setting, so that *effective* default is FileSystemPath/BACKUP.
- `BackupDiskWriteSpeedPct`: Sets percentage of data node's allocated maximum write speed (MaxDiskWriteSpeed) to reserve for LCPs when starting backup.
- `BackupLogBufferSize`: Default size of log buffer for backup (in bytes).
- `BackupMaxWriteSize`: Maximum size of file system writes made by backup (in bytes).
- `BackupMemory`: Total memory allocated for backups per node (in bytes).
- `BackupReportFrequency`: Frequency of backup status reports during backup in seconds.
- `BackupWriteSize`: Default size of file system writes made by backup (in bytes).
- `BatchSizePerLocalScan`: Used to calculate number of lock records for scan with hold lock.
- `BuildIndexThreads`: Number of threads to use for building ordered indexes during system or node restart. Also applies when running ndb_restore --rebuild-indexes. Setting this parameter to 0 disables multithreaded building of ordered indexes.
- `CompressedBackup`: Use zlib to compress backups as they are written.
- `CompressedLCP`: Write compressed LCPs using zlib.
- `ConnectCheckIntervalDelay`: Time between data node connectivity check stages. Data node is considered suspect after 1 interval and dead after 2 intervals with no response.
- `CrashOnCorruptedTuple`: When enabled, forces node to shut down whenever it detects corrupted tuple.
- `DataDir`: Data directory for this node.
- `DataMemory`: Number of bytes on each data node allocated for storing data; subject to available system RAM and size of IndexMemory.
- `DefaultHashMapSize`: Set size (in buckets) to use for table hash maps. Three values are supported: 0, 240, and 3840.
- `DictTrace`: Enable DBDICT debugging; for NDB development.
- `DiskDataUsingSameDisk`: Set to false if Disk Data tablespaces are located on separate physical disks.
- `DiskIOThreadPool`: Number of unbound threads for file access, applies to disk data only.
- `Diskless`: Run without using disk.
- `DiskPageBufferEntries`: Memory to allocate in DiskPageBufferMemory; very large disk transactions may require increasing this value.
- `DiskPageBufferMemory`: Number of bytes on each data node allocated for disk page buffer cache.
- `DiskSyncSize`: Amount of data written to file before synch is forced.
- `EnablePartialLcp`: Enable partial LCP (true); if this is disabled (false), all LCPs write full checkpoints.
- `EnableRedoControl`: Enable adaptive checkpointing speed for controlling redo log usage.
- `EncryptedFileSystem`: Encrypt local checkpoint and tablespace files. EXPERIMENTAL; NOT SUPPORTED IN PRODUCTION.
- `EventLogBufferSize`: Size of circular buffer for NDB log events within data nodes.

- `ExecuteOnComputer`: String referencing earlier defined COMPUTER.
- `ExtraSendBufferMemory`: Memory to use for send buffers in addition to any allocated by TotalSendBufferMemory or SendBufferMemory. Default (0) allows up to 16MB.
- `FileSystemPath`: Path to directory where data node stores its data (directory must exist).
- `FileSystemPathDataFiles`: Path to directory where data node stores its Disk Data files. Default value is FilesystemPathDD, if set; otherwise, FilesystemPath is used if it is set; otherwise, value of DataDir is used.
- `FileSystemPathDD`: Path to directory where data node stores its Disk Data and undo files. Default value is FileSystemPath, if set; otherwise, value of DataDir is used.
- `FileSystemPathUndoFiles`: Path to directory where data node stores its undo files for Disk Data. Default value is FilesystemPathDD, if set; otherwise, FilesystemPath is used if it is set; otherwise, value of DataDir is used.
- `FragmentLogFileSize`: Size of each redo log file.
- `HeartbeatIntervalDbApi`: Time between API node-data node heartbeats. (API connection closed after 3 missed heartbeats).
- `HeartbeatIntervalDbDb`: Time between data node-to-data node heartbeats; data node considered dead after 3 missed heartbeats.
- `HeartbeatOrder`: Sets order in which data nodes check each others' heartbeats for determining whether given node is still active and connected to cluster. Must be zero for all data nodes or distinct nonzero values for all data nodes; see documentation for further guidance.
- `HostName`: Host name or IP address for this data node.
- `IndexMemory`: Number of bytes on each data node allocated for storing indexes; subject to available system RAM and size of DataMemory.
- `IndexStatAutoCreate`: Enable/disable automatic statistics collection when indexes are created.
- `IndexStatAutoUpdate`: Monitor indexes for changes and trigger automatic statistics updates.
- `IndexStatSaveScale`: Scaling factor used in determining size of stored index statistics.
- `IndexStatSaveSize`: Maximum size in bytes for saved statistics per index.
- `IndexStatTriggerPct`: Threshold percent change in DML operations for index statistics updates. Value is scaled down by IndexStatTriggerScale.
- `IndexStatTriggerScale`: Scale down IndexStatTriggerPct by this amount, multiplied by base 2 logarithm of index size, for large index. Set to 0 to disable scaling.
- `IndexStatUpdateDelay`: Minimum delay between automatic index statistics updates for given index. 0 means no delay.
- `InitFragmentLogFiles`: Initialize fragment log files, using sparse or full format.
- `InitialLogFileGroup`: Describes log file group that is created during initial start. See documentation for format.
- `InitialNoOfOpenFiles`: Initial number of files open per data node. (One thread is created per file).
- `InitialTablespace`: Describes tablespace that is created during initial start. See documentation for format.
- `InsertRecoveryWork`: Percentage of RecoveryWork used for inserted rows; has no effect unless partial local checkpoints are in use.

- LateAlloc: Allocate memory after connection to management server has been established.
- LcpScanProgressTimeout: Maximum time that local checkpoint fragment scan can be stalled before node is shut down to ensure systemwide LCP progress. Use 0 to disable.
- LockExecuteThreadToCPU: Comma-delimited list of CPU IDs.
- LockMaintThreadsToCPU: CPU ID indicating which CPU runs maintenance threads.
- LockPagesInMainMemory: 0=disable locking, 1=lock after memory allocation, 2=lock before memory allocation.
- LogLevelCheckpoint: Log level of local and global checkpoint information printed to stdout.
- LogLevelCongestion: Level of congestion information printed to stdout.
- LogLevelConnection: Level of node connect/disconnect information printed to stdout.
- LogLevelError: Transporter, heartbeat errors printed to stdout.
- LogLevelInfo: Heartbeat and log information printed to stdout.
- LogLevelNodeRestart: Level of node restart and node failure information printed to stdout.
- LogLevelShutdown: Level of node shutdown information printed to stdout.
- LogLevelStartup: Level of node startup information printed to stdout.
- LogLevelStatistic: Level of transaction, operation, and transporter information printed to stdout.
- LongMessageBuffer: Number of bytes allocated on each data node for internal long messages.
- MaxAllocate: No longer used; has no effect.
- MaxBufferedEpochs: Allowed numbered of epochs that subscribing node can lag behind (unprocessed epochs). Exceeding causes lagging subscribers to be disconnected.
- MaxBufferedEpochBytes: Total number of bytes allocated for buffering epochs.
- MaxDiskDataLatency: Maximum allowed mean latency of disk access (ms) before starting to abort transactions.
- MaxDiskWriteSpeed: Maximum number of bytes per second that can be written by LCP and backup when no restarts are ongoing.
- MaxDiskWriteSpeedOtherNodeRestart: Maximum number of bytes per second that can be written by LCP and backup when another node is restarting.
- MaxDiskWriteSpeedOwnRestart: Maximum number of bytes per second that can be written by LCP and backup when this node is restarting.
- MaxFKBuildBatchSize: Maximum scan batch size to use for building foreign keys. Increasing this value may speed up builds of foreign keys but impacts ongoing traffic as well.
- MaxDMLOperationsPerTransaction: Limit size of transaction; aborts transaction if it requires more than this many DML operations.
- MaxLCPStartDelay: Time in seconds that LCP polls for checkpoint mutex (to allow other data nodes to complete metadata synchronization), before putting itself in lock queue for parallel recovery of table data.
- MaxNoOfAttributes: Suggests total number of attributes stored in database (sum over all tables).
- MaxNoOfConcurrentIndexOperations: Total number of index operations that can execute simultaneously on one data node.

- `MaxNoOfConcurrentOperations`: Maximum number of operation records in transaction coordinator.
- `MaxNoOfConcurrentScans`: Maximum number of scans executing concurrently on data node.
- `MaxNoOfConcurrentSubOperations`: Maximum number of concurrent subscriber operations.
- `MaxNoOfConcurrentTransactions`: Maximum number of transactions executing concurrently on this data node, total number of transactions that can be executed concurrently is this value times number of data nodes in cluster.
- `MaxNoOfFiredTriggers`: Total number of triggers that can fire simultaneously on one data node.
- `MaxNoOfLocalOperations`: Maximum number of operation records defined on this data node.
- `MaxNoOfLocalScans`: Maximum number of fragment scans in parallel on this data node.
- `MaxNoOfOpenFiles`: Maximum number of files open per data node.(One thread is created per file).
- `MaxNoOfOrderedIndexes`: Total number of ordered indexes that can be defined in system.
- `MaxNoOfSavedMessages`: Maximum number of error messages to write in error log and maximum number of trace files to retain.
- `MaxNoOfSubscribers`: Maximum number of subscribers.
- `MaxNoOfSubscriptions`: Maximum number of subscriptions (default 0 = MaxNoOfTables).
- `MaxNoOfTables`: Suggests total number of NDB tables stored in database.
- `MaxNoOfTriggers`: Total number of triggers that can be defined in system.
- `MaxNoOfUniqueHashIndexes`: Total number of unique hash indexes that can be defined in system.
- `MaxParallelCopyInstances`: Number of parallel copies during node restarts. Default is 0, which uses number of LDMs on both nodes, to maximum of 16.
- `MaxParallelScansPerFragment`: Maximum number of parallel scans per fragment. Once this limit is reached, scans are serialized.
- `MaxReorgBuildBatchSize`: Maximum scan batch size to use for reorganization of table partitions. Increasing this value may speed up table partition reorganization but impacts ongoing traffic as well.
- `MaxStartFailRetries`: Maximum retries when data node fails on startup, requires StopOnError = 0. Setting to 0 causes start attempts to continue indefinitely.
- `MaxUIBuildBatchSize`: Maximum scan batch size to use for building unique keys. Increasing this value may speed up builds of unique keys but impacts ongoing traffic as well.
- `MemReportFrequency`: Frequency of memory reports in seconds; 0 = report only when exceeding percentage limits.
- `MinDiskWriteSpeed`: Minimum number of bytes per second that can be written by LCP and backup.
- `MinFreePct`: Percentage of memory resources to keep in reserve for restarts.
- `NodeGroup`: Node group to which data node belongs; used only during initial start of cluster.
- `NodeGroupTransporters`: Number of transporters to use between nodes in same node group.
- `NodeId`: Number uniquely identifying data node among all nodes in cluster.
- `NoOfFragmentLogFiles`: Number of 16 MB redo log files in each of 4 file sets belonging to data node.

- `NoOfReplicas`: Number of copies of all data in database.
- `Numa`: (Linux only; requires libnuma) Controls NUMA support. Setting to 0 permits system to determine use of interleaving by data node process; 1 means that it is determined by data node.
- `ODirect`: Use O_DIRECT file reads and writes when possible.
- `ODirectSyncFlag`: O_DIRECT writes are treated as synchronized writes; ignored when ODirect is not enabled, InitFragmentLogFiles is set to SPARSE, or both.
- `RealtimeScheduler`: When true, data node threads are scheduled as real-time threads. Default is false.
- `RecoveryWork`: Percentage of storage overhead for LCP files: greater value means less work in normal operations, more work during recovery.
- `RedoBuffer`: Number of bytes on each data node allocated for writing redo logs.
- `RedoOverCommitCounter`: When RedoOverCommitLimit has been exceeded this many times, transactions are aborted, and operations are handled as specified by DefaultOperationRedoProblemAction.
- `RedoOverCommitLimit`: Each time that flushing current redo buffer takes longer than this many seconds, number of times that this has happened is compared to RedoOverCommitCounter.
- `ReservedConcurrentIndexOperations`: Number of simultaneous index operations having dedicated resources on one data node.
- `ReservedConcurrentOperations`: Number of simultaneous operations having dedicated resources in transaction coordinators on one data node.
- `ReservedConcurrentScans`: Number of simultaneous scans having dedicated resources on one data node.
- `ReservedConcurrentTransactions`: Number of simultaneous transactions having dedicated resources on one data node.
- `ReservedFiredTriggers`: Number of triggers having dedicated resources on one data node.
- `ReservedLocalScans`: Number of simultaneous fragment scans having dedicated resources on one data node.
- `ReservedTransactionBufferMemory`: Dynamic buffer space (in bytes) for key and attribute data allocated to each data node.
- `RestartOnErrorInsert`: Control type of restart caused by inserting error (when StopOnError is enabled).
- `SchedulerExecutionTimer`: Number of microseconds to execute in scheduler before sending.
- `SchedulerResponsiveness`: Set NDB scheduler response optimization 0-10; higher values provide better response time but lower throughput.
- `SchedulerSpinTimer`: Number of microseconds to execute in scheduler before sleeping.
- `ServerPort`: Port used to set up transporter for incoming connections from API nodes.
- `SharedGlobalMemory`: Total number of bytes on each data node allocated for any use.
- `SpinMethod`: Determines spin method used by data node; see documentation for details.
- `StartFailRetryDelay`: Delay in seconds after start failure prior to retry; requires StopOnError = 0.
- `StartFailureTimeout`: Milliseconds to wait before terminating. (0=Wait forever).

- `StartNoNodeGroupTimeout`: Time to wait for nodes without nodegroup before trying to start (0=forever).
- `StartPartialTimeout`: Milliseconds to wait before trying to start without all nodes. (0=Wait forever).
- `StartPartitionedTimeout`: Milliseconds to wait before trying to start partitioned. (0=Wait forever).
- `StartupStatusReportFrequency`: Frequency of status reports during startup.
- `StopOnError`: When set to 0, data node automatically restarts and recovers following node failures.
- `StringMemory`: Default size of string memory (0 to 100 = % of maximum, 101+ = actual bytes).
- `TcpBind_INADDR_ANY`: Bind IP_ADDR_ANY so that connections can be made from anywhere (for autogenerated connections).
- `TimeBetweenEpochs`: Time between epochs (synchronization used for replication).
- `TimeBetweenEpochsTimeout`: Timeout for time between epochs. Exceeding causes node shutdown.
- `TimeBetweenGlobalCheckpoints`: Time between group commits of transactions to disk.
- `TimeBetweenGlobalCheckpointsTimeout`: Minimum timeout for group commit of transactions to disk.
- `TimeBetweenInactiveTransactionAbortCheck`: Time between checks for inactive transactions.
- `TimeBetweenLocalCheckpoints`: Time between taking snapshots of database (expressed in base-2 logarithm of bytes).
- `TimeBetweenWatchDogCheck`: Time between execution checks inside data node.
- `TimeBetweenWatchDogCheckInitial`: Time between execution checks inside data node (early start phases when memory is allocated).
- `TotalSendBufferMemory`: Total memory to use for all transporter send buffers..
- `TransactionBufferMemory`: Dynamic buffer space (in bytes) for key and attribute data allocated for each data node.
- `TransactionDeadlockDetectionTimeout`: Time transaction can spend executing within data node. This is time that transaction coordinator waits for each data node participating in transaction to execute request. If data node takes more than this amount of time, transaction is aborted.
- `TransactionInactiveTimeout`: Milliseconds that application waits before executing another part of transaction. This is time transaction coordinator waits for application to execute or send another part (query, statement) of transaction. If application takes too much time, then transaction is aborted. Timeout = 0 means that application never times out.
- `TransactionMemory`: Memory allocated for transactions on each data node.
- `TwoPassInitialNodeRestartCopy`: Copy data in 2 passes during initial node restart, which enables multithreaded building of ordered indexes for such restarts.
- `UndoDataBuffer`: Unused; has no effect.
- `UndoIndexBuffer`: Unused; has no effect.
- `UseShm`: Use shared memory connections between this data node and API node also running on this host.

The following parameters are specific to `ndbmtd`:

- `AutomaticThreadConfig`: Use automatic thread configuration; overrides any settings for ThreadConfig and MaxNoOfExecutionThreads, and disables ClassicFragmentation.
- `ClassicFragmentation`: When true, use traditional table fragmentation; set false to enable flexible distribution of fragments among LDMs. Disabled by AutomaticThreadConfig.
- `EnableMultithreadedBackup`: Enable multi-threaded backup.
- `MaxNoOfExecutionThreads`: For ndbmtd only, specify maximum number of execution threads.
- `NoOfFragmentLogParts`: Number of redo log file groups belonging to this data node.
- `NumCPUs`: Specify number of CPUs to use with AutomaticThreadConfig.
- `PartitionsPerNode`: Determines the number of table partitions created on each data node; not used if ClassicFragmentation is enabled.
- `ThreadConfig`: Used for configuration of multithreaded data nodes (ndbmtd). Default is empty string; see documentation for syntax and other information.

23.4.2.2 NDB Cluster Management Node Configuration Parameters

The listing in this section provides information about parameters used in the `[ndb_mgmd]` or `[mgm]` section of a `config.ini` file for configuring NDB Cluster management nodes. For detailed descriptions and other additional information about each of these parameters, see Section 23.4.3.5, “Defining an NDB Cluster Management Server”.

- `ArbitrationDelay`: When asked to arbitrate, arbitrator waits this long before voting (milliseconds).
- `ArbitrationRank`: If 0, then management node is not arbitrator. Kernel selects arbitrators in order 1, 2.
- `DataDir`: Data directory for this node.
- `ExecuteOnComputer`: String referencing earlier defined COMPUTER.
- `ExtraSendBufferMemory`: Memory to use for send buffers in addition to any allocated by TotalSendBufferMemory or SendBufferMemory. Default (0) allows up to 16MB.
- `HeartbeatIntervalMgmdMgmd`: Time between management-node-to-management-node heartbeats; connection between management nodes is considered lost after 3 missed heartbeats.
- `HeartbeatThreadPriority`: Set heartbeat thread policy and priority for management nodes; see manual for allowed values.
- `HostName`: Host name or IP address for this management node.
- `Id`: Number identifying management node. Now deprecated; use NodeId instead.
- `LogDestination`: Where to send log messages: console, system log, or specified log file.
- `NodeId`: Number uniquely identifying management node among all nodes in cluster.
- `PortNumber`: Port number to send commands to and fetch configuration from management server.
- `PortNumberStats`: Port number used to get statistical information from management server.
- `TotalSendBufferMemory`: Total memory to use for all transporter send buffers.
- `wan`: Use WAN TCP setting as default.

Note

After making changes in a management node's configuration, it is necessary to perform a rolling restart of the cluster for the new configuration to take effect.

> See Section 23.4.3.5, “Defining an NDB Cluster Management Server”, for more information.
>
> To add new management servers to a running NDB Cluster, it is also necessary perform a rolling restart of all cluster nodes after modifying any existing `config.ini` files. For more information about issues arising when using multiple management nodes, see Section 23.2.7.10, “Limitations Relating to Multiple NDB Cluster Nodes”.

23.4.2.3 NDB Cluster SQL Node and API Node Configuration Parameters

The listing in this section provides information about parameters used in the `[mysqld]` and `[api]` sections of a `config.ini` file for configuring NDB Cluster SQL nodes and API nodes. For detailed descriptions and other additional information about each of these parameters, see Section 23.4.3.7, “Defining SQL and Other API Nodes in an NDB Cluster”.

- `ApiVerbose`: Enable NDB API debugging; for NDB development.
- `ArbitrationDelay`: When asked to arbitrate, arbitrator waits this many milliseconds before voting.
- `ArbitrationRank`: If 0, then API node is not arbitrator. Kernel selects arbitrators in order 1, 2.
- `AutoReconnect`: Specifies whether an API node should reconnect fully when disconnected from cluster.
- `BatchByteSize`: Default batch size in bytes.
- `BatchSize`: Default batch size in number of records.
- `ConnectBackoffMaxTime`: Specifies longest time in milliseconds (~100ms resolution) to allow between connection attempts to any given data node by this API node. Excludes time elapsed while connection attempts are ongoing, which in worst case can take several seconds. Disable by setting to 0. If no data nodes are currently connected to this API node, StartConnectBackoffMaxTime is used instead.
- `ConnectionMap`: Specifies which data nodes to connect.
- `DefaultHashMapSize`: Set size (in buckets) to use for table hash maps. Three values are supported: 0, 240, and 3840.
- `DefaultOperationRedoProblemAction`: How operations are handled in event that RedoOverCommitCounter is exceeded.
- `ExecuteOnComputer`: String referencing earlier defined COMPUTER.
- `ExtraSendBufferMemory`: Memory to use for send buffers in addition to any allocated by TotalSendBufferMemory or SendBufferMemory. Default (0) allows up to 16MB.
- `HeartbeatThreadPriority`: Set heartbeat thread policy and priority for API nodes; see manual for allowed values.
- `HostName`: Host name or IP address for this SQL or API node.
- `Id`: Number identifying MySQL server or API node (Id). Now deprecated; use NodeId instead.
- `MaxScanBatchSize`: Maximum collective batch size for one scan.
- `NodeId`: Number uniquely identifying SQL node or API node among all nodes in cluster.
- `StartConnectBackoffMaxTime`: Same as ConnectBackoffMaxTime except that this parameter is used in its place if no data nodes are connected to this API node.
- `TotalSendBufferMemory`: Total memory to use for all transporter send buffers.
- `wan`: Use WAN TCP setting as default.

