

## **MySQL 5.1 Cluster DBA Certification Study Guide**

This document is an excerpt of the MySQL 5.1 Cluster DBA Certification Study Guide. You can purchase the full version of the book at:

[http://store.vervante.com/c/v/595352502.html?base\\_cat=Sun%20Microsystems%3aMySQL&pard=sun](http://store.vervante.com/c/v/595352502.html?base_cat=Sun%20Microsystems%3aMySQL&pard=sun)

---

## Chapter 6. Cluster Management

**Description:** This chapter covers the following topics relating to administration of a MySQL Cluster, a basic knowledge of which is expected for the certification exam:

- The basic functions and purpose of the MySQL Cluster management client (`ndb_mgm`)
- Methods of starting, stopping, and restarting a cluster and individual cluster nodes
- Rolling restarts - what they are, and how to perform them
- The basics of monitoring a running cluster's status
- Logging events and levels; how to control what is recorded in the cluster logs
- How to enter and exit single user mode, and when it necessary or appropriate to make use of it

### 6.1. Management Client Basics

In this section, we discuss basic usage of `ndb_mgm`, the MySQL Cluster Management client.

The following topics are covered:

- Starting and exiting the management client
- The `START`, `STOP`, `RESTART`, and `SHUTDOWN` commands.

The management client is a command-line tool, usually located in the `mysql/bin` directory; typically this is `/usr/local/mysql/bin`, but this may differ slightly, depending on how MySQL was installed.

To start the management client, it is necessary to invoke `ndb_mgm` from the command line. If this is on the same machine on which the management server is running, and the management server is using the default port, then you can invoke it without any extra arguments. That is, the default connectstring is `localhost:1186`.

Assuming that you have navigated to the directory where `ndb_mgm` is located, you can in such cases invoke it like this:

```
shell> ./ndb_mgm
```

To specify another host, port, or both, you can use the `--connect-string=connectstring` option (or its short form `-c connectstring`) to pass a connectstring to the management client. This means that it is possible to connect to a management server running on another machine. Consider the following examples:

- The management client connects to a management server running at IP address 10.0.0.1 and on the default port (1186):

```
shell> ./ndb_mgm -c 10.0.0.1
```

---

## 6.1. Management Client Basics

---

- The management client connects to a management server running on a computer whose hostname is `sakila`, on the default port:

```
shell> ./ndb_mgm -c sakila
```

### Note

`ndb_mgm` uses whatever facilities are available on the system where it is run to perform hostname resolution, such as a DNS client or the `/etc/hosts` file.

- The management client connects to a management server running at IP address `10.1.1.100`, and on port `1100`:

```
shell> ./ndb_mgm -c 10.1.1.100:1100
```

- By using `localhost` as the hostname, you can use the management client to connect to a management server that is running on the same machine, but using a port other than `1186`, like this:

```
shell> ./ndb_mgm -c localhost:2020
```

In this case, the management client attempts to connect to a management node running on the same host but using port `2020` instead of `1186`.

You can tell that you are in the management client rather than a system shell by the management client's distinctive prompt, as shown here in **bold type**:

```
shell> ./ndb_mgm
-- NDB Cluster -- Management Client --
ndb_mgm>
```

**Basic `ndb_mgm` commands.** The management client supports the following commands that can be used to connect to a local management server; to cause data nodes to join the cluster; to stop or restart data nodes; and to exit `ndb_mgm` when you are finished using it:

- The `CONNECT` command can be used in the management client to connect to a management server, which is useful in situations where `ndb_mgm` is already running before the management server is started, or where `ndb_mgmd` has been restarted.

The syntax for this command is shown here:

```
CONNECT hostname[:port]
```

It is also useful in the event you mistype or forget to include the connectstring when starting the management client. For example, suppose you wish to connect to a management server at `10.0.0.1`, and then use the `SHOW` command. However, you accidentally type `"10.0.0.11"` instead. When `ndb_mgm` is unable to connect to the management server, no error is displayed until you actually

---

## 6.1. Management Client Basics

---

attempt to execute a management client command. This example illustrates what happens in such a case:

```
shell> ./ndb_mgm -c 10.0.0.11
-- NDB Cluster -- Management Client --
ndb_mgm> SHOW
Unable to connect with connect string: nodeid=0,10.0.0.11:1186
Retrying every 5 seconds. Attempts left: 2 1, failed.
ndb_mgm>
```

Rather than exiting the management client and restarting it with the correct connectstring, you can use the **CONNECT** command, as shown here:

```
ndb_mgm> CONNECT 10.0.0.1
Connected to Management Server at: 10.0.0.1:1186
ndb_mgm> SHOW
Cluster Configuration
-----
[ndbd(NDB)] 4 node(s)
id=2 @10.0.0.2 (Version: 5.1.24-ndb-6.2.16, Nodegroup: 0)
id=3 @10.0.0.3 (Version: 5.1.24-ndb-6.2.16, Nodegroup: 0)
id=4 @10.0.0.4 (Version: 5.1.24-ndb-6.2.16, Nodegroup: 1)
id=5 @10.0.0.5 (Version: 5.1.24-ndb-6.2.16, Nodegroup: 1, Master)

[ndb_mgmd(MGM)] 1 node(s)
id=1 @10.0.0.1 (Version: 5.1.24-ndb-6.2.16)

[mysqld(API)] 2 node(s)
id=6 @10.0.0.6 (Version: 5.1.24-ndb-6.2.16)
id=7 (not connected, accepting connect from any host)
```

Another scenario in which this can be useful is when contact is lost with one management server in a cluster having multiple management servers, in which case **CONNECT** can be used to contact a different management server.

Notice that `ndb_mgm` itself is *not* represented in the output of the **SHOW** command. This is because `ndb_mgm` is not a data node, management node, or API node. `ndb_mgm` does not connect to data nodes or API nodes, but rather connects directly to a management server only. This means that the following statements are true:

- Unlike API nodes, `ndb_mgm` can connect to a management server even when no data nodes are running.
- Unlike `mysql` or other MySQL client applications, `ndb_mgm` does not require that a MySQL server be running in order to connect.
- `ndb_mgm` cannot connect to the cluster if no cluster management server is running.
- The **START** command can be used with one or more data nodes that were started using the `-nostart` or `-n` option, as discussed in Section 6.2.1.2, “Starting the Cluster Using the Management Client”, to cause the node or nodes to join the cluster.

The syntax for this command is as follows:

```
{node_id|ALL} START
```

## 6.1. Management Client Basics

---

To cause a data node to join the cluster, this command must be preceded by that data node's node ID. Alternatively, you may use `ALL`, which causes all waiting data nodes to join the cluster.

The `START` command affects data nodes only, and cannot be used to start management or API nodes, nor can it be used to start a data node process; the `ndbd` executable must already have been invoked. For additional information on the use of this command and on starting and restarting other types of nodes, see Section 6.2.1, “Methods for Starting a MySQL Cluster”.

- The `STOP` command is used to stop a data node. Its syntax is as shown here:

```
{node_id|ALL} STOP
```

Issuing `ALL STOP` differs from using the `SHUTDOWN` command in that `ALL STOP` stops all data nodes (and only data nodes), whereas `SHUTDOWN` stops all cluster data and management nodes.

- The `SHUTDOWN` command is used to shut down the cluster. It stops all data and management nodes.

### Note

When the cluster is shut down, any data that was committed since the last global checkpoint is lost. See Section 3.7.3, “Node Recovery”, for more information about checkpointing.

- The `RESTART` command can be used to restart one or more data nodes which are already running, and to cause them to rejoin the cluster. Its syntax, including the use of the `ALL` keyword, is identical to that for the `START` and `STOP` commands:

```
{node_id|ALL} RESTART
```

This command can also be used to restart a management server. `ALL RESTART` restarts all data and management nodes that are running at the time the command is issued.

- To exit the management client, use the `EXIT` command or its alias `QUIT`, as shown here:

```
ndb_mgm> EXIT  
shell>
```

or

```
ndb_mgm> QUIT  
shell>
```

This command does not affect the operation of the cluster.

None of the commands `START`, `RESTART`, `STOP`, or `SHUTDOWN` has any effect on API nodes, including SQL nodes.

`STOP` cannot be used to stop a data node when it is the only operational data node in its node group, as shown in this example:

## 6.2. Starting, Stopping, and Restarting the Cluster

---

```
ndb_mgm> SHOW
Cluster Configuration
-----
[ndbd(NDB)] 4 node(s)
id=2      @10.0.0.2 (Version: 5.1.24-ndb-6.2.16, Nodegroup: 0)
id=3      @10.0.0.3 (Version: 5.1.24-ndb-6.2.16, Nodegroup: 0)
id=4      @10.0.0.4 (Version: 5.1.24-ndb-6.2.16, Nodegroup: 1)
id=5      @10.0.0.5 (Version: 5.1.24-ndb-6.2.16, Nodegroup: 1, Master)

[ndb_mgmd(MGM)] 1 node(s)
id=1      @10.0.0.1 (Version: 5.1.24-ndb-6.2.16)

[mysqld(API)] 2 node(s)
id=6      @10.0.0.6 (Version: 5.1.24-ndb-6.2.16)
id=7 (not connected, accepting connect from any host)
```

Node group 0 has two nodes, Node 2 and Node 3. First, shut down Node 2 using STOP:

```
ndb_mgm> 2 STOP
Node 2: Node shutdown initiated
Node 2: Node shutdown completed.
Node 2 has shutdown.

ndb_mgm> SHOW
Cluster Configuration
-----
[ndbd(NDB)] 4 node(s)
id=2 (not connected, accepting connect from 10.0.0.2)
id=3      @10.0.0.3 (Version: 5.1.24-ndb-6.2.16, Nodegroup: 0)
id=4      @10.0.0.4 (Version: 5.1.24-ndb-6.2.16, Nodegroup: 1)
id=5      @10.0.0.5 (Version: 5.1.24-ndb-6.2.16, Nodegroup: 1, Master)

[ndb_mgmd(MGM)] 1 node(s)
id=1      @10.0.0.1 (Version: 5.1.24-ndb-6.2.16)

[mysqld(API)] 2 node(s)
id=6      @10.0.0.6 (Version: 5.1.24-ndb-6.2.16)
id=7 (not connected, accepting connect from any host)
```

If you try to stop the remaining node in Node Group 0, an error results:

```
ndb_mgm> 3 STOP
Node 2: Node shutdown aborted
Shutdown failed.
* 2002: Stop failed
*      Node shutdown would cause system crash
```

Similarly, RESTART cannot be used to restart a data node when it is the only “live” node in its node group.

Additional management client commands are available for monitoring the status of the cluster (and of individual cluster nodes) as well as for setting logging levels. These are covered in, respectively, Section 6.3.1, “Monitoring the Cluster”, and Section 6.3.2, “Understanding and Using Cluster Logs”.

## 6.2. Starting, Stopping, and Restarting the Cluster

---

### 6.2.1. Methods for Starting a MySQL Cluster

---

This section covers methods of stopping, starting, and restarting a MySQL Cluster, including cluster startup and restart types, as well as when each of these types is appropriate.

Topics to be covered:

- Starting the cluster without making use of the management client.
- `ndbd` options required to enable `START` in the management client
- How to perform a rolling restart of the cluster; how to perform a rolling upgrade (or downgrade).

### 6.2.1. Methods for Starting a MySQL Cluster

Starting a MySQL Cluster involves the following steps:

1. First, you need to start the management server (`ndb_mgmd` process).

(It is possible to have more than one management server for a cluster; we will discuss this type of setup later.)

2. Once the management server is running, you must start the data nodes (`ndbd` processes).
3. After all data nodes are fully started and are connected to the management server, you can start any SQL nodes that are to be used with the cluster.

You can choose to start a MySQL Cluster with or without relying on the Cluster management client (`ndb_mgm`). Both scenarios follow the general steps above, but differ in how the data nodes are started. We cover these separately in the next two sections.

Whether or not you employ the management client in starting the cluster, you must be able to use a login account on each computer acting as a node host. Physical access to the machines is not required; a remote login is sufficient. However, each login account must have appropriate privileges for the host on which it is used.

Before moving on to the actual startup procedures, it is best to be clear on the minimum configuration requirements for cluster nodes. The requirements for which parameters must be set in the cluster's global configuration (`config.ini`) file vary according to whether the cluster has only one, or more than one, management node, as shown in the following table:

Section of Cluster Configuration File	Required Parameters (Single Management Node)	Required Parameters (Multiple Management Nodes)
[ndbd default]	NoOfReplicas (no default value)	NoOfReplicas (no default value)
[ndbd]	HostName	HostName, NodeId
[ndb_mgmd]	HostName	HostName, NodeId

### 6.2.1. Methods for Starting a MySQL Cluster

Section of Cluster Configuration File	Required Parameters (Single Management Node)	Required Parameters (Multiple Management Nodes)
[api] / [mysqld]	[none]	[none]

In addition, you must specify a *connectstring* for each data node and API node so that it can locate the host and port on which the management server is running. **Exception:** For a node running on the same host as the management node, where the management node is using the default port, no connectstring need be supplied, since the default connectstring is `localhost:1186`. For more information about these parameters, see Section 4.3.3, “Sections in the `config.ini` Configuration File”.

The connectstring can be specified in one of two ways:

- *In the local `my.cnf` configuration file on the computer hosting each data node or SQL node:*

For a data node, this is accomplished by setting the `connect-string` option in an `[ndbd]` section or the `ndb-connectstring` option in the `[mysql_cluster]` section.

For an SQL node, you can set the `ndb-connectstring` option in a `[mysqld]` or `[mysql_cluster]` section.

- *On the command line:*

Start the data node using the `--connect-string` or `-c` option.

Start the SQL node using the `--ndb-connectstring` option.

For a cluster with multiple management servers, the connectstring should include the hostnames or IP addresses (and, if necessary, the ports) for all of the management servers. A complete connectstring for a data node with node ID 5, connecting to two management servers using non-default ports, might look like this:

```
nodeid=5,192.168.0.179:1100,192.168.0.181:1200
```

For detailed information about MySQL Cluster connectstrings, see Section 4.2, “The Cluster Connectstring”.

In addition, each SQL node must be started with the `ndbcluster` (`my.cnf` file) or `-ndbcluster` (command line) option. This is described in more detail in the next section.

#### 6.2.1.1. Starting the Cluster without the Management Client

It is possible to start the cluster with or without the use of the MySQL Cluster management client. In this section, we discuss the latter, which requires the following steps:



## 6.2.1. Methods for Starting a MySQL Cluster

---

### 1. Start the management server (ndb\_mgmd process).

- a. In the operating system shell (such as `bash`), navigate to the directory containing the `ndb_mgmd` binary. This is typically the `/usr/local/mysql/bin` directory, but may vary according to the type and method of installation that was used. (For example, when installing from source, `ndbd` and `ndb_mgmd` are placed by default in `/usr/local/mysql/libexec`, but this can be changed by using the `--prefix` option with `configure` or one of the build scripts found in the source tree's `BUILD` directory.)

For setups with multiple management servers, it is necessary to start all of the management servers before starting any of the data nodes. Otherwise, some data nodes may connect only to one management server, and others only to a different management server, which could result in the cluster never fully starting or even in a “split-brain” situation (see Section 3.7.2, “Arbitration”).

It is not strictly necessary to start Cluster executables such as `ndb_mgmd` from the directories in they reside; if you choose not to do so, be sure to include the path to the executable when invoking it.

You can also create links (preferably symbolic links) to MySQL Cluster executables just as you would with any other executable program.

- b. Determine the location of the cluster global configuration file. The name and location of this file are arbitrary, but a typical value for these is `/var/lib/mysql-cluster/config.ini`.
- c. Invoke the management server with the `--config-file` or `-f` option pointing to the global configuration file. Using the typical file location given above and assuming that you have navigated to the `libexec` directory, this means that the management server would be invoked using either of the following commands in the system shell:

```
shell> ./ndb_mgmd --config-file=/var/lib/mysql-cluster/config.ini
```

or

```
shell> ./ndb_mgmd -f /var/lib/mysql-cluster/config.ini
```

### Note

The path used with the `--config-file` or `-f` option can be relative or absolute.

You can verify that the management server process is running by viewing the output of a system command such as this one:

```
shell> ps aux | grep ndb
jon 16547 17.3 0.7 8392 2480 ? Ssl 15:20 2:03
./ndb_mgmd -f /var/lib/mysql-cluster/config.ini
```

The command and output shown is for the `bash` shell running on a Linux PC; you should

## 6.2.1. Methods for Starting a MySQL Cluster

---

use whatever is supported by your operating system for this purpose.

You can also use the management client to determine whether or not the management server is running. Start the management client (using a connectstring if the management server is not running on `localhost:1186`), then try to execute a `SHOW` command. If the management server is running, the command should execute normally; otherwise, the command fails with the error message `Unable to connect with connectstring: ....` You can also use the `SHOW` command from the system shell by invoking `ndb_mgm` with the `--execute` or `-e` switch, as shown here:

```
shell> ndb_mgm -e "SHOW"
```

or

```
shell> ndb_mgm --execute="SHOW"
```

### Important

If you do not specify the location for the MySQL Cluster global configuration file, the management server will attempt to read configuration parameters from the `[mysql_cluster]` section of a `my.cnf` file on the management node host. While we use (and recommend) a separate Cluster configuration file for most examples in this Guide, there may be times when you prefer using a `my.cnf` file instead.

In any case, if the management server cannot find its configuration parameters, the result is an error, indicated by a message similar to this one:

```
Error line 0: At least one database node should be defined in config file
Unable to read config file
```

2. **Start the data nodes (ndbd processes).** For each MySQL Cluster data node, you must invoke the `ndbd` binary using the steps listed below. These must be performed on each data node host.
  - a. Locate the `ndbd` binary and navigate to this directory. Like `ndb_mgmd`, `ndbd` is typically found (when installed from the Linux `.tar.gz` archive) in the `bin` directory or, to be more precise, in `/usr/local/mysql/bin`. However, the exact location may vary according to the type of installation that was performed (when installing from source, for example, `ndbd` and `ndb_mgmd` are placed by default in `/usr/local/mysql/libexec`).
  - b. Start the data node process by invoking `ndbd`. Remember that `ndbd` must be given the hostnames or IP addresses of any management servers to which it is supposed to connect. This can be supplied to the data node process in either of two ways:
    - By including the hostname or address via the `ndb-connectstring` (`connect-string`) option in an `[ndbd]` or `[mysql_cluster]` section of a `my.cnf` configuration file that is local to the data node host (see Section 4.1.1, “Local Configuration Files”), in any one of the four ways shown here:

## 6.2.1. Methods for Starting a MySQL Cluster

---

i.

```
[ndbd]  
ndb-connectstring=10.0.0.1
```

ii.

```
[ndbd]  
connectstring=10.0.0.1
```

iii.

```
[mysql_cluster]  
ndb-connectstring=10.0.0.1
```

iv.

```
[mysql_cluster]  
connectstring=10.0.0.1
```

In this case, the data node can be started simply by invoking `ndbd`:

```
shell> ./ndbd
```

Here, we assume that you have navigated to the directory containing the `ndbd` executable. Otherwise, you need to include the path to `ndbd` when invoking it, or to create an appropriate link.

- By passing the cluster connectstring on the command line. You can use the `-connect-string` option, like this:

```
shell> ./ndbd --connect-string=10.0.0.1
```

You can also use the short form `-c` for this option, like this:

```
shell> ./ndbd -c 10.0.0.1
```

Connectstrings can also include ports, and be used to connect to multiple management servers. For the complete syntax, refer to Section 4.2, “The Cluster Connectstring”.

### Note

It is not necessary to use `&` when starting `ndbd`; by default, it runs as a daemon background process. If you wish to override this behavior, use the `--nodaemon` startup option. For more information, see Section 3.1.2, “Data Node Process (ndbd)”.

You can tell whether the data node process is started by checking the output of `ps` or a similar command, like this:

```
shell> ps aux | grep ndbd  
root 19000 0.0 0.4 7768 2028 ? Ss Sep02 0:00 ./ndbd -c 10.0.0.1 --initial  
root 19001 0.0 25.4 309384 114920 ? Sl Sep02 34:07 ./ndbd -c 10.0.0.1 --initial
```

(In this case, the node was started using the `--initial` option. More information about this option is provided later in this section.) Note that there are normally *two* `ndbd` processes

## 6.2.1. Methods for Starting a MySQL Cluster

---

per data node, one being the “angel” process discussed in Section 3.1.2, “Data Node Process (ndbd)”.

You can verify that the data node is connected to the cluster using the cluster management client (`ndb_mgm`), as described in Section 6.3, “Monitoring and Logging”.

When starting `ndbd`, you might see an error such as this one shown here:

```
shell> ./ndbd
Unable to connect with connect string: nodeid=0,localhost:1186
Retrying every 5 seconds. Attempts left: 12
```

This is because the `connectstring` was not passed to the data node process — either you forgot to specify the `connectstring` on the command line, or the data node process was unable to find a `connectstring` value set in a `my.cnf` file. In the latter case, this could be due to a missing configuration file, one with incorrect permissions, or the file contained one or more incorrect values for the management server hostname or IP address.

As mentioned previously, these steps given above must be repeated for *each* data node in the cluster.

The `--initial` option causes `ndbd` to rebuild the data node filesystem. That is, using this option causes the deletion and re-creation of all files in the node's `DataDir`. (See later in this section for two important exceptions to this.) You should *not* use this option when restarting a data node in an existing cluster, except under certain special circumstances, which include:

- When the data node filesystem becomes damaged or corrupted in some way, and the node will not start otherwise
- When you have replaced or reformatted the disk used by the data node for storing its filesystem
- As part of some (but not all) cluster software upgrades
- When restoring the cluster from a backup
- When making certain global configuration changes, such as changing the number of data nodes or replicas in the cluster

### Important

The `--initial` option does *not* remove Cluster backup files or Disk Data files. If you want to remove these files, do so manually on the filesystem level using an operating system command such as `rm`.

3. **Start the SQL nodes (`mysqld` processes).** Starting a MySQL server that acts as a MySQL Cluster SQL node is different in two main respects from starting the standard server:
  - a. The `mysqld` process must be started with the `ndbcluster` option, in order to activate support for the NDB storage engine. This can be done in either of two ways:
    - Passing the option as `--ndbcluster` on the command line to `mysqld`, or by passing

## 6.2.1. Methods for Starting a MySQL Cluster

---

it to a MySQL server startup script such as `mysqld_safe` or `mysql.server`.

- Adding `ndbcluster` to the `[mysqld]` section of a `my.cnf` file that can be read by the MySQL server process.

For example, if you wanted to use the `mysqld_safe` script to start the MySQL Server with NDB support activated, you could do so like this:

```
shell> ./mysqld_safe --ndbcluster &
```

The previous command as shown causes the `mysqld` process to attempt connecting to a management server at `localhost:1186` (the default). The next item discusses how to connect to management nodes on remote hosts.

- b. The MySQL server must be able to connect to a running management server. As with cluster data nodes, this is accomplished using a MySQL Cluster connectstring. You can do this by setting the `ndb-connectstring` option in a `[mysqld]` or `[mysql_cluster]` section of `my.cnf`.

You can also pass the connectstring to `mysqld` or to a MySQL startup script using the `--ndb-connectstring` option when starting MySQL from the command line, as shown here:

```
shell> ./mysqld_safe --ndb-connectstring=10.0.0.1 &
```

This causes the MySQL server to attempt connecting to a cluster management server whose IP address is `10.0.0.1`.

To pass both of the required options to `mysqld` on the command line, you could use something like this:

```
shell> ./mysqld_safe --ndbcluster --ndb-connectstring=10.0.0.1 &
```

Here, we have used the `mysqld_safe` startup script to start the MySQL Server with NDB support activated, and such that it attempts to connect to a management server running at `10.0.0.1`.

There are two ways to verify that the SQL node is connected to the cluster:

- a. Use the `SHOW` command in the management client. See Section 6.3.1.1, “The `SHOW`, `STATUS`, and `DUMP` Commands”, for a discussion of this command.
- b. Issue the statement `SHOW ENGINE NDB STATUS` in the MySQL Monitor. See Section 6.3.1.2, “The `SHOW ENGINE NDB STATUS` Statement”, where this statement is covered in more detail.

### Warning

No error is issued on the command line or in the MySQL error (`hostname.err`) log by a failure of `mysqld` to connect to the cluster. Furthermore, the output of `SHOW EN-`

---

## 6.2.1. Methods for Starting a MySQL Cluster

---

GINES displays YES in the support column for the `ndbcluster` row. You should use the management client `SHOW` command or the `SHOW ENGINE NDB STATUS` SQL statement to verify the SQL node's connection status for this reason.

### 6.2.1.2. Starting the Cluster Using the Management Client

It is also possible to start a MySQL Cluster with the help of the cluster management client. The main reason for starting the cluster in this way is to avoid having the startup process time out before all nodes have time to connect; if this happens, the cluster fails to start. This can happen when there are a large number of nodes in the cluster. Using the cluster management client can help you to avoid this problem.

Starting a cluster using the Cluster management client is similar to starting the cluster without it. The principal differences are that all `ndbd` instances are started with an option that causes them to wait for the management client `START` command before completing the startup process, and that an extra step (issuing the `START` command) is required before starting the SQL nodes. The main steps are:

1. **Start the management server.** This is done in exactly the same way as described in Section 6.2.1.1, “Starting the Cluster without the Management Client”.
2. **Start the data nodes with `--nostart`.** Each data node is started in the same way as described in the previous section, except that it is started with the `--nostart` (or `-n`) option:

```
shell> ./ndbd --nostart
```

or

```
shell> ./ndbd -n
```

This option can be and often is used in combination with other `ndbd` options such as `-c` or `--connect-string`. A more complete example is shown here:

```
shell> ./ndbd --nostart --connect-string=10.0.0.1
```

Using `--nostart` causes the `ndbd` process to connect to the management server and obtain configuration data from it. However, the data node is at this point not yet online — that is, it does not yet actually begin to process any data.

This can also be accomplished by including the `nostart` in the `[mysql_cluster]` or `[ndbd]` section of the data node's `my.cnf` file.

When a data node has been started in this way, its status as reported by the management client `SHOW` command will include `not started`, as shown here (omitting those portions of the output not relating to the data nodes):

```
ndb_mgm> SHOW
Cluster Configuration
-----
[ndbd(NDB)] 4 node(s)
id=2 @10.0.0.2 (Version: 5.1.24-ndb-6.2.16, not started)
```

## 6.2.1. Methods for Starting a MySQL Cluster

---

```
id=3 @10.0.0.3 (Version: 5.1.24-ndb-6.2.16, not started)
id=4 @10.0.0.4 (Version: 5.1.24-ndb-6.2.16, not started)
id=5 @10.0.0.5 (Version: 5.1.24-ndb-6.2.16, not started)
```

This is also true of the STATUS command:

```
ndb_mgm> ALL STATUS
Node 2: not started (Version 5.1.24-ndb-6.2.16)
Node 3: not started (Version 5.1.24-ndb-6.2.16)
Node 4: not started (Version 5.1.24-ndb-6.2.16)
Node 5: not started (Version 5.1.24-ndb-6.2.16)
```

A data node cannot be brought online using the management client unless `ndbd` was invoked using `--nostart` or `-n` from the command line, or using `nostart` in the data node's `my.cnf` file.

3. Once all data nodes have been started, the management client's `START` command is used to bring them online, as discussed in Section 3.1.1.2, “Management Client (`ndb_mgm`)”. You can do this in one of two ways:
  - a. Allow the Cluster management server to select the starting order by issuing the `ALL START` command in the management client:

```
ndb_mgm> ALL START
NDB Cluster is being started.
NDB Cluster is being started.
NDB Cluster is being started.
NDB Cluster is being started.

Node 2: Start initiated (version 5.1.24-ndb-6.2.16)
Node 3: Start initiated (version 5.1.24-ndb-6.2.16)
Node 4: Start initiated (version 5.1.24-ndb-6.2.16)
Node 5: Start initiated (version 5.1.24-ndb-6.2.16)

Node 2: Started (version 5.1.24-ndb-6.2.16)
Node 3: Started (version 5.1.24-ndb-6.2.16)
Node 4: Started (version 5.1.24-ndb-6.2.16)
Node 5: Started (version 5.1.24-ndb-6.2.16)
```

Usually this is sufficient.

- b. However, if you wish to bring the nodes online one at a time or in a specific order, use `node_id START` as shown in this example (in which the order is purely arbitrary):

```
ndb_mgm> 2 START
Database node 2 is being started.

Node 2: Start initiated (version 5.1.24-ndb-6.2.16)

Node 2: Started (version 5.1.24-ndb-6.2.16)

ndb_mgm> 4 START
Database node 4 is being started.

Node 4: Start initiated (version 5.1.24-ndb-6.2.16)

Node 4: Started (version 5.1.24-ndb-6.2.16)
```

## 6.2.1. Methods for Starting a MySQL Cluster

---

```
ndb_mgm> 5 START
Database node 5 is being started.

Node 5: Start initiated (version 5.1.24-ndb-6.2.16)

Node 5: Started (version 5.1.24-ndb-6.2.16)

ndb_mgm> 3 START
Database node 3 is being started.

Node 3: Start initiated (version 5.1.24-ndb-6.2.16)

Node 3: Started (version 5.1.24-ndb-6.2.16)
```

This is often required to perform a partial start of the cluster. (See Section 6.2.1.3, “Partial Starts”.)

4. Once all data nodes have been started in this way, you can proceed to start the cluster's SQL nodes exactly as described previously, in Section 6.2.1.1, “Starting the Cluster without the Management Client”.

### 6.2.1.3. Partial Starts

Sometimes it is desirable to perform a *partial start* of the cluster. For example, you may need to start the cluster even though one or more data node hosts are unavailable due to hardware or software issues. It is possible to start the cluster when one or more data nodes are not running. However, *there must be at least one “live” data node from each node group in order to form a viable cluster, even when performing a partial start.* As discussed in Section 2.2.3, “Nodes and Node Groups”, unless all node groups are represented, the cluster does not have a complete copy of all its data, and so is not viable.

Normally, the management server waits until all data nodes have been started before bringing the cluster online. You can override this behavior by starting each data node with the `--nowait-nodes` option, which is used to start the cluster without all data nodes being available. This option takes a comma-separated list of data nodes which for which the cluster does not wait before starting:

```
--nowait-nodes=node_id_1[, node_id_2[, ...]]
```

For example, suppose your cluster has 4 data nodes in 2 node groups, the data nodes having node IDs 2 through 5, and you wish to perform a partial start omitting node 3. Assuming that each data node is running on a different host, you could perform a partial start of the cluster by starting `ndb_mgmd` as normal, then invoking `ndbd` on each data node host as shown here:

```
# On the machine hosting node 2:
shell> ndbd --no-wait-nodes=3

# On the machine hosting node 4:
shell> ndbd --no-wait-nodes=3

# On the machine hosting node 5:
shell> ndbd --no-wait-nodes=3
```



## 6.2.2. Rolling Restarts

---

In this case, the cluster starts as soon as nodes 2, 4, and 5 connect, and does not wait `StartPartitionedTimeout` milliseconds for nodes 3 to connect as it would otherwise.