For a discussion of MySQL server options for NDB Cluster, see MySQL Server Options for NDB Cluster. For information about MySQL server system variables relating to NDB Cluster, see NDB Cluster System Variables.

Note

To add new SQL or API nodes to the configuration of a running NDB Cluster, it is necessary to perform a rolling restart of all cluster nodes after adding new `[mysqld]` or `[api]` sections to the `config.ini` file (or files, if you are using more than one management server). This must be done before the new SQL or API nodes can connect to the cluster.

It is *not* necessary to perform any restart of the cluster if new SQL or API nodes can employ previously unused API slots in the cluster configuration to connect to the cluster.

23.4.2.4 Other NDB Cluster Configuration Parameters

The listings in this section provide information about parameters used in the `[computer]`, `[tcp]`, and `[shm]` sections of a `config.ini` file for configuring NDB Cluster. For detailed descriptions and additional information about individual parameters, see Section 23.4.3.10, “NDB Cluster TCP/IP Connections”, or Section 23.4.3.12, “NDB Cluster Shared-Memory Connections”, as appropriate.

The following parameters apply to the `config.ini` file's `[computer]` section:

- `HostName`: Host name or IP address of this computer.
- `Id`: Unique identifier for this computer.

The following parameters apply to the `config.ini` file's `[tcp]` section:

- `AllowUnresolvedHostNames`: When false (default), failure by management node to resolve host name results in fatal error; when true, unresolved host names are reported as warnings only.
- `Checksum`: If checksum is enabled, all signals between nodes are checked for errors.
- `Group`: Used for group proximity; smaller value is interpreted as being closer.
- `HostName1`: Name or IP address of first of two computers joined by TCP connection.
- `HostName2`: Name or IP address of second of two computers joined by TCP connection.
- `NodeId1`: ID of node (data node, API node, or management node) on one side of connection.
- `NodeId2`: ID of node (data node, API node, or management node) on one side of connection.
- `NodeIdServer`: Set server side of TCP connection.
- `OverloadLimit`: When more than this many unsent bytes are in send buffer, connection is considered overloaded.
- `PreferIPVersion`: Indicate DNS resolver preference for IP version 4 or 6.
- `PreSendChecksum`: If this parameter and Checksum are both enabled, perform pre-send checksum checks, and check all TCP signals between nodes for errors.
- `Proxy`: ....
- `ReceiveBufferMemory`: Bytes of buffer for signals received by this node.
- `SendBufferMemory`: Bytes of TCP buffer for signals sent from this node.
- `SendSignalId`: Sends ID in each signal. Used in trace files. Defaults to true in debug builds.
- `TCP_MAXSEG_SIZE`: Value used for TCP_MAXSEG.

- `TCP_RCV_BUF_SIZE`: Value used for SO_RCVBUF.
- `TCP_SND_BUF_SIZE`: Value used for SO_SNDBUF.
- `TcpBind_INADDR_ANY`: Bind InAddrAny instead of host name for server part of connection.

The following parameters apply to the `config.ini` file's `[shm]` section:

- `Checksum`: If checksum is enabled, all signals between nodes are checked for errors.
- `Group`: Used for group proximity; smaller value is interpreted as being closer.
- `HostName1`: Name or IP address of first of two computers joined by SHM connection.
- `HostName2`: Name or IP address of second of two computers joined by SHM connection.
- `NodeId1`: ID of node (data node, API node, or management node) on one side of connection.
- `NodeId2`: ID of node (data node, API node, or management node) on one side of connection.
- `NodeIdServer`: Set server side of SHM connection.
- `OverloadLimit`: When more than this many unsent bytes are in send buffer, connection is considered overloaded.
- `PreSendChecksum`: If this parameter and Checksum are both enabled, perform pre-send checksum checks, and check all SHM signals between nodes for errors.
- `SendBufferMemory`: Bytes in shared memory buffer for signals sent from this node.
- `SendSignalId`: Sends ID in each signal. Used in trace files.
- `ShmKey`: Shared memory key; when set to 1, this is calculated by NDB.
- `ShmSpinTime`: When receiving, number of microseconds to spin before sleeping.
- `ShmSize`: Size of shared memory segment.
- `Signum`: Signal number to be used for signalling.

23.4.2.5 NDB Cluster mysqld Option and Variable Reference

The following list includes command-line options, system variables, and status variables applicable within `mysqld` when it is running as an SQL node in an NDB Cluster. For a reference to *all* command-line options, system variables, and status variables used with or relating to `mysqld`, see Section 5.1.4, “Server Option, System Variable, and Status Variable Reference”.

- `Com_show_ndb_status`: Count of SHOW NDB STATUS statements.
- `Handler_discover`: Number of times that tables have been discovered.
- `ndb-applier-allow-skip-epoch`: Lets replication applier skip epochs.
- `ndb-batch-size`: Size (in bytes) to use for NDB transaction batches.
- `ndb-blob-read-batch-bytes`: Specifies size in bytes that large BLOB reads should be batched into. 0 = no limit.
- `ndb-blob-write-batch-bytes`: Specifies size in bytes that large BLOB writes should be batched into. 0 = no limit.
- `ndb-cluster-connection-pool`: Number of connections to cluster used by MySQL.
- `ndb-cluster-connection-pool-nodeids`: Comma-separated list of node IDs for connections to cluster used by MySQL; number of nodes in list must match value set for --ndb-cluster-connection-pool.

- `ndb-connectstring`: Address of NDB management server distributing configuration information for this cluster.
- `ndb-default-column-format`: Use this value (FIXED or DYNAMIC) by default for COLUMN_FORMAT and ROW_FORMAT options when creating or adding table columns.
- `ndb-deferred-constraints`: Specifies that constraint checks on unique indexes (where these are supported) should be deferred until commit time. Not normally needed or used; for testing purposes only.
- `ndb-distribution`: Default distribution for new tables in NDBCLUSTER (KEYHASH or LINHASH, default is KEYHASH).
- `ndb-log-apply-status`: Cause MySQL server acting as replica to log mysql.ndb_apply_status updates received from its immediate source in its own binary log, using its own server ID. Effective only if server is started with --ndbcluster option.
- `ndb-log-empty-epochs`: When enabled, causes epochs in which there were no changes to be written to ndb_apply_status and ndb_binlog_index tables, even when --log-slave-updates is enabled.
- `ndb-log-empty-update`: When enabled, causes updates that produced no changes to be written to ndb_apply_status and ndb_binlog_index tables, even when --log-slave-updates is enabled.
- `ndb-log-exclusive-reads`: Log primary key reads with exclusive locks; allow conflict resolution based on read conflicts.
- `ndb-log-fail-terminate`: Terminate mysqld process if complete logging of all found row events is not possible.
- `ndb-log-orig`: Log originating server id and epoch in mysql.ndb_binlog_index table.
- `ndb-log-transaction-dependency`: Make binary log thread calculate transaction dependencies for every transaction it writes to binary log.
- `ndb-log-transaction-id`: Write NDB transaction IDs in binary log. Requires --log-bin-v1-events=OFF.
- `ndb-log-update-minimal`: Log updates in minimal format.
- `ndb-log-updated-only`: Log complete rows (ON) or updates only (OFF).
- `ndb-log-update-as-write`: Toggles logging of updates on source between updates (OFF) and writes (ON).
- `ndb-mgmd-host`: Set host (and port, if desired) for connecting to management server.
- `ndb-nodeid`: NDB Cluster node ID for this MySQL server.
- `ndb-optimized-node-selection`: Enable optimizations for selection of nodes for transactions. Enabled by default; use --skip-ndb-optimized-node-selection to disable.
- `ndb-transid-mysql-connection-map`: Enable or disable ndb_transid_mysql_connection_map plugin; that is, enable or disable INFORMATION_SCHEMA table having that name.
- `ndb-wait-connected`: Time (in seconds) for MySQL server to wait for connection to cluster management and data nodes before accepting MySQL client connections.
- `ndb-wait-setup`: Time (in seconds) for MySQL server to wait for NDB engine setup to complete.
- `ndb-allow-copying-alter-table`: Set to OFF to keep ALTER TABLE from using copying operations on NDB tables.
- `Ndb_api_adaptive_send_deferred_count`: Number of adaptive send calls not actually sent by this MySQL Server (SQL node).

- `Ndb_api_adaptive_send_deferred_count_session`: Number of adaptive send calls not actually sent in this client session.
- `Ndb_api_adaptive_send_deferred_count_replica`: Number of adaptive send calls not actually sent by this replica.
- `Ndb_api_adaptive_send_deferred_count_slave`: Number of adaptive send calls not actually sent by this replica.
- `Ndb_api_adaptive_send_forced_count`: Number of adaptive sends with forced-send set sent by this MySQL Server (SQL node).
- `Ndb_api_adaptive_send_forced_count_session`: Number of adaptive sends with forced-send set in this client session.
- `Ndb_api_adaptive_send_forced_count_replica`: Number of adaptive sends with forced-send set sent by this replica.
- `Ndb_api_adaptive_send_forced_count_slave`: Number of adaptive sends with forced-send set sent by this replica.
- `Ndb_api_adaptive_send_unforced_count`: Number of adaptive sends without forced-send sent by this MySQL Server (SQL node).
- `Ndb_api_adaptive_send_unforced_count_session`: Number of adaptive sends without forced-send in this client session.
- `Ndb_api_adaptive_send_unforced_count_replica`: Number of adaptive sends without forced-send sent by this replica.
- `Ndb_api_adaptive_send_unforced_count_slave`: Number of adaptive sends without forced-send sent by this replica.
- `Ndb_api_bytes_received_count`: Quantity of data (in bytes) received from data nodes by this MySQL Server (SQL node).
- `Ndb_api_bytes_received_count_session`: Quantity of data (in bytes) received from data nodes in this client session.
- `Ndb_api_bytes_received_count_replica`: Quantity of data (in bytes) received from data nodes by this replica.
- `Ndb_api_bytes_received_count_slave`: Quantity of data (in bytes) received from data nodes by this replica.
- `Ndb_api_bytes_sent_count`: Quantity of data (in bytes) sent to data nodes by this MySQL Server (SQL node).
- `Ndb_api_bytes_sent_count_session`: Quantity of data (in bytes) sent to data nodes in this client session.
- `Ndb_api_bytes_sent_count_replica`: Qunatity of data (in bytes) sent to data nodes by this replica.
- `Ndb_api_bytes_sent_count_slave`: Qunatity of data (in bytes) sent to data nodes by this replica.
- `Ndb_api_event_bytes_count`: Number of bytes of events received by this MySQL Server (SQL node).
- `Ndb_api_event_bytes_count_injector`: Number of bytes of event data received by NDB binary log injector thread.

- `Ndb_api_event_data_count`: Number of row change events received by this MySQL Server (SQL node).
- `Ndb_api_event_data_count_injector`: Number of row change events received by NDB binary log injector thread.
- `Ndb_api_event_nondata_count`: Number of events received, other than row change events, by this MySQL Server (SQL node).
- `Ndb_api_event_nondata_count_injector`: Number of events received, other than row change events, by NDB binary log injector thread.
- `Ndb_api_pk_op_count`: Number of operations based on or using primary keys by this MySQL Server (SQL node).
- `Ndb_api_pk_op_count_session`: Number of operations based on or using primary keys in this client session.
- `Ndb_api_pk_op_count_replica`: Number of operations based on or using primary keys by this replica.
- `Ndb_api_pk_op_count_slave`: Number of operations based on or using primary keys by this replica.
- `Ndb_api_pruned_scan_count`: Number of scans that have been pruned to one partition by this MySQL Server (SQL node).
- `Ndb_api_pruned_scan_count_session`: Number of scans that have been pruned to one partition in this client session.
- `Ndb_api_pruned_scan_count_replica`: Number of scans that have been pruned to one partition by this replica.
- `Ndb_api_pruned_scan_count_slave`: Number of scans that have been pruned to one partition by this replica.
- `Ndb_api_range_scan_count`: Number of range scans that have been started by this MySQL Server (SQL node).
- `Ndb_api_range_scan_count_session`: Number of range scans that have been started in this client session.
- `Ndb_api_range_scan_count_replica`: Number of range scans that have been started by this replica.
- `Ndb_api_range_scan_count_slave`: Number of range scans that have been started by this replica.
- `Ndb_api_read_row_count`: Total number of rows that have been read by this MySQL Server (SQL node).
- `Ndb_api_read_row_count_session`: Total number of rows that have been read in this client session.
- `Ndb_api_read_row_count_replica`: Total number of rows that have been read by this replica.
- `Ndb_api_read_row_count_slave`: Total number of rows that have been read by this replica.
- `Ndb_api_scan_batch_count`: Number of batches of rows received by this MySQL Server (SQL node).
- `Ndb_api_scan_batch_count_session`: Number of batches of rows received in this client session.

- Ndb_api_scan_batch_count_replica: Number of batches of rows received by this replica.
- Ndb_api_scan_batch_count_slave: Number of batches of rows received by this replica.
- Ndb_api_table_scan_count: Number of table scans that have been started, including scans of internal tables, by this MySQL Server (SQL node).
- Ndb_api_table_scan_count_session: Number of table scans that have been started, including scans of internal tables, in this client session.
- Ndb_api_table_scan_count_replica: Number of table scans that have been started, including scans of internal tables, by this replica.
- Ndb_api_table_scan_count_slave: Number of table scans that have been started, including scans of internal tables, by this replica.
- Ndb_api_trans_abort_count: Number of transactions aborted by this MySQL Server (SQL node).
- Ndb_api_trans_abort_count_session: Number of transactions aborted in this client session.
- Ndb_api_trans_abort_count_replica: Number of transactions aborted by this replica.
- Ndb_api_trans_abort_count_slave: Number of transactions aborted by this replica.
- Ndb_api_trans_close_count: Number of transactions aborted (may be greater than sum of TransCommitCount and TransAbortCount) by this MySQL Server (SQL node).
- Ndb_api_trans_close_count_session: Number of transactions aborted (may be greater than sum of TransCommitCount and TransAbortCount) in this client session.
- Ndb_api_trans_close_count_replica: Number of transactions aborted (may be greater than sum of TransCommitCount and TransAbortCount) by this replica.
- Ndb_api_trans_close_count_slave: Number of transactions aborted (may be greater than sum of TransCommitCount and TransAbortCount) by this replica.
- Ndb_api_trans_commit_count: Number of transactions committed by this MySQL Server (SQL node).
- Ndb_api_trans_commit_count_session: Number of transactions committed in this client session.
- Ndb_api_trans_commit_count_replica: Number of transactions committed by this replica.
- Ndb_api_trans_commit_count_slave: Number of transactions committed by this replica.
- Ndb_api_trans_local_read_row_count: Total number of rows that have been read by this MySQL Server (SQL node).
- Ndb_api_trans_local_read_row_count_session: Total number of rows that have been read in this client session.
- Ndb_api_trans_local_read_row_count_replica: Total number of rows that have been read by this replica.
- Ndb_api_trans_local_read_row_count_slave: Total number of rows that have been read by this replica.
- Ndb_api_trans_start_count: Number of transactions started by this MySQL Server (SQL node).
- Ndb_api_trans_start_count_session: Number of transactions started in this client session.
- Ndb_api_trans_start_count_replica: Number of transactions started by this replica.

- Ndb_api_trans_start_count_slave: Number of transactions started by this replica.
- Ndb_api_uk_op_count: Number of operations based on or using unique keys by this MySQL Server (SQL node).
- Ndb_api_uk_op_count_session: Number of operations based on or using unique keys in this client session.
- Ndb_api_uk_op_count_replica: Number of operations based on or using unique keys by this replica.
- Ndb_api_uk_op_count_slave: Number of operations based on or using unique keys by this replica.
- Ndb_api_wait_exec_complete_count: Number of times thread has been blocked while waiting for operation execution to complete by this MySQL Server (SQL node).
- Ndb_api_wait_exec_complete_count_session: Number of times thread has been blocked while waiting for operation execution to complete in this client session.
- Ndb_api_wait_exec_complete_count_replica: Number of times thread has been blocked while waiting for operation execution to complete by this replica.
- Ndb_api_wait_exec_complete_count_slave: Number of times thread has been blocked while waiting for operation execution to complete by this replica.
- Ndb_api_wait_meta_request_count: Number of times thread has been blocked waiting for metadata-based signal by this MySQL Server (SQL node).
- Ndb_api_wait_meta_request_count_session: Number of times thread has been blocked waiting for metadata-based signal in this client session.
- Ndb_api_wait_meta_request_count_replica: Number of times thread has been blocked waiting for metadata-based signal by this replica.
- Ndb_api_wait_meta_request_count_slave: Number of times thread has been blocked waiting for metadata-based signal by this replica.
- Ndb_api_wait_nanos_count: Total time (in nanoseconds) spent waiting for some type of signal from data nodes by this MySQL Server (SQL node).
- Ndb_api_wait_nanos_count_session: Total time (in nanoseconds) spent waiting for some type of signal from data nodes in this client session.
- Ndb_api_wait_nanos_count_replica: Total time (in nanoseconds) spent waiting for some type of signal from data nodes by this replica.
- Ndb_api_wait_nanos_count_slave: Total time (in nanoseconds) spent waiting for some type of signal from data nodes by this replica.
- Ndb_api_wait_scan_result_count: Number of times thread has been blocked while waiting for scan-based signal by this MySQL Server (SQL node).
- Ndb_api_wait_scan_result_count_session: Number of times thread has been blocked while waiting for scan-based signal in this client session.
- Ndb_api_wait_scan_result_count_replica: Number of times thread has been blocked while waiting for scan-based signal by this replica.
- Ndb_api_wait_scan_result_count_slave: Number of times thread has been blocked while waiting for scan-based signal by this replica.
- ndb_autoincrement_prefetch_sz: NDB auto-increment prefetch size.

- `ndb_cache_check_time`: Number of milliseconds between checks of cluster SQL nodes made by MySQL query cache.
- `ndb_clear_apply_status`: Causes RESET SLAVE/RESET REPLICA to clear all rows from ndb_apply_status table; ON by default.
- `Ndb_cluster_node_id`: Node ID of this server when acting as NDB Cluster SQL node.
- `Ndb_config_from_host`: NDB Cluster management server host name or IP address.
- `Ndb_config_from_port`: Port for connecting to NDB Cluster management server.
- `Ndb_config_generation`: Generation number of the current configuration of the cluster.
- `Ndb_conflict_fn_epoch`: Number of rows that have been found in conflict by NDB$EPOCH() NDB replication conflict detection function.
- `Ndb_conflict_fn_epoch2`: Number of rows that have been found in conflict by NDB replication NDB$EPOCH2() conflict detection function.
- `Ndb_conflict_fn_epoch2_trans`: Number of rows that have been found in conflict by NDB replication NDB$EPOCH2_TRANS() conflict detection function.
- `Ndb_conflict_fn_epoch_trans`: Number of rows that have been found in conflict by NDB $EPOCH_TRANS() conflict detection function.
- `Ndb_conflict_fn_max`: Number of times that NDB replication conflict resolution based on "greater timestamp wins" has been applied to update and delete operations.
- `Ndb_conflict_fn_max_del_win`: Number of times that NDB replication conflict resolution based on outcome of NDB$MAX_DELETE_WIN() has been applied to update and delete operations.
- `Ndb_conflict_fn_max_ins`: Number of times that NDB replication conflict resolution based on "greater timestamp wins" has been applied to insert operations.
- `Ndb_conflict_fn_max_del_win_ins`: Number of times that NDB replication conflict resolution based on outcome of NDB$MAX_DEL_WIN_INS() has been applied to insert operations.
- `Ndb_conflict_fn_old`: Number of times in NDB replication "same timestamp wins" conflict resolution has been applied.
- `Ndb_conflict_last_conflict_epoch`: Most recent NDB epoch on this replica in which some conflict was detected.
- `Ndb_conflict_last_stable_epoch`: Number of rows found to be in conflict by transactional conflict function.
- `Ndb_conflict_reflected_op_discard_count`: Number of reflected operations that were not applied due error during execution.
- `Ndb_conflict_reflected_op_prepare_count`: Number of reflected operations received that have been prepared for execution.
- `Ndb_conflict_refresh_op_count`: Number of refresh operations that have been prepared.
- `ndb_conflict_role`: Role for replica to play in conflict detection and resolution. Value is one of PRIMARY, SECONDARY, PASS, or NONE (default). Can be changed only when replication SQL thread is stopped. See documentation for further information.
- `Ndb_conflict_trans_conflict_commit_count`: Number of epoch transactions committed after requiring transactional conflict handling.
- `Ndb_conflict_trans_detect_iter_count`: Number of internal iterations required to commit epoch transaction. Should be (slightly) greater than or equal to Ndb_conflict_trans_conflict_commit_count.

- `Ndb_conflict_trans_reject_count`: Number of transactions rejected after being found in conflict by transactional conflict function.
- `Ndb_conflict_trans_row_conflict_count`: Number of rows found in conflict by transactional conflict function. Includes any rows included in or dependent on conflicting transactions.
- `Ndb_conflict_trans_row_reject_count`: Total number of rows realigned after being found in conflict by transactional conflict function. Includes Ndb_conflict_trans_row_conflict_count and any rows included in or dependent on conflicting transactions.
- `ndb_data_node_neighbour`: Specifies cluster data node "closest" to this MySQL Server, for transaction hinting and fully replicated tables.
- `ndb_default_column_format`: Sets default row format and column format (FIXED or DYNAMIC) used for new NDB tables.
- `ndb_deferred_constraints`: Specifies that constraint checks should be deferred (where these are supported). Not normally needed or used; for testing purposes only.
- `ndb_dbg_check_shares`: Check for any lingering shares (debug builds only).
- `ndb-schema-dist-timeout`: How long to wait before detecting timeout during schema distribution.
- `ndb_distribution`: Default distribution for new tables in NDBCLUSTER (KEYHASH or LINHASH, default is KEYHASH).
- `Ndb_epoch_delete_delete_count`: Number of delete-delete conflicts detected (delete operation is applied, but row does not exist).
- `ndb_eventbuffer_free_percent`: Percentage of free memory that should be available in event buffer before resumption of buffering, after reaching limit set by ndb_eventbuffer_max_alloc.
- `ndb_eventbuffer_max_alloc`: Maximum memory that can be allocated for buffering events by NDB API. Defaults to 0 (no limit).
- `Ndb_execute_count`: Number of round trips to NDB kernel made by operations.
- `ndb_extra_logging`: Controls logging of NDB Cluster schema, connection, and data distribution events in MySQL error log.
- `ndb_force_send`: Forces sending of buffers to NDB immediately, without waiting for other threads.
- `ndb_fully_replicated`: Whether new NDB tables are fully replicated.
- `ndb_index_stat_enable`: Use NDB index statistics in query optimization.
- `ndb_index_stat_option`: Comma-separated list of tunable options for NDB index statistics; list should contain no spaces.
- `ndb_join_pushdown`: Enables pushing down of joins to data nodes.
- `Ndb_last_commit_epoch_server`: Epoch most recently committed by NDB.
- `Ndb_last_commit_epoch_session`: Epoch most recently committed by this NDB client.
- `ndb_log_apply_status`: Whether or not MySQL server acting as replica logs mysql.ndb_apply_status updates received from its immediate source in its own binary log, using its own server ID.
- `ndb_log_bin`: Write updates to NDB tables in binary log. Effective only if binary logging is enabled with --log-bin.

- ndb_log_binlog_index: Insert mapping between epochs and binary log positions into ndb_binlog_index table. Defaults to ON. Effective only if binary logging is enabled.
- ndb_log_empty_epochs: When enabled, epochs in which there were no changes are written to ndb_apply_status and ndb_binlog_index tables, even when log_replica_updates or log_slave_updates is enabled.
- ndb_log_empty_update: When enabled, updates which produce no changes are written to ndb_apply_status and ndb_binlog_index tables, even when log_replica_updates or log_slave_updates is enabled.
- ndb_log_exclusive_reads: Log primary key reads with exclusive locks; allow conflict resolution based on read conflicts.
- ndb_log_orig: Whether id and epoch of originating server are recorded in mysql.ndb_binlog_index table. Set using --ndb-log-orig option when starting mysqld.
- ndb_log_transaction_id: Whether NDB transaction IDs are written into binary log (Read-only).
- ndb_log_transaction_compression: Whether to compress NDB binary log; can also be enabled on startup by enabling --binlog-transaction-compression option.
- ndb_log_transaction_compression_level_zstd: The ZSTD compression level to use when writing compressed transactions to the NDB binary log.
- ndb_metadata_check: Enable auto-detection of NDB metadata changes with respect to MySQL data dictionary; enabled by default.
- Ndb_metadata_blacklist_size: Number of NDB metadata objects that NDB binlog thread has failed to synchronize; renamed in NDB 8.0.22 as Ndb_metadata_excluded_count.
- ndb_metadata_check_interval: Interval in seconds to perform check for NDB metadata changes with respect to MySQL data dictionary.
- Ndb_metadata_detected_count: Number of times NDB metadata change monitor thread has detected changes.
- Ndb_metadata_excluded_count: Number of NDB metadata objects that NDB binlog thread has failed to synchronize.
- ndb_metadata_sync: Triggers immediate synchronization of all changes between NDB dictionary and MySQL data dictionary; causes ndb_metadata_check and ndb_metadata_check_interval values to be ignored. Resets to false when synchronization is complete.
- Ndb_metadata_synced_count: Number of NDB metadata objects which have been synchronized.
- Ndb_number_of_data_nodes: Number of data nodes in this NDB cluster; set only if server participates in cluster.
- ndb-optimization-delay: Number of milliseconds to wait between processing sets of rows by OPTIMIZE TABLE on NDB tables.
- ndb_optimized_node_selection: Determines how SQL node chooses cluster data node to use as transaction coordinator.
- Ndb_pruned_scan_count: Number of scans executed by NDB since cluster was last started where partition pruning could be used.
- Ndb_pushed_queries_defined: Number of joins that API nodes have attempted to push down to data nodes.
- Ndb_pushed_queries_dropped: Number of joins that API nodes have tried to push down, but failed.

- Ndb_pushed_queries_executed: Number of joins successfully pushed down and executed on data nodes.
- Ndb_pushed_reads: Number of reads executed on data nodes by pushed-down joins.
- ndb_read_backup: Enable read from any replica for all NDB tables; use NDB_TABLE=READ_BACKUP={0|1} with CREATE TABLE or ALTER TABLE to enable or disable for individual NDB tables.
- ndb_recv_thread_activation_threshold: Activation threshold when receive thread takes over polling of cluster connection (measured in concurrently active threads).
- ndb_recv_thread_cpu_mask: CPU mask for locking receiver threads to specific CPUs; specified as hexadecimal. See documentation for details.
- Ndb_replica_max_replicated_epoch: Most recently committed NDB epoch on this replica. When this value is greater than or equal to Ndb_conflict_last_conflict_epoch, no conflicts have yet been detected.
- ndb_replica_batch_size: Batch size in bytes for replica applier.
- ndb_report_thresh_binlog_epoch_slip: NDB 7.5 and later: Threshold for number of epochs completely buffered, but not yet consumed by binlog injector thread which when exceeded generates BUFFERED_EPOCHS_OVER_THRESHOLD event buffer status message; prior to NDB 7.5: Threshold for number of epochs to lag behind before reporting binary log status.
- ndb_report_thresh_binlog_mem_usage: Threshold for percentage of free memory remaining before reporting binary log status.
- ndb_row_checksum: When enabled, set row checksums; enabled by default.
- Ndb_scan_count: Total number of scans executed by NDB since cluster was last started.
- ndb_schema_dist_lock_wait_timeout: Time during schema distribution to wait for lock before returning error.
- ndb_schema_dist_timeout: Time to wait before detecting timeout during schema distribution.
- ndb_schema_dist_upgrade_allowed: Allow schema distribution table upgrade when connecting to NDB.
- ndb_show_foreign_key_mock_tables: Show mock tables used to support foreign_key_checks=0.
- ndb_slave_conflict_role: Role for replica to play in conflict detection and resolution. Value is one of PRIMARY, SECONDARY, PASS, or NONE (default). Can be changed only when replication SQL thread is stopped. See documentation for further information.
- Ndb_slave_max_replicated_epoch: Most recently committed NDB epoch on this replica. When this value is greater than or equal to Ndb_conflict_last_conflict_epoch, no conflicts have yet been detected.
- Ndb_system_name: Configured cluster system name; empty if server not connected to NDB.
- ndb_table_no_logging: NDB tables created when this setting is enabled are not checkpointed to disk (although table schema files are created). Setting in effect when table is created with or altered to use NDBCLUSTER persists for table's lifetime.
- ndb_table_temporary: NDB tables are not persistent on disk: no schema files are created and tables are not logged.
- Ndb_trans_hint_count_session: Number of transactions using hints that have been started in this session.

- `ndb_use_copying_alter_table`: Use copying ALTER TABLE operations in NDB Cluster.
- `ndb_use_exact_count`: Forces NDB to use a count of records during SELECT COUNT(*) query planning to speed up this type of query.
- `ndb_use_transactions`: Set to OFF, to disable transaction support by NDB. Not recommended except in certain special cases; see documentation for details.
- `ndb_version`: Shows build and NDB engine version as an integer.
- `ndb_version_string`: Shows build information including NDB engine version in ndb-x.y.z format.
- `ndbcluster`: Enable NDB Cluster (if this version of MySQL supports it). Disabled by `--skip-ndbcluster`.
- `ndbinfo`: Enable ndbinfo plugin, if supported.
- `ndbinfo_database`: Name used for NDB information database; read only.
- `ndbinfo_max_bytes`: Used for debugging only.
- `ndbinfo_max_rows`: Used for debugging only.
- `ndbinfo_offline`: Put ndbinfo database into offline mode, in which no rows are returned from tables or views.
- `ndbinfo_show_hidden`: Whether to show ndbinfo internal base tables in mysql client; default is OFF.
- `ndbinfo_table_prefix`: Prefix to use for naming ndbinfo internal base tables; read only.
- `ndbinfo_version`: ndbinfo engine version; read only.
- `replica_allow_batching`: Turns update batching on and off for replica.
- `server_id_bits`: Number of least significant bits in server_id actually used for identifying server, permitting NDB API applications to store application data in most significant bits. server_id must be less than 2 to power of this value.
- `skip-ndbcluster`: Disable NDB Cluster storage engine.
- `slave_allow_batching`: Turns update batching on and off for replica.
- `transaction_allow_batching`: Allows batching of statements within one transaction. Disable AUTOCOMMIT to use.

23.4.3 NDB Cluster Configuration Files

Configuring NDB Cluster requires working with two files:

- `my.cnf`: Specifies options for all NDB Cluster executables. This file, with which you should be familiar with from previous work with MySQL, must be accessible by each executable running in the cluster.
- `config.ini`: This file, sometimes known as the *global configuration file*, is read only by the NDB Cluster management server, which then distributes the information contained therein to all processes participating in the cluster. `config.ini` contains a description of each node involved in the cluster. This includes configuration parameters for data nodes and configuration parameters for connections between all nodes in the cluster. For a quick reference to the sections that can appear in this file, and what sorts of configuration parameters may be placed in each section, see Sections of the `config.ini` File.

Caching of configuration data. `NDB` uses *stateful configuration*. Rather than reading the global configuration file every time the management server is restarted, the management server caches the

configuration the first time it is started, and thereafter, the global configuration file is read only when one of the following conditions is true:

- **The management server is started using the --initial option.** When `--initial` is used, the global configuration file is re-read, any existing cache files are deleted, and the management server creates a new configuration cache.
- **The management server is started using the --reload option.** The `--reload` option causes the management server to compare its cache with the global configuration file. If they differ, the management server creates a new configuration cache; any existing configuration cache is preserved, but not used. If the management server's cache and the global configuration file contain the same configuration data, then the existing cache is used, and no new cache is created.
- **The management server is started using --config-cache=FALSE.** This disables `--config-cache` (enabled by default), and can be used to force the management server to bypass configuration caching altogether. In this case, the management server ignores any configuration files that may be present, always reading its configuration data from the `config.ini` file instead.
- **No configuration cache is found.** In this case, the management server reads the global configuration file and creates a cache containing the same configuration data as found in the file.

Configuration cache files. The management server by default creates configuration cache files in a directory named `mysql-cluster` in the MySQL installation directory. (If you build NDB Cluster from source on a Unix system, the default location is `/usr/local/mysql-cluster`.) This can be overridden at runtime by starting the management server with the `--configdir` option. Configuration cache files are binary files named according to the pattern `ndb_`*`node_id`*`_config.bin.`*`seq_id`*, where *`node_id`* is the management server's node ID in the cluster, and *`seq_id`* is a cache identifier. Cache files are numbered sequentially using *`seq_id`*, in the order in which they are created. The management server uses the latest cache file as determined by the *`seq_id`*.

Note

It is possible to roll back to a previous configuration by deleting later configuration cache files, or by renaming an earlier cache file so that it has a higher *`seq_id`*. However, since configuration cache files are written in a binary format, you should not attempt to edit their contents by hand.

For more information about the `--configdir`, `--config-cache`, `--initial`, and `--reload` options for the NDB Cluster management server, see Section 23.5.4, “ndb_mgmd — The NDB Cluster Management Server Daemon”.

We are continuously making improvements in NDB Cluster configuration and attempting to simplify this process. Although we strive to maintain backward compatibility, there may be times when introduce an incompatible change. In such cases we try to let NDB Cluster users know in advance if a change is not backward compatible. If you find such a change and we have not documented it, please report it in the MySQL bugs database using the instructions given in Section 1.5, “How to Report Bugs or Problems”.

23.4.3.1 NDB Cluster Configuration: Basic Example

To support NDB Cluster, you should update `my.cnf` as shown in the following example. You may also specify these parameters on the command line when invoking the executables.

Note

The options shown here should not be confused with those that are used in `config.ini` global configuration files. Global configuration options are discussed later in this section.

```
# my.cnf
# example additions to my.cnf for NDB Cluster
# (valid in MySQL 8.0)
```

```
# enable ndbcluster storage engine, and provide connection string for
# management server host (default port is 1186)
[mysqld]
ndbcluster
ndb-connectstring=ndb_mgmd.mysql.com


# provide connection string for management server host (default port: 1186)
[ndbd]
connect-string=ndb_mgmd.mysql.com

# provide connection string for management server host (default port: 1186)
[ndb_mgm]
connect-string=ndb_mgmd.mysql.com

# provide location of cluster configuration file
# IMPORTANT: When starting the management server with this option in the
# configuration file, the use of --initial or --reload on the command line when
# invoking ndb_mgmd is also required.
[ndb_mgmd]
config-file=/etc/config.ini
```

(For more information on connection strings, see Section 23.4.3.3, “NDB Cluster Connection Strings”.)

```
# my.cnf
# example additions to my.cnf for NDB Cluster
# (works on all versions)

# enable ndbcluster storage engine, and provide connection string for management
# server host to the default port 1186
[mysqld]
ndbcluster
ndb-connectstring=ndb_mgmd.mysql.com:1186
```

Important

Once you have started a `mysqld` process with the `NDBCLUSTER` and `ndb-connectstring` parameters in the `[mysqld]` in the `my.cnf` file as shown previously, you cannot execute any `CREATE TABLE` or `ALTER TABLE` statements without having actually started the cluster. Otherwise, these statements fail with an error. *This is by design*.

You may also use a separate `[mysql_cluster]` section in the cluster `my.cnf` file for settings to be read and used by all executables:

```
# cluster-specific settings
[mysql_cluster]
ndb-connectstring=ndb_mgmd.mysql.com:1186
```

For additional `NDB` variables that can be set in the `my.cnf` file, see NDB Cluster System Variables.

The NDB Cluster global configuration file is by convention named `config.ini` (but this is not required). If needed, it is read by `ndb_mgmd` at startup and can be placed in any location that can be read by it. The location and name of the configuration are specified using `--config-file=`*`path_name`* with `ndb_mgmd` on the command line. This option has no default value, and is ignored if `ndb_mgmd` uses the configuration cache.

The global configuration file for NDB Cluster uses INI format, which consists of sections preceded by section headings (surrounded by square brackets), followed by the appropriate parameter names and values. One deviation from the standard INI format is that the parameter name and value can be separated by a colon (`:`) as well as the equal sign (`=`); however, the equal sign is preferred. Another deviation is that sections are not uniquely identified by section name. Instead, unique sections (such as two different nodes of the same type) are identified by a unique ID specified as a parameter within the section.

Default values are defined for most parameters, and can also be specified in `config.ini`. To create a default value section, simply add the word `default` to the section name. For example, an `[ndbd]`

section contains parameters that apply to a particular data node, whereas an `[ndbd default]` section contains parameters that apply to all data nodes. Suppose that all data nodes should use the same data memory size. To configure them all, create an `[ndbd default]` section that contains a `DataMemory` line to specify the data memory size.

If used, the `[ndbd default]` section must precede any `[ndbd]` sections in the configuration file. This is also true for `default` sections of any other type.

Note

In some older releases of NDB Cluster, there was no default value for `NoOfReplicas`, which always had to be specified explicitly in the `[ndbd default]` section. Although this parameter now has a default value of 2, which is the recommended setting in most common usage scenarios, it is still recommended practice to set this parameter explicitly.

The global configuration file must define the computers and nodes involved in the cluster and on which computers these nodes are located. An example of a simple configuration file for a cluster consisting of one management server, two data nodes and two MySQL servers is shown here:

```
# file "config.ini" - 2 data nodes and 2 SQL nodes
# This file is placed in the startup directory of ndb_mgmd (the
# management server)
# The first MySQL Server can be started from any host. The second
# can be started only on the host mysqld_5.mysql.com

[ndbd default]
NoOfReplicas= 2
DataDir= /var/lib/mysql-cluster

[ndb_mgmd]
Hostname= ndb_mgmd.mysql.com
DataDir= /var/lib/mysql-cluster

[ndbd]
HostName= ndbd_2.mysql.com

[ndbd]
HostName= ndbd_3.mysql.com

[mysqld]
[mysqld]
HostName= mysqld_5.mysql.com
```

Note

The preceding example is intended as a minimal starting configuration for purposes of familiarization with NDB Cluster , and is almost certain not to be sufficient for production settings. See Section 23.4.3.2, “Recommended Starting Configuration for NDB Cluster”, which provides a more complete example starting configuration.

Each node has its own section in the `config.ini` file. For example, this cluster has two data nodes, so the preceding configuration file contains two `[ndbd]` sections defining these nodes.

Note

Do not place comments on the same line as a section heading in the `config.ini` file; this causes the management server not to start because it cannot parse the configuration file in such cases.

Sections of the config.ini File

There are six different sections that you can use in the `config.ini` configuration file, as described in the following list:

- `[computer]`: Defines cluster hosts. This is not required to configure a viable NDB Cluster, but be may used as a convenience when setting up a large cluster. See Section 23.4.3.4, “Defining Computers in an NDB Cluster”, for more information.
- `[ndbd]`: Defines a cluster data node (`ndbd` process). See Section 23.4.3.6, “Defining NDB Cluster Data Nodes”, for details.
- `[mysqld]`: Defines the cluster's MySQL server nodes (also called SQL or API nodes). For a discussion of SQL node configuration, see Section 23.4.3.7, “Defining SQL and Other API Nodes in an NDB Cluster”.
- `[mgm]` or `[ndb_mgmd]`: Defines a cluster management server (MGM) node. For information concerning the configuration of management nodes, see Section 23.4.3.5, “Defining an NDB Cluster Management Server”.
- `[tcp]`: Defines a TCP/IP connection between cluster nodes, with TCP/IP being the default transport protocol. Normally, `[tcp]` or `[tcp default]` sections are not required to set up an NDB Cluster, as the cluster handles this automatically; however, it may be necessary in some situations to override the defaults provided by the cluster. See Section 23.4.3.10, “NDB Cluster TCP/IP Connections”, for information about available TCP/IP configuration parameters and how to use them. (You may also find Section 23.4.3.11, “NDB Cluster TCP/IP Connections Using Direct Connections” to be of interest in some cases.)
- `[shm]`: Defines shared-memory connections between nodes. In MySQL 8.0, it is enabled by default, but should still be considered experimental. For a discussion of SHM interconnects, see Section 23.4.3.12, “NDB Cluster Shared-Memory Connections”.
- `[sci]`: Defines Scalable Coherent Interface connections between cluster data nodes. Not supported in NDB 8.0.

You can define `default` values for each section. If used, a `default` section should come before any other sections of that type. For example, an `[ndbd default]` section should appear in the configuration file before any `[ndbd]` sections.

NDB Cluster parameter names are case-insensitive, unless specified in MySQL Server `my.cnf` or `my.ini` files.

23.4.3.2 Recommended Starting Configuration for NDB Cluster

Achieving the best performance from an NDB Cluster depends on a number of factors including the following:

- NDB Cluster software version
- Numbers of data nodes and SQL nodes
- Hardware
- Operating system
- Amount of data to be stored
- Size and type of load under which the cluster is to operate

Therefore, obtaining an optimum configuration is likely to be an iterative process, the outcome of which can vary widely with the specifics of each NDB Cluster deployment. Changes in configuration are also likely to be indicated when changes are made in the platform on which the cluster is run, or in applications that use the NDB Cluster 's data. For these reasons, it is not possible to offer a single configuration that is ideal for all usage scenarios. However, in this section, we provide a recommended base configuration.