To perform a partial start of the same cluster with only nodes 2 and 4 — that is, omitting nodes 3 and 5, you would need to start nodes 2 and 4 as shown here:

```
# On the machine hosting node 2:
shell> ndbd --no-wait-nodes=3,5

# On the machine hosting node 4:
shell> ndbd --no-wait-nodes=3,5
```

To perform a partial initial start of the cluster, use the `--initial-start` option. This shows how the previous example would be modified to perform a partial initial start:

```
# On the machine hosting node 2:
shell> ndbd --no-wait-nodes=3,5 --initial-start

# On the machine hosting node 4:
shell> ndbd --no-wait-nodes=3,5 --initial-start
```

### Note

If multiple data nodes are being run on the same computer, you must also use the `--ndb-nodeid` option. (While it is true that deploying multiple data nodes on a single host is not currently supported for production environments, we include this information because you may find it useful for testing or related proposes.) Suppose that, there are 2 data node hosts, each configured to run 2 nodes so that nodes 2 and 4 run on one host, and nodes 3 and 5 run on the other. To perform a partial start of the cluster omitting node 3, you would use the following commands to start nodes 2, 4, and 5:

```
# On the machine hosting nodes 2 and 4:
shell> ndbd --ndb-nodeid=2 --no-wait-nodes=3
shell> ndbd --ndb-nodeid=4 --no-wait-nodes=3

# On the machine hosting nodes 3 and 5:
shell> ndbd --ndb-nodeid=5 --no-wait-nodes=3
```

## 6.2.2. Rolling Restarts

The term *rolling restart* refers to restarting all MySQL Cluster nodes in turn without shutting down the entire cluster. This technique is useful when you want to perform one of the following tasks while keeping the cluster itself running:

- Setting one or more configuration parameters to new values.

### Note

## 6.2.2. Rolling Restarts

---

Different Cluster configuration parameters may require individual nodes, all data nodes, or the entire cluster to be restarted before taking effect. In some cases, you may need to stop, then start nodes again; in others, it may be possible to update the appropriate configuration file and then restart the node.

You can find out which parameters require which restart types by consulting Section 4.3.4, “`config.ini` Parameters”.

- Upgrading the cluster to a newer version of the MySQL Cluster software (or downgrading it to an older version).
- To make changes in the hardware or operating system of the hosts on which one or more cluster nodes are running.
- Resetting the cluster because it has reached an undesirable state.
- Freeing memory allocated to a specific table by successive `INSERT` and `DELETE` operations on that table for re-use by other MySQL Cluster tables.

Restarting a data node can be accomplished using the management client `node_id` `RESTART` command, where `node_id` is the node ID. This can also be used to restart a management node. In this case, the management client loses the connection, but the next command (such as `CONNECT` or `SHOW`) automatically reconnects it. Alternatively, you can stop the management server using either the management client `STOP` command or an operating system command such as `kill`, then start it again from the command line. To restart an SQL node, you can use any of the standard methods for restarting a MySQL server.

Generally, the steps in performing a rolling restart of a MySQL Cluster are as follows:

1. Stop all cluster management nodes (`ndb_mgmd` processes), reconfigure them, then restart them
2. Stop, reconfigure, then restart each cluster data node (`ndbd` process) in turn
3. Stop, reconfigure, then restart each cluster SQL node (`mysqld` process) in turn

The exact steps for implementing a particular rolling upgrade depend upon the actual changes being made. As previously mentioned, a restart requires completely stopping then starting the node again, or you may need only to update the appropriate configuration file and then restart the node (using the management client `RESTART` command). The following diagram shows more specific information for different restart scenarios:

## 6.2.2. Rolling Restarts

RESTART TYPE:				
Cluster Configuration Change	Cluster Software Upgrade or Downgrade	Change on Node Host	Cluster Reset	
<b>A. Management node (<i>ndb_mgmd</i>) processes...</b>				
1. Stop all <i>ndb_mgmd</i> processes  2. Make changes in global configuration file(s)  3. Start all <i>ndb_mgmd</i> processes	1. Stop all <i>ndb_mgmd</i> processes  2. Replace each <i>ndb_mgmd</i> binary with new version  3. Start <i>ndb_mgmd</i> processes	1. Stop all <i>ndb_mgmd</i> processes  2. Make desired changes in hardware, operating system, or both  3. Start all <i>ndb_mgmd</i> processes	( OR )  1. Stop all <i>ndb_mgmd</i> processes  2. Start all <i>ndb_mgmd</i> processes	Restart all <i>ndb_mgmd</i> processes (optional)
<b>B. For each data node (<i>ndbd</i>) process...</b>				
( OR )  1. Stop <i>ndbd</i>  2. Start <i>ndbd</i>	Restart <i>ndbd</i>  1. Stop <i>ndbd</i>  2. Replace <i>ndbd</i> binary with new version  3. Start <i>ndbd</i>	1. Stop <i>ndbd</i>  2. Make desired changes in hardware, operating system, or both  3. Start <i>ndbd</i>	( OR )  1. Stop <i>ndbd</i>  2. Start <i>ndbd</i>	Restart <i>ndbd</i>
<b>C. For each SQL node (<i>mysqld</i>) process...</b>				
( OR )  1. Stop <i>mysqld</i>  2. Start <i>mysqld</i>	Restart <i>mysqld</i>  1. Stop <i>mysqld</i>  2. Replace <i>mysqld</i> binary with new version  3. Start <i>mysqld</i>	1. Stop <i>mysqld</i>  2. Make desired changes in hardware, operating system, or both  3. Start <i>mysqld</i>	( OR )  1. Stop <i>mysqld</i>  2. Start <i>mysqld</i>	Restart <i>mysqld</i>

In the previous diagram, *Stop* and *Start* indicate that the process must be stopped completely using a shell command (such as `kill`), the management client `STOP` command, or — in the case of an SQL node — `mysqladmin shutdown`, then started again from a system shell by invoking the `ndbd`, `ndb_mgmd`, or `mysqld` executable as appropriate. (In the case of `mysqld`, this is generally accomplished via a startup script such as `mysqld_safe`.) *Restart* indicates the process may be restarted using the `ndb_mgm` management client `RESTART` command, where applicable.

In some cases, depending on the type of configuration changes that have been made, data nodes may need to be restarted using the `--initial` option in order to rebuild the data node filesystems. You

---

## 6.3. Monitoring and Logging

---

can find out when this is necessary by consulting the MySQL Manual's *Overview of Cluster Configuration Parameters* [<http://dev.mysql.com/doc/refman/5.1/en/mysql-cluster-config-params-overview.html>].

## 6.3. Monitoring and Logging

**Description:** This section covers the use of the Cluster management client, the `ndb_config` utility, and the MySQL `SHOW ENGINE NDB STATUS` statement for monitoring node status, cluster status, and cluster configuration. It also discusses Cluster log files, their purposes and locations, and how to understand their contents.

### 6.3.1. Monitoring the Cluster

This section covers available methods for obtaining information about the current status of the cluster and cluster nodes, including the following topics:

- The management client commands `SHOW`, `STATUS`, and `DUMP` commands.
- The MySQL Server's `SHOW ENGINE NDB STATUS` statement

In addition, the `ndb_config` utility can be used to obtain cluster configuration data from the `config.ini` file. For more information about `ndb_config`, see Section 4.4, “Utility Programs”.

#### 6.3.1.1. The `SHOW`, `STATUS`, and `DUMP` Commands

The MySQL Cluster management client provides two commands — `SHOW` and `STATUS` — that can be used to monitor the status of the cluster. In addition, the `DUMP` command writes the values of node parameters to the cluster log.

**The `SHOW` command.** This command displays the status of each cluster node for which a node ID has been allocated in the global cluster configuration (`config.ini`) file.

The information provided by `SHOW` is organized by node ID, and includes:

- The IP address or hostname and the port number of the management server to which the current instance of `ndb_mgm` is connected.
- **Data nodes.** For all data nodes as a group the total number of data nodes in the cluster is shown. In addition, `SHOW` displays the IP address or hostname to which each data node is assigned, whether or not the node is connected to the cluster.

For each data node that is connected to the cluster, the `SHOW` command displays:

- The node ID
- The hostname or IP address of the host to which that node ID is assigned
- The MySQL Cluster software version
- The node group to which the node belongs

---

### 6.3.1. Monitoring the Cluster

---

In addition, the data node acting as the cluster's *master node* is indicated as such. The master node is used to provide a baseline copy of the cluster's table definitions against which the other data nodes compare their copies. The master node is also responsible for determining which data nodes create which portions of a cluster backup.

For each data node that is not connected to the cluster, a `not connected` message is displayed.

- **Management nodes.** The total number of management nodes for which node IDs are allocated in `config.ini` is always displayed.

For connected management servers, the IP address or hostname and the MySQL Cluster software version is displayed.

For management servers which are configured in the `config.ini` file but are not running, the status message `not connected; accepting connect from hostname` is displayed.

- **API nodes.** The total number of API nodes for which node IDs are allocated in `config.ini` is always displayed.

For connected API nodes, the IP address or hostname from which the node has connected and the MySQL Cluster software version is displayed.

For API nodes which are configured in `config.ini` but are not running, the message `not connected; accepting connect from hostname` is displayed if a host for this API node has been specified in the configuration file. If no host is specified, then the status message reads `not connected, accepting connect from any host`.

The following example demonstrates how this command is used, and displays some sample output:

```
ndb_mgm> SHOW
Connected to Management Server at: 10.0.0.1:1186
Cluster Configuration
-----
[ndbd(NDB)] 4 node(s)
id=2   @10.0.0.2 (Version: 5.1.24-ndb-6.2.16, Nodegroup: 0)
id=3   @10.0.0.3 (Version: 5.1.24-ndb-6.2.16, Nodegroup: 0)
id=4   @10.0.0.4 (Version: 5.1.24-ndb-6.2.16, Nodegroup: 1)
id=5   @10.0.0.5 (Version: 5.1.24-ndb-6.2.16, Nodegroup: 1, Master)

[ndb_mgmd(MGM)] 1 node(s)
id=1   @10.0.0.1 (Version: 5.1.24-ndb-6.2.16)

[mysqld(API)] 2 node(s)
id=6   @10.0.0.6 (Version: 5.1.24-ndb-6.2.16)
id=7   (not connected, accepting connect from any host)
```

This tells us that:

- All four data nodes configured for this cluster are connected and active
- There are two node groups — data nodes 2 and 3 belong to one node group, while data nodes 4 and 5 belong to the other

### 6.3.1. Monitoring the Cluster

---

- The data node running on the host at 10.0.0.5 (node 5) is acting as the master
- The single management server configured for this cluster is running on the host at 10.0.0.1
- Of the two API nodes allocated to this cluster, one is connected (from IP address 10.0.0.6), and another may connect to the cluster from any hostname or IP address

**The STATUS command.** This `ndb_mgm` command can be used to obtain limited status information about one or all data nodes connected to the cluster — unlike `SHOW`, it displays only the connection status and the MySQL Cluster software version. As with `START` and `RESTART`, when invoking this command, you must precede it with either a data node ID or the keyword `ALL`:

```
{node_id | ALL} STATUS
```

To display status information about a single node, use that node's ID, as shown here:

```
ndb_mgm> 1 STATUS
Node 1: started (Version 5.1.24-ndb-6.2.16)

ndb_mgm> 4 STATUS
Node 4: started (Version 5.1.24-ndb-6.2.16)

ndb_mgm> 6 STATUS
Node 6: started (Version 5.1.24-ndb-6.2.16)
```

To display status information about all data nodes in the cluster, use the `ALL` keyword as shown here:

```
ndb_mgm> ALL STATUS
Node 2: started (Version 5.1.24-ndb-6.2.16)
Node 3: started (Version 5.1.24-ndb-6.2.16)
Node 4: started (Version 5.1.24-ndb-6.2.16)
Node 5: started (Version 5.1.24-ndb-6.2.16)
```

It is important to remember that `ALL STATUS` displays information about data nodes only, but that `node_id STATUS` can be used to check the connection status of any node in the cluster, regardless of its type.

#### Note

In some MySQL Cluster releases, you might see output such as this for the `STATUS` command, where `Version` is replaced with the prefix `mysql-`:

```
ndb_mgm> 1 STATUS
Node 1: started (mysql-5.1.24-ndb-6.2.16)
```

However, the essential parts of the output (the node's connection status and software version number) remain the same as shown previously.

**The DUMP command.** This command prints state information for one or more nodes to the cluster log. Its syntax is shown here:

```
{node_id|ALL} DUMP code [arguments]
```

---

### 6.3.1. Monitoring the Cluster

---

As with `START`, `STOP`, and many other management client commands, the `DUMP` keyword is preceded either by the ID of the node for which information is to be written to the cluster log or by `ALL`, in which case information from all nodes will be logged. *code* is a numeric code, which is sometimes followed by one or more *arguments*, depending on the function performed by the code that is used. The only `DUMP` code we cover here is that used for generating a report of data node memory usage, as shown here for a cluster with two data nodes:

```
ndb_mgm> ALL DUMP 1000
Sending dump signal with data:
0x000003e8 Sending dump signal with data:
0x000003e8
```

To see the results of this command, it is necessary to examine the last few lines of the cluster log:

```
ndb_mgm> EXIT
shell> tail -n 6 ndb_1_cluster.log
2007-02-14 03:52:09 [MgmSrvr] INFO -- Node 2: Data usage is 2%(52 32K pages of total 2560)
2007-02-14 03:52:09 [MgmSrvr] INFO -- Node 2: Index usage is 0%(22 8K pages of total 2336)
2007-02-14 03:52:09 [MgmSrvr] INFO -- Node 2: Resource 0 min: 0 max: 639 curr: 0
2007-02-14 03:52:09 [MgmSrvr] INFO -- Node 3: Data usage is 2%(68 32K pages of total 2560)
2007-02-14 03:52:09 [MgmSrvr] INFO -- Node 3: Index usage is 0%(22 8K pages of total 2336)
2007-02-14 03:52:09 [MgmSrvr] INFO -- Node 3: Resource 0 min: 0 max: 639 curr: 0
```

The output includes per-node usage data for `DataMemory` and `IndexMemory`, which are displayed both as a number of 32K memory pages and as a percentage of the total amount available. Also included are per-node *resource group reports*. In MySQL 5.1, there are 3 resource groups:

- **Group 0.** This is an aggregate of all resource groups.
- **Group 1.** This group includes disk operation resources such as `MaxNoOfConcurrentOperations` and `MaxNoOfConcurrentScans` which are needed to perform Disk Data transactions.
- **Group 2.** These include disk records and memory used for Disk Data objects such as tablespaces and log file groups.

Additional resource groups are expected to be implemented in the future. No reports for resource groups 1 and 2 are displayed if the cluster contains no Disk Data objects.

Other `DUMP` commands are covered in the *MySQL Cluster API Developer's Guide*.

#### 6.3.1.2. The `SHOW ENGINE NDB STATUS` Statement

`SHOW ENGINE NDB STATUS` is an SQL statement that can be used to obtain some basic information about a MySQL Cluster to which a MySQL Server is connected as an SQL node. This statement is invoked in the MySQL Monitor as shown here:

```
mysql> SHOW ENGINE NDB STATUS;
```

This statement takes no additional clauses or parameters.

The result set returned by this statement when run on an SQL node connected to a typical 4-node MySQL Cluster contains 14 rows (discussed later in this section) and three columns (`Type`, `Name`, and `Status`) as shown here (note that we have wrapped the output of some of the columns due to

### 6.3.1. Monitoring the Cluster

---

considerations of space):

```
mysql> SHOW ENGINE NDB STATUS\G
***** 1. row *****
  Type: ndbcluster
  Name: connection
Status: cluster_node_id=6,
        connected_host=10.0.0.1,
        connected_port=1186,
        number_of_data_nodes=4,
        number_of_ready_data_nodes=4,
        connect_count=4
***** 2. row *****
  Type: ndbcluster
  Name: NdbTransaction
Status: created=4, free=0, sizeof=212
***** 3. row *****
  Type: ndbcluster
  Name: NdbOperation
Status: created=8, free=8, sizeof=660
***** 4. row *****
  Type: ndbcluster
  Name: NdbIndexScanOperation
Status: created=1, free=1, sizeof=744
***** 5. row *****
  Type: ndbcluster
  Name: NdbIndexOperation
Status: created=0, free=0, sizeof=664
***** 6. row *****
  Type: ndbcluster
  Name: NdbRecAttr
Status: created=1285, free=1285, sizeof=60
***** 7. row *****
  Type: ndbcluster
  Name: NdbApiSignal
Status: created=16, free=16, sizeof=136
***** 8. row *****
  Type: ndbcluster
  Name: NdbLabel
Status: created=0, free=0, sizeof=196
***** 9. row *****
  Type: ndbcluster
  Name: NdbBranch
Status: created=0, free=0, sizeof=24
***** 10. row *****
  Type: ndbcluster
  Name: NdbSubroutine
Status: created=0, free=0, sizeof=68
***** 11. row *****
  Type: ndbcluster
  Name: NdbCall
Status: created=0, free=0, sizeof=16
***** 12. row *****
  Type: ndbcluster
  Name: NdbBlob
Status: created=0, free=0, sizeof=264
***** 13. row *****
  Type: ndbcluster
  Name: NdbReceiver
Status: created=4, free=0, sizeof=68
***** 14. row *****
```



---

### 6.3.1. Monitoring the Cluster

---

```
Type: ndbcluster
Name: binlog
Status: latest_epoch=97316,
        latest_trans_epoch=5739,
        latest_received_binlog_epoch=0,
        latest_handled_binlog_epoch=0,
        latest_applied_binlog_epoch=0
14 rows in set (0.00 sec)
```

The value displayed in the `Type` column is `ndbcluster` for all rows. The most important of these are the first and last rows, named `connection` and `binlog`, respectively. The string displayed in the `Status` column for each row contains a comma-delimited list of name-value pairs. We examine the name and values appearing in these two rows here:

1. **connection row.** The `Status` column of this row contains information relating to the management server and the number of nodes connected to the cluster. The name-value pairs displayed in that column, and their meanings, are shown here:
  - `cluster_node_id`: The node ID of the SQL node on which the `SHOW ENGINE NDB STATUS` statement was issued.
  - `connected_host`: The hostname or IP address of the management server to which the SQL is connected.
  - `connected_port`: The number of the port on which the management server (`connected_host`) is running
  - `number_of_data_nodes`: The total number of data nodes configured for the cluster.
  - `number_of_ready_data_nodes`: The number of data nodes which are connected to and active in the cluster.
  - `connect_count`: The number of data nodes connected to the cluster. This number can be greater than `number_of_ready_data_nodes` if one or more data nodes are not running (for example, if a data node was stopped using the management client `STOP` command after the cluster was started).
2. **binlog row.** The `Status` column in this row contains information relating primarily to cluster external replication. For a more detailed explanation of the `binlog` values, see Chapter 10, *MySQL Cluster External Replication*.

The remaining twelve rows in the output of `SHOW ENGINE NDB STATUS` are not covered in the certification exam; however, we list here by Name those which are most likely to be useful:

- `NdbTransaction`: The number and size of `NdbTransaction` objects that have been created. An `NdbTransaction` is created each time a table schema operation (such as `CREATE TABLE` or `ALTER TABLE`) is performed on an NDB table.
- `NdbOperation`: The number and size of `NdbOperation` objects that have been created.
- `NdbIndexScanOperation`: The number and size of `NdbIndexScanOperation` objects that have been created.
- `NdbIndexOperation`: The number and size of `NdbIndexOperation` objects that have been created.

---

### 6.3.2. Understanding and Using Cluster Logs

---

- `NdbRecAttr`: The number and size of `NdbRecAttr` objects that have been created. In general, one of these is created each time a data manipulation statement is performed by an SQL node.
- `NdbBlob`: The number and size of `NdbBlob` objects that have been created. An `NdbBlob` is created for each new operation involving a BLOB column in an NDB table.
- `NdbReceiver`: The number and size of any `NdbReceiver` object that have been created. The number in the `created` column is the same as the number of data nodes in the cluster to which the MySQL server has connected.

#### Note

`SHOW ENGINE NDBCLUSTER STATUS` is a synonym for `SHOW ENGINE NDB STATUS`.

If MySQL Cluster support is enabled for the MySQL Server — that is, if the `mysqld` process was started with `--ndbcluster` — but it is not connected to a cluster management server, then the value for each of the parameters in both rows is 0.

If MySQL Cluster support is not enabled for `mysqld`, then the output is as shown here:

```
mysql> SHOW ENGINE NDB STATUS;  
ERROR 1286 (42000): Unknown table engine 'NDB'
```

### 6.3.2. Understanding and Using Cluster Logs

This section discusses the cluster log and log events. It also covers the classification of log events according to level, category, and severity.

Topics in this section include:

- The definition of *cluster log event*
- Event categories, thresholds, and severity levels
- How an event is reported in the cluster log, and how to interpret a log event report
- The management client commands `CLUSTERLOG ON`, `CLUSTERLOG OFF`, and `CLUSTERLOG INFO`
- How to set logging categories and thresholds
- How to set logging severity levels

#### 6.3.2.1. Event Logs and Log Event Types

In this section, we discuss the types of event logs that are provided by MySQL Cluster, and the types of events that are recorded in these logs.

MySQL Cluster provides two types of event logs. These are:

---

## 6.3.2. Understanding and Using Cluster Logs

---

- The *cluster log*, which includes events generated by all cluster nodes
- The *node logs*, which are local to each data node

Both types of logs are plain text files that can be viewed with standard Unix utilities such as `more` or any text editor. Unix `tail` is also very useful for viewing the most recent content of log files.

Output generated by cluster event logging can have one or more destinations including a file, the management server console window, or `syslog`. By default, the cluster event log (often referred to as “the cluster log”) is written to the file `ndb_nodeid_cluster.log`, and node event logs are written to `ndb_nodeid_out.log`, where *nodeid* is the node ID of the node in question. Both types of event log are kept in the node's `DataDir`.

Both types of event logs can be made to record different subsets of events.

### Note

It is highly recommended that you monitor the event logs regularly. The cluster log is likely to be more useful than the node logs, since it provides logging information for an entire cluster in a single file. Node logs are intended to be used only during application development, or for debugging application code. For this reason, we will concentrate on the cluster log.

Each reportable event can be distinguished according to three different criteria:

- **Category.** This can be any one of the following values: `STARTUP`, `SHUTDOWN`, `STATISTICS`, `CHECKPOINT`, `NODERESTART`, `CONNECTION`, `ERROR`, or `INFO`.
- **Priority.** This is represented by one of the numbers from 1 to 15 inclusive, where 1 indicates “most important” and 15 “least important”.
- **Severity Level.** This can be any one of the following values: `ALERT`, `CRITICAL`, `ERROR`, `WARNING`, `INFO`, or `DEBUG`.

Both the cluster log and the node log can be filtered on these properties.

### 6.3.2.2. Logging Management Commands

The following management client commands are related to the cluster log:

- `CLUSTERLOG ON`  
Turns the cluster log on.
- `CLUSTERLOG OFF`  
Turns the cluster log off.
- `CLUSTERLOG INFO`

## 6.3.2. Understanding and Using Cluster Logs

---

This command provides information about cluster log severity level settings. An example of this command's usage and output is shown here:

```
ndb_mgm< CLUSTERLOG INFO
Connected to Management Server at: 10.0.0.1:1186
Severities enabled: INFO WARNING ERROR CRITICAL ALERT
```

- `node_id CLUSTERLOG category=threshold`

This causes *category* events with a priority less than or equal to *threshold* to be recorded in the cluster log.

- `CLUSTERLOG FILTER severity_level`

This command toggles the logging of events with the specified *severity\_level*.

The following table shows the default settings (for all data nodes) of the cluster log category threshold. If an event has a priority with a value lower than or equal to the priority threshold shown, then it is reported in the cluster log.

Note that events are reported per data node, and that the threshold can be set to different values on different nodes.

Category	Default threshold (All data nodes)
STARTUP	7
SHUTDOWN	7
STATISTICS	7
CHECKPOINT	7
NODERESTART	7
CONNECTION	7
ERROR	15
INFO	7

Thresholds are used to filter events within each category. For example, a `STARTUP` event with a priority of 3 is not logged unless the threshold for `STARTUP` is changed to 3 or higher. Only events with priority 3 or lower are sent if the threshold is 3.

The following table shows the event severity levels.

1	ALERT	A condition that should be corrected immediately, such as a corrupted system database
---	-------	---

## 6.3.2. Understanding and Using Cluster Logs

2	CRITICAL	Critical conditions, such as device errors or insufficient resources
3	ERROR	Conditions that should be corrected, such as configuration errors
4	WARNING	Conditions that are not errors, but that might require special handling
5	INFO	Informational messages
6	DEBUG	Debugging messages used for NDB Cluster development

### Note

Except for LOG\_EMERG and LOG\_NOTICE, these severity levels correspond to Unix syslog levels.

Event severity levels can be turned on or off using `CLUSTERLOG FILTER`, as discussed previously. If a severity level is turned on, then all events with a priority less than or equal to the category thresholds are logged. If the severity level is turned off then no events belonging to that severity level are logged.

### 6.3.2.3. Log Events

An event report reported in an event log has the following format:

```
datetime [string] severity -- message
```

For example:

```
2006-11-15 21:25:19 [MgmSrvr] INFO -- Node 4: Local checkpoint 2557 started.  
Keep GCI = 3085924 oldest restorable GCI = 2967870
```

This section discusses all reportable events, ordered by category and severity level within each category.

In the event descriptions, GCP and LCP mean “Global Checkpoint” and “Local Checkpoint”, respectively.

**CONNECTION Events.** These events are associated with connections between Cluster nodes:

Event	Priority	Severity Level	Description
data nodes connected	8	INFO	Data nodes connected
data nodes disconnected	8	INFO	Data nodes disconnected
Communication closed	8	INFO	SQL node or data node connection closed

### 6.3.2. Understanding and Using Cluster Logs

Communication opened	8	INFO	SQL node or data node connection opened
----------------------	---	------	---

**CHECKPOINT Events.** The logging messages shown here are associated with checkpoints:

Event	Priority	Severity Level	Description
LCP stopped in calc keep GCI	0	ALERT	LCP stopped
Local checkpoint fragment completed	11	INFO	LCP on a fragment has been completed
Global checkpoint completed	10	INFO	GCP finished
Global checkpoint started	9	INFO	Start of GCP: REDO log is written to disk
Local checkpoint completed	8	INFO	LCP completed normally
Local checkpoint started	7	INFO	Start of LCP: data written to disk
Report undo log blocked	7	INFO	UNDO logging blocked; buffer near overflow

**STARTUP Events.** The following events are generated in response to the startup of a node or of the cluster and of its success or failure.

Event	Priority	Severity Level	Description
Internal start signal received ST-TORRY (that is a response indicating that indicating the startup signal was received)	15	INFO	Blocks received after completion of restart
Undo records executed	15	INFO	
New REDO log started	10	INFO	GCI keep <i>X</i> , newest restorable GCI <i>Y</i>
New log started	10	INFO	Log part <i>X</i> , start MB <i>Y</i> , stop MB <i>Z</i>
Node has been refused for inclusion in the cluster	8	INFO	Node cannot be included in cluster due to misconfiguration, inability to establish communication, or other problem
data node neighbors	8	INFO	Shows neighboring data nodes
data node start phase <i>X</i> completed	4	INFO	A data node start phase has been completed
Node has been successfully included into the cluster	3	INFO	Displays the node, managing node, and dynamic ID

### 6.3.2. Understanding and Using Cluster Logs

data node start phases initiated	1	INFO	NDB Cluster nodes starting
data node all start phases completed	1	INFO	NDB Cluster nodes started
data node shutdown initiated	1	INFO	Shutdown of data node has commenced
data node shutdown aborted	1	INFO	Unable to shut down data node normally

STARTUP events also provide information relating to the progress of the startup process, including information concerning logging activities.

**NODERESTART Events.** The following events are generated when restarting a node, and relate to the success or failure of the node restart process:

Event	Priority	Severity Level	Description
Node failure phase completed	8	ALERT	Reports completion of node failure phases
Node has failed, node state was <i>X</i>	8	ALERT	Reports that a node has failed
Report arbitrator results	2	ALERT	There are eight different possible results for arbitration attempts: <ul style="list-style-type: none"><li>• Arbitration check failed — less than 1/2 nodes left</li><li>• Arbitration check succeeded — node group majority</li><li>• Arbitration check failed — missing node group</li><li>• Network partitioning — arbitration required</li><li>• Arbitration succeeded — affirmative response from node <i>X</i></li><li>• Arbitration failed — negative response from node <i>X</i></li><li>• Network partitioning — no arbitrator available</li><li>• Network partitioning — no arbitrator configured</li></ul>
Completed copying a fragment	10	INFO	
Completed copying of dictionary information	8	INFO	

### 6.3.2. Understanding and Using Cluster Logs

Completed copying distribution information	8	INFO	
Starting to copy fragments	8	INFO	
Completed copying all fragments	8	INFO	
GCP takeover started	7	INFO	
GCP takeover completed	7	INFO	
LCP takeover started	7	INFO	
LCP takeover completed (state = X)	7	INFO	
Report whether an arbitrator is found or not	6	INFO	<p>There are seven different possible outcomes when seeking an arbitrator:</p> <ul style="list-style-type: none"><li>• Management server restarts arbitration thread [state=X]</li><li>• Prepare arbitrator node X [ticket=Y]</li><li>• Receive arbitrator node X [ticket=Y]</li><li>• Started arbitrator node X [ticket=Y]</li><li>• Lost arbitrator node X - process failure [state=Y]</li><li>• Lost arbitrator node X - process exit [state=Y]</li><li>• Lost arbitrator node X <i>error_msg</i> [state=Y]</li></ul>

**STATISTICS Events.** Events of this type are of a statistical nature. They provide information such as numbers of transactions and other operations, amount of data sent or received by individual nodes, and memory usage.

Event	Priority	Severity Level	Description
Report job scheduling statistics	9	INFO	Mean internal job scheduling statistics
Sent number of bytes	9	INFO	Mean number of bytes sent to node X
Received # of bytes	9	INFO	Mean number of bytes received from node X
Report transaction statistics	8	INFO	Numbers of: transactions, commits, reads, simple reads, writes, concurrent operations, attribute information, and aborts



### 6.3.2. Understanding and Using Cluster Logs

Report operations	8	INFO	Number of operations
Report table create	7	INFO	
Memory usage	5	INFO	Data and index memory usage (80%, 90%, and 100%)

**ERROR Events.** These events relate to Cluster errors and warnings. The presence of one or more of these generally indicates that a major malfunction or failure has occurred, and should be investigated further as soon as possible. In particular, events having severity level ALERT or ERROR should be investigated and rectified immediately.

Event	Priority	Severity	Description
Dead due to missed heartbeat	8	ALERT	Node <i>X</i> declared “dead” due to missed heartbeat
Transporter errors	2	ERROR	
Transporter warnings	8	WARN- ING	
Missed heartbeats	8	WARN- ING	Node <i>X</i> missed heartbeat # <i>Y</i>
General warning events	2	WARN- ING	

**INFO Events.** These events provide general information about the state of the cluster and activities associated with Cluster maintenance, such as logging and heartbeat transmission.