Starting config.ini file. The following `config.ini` file is a recommended starting point for configuring a cluster running NDB Cluster 8.0:

```
# TCP PARAMETERS

[tcp default]
SendBufferMemory=2M
ReceiveBufferMemory=2M

# Increasing the sizes of these 2 buffers beyond the default values
# helps prevent bottlenecks due to slow disk I/O.

# MANAGEMENT NODE PARAMETERS

[ndb_mgmd default]
DataDir=path/to/management/server/data/directory

# It is possible to use a different data directory for each management
# server, but for ease of administration it is preferable to be
# consistent.

[ndb_mgmd]
HostName=management-server-A-hostname
# NodeId=management-server-A-nodeid

[ndb_mgmd]
HostName=management-server-B-hostname
# NodeId=management-server-B-nodeid

# Using 2 management servers helps guarantee that there is always an
# arbitrator in the event of network partitioning, and so is
# recommended for high availability. Each management server must be
# identified by a HostName. You may for the sake of convenience specify
# a NodeId for any management server, although one is allocated
# for it automatically; if you do so, it must be in the range 1-255
# inclusive and must be unique among all IDs specified for cluster
# nodes.

# DATA NODE PARAMETERS

[ndbd default]
NoOfReplicas=2

# Using two fragment replicas is recommended to guarantee availability of data;
# using only one fragment replica does not provide any redundancy, which means
# that the failure of a single data node causes the entire cluster to shut down.
# It is also possible (but not required) in NDB 8.0 to use more than two
# fragment replicas, although two fragment replicas are sufficient to provide
# high availability.

LockPagesInMainMemory=1

# On Linux and Solaris systems, setting this parameter locks data node
# processes into memory. Doing so prevents them from swapping to disk,
# which can severely degrade cluster performance.

DataMemory=3456M

# The value provided for DataMemory assumes 4 GB RAM
# per data node. However, for best results, you should first calculate
# the memory that would be used based on the data you actually plan to
# store (you may find the ndb_size.pl utility helpful in estimating
# this), then allow an extra 20% over the calculated values. Naturally,
# you should ensure that each data node host has at least as much
# physical memory as the sum of these two values.

# ODirect=1

# Enabling this parameter causes NDBCLUSTER to try using O_DIRECT
# writes for local checkpoints and redo logs; this can reduce load on
# CPUs. We recommend doing so when using NDB Cluster on systems running
# Linux kernel 2.6 or later.
```

```
NoOfFragmentLogFiles=300
DataDir=path/to/data/node/data/directory
MaxNoOfConcurrentOperations=100000

SchedulerSpinTimer=400
SchedulerExecutionTimer=100
RealTimeScheduler=1
# Setting these parameters allows you to take advantage of real-time scheduling
# of NDB threads to achieve increased throughput when using ndbd. They
# are not needed when using ndbmtd; in particular, you should not set
# RealTimeScheduler for ndbmtd data nodes.

TimeBetweenGlobalCheckpoints=1000
TimeBetweenEpochs=200
RedoBuffer=32M

# CompressedLCP=1
# CompressedBackup=1
# Enabling CompressedLCP and CompressedBackup causes, respectively, local
checkpoint files and backup files to be compressed, which can result in a space
savings of up to 50% over noncompressed LCPs and backups.

# MaxNoOfLocalScans=64
MaxNoOfTables=1024
MaxNoOfOrderedIndexes=256

[ndbd]
HostName=data-node-A-hostname
# NodeId=data-node-A-nodeid

LockExecuteThreadToCPU=1
LockMaintThreadsToCPU=0
# On systems with multiple CPUs, these parameters can be used to lock NDBCLUSTER
# threads to specific CPUs

[ndbd]
HostName=data-node-B-hostname
# NodeId=data-node-B-nodeid

LockExecuteThreadToCPU=1
LockMaintThreadsToCPU=0

# You must have an [ndbd] section for every data node in the cluster;
# each of these sections must include a HostName. Each section may
# optionally include a NodeId for convenience, but in most cases, it is
# sufficient to allow the cluster to allocate node IDs dynamically. If
# you do specify the node ID for a data node, it must be in the range 1
# to 144 inclusive and must be unique among all IDs specified for
# cluster nodes.

# SQL NODE / API NODE PARAMETERS

[mysqld]
# HostName=sql-node-A-hostname
# NodeId=sql-node-A-nodeid

[mysqld]

[mysqld]

# Each API or SQL node that connects to the cluster requires a [mysqld]
# or [api] section of its own. Each such section defines a connection
# “slot”; you should have at least as many of these sections in the
# config.ini file as the total number of API nodes and SQL nodes that
# you wish to have connected to the cluster at any given time. There is
# no performance or other penalty for having extra slots available in
# case you find later that you want or need more API or SQL nodes to
# connect to the cluster at the same time.
# If no HostName is specified for a given [mysqld] or [api] section,
# then any API or SQL node may use that slot to connect to the
# cluster. You may wish to use an explicit HostName for one connection slot
```

```
# to guarantee that an API or SQL node from that host can always
# connect to the cluster. If you wish to prevent API or SQL nodes from
# connecting from other than a desired host or hosts, then use a
# HostName for every [mysqld] or [api] section in the config.ini file.
# You can if you wish define a node ID (NodeId parameter) for any API or
# SQL node, but this is not necessary; if you do so, it must be in the
# range 1 to 255 inclusive and must be unique among all IDs specified
# for cluster nodes.
```

Required my.cnf options for SQL nodes. MySQL servers acting as NDB Cluster SQL nodes must always be started with the `--ndbcluster` and `--ndb-connectstring` options, either on the command line or in `my.cnf`.

23.4.3.3 NDB Cluster Connection Strings

With the exception of the NDB Cluster management server (`ndb_mgmd`), each node that is part of an NDB Cluster requires a *connection string* that points to the management server's location. This connection string is used in establishing a connection to the management server as well as in performing other tasks depending on the node's role in the cluster. The syntax for a connection string is as follows:

```
[nodeid=node_id, ]host-definition[, host-definition[, ...]]

host-definition:
    host_name[:port_number]
```

`node_id` is an integer greater than or equal to 1 which identifies a node in `config.ini`. *`host_name`* is a string representing a valid Internet host name or IP address. *`port_number`* is an integer referring to a TCP/IP port number.

```
example 1 (long):    "nodeid=2,myhost1:1100,myhost2:1100,198.51.100.3:1200"
example 2 (short):   "myhost1"
```

`localhost:1186` is used as the default connection string value if none is provided. If *`port_num`* is omitted from the connection string, the default port is 1186. This port should always be available on the network because it has been assigned by IANA for this purpose (see http://www.iana.org/assignments/port-numbers for details).

By listing multiple host definitions, it is possible to designate several redundant management servers. An NDB Cluster data or API node attempts to contact successive management servers on each host in the order specified, until a successful connection has been established.

It is also possible to specify in a connection string one or more bind addresses to be used by nodes having multiple network interfaces for connecting to management servers. A bind address consists of a hostname or network address and an optional port number. This enhanced syntax for connection strings is shown here:

```
[nodeid=node_id, ]
    [bind-address=host-definition, ]
    host-definition[; bind-address=host-definition]
    host-definition[; bind-address=host-definition]
    [, ...]]

host-definition:
    host_name[:port_number]
```

If a single bind address is used in the connection string *prior* to specifying any management hosts, then this address is used as the default for connecting to any of them (unless overridden for a given management server; see later in this section for an example). For example, the following connection string causes the node to use `198.51.100.242` regardless of the management server to which it connects:

```
bind-address=198.51.100.242, poseidon:1186, perch:1186
```

If a bind address is specified *following* a management host definition, then it is used only for connecting to that management node. Consider the following connection string:

```
poseidon:1186;bind-address=localhost, perch:1186;bind-address=198.51.100.242
```

In this case, the node uses `localhost` to connect to the management server running on the host named `poseidon` and `198.51.100.242` to connect to the management server running on the host named `perch`.

You can specify a default bind address and then override this default for one or more specific management hosts. In the following example, `localhost` is used for connecting to the management server running on host `poseidon`; since `198.51.100.242` is specified first (before any management server definitions), it is the default bind address and so is used for connecting to the management servers on hosts `perch` and `orca`:

```
bind-address=198.51.100.242,poseidon:1186;bind-address=localhost,perch:1186,orca:2200
```

There are a number of different ways to specify the connection string:

- Each executable has its own command-line option which enables specifying the management server at startup. (See the documentation for the respective executable.)
- It is also possible to set the connection string for all nodes in the cluster at once by placing it in a `[mysql_cluster]` section in the management server's `my.cnf` file.
- For backward compatibility, two other options are available, using the same syntax:

  1. Set the `NDB_CONNECTSTRING` environment variable to contain the connection string.
  2. Write the connection string for each executable into a text file named `Ndb.cfg` and place this file in the executable's startup directory.

  However, these are now deprecated and should not be used for new installations.

The recommended method for specifying the connection string is to set it on the command line or in the `my.cnf` file for each executable.

23.4.3.4 Defining Computers in an NDB Cluster

The `[computer]` section has no real significance other than serving as a way to avoid the need of defining host names for each node in the system. All parameters mentioned here are required.

- `Id`

Version (or later)	NDB 8.0.13
Type or units	string
Default	[...]
Range	...
Restart Type	**Initial System Restart:** Requires a complete shutdown of the cluster, wiping and restoring the cluster file system from a backup, and then restarting the cluster. (NDB 8.0.13)

This is a unique identifier, used to refer to the host computer elsewhere in the configuration file.

Important

The computer ID is *not* the same as the node ID used for a management, API, or data node. Unlike the case with node IDs, you cannot use `NodeId` in place of `Id` in the `[computer]` section of the `config.ini` file.

- `HostName`

Version (or later)	NDB 8.0.13
Type or units	name or IP address
Default	[...]
Range	...
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

This is the computer's hostname or IP address.

Restart types. Information about the restart types used by the parameter descriptions in this section is shown in the following table:

Table 23.8 NDB Cluster restart types

Symbol	Restart Type	Description
N	Node	The parameter can be updated using a rolling restart (see Section 23.6.5, "Performing a Rolling Restart of an NDB Cluster")
S	System	All cluster nodes must be shut down completely, then restarted, to effect a change in this parameter
I	Initial	Data nodes must be restarted using the `--initial` option

23.4.3.5 Defining an NDB Cluster Management Server

The `[ndb_mgmd]` section is used to configure the behavior of the management server. If multiple management servers are employed, you can specify parameters common to all of them in an `[ndb_mgmd default]` section. `[mgm]` and `[mgm default]` are older aliases for these, supported for backward compatibility.

All parameters in the following list are optional and assume their default values if omitted.

Note

If neither the `ExecuteOnComputer` nor the `HostName` parameter is present, the default value `localhost` is assumed for both.

- `Id`

Version (or later)	NDB 8.0.13
Type or units	unsigned
Default	[...]
Range	1 - 255
Restart Type	**Initial System Restart:** Requires a complete shutdown of the cluster, wiping and restoring the cluster file system from a backup, and then restarting the cluster. (NDB 8.0.13)

Each node in the cluster has a unique identity. For a management node, this is represented by an integer value in the range 1 to 255, inclusive. This ID is used by all internal cluster messages for addressing the node, and so must be unique for each NDB Cluster node, regardless of the type of node.

Note

Data node IDs must be less than 145. If you plan to deploy a large number of data nodes, it is a good idea to limit the node IDs for management nodes (and API nodes) to values greater than 144.

The use of the `Id` parameter for identifying management nodes is deprecated in favor of `NodeId`. Although `Id` continues to be supported for backward compatibility, it now generates a warning and is subject to removal in a future version of NDB Cluster.

- `NodeId`

Version (or later)	NDB 8.0.13
Type or units	unsigned
Default	[...]
Range	1 - 255
Restart Type	**Initial System Restart:** Requires a complete shutdown of the cluster, wiping and restoring the cluster file system from a backup, and then restarting

	the cluster. (NDB 8.0.13)

Each node in the cluster has a unique identity. For a management node, this is represented by an integer value in the range 1 to 255 inclusive. This ID is used by all internal cluster messages for addressing the node, and so must be unique for each NDB Cluster node, regardless of the type of node.

Note

Data node IDs must be less than 145. If you plan to deploy a large number of data nodes, it is a good idea to limit the node IDs for management nodes (and API nodes) to values greater than 144.

`NodeId` is the preferred parameter name to use when identifying management nodes. Although the older `Id` continues to be supported for backward compatibility, it is now deprecated and generates a warning when used; it is also subject to removal in a future NDB Cluster release.

- `ExecuteOnComputer`

Version (or later)	NDB 8.0.13
Type or units	name
Default	[...]
Range	...
Deprecated	Yes (in NDB 7.5)
Restart Type	**System Restart:** Requires a complete shutdown and restart of the cluster. (NDB 8.0.13)

This refers to the `Id` set for one of the computers defined in a `[computer]` section of the `config.ini` file.

Important

This parameter is deprecated, and is subject to removal in a future release. Use the `HostName` parameter instead.

- `PortNumber`

Version (or later)	NDB 8.0.13
Type or units	unsigned
Default	1186
Range	0 - 64K
Restart Type	**System Restart:** Requires a complete shutdown and

	restart of the cluster. (NDB 8.0.13)

This is the port number on which the management server listens for configuration requests and management commands.

-

The node ID for this node can be given out only to connections that explicitly request it. A management server that requests “any” node ID cannot use this one. This parameter can be used when running multiple management servers on the same host, and `HostName` is not sufficient for distinguishing among processes. Intended for use in testing.

- `HostName`

Version (or later)	NDB 8.0.13
Type or units	name or IP address
Default	[...]
Range	...
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

Specifying this parameter defines the hostname of the computer on which the management node is to reside. To specify a hostname other than `localhost`, either this parameter or `ExecuteOnComputer` is required.

- `LocationDomainId`

Version (or later)	NDB 8.0.13
Type or units	integer
Default	0
Range	0 - 16
Restart Type	**System Restart:** Requires a complete shutdown and restart of the cluster. (NDB 8.0.13)

Assigns a management node to a specific availability domain (also known as an availability zone) within a cloud. By informing `NDB` which nodes are in which availability domains, performance can be improved in a cloud environment in the following ways:

- If requested data is not found on the same node, reads can be directed to another node in the same availability domain.

- Communication between nodes in different availability domains are guaranteed to use `NDB` transporters' WAN support without any further manual intervention.
- The transporter's group number can be based on which availability domain is used, such that also SQL and other API nodes communicate with local data nodes in the same availability domain whenever possible.
- The arbitrator can be selected from an availability domain in which no data nodes are present, or, if no such availability domain can be found, from a third availability domain.

`LocationDomainId` takes an integer value between 0 and 16 inclusive, with 0 being the default; using 0 is the same as leaving the parameter unset.

- `LogDestination`

Version (or later)	NDB 8.0.13
Type or units	{CONSOLE|SYSLOG|FILE}
Default	FILE: filename=ndb_nodeid_cluster.log, maxsize=1000000, maxfiles=6
Range	...
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

This parameter specifies where to send cluster logging information. There are three options in this regard—`CONSOLE`, `SYSLOG`, and `FILE`—with `FILE` being the default:

- `CONSOLE` outputs the log to `stdout`:

```
CONSOLE
```

- `SYSLOG` sends the log to a `syslog` facility, possible values being one of `auth`, `authpriv`, `cron`, `daemon`, `ftp`, `kern`, `lpr`, `mail`, `news`, `syslog`, `user`, `uucp`, `local0`, `local1`, `local2`, `local3`, `local4`, `local5`, `local6`, or `local7`.

Note

Not every facility is necessarily supported by every operating system.

```
SYSLOG:facility=syslog
```

- `FILE` pipes the cluster log output to a regular file on the same machine. The following values can be specified:

  - `filename`: The name of the log file.

    The default log file name used in such cases is `ndb_nodeid_cluster.log`.

  - `maxsize`: The maximum size (in bytes) to which the file can grow before logging rolls over to a new file. When this occurs, the old log file is renamed by appending `.N` to the file name, where `N` is the next number not yet used with this name.

  - `maxfiles`: The maximum number of log files.

  ```
  FILE:filename=cluster.log,maxsize=1000000,maxfiles=6
  ```

  The default value for the `FILE` parameter is `FILE:filename=ndb_node_id_cluster.log,maxsize=1000000,maxfiles=6`, where `node_id` is the ID of the node.

It is possible to specify multiple log destinations separated by semicolons as shown here:

```
CONSOLE;SYSLOG:facility=local0;FILE:filename=/var/log/mgmd
```

- `ArbitrationRank`

Version (or later)	NDB 8.0.13
Type or units	0-2
Default	1
Range	0 - 2
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

This parameter is used to define which nodes can act as arbitrators. Only management nodes and SQL nodes can be arbitrators. `ArbitrationRank` can take one of the following values:

- `0`: The node is never used as an arbitrator.
- `1`: The node has high priority; that is, it is preferred as an arbitrator over low-priority nodes.
- `2`: Indicates a low-priority node which is used as an arbitrator only if a node with a higher priority is not available for that purpose.

Normally, the management server should be configured as an arbitrator by setting its `ArbitrationRank` to 1 (the default for management nodes) and those for all SQL nodes to 0 (the default for SQL nodes).

You can disable arbitration completely either by setting `ArbitrationRank` to 0 on all management and SQL nodes, or by setting the `Arbitration` parameter in the `[ndbd default]` section of the `config.ini` global configuration file. Setting `Arbitration` causes any settings for `ArbitrationRank` to be disregarded.

- `ArbitrationDelay`

Version (or later)	NDB 8.0.13
Type or units	milliseconds
Default	0
Range	0 - 4294967039 (0xFFFFFEFF)
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

An integer value which causes the management server's responses to arbitration requests to be delayed by that number of milliseconds. By default, this value is 0; it is normally not necessary to change it.

- `DataDir`

Version (or later)	NDB 8.0.13
Type or units	path
Default	.
Range	...
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

This specifies the directory where output files from the management server are placed. These files include cluster log files, process output files, and the daemon's process ID (PID) file. (For log files, this location can be overridden by setting the `FILE` parameter for `LogDestination`, as discussed previously in this section.)

The default value for this parameter is the directory in which `ndb_mgmd` is located.

- `PortNumberStats`

Version (or later)	NDB 8.0.13
Type or units	unsigned
Default	[...]
Range	0 - 64K
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

This parameter specifies the port number used to obtain statistical information from an NDB Cluster management server. It has no default value.

- `Wan`

Version (or later)	NDB 8.0.13
Type or units	boolean
Default	false
Range	true, false
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

Use WAN TCP setting as default.

- `HeartbeatThreadPriority`

Version (or later)	NDB 8.0.13
Type or units	string
Default	[...]
Range	...
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

Set the scheduling policy and priority of heartbeat threads for management and API nodes.

The syntax for setting this parameter is shown here:

```
HeartbeatThreadPriority = policy[, priority]

policy:
  {FIFO | RR}
```

When setting this parameter, you must specify a policy. This is one of `FIFO` (first in, first out) or `RR` (round robin). The policy value is followed optionally by the priority (an integer).

- `ExtraSendBufferMemory`

Version (or later)	NDB 8.0.13
Type or units	bytes
Default	0
Range	0 - 32G
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

This parameter specifies the amount of transporter send buffer memory to allocate in addition to any that has been set using `TotalSendBufferMemory`, `SendBufferMemory`, or both.

- `TotalSendBufferMemory`

Version (or later)	NDB 8.0.13
Type or units	bytes
Default	0
Range	256K - 4294967039 (0xFFFFFEFF)
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

This parameter is used to determine the total amount of memory to allocate on this node for shared send buffer memory among all configured transporters.

If this parameter is set, its minimum permitted value is 256KB; 0 indicates that the parameter has not been set. For more detailed information, see Section 23.4.3.14, “Configuring NDB Cluster Send Buffer Parameters”.

- `HeartbeatIntervalMgmdMgmd`

Version (or later)	NDB 8.0.13
Type or units	milliseconds
Default	1500
Range	100 - 4294967039 (0xFFFFFEFF)
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

Specify the interval between heartbeat messages used to determine whether another management node is on contact with this one. The management node waits after 3 of these intervals to declare the connection dead; thus, the default setting of 1500 milliseconds causes the management node to wait for approximately 1600 ms before timing out.

Note

After making changes in a management node's configuration, it is necessary to perform a rolling restart of the cluster for the new configuration to take effect.

To add new management servers to a running NDB Cluster, it is also necessary to perform a rolling restart of all cluster nodes after modifying any existing `config.ini` files. For more information about issues arising when using multiple management nodes, see Section 23.2.7.10, “Limitations Relating to Multiple NDB Cluster Nodes”.

Restart types. Information about the restart types used by the parameter descriptions in this section is shown in the following table:

Table 23.9 NDB Cluster restart types

Symbol	Restart Type	Description
N	Node	The parameter can be updated using a rolling restart (see Section 23.6.5, "Performing a Rolling Restart of an NDB Cluster")
S	System	All cluster nodes must be shut down completely, then restarted, to effect a change in this parameter
I	Initial	Data nodes must be restarted using the `--initial` option

23.4.3.6 Defining NDB Cluster Data Nodes

The `[ndbd]` and `[ndbd default]` sections are used to configure the behavior of the cluster's data nodes.

`[ndbd]` and `[ndbd default]` are always used as the section names whether you are using `ndbd` or `ndbmtd` binaries for the data node processes.

There are many parameters which control buffer sizes, pool sizes, timeouts, and so forth. The only mandatory parameter is either one of `ExecuteOnComputer` or `HostName`; this must be defined in the local `[ndbd]` section.

The parameter `NoOfReplicas` should be defined in the `[ndbd default]` section, as it is common to all Cluster data nodes. It is not strictly necessary to set `NoOfReplicas`, but it is good practice to set it explicitly.

Most data node parameters are set in the `[ndbd default]` section. Only those parameters explicitly stated as being able to set local values are permitted to be changed in the `[ndbd]` section. Where present, `HostName`, `NodeId` and `ExecuteOnComputer` *must* be defined in the local `[ndbd]` section, and not in any other section of `config.ini`. In other words, settings for these parameters are specific to one data node.

For those parameters affecting memory usage or buffer sizes, it is possible to use `K`, `M`, or `G` as a suffix to indicate units of 1024, 1024×1024, or 1024×1024×1024. (For example, `100K` means 100 × 1024 = 102400.)

Parameter names and values are case-insensitive, unless used in a MySQL Server `my.cnf` or `my.ini` file, in which case they are case-sensitive.

Information about configuration parameters specific to NDB Cluster Disk Data tables can be found later in this section (see Disk Data Configuration Parameters).

All of these parameters also apply to `ndbmtd` (the multithreaded version of `ndbd`). Three additional data node configuration parameters—`MaxNoOfExecutionThreads`, `ThreadConfig`, and `NoOfFragmentLogParts`—apply to `ndbmtd` only; these have no effect when used with `ndbd`. For more information, see Multi-Threading Configuration Parameters (ndbmtd). See also Section 23.5.3, "ndbmtd — The NDB Cluster Data Node Daemon (Multi-Threaded)".

Identifying data nodes. The `NodeId` or `Id` value (that is, the data node identifier) can be allocated on the command line when the node is started or in the configuration file.

- `NodeId`

Version (or later)	NDB 8.0.13

Type or units	unsigned
Default	[...]
Range	1 - 48
Version (or later)	NDB 8.0.18
Type or units	unsigned
Default	[...]
Range	1 - 144
Restart Type	**Initial System Restart:** Requires a complete shutdown of the cluster, wiping and restoring the cluster file system from a backup, and then restarting the cluster. (NDB 8.0.13)

A unique node ID is used as the node's address for all cluster internal messages. For data nodes, this is an integer in the range 1 to 144 inclusive. Each node in the cluster must have a unique identifier.

`NodeId` is the only supported parameter name to use when identifying data nodes.

- `ExecuteOnComputer`

Version (or later)	NDB 8.0.13
Type or units	name
Default	[...]
Range	...
Deprecated	Yes (in NDB 7.5)
Restart Type	**System Restart:** Requires a complete shutdown and restart of the cluster. (NDB 8.0.13)

This refers to the `Id` set for one of the computers defined in a `[computer]` section.

Important

This parameter is deprecated, and is subject to removal in a future release. Use the `HostName` parameter instead.

-

  The node ID for this node can be given out only to connections that explicitly request it. A management server that requests “any” node ID cannot use this one. This parameter can be used when running multiple management servers on the same host, and `HostName` is not sufficient for distinguishing among processes. Intended for use in testing.

- `HostName`

Version (or later)	NDB 8.0.13
Type or units	name or IP address
Default	localhost
Range	...
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

  Specifying this parameter defines the hostname of the computer on which the data node is to reside. To specify a hostname other than `localhost`, either this parameter or `ExecuteOnComputer` is required.

- `ServerPort`

Version (or later)	NDB 8.0.13
Type or units	unsigned
Default	[...]
Range	1 - 64K
Restart Type	**System Restart:** Requires a complete shutdown and restart of the cluster. (NDB 8.0.13)

  Each node in the cluster uses a port to connect to other nodes. By default, this port is allocated dynamically in such a way as to ensure that no two nodes on the same host computer receive the same port number, so it should normally not be necessary to specify a value for this parameter.

  However, if you need to be able to open specific ports in a firewall to permit communication between data nodes and API nodes (including SQL nodes), you can set this parameter to the number of the desired port in an `[ndbd]` section or (if you need to do this for multiple data nodes) the `[ndbd default]` section of the `config.ini` file, and then open the port having that number for incoming connections from SQL nodes, API nodes, or both.

  Note

  Connections from data nodes to management nodes is done using the `ndb_mgmd` management port (the management server's `PortNumber`) so

outgoing connections to that port from any data nodes should always be permitted.

- `TcpBind_INADDR_ANY`

  Setting this parameter to `TRUE` or `1` binds `IP_ADDR_ANY` so that connections can be made from anywhere (for autogenerated connections). The default is `FALSE` (`0`).

- `NodeGroup`

Version (or later)	NDB 8.0.13
Type or units	unsigned
Default	[...]
Range	0 - 65536
Restart Type	**Initial System Restart:** Requires a complete shutdown of the cluster, wiping and restoring the cluster file system from a backup, and then restarting the cluster. (NDB 8.0.13)

  This parameter can be used to assign a data node to a specific node group. It is read only when the cluster is started for the first time, and cannot be used to reassign a data node to a different node group online. It is generally not desirable to use this parameter in the `[ndbd default]` section of the `config.ini` file, and care must be taken not to assign nodes to node groups in such a way that an invalid numbers of nodes are assigned to any node groups.

  The `NodeGroup` parameter is chiefly intended for use in adding a new node group to a running NDB Cluster without having to perform a rolling restart. For this purpose, you should set it to 65536 (the maximum value). You are not required to set a `NodeGroup` value for all cluster data nodes, only for those nodes which are to be started and added to the cluster as a new node group at a later time. For more information, see Section 23.6.7.3, “Adding NDB Cluster Data Nodes Online: Detailed Example”.

- `LocationDomainId`

Version (or later)	NDB 8.0.13
Type or units	integer
Default	0
Range	0 - 16
Restart Type	**System Restart:** Requires a complete shutdown and restart of the

	cluster. (NDB 8.0.13)

Assigns a data node to a specific availability domain (also known as an availability zone) within a cloud. By informing `NDB` which nodes are in which availability domains, performance can be improved in a cloud environment in the following ways:

- If requested data is not found on the same node, reads can be directed to another node in the same availability domain.
- Communication between nodes in different availability domains are guaranteed to use `NDB` transporters' WAN support without any further manual intervention.
- The transporter's group number can be based on which availability domain is used, such that also SQL and other API nodes communicate with local data nodes in the same availability domain whenever possible.
- The arbitrator can be selected from an availability domain in which no data nodes are present, or, if no such availability domain can be found, from a third availability domain.

`LocationDomainId` takes an integer value between 0 and 16 inclusive, with 0 being the default; using 0 is the same as leaving the parameter unset.

- `NoOfReplicas`

Version (or later)	NDB 8.0.13
Type or units	integer
Default	2
Range	1 - 2
Version (or later)	NDB 8.0.19
Type or units	integer
Default	2
Range	1 - 4
Restart Type	**Initial System Restart:** Requires a complete shutdown of the cluster, wiping and restoring the cluster file system from a backup, and then restarting the cluster. (NDB 8.0.13)

This global parameter can be set only in the `[ndbd default]` section, and defines the number of fragment replicas for each table stored in the cluster. This parameter also specifies the size of node groups. A node group is a set of nodes all storing the same information.

Node groups are formed implicitly. The first node group is formed by the set of data nodes with the lowest node IDs, the next node group by the set of the next lowest node identities, and so on. By way of example, assume that we have 4 data nodes and that `NoOfReplicas` is set to 2. The four data

nodes have node IDs 2, 3, 4 and 5. Then the first node group is formed from nodes 2 and 3, and the second node group by nodes 4 and 5. It is important to configure the cluster in such a manner that nodes in the same node groups are not placed on the same computer because a single hardware failure would cause the entire cluster to fail.

If no node IDs are provided, the order of the data nodes is the determining factor for the node group. Whether or not explicit assignments are made, they can be viewed in the output of the management client's `SHOW` command.

The default value for `NoOfReplicas` is 2. This is the recommended value for most production environments. In NDB 8.0, setting this parameter's value to 3 or 4 is fully tested and supported in production.

Warning

Setting `NoOfReplicas` to 1 means that there is only a single copy of all Cluster data; in this case, the loss of a single data node causes the cluster to fail because there are no additional copies of the data stored by that node.

The number of data nodes in the cluster must be evenly divisible by the value of this parameter. For example, if there are two data nodes, then `NoOfReplicas` must be equal to either 1 or 2, since 2/3 and 2/4 both yield fractional values; if there are four data nodes, then `NoOfReplicas` must be equal to 1, 2, or 4.

- `DataDir`

Version (or later)	NDB 8.0.13
Type or units	path
Default	.
Range	...
Restart Type	**Initial Node Restart:** Requires a rolling restart of the cluster; each data node must be restarted with `--initial`. (NDB 8.0.13)

This parameter specifies the directory where trace files, log files, pid files and error logs are placed.

The default is the data node process working directory.

- `FileSystemPath`

Version (or later)	NDB 8.0.13
Type or units	path
Default	DataDir
Range	...
Restart Type	**Initial Node Restart:** Requires a

	rolling restart of the cluster; each data node must be restarted with `--initial`. (NDB 8.0.13)

This parameter specifies the directory where all files created for metadata, REDO logs, UNDO logs (for Disk Data tables), and data files are placed. The default is the directory specified by `DataDir`.

Note

This directory must exist before the `ndbd` process is initiated.

The recommended directory hierarchy for NDB Cluster includes `/var/lib/mysql-cluster`, under which a directory for the node's file system is created. The name of this subdirectory contains the node ID. For example, if the node ID is 2, this subdirectory is named `ndb_2_fs`.

- `BackupDataDir`

Version (or later)	NDB 8.0.13
Type or units	path
Default	FileSystemPath
Range	...
Restart Type	**Initial Node Restart:** Requires a rolling restart of the cluster; each data node must be restarted with `--initial`. (NDB 8.0.13)

This parameter specifies the directory in which backups are placed.

Important

The string '`/BACKUP`' is always appended to this value. For example, if you set the value of `BackupDataDir` to `/var/lib/cluster-data`, then all backups are stored under `/var/lib/cluster-data/BACKUP`. This also means that the *effective* default backup location is the directory named `BACKUP` under the location specified by the `FileSystemPath` parameter.

Data Memory, Index Memory, and String Memory

`DataMemory` and `IndexMemory` are `[ndbd]` parameters specifying the size of memory segments used to store the actual records and their indexes. In setting values for these, it is important to understand how `DataMemory` is used, as it usually needs to be updated to reflect actual usage by the cluster.

Note

`IndexMemory` is deprecated, and subject to removal in a future version of NDB Cluster. See the descriptions that follow for further information.

- `DataMemory`

Version (or later)	NDB 8.0.13
Type or units	bytes
Default	98M
Range	1M - 1T
Version (or later)	NDB 8.0.19
Type or units	bytes
Default	98M
Range	1M - 16T
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

This parameter defines the amount of space (in bytes) available for storing database records. The entire amount specified by this value is allocated in memory, so it is extremely important that the machine has sufficient physical memory to accommodate it.

The memory allocated by `DataMemory` is used to store both the actual records and indexes. There is a 16-byte overhead on each record; an additional amount for each record is incurred because it is stored in a 32KB page with 128 byte page overhead (see below). There is also a small amount wasted per page due to the fact that each record is stored in only one page.

For variable-size table attributes, the data is stored on separate data pages, allocated from `DataMemory`. Variable-length records use a fixed-size part with an extra overhead of 4 bytes to reference the variable-size part. The variable-size part has 2 bytes overhead plus 2 bytes per attribute.

In NDB 8.0, the maximum record size is 30000 bytes.

Resources assigned to `DataMemory` are used for storing all data and indexes. (Any memory configured as `IndexMemory` is automatically added to that used by `DataMemory` to form a common resource pool.)

The memory space allocated by `DataMemory` consists of 32KB pages, which are allocated to table fragments. Each table is normally partitioned into the same number of fragments as there are data nodes in the cluster. Thus, for each node, there are the same number of fragments as are set in `NoOfReplicas`.

Once a page has been allocated, it is currently not possible to return it to the pool of free pages, except by deleting the table. (This also means that `DataMemory` pages, once allocated to a given table, cannot be used by other tables.) Performing a data node recovery also compresses the partition because all records are inserted into empty partitions from other live nodes.

The `DataMemory` memory space also contains UNDO information: For each update, a copy of the unaltered record is allocated in the `DataMemory`. There is also a reference to each copy in the ordered table indexes. Unique hash indexes are updated only when the unique index columns are updated, in which case a new entry in the index table is inserted and the old entry is deleted upon commit. For this reason, it is also necessary to allocate enough memory to handle the largest

transactions performed by applications using the cluster. In any case, performing a few large transactions holds no advantage over using many smaller ones, for the following reasons:

- Large transactions are not any faster than smaller ones
- Large transactions increase the number of operations that are lost and must be repeated in event of transaction failure
- Large transactions use more memory

The default value for `DataMemory` in NDB 8.0 is 98MB. The minimum value is 1MB. There is no maximum size, but in reality the maximum size has to be adapted so that the process does not start swapping when the limit is reached. This limit is determined by the amount of physical RAM available on the machine and by the amount of memory that the operating system may commit to any one process. 32-bit operating systems are generally limited to 2−4GB per process; 64-bit operating systems can use more. For large databases, it may be preferable to use a 64-bit operating system for this reason.

- `IndexMemory`

Version (or later)	NDB 8.0.13
Type or units	bytes
Default	0
Range	1M - 1T
Deprecated	Yes (in NDB 7.6)
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

The `IndexMemory` parameter is deprecated (and subject to future removal); any memory assigned to `IndexMemory` is allocated instead to the same pool as `DataMemory`, which is solely responsible for all resources needed for storing data and indexes in memory. In NDB 8.0, the use of `IndexMemory` in the cluster configuration file triggers a warning from the management server.

You can estimate the size of a hash index using this formula:

```
size  = ( (fragments * 32K) + (rows * 18) )
        * fragment_replicas
```

`fragments` is the number of fragments, *`fragment_replicas`* is the number of fragment replicas (normally 2), and *`rows`* is the number of rows. If a table has one million rows, eight fragments, and two fragment replicas, the expected index memory usage is calculated as shown here:

```
((8 * 32K) + (1000000 * 18)) * 2 = ((8 * 32768) + (1000000 * 18)) * 2
= (262144 + 18000000) * 2
= 18262144 * 2 = 36524288 bytes = ~35MB
```

Index statistics for ordered indexes (when these are enabled) are stored in the `mysql.ndb_index_stat_sample` table. Since this table has a hash index, this adds to index memory usage. An upper bound to the number of rows for a given ordered index can be calculated as follows:

```
sample_size= key_size + ((key_attributes + 1) * 4)

sample_rows = IndexStatSaveSize
              * ((0.01 * IndexStatSaveScale * log2(rows * sample_size)) + 1)
```

```
                / sample_size
```

In the preceding formula, *`key_size`* is the size of the ordered index key in bytes, *`key_attributes`* is the number of attributes in the ordered index key, and *`rows`* is the number of rows in the base table.

Assume that table `t1` has 1 million rows and an ordered index named `ix1` on two four-byte integers. Assume in addition that `IndexStatSaveSize` and `IndexStatSaveScale` are set to their default values (32K and 100, respectively). Using the previous 2 formulas, we can calculate as follows:

```
  sample_size = 8  + ((1 + 2) * 4) = 20 bytes

  sample_rows = 32K
                * ((0.01 * 100 * log2(1000000*20)) + 1)
                / 20
                = 32768 * ( (1 * ~16.811) +1) / 20
                = 32768 * ~17.811 / 20
                = ~29182 rows
```

The expected index memory usage is thus 2 * 18 * 29182 = ~1050550 bytes.

In NDB 8.0, the minimum and default value for this parameter is 0 (zero).

- `StringMemory`

Version (or later)	NDB 8.0.13
Type or units	% or bytes
Default	25
Range	0 - 4294967039 (0xFFFFFEFF)
Restart Type	**System Restart:** Requires a complete shutdown and restart of the cluster. (NDB 8.0.13)

This parameter determines how much memory is allocated for strings such as table names, and is specified in an `[ndbd]` or `[ndbd default]` section of the `config.ini` file. A value between `0` and `100` inclusive is interpreted as a percent of the maximum default value, which is calculated based on a number of factors including the number of tables, maximum table name size, maximum size of `.FRM` files, `MaxNoOfTriggers`, maximum column name size, and maximum default column value.

A value greater than `100` is interpreted as a number of bytes.

The default value is 25—that is, 25 percent of the default maximum.

Under most circumstances, the default value should be sufficient, but when you have a great many `NDB` tables (1000 or more), it is possible to get Error 773 `Out of string memory, please modify StringMemory config parameter: Permanent error: Schema error`, in which case you should increase this value. `25` (25 percent) is not excessive, and should prevent this error from recurring in all but the most extreme conditions.

The following example illustrates how memory is used for a table. Consider this table definition:

```
CREATE TABLE example (
  a INT NOT NULL,
```

```
  b INT NOT NULL,
  c INT NOT NULL,
  PRIMARY KEY(a),
  UNIQUE(b)
) ENGINE=NDBCLUSTER;
```

For each record, there are 12 bytes of data plus 12 bytes overhead. Having no nullable columns saves 4 bytes of overhead. In addition, we have two ordered indexes on columns `a` and `b` consuming roughly 10 bytes each per record. There is a primary key hash index on the base table using roughly 29 bytes per record. The unique constraint is implemented by a separate table with `b` as primary key and `a` as a column. This other table consumes an additional 29 bytes of index memory per record in the `example` table as well 8 bytes of record data plus 12 bytes of overhead.

Thus, for one million records, we need 58MB for index memory to handle the hash indexes for the primary key and the unique constraint. We also need 64MB for the records of the base table and the unique index table, plus the two ordered index tables.

You can see that hash indexes takes up a fair amount of memory space; however, they provide very fast access to the data in return. They are also used in NDB Cluster to handle uniqueness constraints.

Currently, the only partitioning algorithm is hashing and ordered indexes are local to each node. Thus, ordered indexes cannot be used to handle uniqueness constraints in the general case.

An important point for both `IndexMemory` and `DataMemory` is that the total database size is the sum of all data memory and all index memory for each node group. Each node group is used to store replicated information, so if there are four nodes with two fragment replicas, there are two node groups. Thus, the total data memory available is 2 × `DataMemory` for each data node.

It is highly recommended that `DataMemory` and `IndexMemory` be set to the same values for all nodes. Data distribution is even over all nodes in the cluster, so the maximum amount of space available for any node can be no greater than that of the smallest node in the cluster.

`DataMemory` can be changed, but decreasing it can be risky; doing so can easily lead to a node or even an entire NDB Cluster that is unable to restart due to there being insufficient memory space. Increasing these values should be acceptable, but it is recommended that such upgrades are performed in the same manner as a software upgrade, beginning with an update of the configuration file, and then restarting the management server followed by restarting each data node in turn.

MinFreePct. A proportion (5% by default) of data node resources including `DataMemory` is kept in reserve to insure that the data node does not exhaust its memory when performing a restart. This can be adjusted using the `MinFreePct` data node configuration parameter (default 5).

Version (or later)	NDB 8.0.13
Type or units	unsigned
Default	5
Range	0 - 100
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

Updates do not increase the amount of index memory used. Inserts take effect immediately; however, rows are not actually deleted until the transaction is committed.

Transaction parameters. The next few `[ndbd]` parameters that we discuss are important because they affect the number of parallel transactions and the sizes of transactions that can be handled by the

system. MaxNoOfConcurrentTransactions sets the number of parallel transactions possible in a node. MaxNoOfConcurrentOperations sets the number of records that can be in update phase or locked simultaneously.

Both of these parameters (especially MaxNoOfConcurrentOperations) are likely targets for users setting specific values and not using the default value. The default value is set for systems using small transactions, to ensure that these do not use excessive memory.

MaxDMLOperationsPerTransaction sets the maximum number of DML operations that can be performed in a given transaction.

- MaxNoOfConcurrentTransactions

Version (or later)	NDB 8.0.13
Type or units	integer
Default	4096
Range	32 - 4294967039 (0xFFFFFEFF)
Deprecated	NDB 8.0.19
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

Each cluster data node requires a transaction record for each active transaction in the cluster. The task of coordinating transactions is distributed among all of the data nodes. The total number of transaction records in the cluster is the number of transactions in any given node times the number of nodes in the cluster.

Transaction records are allocated to individual MySQL servers. Each connection to a MySQL server requires at least one transaction record, plus an additional transaction object per table accessed by that connection. This means that a reasonable minimum for the total number of transactions in the cluster can be expressed as

```
TotalNoOfConcurrentTransactions =
    (maximum number of tables accessed in any single transaction + 1)
    * number of SQL nodes
```

Suppose that there are 10 SQL nodes using the cluster. A single join involving 10 tables requires 11 transaction records; if there are 10 such joins in a transaction, then 10 * 11 = 110 transaction records are required for this transaction, per MySQL server, or 110 * 10 = 1100 transaction records total. Each data node can be expected to handle TotalNoOfConcurrentTransactions / number of data nodes. For an NDB Cluster having 4 data nodes, this would mean setting MaxNoOfConcurrentTransactions on each data node to 1100 / 4 = 275. In addition, you should provide for failure recovery by ensuring that a single node group can accommodate all concurrent transactions; in other words, that each data node's MaxNoOfConcurrentTransactions is sufficient to cover a number of transactions equal to TotalNoOfConcurrentTransactions / number of node groups. If this cluster has a single node group, then MaxNoOfConcurrentTransactions should be set to 1100 (the same as the total number of concurrent transactions for the entire cluster).

In addition, each transaction involves at least one operation; for this reason, the value set for MaxNoOfConcurrentTransactions should always be no more than the value of MaxNoOfConcurrentOperations.

This parameter must be set to the same value for all cluster data nodes. This is due to the fact that, when a data node fails, the oldest surviving node re-creates the transaction state of all transactions that were ongoing in the failed node.

It is possible to change this value using a rolling restart, but the amount of traffic on the cluster must be such that no more transactions occur than the lower of the old and new levels while this is taking place.

The default value is 4096.

- `MaxNoOfConcurrentOperations`

Version (or later)	NDB 8.0.13
Type or units	integer
Default	32K
Range	32 - 4294967039 (0xFFFFFEFF)
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

It is a good idea to adjust the value of this parameter according to the size and number of transactions. When performing transactions which involve only a few operations and records, the default value for this parameter is usually sufficient. Performing large transactions involving many records usually requires that you increase its value.

Records are kept for each transaction updating cluster data, both in the transaction coordinator and in the nodes where the actual updates are performed. These records contain state information needed to find UNDO records for rollback, lock queues, and other purposes.

This parameter should be set at a minimum to the number of records to be updated simultaneously in transactions, divided by the number of cluster data nodes. For example, in a cluster which has four data nodes and which is expected to handle one million concurrent updates using transactions, you should set this value to 1000000 / 4 = 250000. To help provide resiliency against failures, it is suggested that you set this parameter to a value that is high enough to permit an individual data node to handle the load for its node group. In other words, you should set the value equal to `total number of concurrent operations / number of node groups`. (In the case where there

is a single node group, this is the same as the total number of concurrent operations for the entire cluster.)

Because each transaction always involves at least one operation, the value of `MaxNoOfConcurrentOperations` should always be greater than or equal to the value of `MaxNoOfConcurrentTransactions`.

Read queries which set locks also cause operation records to be created. Some extra space is allocated within individual nodes to accommodate cases where the distribution is not perfect over the nodes.

When queries make use of the unique hash index, there are actually two operation records used per record in the transaction. The first record represents the read in the index table and the second handles the operation on the base table.

The default value is 32768.