Event	Priority	Severity	Description
Sent heartbeat	12	INFO	Heartbeat sent to node <i>X</i>
Create log bytes	11	INFO	Log part, log file, MB
General information events	2	INFO	

#### 6.3.2.4. Using the `CLUSTERLOG STATISTICS` Command

The MySQL Cluster management client's `CLUSTERLOG STATISTICS` command can provide a number of useful statistics in its output. The following statistics are reported by the transaction coordinator:

Statistic	Description (Number of...)
-----------	----------------------------

### 6.3.2. Understanding and Using Cluster Logs

---

Trans. Count	Transactions attempted with this node as coordinator
Commit Count	Transactions committed with this node as coordinator
Read Count	Primary key reads (all)
Simple Read Count	Primary key reads reading the latest committed value
Write Count	Primary key writes (includes all INSERT, UPDATE, and DELETE operations)
AttrInfoCount	Data words used to describe all reads and writes received
Concurrent Operations	All concurrent operations ongoing at the moment the report is taken
Abort Count	Transactions with this node as coordinator that were aborted
Scans	Scans (all)
Range Scans	Index scans

The `ndbd` process has a scheduler that runs in a continuous loop. During each loop, the scheduler performs the following tasks:

1. Read any incoming messages from sockets into a job buffer.
2. Check whether there are any timed messages to be executed; if so, put these into the job buffer as well.
3. Execute (in a loop) any messages in the job buffer.
4. Send any distributed messages that were generated by executing the messages in the job buffer.
5. Wait for any new incoming messages.

The number of loops executed in the third step is reported as the `Mean Loop Counter`. This statistic increases in size as the utilization of the TCP/IP buffer improves. You can use this to monitor performance as you add new processes to the cluster.

The `Mean send size` and `Mean receive size` statistics allow you to gauge the efficiency of writes and reads (respectively) between nodes. These values are given in bytes. Higher values mean a lower cost per byte sent or received; the maximum is 64k.

To generate a report of all cluster log statistics, you can use the following command in `ndb_mgm`:

```
ndb_mgm> ALL CLUSTERLOG STATISTICS=15
Executing CLUSTERLOG STATISTICS=15 on node 2 OK!
Executing CLUSTERLOG STATISTICS=15 on node 3 OK!
Executing CLUSTERLOG STATISTICS=15 on node 4 OK!
Executing CLUSTERLOG STATISTICS=15 on node 5 OK!
```

The response from the management client should be similar to that shown, with a confirmation mes-

sage printed for each data node in the cluster.

### 6.4. Single User Mode

The management client can also be used to place the cluster in *single user mode*, in which only a single API node may connect to the cluster. There are two management client commands relating to single user mode:

- ENTER SINGLE USER MODE *node\_id*

This command causes the cluster to enter single user mode, whereby only the MySQL server (or other API node) identified by the node ID *node\_id* is permitted to access the cluster. You will see a confirmation when the command is accepted, as shown in this example:

```
ndb_mgm> ENTER SINGLE USER MODE 6
Single user mode entered
Access is granted for API node 6 only.
```

You can verify that the cluster is in single user mode by examining the lines relating to data nodes in the output of the SHOW or STATUS commands, as shown here:

```
ndb_mgm> SHOW
Cluster Configuration
-----
[ndbd(NDB)]      4 node(s)
id=2      @10.0.0.2  (Version: 5.1.24-ndb-6.2.16, single user mode, Nodegroup: 0)
id=3      @10.0.0.3  (Version: 5.1.24-ndb-6.2.16, single user mode, Nodegroup: 0)
id=4      @10.0.0.4  (Version: 5.1.24-ndb-6.2.16, single user mode, Nodegroup: 1)
id=5      @10.0.0.5  (Version: 5.1.24-ndb-6.2.16, single user mode, Nodegroup: 1, Master)

[ndb_mgmd(MGM)]  1 node(s)
id=1      @10.0.0.1  (Version: 5.1.24-ndb-6.2.16)

[mysqld(API)]    4 node(s)
id=6      @10.0.0.6  (Version: 5.1.24-ndb-6.2.16)
id=7      @10.0.0.10 (Version: 5.1.24-ndb-6.2.16)

ndb_mgm> ALL STATUS
Node 2: single user mode (Version 5.1.24-ndb-6.2.16)
Node 3: single user mode (Version 5.1.24-ndb-6.2.16)
Node 4: single user mode (Version 5.1.24-ndb-6.2.16)
Node 5: single user mode (Version 5.1.24-ndb-6.2.16)
```

- EXIT SINGLE USER MODE

This command causes the cluster to exit single user mode, allowing all API nodes once again to access the cluster:

```
ndb_mgm> EXIT SINGLE USER MODE
Exiting single user mode in progress.
Use ALL STATUS or SHOW to see when single user mode has been exited.
```

---

## 6.5. Cluster Management — Exercises

---

Single user mode is especially useful for making backups and restoration of the cluster from a backup:

- When making a backup using `mysqldump`, access to the cluster can be restricted to the SQL node to which `mysqldump` connects, which can help guarantee a consistent backup.
- When restoring the cluster using `ndb_restore`, single user mode can be used to prevent any other NDB clients from connecting to the cluster.
- When restoring from a dump made using `mysqldump`, single user mode can be used to restrict access to the single MySQL server that is loading the data from the dump file.

For more information about backing up a MySQL Cluster and restoring it from backup, see Chapter 9, *Cluster Backup and Recovery*.

It is also advisable to employ single user mode when performing schema changes on NDB tables, due to the fact performing an `ALTER TABLE` while transactions are taking place can lead to table corruption. Doing so also minimizes the chances for conflicting data definition statements to be issued concurrently.

The term “single user mode” can be somewhat misleading, and it might be better for you to think of it as “single API mode”. It is important to remember that MySQL Cluster's single user mode does nothing to prevent multiple connections to `mysqld` or other cluster applications that allow multiple clients to connect.

In other words, while only one SQL node (MySQL server) may be connected to the cluster, it is still possible for multiple MySQL users to connect to this MySQL server. To make sure that this is not the case, check the output of `SHOW FULL PROCESSLIST`.

To prevent remote clients from connecting to a MySQL server, you can start `mysqld` with the `--skip-networking` option. Use of this option does not prevent MySQL from connecting to a MySQL Cluster. For example, you can start the server like this:

```
shell> mysqld_safe --ndbcluster »  
--ndb-connectstring=nodeid=6,10.0.0.1 --skip-networking &
```

After starting MySQL as shown, you can then execute this statement in the `ndb_mgm` client:

```
ndb_mgm> ENTER SINGLE USER MODE 6
```

Now, only a single SQL node (the `mysqld` instance that was just started) can access the cluster, and no remote MySQL clients can connect to that SQL node.

Single user mode does *not* prevent multiple instances of `ndb_mgm` from connecting to the cluster. See Section 6.1, “Management Client Basics”.

## 6.5. Cluster Management — Exercises

## 6.5. Cluster Management — Exercises

---

These are sample questions and answers for the *Cluster Management* Chapter.

Question 1:

The Cluster Management Client is always required to start a MySQL Cluster:

- a. True
- b. False

Question 2:

A connectstring does not need to be specified for cluster processes that run on the same server as the cluster's management node:

- a. True
- b. False

Question 3:

When do data nodes contact the management node for configuration details? [*check all that apply*]:

- a. At startup
- b. At regular intervals, while the cluster is running
- c. When the cluster shuts down
- d. Never — the data nodes communicate amongst themselves for configuration data

Question 4:

The management server process starts with the command:

- a. `ndb_mgm`
- b. `ndb_mgmd`
- c. `mysqld`
- d. `ndbd`

Question 5:

By default, how long does a data node wait after `ndbd` is invoked for the entire cluster to start?:

- a. 30 seconds
- b. 60 seconds
- c. 1440 seconds
- d. Indefinitely

## 6.5. Cluster Management — Exercises

---

Question 6:

Which configuration parameter changes the amount of time that a data node will wait for the cluster to start after `ndbd` is invoked?

- a. `TimeBetweenGlobalCheckpoints`
- b. `--initial`
- c. `StartPartialTimeout`
- d. This value cannot be changed

Question 7:

If a cluster configuration file is not specified, then which section of `my.cnf` is used for MySQL Cluster configuration parameters? *[check all that apply]*

- a. `[cluster]`
- b. `[mysql_cluster]`
- c. `[ndb_cluster]`
- d. `[mysqld]`

Question 8:

Which of the following are methods to determine if a SQL node is connected to a specific cluster? *[check all that apply]*:

- a. Issuing the `SHOW` command in the management client
- b. Issuing the `SHOW NODE` command in the management client
- c. Issuing the statement `SHOW ENGINES` from the MySQL command prompt
- d. Issuing the statement `SHOW ENGINE NDB STATUS` in the MySQL Monitor

Question 9:

`ndb_mgm` can accept `connectstring` parameters using `--connect-string`.

- a. True
- b. False

Question 10:

If `ndb_mgm` is unable to connect to the management server, you can get further details with the command `SHOW WARNINGS`.

- a. True

## 6.5. Cluster Management — Exercises

---

- b. False

Question 11:

What commands are used to exit from the management client? [*check all that apply*]:

- a. STOP
- b. EXIT
- c. QUIT
- d. TERM

Question 12:

What commands are used to start the cluster management server process from the management client? [*check all that apply*]:

- a. START
- b. BEGIN
- c. INIT
- d. There is no command to start the management server process from the management client.

Question 13:

Restarting all the cluster nodes in turn without first shutting down the entire cluster is known as a:

- a. Warm reboot
- b. Cold reboot
- c. Rolling restart
- d. Hot cycle

Question 14:

Which of the following are valid management client commands displaying information about data nodes? [*check all that apply*]:

- a. SHOW ALL
- b. ALL SHOW
- c. STATUS ALL
- d. ALL STATUS
- e. None of the above

Question 15:

## 6.5. Cluster Management — Exercises

---

What is the function of `ndb_config`? [*check all that apply*]:

- a. Configure any data node in the cluster
- b. Configure any management node in the cluster
- c. Configure any SQL node in the cluster
- d. Extract configuration information about data nodes and API nodes

Question 16:

By default, what is the name of the file that cluster event information is written to? [*check all that apply*]:

- a. `ndb_nodeid_cluster.log`
- b. `cluster_nodeid_ndb.log`
- c. `hostname.err`
- d. `/var/log/messages`

### Answers to Exercises

Answer 1:

*False.* A cluster can be started without the aid of the cluster management client.

Answer 2:

*True.* The `connectstring`'s default value is `localhost:1186`, which allows cluster processes to connect to a management node running on the same host. This is true for `mysqld`, `ndb_mgm`, and even of `ndbd` (although running a data node and a management node on the same host is not recommended in a production setting).

Note that, in the case of a cluster having multiple management nodes, all of them must be specified in a cluster process `connectstring` even if it is running on the same host as one of the management servers.

Answer 3:

**a** (*At startup*).

Answer **b** (*At regular intervals*) is incorrect because once the cluster is up and running, the data nodes no longer need the management node to coordinate traffic. In fact, one can even remove the management node while the cluster is running with little or no impact on performance. However, a management node is still required whenever a node joins the cluster. Answer **c** (*When the cluster shuts down*) is incorrect because there is no longer any reason to distribute configuration information at that point. Answer **d** (*Never*) is incorrect because this would make the management node's role in the cluster irrelevant.



## 6.5. Cluster Management — Exercises

---

evant.

Answer 4:

*ndb\_mgmd* (**b**) is correct.

Answer **a** (*ndb\_mgm*) is incorrect because this is the cluster management client. Answer **c** (*mysqld*) is incorrect because this is the MySQL server process. Answer **d** (*ndbd*) is incorrect because this is the cluster data node process.

Answer 5:

The correct answer is *30 seconds* (**a**).

Answer 6:

*StartPartialTimeout* (**c**) is correct.

Answer **a** (*TimeBetweenGlobalCheckpoints*) is incorrect because this represents the interval, in seconds, between commits of checkpoints to disk. Answer **b** (*initial*) is incorrect because *-initial* is an option that can be passed to *ndbd* when starting a data node. Answer **d** (*This value cannot be changed*) is incorrect because it is possible to change it.

Answer 7:

*[mysql\_cluster]* (**b**) and *[ndb\_cluster]* (**c**) are correct. (Note that *[mysql\_cluster]* is preferred.)

Answer **a** (*[cluster]*) is incorrect because this is not a valid section label. Answer **d** (*[mysqld]*) is incorrect because this section is reserved for parameters affecting the *mysqld* process.

Answer 8:

*Issuing the SHOW command in the management client* (**a**) and *Issuing the statement SHOW ENGINE NDB STATUS in the MySQL Monitor* (**d**) are both correct. (Note that *SHOW ENGINE NDB-CLUSTER STATUS* is a valid synonym for the latter.)

Answer **b** (*Issuing the SHOW NODE command in the management client*) is incorrect because there is no such management client command. Answer **c** (*Issuing the statement SHOW ENGINES from the MySQL command prompt*) is incorrect because this shows only that the NDB storage engine is available; it is not necessarily a reliable indicator that the node is connected to a specific cluster.

Answer 9:

*True.*

## 6.5. Cluster Management — Exercises

---

You may also use the short form `-c` for this option.

Answer 10:

*False.* No error is displayed until you actually attempt to execute a management client command. `SHOW WARNINGS` is specific to the MySQL server.

Answer 11:

Answers **b** (*EXIT*) and **c** (*QUIT*) are both correct.

Answer **a** (*STOP*) is incorrect because this command is used stop data nodes and management nodes. Answer **d** (*TERM*) is incorrect because this is not a valid MySQL Cluster management client command.

Answer 12:

Answer **d** (*There is no command for starting the management server process from the management client*) is correct.

Answer **a** (*START*) is incorrect because this command is used start *data* nodes. Answers **b** (*BEGIN*) and **c** (*INIT*) are incorrect because these commands do not exist in the management client.

Answer 13:

Answer **c** (**Rolling restart**) is correct.

Answer **a** (*Warm reboot*) refers to restarting any computer in general without cycling the power. Answer **b** (*Cold reboot*) refers to restarting any computer in general by cycling the power. Answer **d** (*Hot cycle*) is incorrect because there is no such term used within the scope of MySQL Cluster.

Answer 14:

Answer **d** (*ALL STATUS*) is correct. All of the other answers represent invalid commands.

Answer 15:

Answer **d** (*Extract configuration information about data nodes and API nodes*) is correct. All the other answers are incorrect because `ndb_config` cannot be used to remotely configure other nodes.

Answer 16:

Answer **a** (`ndb_nodeid_cluster.log`) is correct.

Answer **b** (`cluster_nodeid_ndb.log`) is misleading because it is a permutation of the correct answer. Answer **c** (`hostname.err`) is the default name for an SQL node log file. Answer **d** (`/var/log/messages`) is the name of the file used by the Linux operating system to log system

## 6.5. Cluster Management — Exercises

---

events.

---

## **Part III. Internals**

---

---

---

---

## Chapter 7. The NDB Storage Engine

The NDB Storage engine is a component of the MySQL server which enables communication with MySQL Cluster data nodes. This NDB storage engine allows a MySQL Server to delegate storage and retrieval of data to MySQL Cluster data nodes, offering the scalability and high availability features that are characteristic of a MySQL Cluster.

A MySQL Server may be thought of as an SQL-aware front end for a scalable and fault tolerant back end data store provided by the data nodes. A MySQL Server that is used in this way is referred to as an SQL Node.

The NDB storage engine forms the logical bridge between the SQL nodes and the data nodes. This chapter describes some of the details concerning the implementation of the actual storage engine. Apart from that, this chapter describes how to use the NDB storage engine and how SQL statements are mapped to operations performed in the data nodes.

### Note

Within the MySQL server, two different names may be used to refer to the storage engine: NDB and NDBCLUSTER. The former is an abbreviation of the latter; both are treated in exactly the same way (just as INNODB and INNOBASE are synonymous, and interpreted the same way when creating InnoDB tables).

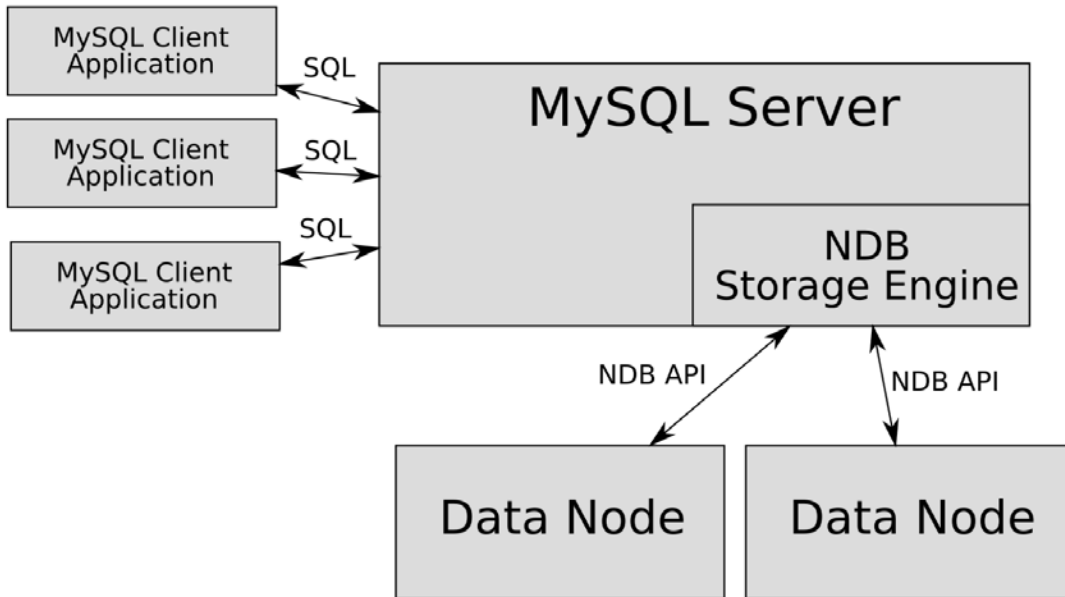
## 7.1. Overview

The NDB storage engine is the interface between the data nodes and the MySQL Servers (SQL nodes) that are part of a MySQL cluster. In order to understand the role of this storage engine in the MySQL architecture, it is useful to take a closer look at what a MySQL Server actually does, and how it performs its tasks.

### 7.1.1. The MySQL Server and the NDB Storage Engine

### 7.1.1. The MySQL Server and the NDB Storage Engine

---



Each MySQL server (SQL node) listens for connection requests from MySQL clients. Once a connection is established, the MySQL client may send requests to the SQL node. Such requests are made as SQL statements.

An SQL statement can be thought of as a high level specification of a particular collection of data that the client wants to obtain, or of an operation the client wants to perform on a collection of data. When an SQL node receives such a request, it tries to perform the operations specified by the SQL statement, or to retrieve the data specified. This process, known as *statement execution*, follows a number of distinct stages.

To perform operations on data, SQL statement execution eventually requires the actual data store to be located and manipulated. In relational database management systems like MySQL, data is logically stored in tables. In the MySQL server, the actual, physical storage of data is managed by particular MySQL server components called *storage engines*.

Multiple storage engines can co-exist within a single MySQL server, and a single SQL statement may refer to tables managed by different storage engines. Each table is always managed by exactly one storage engine; however, one storage engine can manage many tables. Each storage engine implements a particular application programming interface referred to as the *storage engine API*. Internally, the MySQL server uses this API to control a given storage engine to store or retrieve data as part of the SQL statement execution process.

For MySQL Cluster, one storage engine is of special interest: the NDB storage engine allows a MySQL

Server to delegate physical storage and retrieval to MySQL Cluster data nodes. (See Section 2.1.3, “Cluster Node Types”, for more information specific to cluster data nodes.)

MySQL Cluster data nodes are controlled using an application programming interface called the *NDB API*. Programs that use the NDB API to control data nodes are called *API nodes*. The NDB storage engine maps storage engine API calls performed by the MySQL server to a set of NDB API calls. Essentially, the NDB storage engine turns an ordinary MySQL server into a MySQL Cluster API node. Because the main job of the MySQL Server is to process and execute SQL statements, a MySQL server that uses the NDB storage engine to delegate storage and retrieval of data to MySQL Cluster data nodes is referred to as an *SQL node*.

NDB API calls may result in data to be returned from one or more data nodes to the SQL node. Depending on the original SQL statement, the SQL node can either return the data directly to the client, or retain it in order to perform additional processing. For example, SQL constructs such as joins and sub-queries require additional processing by the SQL node on the intermediate data sets. Other SQL constructs (such as WHERE clauses) can in part be delegated to the data nodes.

A MySQL server is independent of any particular storage engine. A given storage engine may or may not be available to a MySQL Server; it is entirely possible for the MySQL Server not to include support for the NDB storage engine. Even if the NDB storage engine is included, the MySQL server may not be configured to make actual use of it. (See Section 4.3.2.2, “SQL Node Options”.)

- Examining the output of the `SHOW ENGINES` statement, or querying the `INFORMATION_SCHEMA.ENGINES` table.
- Examining the output of the `SHOW PLUGINS` statement, or querying the `INFORMATION_SCHEMA.PLUGINS` table.
- Checking the value of the `have_ndbcluster` server variable.

For practical purposes, the `SHOW ENGINES` statement is usually sufficient. Typical output of this statement is shown here:



### 7.1.2. Checking for NDB Storage Engine Support

Engine	Support	Comment
ndbcluster	YES	Clustered, fault-tolerant tables
MRG_MYISAM	YES	Collection of identical MyISAM tables
CSV	YES	CSV storage engine
BLACKHOLE	YES	/dev/null storage engine (anything you write to it disappears)
FEDERATED	YES	Federated MySQL storage engine
MEMORY	YES	Hash based, stored in memory, useful for temporary tables
ARCHIVE	YES	Archive storage engine
InnoDB	YES	Supports transactions, row-level locking, and foreign keys
MyISAM	DEFAULT	Default engine as of MySQL 3.23 with great performance
EXAMPLE	YES	Example storage engine

(Some columns removed from the output) The row for which the Engine column value is `ndbcluster` corresponds to the status of the NDB storage engine. There are three possible values for the Support column of that row, indicating the status of the engine:

- **NO:** No support whatsoever for the NDB storage engine is included. That is, the MySQL server was not compiled with the NDB storage engine.
- **DISABLED:** The NDB storage engine is included, but it is currently not enabled. That is to say, the MySQL server has not been configured with the `ndbcluster` option.
- **YES:** Support for the NDB storage engine is included in the server, and the engine enabled. The NDB storage engine can be used to create tables.

The information in the `ENGINES` table in `INFORMATION_SCHEMA` is equivalent to the output of the `SHOW ENGINES` statement. However, because `INFORMATION_SCHEMA.ENGINES` is a table, you can apply a `WHERE` clause to the query in order to select only the row that corresponds to the NDB storage engine:

```
mysql> SELECT * FROM INFORMATION_SCHEMA.ENGINES
-> WHERE ENGINE = 'ndbcluster'\G
***** 1. row *****
ENGINE: ndbcluster
SUPPORT: ENABLED
COMMENT: Clustered, fault-tolerant tables
TRANSACTIONS: YES
XA: NO
SAVEPOINTS: NO
```

You can determine whether the MySQL Server was compiled with the NDB storage engine by inspecting the output of the `SHOW PLUGINS` statement, which contains a row having the value `ndbcluster` in the Name column and the value `ACTIVE` in the Status column if the server was compiled with support for the NDB engine. Even if the engine is disabled, the Status column value is `ACTIVE`. The `SHOW PLUGINS` statement is less useful than the `SHOW ENGINES` statement in this respect, because `SHOW ENGINES` also contains information on whether the MySQL server is configured to use NDB.

```
mysql> SHOW PLUGINS;
+-----+-----+-----+-----+-----+
| Name      | Status | Type      | Library | License |
+-----+-----+-----+-----+-----+
```

## 7.1.2. Checking for NDB Storage Engine Support

binlog	ACTIVE	STORAGE ENGINE	NULL	GPL
partition	ACTIVE	STORAGE ENGINE	NULL	GPL
ARCHIVE	ACTIVE	STORAGE ENGINE	NULL	GPL
BLACKHOLE	ACTIVE	STORAGE ENGINE	NULL	GPL
CSV	ACTIVE	STORAGE ENGINE	NULL	GPL
FEDERATED	ACTIVE	STORAGE ENGINE	NULL	GPL
MEMORY	ACTIVE	STORAGE ENGINE	NULL	GPL
InnoDB	ACTIVE	STORAGE ENGINE	NULL	GPL
MyISAM	ACTIVE	STORAGE ENGINE	NULL	GPL
MRG_MYISAM	ACTIVE	STORAGE ENGINE	NULL	GPL
ndbcluster	ACTIVE	STORAGE ENGINE	NULL	GPL

The contents of the `INFORMATION_SCHEMA.PLUGINS` table are more verbose than the output returned by the `SHOW PLUGINS` statement. However — like the output from the `SHOW PLUGINS` statement — the `INFORMATION_SCHEMA.PLUGINS` table can be used to see only whether the server was compiled with NDB engine support.

```
mysql> SELECT * FROM INFORMATION_SCHEMA.PLUGINS
-> WHERE PLUGIN_NAME = 'ndbcluster'\G
***** 1. row *****
      PLUGIN_NAME: ndbcluster
      PLUGIN_VERSION: 1.0
      PLUGIN_STATUS: ACTIVE
      PLUGIN_TYPE: STORAGE ENGINE
      PLUGIN_TYPE_VERSION: 50114.0
      PLUGIN_LIBRARY: NULL
      PLUGIN_LIBRARY_VERSION: NULL
      PLUGIN_AUTHOR: MySQL AB
      PLUGIN_DESCRIPTION: Clustered, fault-tolerant tables
      PLUGIN_LICENSE: GPL
1 row in set (0.00 sec)
```

If you need to find out only whether the MySQL Server was compiled with the NDB engine, it is probably more straightforward to check the value of the `have_ndbcluster` global server variable. You can do this with a `SHOW VARIABLES` statement such as this one:

```
mysql> SHOW VARIABLES LIKE 'have_ndbcluster';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| have_ndbcluster | YES  |
+-----+-----+
```

Alternatively, you can use `@@` notation and use the variable in a `SELECT` statement:

```
mysql> SELECT @@have_ndbcluster;
+-----+
| @@have_ndbcluster |
+-----+
| YES                |
+-----+
```

In either case, the variable has the value `YES` if the MySQL server was compiled with the NDB storage engine and the engine is enabled. If the MySQL server was compiled with the NDB storage engine but

### 7.1.2. Checking for NDB Storage Engine Support

---

the engine is not enabled, then this variable has the value `DISABLED`.

You must remember that merely having the NDB engine enabled does not mean that you can use the MySQL server to access a MySQL cluster. The MySQL server also needs to be connected to a running cluster (that is, a set of data nodes) in order to perform queries. You can see which MySQL servers are connected to the cluster by issuing the `SHOW` command in the management client, or by reviewing the contents of the cluster log.

```
ndb_mgm> SHOW
Connected to Management Server at: localhost:1186
Cluster Configuration
-----
[ndbd(NDB)] 2 node(s)
id=2 @10.0.0.2 (Version: 5.1.24-ndb-6.2.16, Nodegroup: 0, Master)
id=3 @10.0.0.3 (Version: 5.1.24-ndb-6.2.16, Nodegroup: 0)

[ndb_mgmd(MGM)] 1 node(s)
id=1 @10.0.0.1 (Version: 5.1.24-ndb-6.2.16)

[mysqld(API)] 2 node(s)
id=4 @10.0.0.4 (Version: 5.1.24-ndb-6.2.16)
id=5 (not connected, accepting connect from 10.0.0.5)
```

The MySQL server also maintains several status variables that can be used to discover whether it is currently connected to a running cluster. All of these status variables have the prefix `Ndb_`; you can check their values using a `SHOW STATUS` statement like this one:

```
mysql> SHOW STATUS LIKE 'ndb%';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| Ndb_cluster_node_id | 4 |
| Ndb_config_from_host | 10.0.0.1 |
| Ndb_config_from_port | 1186 |
| Ndb_number_of_data_nodes | 2 |
+-----+-----+
4 rows in set (0.08 sec)
```

The output shown here indicates that the MySQL server acts as an SQL node in a MySQL cluster. The `Ndb_cluster_node_id` status variable indicates that its node ID in the cluster is 4; `Ndb_number_of_data_nodes` tells us that the cluster is configured with 2 data nodes. The `Ndb_config_from_host` and `Ndb_config_from_port` status variables identify, respectively, the host name and port of the management server from which the cluster configuration was obtained at the time the SQL node joined the cluster.

When the MySQL server is started with the `--ndbcluster` option (but without a connectstring to connect it to a cluster), the values of these status variables are quite different, as shown here:

```
mysql> SHOW STATUS LIKE 'ndb%';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| Ndb_cluster_node_id | 0 |
| Ndb_config_from_host | |
+-----+-----+
```

### 7.1.3. Using the NDB Storage Engine

```
| Ndb_config_from_port | 0 |  
| Ndb_number_of_data_nodes | 0 |  
+-----+  
4 rows in set (0.00 sec)
```

#### Note

If MySQL is started without either of the `--ndbcluster` or `--ndb-connectstring` options, the result is an empty set:

```
mysql> SHOW STATUS LIKE 'ndb%';  
Empty set (0.00 sec)
```

### 7.1.3. Using the NDB Storage Engine

If the MySQL Server is connected to a running cluster, it is possible to create tables using the NDB engine. NDB tables are created by appending an `ENGINE` clause to the table definition to specify that the table is to be backed by the NDB engine. (Instead of NDB, the synonym `NDBCLUSTER` may also be used.) The `CREATE TABLE` statement shown here creates the table `City` as an NDB table:

```
CREATE TABLE City (  
  ID int NOT NULL AUTO_INCREMENT,  
  Name char(35) NOT NULL,  
  CountryCode char(3) NOT NULL,  
  District char(20) NOT NULL,  
  Population int NOT NULL,  
  PRIMARY KEY (ID)  
) ENGINE = NDB
```

The `ENGINE` clause can also be used to change the storage engine of an already existing table. For example, the following statement will alter the existing `Country` table to use the NDB storage engine:

```
ALTER TABLE Country  
ENGINE = NDB
```

By default, NDB tables use in-memory storage of data, so the NDB tables defined in the preceding syntax examples will be in-memory NDB tables. In-memory NDB tables are described in more detail later in this chapter in Section 7.2, “In-Memory Tables”. In MySQL 5.1, NDB tables may also store data on disk; disk-based storage is described in Section 7.4, “Disk-Based Tables”.

### 7.1.4. Propagation of Databases and NDB Tables

To some extent, the SQL nodes that are part of a MySQL cluster support automatic discovery of databases and NDB tables. This allows you to work with multiple MySQL Servers as if they manage the same databases and NDB tables. This may seem trivial, but it is important to remember that the MySQL

### 7.1.4. Propagation of Databases and NDB Tables

---

server was originally designed to work with a single set of databases and tables.