This parameter actually handles two values that can be configured separately. The first of these specifies how many operation records are to be placed with the transaction coordinator. The second part specifies how many operation records are to be local to the database.

A very large transaction performed on an eight-node cluster requires as many operation records in the transaction coordinator as there are reads, updates, and deletes involved in the transaction. However, the operation records of the are spread over all eight nodes. Thus, if it is necessary to configure the system for one very large transaction, it is a good idea to configure the two parts separately. `MaxNoOfConcurrentOperations` is always used to calculate the number of operation records in the transaction coordinator portion of the node.

It is also important to have an idea of the memory requirements for operation records. These consume about 1KB per record.

- `MaxNoOfLocalOperations`

Version (or later)	NDB 8.0.13
Type or units	integer
Default	UNDEFINED
Range	32 - 4294967039 (0xFFFFFEFF)
Deprecated	NDB 8.0.19
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

By default, this parameter is calculated as 1.1 × `MaxNoOfConcurrentOperations`. This fits systems with many simultaneous transactions, none of them being very large. If there is a need to handle one very large transaction at a time and there are many nodes, it is a good idea to override the default value by explicitly specifying this parameter.

This parameter is deprecated in NDB 8.0, and is subject to removal in a future NDB Cluster release. In addition, this parameter is incompatible with the `TransactionMemory` parameter; if you try to set values for both parameters in the cluster configuration file (`config.ini`), the management server refuses to start.

- `MaxDMLOperationsPerTransaction`

Version (or later)	NDB 8.0.13
Type or units	operations (DML)
Default	4294967295
Range	32 - 4294967295
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

This parameter limits the size of a transaction. The transaction is aborted if it requires more than this many DML operations.

The value of this parameter cannot exceed that set for `MaxNoOfConcurrentOperations`.

Transaction temporary storage. The next set of `[ndbd]` parameters is used to determine temporary storage when executing a statement that is part of a Cluster transaction. All records are released when the statement is completed and the cluster is waiting for the commit or rollback.

The default values for these parameters are adequate for most situations. However, users with a need to support transactions involving large numbers of rows or operations may need to increase these values to enable better parallelism in the system, whereas users whose applications require relatively small transactions can decrease the values to save memory.

- `MaxNoOfConcurrentIndexOperations`

Version (or later)	NDB 8.0.13
Type or units	integer
Default	8K
Range	0 - 4294967039 (0xFFFFFEFF)
Deprecated	NDB 8.0.19
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

For queries using a unique hash index, another temporary set of operation records is used during a query's execution phase. This parameter sets the size of that pool of records. Thus, this record is allocated only while executing a part of a query. As soon as this part has been executed, the record is released. The state needed to handle aborts and commits is handled by the normal operation records, where the pool size is set by the parameter `MaxNoOfConcurrentOperations`.

The default value of this parameter is 8192. Only in rare cases of extremely high parallelism using unique hash indexes should it be necessary to increase this value. Using a smaller value is possible and can save memory if the DBA is certain that a high degree of parallelism is not required for the cluster.

This parameter is deprecated in NDB 8.0, and is subject to removal in a future NDB Cluster release. In addition, this parameter is incompatible with the `TransactionMemory` parameter; if you try to set

values for both parameters in the cluster configuration file (`config.ini`), the management server refuses to start.

- `MaxNoOfFiredTriggers`

Version (or later)	NDB 8.0.13
Type or units	integer
Default	4000
Range	0 - 4294967039 (0xFFFFFEFF)
Deprecated	NDB 8.0.19
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

The default value of `MaxNoOfFiredTriggers` is 4000, which is sufficient for most situations. In some cases it can even be decreased if the DBA feels certain the need for parallelism in the cluster is not high.

A record is created when an operation is performed that affects a unique hash index. Inserting or deleting a record in a table with unique hash indexes or updating a column that is part of a unique hash index fires an insert or a delete in the index table. The resulting record is used to represent this index table operation while waiting for the original operation that fired it to complete. This operation is short-lived but can still require a large number of records in its pool for situations with many parallel write operations on a base table containing a set of unique hash indexes.

This parameter is deprecated in NDB 8.0, and is subject to removal in a future NDB Cluster release. In addition, this parameter is incompatible with the `TransactionMemory` parameter; if you try to set values for both parameters in the cluster configuration file (`config.ini`), the management server refuses to start.

- `TransactionBufferMemory`

Version (or later)	NDB 8.0.13
Type or units	bytes
Default	1M
Range	1K - 4294967039 (0xFFFFFEFF)
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

The memory affected by this parameter is used for tracking operations fired when updating index tables and reading unique indexes. This memory is used to store the key and column information for

these operations. It is only very rarely that the value for this parameter needs to be altered from the default.

The default value for `TransactionBufferMemory` is 1MB.

Normal read and write operations use a similar buffer, whose usage is even more short-lived. The compile-time parameter `ZATTRBUF_FILESIZE` (found in `ndb/src/kernel/blocks/Dbtc/Dbtc.hpp`) set to 4000 × 128 bytes (500KB). A similar buffer for key information, `ZDATABUF_FILESIZE` (also in `Dbtc.hpp`) contains 4000 × 16 = 62.5KB of buffer space. `Dbtc` is the module that handles transaction coordination.

Transaction resource allocation parameters. The parameters in the following list are used to allocate transaction resources in the transaction coordinator (`DBTC`). Leaving any one of these set to the default (0) dedicates transaction memory for 25% of estimated total data node usage for the corresponding resource. The actual maximum possible values for these parameters are typically limited by the amount of memory available to the data node; setting them has no impact on the total amount of memory allocated to the data node. In addition, you should keep in mind that they control numbers of reserved internal records for the data node independent of any settings for `MaxDMLOperationsPerTransaction`, `MaxNoOfConcurrentIndexOperations`, `MaxNoOfConcurrentOperations`, `MaxNoOfConcurrentScans`, `MaxNoOfConcurrentTransactions`, `MaxNoOfFiredTriggers`, `MaxNoOfLocalScans`, or `TransactionBufferMemory` (see Transaction parameters and Transaction temporary storage).

- `ReservedConcurrentIndexOperations`

Version (or later)	NDB 8.0.16
Type or units	numeric
Default	0
Range	0 - 4294967039 (0xFFFFFEFF)
Added	NDB 8.0.16
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

Number of simultaneous index operations having dedicated resources on one data node.

- `ReservedConcurrentOperations`

Version (or later)	NDB 8.0.16
Type or units	numeric
Default	0
Range	0 - 4294967039 (0xFFFFFEFF)
Added	NDB 8.0.16
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

Number of simultaneous operations having dedicated resources in transaction coordinators on one data node.

- `ReservedConcurrentScans`

Version (or later)	NDB 8.0.16
Type or units	numeric
Default	0
Range	0 - 4294967039 (0xFFFFFEFF)
Added	NDB 8.0.16
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

Number of simultaneous scans having dedicated resources on one data node.

- `ReservedConcurrentTransactions`

Version (or later)	NDB 8.0.16
Type or units	numeric
Default	0
Range	0 - 4294967039 (0xFFFFFEFF)
Added	NDB 8.0.16
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

Number of simultaneous transactions having dedicated resources on one data node.

- `ReservedFiredTriggers`

Version (or later)	NDB 8.0.16
Type or units	numeric
Default	0
Range	0 - 4294967039 (0xFFFFFEFF)
Added	NDB 8.0.16
Restart Type	**Node Restart:** Requires a rolling restart

	of the cluster. (NDB 8.0.13)

Number of triggers that have dedicated resources on one ndbd(DB) node.

- `ReservedLocalScans`

Version (or later)	NDB 8.0.16
Type or units	numeric
Default	0
Range	0 - 4294967039 (0xFFFFFEFF)
Added	NDB 8.0.16
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

Number of simultaneous fragment scans having dedicated resources on one data node.

- `ReservedTransactionBufferMemory`

Version (or later)	NDB 8.0.16
Type or units	numeric
Default	0
Range	0 - 4294967039 (0xFFFFFEFF)
Added	NDB 8.0.16
Deprecated	NDB 8.0.19
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

Dynamic buffer space (in bytes) for key and attribute data allocated to each data node.

- `TransactionMemory`

Version (or later)	NDB 8.0.19
Type or units	bytes
Default	0
Range	0 - 16384G
Added	NDB 8.0.19
Restart Type	**Node Restart:** Requires a rolling restart

	of the cluster. (NDB 8.0.13)

This parameter determines the memory (in bytes) allocated for transactions on each data node. Setting of transaction memory can be handled in any one of the three ways listed here:

- A number of configuration parameters are incompatible with `TransactionMemory`. If any of these are set, transaction memory is calculated as it was previous to NDB 8.0. You should be aware that it is not possible to set any of these parameters concurrently with `TransactionMemory`; if you attempt to do so, the management server is unable to start (see Parameters incompatible with TransactionMemory).
- If `TransactionMemory` is set, this value is used for determining transaction memory.
- If neither any incompatible parameters are set nor `TransactionMemory` is set, transaction memory is set by `NDB` to 10% of the value of the `DataMemory` configuration parameter.

Parameters incompatible with TransactionMemory. The following parameters cannot be used concurrently with `TransactionMemory` and are deprecated in NDB 8.0:

- `MaxNoOfConcurrentIndexOperations`
- `MaxNoOfFiredTriggers`
- `MaxNoOfLocalOperations`
- `MaxNoOfLocalScans`

Explicitly setting any of the parameters just listed when `TransactionMemory` has also been set in the cluster configuration file (`config.ini`) keeps the management node from starting.

For more information regarding resource allocation in NDB Cluster data nodes, see Section 23.4.3.13, “Data Node Memory Management”.

Scans and buffering. There are additional `[ndbd]` parameters in the `Dblqh` module (in `ndb/src/kernel/blocks/Dblqh/Dblqh.hpp`) that affect reads and updates. These include `ZATTRINBUF_FILESIZE`, set by default to 10000 × 128 bytes (1250KB) and `ZDATABUF_FILE_SIZE`, set by default to 10000*16 bytes (roughly 156KB) of buffer space. To date, there have been neither any reports from users nor any results from our own extensive tests suggesting that either of these compile-time limits should be increased.

- `BatchSizePerLocalScan`

Version (or later)	NDB 8.0.13
Type or units	integer
Default	256
Range	1 - 992
Deprecated	NDB 8.0.19
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

This parameter is used to calculate the number of lock records used to handle concurrent scan operations.

`BatchSizePerLocalScan` has a strong connection to the `BatchSize` defined in the SQL nodes.

Deprecated in NDB 8.0.

- `LongMessageBuffer`

Version (or later)	NDB 8.0.13
Type or units	bytes
Default	64M
Range	512K - 4294967039 (0xFFFFFEFF)
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

This is an internal buffer used for passing messages within individual nodes and between nodes. The default is 64MB.

This parameter seldom needs to be changed from the default.

- `MaxFKBuildBatchSize`

Version (or later)	NDB 8.0.13
Type or units	integer
Default	64
Range	16 - 512
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

Maximum scan batch size used for building foreign keys. Increasing the value set for this parameter may speed up building of foreign key builds at the expense of greater impact to ongoing traffic.

- `MaxNoOfConcurrentScans`

Version (or later)	NDB 8.0.13
Type or units	integer
Default	256
Range	2 - 500
Restart Type	**Node Restart:** Requires a rolling restart

	of the cluster. (NDB 8.0.13)

This parameter is used to control the number of parallel scans that can be performed in the cluster. Each transaction coordinator can handle the number of parallel scans defined for this parameter. Each scan query is performed by scanning all partitions in parallel. Each partition scan uses a scan record in the node where the partition is located, the number of records being the value of this parameter times the number of nodes. The cluster should be able to sustain `MaxNoOfConcurrentScans` scans concurrently from all nodes in the cluster.

Scans are actually performed in two cases. The first of these cases occurs when no hash or ordered indexes exists to handle the query, in which case the query is executed by performing a full table scan. The second case is encountered when there is no hash index to support the query but there is an ordered index. Using the ordered index means executing a parallel range scan. The order is kept on the local partitions only, so it is necessary to perform the index scan on all partitions.

The default value of `MaxNoOfConcurrentScans` is 256. The maximum value is 500.

- `MaxNoOfLocalScans`

Version (or later)	NDB 8.0.13
Type or units	integer
Default	4 * MaxNoOfConcurrentScans * [# of data nodes] + 2
Range	32 - 4294967039 (0xFFFFFEFF)
Deprecated	NDB 8.0.19
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

Specifies the number of local scan records if many scans are not fully parallelized. When the number of local scan records is not provided, it is calculated as shown here:

```
4 * MaxNoOfConcurrentScans * [# data nodes] + 2
```

This parameter is deprecated in NDB 8.0, and is subject to removal in a future NDB Cluster release. In addition, this parameter is incompatible with the `TransactionMemory` parameter; if you try to set values for both parameters in the cluster configuration file (`config.ini`), the management server refuses to start.

- `MaxParallelCopyInstances`

Version (or later)	NDB 8.0.13
Type or units	integer
Default	0
Range	0 - 64
Restart Type	**Node Restart:** Requires a

	rolling restart of the cluster. (NDB 8.0.13)

This parameter sets the parallelization used in the copy phase of a node restart or system restart, when a node that is currently just starting is synchronised with a node that already has current data by copying over any changed records from the node that is up to date. Because full parallelism in such cases can lead to overload situations, `MaxParallelCopyInstances` provides a means to decrease it. This parameter's default value 0. This value means that the effective parallelism is equal to the number of LDM instances in the node just starting as well as the node updating it.

- `MaxParallelScansPerFragment`

Version (or later)	NDB 8.0.13
Type or units	bytes
Default	256
Range	1 - 4294967039 (0xFFFFFEFF)
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

It is possible to configure the maximum number of parallel scans (`TUP` scans and `TUX` scans) allowed before they begin queuing for serial handling. You can increase this to take advantage of any unused CPU when performing large number of scans in parallel and improve their performance.

- `MaxReorgBuildBatchSize`

Version (or later)	NDB 8.0.13
Type or units	integer
Default	64
Range	16 - 512
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

Maximum scan batch size used for reorganization of table partitions. Increasing the value set for this parameter may speed up reorganization at the expense of greater impact to ongoing traffic.

- `MaxUIBuildBatchSize`

Version (or later)	NDB 8.0.13
Type or units	integer
Default	64
Range	16 - 512
Restart Type	**Node Restart:**
	Requires a

	rolling restart of the cluster. (NDB 8.0.13)

Maximum scan batch size used for building unique keys. Increasing the value set for this parameter may speed up such builds at the expense of greater impact to ongoing traffic.

Memory Allocation

`MaxAllocate`

Version (or later)	NDB 8.0.13
Type or units	unsigned
Default	32M
Range	1M - 1G
Deprecated	NDB 8.0.27
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

This parameter was used in older versions of NDB Cluster, but has no effect in NDB 8.0. It is deprecated as of NDB 8.0.27, and subject to removal in a future release.

Multiple Transporters

Beginning with version 8.0.20, `NDB` allocates multiple transporters for communication between pairs of data nodes. The number of transporters so allocated can be influenced by setting an appropriate value for the `NodeGroupTransporters` parameter introduced in that release.

`NodeGroupTransporters`

Version (or later)	NDB 8.0.20
Type or units	integer
Default	0
Range	0 - 32
Added	NDB 8.0.20
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

This parameter determines the number of transporters used between nodes in the same node group. The default value (0) means that the number of transporters used is the same as the number of LDMs in the node. This should be sufficient for most use cases; thus it should seldom be necessary to change this value from its default.

Setting `NodeGroupTransporters` to a number greater than the number of LDM threads or the number of TC threads, whichever is higher, causes `NDB` to use the maximum of these two numbers of threads. This means that a value greater than this is effectively ignored.

Hash Map Size

`DefaultHashMapSize`

Version (or later)	NDB 8.0.13

Type or units	LDM threads
Default	240
Range	0 - 3840
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

The original intended use for this parameter was to facilitate upgrades and especially downgrades to and from very old releases with differing default hash map sizes. This is not an issue when upgrading from NDB Cluster 7.3 (or later) to later versions.

Decreasing this parameter online after any tables have been created or modified with `DefaultHashMapSize` equal to 3840 is not currently supported.

Logging and checkpointing. The following `[ndbd]` parameters control log and checkpoint behavior.

- `FragmentLogFileSize`

Version (or later)	NDB 8.0.13
Type or units	bytes
Default	16M
Range	4M - 1G
Restart Type	**Initial Node Restart:** Requires a rolling restart of the cluster; each data node must be restarted with `--initial`. (NDB 8.0.13)

Setting this parameter enables you to control directly the size of redo log files. This can be useful in situations when NDB Cluster is operating under a high load and it is unable to close fragment log files quickly enough before attempting to open new ones (only 2 fragment log files can be open at one time); increasing the size of the fragment log files gives the cluster more time before having to open each new fragment log file. The default value for this parameter is 16M.

For more information about fragment log files, see the description for `NoOfFragmentLogFiles`.

- `InitialNoOfOpenFiles`

Version (or later)	NDB 8.0.13
Type or units	files
Default	27
Range	20 - 4294967039 (0xFFFFFEFF)

Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

This parameter sets the initial number of internal threads to allocate for open files.

The default value is 27.

- `InitFragmentLogFiles`

Version (or later)	NDB 8.0.13
Type or units	[see values]
Default	SPARSE
Range	SPARSE, FULL
Restart Type	**Initial Node Restart:** Requires a rolling restart of the cluster; each data node must be restarted with `--initial`. (NDB 8.0.13)

By default, fragment log files are created sparsely when performing an initial start of a data node—that is, depending on the operating system and file system in use, not all bytes are necessarily written to disk. However, it is possible to override this behavior and force all bytes to be written, regardless of the platform and file system type being used, by means of this parameter. `InitFragmentLogFiles` takes either of two values:

- `SPARSE`. Fragment log files are created sparsely. This is the default value.
- `FULL`. Force all bytes of the fragment log file to be written to disk.

Depending on your operating system and file system, setting `InitFragmentLogFiles=FULL` may help eliminate I/O errors on writes to the redo log.

- `EnablePartialLcp`

Version (or later)	NDB 8.0.13
Type or units	boolean
Default	true
Range	...
Restart Type	**Node Restart:** Requires a rolling restart

	of the cluster. (NDB 8.0.13)

When `true`, enable partial local checkpoints: This means that each LCP records only part of the full database, plus any records containing rows changed since the last LCP; if no rows have changed, the LCP updates only the LCP control file and does not update any data files.

If `EnablePartialLcp` is disabled (`false`), each LCP uses only a single file and writes a full checkpoint; this requires the least amount of disk space for LCPs, but increases the write load for each LCP. The default value is enabled (`true`). The proportion of space used by partial LCPS can be modified by the setting for the `RecoveryWork` configuration parameter.

For more information about files and directories used for full and partial LCPs, see NDB Cluster Data Node File System Directory.

Setting this parameter to `false` also disables the calculation of disk write speed used by the adaptive LCP control mechanism.

- `LcpScanProgressTimeout`

Version (or later)	NDB 8.0.13
Type or units	second
Default	60
Range	0 - 4294967039 (0xFFFFFEFF)
Version (or later)	NDB 8.0.19
Type or units	second
Default	180
Range	0 - 4294967039 (0xFFFFFEFF)
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

A local checkpoint fragment scan watchdog checks periodically for no progress in each fragment scan performed as part of a local checkpoint, and shuts down the node if there is no progress after a given amount of time has elapsed. This interval can be set using the `LcpScanProgressTimeout` data node configuration parameter, which sets the maximum time for which the local checkpoint can be stalled before the LCP fragment scan watchdog shuts down the node.

The default value is 60 seconds (providing compatibility with previous releases). Setting this parameter to 0 disables the LCP fragment scan watchdog altogether.

- `MaxNoOfOpenFiles`

Version (or later)	NDB 8.0.13
Type or units	unsigned
Default	0

Range	20 - 4294967039 (0xFFFFFEFF)
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

This parameter sets a ceiling on how many internal threads to allocate for open files. *Any situation requiring a change in this parameter should be reported as a bug*.

The default value is 0. However, the minimum value to which this parameter can be set is 20.

- `MaxNoOfSavedMessages`

Version (or later)	NDB 8.0.13
Type or units	integer
Default	25
Range	0 - 4294967039 (0xFFFFFEFF)
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

This parameter sets the maximum number of errors written in the error log as well as the maximum number of trace files that are kept before overwriting the existing ones. Trace files are generated when, for whatever reason, the node crashes.

The default is 25, which sets these maximums to 25 error messages and 25 trace files.

- `MaxLCPStartDelay`

Version (or later)	NDB 8.0.13
Type or units	seconds
Default	0
Range	0 - 600
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

In parallel data node recovery, only table data is actually copied and synchronized in parallel; synchronization of metadata such as dictionary and checkpoint information is done in a serial fashion. In addition, recovery of dictionary and checkpoint information cannot be executed in parallel with performing of local checkpoints. This means that, when starting or restarting many data nodes

concurrently, data nodes may be forced to wait while a local checkpoint is performed, which can result in longer node recovery times.

It is possible to force a delay in the local checkpoint to permit more (and possibly all) data nodes to complete metadata synchronization; once each data node's metadata synchronization is complete, all of the data nodes can recover table data in parallel, even while the local checkpoint is being executed. To force such a delay, set `MaxLCPStartDelay`, which determines the number of seconds the cluster can wait to begin a local checkpoint while data nodes continue to synchronize metadata. This parameter should be set in the `[ndbd default]` section of the `config.ini` file, so that it is the same for all data nodes. The maximum value is 600; the default is 0.