Each MySQL Server maintains one directory per database on the local filesystem, this directory being named after the database. Inside the database directory, the MySQL server maintains a file for each table; this file contains the table metadata. This file is named after that table, and has a `.FRM` extension. For example, a table named `City` has a corresponding metadata file named `City.FRM`. This organization is common to all databases and tables known to a given MySQL server, and is not unique to NDB tables. However, in the case of NDB tables, each data node maintains its own table metadata, which is automatically synchronized with the other data nodes. This synchronization occurs whenever a DDL statement is issued. When this occurs, the metadata maintained by a given SQL node is updated as the DDL statement is processed in the MySQL Server; when such a statement affects an NDB table, it also results in an equivalent operation being executed on all the other data nodes in the cluster, and their local metadata being updated accordingly. To some extent, this process is executed automatically.

This is known as *automatic discovery* (also sometimes called *autodiscovery*) of databases and NDB tables by the other SQL nodes. This propagation is taken care of by the same thread that maintains the binary log used for MySQL Cluster master-slave replication, known as the *binary log injector thread*, sometimes also referred to as the *binlog injector thread*. It is important to realize DDL statements affecting NDB tables are propagated asynchronously; there is no guarantee regarding exactly when a DDL statement originating on one SQL node is executed on the others.

#### Note

The role of the binary log injector thread for cluster master-slave replication is discussed in more detail in Section 10.1.1, “Cluster Replication Logging (binlog)”.

The following statements are automatically propagated:

- `ALTER DATABASE` or `ALTER SCHEMA`
- `CREATE DATABASE` or `CREATE SCHEMA`
- `DROP DATABASE` or `DROP SCHEMA`
- `ALTER TABLE`
- `CREATE TABLE`
- `DROP TABLE`

However, automatic propagation of table DDL is not always possible. Consider, for example, a MySQL cluster with 2 SQL nodes having node IDs 1 and 2, each of which contains an (empty) database named `world`.

Suppose that, in the `world` database on SQL node 1, a table called `City` based on the MyISAM storage engine has been created. Since this is not an NDB table, the DDL used to create it is not propagated to SQL node 2. This means that the `world` database on SQL node 2 does not contain a `City` table.

Now suppose that someone attempts to create its own `City` table in the `world` database using the

---

## 7.2. In-Memory Tables

---

NDB storage engine on SQL node 2. Assuming that the attempt succeeds, the DDL for creating this table propagates to SQL node 1. However, a table having that name already exists in the `world` database on SQL node 1. The following statements show what happens in such a case:

```
mysql> SHOW TABLES FROM world LIKE 'City';
+-----+
| Tables_in_world (City) |
+-----+
| City                    |
+-----+
1 row in set, 1 warning (0.01 sec)

mysql> SHOW WARNINGS;
+-----+-----+-----+
| Level | Code | Message
+-----+-----+-----+
| Warning | 1050 | Local table world.City shadows ndb table
+-----+-----+-----+
1 row in set (0.00 sec)
```

The existing `City` table is preserved, and issuing a `SHOW TABLES` statement on SQL node 1 generates a warning for each table that appears in the result for which an NDB table also exists. *However, SQL node 2 is not notified that any shadow tables exist on SQL node 1.*

Tables that do not use the NDB engine are not stored on cluster data nodes. This is also true of database objects that are not tables, such as views, stored procedures, triggers, and privileges. DDL creating or otherwise affecting such objects is not propagated; these objects must be created locally on all SQL nodes if they are needed there.

### Warning

It is true that some database objects such as user privileges and stored routines are stored as rows of tables found in the `mysql` system database. For this reason, you might be tempted to alter tables such as `mysql.proc` or `mysql.user` so that they employ the NDB storage engine. However, you must not try to do so. *Altering the storage engine used by any system table is not supported by MySQL.*

## 7.2. In-Memory Tables

By default, NDB tables are kept in memory — that is, they use RAM as their storage medium.

When starting, each data node allocates for storage an amount of RAM specified by the `DataMemory` configuration parameter. Memory required for in-memory tables is taken out of that portion of allocated memory. After this, no additional memory is ever allocated.

In many cases, in-memory tables use by far the largest portion of the `DataMemory`. (Ordered indexes, which are discussed in Section 7.5.2, “Ordered Indexes”, also draw from `DataMemory`.) That is why it is important to determine the maximum amount of memory required for in-memory tables before creating them. Calculation of memory requirements to migrate an existing database to the NDB engine using the `ndb_size.pl` script is described in Section 4.4.2, “Using `ndb_size.pl` to Estimate

---

## 7.3. Storage Requirements

---

Memory Requirements”. The following section explains how memory is used to store data, and can be used as a guideline when calculating storage requirements manually.

### 7.3. Storage Requirements

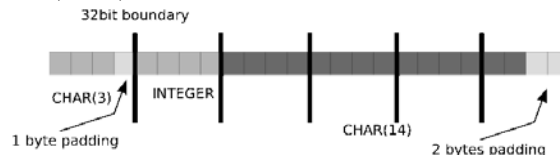
MySQL's storage requirements for different data types generally hold true for MySQL Cluster as well. (For a detailed discussion, see *Data Storage Requirements*, in the MySQL 5.1 Manual [<http://dev.mysql.com/doc/refman/5.1/en/storage-requirements.html>].) However, there are some extra considerations when calculating storage requirements for NDB tables.

#### 7.3.1. 4-Byte Alignment

In MySQL Cluster, each column is 4-byte (32-bit) aligned. This is because many computer architectures are optimized for 32-bit aligned data access and non-aligned access results in a heavy performance penalty. For example, a `BINARY(15)` column requires at least 15 bytes of storage, but it actually occupies 16 bytes, padding 1 (unused) byte in order to achieve 4-byte alignment. This 4-byte alignment is calculated for each table column independently of any other columns; if an NDB table has two `BINARY(14)` columns, each column requires 16 bytes, because 2 bytes are padded in order to achieve 4-byte alignment. Together, these columns occupy a total of  $16 + 16 = 32$  bytes.

You might think, since  $14 + 14 = 28$  and 28 is evenly divisible by 4 (and thus 4-byte aligned), that the columns in the previous example would require only 28 bytes of storage. However, 4-byte alignment is performed for each individual column, and not to any groups of columns taken together.

The following diagram shows how 4-byte alignment affects the storage requirements for three columns having the types `CHAR(3)`, `INT`, and `CHAR(14)`:



Here, you can see that the `CHAR(3)` column requires 4 bytes and the `CHAR(14)` column requires 16 bytes. Since the `INT` column already takes up 8 bytes, no additional alignment is required for it.

#### 7.3.2. Fixed and Variable Types

Previous to MySQL 5.1, all columns in NDB tables were fixed in size and stored in memory. However, in MySQL 5.1, the NDB storage engine supports three types of columns:

- Fixed size in memory
- Variable size in memory
- Fixed size on disk

A fixed-size column always occupies the same amount of space, even if not all bytes are used for stor-

---

### 7.3.3. Storage of BLOB and TEXT Types

---

age of a given column value. For example, assuming a single-byte character set, a column of type `CHAR(12)` always requires 12 bytes of memory per row, even if a given row stores only a single character in the column.

Disk Data columns are always stored as fixed-width columns. This means, for example, that a `VARBINARY(100)` column stored on disk always uses 100 bytes per row, plus 1 byte for length (and 3 bytes padding to make it 4-byte aligned) no matter how much data is stored in that column.

An in-memory `VARCHAR(100)` column is stored as a variable-sized column.

#### Note

The NDB API does make it possible to store variable data types such as `VARCHAR` as fixed types if desired. However, this functionality is not available through SQL, and thus it is outside the scope of this book and of the certification exam.

The variable-sized part of the row has a small amount of overhead, as does the column itself — we need to track how large the column is.

Each row is stored in one or two parts, depending on whether it has any variable-size columns. All fixed-sized columns are stored together as the fixed-size part of the row. If the row has any variable-sized columns, then the fixed-size part of the row contains a pointer to a variable-size part of the row where data in variable-size columns is stored. Although this pointer uses some memory, its cost is offset by the variable-size part saving space.

### 7.3.3. Storage of BLOB and TEXT Types

BLOB or TEXT columns are a special case for MySQL Cluster. Each of these types is stored in the same way: the first 256 bytes of the BLOB or TEXT column are stored in the fixed-sized part of the row. For `TINYBLOB` and `TINYTEXT` this is all of the storage that is used, since these two data types are limited to a maximum size of 256 bytes.

For other TEXT and BLOB types, the remainder of the record is stored in a hidden table (internal to NDB) in a series of discrete “chunks” or “parts”, whose sizes vary according to the data type, as shown in the following table:

Data type	Chunk size
BLOB or TEXT	2000 bytes
MEDIUMBLOB or MEDIUMTEXT	4000 bytes
LONGBLOB or LONGTEXT	8000 bytes

For example, a 9000-byte record in a `LONGTEXT` column uses two chunks; one to hold the first 8000 bytes, and the second to store the remaining 1000 bytes.



### 7.3.4. Per-Row Storage Requirements

---

Each chunk in this hidden table also uses 12 bytes of overhead (the space used by the primary key for this table) in addition to the standard row overhead. This means that a 2000-byte chunk for BLOB column data actually uses 2046 bytes of memory (an extra 10 bytes being used for the hash index on the primary key). For TEXT and BLOB columns stored on disk, the chunks are stored in the disk based part of the helper table but the indexed columns remain in memory.

### 7.3.4. Per-Row Storage Requirements

Each row stored in an NDB table requires an overhead of 16 bytes per row in addition to any added due to 4-byte alignment (see Section 7.3.1, “4-Byte Alignment”). For a variable-size column, 8 bytes of this is used as a pointer to the variable-sized part of the row. However, even if the row has no variable-sized part, this space is still allotted to it.

Indexes also add to per-row storage requirements. Each ordered index uses 10 bytes of DataMemory per row. A hash index uses 25 bytes of IndexMemory per row.

#### Important

Defining a PRIMARY KEY or a UNIQUE index (or unique constraint) automatically results in creation of a unique hash index as well as an implicit ordered index, resulting in an additional storage requirement of 35 bytes per row. If it is clear that the implicit ordered index is not required, its creation may be suppressed by adding a USING HASH clause to DDL statements that creates the PRIMARY KEY or UNIQUE index. For more information, see Section 7.5, “Indexes in NDB Tables”.

### 7.3.5. Page Management

DataMemory and IndexMemory are allocated to tables one *memory page* at a time. For DataMemory, each page is 32KB in size; an IndexMemory page is 8KB in size. DataMemory pages and IndexMemory pages that are not in use can be thought of as belonging to two pools of free pages, one each for DataMemory and IndexMemory.

Once a memory page has been allocated for a particular table, it remains available for data storage to only that table. Deleting data (even all of the data) from a page will not return the page to the free pool. Pages are returned to the free pool only when dropping the table or when truncating it (which implicitly drops the table and then re-creates the table).

Fixed-size and variable-size parts are stored in separate pages allocated from the free pool. A new table with some fixed-size and some variable-size columns with one row inserted in it uses two pages of DataMemory: one for the fixed part, and one for the variable part.

## 7.4. Disk-Based Tables

Beginning with MySQL 5.1, the NDB storage engine is capable of utilizing disk based storage. (In pre-

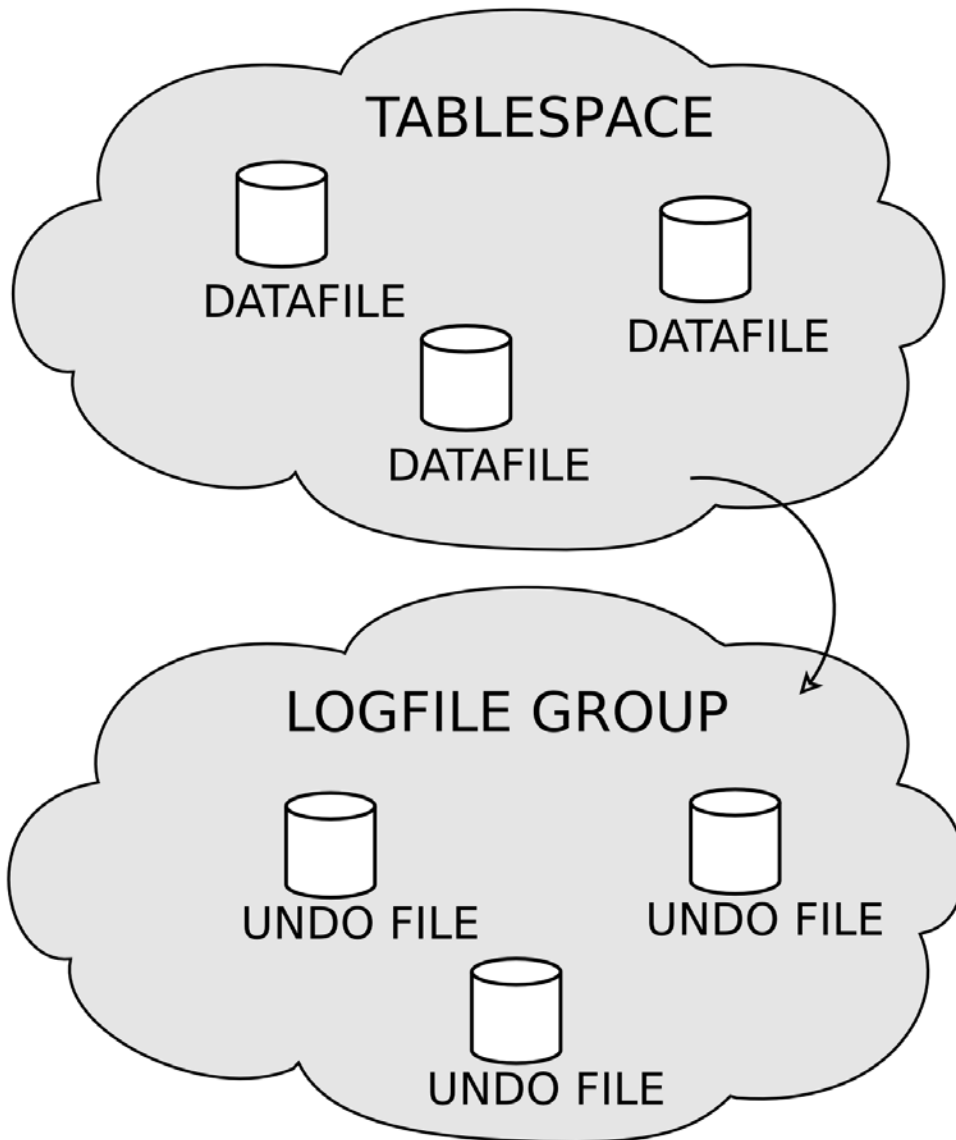
## 7.4. Disk-Based Tables

---

vious versions, NDB was limited to in-memory storage.) Because disk space is rather inexpensive as compared to RAM, this feature greatly expands the applicability of MySQL Cluster applications as compared with previous versions.

Currently, disk based data does not support variable sized storage. This does not mean that columns with a variable sized data type cannot be stored on disk; Rather, variable sized data types are internally stored in a fixed size format.

A disk-based table stores its data in a *tablespace*, which is made up of one or more *data files*. In order to facilitate rollbacks, a disk-based table also stores undo data in a *log file group* made up of one or more *undo log files* (sometimes referred to as *undo logs* or *undo files*). The relationships between these objects are shown in the following diagram:



The management of each of these objects is described in the next few sections.

### 7.4.1. Creating Disk-Based Tables

SQL in MySQL allows one to manage the storage medium on the table level for NDB tables. That is, for each NDB table, the storage medium may be controlled through an SQL DDL statement. The default storage medium is main memory (RAM). (In-memory tables have already been discussed in Section 7.2, “In-Memory Tables”.) In this section, we discuss how to use the disk as the storage medium.

The following syntax example shows how to create a table that uses disk-based storage:

```
CREATE TABLE City (  
  ID int(11) NOT NULL AUTO_INCREMENT,  
  Name char(35) NOT NULL,  
  Country char(3) NOT NULL,  
  District char(20) NOT NULL,  
  Population int(11) NOT NULL,  
  PRIMARY KEY (ID)  
)  
TABLESPACE ts1  
STORAGE DISK  
ENGINE=NDB
```

The `CREATE TABLE` statement must include the following clauses:

- An `ENGINE` clause specifying that the table is an NDB table. (This is no different from creating an in-memory NDB table.)
- A `STORAGE DISK` clause is required to indicate that disk-based storage is to be used.
- A *tablespace* must be specified using a `TABLESPACE` clause. A tablespace is a logical object that manages a number of *data files* in which the data is actually stored. Tablespaces and data files are discussed in detail in Section 7.4.2, “Tablespaces”.

Each of these three clauses must be present; however, they may be specified in any order following the table's column definitions.

Existing tables can be made to use disk based storage, using an `ALTER TABLE` statement. For example, the following statement changes an existing `Country` table to an NDB table that uses disk based storage:

```
ALTER TABLE Country  
TABLESPACE ts1  
STORAGE DISK  
ENGINE = NDB
```

As in the case of the previous `CREATE TABLE` statement, the `TABLESPACE`, `STORAGE DISK` and `ENGINE` clauses all need to be present — however, they may be specified in any order.

#### Important

Even if the table that is to be converted to use disk storage is already an NDB table, the `ENGINE` clause must be present in the `ALTER TABLE` statement used.

It is important to keep in mind that changing an in-memory NDB table into a NDB table that uses disk

---

## 7.4.2. Tablespaces

---

based storage compromises availability. During the time needed to complete the `ALTER TABLE` operation, the table should not be used by any application. The safest way to alter an NDB table is to use single user mode as described in Section 6.4, “Single User Mode”.

### 7.4.2. Tablespaces

MySQL Cluster data on disk is stored in *data files* kept in a named, logical container known as a *tablespace*. The next few sections of this chapter discuss the management of data files and tablespaces.

#### 7.4.2.1. Creating Tablespaces

Tablespaces are created using a `CREATE TABLESPACE` statement. The syntax for the `CREATE TABLESPACE` statement is shown here:

```
CREATE TABLESPACE tablespace_name
ADD DATAFILE 'path/to/file'
USE LOGFILE GROUP log_file_group_name
[EXTENT_SIZE [=] extent_size]
[INITIAL_SIZE [=] initial_size]
ENGINE [=] {NDB|NDBCLUSTER}
```

`CREATE TABLESPACE` requires an `ENGINE` clause which determines the storage engine used. Currently, the only accepted value for `ENGINE` is `NDB` (or `NDBCLUSTER`). This clause is required because, in the future, this syntax for tablespaces might be supported for other MySQL storage engines in addition to `NDB`.

Each tablespace must have a name that is unique for the entire cluster. The tablespace name can then be used in `CREATE TABLE` and `ALTER TABLE` statements to specify that a particular tablespace needs to be used for a particular NDB table. This is explained in more detail in Section 7.4.1, “Creating Disk-Based Tables”. Currently, there is no way to tie a tablespace to a database, it must be specified per table.

The `CREATE TABLESPACE` syntax includes mandatory `USE LOGFILE GROUP` clause. This clause is mandatory and specifies a particular log file group that is to be used to perform undo logging. Log file groups are discussed in more detail in Section 7.4.3, “Log File Groups”.

The mandatory `ADD DATAFILE` clause implicitly creates one *data file*. A tablespace can contain a number of these data files, which are used for the actual data storage. `INITIAL_SIZE` specifies the file size for the data file. Each data file is organized in a number of chunks called *extents*. The `EXTENT_SIZE` clause specifies how large these chunks are. The specified `EXTENT_SIZE` applies to each data file associated with the tablespace, so the extent size applies to the entire tablespace rather than to an individual data file. (In other words, you cannot specify an extent size when adding new data files to the tablespace; see Section 7.4.2.5, “Extents”.)

The values supplied for `INITIAL_SIZE` and `EXTENT_SIZE` are interpreted as numbers of bytes; each can be qualified with a suffix indicating the units to be employed, as shown in the following table:

---

## 7.4.2. Tablespaces

---

Suffix	Units
K	kilobytes (1024 bytes)
M	megabytes (1048576 bytes)
G	gigabytes (1073741824 bytes)

(These suffixes also apply to the `INITIAL_SIZE` specified for an undo log file as part of a `CREATE LOGFILE GROUP` or `ALTER LOGFILE GROUP` statement; see Section 7.4.3, “Log File Groups”, for more information.)

*path/to/file* indicates the location of the data file, and must not be the location of a file that already exists on any of the data nodes. Otherwise, an error occurs, and the data file is not created. If the `ADD DATAFILE` clause is part of a `CREATE TABLESPACE` statement, then the tablespace is also not created.

*path/to/file* can be an absolute or a relative path. If a relative path is used, the file is stored at the specified location beneath a subdirectory named `ndb_node_id_fs` in the data node's data directory, where *node\_id* represents the data node's node ID. For example, suppose that the `[ndbd de-fault]` section of global configuration file contains `DataDirectory = /var/lib/mysql-cluster`. Then the following clause creates one file named `datafile01.dat` in the directory `/var/lib/mysql-cluster/ndb_node_id_fs` on each data node:

```
ADD DATAFILE 'datafile01.dat'
```

See Section 4.3.4.2, “File and Directory Location Parameters”, and Section 4.3.4.1, “General Parameters”, for more information about the cluster global configuration file.

`INITIAL_SIZE` specifies the size of the data file when it is created. MySQL Cluster 5.1 does not support the expansion of these files, which means that the initial size is also the maximum size.

More information about data files and extents can be found in Section 7.4.2.3, “Creating Data Files”, and Section 7.4.2.4, “Dropping Data Files”.

### 7.4.2.2. Dropping Tablespaces

A tablespace can be dropped using the `DROP TABLESPACE` statement, whose syntax is shown here:

```
DROP TABLESPACE tablespace_name
ENGINE [=] {NDB|NDBCLUSTER}
```

*tablespace\_name* is the name of an existing tablespace. As with `CREATE TABLESPACE`, the `DROP TABLESPACE` statement includes a mandatory `ENGINE` clause. The only accepted values for `ENGINE` in MySQL 5.1 are `NDB` and `NDBCLUSTER`.

---

## 7.4.2. Tablespaces

---

The tablespace must be empty before it can be dropped. Dropping the tablespace fails if the tablespace still contains any data files, causing the following error message to be issued:

```
mysql> DROP TABLESPACE ts1 ENGINE = NDB;  
ERROR 1517 (HY000): Failed to drop TABLESPACE
```

This error message is not very clear, but the warnings that are also generated provide a better explanation why the DROP TABLESPACE statement failed:

```
mysql> SHOW WARNINGS;  
+-----+-----+-----+  
| Level | Code | Message                                                                 |  
+-----+-----+-----+  
| Error | 1296 | Got error 768 'Cant drop filegroup, filegroup is used' from NDB      |  
| Error | 1517 | Failed to drop TABLESPACE                                             |  
+-----+-----+-----+  
2 rows in set (0.00 sec)
```

Before dropping a tablespace, any data files that it contains first need to be removed. Removing data files is discussed in more detail in Section 7.4.2.4, “Dropping Data Files”.

### 7.4.2.3. Creating Data Files

As mentioned elsewhere, a tablespace holds a collection of *data files* in which NDB table disk data is stored, and which reside on the filesystem of each data node. One data file must always be specified in the ADD DATAFILE clause of a CREATE TABLESPACE statement.

Once the tablespace is created, more data files may be added to increase the total storage capacity of the tablespace, using the statement ALTER TABLESPACE . . . ADD DATAFILE. Since data files are of a fixed size, this is the only way to increase the capacity of the tablespace. The syntax for this statement is shown here:

```
ALTER TABLESPACE tablespace_name  
ADD DATAFILE 'path/to/file'  
[INITIAL_SIZE [=] initial_size]  
ENGINE [=] {NDB|NDBCLUSTER}
```

As for other statements affecting Disk Data tablespaces, the ENGINE clause is required. NDB and NDBCLUSTER are the only values that may be used for ENGINE in MySQL 5.1.

As is the case when ADD DATAFILE is used with CREATE TABLESPACE, *path/to/file* must not be a path to an existing file, and may be either relative or absolute. See Section 7.4.2.1, “Creating Tablespaces”.

INITIAL\_SIZE specifies the size of the data file when it is created. MySQL Cluster 5.1 does not support the expansion of these files, which means that the initial size is also the maximum size.

---

## 7.4.2. Tablespaces

---

### 7.4.2.4. Dropping Data Files

An existing data file may be dropped using `DROP DATAFILE` as part of an `ALTER TABLESPACE` statement, as shown here:

```
ALTER TABLESPACE tablespace_name
DROP DATAFILE 'path/to/file'
ENGINE [=] {NDB|NDBCLUSTER}
```

Here, *tablespace\_name* is the name of the tablespace containing the data file represented by *path/to/file*. As with other tablespace DDL statements, the `ENGINE` clause is required, and must specify either one of the values `NDB` or `NDBCLUSTER`.

The data file can be dropped only if it is not in use — that is, if it does not contain any data. Otherwise, an error results, as shown here:

```
mysql> ALTER TABLESPACE ts1
      -> DROP DATAFILE 'data02.dat'
      -> ENGINE = NDB;
ERROR 1521 (HY000): Failed to alter: DROP DATAFILE
```

The `SHOW WARNINGS` statement can be used to get a better indication of the problem's source:

```
mysql> SHOW WARNINGS;
+-----+-----+-----+
| Level | Code | Message                                     |
+-----+-----+-----+
| Error | 1296 | Got error 770 'Cant drop file, file is used' from NDB |
| Error | 1521 | Failed to alter: DROP DATAFILE             |
+-----+-----+-----+
2 rows in set (0.00 sec)
```

### 7.4.2.5. Extents

MySQL Cluster data files are split up into allocation units called *extents*. The size of the extents within a given data file can be defined in an `EXTENT_SIZE` clause in a `CREATE TABLESPACE` statement. This is discussed further in Section 7.4.2, “Tablespaces”.

When a table requires disk space for storing new rows, any extents that are currently allocated for its use are checked for free space. If no free space is found, a free extent in one of the data files of the tablespace is allocated to that table.

#### Important

An extent is allocated to a single table, and no extent stores data from more than one table.

If an extent cannot be allocated (that is, if there are no free extents), no more rows can be stored in the table, which causes this error message to be issued:

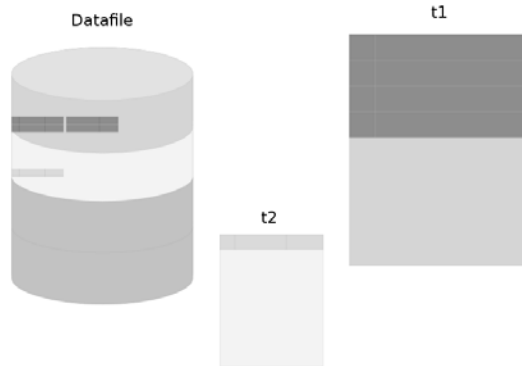


### 7.4.3. Log File Groups

---

```
ERROR 1114 (HY000): The table 'tablename' is full
```

The figure below shows how one extent is allocated for each of the tables `t1` and `t2`:



Once allocated to a table, an extent is not freed until that table is dropped using a `DROP TABLE` statement. A `DELETE` statement will discard rows, but the storage occupied by the deleted rows will remain allocated to the table. That is, other rows in the same table can utilize the space, but other tables cannot.

### 7.4.3. Log File Groups

Disk based storage requires an *undo log*, which is used for crash-recovery purposes. Undo logging is managed via a *log file group*, where a log file group acts as a container for one or more undo files. Currently, only one log file group is permitted for the entire cluster.

All Disk Data tablespaces must use the log file group. Therefore, a log file group must be created before creating any tablespaces, because a tablespace definition must specify explicitly the log file group it uses. Creating a log file group is done using the `CREATE LOGFILE GROUP` statement, which is shown here:

```
CREATE LOGFILE GROUP logfile_group
  ADD UNDOFILE 'path/to/file'
  [INITIAL_SIZE [=] initial_size]
  [UNDO_BUFFER_SIZE [=] undo_buffer_size]
  ENGINE [=] {NDB|NDBCLUSTER}
```

*logfile\_group* specifies the name of the log file group. Even though only one log file group is allowed currently, support for multiple log file groups may be added in the future. For this reason, a log file group must be uniquely identifiable.

The `UNDO_BUFFER_SIZE` clause specifies the size of the UNDO buffer. The `UNDO_BUFFER_SIZE` clause is optional. When omitted, a default size of 8MB is used.

#### 7.4.4. Disk Storage Metadata

---

The `ENGINE` option is required and must specify the NDB engine (NDBCLUSTER may also be used).

A log file group is always created together with an undo file. For this reason, an `ADD UNDOFILE` clause is mandatory. This clause specifies one undo file by its location on the data node filesystem *path/to/file*, which can be an absolute path or a relative one. If a relative path is used, the file is created at the specified location relative to a subdirectory named `ndb_node_id_fs` of the data node's data directory, where *node\_id* is the data node's node ID.

##### Note

The undo file is created at the specified location on each data node, which means that you cannot use an absolute path when running multiple data nodes on the same physical host.

The file specified by *path/to/file* must not correspond to an existing file. Otherwise, an error occurs and the log file group is not created.

The `INITIAL_SIZE` clause has the same semantics as with respect to data files; it specifies the size of the undo file. (See Section 7.4.2.3, “Creating Data Files”.) If this clause is omitted, the default file size is 128MB.

##### Note

The log file group must have sufficient capacity to record all changes occurring during the time needed to write three local checkpoints. As a rule of thumb, this minimum can be calculated by multiplying the value of the `DataMemory` configuration parameter by three.

As is the case for data files in tablespaces, undo files cannot grow in size once they are created. However, a log file group can contain multiple undo files. Undo files can be added to an existing log file group using the `ALTER LOGFILE GROUP` statement, as shown here:

```
ALTER LOGFILE GROUP logfile_group
ADD UNDOFILE 'path/to/file'
[INITIAL_SIZE [=] initial_size]
ENGINE [=] {NDB|NDBCLUSTER}
```

#### 7.4.4. Disk Storage Metadata

The `FILES` table in the `INFORMATION_SCHEMA` database contains information on files related to disk based storage. It is a MySQL extension to `INFORMATION_SCHEMA` and so is not standardized across different relational database management systems, although it should not be entirely foreign to those used to other systems.

Each row in the `INFORMATION_SCHEMA.FILES` table corresponds either to a data file or to an undo file, with the exception of an additional row for log file groups, since free space is reported per log file group, not per undo file. Because each data file and each undo file is present on each data node

## 7.4.4. Disk Storage Metadata

of the cluster, there are as many rows for each file as there are online data nodes in the cluster. The node ID of the cluster node on which the file resides is listed in the `EXTRA` column of the `FILES` table. The full list of columns is shown below.

```
mysql> DESCRIBE INFORMATION_SCHEMA.FILES;
```

Field	Type	Null	Key	Default	Extra
FILE_ID	bigint(4)	NO		0	
FILE_NAME	varchar(64)	YES		NULL	
FILE_TYPE	varchar(20)	NO			
TABLESPACE_NAME	varchar(64)	YES		NULL	
TABLE_CATALOG	varchar(64)	YES		NULL	
TABLE_SCHEMA	varchar(64)	YES		NULL	
TABLE_NAME	varchar(64)	YES		NULL	
LOGFILE_GROUP_NAME	varchar(64)	YES		NULL	
LOGFILE_GROUP_NUMBER	bigint(4)	YES		NULL	
ENGINE	varchar(64)	NO			
FULLTEXT_KEYS	varchar(64)	YES		NULL	
DELETED_ROWS	bigint(4)	YES		NULL	
UPDATE_COUNT	bigint(4)	YES		NULL	
FREE_EXTENTS	bigint(4)	YES		NULL	
TOTAL_EXTENTS	bigint(4)	YES		NULL	
EXTENT_SIZE	bigint(4)	NO		0	
INITIAL_SIZE	bigint(21)	YES		NULL	
MAXIMUM_SIZE	bigint(21)	YES		NULL	
AUTOEXTEND_SIZE	bigint(21)	YES		NULL	
CREATION_TIME	datetime	YES		NULL	
LAST_UPDATE_TIME	datetime	YES		NULL	
LAST_ACCESS_TIME	datetime	YES		NULL	
RECOVER_TIME	bigint(4)	YES		NULL	
TRANSACTION_COUNTER	bigint(4)	YES		NULL	
VERSION	bigint(21)	YES		NULL	
ROW_FORMAT	varchar(10)	YES		NULL	
TABLE_ROWS	bigint(21)	YES		NULL	
AVG_ROW_LENGTH	bigint(21)	YES		NULL	
DATA_LENGTH	bigint(21)	YES		NULL	
MAX_DATA_LENGTH	bigint(21)	YES		NULL	
INDEX_LENGTH	bigint(21)	YES		NULL	
DATA_FREE	bigint(21)	YES		NULL	
CREATE_TIME	datetime	YES		NULL	
UPDATE_TIME	datetime	YES		NULL	
CHECK_TIME	datetime	YES		NULL	
CHECKSUM	bigint(21)	YES		NULL	
STATUS	varchar(20)	NO			
EXTRA	varchar(255)	YES		NULL	

```
38 rows in set (0.00 sec)
```

Not all of the columns in `INFORMATION_SCHEMA.FILES` are regularly needed or even used by the NDB storage engine. The most common questions that we are likely to ask by querying this table are probably “Am I running out of space?” and “Should I add another data file to this tablespace?”; these can be answered by checking the result of a query like the one we show here, in the form of a view:

```
CREATE OR REPLACE VIEW ndb_dd_df AS
SELECT
  FILE_NAME,
  (TOTAL_EXTENTS * EXTENT_SIZE)/(1024*1024) AS 'Total MB',
```

## 7.5. Indexes in NDB Tables

---

```
(FREE_EXTENTS * EXTENT_SIZE)/(1024*1024)    AS 'Free MB',
(
  ((FREE_EXTENTS * EXTENT_SIZE)*100)
  /(TOTAL_EXTENTS * EXTENT_SIZE)
)
EXTRA
FROM
  INFORMATION_SCHEMA.FILES
WHERE
  ENGINE      = "NDBCLUSTER"
  AND FILE_TYPE = "DATAFILE";
```

For example, first suppose that you create a log file group, a tablespace and some data files, as shown here:

```
CREATE LOGFILE GROUP lg1
  ADD UNDOFILE 'undofile.dat'
  UNDO_BUFFER_SIZE 1048576
  INITIAL_SIZE 16777216
  ENGINE NDBCLUSTER;

CREATE TABLESPACE ts1
  ADD DATAFILE 'datafile.dat'
  USE LOGFILE GROUP lg1
  EXTENT_SIZE 1048576
  INITIAL_SIZE 52428800
  ENGINE NDBCLUSTER;

ALTER TABLESPACE ts1
  ADD DATAFILE 'datafile1.dat'
  INITIAL_SIZE 52428800
  ENGINE NDBCLUSTER;
```

Suppose further that you create one or more Disk Data tables and insert some data into them (not shown here).

To check the available free space, use the view `ndb_dd_df` created previously:

```
mysql> SELECT * FROM ndb_dd_df;
```

FILE_NAME	Total MB	Free MB	% Free	EXTRA
datafile1.dat	50.0000	50.0000	100.0000	CLUSTER_NODE=1
datafile1.dat	50.0000	50.0000	100.0000	CLUSTER_NODE=2
datafile.dat	50.0000	2.0000	4.0000	CLUSTER_NODE=1
datafile.dat	50.0000	2.0000	4.0000	CLUSTER_NODE=2

4 rows in set (0.00 sec)

Note that there may be free space inside the extents allocated to tables, so even if a new table were to consume all the free extents, existing tables may be able to grow significantly before we get any Table is full error messages.

## 7.5. Indexes in NDB Tables

---

### 7.5.1. Hash Indexes

---

Indexes are data access structures that can be used to search for a particular row or set of rows quickly. As such, indexes are an important tool to speed up query execution. In some cases, particular indexes can also be used to enforce that a table only contains unique rows. A good understanding of indexes is crucial to successfully tuning query performance. That topic is described in detail in Chapter 8, *Performance and Tuning*.

The NDB storage engine supports two types of indexes — *hash indexes* and *ordered indexes*. While NDB does not introduce any new syntax for defining indexes, it does interpret DDL statements creating indexes differently from how other engines do so. The next few sections of this chapter deal with these two types of indexes, how they are created and managed, and how they affect storage requirements and performance.

#### 7.5.1. Hash Indexes

A hash index is implemented using an in-memory key-value map called a *hash table*. When a row is added to a table with a hash index, the indexed columns are used to calculate a `hash` value, which is used as key to map a value that acts as a pointer to the physical location of the record in the table. This location is usually represented as an address in memory, or as a particular sector on a hard disk.

The hash value is calculated by a *hash function* which is chosen in a manner such that returned hash values will be (nearly) unique. This makes hash indexes highly suitable for implementing unique indexes, and this is how they are used by the NDB storage engine.

For the NDB storage engine, the hash value has an additional significance. Each NDB table always contains a primary key, even if it is not explicitly defined, implemented by means of a hash index. A hash value that is calculated for a given record in order to add it to the primary key index is also used to partition data across cluster data nodes. The hash function is chosen so that a given hash value corresponds uniquely to the data node that houses the primary fragment where the record is to be stored. Partitioning is described in more detail in Section 3.6, “Data Partitioning”.

Hash indexes are particularly useful for looking up individual values quickly, but they are not helpful for finding a range of values. The manner in which NDB hash indexes are used for lookup operations is described in more detail in Chapter 8, *Performance and Tuning*.

NDB hash indexes are maintained exclusively in main memory. The memory for hash indexes is allocated from the pool of memory allocated by the `IndexMemory` parameter, whose configuration is discussed in detail in Section 4.3.4.3, “Storage Parameters”.

#### 7.5.2. Ordered Indexes

Ordered indexes are so called because they organize entries in a particular order that is derived from the indexed values. This is contrast with hash indexes, which organize entries according to hash values. Although hash values are computed from data values, the order of the hash values does not relate in any other way to the order of the original data values.

---

### 7.5.3. Comparison of Hash and T-tree Indexes

---

The ordered indexes implemented by the NDB engine are implemented using *T-trees*. However, in the context of the NDB engine, ordered indexes are almost never referred to as “T-tree indexes”. A detailed discussion of T-trees is outside the scope of this book (and of the certification exam); however, we describe T-trees here in brief, as background information.

The term “T-tree” might incline you to think of an NDB ordered index as something that is not unlike the BTREE index that is implemented by the MyISAM and InnoDB storage engines. However, BTREE and T-tree indexes are not in any way similar, although these types of indexes do have more in common with each other than either has with hash indexes.

Both BTREE and T-tree indexes rely on organizing entries according to the sorting order of the values that make up the entries. As such, both types can both be thought of as “ordered”. However, in this book and in the MySQL Cluster certification exam, the term “ordered index” is always used to refer to a non-unique index as implemented by MySQL Cluster.

Ordering of index entries (or lack thereof) accounts for most of the functional differences between ordered indexes and hash indexes. Like hash indexes, NDB T-tree indexes are maintained entirely in memory. The memory for T-tree indexes is obtained from the memory allocated by the `DataMemory` parameter. (Memory for hash indexes is obtained from the pool of memory allocated by the `IndexMemory` parameter.) This is important to realize when calculating memory size requirements. The configuration of the `IndexMemory` and `DataMemory` parameters is discussed in detail in Section 4.3.4.3, “Storage Parameters”.

NDB ordered indexes can be created explicitly or implicitly. An ordered index is explicitly created whenever an SQL DDL statement defines a non-unique index, either directly using a `CREATE INDEX` statement, or as part of a `CREATE TABLE` or `ALTER TABLE` statement. See Section 7.5.4.4, “Non-Unique Indexes”, for further information.

Ordered indexes are created implicitly whenever an SQL DDL statement defines a primary key, a unique constraint, or a unique index, as discussed in Section 7.5.4, “SQL Mapping”.

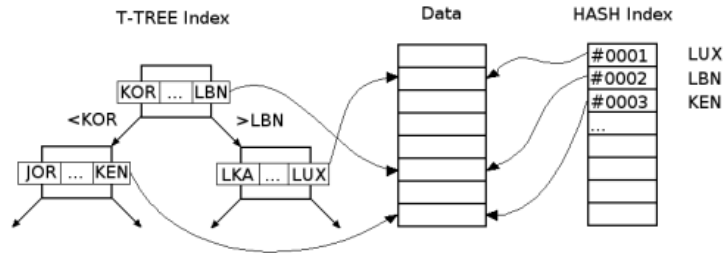
The implicit creation of an ordered index in such cases can be suppressed by including the `USING HASH` clause. See Section 7.5.4.3, “Creating Indexes with `USING HASH`”.

### 7.5.3. Comparison of Hash and T-tree Indexes

The following illustration compares hash and ordered (T-tree) indexes:

## 7.5.4. SQL Mapping

---



In this diagram, the rectangle labeled *Data* represents an NDB table; for the sake of example, assume that this is the *Country* table from the *world\_ndb* sample database. Both indexes allow access to individual rows once a pointer to a particular entry is found; the indexes allow a quick lookup of the physical storage location, which in this case is a memory address.

The right side of the diagram shows the organization of a hash index, which consists of a hash table that stores entries that map a hash value calculated from the indexed value to the physical location of the row. Note that the hash values #0001, #0002 and #0003 do not relate directly to the values LUX, LBN and KEN in the *Code* column from which the hashes were calculated. In addition, the order of the index entries does not relate directly to the order of the rows; only the pointer in the index entry identifies the corresponding row.

The organization of the T-tree index is shown on the left side. As the name indicates, T-tree indexes have a tree-like structure, consisting of nodes that are internally connected with pointers. Three nodes are shown here, each containing an array that maps an indexed value to a physical storage location. This array is depicted as a horizontally laid out series of index compartments. Within the array, the entries are stored in order. For example, the first entry in the array of the first node contains the value KOR, and the last entry contains the value LBN. All values . . . are in between KOR and LBN in ascending alphabetical order. Any values larger than LBN are stored in the right branch “beneath” this node, either directly in the array of entries of the right child node, or in one of the branches beneath that node. Likewise, all values smaller than KOR are stored somewhere in the left branch.

## 7.5.4. SQL Mapping

Usually, indexes are created explicitly using the SQL statements `CREATE INDEX` and `ALTER TABLE ADD INDEX`. Unique indexes are created implicitly to implement `UNIQUE` and `PRIMARY KEY` constraints. Constraints are also created through SQL statements such as `ALTER TABLE ADD CONSTRAINT`.

With regard to creating indexes by executing these SQL statements, there are a number of differences between the NDB engine and other engines such as `InnoDB` and `MyISAM`. The NDB engine does not introduce new syntax in order to accomplish this, but rather employs a slightly different interpretation of the SQL statements that imply index creation.

It is important to realize that the SQL statements are initially parsed and interpreted by the SQL nodes,

## 7.5.4. SQL Mapping

---

although they are actually executed inside the data nodes. Successful execution of these statements changes the state of both the SQL nodes and the data nodes; the data nodes actually create the indexes, but both the data nodes and the SQL nodes record the index definitions as part of the table definition. This means that, even though SQL nodes themselves do not actually create any indexes, they do update their local table metadata to keep track of the index definitions created for each table. SQL nodes use this metadata when creating query execution plans.

There are a few slight but important differences between the metadata maintained by the SQL nodes and the actual structures created inside the data nodes. Understanding the effect of a particular DDL statement creating an index is crucial in creating efficient NDB database schemas. This section discusses the effects of some typical DDL statements on index creation. Such a statement, when passed to an SQL node, is parsed and then sent to the data nodes where its actual effect is sometimes slightly different from what you might expect. For example, an SQL statement that implies creation of a (unique) hash index on an NDB table results in the creation of both an ordered index and a hash index, each of which indexes an identical set of columns. However, if the SQL statement includes the `USING HASH` clause, the ordered index is not created; only the hash index is created in such a case.

Not creating the implicit ordered index can be advantageous in some ways, since it saves 10 bytes of `DataMemory` per row, and allows for faster updates to the table because there is one less index to update. However, `USING HASH` should be employed with caution. This is because, in certain cases, hash indexes cannot be used to speed up query execution where an ordered index might have been efficiently used. The circumstances which allow utilization of an index to speed up query execution are discussed in detail in Section 8.3, “Data Access Methods”.

### 7.5.4.1. Primary Key

As discussed in Section 7.5.1, “Hash Indexes”, an NDB table always contains a primary key, backed by a hash index, even if it is not explicitly created using `PRIMARY KEY` as part of a `CREATE TABLE` statement. If the primary key is not explicitly defined, the NDB engine adds a hidden `BIGINT` column with a hash index on that column. For each row that is inserted into the table, an incrementing value is automatically supplied for the primary key column. Consider the following SQL `CREATE TABLE` statement:

```
# CREATE TABLE statement as issued on an SQL node:

CREATE TABLE City (
  Name CHAR(35) NOT NULL,
  CountryCode CHAR(3) NOT NULL
) ENGINE=NDB;
```

Notice that this statement does *not* include a `PRIMARY KEY` clause.

The objects actually created on the data nodes by this `CREATE TABLE` statement can be represented by the following SQL statement:

```
-- CREATE TABLE statement illustrating which structures
-- are actually created on the data nodes by the previous
```



## 7.5.4. SQL Mapping

---

```
-- statement:

CREATE TABLE City (
  __hidden BIGINT UNSIGNED NOT NULL AUTO_INCREMENT, -- hidden surrogate key column
  Name CHAR(35) NOT NULL,
  CountryCode CHAR(3) NOT NULL,
  PRIMARY KEY(__hidden), -- hidden unique hash index
  INDEX(__hidden) -- hidden non-unique ordered index.
) ENGINE=NDB;
```

There are three items that appear in the second statement that were not explicitly defined in the original statement:

- A hidden, auto-incrementing integer column, which we refer to by the name `__hidden` — however, it is extremely important that you realize that, at least from the SQL node's point of view, there is no such column and we are using this name purely for the sake of convenience
- A hash index implementing a primary key on `__hidden`
- An ordered index, indexing the same column(s) as the primary key

However, the cluster's SQL nodes are not aware of the existence of this hidden primary key; they know only as much about the `City` table as may be derived the original SQL statement. This means that the hidden column cannot be referenced from SQL, and indexes making up this primary key cannot be dropped or altered in any way from a MySQL server or its clients. In addition, the SQL node cannot use these indexes in constructing query plans, since it has no knowledge of their existence.

Of course, it is also possible to create a primary key on an NDB table explicitly using SQL, just as it is with other MySQL storage engines, as shown in this statement:

```
-- CREATE TABLE statement which includes an explicit PRIMARY KEY:

CREATE TABLE City (
  Name char(35) NOT NULL,
  CountryCode char(3) NOT NULL,
  PRIMARY KEY(Country, Name)
) ENGINE=NDB;
```

The following statement illustrates which structures are actually created on the data nodes:

```
-- CREATE TABLE statement illustrating the structures
-- actually created on the data nodes by the previous example

CREATE TABLE City (
  Name char(35) NOT NULL,
  CountryCode char(3) NOT NULL,
  PRIMARY KEY(Country, Name), -- explicit primary key, backed by unique hash index
  INDEX (Country, Name) -- implicit, hidden ordered index
) ENGINE=NDB;
```

---

## 7.5.4. SQL Mapping

---

In this case, no hidden column is added because of the explicit `PRIMARY KEY` clause, which is used on the data nodes; however, as part of the primary key's implementation, a unique hash index is also created. The SQL node is aware of the existence of this primary key and can use it in constructing and executing query plans to speed up query execution.

Although this `CREATE TABLE` statement — unlike that in the previous example — explicitly creates a primary key, it also implicitly creates a hidden ordered index (on the same columns that are already indexed by the unique hash index). Just as before, the SQL node is unaware that there are actually two indexes. However, the hidden index can also be utilized to speed up query execution; depending on the specific query, either index can be used when executing the query.

### 7.5.4.2. Unique Indexes

A unique index can be specified as part of a `CREATE TABLE` statement, and can also be created on columns of an existing table using SQL DDL statements such as `ALTER TABLE ADD UNIQUE` or `CREATE UNIQUE INDEX`. From the viewpoint of a MySQL server, all of these statements create essentially the same access structure. However, for NDB tables, creating a unique index using SQL also implicitly creates an additional, hidden ordered index. This is similar to the behavior seen for primary keys, as discussed in Section 7.5.4.1, “Primary Key”. The creation of the hidden ordered index can be suppressed by adding a `USING HASH` clause to the SQL statement used to create the unique index; this is described in more detail in Section 7.5.4.3, “Creating Indexes with `USING HASH`”.

For NDB tables, unique indexes are implemented by creating a separate NDB table, which is sometimes referred to as a *supporting table*. This supporting table — which is not directly visible to SQL nodes — may be thought of as defining two sets of columns:

- One set of columns corresponds to the columns defining the unique index on the original table. These columns form the primary key of the supporting table.
- A second set of columns map to the columns that make up the primary key of the original table (that is, the original table for which the unique index was defined).

In effect, the supporting table uses its own primary key to enforce unique entries. For each unique entry, the second set of columns is used as a foreign key that references the primary key of the corresponding row in the original table.

To picture the structures involved in implementing the unique index, consider the following statement:

```
# CREATE TABLE statement issued on an SQL node:
```

```
CREATE TABLE City (  
  ID          INT          NOT NULL,  
  CountryCode CHAR(3)      NOT NULL,  
  District    CHAR(20)     NOT NULL,  
  Name        CHAR(35)     NOT NULL,  
  PRIMARY KEY (ID)  
    USING HASH,  
  UNIQUE (CountryCode,District,Name)  
    USING HASH
```

## 7.5.4. SQL Mapping

---

```
) ENGINE=NDB
```

The following statements illustrate the structures are actually created on the data nodes when the previous statement is executed:

```
# CREATE TABLE statement illustrating
# the structures created in the data nodes:

CREATE TABLE City (
  ID          INT          NOT NULL,
  CountryCode CHAR(3)      NOT NULL,
  District    CHAR(20)     NOT NULL,
  Name        CHAR(35)     NOT NULL,
  PRIMARY KEY (ID)
) ENGINE=NDB

# CREATE TABLE statement illustrating
# the structure of the supporting table:

CREATE TABLE __hidden_City$Unique (
  CountryCode CHAR(3) NOT NULL,
  District    CHAR(20) NOT NULL,
  Name        CHAR(35) NOT NULL,
  ID          INT      NOT NULL,
  PRIMARY KEY(CountryCode,District,Name)
  FOREIGN KEY(ID)
    REFERENCES City(ID)
) ENGINE=NDB
```

In other words, the original statement creates the functional equivalent of the following structures:

- A user table called `City` with a column structure exactly as specified in the original statement.
- A unique hash index on the `ID` column of the `City` table to implement the primary key constraint on the `City` table.
- A hidden, supporting table (labeled `__hidden_City$Unique` in the example) is created to implement the unique index. This table is internal to NDB and is not visible to MySQL.
- A unique hash index on the `CountryCode`, `District`, and `Name` columns of the supporting table. This implements a primary key on the supporting table.
- A foreign key on the `ID` column of the supporting table, referencing the `ID` column of the original `City` table.

It is important to realize that the prior list is just a means making the implementation of NDB unique indexes more easily understood. It is not at all intended as a literal account of what goes on inside the NDB kernel. For example, the foreign key referred to in the list should not be understood to be a declarative foreign key constraint, since NDB does not support foreign keys. Rather, there is a structure that is used just as if such a foreign key exists.

In the example used here, the original `CREATE TABLE` statement included `USING HASH` clauses for

---

## 7.5.4. SQL Mapping

---

both the `PRIMARY KEY` and the `UNIQUE` constraints. Without the `USING HASH` clauses, additional ordered indexes would have been created on the data nodes. The `USING HASH` clauses have been included here to simplify this example.

### 7.5.4.3. Creating Indexes with `USING HASH`

It is possible to suppress the creation of an implicit ordered index when creating hash indexes on NDB tables by including `USING HASH` part of the clause that causes the hash index to be created. As such, `USING HASH` can be appended to a `PRIMARY KEY` or `UNIQUE` constraint, or to a `CREATE UNIQUE INDEX` statement. Consider, for example, the following statement:

```
# CREATE TABLE statement issued on an SQL node;
# resulting table on the data nodes matches it exactly

CREATE TABLE City (
    Name CHAR(35) NOT NULL,
    CountryCode CHAR(3) NOT NULL,
    PRIMARY KEY(Country, Name) USING HASH # Note USING HASH clause
) ENGINE=NDB;
```

This statement causes the specified structure to be created on the data nodes. Only one hash index is created to satisfy the `PRIMARY KEY` constraint, and no ordered index is created. This saves some storage overhead, but it also eliminates the possibility of performing any range scans. For applications that perform only primary key lookups for the particular table, it is safe to omit the ordered index. In such cases, omitting the ordered index reduces the amount of `DataMemory` required, since ordered indexes are allocated from the memory allocated by the `DataMemory` configuration option. In addition, omitting the index can help improve the performance of write operations. Maintaining and reorganizing the additional index as new entries are added costs some processing resources; this can be avoided entirely by not creating it.

`USING` is not specific to indexes on NDB tables. However, the index types permitted and the `USING` keyword's interpretation depend on the storage engine used. For example, the `MEMORY` storage engine allows index definitions with either `USING BTREE` or `USING HASH`, either of which creates an index of the corresponding type. However, the `MyISAM` and `InnoDB` storage engines implement only `BTREE` indexes; specifying `HASH` when creating an index on tables using one of these storage engines silently creates a `BTREE` index instead.

For the NDB engine, the semantics of `USING` are quite a bit different as compared to other storage engines. The only cases where `USING` has any effect on NDB tables is when specifying `USING HASH` for primary keys, unique constraints, and unique indexes. In all of these cases, a hash index is always created, but `USING HASH` prevents the implicit creation of an ordered index. In addition, applying `USING HASH` to a non-unique index on an NDB table results in an error message. This is not unreasonable, because a non-unique index is always implemented as an ordered index. This behavior is different from that of other storage engines in that the `USING` clause is *not* silently ignored, as it is with `MyISAM` and `InnoDB`.

### 7.5.4.4. Non-Unique Indexes

Non-unique indexes can be created explicitly using `INDEX` or `KEY` (the two keywords are in this case synonymous) as part of a `CREATE TABLE` or `ALTER TABLE` statement. They can also be created using the `CREATE INDEX` statement. Unlike such statements using `PRIMARY KEY`, `UNIQUE`, or `UNIQUE INDEX`, these statements cannot include a `USING HASH` clause and an attempt to do results in a runtime error, as shown here:

```
mysql> CREATE TABLE City (
->   ID INT NOT NULL,
->   CountryCode CHAR(3) NOT NULL,
->   District CHAR(20) NOT NULL,
->   Name CHAR(35) NOT NULL,
->   PRIMARY KEY (ID) USING HASH,
->   INDEX CountryCode (CountryCode,District,Name) USING HASH
-> ) ENGINE=NDB;
ERROR 1005 (HY000): Can't create table 'test.City' (errno: 138)

mysql> SHOW WARNINGS;
***** 1. row *****
Level: Error
Code: 1466
Message: Table storage engine 'ndbcluster' does not support the create
option 'Ndb does not support non-unique hash based indexes'
***** 2. row *****
Level: Error
Code: 1005
Message: Can't create table 'test.City' (errno: 138)
2 rows in set (0.00 sec)
```

Non-unique indexes are also created implicitly whenever a (unique) hash index is created on an NDB table, except when a `USING HASH` clause is included. This is discussed in detail in Section 7.5.4.1, “Primary Key”, Section 7.5.4.2, “Unique Indexes”, and Section 7.5.4.3, “Creating Indexes with `USING HASH`”. However, these implicitly created indexes cannot be manipulated separately from the hash index for which they were created - at least not using ordinary SQL DDL statements.

Consider the following `CREATE TABLE` statement:

```
CREATE TABLE City (
  ID          INT          NOT NULL,
  CountryCode CHAR(3)      NOT NULL,
  District    CHAR(20)     NOT NULL,
  Name        CHAR(35)     NOT NULL,
  Population  INT UNSIGNED NOT NULL,
  PRIMARY KEY (ID) USING HASH,
  UNIQUE (CountryCode,District,Name),
  INDEX (Population)
) ENGINE=NDB
```

Apart from creating the table, this statement creates the following database objects:

- A `HASH` index on the `ID` column, satisfying the `PRIMARY KEY` definition.

## 7.6. Transactions in NDB

---

Note that the `PRIMARY KEY` definition includes `USING HASH`. This suppresses the implicit creation of an ordered index on the `ID` column.

- A `HASH` index on the `CountryCode`, `District` and `Name` columns, satisfying the `UNIQUE` definition.
- An ordered index on the `CountryCode`, `District` and `Name` columns. This ordered index is created implicitly as a result of the `UNIQUE` definition.
- An ordered index on the `Population` column. This ordered index is created explicitly as a result of the `INDEX` definition.

In all of these cases, the hash indexes are unique indexes and the ordered indexes are non-unique indexes.

Non-unique indexes created on NDB tables are implemented as T-tree indexes. These are discussed in Section 7.5.2, “Ordered Indexes”. There is no automatic creation of any hidden objects to implement the ordered indexes.

## 7.6. Transactions in NDB

In general, the concept of a *transaction* entails that many single operations are treated as one single logical unit. When a transaction finishes, all the changes that occurred on any transactional tables during the transaction are either *committed* or *rolled back*. That is, the changes are either stored permanently in the database or completely undone. The ability to treat many single operations, possibly caused by different statements, as a single unit is necessary to implement the required business logic for many applications.

The NDB storage engine is a *transactional storage engine*. A storage engine is said to be transactional if it is capable of supporting transactions. A table that is managed by such a storage engine is similarly called a *transactional table*.

The well-known `InnoDB` storage engine is also a transactional storage engine supported by MySQL AB. The remainder of this section describes a number of differences and similarities between the `InnoDB` and NDB storage engines. In order to do that, a number of general transaction concepts need to be introduced first; see Section 7.6.1, “Database transactions and ACID properties”.

### 7.6.1. Database transactions and ACID properties

In the context of relational databases, the concept of a transaction has rather specific semantics. For (relational) database systems, the operations that are executed within a transaction should be thought of as operations on the row level — that is, the addition, modification or removal of an individual row. The exact definition of “treated as a single logical unit” is less straightforward. Often, the exact meaning is characterized in terms of the so-called *ACID* properties:

- Atomicity
- Consistency

## 7.6.1. Database transactions and ACID properties

---

- Isolation
- Durability

We discuss each of these in turn in the following sections.

### 7.6.1.1. Atomicity

The *atomicity* property of transactions holds that transactions must be indivisible (atomic). This means that, somehow, changes that are introduced within a transaction can be executed in such a manner that it seems as if the transaction really is a single action.

At a glance, the atomicity property may seem paradoxical: How is it possible to demand that the individual changes introduced during a transaction appear to be one action? If we effect the changes one by one, does that not contradict the appearance of a single action?

The answer is actually quite simple. The atomicity property just holds that when a transaction is finished, all of the individual changes introduced inside the transaction must either be completed, or all of them must be undone in a way that it seems as if the changes never occurred. We say that the transaction is *committed* when all changes are successfully and completely performed. If, on the other hand, all changes are undone, then we say that the transaction has been *rolled back*.

There are no differences between the InnoDB and NDB storage engines with regard to the atomicity property.

### 7.6.1.2. Consistency

The *consistency* property holds that performing a transaction cannot leave the database in a conflicting state.

A database schema may define all kinds of constraints that enforce restrictions on the data that may be entered into the database. These constraints maintain the integrity of the data managed by the database management system. The consistency property holds that when a transaction is committed or rolled back, none of the integrity rules may be violated as a result of executing the transaction.

The NDB storage engine supports consistency in full, just like the InnoDB engine. However, there is a difference in the way transactions are treated when a database constraint violation is encountered. This is explained in further detail in Section 7.6.2, “Transactions and Constraint Violations on NDB Tables”.

### 7.6.1.3. Isolation

The *isolation* property refers to the ability to shield an ongoing transaction from the changes performed by other transactions. Taking our initial definition for a transaction into account — that is, treating many different small tasks as a single large task — it is not hard to see why isolation is a desirable property: If an ongoing transaction can see the changes made by another transaction (whether the other transaction is ongoing or already completed), it becomes difficult to preserve the illusion that the transaction is one large task instead of many small ones.

### 7.6.1. Database transactions and ACID properties

---

Isolation is perhaps best illustrated by an example such as this one: Assume that we have a transaction consisting of two operations, each operation consisting of obtaining the total number of records in an initially non-empty table *A* and inserting this value into an initially empty table *B*. Now assume that a second transaction completely empties table *A*. In the absence of isolation, the first transaction can see the changes caused by the second transaction, and *vice versa*. If the second transaction were to be executed between the two insert operations executed in the first transaction, the number of records in table *A* is a nonzero value prior to the execution of the second transaction, and zero after its execution. It becomes impossible to explain the result of the first transaction as if it really was a single logical action, because the result proves there were at least two different occasions on which the number of records in table *A* were counted.

It is possible to implement perfect (or near-perfect) isolation; however, it usually comes at a considerable performance penalty. Therefore, in practical database systems, perfect isolation is rarely used. Rather, RDBMS use the notion of *isolation levels*. Each level defines a number of phenomena that indicate the existence of other ongoing transactions. These phenomena are:

- *Dirty reads*: A read in one transaction sees the changes brought about as the result of another ongoing transaction.
- *Non-repeatable reads*: In an ongoing transaction, a repeated attempt to read a row may at first succeed and then subsequently fail because the row has been deleted by another (committed) transaction in the meantime.
- *Phantom reads*: In an ongoing transaction, a repeated attempt to read a row may first fail and afterwards succeed because the row has been inserted by another (committed) transaction in the meantime.