- `NoOfFragmentLogFiles`

Version (or later)	NDB 8.0.13
Type or units	integer
Default	16
Range	3 - 4294967039 (0xFFFFFEFF)
Restart Type	**Initial Node Restart:** Requires a rolling restart of the cluster; each data node must be restarted with `--initial`. (NDB 8.0.13)

This parameter sets the number of REDO log files for the node, and thus the amount of space allocated to REDO logging. Because the REDO log files are organized in a ring, it is extremely important that the first and last log files in the set (sometimes referred to as the “head” and “tail” log files, respectively) do not meet. When these approach one another too closely, the node begins aborting all transactions encompassing updates due to a lack of room for new log records.

A `REDO` log record is not removed until both required local checkpoints have been completed since that log record was inserted. Checkpointing frequency is determined by its own set of configuration parameters discussed elsewhere in this chapter.

The default parameter value is 16, which by default means 16 sets of 4 16MB files for a total of 1024MB. The size of the individual log files is configurable using the `FragmentLogFileSize` parameter. In scenarios requiring a great many updates, the value for `NoOfFragmentLogFiles` may need to be set as high as 300 or even higher to provide sufficient space for REDO logs.

If the checkpointing is slow and there are so many writes to the database that the log files are full and the log tail cannot be cut without jeopardizing recovery, all updating transactions are aborted with internal error code 410 (`Out of log file space temporarily`). This condition prevails until a checkpoint has completed and the log tail can be moved forward.

Important

This parameter cannot be changed “on the fly”; you must restart the node using `--initial`. If you wish to change this value for all data nodes in a running cluster, you can do so using a rolling node restart (using `--initial` when starting each data node).

- `RecoveryWork`

Version (or later)	NDB 8.0.13
Type or units	integer
Default	60
Range	25 - 100
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

Percentage of storage overhead for LCP files. This parameter has an effect only when `EnablePartialLcp` is true, that is, only when partial local checkpoints are enabled. A higher value means:

- Fewer records are written for each LCP, LCPs use more space
- More work is needed during restarts

A lower value for `RecoveryWork` means:

- More records are written during each LCP, but LCPs require less space on disk.
- Less work during restart and thus faster restarts, at the expense of more work during normal operations

For example, setting `RecoveryWork` to 60 means that the total size of an LCP is roughly 1 + 0.6 = 1.6 times the size of the data to be checkpointed. This means that 60% more work is required during the restore phase of a restart compared to the work done during a restart that uses full checkpoints. (This is more than compensated for during other phases of the restart such that the restart as a whole is still faster when using partial LCPs than when using full LCPs.) In order not to fill up the redo log, it is necessary to write at 1 + (1 / `RecoveryWork`) times the rate of data changes during checkpoints—thus, when `RecoveryWork` = 60, it is necessary to write at approximately 1 + (1 / 0.6) = 2.67 times the change rate. In other words, if changes are being written at 10 MByte per second, the checkpoint needs to be written at roughly 26.7 MByte per second.

Setting `RecoveryWork` = 40 means that only 1.4 times the total LCP size is needed (and thus the restore phase takes 10 to 15 percent less time. In this case, the checkpoint write rate is 3.5 times the rate of change.

The NDB source distribution includes a test program for simulating LCPs. `lcp_simulator.cc` can be found in `storage/ndb/src/kernel/blocks/backup/`. To compile and run it on Unix platforms, execute the commands shown here:

```
$> gcc lcp_simulator.cc
$> ./a.out
```

This program has no dependencies other than `stdio.h`, and does not require a connection to an NDB cluster or a MySQL server. By default, it simulates 300 LCPs (three sets of 100 LCPs, each consisting of inserts, updates, and deletes, in turn), reporting the size of the LCP after each one. You can alter the simulation by changing the values of `recovery_work`, `insert_work`, and `delete_work` in the source and recompiling. For more information, see the source of the program.

- `InsertRecoveryWork`

Version (or later)	NDB 8.0.13

Type or units	integer
Default	40
Range	0 - 70
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

Percentage of `RecoveryWork` used for inserted rows. A higher value increases the number of writes during a local checkpoint, and decreases the total size of the LCP. A lower value decreases the number of writes during an LCP, but results in more space being used for the LCP, which means that recovery takes longer. This parameter has an effect only when `EnablePartialLcp` is true, that is, only when partial local checkpoints are enabled.

- `EnableRedoControl`

Version (or later)	NDB 8.0.13
Type or units	boolean
Default	false
Range	...
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

Enable adaptive checkpointing speed for controlling redo log usage. Set to `false` to disable (the default). Setting `EnablePartialLcp` to `false` also disables the adaptive calculation.

When enabled, `EnableRedoControl` allows the data nodes greater flexibility with regard to the rate at which they write LCPs to disk. More specifically, enabling this parameter means that higher write rates can be employed, so that LCPs can complete and Redo logs be trimmed more quickly, thereby reducing recovery time and disk space requirements. This functionality allows data nodes to make better use of the higher rate of I/O and greater bandwidth available from modern solid-state storage devices and protocols, such as solid-state drives (SSDs) using Non-Volatile Memory Express (NVMe).

The parameter currently defaults to `false` (disabled) due to the fact that `NDB` is still deployed widely on systems whose I/O or bandwidth is constrained relative to those employing solid-state technology, such as those using conventional hard disks (HDDs). In settings such as these, the `EnableRedoControl` mechanism can easily cause the I/O subsystem to become saturated, increasing wait times for data node input and output. In particular, this can cause issues with NDB Disk Data tables which have tablespaces or log file groups sharing a constrained IO subsystem with data node LCP and redo log files; such problems potentially include node or cluster failure due to GCP stop errors.

Metadata objects. The next set of `[ndbd]` parameters defines pool sizes for metadata objects, used to define the maximum number of attributes, tables, indexes, and trigger objects used by indexes, events, and replication between clusters.

Note

These act merely as “suggestions” to the cluster, and any that are not specified revert to the default values shown.

- `MaxNoOfAttributes`

Version (or later)	NDB 8.0.13
Type or units	integer
Default	1000
Range	32 - 4294967039 (0xFFFFFEFF)
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

This parameter sets a suggested maximum number of attributes that can be defined in the cluster; like `MaxNoOfTables`, it is not intended to function as a hard upper limit.

(In older NDB Cluster releases, this parameter was sometimes treated as a hard limit for certain operations. This caused problems with NDB Cluster Replication, when it was possible to create more tables than could be replicated, and sometimes led to confusion when it was possible [or not possible, depending on the circumstances] to create more than `MaxNoOfAttributes` attributes.)

The default value is 1000, with the minimum possible value being 32. The maximum is 4294967039. Each attribute consumes around 200 bytes of storage per node due to the fact that all metadata is fully replicated on the servers.

When setting `MaxNoOfAttributes`, it is important to prepare in advance for any `ALTER TABLE` statements that you might want to perform in the future. This is due to the fact, during the execution of `ALTER TABLE` on a Cluster table, 3 times the number of attributes as in the original table are used, and a good practice is to permit double this amount. For example, if the NDB Cluster table having the greatest number of attributes (*`greatest_number_of_attributes`*) has 100 attributes, a good starting point for the value of `MaxNoOfAttributes` would be `6 * greatest_number_of_attributes = 600`.

You should also estimate the average number of attributes per table and multiply this by `MaxNoOfTables`. If this value is larger than the value obtained in the previous paragraph, you should use the larger value instead.

Assuming that you can create all desired tables without any problems, you should also verify that this number is sufficient by trying an actual `ALTER TABLE` after configuring the parameter. If this is not successful, increase `MaxNoOfAttributes` by another multiple of `MaxNoOfTables` and test it again.

- `MaxNoOfTables`

Version (or later)	NDB 8.0.13
Type or units	integer
Default	128
Range	8 - 20320
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

A table object is allocated for each table and for each unique hash index in the cluster. This parameter sets a suggested maximum number of table objects for the cluster as a whole; like `MaxNoOfAttributes`, it is not intended to function as a hard upper limit.

(In older NDB Cluster releases, this parameter was sometimes treated as a hard limit for certain operations. This caused problems with NDB Cluster Replication, when it was possible to create more tables than could be replicated, and sometimes led to confusion when it was possible [or not possible, depending on the circumstances] to create more than `MaxNoOfTables` tables.)

For each attribute that has a `BLOB` data type an extra table is used to store most of the `BLOB` data. These tables also must be taken into account when defining the total number of tables.

The default value of this parameter is 128. The minimum is 8 and the maximum is 20320. Each table object consumes approximately 20KB per node.

Note

The sum of `MaxNoOfTables`, `MaxNoOfOrderedIndexes`, and `MaxNoOfUniqueHashIndexes` must not exceed $2^{32} - 2$ (4294967294).

- `MaxNoOfOrderedIndexes`

Version (or later)	NDB 8.0.13
Type or units	integer
Default	128
Range	0 - 4294967039 (0xFFFFFEFF)
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

For each ordered index in the cluster, an object is allocated describing what is being indexed and its storage segments. By default, each index so defined also defines an ordered index. Each unique index and primary key has both an ordered index and a hash index. `MaxNoOfOrderedIndexes` sets the total number of ordered indexes that can be in use in the system at any one time.

The default value of this parameter is 128. Each index object consumes approximately 10KB of data per node.

Note

The sum of `MaxNoOfTables`, `MaxNoOfOrderedIndexes`, and `MaxNoOfUniqueHashIndexes` must not exceed $2^{32} - 2$ (4294967294).

- `MaxNoOfUniqueHashIndexes`

Version (or later)	NDB 8.0.13
Type or units	integer
Default	64
Range	0 - 4294967039 (0xFFFFFEFF)

Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

For each unique index that is not a primary key, a special table is allocated that maps the unique key to the primary key of the indexed table. By default, an ordered index is also defined for each unique index. To prevent this, you must specify the `USING HASH` option when defining the unique index.

The default value is 64. Each index consumes approximately 15KB per node.

Note

The sum of `MaxNoOfTables`, `MaxNoOfOrderedIndexes`, and `MaxNoOfUniqueHashIndexes` must not exceed $2^{32} - 2$ (4294967294).

- `MaxNoOfTriggers`

Version (or later)	NDB 8.0.13
Type or units	integer
Default	768
Range	0 - 4294967039 (0xFFFFFEFF)
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

Internal update, insert, and delete triggers are allocated for each unique hash index. (This means that three triggers are created for each unique hash index.) However, an *ordered* index requires only a single trigger object. Backups also use three trigger objects for each normal table in the cluster.

Replication between clusters also makes use of internal triggers.

This parameter sets the maximum number of trigger objects in the cluster.

The default value is 768.

- `MaxNoOfSubscriptions`

Version (or later)	NDB 8.0.13
Type or units	unsigned
Default	0
Range	0 - 4294967039 (0xFFFFFEFF)
Restart Type	**Node Restart:** Requires a rolling restart

	of the cluster. (NDB 8.0.13)

Each `NDB` table in an NDB Cluster requires a subscription in the NDB kernel. For some NDB API applications, it may be necessary or desirable to change this parameter. However, for normal usage with MySQL servers acting as SQL nodes, there is not any need to do so.

The default value for `MaxNoOfSubscriptions` is 0, which is treated as equal to `MaxNoOfTables`. Each subscription consumes 108 bytes.

- `MaxNoOfSubscribers`

Version (or later)	NDB 8.0.13
Type or units	unsigned
Default	0
Range	0 - 4294967039 (0xFFFFFEFF)
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

This parameter is of interest only when using NDB Cluster Replication. The default value is 0. Prior to NDB 8.0.26, this was treated as `2 * MaxNoOfTables`; beginning with NDB 8.0.26, it is treated as `2 * MaxNoOfTables + 2 * [number of API nodes]`. There is one subscription per `NDB` table for each of two MySQL servers (one acting as the replication source and the other as the replica). Each subscriber uses 16 bytes of memory.

When using circular replication, multi-source replication, and other replication setups involving more than 2 MySQL servers, you should increase this parameter to the number of `mysqld` processes included in replication (this is often, but not always, the same as the number of clusters). For example, if you have a circular replication setup using three NDB Clusters, with one `mysqld` attached to each cluster, and each of these `mysqld` processes acts as a source and as a replica, you should set `MaxNoOfSubscribers` equal to `3 * MaxNoOfTables`.

For more information, see Section 23.7, “NDB Cluster Replication”.

- `MaxNoOfConcurrentSubOperations`

Version (or later)	NDB 8.0.13
Type or units	unsigned
Default	256
Range	0 - 4294967039 (0xFFFFFEFF)
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

This parameter sets a ceiling on the number of operations that can be performed by all API nodes in the cluster at one time. The default value (256) is sufficient for normal operations, and might need

to be adjusted only in scenarios where there are a great many API nodes each performing a high volume of operations concurrently.

Boolean parameters. The behavior of data nodes is also affected by a set of `[ndbd]` parameters taking on boolean values. These parameters can each be specified as `TRUE` by setting them equal to `1` or `Y`, and as `FALSE` by setting them equal to `0` or `N`.

- `CompressedLCP`

Version (or later)	NDB 8.0.13
Type or units	boolean
Default	false
Range	true, false
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

Setting this parameter to `1` causes local checkpoint files to be compressed. The compression used is equivalent to `gzip --fast`, and can save 50% or more of the space required on the data node to store uncompressed checkpoint files. Compressed LCPs can be enabled for individual data nodes, or for all data nodes (by setting this parameter in the `[ndbd default]` section of the `config.ini` file).

Important

You cannot restore a compressed local checkpoint to a cluster running a MySQL version that does not support this feature.

The default value is `0` (disabled).

Prior to NDB 8.0.29, this parameter had no effect on Windows platforms (BUG#106075, BUG#33727690).

- `CrashOnCorruptedTuple`

Version (or later)	NDB 8.0.13
Type or units	boolean
Default	true
Range	true, false
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

When this parameter is enabled (the default), it forces a data node to shut down whenever it encounters a corrupted tuple.

- `Diskless`

Version (or later)	NDB 8.0.13

Type or units	true|false (1|0)
Default	false
Range	true, false
Restart Type	**Initial System Restart:** Requires a complete shutdown of the cluster, wiping and restoring the cluster file system from a backup, and then restarting the cluster. (NDB 8.0.13)

It is possible to specify NDB Cluster tables as *diskless*, meaning that tables are not checkpointed to disk and that no logging occurs. Such tables exist only in main memory. A consequence of using diskless tables is that neither the tables nor the records in those tables survive a crash. However, when operating in diskless mode, it is possible to run `ndbd` on a diskless computer.

Important

This feature causes the *entire* cluster to operate in diskless mode.

When this feature is enabled, NDB Cluster online backup is disabled. In addition, a partial start of the cluster is not possible.

`Diskless` is disabled by default.

- `EncryptedFileSystem`

Version (or later)	NDB 8.0.13
Type or units	unsigned
Default	0
Range	0 - 1
Added	NDB 8.0.29
Restart Type	**Initial Node Restart:** Requires a rolling restart of the cluster; each data node must be restarted with

	`--initial`. (NDB 8.0.13)

Encrypt LCP and tablespace files, including undo logs and redo logs. Disabled by default (`0`); set to `1` to enable.

Important

When file system encryption is enabled, you must supply a password to each data node when starting it, using one of the options `--filesystem-password` or `--filesystem-password-from-stdin`. Otherwise, the data node cannot start.

For more information, see Section 23.6.14, “File System Encryption for NDB Cluster”.

- `LateAlloc`

Version (or later)	NDB 8.0.13
Type or units	numeric
Default	1
Range	0 - 1
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

Allocate memory for this data node after a connection to the management server has been established. Enabled by default.

- `LockPagesInMainMemory`

Version (or later)	NDB 8.0.13
Type or units	numeric
Default	0
Range	0 - 2
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

For a number of operating systems, including Solaris and Linux, it is possible to lock a process into memory and so avoid any swapping to disk. This can be used to help guarantee the cluster's real-time characteristics.

This parameter takes one of the integer values `0`, `1`, or `2`, which act as shown in the following list:

- `0`: Disables locking. This is the default value.
- `1`: Performs the lock after allocating memory for the process.

- 2: Performs the lock before memory for the process is allocated.

If the operating system is not configured to permit unprivileged users to lock pages, then the data node process making use of this parameter may have to be run as system root. (`LockPagesInMainMemory` uses the `mlockall` function. From Linux kernel 2.6.9, unprivileged users can lock memory as limited by `max locked memory`. For more information, see `ulimit -l` and http://linux.die.net/man/2/mlock).

Note

In older NDB Cluster releases, this parameter was a Boolean. `0` or `false` was the default setting, and disabled locking. `1` or `true` enabled locking of the process after its memory was allocated. NDB Cluster 8.0 treats `true` or `false` for the value of this parameter as an error.

Important

Beginning with `glibc` 2.10, `glibc` uses per-thread arenas to reduce lock contention on a shared pool, which consumes real memory. In general, a data node process does not need per-thread arenas, since it does not perform any memory allocation after startup. (This difference in allocators does not appear to affect performance significantly.)

The `glibc` behavior is intended to be configurable via the `MALLOC_ARENA_MAX` environment variable, but a bug in this mechanism prior to `glibc` 2.16 meant that this variable could not be set to less than 8, so that the wasted memory could not be reclaimed. (Bug #15907219; see also http://sourceware.org/bugzilla/show_bug.cgi?id=13137 for more information concerning this issue.)

One possible workaround for this problem is to use the `LD_PRELOAD` environment variable to preload a `jemalloc` memory allocation library to take the place of that supplied with `glibc`.

- `ODirect`

Version (or later)	NDB 8.0.13
Type or units	boolean
Default	false
Range	true, false
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

Enabling this parameter causes `NDB` to attempt using `O_DIRECT` writes for LCP, backups, and redo logs, often lowering `kswapd` and CPU usage. When using NDB Cluster on Linux, enable `ODirect` if you are using a 2.6 or later kernel.

`ODirect` is disabled by default.

- `ODirectSyncFlag`

Version (or later)	NDB 8.0.13

Type or units	boolean
Default	false
Range	true, false
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

When this parameter is enabled, redo log writes are performed such that each completed file system write is handled as a call to `fsync`. The setting for this parameter is ignored if at least one of the following conditions is true:

- `ODirect` is not enabled.
- `InitFragmentLogFiles` is set to `SPARSE`.

Disabled by default.

- `RestartOnErrorInsert`

Version (or later)	NDB 8.0.13
Type or units	error code
Default	2
Range	0 - 4
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

This feature is accessible only when building the debug version where it is possible to insert errors in the execution of individual blocks of code as part of testing.

This feature is disabled by default.

- `StopOnError`

Version (or later)	NDB 8.0.13
Type or units	boolean
Default	1
Range	0, 1
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

This parameter specifies whether a data node process should exit or perform an automatic restart when an error condition is encountered.

This parameter's default value is 1; this means that, by default, an error causes the data node process to halt.

When an error is encountered and `StopOnError` is 0, the data node process is restarted.

Users of MySQL Cluster Manager should note that, when `StopOnError` equals 1, this prevents the MySQL Cluster Manager agent from restarting any data nodes after it has performed its own restart and recovery. See Starting and Stopping the Agent on Linux, for more information.

- `UseShm`

Version (or later)	NDB 8.0.13
Type or units	boolean
Default	false
Range	true, false
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

Enable a shared memory connection between this data node and the API node also running on this host. Set to 1 to enable.

Controlling Timeouts, Intervals, and Disk Paging

There are a number of `[ndbd]` parameters specifying timeouts and intervals between various actions in Cluster data nodes. Most of the timeout values are specified in milliseconds. Any exceptions to this are mentioned where applicable.

- `TimeBetweenWatchDogCheck`

Version (or later)	NDB 8.0.13
Type or units	milliseconds
Default	6000
Range	70 - 4294967039 (0xFFFFFEFF)
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

To prevent the main thread from getting stuck in an endless loop at some point, a “watchdog” thread checks the main thread. This parameter specifies the number of milliseconds between checks. If the process remains in the same state after three checks, the watchdog thread terminates it.

This parameter can easily be changed for purposes of experimentation or to adapt to local conditions. It can be specified on a per-node basis although there seems to be little reason for doing so.

The default timeout is 6000 milliseconds (6 seconds).

- `TimeBetweenWatchDogCheckInitial`

Version (or later)	NDB 8.0.13
Type or units	milliseconds
Default	6000
Range	70 - 4294967039 (0xFFFFFEFF)
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

This is similar to the `TimeBetweenWatchDogCheck` parameter, except that `TimeBetweenWatchDogCheckInitial` controls the amount of time that passes between execution checks inside a storage node in the early start phases during which memory is allocated.

The default timeout is 6000 milliseconds (6 seconds).

- `StartPartialTimeout`

Version (or later)	NDB 8.0.13
Type or units	milliseconds
Default	30000
Range	0 - 4294967039 (0xFFFFFEFF)
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

This parameter specifies how long the Cluster waits for all data nodes to come up before the cluster initialization routine is invoked. This timeout is used to avoid a partial Cluster startup whenever possible.

This parameter is overridden when performing an initial start or initial restart of the cluster.

The default value is 30000 milliseconds (30 seconds). 0 disables the timeout, in which case the cluster may start only if all nodes are available.

- `StartPartitionedTimeout`

Version (or later)	NDB 8.0.13
Type or units	milliseconds
Default	0
Range	0 - 4294967039 (0xFFFFFEFF)
Restart Type	**Node Restart:** Requires a rolling restart

	of the cluster. (NDB 8.0.13)

If the cluster is ready to start after waiting for `StartPartialTimeout` milliseconds but is still possibly in a partitioned state, the cluster waits until this timeout has also passed. If `StartPartitionedTimeout` is set to 0, the cluster waits indefinitely ($2^{32}-1$ ms, or approximately 49.71 days).

This parameter is overridden when performing an initial start or initial restart of the cluster.