The corresponding isolation levels are:

- The *READ UNCOMMITTED* isolation level allows dirty reads, non-repeatable reads, and phantom reads. Arguably, it is not an isolation level at all; rather, it represents the absence of any form of isolation whatsoever.
- The *READ COMMITTED* isolation level allows ongoing transactions to see all changes brought about by other transactions as soon as they are committed. This isolation level allows non-repeatable reads as well as phantom reads, but prevents dirty reads.
- The *REPEATABLE READ* isolation level allows phantom reads, but disallows both dirty reads and non-repeatable reads.
- *SERIALIZABLE* is the highest possible level of isolation. This isolation level prevents dirty reads, non-repeatable reads and phantom reads. In this isolation level, the transactions are scheduled in a way that makes it appear as if they are executed in sequence, and thus in perfect isolation.

The isolation levels show a clear ordering. The *READ UNCOMMITTED* level represents the virtual absence of isolation, and the *SERIALIZABLE* level is on the opposite end of the spectrum.

#### 7.6.1.4. Durability



## 7.6.2. Transactions and Constraint Violations on NDB Tables

---

The *durability* property of transactions holds that the changes made by a committed transaction are persistent: A transaction, once committed, cannot be rolled back, and it survives even in the event of system failure. Committed changes cannot be lost due to a rollback, a logical error, or even a system failure.

This immediately raises the question as to what qualifies as a system failure. In turn, this raises the question as to what should still be considered a part of the system. We do not explore this topic in depth; rather, we want to raise awareness concerning the limitations of the durability property. It is important to realize that the concept of system failure depends ultimately on an arbitrary notion of what is thought of as “the system” and the system’s environment. Therefore, to understand the meaning of the durability property in a specific ACID implementation, one should always attempt to grasp the implementation’s underlying assumptions regarding the notions of “system” and “failure”.

MySQL Cluster defines a transaction as committed when it can survive node failure. This is ensured by the use of the two-phase commit protocol discussed in Section 3.4, “Two-Phase Commit Protocol”.

However, a committed transaction need not be written to disk immediately. Transactions are written to disk whenever the next Global Check Point (GCP) occurs. This means that, in the event of a total cluster crash, committed transactions that are not part of a global check point will be lost.

This is unlike the InnoDB storage engine, which can be configured such that transactions are saved to disk periodically or at every commit using the `innodb_flush_log_at_trx_commit` system variable.

As far as the NDB storage engine is concerned, there is no difference in durability between tables using disk-based storage and those using in-memory tables. Global checkpointing is the mechanism that is ultimately responsible for the durability of committed transactions on disk, whether an NDB table uses in-memory or disk-based storage.

See Section 3.7.3, “Node Recovery”, for further discussion of NDB checkpointing and node failure.

## 7.6.2. Transactions and Constraint Violations on NDB Tables

NDB and InnoDB differ significantly in the way transactions in which are treated when a database constraint violation occurs. If you attempt to execute an SQL statement that would cause a database constraint violation on an NDB table, the statement fails and the entire transaction is aborted. At this point, all changes made in the course of the transaction are lost. Any attempt to continue the transaction results in a runtime error, as shown in the following example:

```
mysql> CREATE TABLE t_ndb (  
->     id INT PRIMARY KEY  
-> ) ENGINE = NDB;  
Query OK, 0 rows affected (0.04 sec)  
  
mysql> SET AUTOCOMMIT = OFF;  
Query OK, 0 rows affected (0.00 sec)
```

### 7.6.3. NDB READ COMMITTED Isolation Level

```
mysql> INSERT INTO t_ndb VALUES (1),(2),(3);
Query OK, 3 rows affected (0.01 sec)
Records: 3  Duplicates: 0  Warnings: 0

mysql> INSERT INTO t_ndb VALUES (4),(5),(1);
ERROR 1022 (23000): Can't write; duplicate key in table 't_ndb'

mysql> SELECT * FROM t_ndb;
ERROR 1296 (HY000): Got error 4350 'Transaction already aborted' from NDBCLUSTER
```

In order to continue and start a new transaction, an explicit ROLLBACK statement must be issued:

```
mysql> ROLLBACK;
```

However, if the table in question is an InnoDB table, the statement still fails, but the transaction is *not* rolled back. Any changes made during the transaction prior to the failed statement are preserved and can still be committed. The following example demonstrates this behavior for the InnoDB storage engine:

```
mysql> CREATE TABLE t_innodb (
->   id INT PRIMARY KEY
-> ) ENGINE = InnoDB;
Query OK, 0 rows affected (0.04 sec)

mysql> SET AUTOCOMMIT = OFF;
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO t_innodb VALUES (1),(2),(3);
Query OK, 3 rows affected (0.01 sec)
Records: 3  Duplicates: 0  Warnings: 0

mysql> INSERT INTO t_innodb VALUES (4),(5),(1);
ERROR 1582 (23000): Duplicate entry '1' for key 'PRIMARY'

mysql> SELECT * FROM t_innodb;
+----+
| id |
+----+
| 1  |
| 2  |
| 3  |
+----+
3 rows in set (0.00 sec)
```

In this example, an InnoDB table is first created with a PRIMARY KEY constraint. Then, AUTOCOMMIT is disabled in order to execute multiple statements in a single transaction. Then three rows are successfully inserted into the table. However, the second INSERT statement would cause a violation of the PRIMARY KEY, and so, the statement fails. The SELECT statement reveals that the three rows inserted through the first INSERT statement are still present in the table, proving that the transaction was not rolled back as a whole.

### 7.6.3. NDB READ COMMITTED Isolation Level

The NDB storage engine supports only the READ COMMITTED isolation level, whereas InnoDB sup-

### 7.6.3. NDB READ COMMITTED Isolation Level

---

ports all isolation levels described in Section 7.6.1.3, “Isolation”. `READ COMMITTED` isolates transactions from any uncommitted changes caused by other transactions, but still allows non-repeatable reads and phantom reads.

To illustrate this behavior, we need to start two separate user sessions, which can be distinguished from one another by looking at the prompts: for the first session we use the prompt `S1>` ; for the second session, we use the prompt `S2>` , setting these with MySQL's `PROMPT` statement.

We execute the following statements in Session 1:

```
mysql> PROMPT S1>
PROMPT set to 'S1> '

S1> CREATE TABLE t_ndb (
  ->   id INT
  -> ) ENGINE = NDB;
Query OK, 0 rows affected (1.87 sec)

S1> SET autocommit = OFF; -- initiates a new transaction
Query OK, 0 rows affected (0.06 sec)

mysql> INSERT INTO t_ndb VALUES (1); -- record inserted but not yet committed
Query OK, 1 row affected (0.03 sec)
```

After executing these statements in Session 1, we start another session and set the prompt for this session as `S2`:

```
mysql> PROMPT S2>
PROMPT set to 'S2> '

S2> SET autocommit = OFF; -- initiates a new transaction
Query OK, 0 rows affected (0.06 sec)

S2> SELECT * FROM t_ndb;
Empty set (0.01 sec)
```

Because the transaction in session 1 is not yet committed, the transaction in Session 2 cannot yet see the record inserted by Session 1. Now, we commit the transaction in Session 1. After the commit, a new transaction is implicitly started, in which we update the row in the table `t_ndb`:

```
S1> COMMIT;
Query OK, 0 rows affected (0.01 sec)

S1> UPDATE t_ndb SET id = 2;
Query OK, 1 row affected (0.03 sec)
Rows matched: 1  Changed: 1  Warnings: 0
```

Session 2 can now see the row inserted by session 1, thus demonstrating a phantom read. However, the row as seen by Session 2 still has the old value for the `id` column, because the session 1 did not yet commit the second transaction that updated the row in the `t_ndb` table, again demonstrating that uncommitted changes are not visible:

```
S2> SELECT * FROM t_ndb;
+-----+
```

## 7.6.4. NDB and Savepoints

---

```
| id |
+----+
| 1 |
+----+
1 row in set (0.01 sec)
```

We commit the transaction in Session 1:

```
mysql1> COMMIT;
Query OK, 0 rows affected (0.01 sec)
```

The transaction in Session 2 can now see the changes caused by the UPDATE statement, thus demonstrating a non-repeatable read:

```
mysql2> SELECT * FROM t_ndb;
+----+
| id |
+----+
| 2 |
+----+
1 row in set (0.01 sec)
```

For InnoDB tables, the exact results are dependent upon the current isolation level of the transaction. By default, InnoDB operates with the REPEATABLE READ isolation level, which still allows phantom reads but suppresses non-repeatable reads.

The isolation level can be explicitly set using the SET TRANSACTION ISOLATION LEVEL statement. However, setting the transaction isolation level affects the behavior of InnoDB only. The NDB storage engine supports only the REPEATABLE READ isolation level, and completely ignores any explicit setting of the isolation level.

The current isolation level can be discovered by querying the value of the tx\_isolation server variable:

```
mysql2> select @@tx_isolation;
+-----+
| @@tx_isolation |
+-----+
| REPEATABLE-READ |
+-----+
1 row in set (0.00 sec)
```

However, the value of this variable applies only to InnoDB transactions, and provides no information regarding to the NDB storage engine. This can be clearly seen in the previous example, where we witnessed phantom reads as well as non-repeatable reads in Session 2. Even though the value of tx\_isolation seems to preclude such behavior, it is in fact expected. The tx\_isolation server variable applies to InnoDB only, and has no bearing on the single possible isolation level (READ COMMITTED) supported by NDB.

## 7.6.4. NDB and Savepoints

---

## 7.6.4. NDB and Savepoints

---

The InnoDB storage engine includes support for *savepoints*, which can be thought of as named position markers within transactions. After creating a savepoint, it may be referenced in a later ROLLBACK statement in order to undo all changes that occurred beyond the savepoint.

The NDB storage engine does not support savepoints. Whenever data in an NDB table is modified inside a transaction, any attempt to create a savepoint inside that transaction results in a runtime error, as shown here:

```
mysql> CREATE TABLE t_ndb (
->   id INT PRIMARY KEY
-> ) ENGINE = NDB;
Query OK, 0 rows affected (0.04 sec)

mysql> SET autocommit = OFF;
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO t_ndb VALUES (1),(2),(3);
Query OK, 3 rows affected (0.01 sec)
Records: 3  Duplicates: 0  Warnings: 0

mysql> SAVEPOINT my_savepoint;
ERROR 1178 (42000): The storage engine for the table doesn't support SAVEPOINT
```

However, it is possible to roll back to a savepoint created *prior* to issuing the statement that affects the NDB table. An example is shown here:

```
mysql> CREATE TABLE t_innodb (
->   id INT PRIMARY KEY
-> ) ENGINE = InnoDB;
Query OK, 0 rows affected (0.04 sec)

mysql> CREATE TABLE t_ndb (
->   id INT PRIMARY KEY
-> ) ENGINE = NDB;
Query OK, 0 rows affected (0.04 sec)

mysql> SET autocommit = OFF;
Query OK, 0 rows affected (0.00 sec)

mysql> INSERT INTO t_innodb VALUES (1);
Query OK, 3 rows affected (0.01 sec)
Records: 3  Duplicates: 0  Warnings: 0

mysql> SAVEPOINT my_savepoint;

mysql> INSERT INTO t_ndb VALUES (1);
Query OK, 3 rows affected (0.01 sec)
Records: 3  Duplicates: 0  Warnings: 0

mysql> ROLLBACK TO SAVEPOINT my_savepoint;
Query OK, 0 rows affected (0.00 sec)

mysql> SELECT * FROM t_ndb;
Empty set (0.00 sec)

mysql> SELECT * FROM t_innodb;
+----+
| id |
```

---

## 7.6.5. Configuration of NDB Transactions

---

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

## 7.6.5. Configuration of NDB Transactions

There are a number of configuration options that influence the NDB storage engine with regard to behavior of transactions. Most of these parameters apply to the data nodes. These parameters are discussed in Section 4.3.4.5, “Transaction Handling Parameters”.

## 7.7. Limitations

Data Definition Language statements (such as `CREATE TABLE` and `ALTER TABLE`) are not isolated inside transactions and are immediately visible to other transactions. If the node performing a copying `ALTER TABLE` crashes while this statement is being executed, a temporary table (having a name beginning with `#sql`) may be left behind. During a copying `ALTER TABLE`, the table being altered should not be modified by other nodes. Locally, the node performing the `ALTER` operation takes a lock, but this is not a distributed lock; other nodes must be blocked from accessing the table through some other means. One way to do this is to use single user mode (see Section 9.1.3, “Single User Mode”).

The `TRUNCATE [TABLE]` statement is implemented as a `DROP TABLE` followed by `CREATE TABLE`. This means that `TRUNCATE` is non-transactional and cannot be rolled back.

One of the performance limitations of MySQL Cluster has to do with the performance of large joins. For MySQL Cluster, the joining of tables is performed wholly in the MySQL Server, the data nodes having no knowledge of them at all. This means that all the rows involved in a join need to be sent from the data nodes to the MySQL Server. This means that, when there are a lot of rows involved in a join, the network becomes a bottleneck.

Each table involved in such a join is handled sequentially, and not in parallel. This means that, in a join on two tables `t1` and `t2`, the MySQL Cluster Storage Engine first fetches rows from `t1`, and then from `t2`; only then does it actually perform the join and finally return the result to the user. The MySQL optimizer decides in which order to perform the join; this can be seen by using `EXPLAIN` (see *Optimizing Queries with EXPLAIN*, in the MySQL Manual [<http://dev.mysql.com/doc/refman/5.1/en/explain.html>], for more information).

`FULLTEXT` indexes are currently not supported by NDB. A common workaround is to use an external indexing engine which either periodically scans the database or is updated as a result of triggers, or of monitoring via an NDB API application. Another possibility is to replicate into an engine that supports `FULLTEXT` indexes and perform queries on these indexes on the replication slaves.

There are several cluster configuration parameters in the cluster configuration that limit the number of objects that can be managed by the cluster's data nodes. For example, the `MaxNoOfTables` paramet-

---

## 7.8. The NDB Storage Engine — Exercises

---

er places a limit on how many tables can be created. The `MaxNoOfAttributes` parameter places a limit on the number of attributes (columns) there can be. See Chapter 4, *Software Configuration*, for more information about parameters such as these.

Sometimes you may be required to perform transactions on the database which are larger than your configuration allows. The only current workaround is to execute the operation as several smaller transactions. For example, rather than deleting 4 million records from a table in a single transaction, you can achieve the same objective by executing many smaller transactions, using the `LIMIT` clause with the `DELETE` statement, perhaps deleting only 10,000 rows at a time (per transaction).

As discussed in Section 7.6.1.3, “Isolation”, the only transaction isolation level currently supported by MySQL Cluster is `READ COMMITTED`. This means that changed data is only visible to other transactions once the transaction is committed. Other levels are not supported, and the `SET TRANSACTION ISOLATION LEVEL` statement does not affect NDB transaction handling.

MySQL Cluster does not currently support foreign key constraints. Any SQL that specifies a foreign key constraint is parsed, but silently ignored (just as is the case with the `MyISAM` storage engine). In such cases, no constraint is created or enforced.

Columns that use a data type that normally uses variable-length storage, are stored in a fixed-length format when using disk-based storage. For example, each row in a disk-based `VARBINARY(1000)` column always uses 1000 bytes. For rows in a disk-based `VARCHAR(1000)` column, the space that is actually used also depends upon the character set, but the total number of bytes required to store each row always uses up the maximum for the column (in this case, 1000 bytes).

The MySQL Server tracks information about a table in a file on disk with a name ending in `.frm` — commonly known as “the FRM file”. In MySQL Cluster, each data node keeps a local copy of this file, to avoid having to request information from the cluster for every query. After performing schema operations on other MySQL Servers, a subsequent operation on a MySQL Cluster table may be aborted with the error message `Invalid Schema Object Version`. This means that the local copy of the FRM file was out of date when the request was sent to the data nodes. Due to some limitations within the MySQL Server, it is not always possible to retry the operation automatically in this situation, so this error message is sometimes returned to the client. However, retrying the operation processes the transaction with information from the updated FRM file.

## 7.8. The NDB Storage Engine — Exercises

These are sample questions and answers for the *NDB Storage Engine* Chapter.

Question 1:

What is the proper statement in the MySQL client to determine whether the NDB storage engine is enabled? [*check all that apply*]:

- a. `SHOW STORAGE`
- b. `SHOW ENGINE`

## 7.8. The NDB Storage Engine — Exercises

---

- c. `SHOW ENGINES`
- d. This information cannot be determined from the MySQL client

Question 2:

An InnoDB table cannot be directly converted to use the NDB storage engine after it has been created:

- a. True
- b. False

Question 3:

Do DDL (Data Definition Language) statements performed on cluster SQL always propagate to all other SQL nodes? [*Open-ended*]

Question 4:

Views and stored procedures that involve NDB tables are stored and distributed on data nodes within NDB tablespaces:

- a. True
- b. False

Question 5:

What is the maximum number of log file groups that MySQL Cluster can support?

- a. 1
- b. 2
- c. 4
- d. 8

Question 6:

If an NDB log file group is going to be used, when must it be created?

- a. Before any of its associated NDB tablespaces are created
- b. Before any Disk Data tables are created
- c. After the schema is finalized
- d. At any time

Question 7:



## 7.8. The NDB Storage Engine — Exercises

---

NDBCluster supports transactions with full ACID compliance:

- a. True
- b. False

Question 8:

What transaction isolation levels does NDBCluster support? [*check all that apply*]:

- a. READ\_UNCOMMITTED
- b. READ\_COMMITTED
- c. REPEATABLE\_READ
- d. SERIALIZABLE

Question 9:

If a table is created without an explicit primary key, what is the behavior of MySQL Cluster? [*check all that apply*]:

- a. A unique key is chosen and made into a clustered index
- b. A hidden AUTO\_INCREMENT primary key is created
- c. A column with repeating values is created and used as the primary key
- d. None of the above

Question 10:

One of the greatest strengths of MySQL cluster is the degree of parallelism that occurs in processing queries — if multiple tables are joined, the MySQL server will fetch rows from all tables at the same time:

- a. True
- b. False

Question 11:

FULLTEXT indexing is not supported in MySQL Cluster:

- a. True
- b. False

Question 12:

A disk-based table using the NDB storage engine uses only fixed-width columns:

## 7.8. The NDB Storage Engine — Exercises

---

- a. True
- b. False

### *Answers to Exercises*

Answer 1:

`SHOW ENGINES` (c) is correct.

Answers **a** (`SHOW STORAGE`) and **b** (`SHOW ENGINE`) are not valid MySQL statements. Answer **d** (This information cannot be determined from the command prompt) is incorrect because this information can be determined with the `SHOW ENGINES` statement.

Answer 2:

*False.* Any table can be converted to use the NDB storage engine with the statement `ALTER TABLE table_name ENGINE = NDB`, as long as the NDB storage engine is compiled into the MySQL Server and NDB has been enabled in the server's configuration.

Answer 3:

There are some instances of DDL statements not propagating to other nodes. Most notably, in the event that tables having the same name but not using NDB already exist on multiple SQL nodes, converting one of these tables to NDB on one SQL node will not convert its same-name counterparts on the other SQL nodes. Likewise, converting an NDB table to some other storage engine on one node will make the table disappear on other SQL nodes, precluding the possibility of future DDL statements regarding this table from propagating.

Answer 4:

*False.* Such objects are stored only on the MySQL server on which they are created. If you want to use them on another SQL node, you would have to create those objects on those nodes as well.

Answer 5:

*I (a).* In MySQL 5.1, `NDBCluster` can support only a single log file group.

Answer 6:

*Before any of its associated tablespaces are created* (a) is correct.

Answer **b** (Before any Disk Data tables are created) is superficially correct, but does not really address the point that an NDB tablespace must refer to an NDB log file group. Answer **c** (After the schema is finalized) is incorrect because this (also) has no bearing on log file groups or tablespaces. Answer **d** (At any time) is incorrect because the log file group must be created before any tablespace that refers to it.

## 7.8. The NDB Storage Engine — Exercises

---

Answer 7:

*True.* The NDB storage engine provides atomicity, consistency, isolation, and durability of transactions.

Answer 8:

*READ\_COMMITTED (b).* While all of the answers listed represent valid transaction levels that are supported by InnoDB, only *READ\_COMMITTED* is supported by NDB.

Answer 9:

*A hidden AUTO\_INCREMENT primary key is created (b)* is correct. The data type of this column is *BIGINT*.

Answer **a** (A unique key is chosen and made into a clustered index) is incorrect because — although InnoDB may behave in this fashion — NDB does not. Answer **c** (A field with repeating values is created and used as the primary key) is incorrect because by definition, a primary key may not have repeating values (and no such column is created in any case). Answer **d** (None of the above) is obviously incorrect since there is a correct answer.

Answer 10:

*False.* Joins, especially large joins, have their rows retrieved one table at a time. The implied parallelism comes from the ability to work on different *fragments* of the same table simultaneously.

Answer 11:

*True.* NDB supports only hash indexes and ordered indexes. A unique index is implemented by means of a hash index, which is accompanied by an ordered index as well, unless it created with *USING HASH*.

Answer 12:

*True.* An unindexed *VARBINARY(100)* column in a disk-based table, for example, always uses 100 bytes, plus one byte for length (and 3 bytes padding to make it 4-byte aligned).

---

## Chapter 8. Performance and Tuning

Performance is an important topic for most database setups, and MySQL Cluster is no exception. MySQL Cluster performance issues are only to some extent similar to general MySQL performance issues. There are many good reasons to consider MySQL cluster performance as a topic that is distinct from MySQL server performance.

This chapter covers configuration issues (in addition to those found in Chapter 4, *Software Configuration*) and other issues relating to cluster performance.

### 8.1. Introduction

Before we explore this subject in depth, it is necessary to define what we mean exactly by the term “performance”, and how different aspects of performance come into play in MySQL Cluster architecture.

#### 8.1.1. Performance and Tuning Concepts

*Performance* is a general term expressing how well something serves its intended purpose. In the context of databases, two distinct aspects of performance should be considered:

- Response time — that is, how quickly the system answers requests
- Throughput — that is, how much work can be performed in a given period of time

We discuss each of these aspects of performance in the next two sections of this chapter.

##### 8.1.1.1. Response Time

In the context of computer systems, the *response time* is the amount of time that passes between sending a request to the system and receiving a response. Response time is perhaps the aspect of performance that is easiest to grasp, because it can be experienced in real time by the end user. In fact, the response time is often perceived as the only or most important aspect of performance.

In most cases, knowing response times for individual requests is not useful, but rather knowing the distribution of response times. Instead of focusing on response times for individual requests, you should look at the average, minimum, and maximum response times.

Response time is itself a composite metric, which can be divided into a *transfer component* that measures the speed of requests and responses, and a *processing component* that measures the time required to process requests and generate responses. This is not a trivial distinction — possible approaches to decreasing response time depend on whether more time is spent on transfer or on processing.

For MySQL Cluster, the transfer component of response time points to the source of the most common bottlenecks. This is due to the fact that MySQL Cluster relies heavily on network communication between nodes; the efficiency of the network can have a serious impact on the time it takes to transmit

requests, data, or both.

### 8.1.1.2. Throughput

The term *throughput* denotes how much work is performed during a certain amount of time. Throughput can be measured as the amount of data that flows through the system during a specific amount of time. Alternatively, it can be measured as the number of requests that can be processed during a specific amount of time, or as the number of users that can be served simultaneously.

When attempting to increase throughput, you should be aware of any component of the system that acts as a bottleneck. As with response time, it makes sense to distinguish between processing and transfer, since a bottleneck in the network is likely to require a different solution than a bottleneck in processing requests.

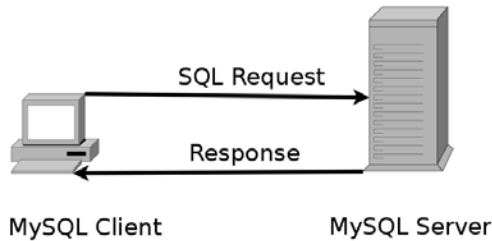
## 8.1.2. MySQL vs MySQL Cluster Performance

Since a MySQL Cluster usually contains multiple SQL nodes, which are MySQL servers, managing MySQL Cluster performance issues consists in large part of managing MySQL server performance issues. However, MySQL server performance is just one aspect of MySQL Cluster performance. The MySQL Cluster architecture is in many ways very different from that of a traditional MySQL database server. Many of these differences affect MySQL Cluster performance, and the methods to address performance issues differ accordingly.

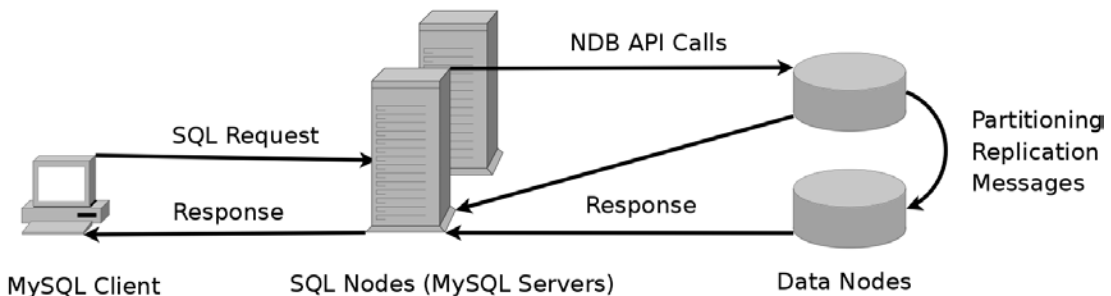
## 8.1.2. MySQL vs MySQL Cluster Performance

---

### Single MySQL Server Architecture



### MySQL Cluster Architecture



The single most important difference between a MySQL cluster and a single MySQL server is the distributed nature of MySQL Cluster. Where MySQL traditionally has used a single server with one or more client-server relationships, MySQL Cluster has many different types of nodes, occupied with different types of tasks. In addition, cluster nodes may communicate directly with one another and may delegate tasks to one another, not unlike how a MySQL client application delegates tasks to a MySQL server.

An understanding of the different tasks performed by the different types of cluster nodes and the interactions involved in the delegation of tasks among them is crucial in managing performance issues. Since the involvement of management nodes in the actual storage, processing, and transfer of data is minimal, we should focus our attention with regard to performance issues to SQL nodes and data nodes.

#### 8.1.2.1. Performance and SQL Nodes

SQL nodes are MySQL Servers that are connected to a MySQL Cluster. They provide users with an SQL interface to the NDB storage engine. Because all queries and data flowing between the user or ap-

---

### 8.1.2. MySQL vs MySQL Cluster Performance

---

plication and the cluster's data nodes must ultimately pass through an SQL node, the SQL nodes also represent a potential point of inefficiency and thus poor performance.

Perhaps the single greatest source of performance bottlenecks associated with SQL nodes is unnecessary processing of data. Not only must data be transferred across the network connecting the data nodes and the SQL node, but in many cases, the SQL node must wait until all data is received before it can perform its own processing. One should always attempt to minimize the amount of data that has to be sent to the SQL nodes.

One way in which this can be accomplished is by utilizing the MySQL query cache. When a given query sent to the MySQL server is matched with one whose results are already stored in the query cache, the results can be sent back to the client without any further processing, and without communicating with data nodes across the network. The fact that these tasks do not need to be performed in such cases can bring about tremendous savings in both processor and network overhead and thus a major increase in performance.

Another means of reducing overhead on SQL nodes is through the use of batched inserts. This is because of less network traffic between the MySQL server and client, as well as the fact that the NDB storage engine is optimized to take advantage of batched operations.

Proper tuning of parameters relating to MySQL's statistics cache can have a positive effect on performance. This is also true of configuration parameters relating to prefetching, transaction use, and other aspects of MySQL and NDB behavior.

All of these areas are covered in greater depth in Section 8.4, “SQL Nodes and MySQL Cluster Performance”.

#### 8.1.2.2. Performance and Data Nodes

As we have seen elsewhere in this book, data nodes are responsible for the actual storage and retrieval of data in NDB tables. The NDB storage engine's implementation of data storage and retrieval is quite different from what is seen in more conventional storage engines such as MyISAM and InnoDB. In most cases, multiple data nodes are involved, leading to distributed storage and retrieval of data.

NDB tables are always partitioned across all available data nodes. Each data node stores a portion of the total amount of rows in the table. This type of partitioning is known as *horizontal partitioning*, and is discussed in detail in Section 3.6, “Data Partitioning”.

Partitioning affects performance in a number of ways. In a number of cases, it can increase response time. The data node that handles an NDB API call might need to delegate the call to another data node actually containing the row. This increases response time because network communication is required to transfer the data from one data node to the other. In addition, the delegation itself requires some processing time on the data node that performs it.

In other cases, however, partitioning can actually decrease response time and increase throughput. Partitioning data has the advantage that some operations can be performed in parallel — that is, on differ-

## 8.2. Network Issues

---

ent data nodes simultaneously. Another advantage of partitioning is that, because a given data node contains only a portion of the data, there is less data for each data node to process, leading to a decrease in response time.

A MySQL Cluster is usually set up in order to provide redundant storage, storing each row on multiple data nodes. Redundant storage is implemented by synchronous replication, and is discussed in detail in Section 3.2, “Cluster Internal Replication”.

The chief consequence of internal replication for performance is that the replication process takes time. Increasing the number of replicas has a tendency to slow down the addition, modification and deletion of data. That is because each operation has to be performed for each replica, and this needs to be synchronized for all data nodes. In many cases, this involves network traffic, since data needs to be transferred to all nodes maintaining a replica.

The fact that the NDB storage engine can store data in memory also represents a great difference as compared to the storage methods available in a single MySQL server. This is also true of the methods used by NDB to access and retrieve data.

While MySQL Cluster implements indexes, its treatment of them differs in many ways from how a single MySQL server does so. In-memory storage of indexes and the use of hash indexes each has a significant bearing on performance.

For a general discussion of the different types of indexes available in the NDB storage engine, see Section 7.5, “Indexes in NDB Tables”. How indexes can be used to speed up retrieval of data is discussed later in this chapter, in Section 8.3, “Data Access Methods”.

## 8.2. Network Issues

The network is a recurring theme in many MySQL Cluster performance issues. MySQL Cluster has a shared nothing architecture, which means that all cluster nodes need to maintain constant communication with one another to remain synchronized. This results in heavy usage of the network connecting the nodes, so the network can become a bottleneck for MySQL Cluster performance.

Many of the methods that can be used to improve MySQL Cluster performance are in some way related to either speeding up or decreasing network traffic in one way or another. A few general methods will be discussed in this section, but later portions of this chapter also refer to various network issues when explaining other methods of increasing performance.

### 8.2.1. Latency and Bandwidth

In all networking applications, there are two network characteristics to consider: *latency* and *bandwidth*.

There is no such thing as instant communication; it always takes some amount of time for a message to



---

## 8.2.2. Transporters

---

travel from source to destination. *Latency* is the term used to indicate the delay that occurs between the sending and receiving of a message via some communication channel. As such, latency can be an important factor in determining response time in MySQL Cluster, and is something that you should always strive to minimize.

A primary determining factor of latency is the networking hardware. For MySQL Cluster, reducing latency is mainly a matter of choosing the appropriate hardware.

Increasing physical distance (length of cables) does increase latency, but for most practical setups this effect is insignificant when compared to other factors that affect MySQL Cluster's latency.

*Bandwidth* is the amount of data that can travel through a communication channel within a given amount of time. As such, bandwidth can affect MySQL Cluster throughput directly, and higher bandwidth allows for higher throughput.

MySQL cluster processes such as replication can consume substantial bandwidth on the network connecting the data nodes. Likewise, a large amount of data can flow between SQL nodes and data nodes; bandwidth may be a limiting factor in this case as well.

As is the case with latency, hardware is also an important determinant for bandwidth.

## 8.2.2. Transporters

As discussed elsewhere in this Guide (see Section 2.1.2, “Transporters”), MySQL Cluster uses an abstraction layer known as a *transporter* to handle communication between nodes. This allows multiple implementations for handling inter-node communication to co-exist and to be handled transparently. This offers an advantage over a single generic implementation, as it becomes possible to utilize specific networking hardware fully. Deploying such hardware can both reduce latency (and therefore, decrease the minimum response time) as well as increase bandwidth (thereby allowing higher throughput).

MySQL Cluster currently implements the following transporters:

- **TCP/IP.** Using this transporter, MySQL cluster nodes can communicate using an ordinary 100 Mb/s or 1 Gb/s TCP/IP network.
- **SHM.** This transporter uses shared memory connections. The advantage of this type of transporter is that this is very fast. The disadvantage is that not all hardware supports it, and that shared memory can act as a single point of failure. It is also debatable whether a setup with shared memory is still a true shared nothing architecture.
- **SCI.** Using this transporter, MySQL cluster nodes can communicate using Scalable Coherent Interface interconnections. This is used primarily to connect the data nodes with each other. Such interconnections can bring about gains of up to 10 times lower latency and higher throughput.

In most cases, MySQL Cluster is set up using the TCP/IP transporter. Setting up other types of transporters is outside the scope of the certification exam (and of this book). However, you should know

what transporters are and be aware of how using different transporters can affect performance.

### 8.2.3. Distance Between Nodes

Distance between nodes, or rather, the distance that is traversed by cluster messages has a measurable impact on latency: The greater the distance, the greater the latency.

Physical separation of redundant components minimizes the chance that an extreme condition such as a fire, a plane crash, or an earthquake causes their simultaneous failure. From the standpoint of high availability, it does make sense to place different data nodes from the same node group in different locations, since, in theory, this would help to eliminate location as a single point of failure.

However, increasing the distance between nodes implies increasing network latency. Currently there is little data available about the requirements for geographical separation of MySQL cluster nodes. Therefore, it is not recommended to separate MySQL cluster nodes widely. Apart from the fact that high latency leads to poor performance, it can also jeopardize availability. This is because excessive latency may result in missed heartbeats, which at some point may become indistinguishable from network partitioning, which leads to nodes being shut down.

## 8.3. Data Access Methods

MySQL Cluster data nodes have at their disposal a number of methods for retrieval of data, whose availability and suitability depend on the existence of a given type of index. Often, cluster performance hinges on creating the proper index to speed up a particular query. These data access methods are listed here:

- Primary key lookup
- Unique index lookup
- Ordered index range scan
- Full table scan

A specific NDB API call is available for each of these methods. The method used to retrieve data in a given situation is at the discretion of the API node. Data nodes can only respond to NDB API calls, and it is the responsibility of an API node to issue whatever NDB API call is necessary to serve the application. In a typical MySQL cluster, the API nodes are MySQL servers, which translate SQL statements into appropriate NDB API calls.

The following sections discuss the different data access methods and their performance aspects. For each access method, we provide an example of a typical case where the particular method is used by an SQL node.

### 8.3.1. Primary Key Lookups

### 8.3.1. Primary Key Lookups

---

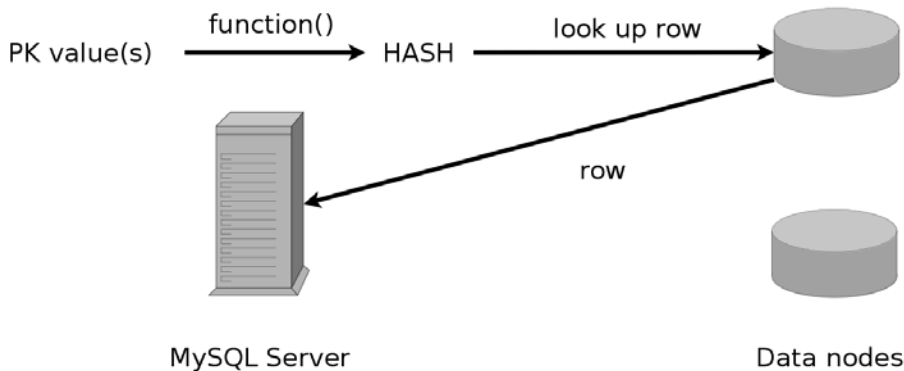
This access method uses the primary key of the NDB table. Recall that NDB tables always have a primary key — if one is not explicitly defined, then the NDB table has an implicit, hidden primary key. The primary key of an NDB table is implemented using a hash index, in which the hash value is used as the partitioning key to distribute the rows over the data nodes.

For more information concerning the creation and implementation of NDB primary keys, see Section 7.5.4.1, “Primary Key”, and Section 7.5.1, “Hash Indexes”.

A primary key lookup involves the following steps:

- An API node uses the values from the primary key column (or columns) to compute a hash value. Because the primary key hash value is also the partitioning key, the API node now knows which of the data nodes in the cluster stores the corresponding row (if the row exists) in its primary fragment.
- The hash is used to obtain the physical address of the row from an in-memory hash table located on the data node. This in-memory hash table is the structure that actually implements the unique index backing the primary key. At this point, the data node may discover that the row does not exist, in which case it sends a notification of the fact to the API node, concluding the process.
- If the hash table contains an entry for the hash, the physical address of the row is used to retrieve the row. This row is then sent to the API node, concluding the process.

#### Primary Key Lookup



Primary key lookups can be very fast because the physical location of the row can be retrieved using only a single lookup from an in-memory hash table. For an NDB table using in-memory storage, the physical location is the actual address (a pointer) where the row data is stored. For an NDB table using disk-based storage, the physical location is a value that can be used to locate the row on disk.

### 8.3.1. Primary Key Lookups

---

Primary key lookups are subject to a number of limitations. These limitations are not unique to NDB cluster primary keys; rather, these limitations apply to hash indexes in general.

#### 8.3.1.1. All Values Requirement

A hash index lookup depends on the calculation of the hash value and a subsequent lookup in the in-memory hash table. This means that all column values making up the primary key are required in order to do the lookup. If one or more values are missing, the hash cannot be calculated, and no lookup can be performed. Hash indexes can only be used efficiently to implement unique indexes. The consequence is that a single primary key lookup results in at most one row.

Consider the following table definition:

```
CREATE TABLE City (  
  CountryCode CHAR(3) NOT NULL,  
  District    CHAR(20) NOT NULL,  
  Name        CHAR(35) NOT NULL,  
  PRIMARY KEY (CountryCode,District,Name)  
    USING HASH  
) ENGINE=NDB
```

This table defines a composite primary key using all three of the table's columns, and employs a `USING HASH` clause as part of the definition. This ensures that only a single hash index is created. Without the `USING HASH` clause, an additional non-unique ordered index would also be created. This behavior is described in more detail in Section 7.5.4, “SQL Mapping”, and in Section 7.5.4.3, “Creating Indexes with `USING HASH`”.

Consider this query:

```
SELECT CountryCode, District, Name  
FROM   City  
WHERE  CountryCode = 'USA'  
AND    District    = 'New York'  
AND    Name        = 'New York'
```

Here, all of the values comprising the primary key are supplied. This query is certain to use a primary key lookup to retrieve the row. We can verify this by using this query in an `EXPLAIN` statement, as shown here:

```
EXPLAIN  
SELECT CountryCode, District, Name  
FROM   City  
WHERE  CountryCode = 'USA'  
AND    District    = 'New York'  
AND    Name        = 'New York'
```

This computes the query plan and returns it as the following result set:

### 8.3.1. Primary Key Lookups

---

```
***** 1. row *****
  id: 1
  select_type: SIMPLE
  table: City
  type: const
possible_keys: PRIMARY
  key: PRIMARY
  key_len: 58
  ref: const,const,const
  rows: 1
  Extra:
```

The key column shows which one of the indexes listed in the `possible_keys` column was used to access the data. The value is `PRIMARY`, indicating that the primary key was indeed used.

#### 8.3.1.2. Insufficiency of Primary Key Prefixes

The necessity of calculating a hash value makes it impossible to use the primary key lookup access method for a query that supplies only part of the primary key values. Consider the following query:

```
SELECT CountryCode, District, Name
FROM   City
WHERE  CountryCode = 'USA'
AND    Name        = 'New York'
```

This query does not use the primary key lookup access method. It is not possible to use that method, because this query does not supply all values that make up the table's primary key.

Even when a comparison is supplied for each indexed column, it may be that a hash index is still not utilized; multiple comparisons can be used only if they can be taken together and applied as a whole. In other words, a comparison for *each* column must be present, and the comparisons must be combined using the `AND` logical operator. Consider the following query:

```
SELECT CountryCode, District, Name
FROM   City
WHERE  CountryCode = 'USA'
AND    District    = 'New York'
OR     Name        = 'New York'
```

While this query supplies a value for each of the columns in the primary key, it cannot use the primary key for accessing all of the data. This is due to the presence of `OR` operator combining the comparison on the `Name` column with the those on the `CountryCode` and `District` columns. The `WHERE` as it appears here essentially defines two different cases, neither of which supply the values for all of the indexed columns. Because of this, this query is unable to utilize the primary key.

Theoretically, it would be possible to retrieve the row that corresponds to the index entry that matches each of the constants supplied in the query. However, this would retrieve at most one row that satisfies the `WHERE` condition. A separate action to retrieve those rows for which the condition `CountryCode`

### 8.3.1. Primary Key Lookups

---

`= 'USA' AND District = 'New York'` is true would be required, and a similar action would be required too to find all other rows for which the condition `CountryCode = 'USA'` is true. Both these actions cannot be performed by utilizing the primary key index, as neither of these sub-conditions supply all values for the primary key columns.

#### Important

Hash indexes cannot be used if values are supplied for only some but not all primary key columns, not even when the supplied values form a prefix of the entire primary key.

We can speak of an index prefix being used in a query when:

- The query contains a condition that combines one or more (but not all) of the columns that make up the index
- These columns are combined in a manner that is equivalent to a combination using only the AND operator
- These columns appear consecutively in the column list of the index definition
- These columns include the first column in the column list of the index definition

The following example illustrates the concept of an index prefix:

```
SELECT CountryCode, District, Name
FROM   City
WHERE  CountryCode = 'USA'
AND    District    = 'New York'
```

The definition of the `City` table includes a `PRIMARY KEY` clause, defining a primary key constraint (and, implicitly, a hash index) on `(CountryCode, District, Name)`. The query contains a condition that combines the `CountryCode` and `District` columns using the AND operator. These two columns constitute a prefix of the index that is used to implement the primary key constraint, because they appear consecutively in the definition of the primary key and the `CountryCode` column is indeed the first column in the index definition.

Index prefixes are commonly used to scan BTree indexes in tables using the MyISAM or InnoDB storage engine, but they cannot be used with hash indexes. When designing a table or analyzing a query, it is not uncommon to make the mistake of forgetting that a hash index cannot be used if only a prefix is present.

Recall that we created the primary key on the `City` table with a `USING HASH` clause, which prevented the creation of an additional ordered index. Had the `USING HASH` clause been omitted, an additional non-unique ordered index would have been created. In that case, this particular query would have been able to use the ordered index. This type of situation is discussed further in Section 8.3.3.3, “Partial Use of Ordered Indexes”.

#### 8.3.1.3. Equality Comparison

---

### 8.3.1. Primary Key Lookups

---

Another condition that follows from the necessity of calculating the hash value for primary key lookups is that they can be used only to test for equality. This means that the primary key lookup access method can be used only for those queries that specify that all of the supplied values are equal to primary key column values. Queries that request some kind of range comparison cannot use the primary key lookup access method.

Consider the following query:

```
SELECT CountryCode, District, Name
FROM   City
WHERE  CountryCode = 'USA'
AND    District    = 'New York'
AND    Name        >= 'New York'
```

The query plan shown by using EXPLAIN reveals that a primary key lookup is not used:

```
***** 1. row *****
      id: 1
    select_type: SIMPLE
      table: City
       type: ALL
possible_keys: PRIMARY
       key: NULL
      key_len: NULL
         ref: NULL
        rows: 126
      Extra: Using where
1 row in set (0.00 sec)
```

We can see that the primary key is considered as a *possible* key, but unlike the case in previous examples, it is not used here. This is because of the greater-than-or-equals comparison on the Name column. Even though all values for the primary key are supplied, it does no good to calculate a hash from it. If a hash would be calculated using those values, it could be used to retrieve at most the one row that corresponds to it. However, additional actions would be needed for retrieving any other rows that satisfy the condition. Therefore, the primary key is not used to execute this query.

Other comparison operators such as LIKE and BETWEEN . . . AND are likewise incompatible with the primary key lookup method. Each these operators specifies a range of values; none of them can utilize the primary key or, for that matter, any other hash index. Ordered indexes can handle these range queries; this is discussed in more detail in Section 8.3.3.5, “Non-Equality Comparisons”.

#### 8.3.1.4. Retrieving Multiple Rows

It is possible to get the impression that primary key lookups are suitable only for retrieving a single row per query. However, this is not the case. Consider the following query:

```
SELECT CountryCode, District, Name
FROM   City
WHERE  (CountryCode, District, Name) = ('NLD','Noord-Holland','Amsterdam')
```

## 8.3.2. Unique Index Lookups

```
OR      (CountryCode, District, Name) = ('ARG','Distrito Federal','Buenos Aires')
OR      (CountryCode, District, Name) = ('USA','New York','New York')
```

This uses exactly the same primary key lookup method as discussed in Section 8.3.1.1, “All Values Requirement”. Since all values for the primary key are supplied at all times, and all comparisons test for equality, there is no reason why the primary key lookup should not be employed.

### 8.3.2. Unique Index Lookups

This access method uses a unique hash index.

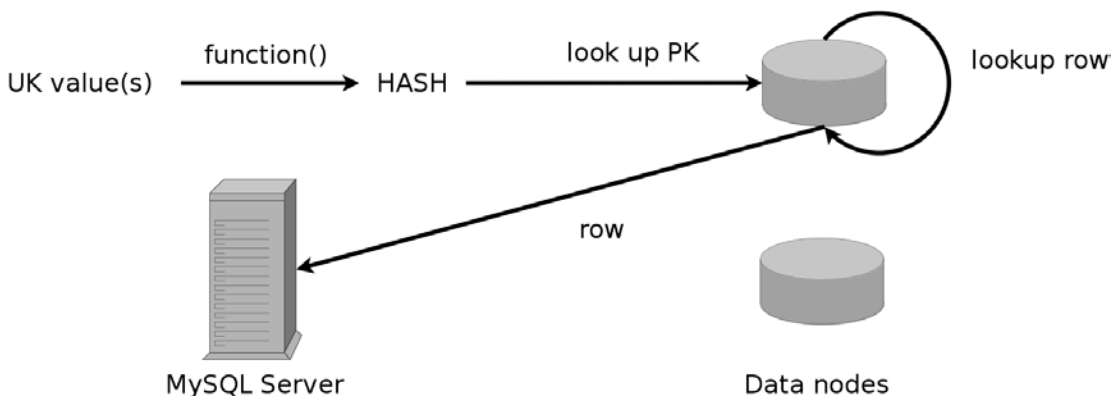
To understand the unique lookup data access method, it is necessary to review some facts about the implementation of unique indexes on NDB tables. As described in Section 7.5.4.2, “Unique Indexes”, the NDB storage engine implements unique indexes by creating a hidden supporting table that defines the columns in the unique index definition as its primary key, mapping this to the primary key of the original table. This implementation directly affects the method that is used to read unique indexes.

#### 8.3.2.1. Chained Primary Key Lookups

The unique index lookup data access method, as discussed in Section 8.3.1, “Primary Key Lookups”, can be thought of as actually performing two primary key lookups in succession. This process consists of the following steps:

1. The values corresponding to the unique index are used to perform a primary key lookup on the supporting table. This obtains the primary key values for the original table.
2. The primary key values obtained in the previous step are used to perform a primary key lookup on the original table.

### Unique Index Lookup





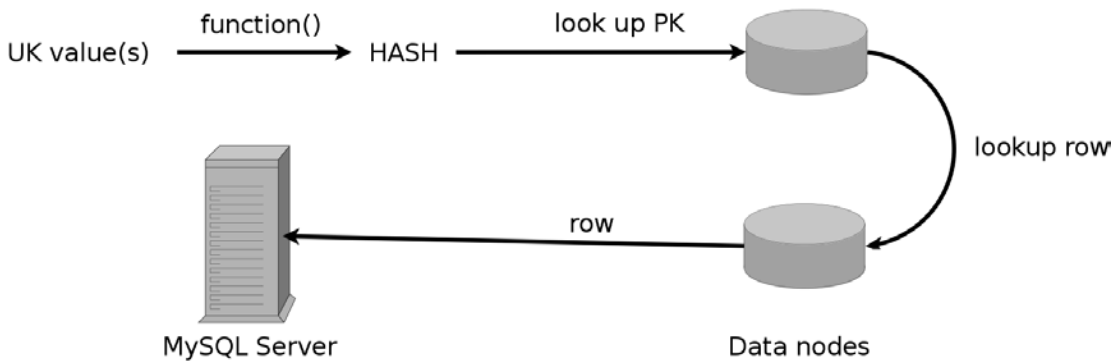
### 8.3.3. Index Scans

---

The primary key lookups making up the unique index lookup are subject to the same limitations as ordinary primary key lookups. All values must be supplied, and the comparisons used must test for equality.

However, some additional considerations apply. The supporting table is an NDB table, and as such, it is partitioned across the data nodes. This means that the two lookups need not be confined to a single data node. The second primary key lookup may need to be performed on a different data node from the first one. Since each data node is normally deployed on its own host computer, the second lookup requires an additional network “hop”, as shown here:

#### Unique Index Lookup may require another network hop



The unique index lookup data access method is slower than the primary key lookup data access method. Because it involves two primary key lookups, it takes approximately twice the amount of time needed for a single primary key lookup.

### 8.3.3. Index Scans

The data access methods discussed earlier involved hash indexes only. The *index scan* access method relies on the presence of what is known as an *ordered index*, which is discussed in more detail in Section 7.5.2, “Ordered Indexes”.

#### 8.3.3.1. Ordered Indexes vs Hash Indexes in NDB

To understand how and why a particular index may or may not be used to speed up a particular query, you should be aware of the differences between ordered indexes and hash indexes. These differences are not unique to MySQL Cluster, but apply to hash and ordered index implementations generally.

Hash indexes are particularly suitable for the speedy retrieval of a single row explicitly identified by

---

### 8.3.3. Index Scans

---

specific values for all columns in the index. It is also relatively easy for a hash index implementation to enforce uniqueness. Hash indexes implemented by the NDB engine are always unique indexes, and all unique indexes in NDB are implemented using a hash index. NDB ordered indexes are always non-unique indexes.

Ordered indexes are particularly suitable for retrieving the rows corresponding to a specific range of index entries, because index entries that are in the same range are close to one another in the sorting order. Since ordered indexes organize their entries according to this sorting order, it is possible to iterate through these entries in a highly efficient manner.

The difference between hash indexes and ordered indexes is discussed in more detail in Section 7.5.3, “Comparison of Hash and T-tree Indexes”. In particular, you may wish to review the illustration included in that discussion, which you may find helpful in understanding ordered index scans in the next section of this chapter.

#### 8.3.3.2. Ordered Index Scans

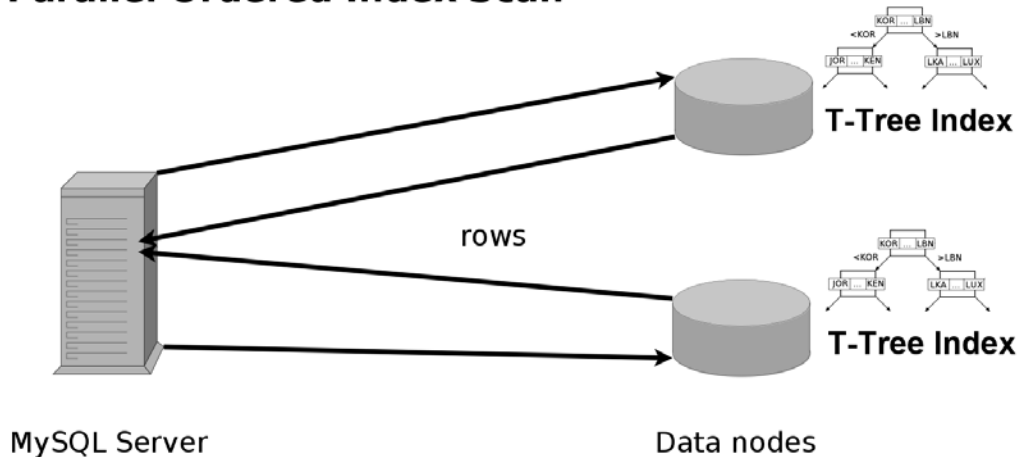
The process of using an ordered index to find a particular entry or set of entries is generally referred to as an *ordered index scan*. This process is called “scanning” because it is not possible to know beforehand how many steps are required to find a particular entry. (It can be shown that this is a finite number of steps; however, we do not offer a proof of this here.) For the primary key access and unique index access methods, how many steps are required either to find an entry or to determine that no such entry exists is always known, and the number of steps is always the same. In these cases, the term “lookup” is used, rather than “scan”.

The tree-like organization of the T-tree index dictates how it can be used to find an entry with a particular value. The scanning process starts at the root node. The value for which we are searching is compared with the first value in the array. If the value is smaller than the first value, the process needs to be repeated on the left branch. If it is larger than the last value, the process needs to be repeated on the right branch. When a node is found where the value is in between the values found for the first and last entries in the array, we must iterate through the array until either the desired entry is found, or it is certain that the entry should have been found already (that is, that there can be no match).

Ordered indexes can be used to retrieve a single row, but in general this is slower than primary key access. This is not surprising, given that locating a single record might require descending through the entire tree and iterating through an array. This can still be a speedy process, but it will almost always be slower than the single lookup required for primary key access, or even for the chained lookups required by unique index access.

A very powerful feature of NDB index scans is that they occur in parallel. A request from an API node to scan the index is sent to all data nodes which will then simultaneously scan the index.

## Parallel Ordered Index Scan



The simultaneous scan of all data nodes is a form of parallelism. The benefit of parallelism increases as increasingly larger volumes of data are more and more partitioned by adding more data nodes. Because of parallelism, and ordered index scan is sometimes called a *Parallel Ordered Index Scan*.

### 8.3.3.3. Partial Use of Ordered Indexes

As described in Section 8.3.1.2, “Insufficiency of Primary Key Prefixes”, hash indexes need values for all columns that make up the index in order to be utilized to query data. Ordered indexes can be used even if not all values are available for the indexed columns. Consider this CREATE TABLE statement:

```
CREATE TABLE City (
  Name char(35) NOT NULL,
  CountryCode char(3) NOT NULL,
  District char(20) NOT NULL,
  PRIMARY KEY (CountryCode,District,Name) USING HASH,
  INDEX I_City (CountryCode,District,Name)
) ENGINE=NDB
```

The table definition includes a non-unique index called `I_City`, and a `PRIMARY KEY`, both of which are created on the columns `CountryCode`, `District`, `Name`. The `PRIMARY KEY` includes the `USING HASH` clause to prevent the implicit creation of the ordered index. Instead, we explicitly created the `I_City` index, which has a structure that is identical to that of the index that would have been created implicitly if the `USING HASH` clause would not have been included in the `PRIMARY KEY` definition. Now consider the following query:

```
SELECT COUNT(*)
FROM City
WHERE CountryCode = 'USA'
AND District = 'California'
```

### 8.3.3. Index Scans

---

In this query, values are supplied for the `CountryCode` and `District` columns. We saw earlier that the hash index that implements the primary key cannot be used to perform this query. Inspecting the output from `EXPLAIN` for this query shows the following plan:

```
***** 1. row *****
      id: 1
    select_type: SIMPLE
      table: City
        type: ref
possible_keys: PRIMARY,I_City
         key: I_City
        key_len: 3
          ref: const
         rows: 10
       Extra: Using where
1 row in set (0.00 sec)
```

We can see in the `possible_keys` column of the `EXPLAIN` output that both the `PRIMARY KEY` and the `I_City` index are initially considered. The `key` column indicates that the `I_City` is actually used.

We deliberately created the `PRIMARY KEY` with the `USING HASH` clause and an explicit ordered index to be able to demonstrate this query plan. Consider for a moment what would happen if we created the `PRIMARY KEY` without the `USING HASH` clause, and omitted the explicit `I_City` index. We know that under the hood, an ordered index just like the `I_City` index would have been present. It would have been utilized too for the previous query. However, in the `EXPLAIN` output, we would have been completely unable to distinguish which one of the two index implementations was used to perform the query. With regard to the `EXPLAIN` output, it is just as if there is really only one index that has the properties of both a hash index as well as an ordered index.

This means that one always has to take great care in interpreting query plans on NDB tables. One always needs to have access to the actual table and index definitions in order to understand the query plan. In some cases, it might be wiser not to rely on the implicit ordered index at all, and always to create `PRIMARY KEY` and `UNIQUE` indexes with the `USING HASH` clause. This will add the burden of having to create ordered indexes explicitly where needed, but it has the advantage that, it will often be easier to interpret query plans because each physical index at the data node level corresponds to one logical index at the SQL node level. Also, this makes it possible to drop and recreate ordered indexes without having to drop and recreate the corresponding hash index.

#### 8.3.3.4. Prefixes

We just saw that an ordered index can be used even if the query does not supply a value for all of the indexed columns. Now consider the following query:

```
SELECT COUNT(*)
FROM   City
WHERE  District = 'California'
AND    Name = 'Santa Clara'
```

---

### 8.3.3. Index Scans

---

Again, two indexed values are supplied, namely for the `District` and the `Name` columns. However, in this case, the query plan looks very different:

```
***** 1. row *****
      id: 1
    select_type: SIMPLE
      table: City
      type: ALL
possible_keys: NULL
  key: NULL
  key_len: NULL
   ref: NULL
  rows: 4078
 Extra: Using where
```

Here, none of the available indexes have even been considered. The query is resolved without accessing any indexes at all, which means that a full table scan is used instead. (Table scans are discussed in Section 8.3.4, “Table Scans”).

So, even though it is true that — in some circumstances — an ordered index can be used when values are supplied for only a subset of the indexed columns, it is not always the case. An index scan can be performed only if the supplied index part forms a *prefix* of the entire index. Any subset of adjacent index columns that includes the first column of the index is a prefix.

In this case, the ordered index is defined on the columns: `CountryCode`, `District`, `Name`, in that order. To utilize such the ordered index, a value must be supplied for a number of adjacent columns in the index, starting from the first column. So, a value supplied for `CountryCode` is a prefix, and a value pair supplied for both `CountryCode` and `District` also form a prefix.

Supplying a value for only `District` or for only `District` and `Name` does not constitute a prefix because, even though the columns are adjacent, the subset of columns does not include the first column. Likewise, supplying values for `CountryCode` and `Name` also does not constitute a prefix in its entirety. Even though this subset of columns includes the first column in the index, the columns are not adjacent; that is, there is a “gap” between both columns. In this case, only the set of columns up to the gap can be used as prefix.

#### 8.3.3.5. Non-Equality Comparisons

In Section 8.3.1.3, “Equality Comparison”, we described why hash indexes can be used only in equality comparisons. The radically different organization of ordered indexes allows them be used with other comparison operators, such as less than, greater than and even `LIKE`. For example, we used the following query to illustrate that it could not benefit at all from a the hash index:

```
SELECT CountryCode, District, Name
FROM   City
WHERE  CountryCode = 'USA'
AND    District    = 'New York'
AND    Name        >= 'New York'
```

### 8.3.3. Index Scans

---

The output from EXPLAIN shows how the ordered index is utilized for this query:

```
***** 1. row *****
      id: 1
    select_type: SIMPLE
      table: City
      type: range
possible_keys: PRIMARY,I_City
      key: I_City
     key_len: 58
        ref: NULL
       rows: 10
     Extra: Using where
```

The particular type of index utilization is called a *range scan*. This is indicated by the *range* in the *type* column of the EXPLAIN output. The following comparison operators can result in a range scan:

- <, <=, >, and >=
- BETWEEN ... AND
- LIKE, depending on the pattern that is matched

The ... BETWEEN ... AND ... is completely equivalent to a ... >= ... AND ... <= ... expression. The LIKE operator is a special case.

The LIKE operator tests to see if the value of the string expression appearing on the left hand side of the LIKE keyword matches the pattern appearing on the right hand side. The pattern is also a string expression. The string value of the pattern is a collection of literal characters and optionally a number of wildcard characters. Consider the following query:

```
SELECT CountryCode, District, Name
FROM   City
WHERE  CountryCode = 'USA'
AND    District    = 'California'
AND    Name        LIKE 'A%'
```

The literal string A% denotes a pattern that matches any string that starts with the character A and is followed by zero or more arbitrary characters, denoted by the % *wildcard* character. Any LIKE pattern whose string value does not start with a wildcard can be transformed into an range expression, and this is exactly what the MySQL Server will do. For example, the previous query could be optimized to something equivalent to this:

```
SELECT CountryCode, District, Name
FROM   City
WHERE  CountryCode = 'USA'
AND    District    = 'California'
AND    Name        >= 'A'
AND    Name        < 'B'
```

### 8.3.4. Table Scans

---

In this particular case of `LIKE` the ordered index can be used to perform a range scan. A `LIKE` comparison can not be rewritten to a range expression if it starts with a wildcard. Consider the following query:

```
SELECT CountryCode, District, Name
FROM City
WHERE CountryCode = 'USA'
AND District = 'California'
AND Name LIKE '%A'
```

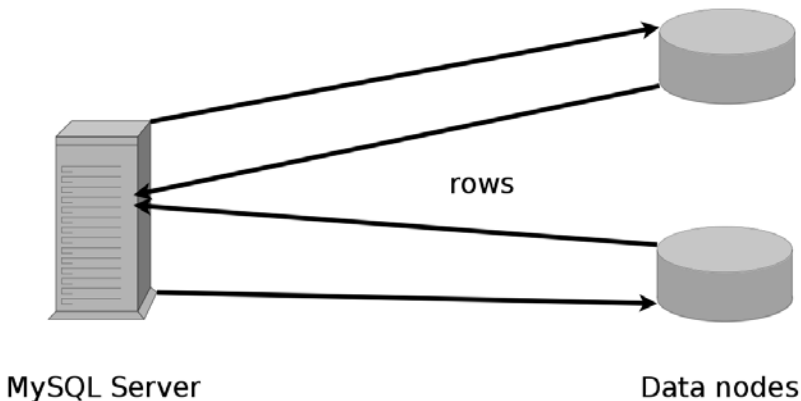
Here the pattern starts with the `%` wildcard, and the `LIKE` expression cannot be optimized to a range expression. However, this does not have to mean that ordered index is not used anymore. In this particular case, the values supplied for the `CountryCode` and `District` columns still form a leftmost prefix of the ordered index, and thus the ordered index can still be utilized for this query.