- `StartFailureTimeout`

Version (or later)	NDB 8.0.13
Type or units	milliseconds
Default	0
Range	0 - 4294967039 (0xFFFFFEFF)
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

If a data node has not completed its startup sequence within the time specified by this parameter, the node startup fails. Setting this parameter to 0 (the default value) means that no data node timeout is applied.

For nonzero values, this parameter is measured in milliseconds. For data nodes containing extremely large amounts of data, this parameter should be increased. For example, in the case of a data node containing several gigabytes of data, a period as long as 10−15 minutes (that is, 600000 to 1000000 milliseconds) might be required to perform a node restart.

- `StartNoNodeGroupTimeout`

Version (or later)	NDB 8.0.13
Type or units	milliseconds
Default	15000
Range	0 - 4294967039 (0xFFFFFEFF)
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

When a data node is configured with `Nodegroup = 65536`, is regarded as not being assigned to any node group. When that is done, the cluster waits `StartNoNodegroupTimeout` milliseconds, then treats such nodes as though they had been added to the list passed to the `--nowait-nodes` option, and starts. The default value is `15000` (that is, the management server waits 15 seconds). Setting this parameter equal to `0` means that the cluster waits indefinitely.

`StartNoNodegroupTimeout` must be the same for all data nodes in the cluster; for this reason, you should always set it in the `[ndbd default]` section of the `config.ini` file, rather than for individual data nodes.

See Section 23.6.7, “Adding NDB Cluster Data Nodes Online”, for more information.

- `HeartbeatIntervalDbDb`

Version (or later)	NDB 8.0.13
Type or units	milliseconds
Default	5000
Range	10 - 4294967039 (0xFFFFFEFF)
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

One of the primary methods of discovering failed nodes is by the use of heartbeats. This parameter states how often heartbeat signals are sent and how often to expect to receive them. Heartbeats cannot be disabled.

After missing four heartbeat intervals in a row, the node is declared dead. Thus, the maximum time for discovering a failure through the heartbeat mechanism is five times the heartbeat interval.

The default heartbeat interval is 5000 milliseconds (5 seconds). This parameter must not be changed drastically and should not vary widely between nodes. If one node uses 5000 milliseconds and the node watching it uses 1000 milliseconds, obviously the node is declared dead very quickly. This parameter can be changed during an online software upgrade, but only in small increments.

See also Network communication and latency, as well as the description of the `ConnectCheckIntervalDelay` configuration parameter.

- `HeartbeatIntervalDbApi`

Version (or later)	NDB 8.0.13
Type or units	milliseconds
Default	1500
Range	100 - 4294967039 (0xFFFFFEFF)
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

Each data node sends heartbeat signals to each MySQL server (SQL node) to ensure that it remains in contact. If a MySQL server fails to send a heartbeat in time it is declared “dead,” in which case all ongoing transactions are completed and all resources released. The SQL node cannot reconnect

until all activities initiated by the previous MySQL instance have been completed. The three-heartbeat criteria for this determination are the same as described for `HeartbeatIntervalDbDb`.

The default interval is 1500 milliseconds (1.5 seconds). This interval can vary between individual data nodes because each data node watches the MySQL servers connected to it, independently of all other data nodes.

For more information, see Network communication and latency.

- `HeartbeatOrder`

Version (or later)	NDB 8.0.13
Type or units	numeric
Default	0
Range	0 - 65535
Restart Type	**System Restart:** Requires a complete shutdown and restart of the cluster. (NDB 8.0.13)

Data nodes send heartbeats to one another in a circular fashion whereby each data node monitors the previous one. If a heartbeat is not detected by a given data node, this node declares the previous data node in the circle “dead” (that is, no longer accessible by the cluster). The determination that a data node is dead is done globally; in other words; once a data node is declared dead, it is regarded as such by all nodes in the cluster.

It is possible for heartbeats between data nodes residing on different hosts to be too slow compared to heartbeats between other pairs of nodes (for example, due to a very low heartbeat interval or temporary connection problem), such that a data node is declared dead, even though the node can still function as part of the cluster. .

In this type of situation, it may be that the order in which heartbeats are transmitted between data nodes makes a difference as to whether or not a particular data node is declared dead. If this declaration occurs unnecessarily, this can in turn lead to the unnecessary loss of a node group and as thus to a failure of the cluster.

Consider a setup where there are 4 data nodes A, B, C, and D running on 2 host computers `host1` and `host2`, and that these data nodes make up 2 node groups, as shown in the following table:

Table 23.10 Four data nodes A, B, C, D running on two host computers host1, host2; each data node belongs to one of two node groups.

Node Group	Nodes Running on `host1`	Nodes Running on `host2`
Node Group 0:	Node A	Node B
Node Group 1:	Node C	Node D

Suppose the heartbeats are transmitted in the order A->B->C->D->A. In this case, the loss of the heartbeat between the hosts causes node B to declare node A dead and node C to declare node B dead. This results in loss of Node Group 0, and so the cluster fails. On the other hand, if the order of transmission is A->B->D->C->A (and all other conditions remain as previously stated), the loss of

the heartbeat causes nodes A and D to be declared dead; in this case, each node group has one surviving node, and the cluster survives.

The `HeartbeatOrder` configuration parameter makes the order of heartbeat transmission user-configurable. The default value for `HeartbeatOrder` is zero; allowing the default value to be used on all data nodes causes the order of heartbeat transmission to be determined by `NDB`. If this parameter is used, it must be set to a nonzero value (maximum 65535) for every data node in the cluster, and this value must be unique for each data node; this causes the heartbeat transmission to proceed from data node to data node in the order of their `HeartbeatOrder` values from lowest to highest (and then directly from the data node having the highest `HeartbeatOrder` to the data node having the lowest value, to complete the circle). The values need not be consecutive. For example, to force the heartbeat transmission order A->B->D->C->A in the scenario outlined previously, you could set the `HeartbeatOrder` values as shown here:

Table 23.11 HeartbeatOrder values to force a heartbeat transition order of A->B->D->C->A.

Node	`HeartbeatOrder` Value
A	10
B	20
C	30
D	25

To use this parameter to change the heartbeat transmission order in a running NDB Cluster, you must first set `HeartbeatOrder` for each data node in the cluster in the global configuration (`config.ini`) file (or files). To cause the change to take effect, you must perform either of the following:

- A complete shutdown and restart of the entire cluster.
- 2 rolling restarts of the cluster in succession. *All nodes must be restarted in the same order in both rolling restarts*.

You can use `DUMP 908` to observe the effect of this parameter in the data node logs.

- `ConnectCheckIntervalDelay`

Version (or later)	NDB 8.0.13
Type or units	milliseconds
Default	0
Range	0 - 4294967039 (0xFFFFFEFF)
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

This parameter enables connection checking between data nodes after one of them has failed heartbeat checks for 5 intervals of up to `HeartbeatIntervalDbDb` milliseconds.

Such a data node that further fails to respond within an interval of `ConnectCheckIntervalDelay` milliseconds is considered suspect, and is considered dead after two such intervals. This can be useful in setups with known latency issues.

The default value for this parameter is 0 (disabled).

- `TimeBetweenLocalCheckpoints`

Version (or later)	NDB 8.0.13
Type or units	number of 4-byte words, as base-2 logarithm
Default	20
Range	0 - 31
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

This parameter is an exception in that it does not specify a time to wait before starting a new local checkpoint; rather, it is used to ensure that local checkpoints are not performed in a cluster where relatively few updates are taking place. In most clusters with high update rates, it is likely that a new local checkpoint is started immediately after the previous one has been completed.

The size of all write operations executed since the start of the previous local checkpoints is added. This parameter is also exceptional in that it is specified as the base-2 logarithm of the number of 4-byte words, so that the default value 20 means 4MB (4×2^{20}) of write operations, 21 would mean 8MB, and so on up to a maximum value of 31, which equates to 8GB of write operations.

All the write operations in the cluster are added together. Setting `TimeBetweenLocalCheckpoints` to 6 or less means that local checkpoints are executed continuously without pause, independent of the cluster's workload.

- `TimeBetweenGlobalCheckpoints`

Version (or later)	NDB 8.0.13
Type or units	milliseconds
Default	2000
Range	20 - 32000
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

When a transaction is committed, it is committed in main memory in all nodes on which the data is mirrored. However, transaction log records are not flushed to disk as part of the commit. The reasoning behind this behavior is that having the transaction safely committed on at least two autonomous host machines should meet reasonable standards for durability.

It is also important to ensure that even the worst of cases—a complete crash of the cluster—is handled properly. To guarantee that this happens, all transactions taking place within a given interval are put into a global checkpoint, which can be thought of as a set of committed transactions that has been flushed to disk. In other words, as part of the commit process, a transaction is placed in a

global checkpoint group. Later, this group's log records are flushed to disk, and then the entire group of transactions is safely committed to disk on all computers in the cluster.

In NDB 8.0, we recommended when you are using solid-state disks (especially those employing NVMe) with Disk Data tables that you reduce this value. In such cases, you should also ensure that `MaxDiskDataLatency` is set to a proper level.

This parameter defines the interval between global checkpoints. The default is 2000 milliseconds.

- `TimeBetweenGlobalCheckpointsTimeout`

Version (or later)	NDB 8.0.13
Type or units	milliseconds
Default	120000
Range	10 - 4294967039 (0xFFFFFEFF)
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

This parameter defines the minimum timeout between global checkpoints. The default is 120000 milliseconds.

- `TimeBetweenEpochs`

Version (or later)	NDB 8.0.13
Type or units	milliseconds
Default	100
Range	0 - 32000
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

This parameter defines the interval between synchronization epochs for NDB Cluster Replication. The default value is 100 milliseconds.

`TimeBetweenEpochs` is part of the implementation of “micro-GCPs”, which can be used to improve the performance of NDB Cluster Replication.

- `TimeBetweenEpochsTimeout`

Version (or later)	NDB 8.0.13
Type or units	milliseconds
Default	0
Range	0 - 256000
Restart Type	**Node Restart:** Requires a

	rolling restart of the cluster. (NDB 8.0.13)

This parameter defines a timeout for synchronization epochs for NDB Cluster Replication. If a node fails to participate in a global checkpoint within the time determined by this parameter, the node is shut down. The default value is 0; in other words, the timeout is disabled.

`TimeBetweenEpochsTimeout` is part of the implementation of “micro-GCPs”, which can be used to improve the performance of NDB Cluster Replication.

The current value of this parameter and a warning are written to the cluster log whenever a GCP save takes longer than 1 minute or a GCP commit takes longer than 10 seconds.

Setting this parameter to zero has the effect of disabling GCP stops caused by save timeouts, commit timeouts, or both. The maximum possible value for this parameter is 256000 milliseconds.

- `MaxBufferedEpochs`

Version (or later)	NDB 8.0.13
Type or units	epochs
Default	100
Range	0 - 100000
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

The number of unprocessed epochs by which a subscribing node can lag behind. Exceeding this number causes a lagging subscriber to be disconnected.

The default value of 100 is sufficient for most normal operations. If a subscribing node does lag enough to cause disconnections, it is usually due to network or scheduling issues with regard to processes or threads. (In rare circumstances, the problem may be due to a bug in the `NDB` client.) It may be desirable to set the value lower than the default when epochs are longer.

Disconnection prevents client issues from affecting the data node service, running out of memory to buffer data, and eventually shutting down. Instead, only the client is affected as a result of the disconnect (by, for example gap events in the binary log), forcing the client to reconnect or restart the process.

- `MaxBufferedEpochBytes`

Version (or later)	NDB 8.0.13
Type or units	bytes
Default	26214400
Range	26214400 (0x01900000) - 4294967039 (0xFFFFFEFF)
Restart Type	**Node Restart:** Requires a rolling restart

	of the cluster. (NDB 8.0.13)

The total number of bytes allocated for buffering epochs by this node.

- `TimeBetweenInactiveTransactionAbortCheck`

Version (or later)	NDB 8.0.13
Type or units	milliseconds
Default	1000
Range	1000 - 4294967039 (0xFFFFFEFF)
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

Timeout handling is performed by checking a timer on each transaction once for every interval specified by this parameter. Thus, if this parameter is set to 1000 milliseconds, every transaction is checked for timing out once per second.

The default value is 1000 milliseconds (1 second).

- `TransactionInactiveTimeout`

Version (or later)	NDB 8.0.13
Type or units	milliseconds
Default	4294967039 (0xFFFFFEFF)
Range	0 - 4294967039 (0xFFFFFEFF)
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

This parameter states the maximum time that is permitted to lapse between operations in the same transaction before the transaction is aborted.

The default for this parameter is `4G` (also the maximum). For a real-time database that needs to ensure that no transaction keeps locks for too long, this parameter should be set to a relatively small value. Setting it to 0 means that the application never times out. The unit is milliseconds.

- `TransactionDeadlockDetectionTimeout`

Version (or later)	NDB 8.0.13
Type or units	milliseconds
Default	1200

Range	50 - 4294967039 (0xFFFFFEFF)
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

When a node executes a query involving a transaction, the node waits for the other nodes in the cluster to respond before continuing. This parameter sets the amount of time that the transaction can spend executing within a data node, that is, the time that the transaction coordinator waits for each data node participating in the transaction to execute a request.

A failure to respond can occur for any of the following reasons:

- The node is “dead”
- The operation has entered a lock queue
- The node requested to perform the action could be heavily overloaded.

This timeout parameter states how long the transaction coordinator waits for query execution by another node before aborting the transaction, and is important for both node failure handling and deadlock detection.

The default timeout value is 1200 milliseconds (1.2 seconds).

The minimum for this parameter is 50 milliseconds.

- `DiskSyncSize`

Version (or later)	NDB 8.0.13
Type or units	bytes
Default	4M
Range	32K - 4294967039 (0xFFFFFEFF)
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

This is the maximum number of bytes to store before flushing data to a local checkpoint file. This is done to prevent write buffering, which can impede performance significantly. This parameter is *not* intended to take the place of `TimeBetweenLocalCheckpoints`.

Note

When `ODirect` is enabled, it is not necessary to set `DiskSyncSize`; in fact, in such cases its value is simply ignored.

The default value is 4M (4 megabytes).

- `MaxDiskWriteSpeed`

Version (or later)	NDB 8.0.13
Type or units	numeric
Default	20M
Range	1M - 1024G
Restart Type	**System Restart:** Requires a complete shutdown and restart of the cluster. (NDB 8.0.13)

Set the maximum rate for writing to disk, in bytes per second, by local checkpoints and backup operations when no restarts (by this data node or any other data node) are taking place in this NDB Cluster.

For setting the maximum rate of disk writes allowed while this data node is restarting, use `MaxDiskWriteSpeedOwnRestart`. For setting the maximum rate of disk writes allowed while other data nodes are restarting, use `MaxDiskWriteSpeedOtherNodeRestart`. The minimum speed for disk writes by all LCPs and backup operations can be adjusted by setting `MinDiskWriteSpeed`.

- `MaxDiskWriteSpeedOtherNodeRestart`

Version (or later)	NDB 8.0.13
Type or units	numeric
Default	50M
Range	1M - 1024G
Restart Type	**System Restart:** Requires a complete shutdown and restart of the cluster. (NDB 8.0.13)

Set the maximum rate for writing to disk, in bytes per second, by local checkpoints and backup operations when one or more data nodes in this NDB Cluster are restarting, other than this node.

For setting the maximum rate of disk writes allowed while this data node is restarting, use `MaxDiskWriteSpeedOwnRestart`. For setting the maximum rate of disk writes allowed when no data nodes are restarting anywhere in the cluster, use `MaxDiskWriteSpeed`. The minimum speed for disk writes by all LCPs and backup operations can be adjusted by setting `MinDiskWriteSpeed`.

- `MaxDiskWriteSpeedOwnRestart`

Version (or later)	NDB 8.0.13
Type or units	numeric
Default	200M

Range	1M - 1024G
Restart Type	**System Restart:** Requires a complete shutdown and restart of the cluster. (NDB 8.0.13)

Set the maximum rate for writing to disk, in bytes per second, by local checkpoints and backup operations while this data node is restarting.

For setting the maximum rate of disk writes allowed while other data nodes are restarting, use `MaxDiskWriteSpeedOtherNodeRestart`. For setting the maximum rate of disk writes allowed when no data nodes are restarting anywhere in the cluster, use `MaxDiskWriteSpeed`. The minimum speed for disk writes by all LCPs and backup operations can be adjusted by setting `MinDiskWriteSpeed`.

- `MinDiskWriteSpeed`

Version (or later)	NDB 8.0.13
Type or units	numeric
Default	10M
Range	1M - 1024G
Restart Type	**System Restart:** Requires a complete shutdown and restart of the cluster. (NDB 8.0.13)

Set the minimum rate for writing to disk, in bytes per second, by local checkpoints and backup operations.

The maximum rates of disk writes allowed for LCPs and backups under various conditions are adjustable using the parameters `MaxDiskWriteSpeed`, `MaxDiskWriteSpeedOwnRestart`, and `MaxDiskWriteSpeedOtherNodeRestart`. See the descriptions of these parameters for more information.

- `ArbitrationTimeout`

Version (or later)	NDB 8.0.13
Type or units	milliseconds
Default	7500
Range	10 - 4294967039 (0xFFFFFEFF)
Restart Type	**Node Restart:** Requires a rolling restart

	of the cluster. (NDB 8.0.13)

This parameter specifies how long data nodes wait for a response from the arbitrator to an arbitration message. If this is exceeded, the network is assumed to have split.

The default value is 7500 milliseconds (7.5 seconds).

- `Arbitration`

Version (or later)	NDB 8.0.13
Type or units	enumeration
Default	Default
Range	Default, Disabled, WaitExternal
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

The `Arbitration` parameter enables a choice of arbitration schemes, corresponding to one of 3 possible values for this parameter:

- **Default.** This enables arbitration to proceed normally, as determined by the `ArbitrationRank` settings for the management and API nodes. This is the default value.
- **Disabled.** Setting `Arbitration = Disabled` in the `[ndbd default]` section of the `config.ini` file to accomplishes the same task as setting `ArbitrationRank` to 0 on all management and API nodes. When `Arbitration` is set in this way, any `ArbitrationRank` settings are ignored.
- **WaitExternal.** The `Arbitration` parameter also makes it possible to configure arbitration in such a way that the cluster waits until after the time determined by `ArbitrationTimeout` has passed for an external cluster manager application to perform arbitration instead of handling arbitration internally. This can be done by setting `Arbitration = WaitExternal` in the `[ndbd default]` section of the `config.ini` file. For best results with the `WaitExternal` setting, it is recommended that `ArbitrationTimeout` be 2 times as long as the interval required by the external cluster manager to perform arbitration.

Important

This parameter should be used only in the `[ndbd default]` section of the cluster configuration file. The behavior of the cluster is unspecified when `Arbitration` is set to different values for individual data nodes.

- `RestartSubscriberConnectTimeout`

Version (or later)	NDB 8.0.13
Type or units	ms
Default	12000
Range	0 - 4294967039 (0xFFFFFEFF)

Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

This parameter determines the time that a data node waits for subscribing API nodes to connect. Once this timeout expires, any “missing” API nodes are disconnected from the cluster. To disable this timeout, set `RestartSubscriberConnectTimeout` to 0.

While this parameter is specified in milliseconds, the timeout itself is resolved to the next-greatest whole second.

- `KeepAliveSendInterval`

Version (or later)	NDB 8.0.13
Type or units	integer
Default	60000
Range	0 - 4294967039 (0xFFFFFEFF)
Added	NDB 8.0.27
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

Beginning with NDB 8.0.27, it is possible to enable and control the interval between keep-alive signals sent between data nodes by setting this parameter. The default for `KeepAliveSendInterval` is 60000 milliseconds (one minute); setting it to 0 disables keep-alive signals. Values between 1 and 10 inclusive are treated as 10.

This parameter may prove useful in environments which monitor and disconnect idle TCP connections, possibly causing unnecessary data node failures when the cluster is idle.

The heartbeat interval between management nodes and data nodes is always 100 milliseconds, and is not configurable.

Buffering and logging. Several `[ndbd]` configuration parameters enable the advanced user to have more control over the resources used by node processes and to adjust various buffer sizes at need.

These buffers are used as front ends to the file system when writing log records to disk. If the node is running in diskless mode, these parameters can be set to their minimum values without penalty due to the fact that disk writes are “faked” by the `NDB` storage engine's file system abstraction layer.

- `UndoIndexBuffer`

Version (or later)	NDB 8.0.13
Type or units	unsigned
Default	2M
Range	1M - 4294967039 (0xFFFFFEFF)

Deprecated	NDB 8.0.27
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

This parameter formerly set the size of the undo index buffer, but has no effect in current versions of NDB Cluster.

In NDB 8.0.27 and later, the use of this parameter in the cluster configuration file raises a deprecation warning; you should expect it to be removed in a future NDB Cluster release.

- `UndoDataBuffer`

Version (or later)	NDB 8.0.13
Type or units	unsigned
Default	16M
Range	1M - 4294967039 (0xFFFFFEFF)
Deprecated	NDB 8.0.27
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

This parameter formerly set the size of the undo data buffer, but has no effect in current versions of NDB Cluster.

In NDB 8.0.27 and later, the use of this parameter in the cluster configuration file raises a deprecation warning; you should expect it to be removed in a future NDB Cluster release.

- `RedoBuffer`

Version (or later)	NDB 8.0.13
Type or units	bytes
Default	32M
Range	1M - 4294967039 (0xFFFFFEFF)
Restart Type	**Node Restart:** Requires a rolling restart of the cluster. (NDB 8.0.13)

All update activities also need to be logged. The REDO log makes it possible to replay these updates whenever the system is restarted. The NDB recovery algorithm uses a “fuzzy” checkpoint of the