### 8.3.4. Table Scans

The previously described data access methods all relied on the existence of an index. Indexes are particularly good for the quick retrieval of a relatively small number of rows. Sometimes, a query requires a lot of data to be fetched. In such cases, using an index might slow query execution down. In other cases, an index is not even present, or index is present but it cannot be utilized for some reason. In these cases, a *table scan* is performed by iterating through all the rows in the table.

Full table scans on NDB tables are executed in parallel: all data nodes scan their primary fragment simultaneously, and return the results to the SQL node. Therefore this method is sometimes called the *parallel full table scan* data access method.

#### Parallel full table Scan



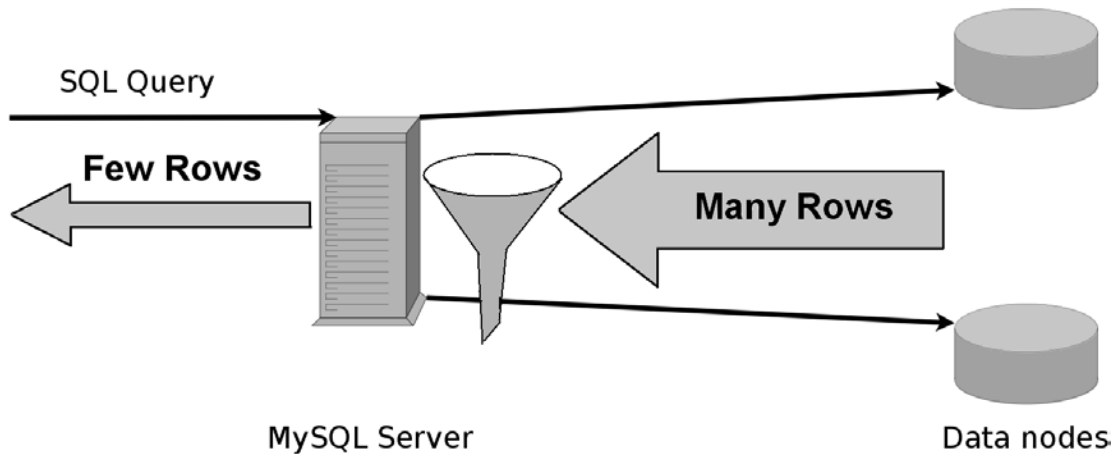
Full table scans are the best method for retrieving a large set of rows.

### 8.3.5. Scans with Engine Condition Pushdown

In our discussion of data access methods, we have already seen how an index can be used for the efficient evaluation of conditions appearing in the WHERE clauses of queries. We have also seen examples of queries where it was not possible to use an index to evaluate the condition in the WHERE clause.

If a WHERE clause cannot be evaluated by means of an index, the condition needs to be applied in some other way. In some cases, part of the condition can be evaluated by using an (ordered) index; in that case, the remainder of the condition still needs to be applied in another way. By default, the SQL node processes the rows received from the data nodes, and if necessary, the evaluation of WHERE conditions is performed by the SQL node.

#### WHERE condition processed in the SQL node

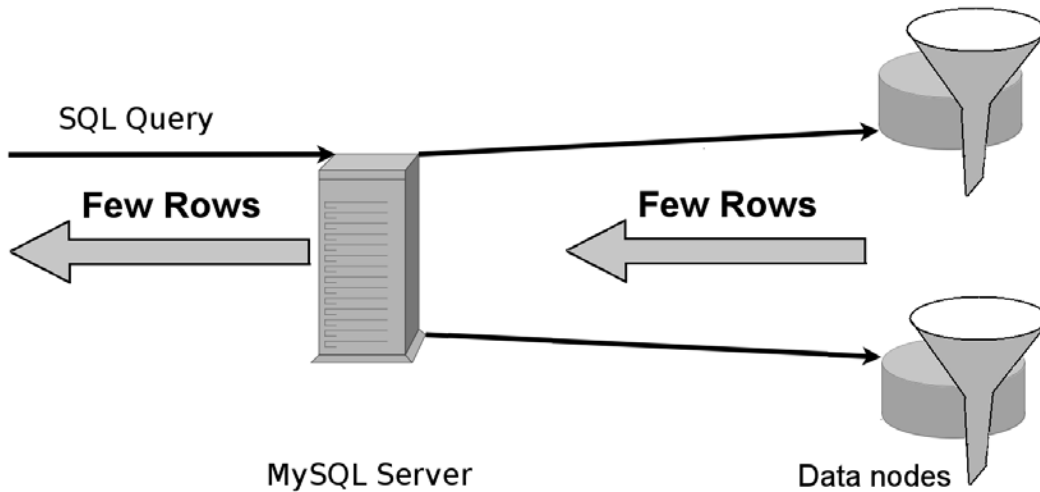


Since SQL nodes and data nodes are best deployed on separate physical hosts, rows retrieved by the data nodes must be sent across the network to the SQL nodes in order to undergo further processing, which might include the evaluation of WHERE conditions.

Performance is very likely to be affected if executing the query involves sending many rows from the data nodes to the SQL nodes. In these cases, performance may be improved by using *condition pushdown* (also referred to as *engine condition pushdown*). When using condition pushdown, the conditions from the WHERE clause are sent to the data nodes as part of the request to scan a table's ordered index. The data nodes then evaluate the condition in order to determine whether a row should be returned to the SQL node.



## WHERE condition pushed down to the data nodes



Evaluating such conditions on the data nodes rather than on the SQL node can improve performance for two reasons:

1. Applying the condition might result in relatively few qualifying rows having to be sent back to the SQL node. Such rows still need to travel over the network, but because there is less to transfer as compared to a method that does not use condition pushdown, this may improve performance.
2. All data nodes can perform the evaluation of the `WHERE` condition in parallel. This is also likely to improve performance.

### 8.3.5.1. Enabling and Disabling Condition Pushdown

By default, condition pushdown is not applied, and must be explicitly enabled via the MySQL server variable `engine_condition_pushdown`. This can be done by setting `engine_condition_pushdown` to 1 or ON. Either of the following statements enables condition pushdown for the current session:

```
SET SESSION engine_condition_pushdown = 1;
```

or

```
SET SESSION engine_condition_pushdown = ON;
```

Alternatively, you can use either of these two statements:

```
SET @@session.engine_condition_pushdown = 1;
```

### 8.3.5. Scans with Engine Condition Pushdown

---

or

```
SET @@session.engine_condition_pushdown = ON;
```

Of course, since session scope is the default, you can simply use either of these two statements:

```
SET engine_condition_pushdown = 1;
```

or

```
SET engine_condition_pushdown = ON;
```

Engine condition pushdown can also be enabled for all sessions. This can be done by using SET GLOBAL rather than just SET:

```
SET GLOBAL engine_condition_pushdown = 1;
```

or

```
SET GLOBAL engine_condition_pushdown = ON;
```

(Again, either one of the previous two statements is sufficient.) Of course, since @@global notation can also be employed (just as with any other MySQL server variable which is to be set server-wide), either one of these two statements also works to set engine\_condition\_pushdown globally:

```
SET @@global.engine_condition_pushdown = 1;
```

or

```
SET @@global.engine_condition_pushdown = ON;
```

You can also specify that condition pushdown is to be used by including either one of the following two lines in the my.cnf file used by the SQL node:

```
...
engine_condition_pushdown=1 # push down conditions to the storage engine
...
engine_condition_pushdown=ON # also pushes down conditions to NDB
...
```

The current setting of the engine\_condition\_pushdown setting can be found out using a query like this one:

```
mysql> SELECT @@engine_condition_pushdown;
+-----+
| @@engine_condition_pushdown |
+-----+
| 1 |
+-----+
1 row in set (0.00 sec)
```

#### Note

The value of @@engine\_condition\_pushdown is the value for the current session, and may not necessarily be the same as the global value; use GLOBAL or @@global if this is a concern.

See *SET Syntax*, in the MySQL 5.1 Manual [<http://dev.mysql.com/doc/refman/5.1/en/set-option.html>], for more information.

Condition pushdown can be disabled by setting engine\_condition\_pushdown to 0 or OFF, using any of the methods previously described.

Condition pushdown is not specific to the NDB storage engine; it is actually a feature of the MySQL storage engine API. As such, condition pushdown may in theory be applied with any storage engine. However, storage engines are not required to implement condition pushdown, and many of them currently do not do so. In fact, NDB is the only storage engine offered by MySQL AB (as of MySQL 5.1) that implements condition pushdown.

If the storage engine implements condition pushdown, then the storage engine can accept conditions which are then pushed down and evaluated by the storage engine rather than by the condition evaluation engine in mysqld. Of course, the particular implementation provided by the storage engine determines the specific characteristics of the evaluation of the pushed down conditions. For example, the NDB engine is capable of offering parallel condition evaluation when there are multiple data nodes.

#### 8.3.5.2. Using Condition Pushdown Effectively

In this section, we show how you can use condition pushdown to improve performance. For this test, we assume the existence of the world\_ndb database described in Section A.1, “Schema Normally Used for Examples and Exercises”, and of a cluster having two data nodes and two replicas, like the one described in Section A.3, “Configuration Setups Used for Examples and Exercises”.

In order to prevent cached results from influencing the outcome, we first disable the query cache:

```
mysql> SET query_cache_type = OFF;  
Query OK, 0 rows affected (0.00 sec)
```

This is desirable because we are trying to test the effect of engine\_condition\_pushdown specifically, and we want to eliminate as many source of potential interference with this as possible.

In the next series of examples, we use a number of queries following the pattern shown here:

```
SELECT COUNT(*)  
FROM world_ndb.City  
WHERE Name  
BETWEEN 'A'  
AND 'Z';
```

### 8.3.5. Scans with Engine Condition Pushdown

---

To perform actual measurements, we substitute real city names for the literal Z.

The `world_ndb.City` table is defined as shown here:

```
CREATE TABLE City (
  ID int NOT NULL AUTO_INCREMENT,
  Name char(35) NOT NULL,
  Country char(3) NOT NULL,
  District char(20) NOT NULL,
  Population int NOT NULL,
  PRIMARY KEY (ID)
) ENGINE=NDB;
```

Because the `Name` column of the `world_ndb.City` table is not indexed, we know that a full table scan (see Section 8.3.4, “Table Scans”) is used to execute this query. Running `EXPLAIN` can be used to verify this assumption.

First, we run `EXPLAIN` while condition pushdown is disabled:

```
mysql> EXPLAIN
-> SELECT COUNT(*)
-> FROM world_ndb.City
-> WHERE Name
-> BETWEEN 'A'
-> AND 'Z'
-> \G
***** 1. row *****
      id: 1
    select_type: SIMPLE
      table: City
        type: ALL
possible_keys: NULL
         key: NULL
        key_len: NULL
         ref: NULL
         rows: 4079
       Extra: Using where
1 row in set (0.00 sec)
```

As expected, the `possible_keys` column reveals that there are no indexes at all that could be used. This means that a table scan is used instead. The `rows` column indicates that the scan must iterate through 4079 rows (that is, through every row in the `City` table). The value `Using where` appearing in the `Extra` column indicates that a condition in the `WHERE` clause is to be evaluated while scanning the rows.

When condition pushdown is enabled, the plan looks only slightly different from the previous one:

```
mysql> SET engine_condition_pushdown = ON;
Query OK, 0 rows affected (0.00 sec)

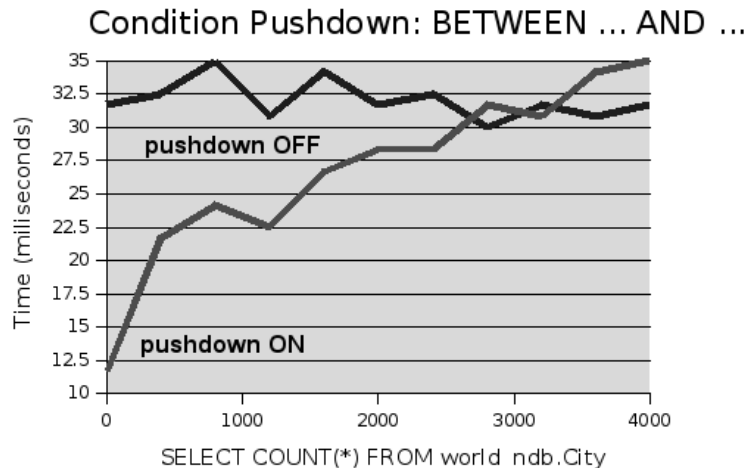
mysql> EXPLAIN
-> SELECT COUNT(*)
-> FROM world_ndb.City
```

### 8.3.5. Scans with Engine Condition Pushdown

```
-> WHERE Name
-> BETWEEN 'A'
-> AND 'Z'
-> \G
***** 1. row *****
      id: 1
    select_type: SIMPLE
      table: City
      type: ALL
possible_keys: NULL
      key: NULL
     key_len: NULL
       ref: NULL
      rows: 4079
    Extra: Using where with pushed condition
1 row in set (0.00 sec)
```

The only difference is seen in the Extra column, where the value Using where with pushed condition clearly indicates that the condition is pushed down.

By using values other than 'A' and 'Z', and recording the response times with and without condition pushdown enabled, we are able to get an indication of the effect of pushing down the WHERE condition. The following graph shows some actual measurements:



In the preceding graph, the line marked "Pushdown OFF" represents a series of queries performed without condition pushdown. The time required to calculate the COUNT(\*) is reasonably independent of the number of rows specified by the BETWEEN ... AND ... condition. No matter how many rows were matched by the condition, about 32 milliseconds were required for executing each query. This result is consistent with the idea that the transfer of data from data node to SQL node is a limiting factor. For all queries in this series, the entire data set needs to be transferred from data nodes to SQL nodes, and if the transfer is the limiting factor this would explain why all queries take about the same amount of time.

## 8.4. SQL Nodes and MySQL Cluster Performance

---

The line marked "Pushdown ON" represents the same series of queries, this time performed using condition pushdown. The graph suggests a strong relationship between the number of rows and the time required to execute each query. The queries processing relatively small numbers of rows are noticeably faster than the same queries performed without condition pushdown. As the number of rows matched by the condition increases, the gain in performance decreases. The graph indicates that condition pushdown results in performance loss when the condition matches 3000 rows or more.

### Note

The measurements shown here have no bearing on performance by other machines and configurations. Our sole purpose here is to measure the difference between outcomes with condition pushdown enabled and disabled, in otherwise identical environments. The graphs might seem to suggest that the number of rows is the main determinant, but in reality, the size of the result transferred from the data nodes to the SQL nodes is a more realistic determinant of performance.

Engine condition pushdown is not a “silver bullet” that solves all MySQL Cluster performance problems. Condition pushdown works best for simple conditions that single out relatively few rows from many. For NDB, enabling condition pushdown may in many cases offers better performance than disabling it. However, it will rarely if ever offer better performance than an index. So, when encountering recurring performance problems, you should research the effect of condition pushdown in each specific case and consider carefully whether it is the right solution before trying to rely on it to the exclusion of other methods for increasing performance.

## 8.4. SQL Nodes and MySQL Cluster Performance

As discussed elsewhere in this Guide (see Section 2.1.3.3, “API Nodes”), SQL nodes are merely MySQL Servers that are part of a MySQL cluster. They receive requests in the form of SQL statements, and process these statements in order to construct and return an adequate response to the client.

Whereas previous sections of this chapter focused primarily on data nodes and on the processes handled by them, this section focuses on those aspects of statement processing in the SQL node that can affect overall performance.

### 8.4.1. The Query Cache

The MySQL query cache is a component of the MySQL server. It is a simple in-memory key/value store where the key is a hash of the statement text of an SQL `SELECT` statement and the value is the corresponding result set. The query cache can provide for a very fast lookup of a result set, and using it can result in an enormous increase in overall performance.

When a MySQL Server receives a `SELECT` statement, it first checks the query cache for a match. If there is one, it sends the result directly from the query cache. If no match is found, only then must the MySQL Server parse the statement (possibly rewriting the statement in order to optimize it), choose a

---

### 8.4.1. The Query Cache

---

query plan, retrieve data from one or more storage engines, and, finally, form and send the result set in response to the request. After the result set has been constructed, it can be stored in the query cache so it can be looked up very quickly the next time it is needed.

We do not intend to discuss the query cache in complete detail here. Rather, we highlight those issues which are most relevant with respect to using the query cache in combination with MySQL Cluster. General information about the query cache can be found in *The MySQL Query Cache*, in the MySQL Manual [<http://dev.mysql.com/doc/refman/5.1/en/query-cache.html>].

However, at a minimum, you should keep in mind these two points concerning the behavior of the query cache:

- Because the query cache uses a hash of the literal text of SQL statements, two otherwise identical queries with different capitalization are stored as two entries in the query cache. For example, this means that, from the point of view of the query cache, the queries `SELECT * FROM City` and `SELECT * From City` are different queries, and the result of the first, while identical to that of the second, will not be fetched by the second query even though it is already stored as the result of a previous call to the first query.
- The query cache is specific to each MySQL Server and is not shared via the Cluster at all — each SQL node's query cache is totally independent and a given MySQL Server in a MySQL Cluster has no knowledge of the query cache of any other MySQL Server, even if it is connected to the same cluster.

#### 8.4.1.1. How the Query Cache Can Reduce Network Hops

MySQL Server performance can benefit in general from the query cache in because less time and resources have to be spent on processing SQL statements, retrieving data and constructing result sets.

For MySQL Cluster, there is the additional benefit of reducing the number of times that data must be transmitted over the network between SQL nodes and data nodes. Once a result set can be retrieved from the query cache, such network hops can be avoided altogether, leading to improved response time. This may also lead to an overall increase in throughput because bandwidth that would have been required to send the data from the data nodes to the SQL node is conserved and can be used for other purposes.

#### 8.4.1.2. The Query Cache is Local: Consequences

The query cache is a component of an individual MySQL Server, and in a MySQL Cluster, each SQL node has its own query cache. This means that if an SQL statement is processed by a particular SQL node, the corresponding result set can be retrieved only from the query cache of that SQL node. Other cluster SQL nodes have no knowledge of the contents of that SQL node's query cache.

Depending upon how clients connect to the SQL nodes, the query cache might not be used as efficiently as possible. If each client establishes a connection to a single SQL node, all queries from that application are processed by the same SQL node, putting all of its queries in the Query Cache. This can

## 8.4.2. Batched Operations

---

be inefficient for the following reasons:

- The queries may overflow the cache. The query cache is not used efficiently, because entries are flushed from the query cache before they can be retrieved.
- A client that could benefit from the contents of the query cache of one particular SQL node might connect to a different SQL node. The query cache of the first SQL node cannot be used, and the second client runs the same sequence of statements against the other SQL node, filling up that SQL node's query cache instead.

The client application could try to increase query cache utilization by always selecting the same SQL node for a given class of queries. This would require some application logic for selecting the preferred SQL node for a particular query, as well as some failover mechanism to select a different SQL node in case the preferred SQL node is not available. This would also require the application to open different connections for different queries. Although this scenario will add some overhead, that may be outweighed by the benefit of maximizing query cache utilization.

### 8.4.1.3. Purging of the Query Cache

Normally, queries are purged from the query cache immediately whenever any of their underlying tables are modified. This mechanism is also in effect for cached queries that refer to NDB tables. When a statement has the potential to change the data in a given NDB table, the SQL node that executes the statement purges its query cache of those statements referring to that table.

However, data in a NDB table can be modified by any of the cluster's SQL nodes. For this reason, changing the data in an NDB table requires that *all* SQL nodes to purge their query caches of any queries referring to that table. A dedicated thread runs in the background to periodically check whether any of the tables have changed and purges the query cache if necessary.

The MySQL Server defines a `ndb-cache-check-time` configuration option that can be used to control the frequency of checking and purging the query cache. The `ndb-cache-check-time` configuration option is global, and sets the interval (as a number of milliseconds) between checks of the cache. By default, this value is zero, meaning that each change on an NDB table results in checking the cache immediately.

The `ndb-cache-check-time` configuration option can be configured in the `my.cnf` configuration file or passed as a command line argument to `mysqld`. The option can also be changed at runtime using a `SET` statement, in which case it should be referred to as `ndb_cache_check_time`.

Continuously checking whether NDB tables have changed results in some overhead, and increasing the `ndb-cache-check-time` configuration option to a value greater than zero can increase performance. However, doing so can result in “stale” data being returned from the query cache. Careful consideration needs to be made regarding requirements for consistency between MySQL Servers before changing `ndb-cache-check-time` to some value other than the default.

## 8.4.2. Batched Operations



---

## 8.4.2. Batched Operations

---

It has already been noted that the NDB storage engine has the task of translating storage engine API calls into NDB API calls. Although the end user has no direct control over this process, it is to some extent possible to influence the type and number of NDB API calls that are generated in order to achieve a given task.

In some cases, a single SQL statement can be used to manipulate multiple rows. Although this is eventually translated into single row operations at the NDB API level, it is possible to send several single row operations to the data nodes in a single batch. Batching of operations is good for performance, because it saves a number of network round trips.

### 8.4.2.1. Batched Key Lookups

The NDB storage engine can batch multiple unique hash key entries in a single message. For example, suppose there is a need to retrieve three rows by primary key. Consider the following query:

```
SELECT *
FROM   t1
WHERE  pk IN (1, 2, 3);
```

The NDB engine sends the values 1, 2, 3 to the data nodes as a single message. When the message arrives at the data nodes, key lookups are performed for each individual key value.

### 8.4.2.2. Batched Inserts

MySQL supports a special syntax for inserting multiple rows into a table as a single SQL statement. Consider this series of insert statements:

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

mysql> INSERT INTO t_ndb VALUES (2);
Query OK, 1 row affected (0.00 sec)

mysql> INSERT INTO t_ndb VALUES (3);
Query OK, 1 row affected (0.00 sec)
```

In MySQL, this can be more efficiently written using this variation of the INSERT statement:

```
mysql> INSERT INTO t_ndb VALUES (1),(2),(3);
Query OK, 3 rows affected (0.01 sec)
Records: 3  Duplicates: 0  Warnings: 0
```

In most cases, this type of statement improves performance. There are two general reasons that explain why this type of statement performs better than the equivalent series of single-row INSERT statements:

1. For the client, sending a single statement saves overhead as compared to sending multiple state-

### 8.4.3. Statistics Management

---

ments to the MySQL server.

2. The server saves overhead with regard to parsing the statement and creating an execution plan for it.

For NDB there is an additional reason: The NDB storage engine is specifically optimized to use batched operations for this type of insert statement. The MySQL Cluster storage engine sends all of the rows to be inserted in a single batch to the data nodes, where they are individually inserted.

If autocommit is disabled, the performance gain is even larger, because committing a transaction for each statement (as well as initiating a new one for the following statement) adds to the overhead imposed by executing the statement. Again, this is not specific to NDB; in many cases, disabling autocommit, and committing explicitly only when the application requires it, can lead to a gain in performance, provided the transaction does not grow too large.

### 8.4.3. Statistics Management

The `ndb_index_stat_cache_entries`, `ndb_index_stat_enable`, and `ndb_index_stat_update_freq` server variables control a cache of index statistics kept by the MySQL Server for use by the optimizer in formulating query plans. The `ndb_index_stat_cache_entries` parameter controls the granularity of the statistics; a value larger than the default (32) means that more starting and ending keys are stored.

By default, the `ndb_index_stat_enable` option enables the use of MySQL Cluster index statistics in query optimization. Disabling the use of index statistics can quite likely make the optimizer produce different query plans; however, in some rare situations doing so may improve performance. Every `ndb_index_stat_update_freq`<sup>th</sup> query (the default is every 20 queries), the index statistics are fetched from the data nodes instead of the statistics cache. Increasing the value for this option can improve performance because of fewer network trips, but may in some situations result in poorer query plans if set too high.

The `ndb_use_exact_count` parameter controls whether the MySQL Server should always ask MySQL Cluster for an exact row count of rows during certain stages of query planning. To avoid multiple trips to the data nodes during query planning, it is possible to specify whether exact or approximate counts of rows are used. When `ndb_use_exact_count` is disabled (by default, it is enabled) the optimizer always assumes a row count of 100 for any table, instead of asking the data nodes for an exact count. In some situations, disabling this parameter can lead to better performance but can also result in poorer query plans for some queries.

### 8.4.4. Other `mysql`d Performance Issues

The `ndb_autoincrement_prefetch_sz` option for the MySQL Server determines how many auto increment values are (by default) prefetched by the MySQL Server. In the case of large batch inserts, the MySQL Server automatically requests a bigger range. This affects the performance of concurrent (and sequential) inserts on tables having `AUTO_INCREMENT` columns. If

---

## 8.5. Data Nodes and MySQL Cluster Performance

---

`ndb_autoincrement_prefetch_sz` is set to a relatively small value such as 1, then the MySQL Server needs to make a round trip to the data nodes for an `AUTO_INCREMENT` value for each row inserted. If this option is set to a higher value, such as 32 (the default), each MySQL Server prefetches a range of 32 `AUTO_INCREMENT` numbers for the table. Although this decreases the likelihood of consecutive `AUTO_INCREMENT` numbers across multiple MySQL Servers, it can greatly improve insert performance.

By disabling the `ndb_use_transactions` option, you can disable MySQL Cluster transaction support. While this is not recommended, it can in some rare situations potentially increase performance.

If `ndb_force_send` is disabled for a given SQL node (the default is for it be enabled), then that MySQL Server waits for other threads to complete before sending data to other nodes. For some workloads, disabling `ndb_force_send` can improve throughput, although (generally speaking) each individual transaction may take longer.

## 8.5. Data Nodes and MySQL Cluster Performance

We have already discussed data access methods used by data nodes for retrieving data and how these can have an impact to performance. In this section, we examine a number of other issues relating to data nodes and MySQL Cluster performance. Most of these are issues involving specific MySQL Cluster configuration parameters.

In general, for data nodes, if any of `MaxNoOfConcurrentOperations`, `MaxNoOfConcurrentTransactions` and related parameters are set too low, insufficient resource errors may occur, having a negative impact on performance. It is important to monitor such errors and to make sure that there are sufficient resources available for your transaction loads.

### 8.5.1. The `MaxNoOfConcurrentTransactions` Parameter and Data Node Performance

The data node parameter `MaxNoOfConcurrentTransactions` limits the number of simultaneous transactions that can be active at any one time. If this resource is exhausted, applications must wait and then retry any transactions that failed due to this. Each transaction requires some memory to be allocated for it by the data nodes, so a very high setting is not possible. A balance needs to be struck between parallelism, memory usage, and response time, which last can be affected by the available processing capacity of the data nodes.

### 8.5.2. The `MaxNoOfConcurrentOperations` Parameter and Data Node Performance

The data node parameter `MaxNoOfConcurrentOperations` limits the number of rows that can be part of an ongoing transaction. Like `MaxNoOfConcurrentTransactions`, this resource uses memory and so a balance needs to be struck. Logging and monitoring resource errors from applications

### 8.5.3. The TransactionDeadlockDetectionTimeout Parameter

---

can be used to provide warnings to the end user or system administrator about possible problems.

### 8.5.3. The TransactionDeadlockDetectionTimeout Parameter and Data Node Performance

The data node parameter `TransactionDeadlockDetectionTimeout` controls how long the transaction coordinator may wait for another node to complete its part of a transaction. A failure for the other data node to respond could be due to any of several reasons — perhaps the node has died, or the operation has entered a lock queue, or the node performing the operation is overloaded. If the operation times out, the transaction coordinator rolls back the transaction. If set to a large value, deadlocked transactions may persist for a long period of time, potentially with locks held that other transactions could use. However, excessively small values may cause transactions to be needlessly rolled back.

### 8.5.4. The TransactionInactiveTimeout Parameter and Data Node Performance

Like `TransactionDeadlockDetectionTimeout`, the data node parameter `TransactionInactiveTimeout` is a timeout value for transactions. This timeout value is by default zero, which allows for an infinite timeout. If this parameter is set to a nonzero value, a transaction that is inactive for that number of milliseconds is rolled back. For real-time applications, the correct setting of this parameter is an important aspect of maintaining reliable response times.

### 8.5.5. The LockPagesInMainMemory Parameter and Data Node Performance

If there is not enough physical memory on a given data node host computers, that host's operating system may have to swap memory to disk. Having any part of the data node processes swapped out by the operating system can have a negative impact not only on performance, but lead to missed heartbeats and the node being declared “dead” as well. If the `LockPagesInMainMemory` parameter is specified, the data node process tells the operating system that it cannot swap out any data node process to disk. This is especially important if any other processes are running on the data node hosts.

### 8.5.6. The NoOfFragmentLogFiles Parameter and Data Node Performance

As discussed elsewhere, the configuration of the database logs (`NoOfFragmentLogFiles` parameter and parameters dealing with undo files) for the data nodes has a direct relation to the transaction load that the cluster supports. If there is insufficient log space, application errors are generated and transactions are unable to complete, negatively affecting performance.

### 8.5.7. The NoOfReplicas Parameter and Data Node Performance

The more replicas (`NoOfReplicas`) there are in the cluster, the more nodes need to be involved during the committing of transactions, which means that more time is required for transactions to be com-

mitted. However, for read operations, an greater number of nodes can result in improved performance due to load balancing. In addition, for queries affecting more than one row, having more fragments means that more nodes are involved in a given transaction; hence, more time is required to commit the transaction.

## 8.6. Performance and Tuning — Exercises

These are sample questions and answers for the *Performance and Tuning* Chapter.

Question 1:

In a MySQL Cluster, which type of node delegates physical storage and retrieval to data nodes?

- a. The SQL node
- b. The data node itself
- c. The management node
- d. None of the above

Question 2:

Partitioning tables using the NDB storage engine could increase response time because [*check all that apply*]:

- a. Network traffic is required to transfer the data from one data node to another
- b. Delegation itself requires some time
- c. Operations on partitions cannot be evaluated in parallel
- d. The data node handling an NDB API call might need to delegate the call to another data node that actually contains the row

Question 3:

Partitioning tables using the NDB storage engine can *decrease* response time because [*check all that apply*]:

- a. All operations are conducted singly, one after another
- b. Some operations can be performed in parallel
- c. A given data node contains only a portion of the data, so there is less data to process
- d. Partitioning tables cannot ever decrease response time

Question 4:

Increasing the physical distance between nodes often has a negative impact on a cluster, such as increasing latency. However, what might be some valid reasons for maximizing the distance between

nodes? *[check all that apply]*

- a. Transactions are logged more frequently in the management node(s)
- b. Transactions are logged more frequently in the data node(s)
- c. Physically separating redundant components minimizes the chance that a disaster such as a fire, a plane crash or an earthquake results in failure of all redundant components at once
- d. Distant, isolated nodes have lower memory requirements

Question 5:

What is the difference between bandwidth and throughput? How does each affect the performance of an NDB cluster? *[open ended]*

Question 6:

A table using the NDB storage engine does not necessarily need a primary key.

- a. True
- b. False

Question 7:

The primary key of a table using the NDB storage engine is implemented with a BTREE index.

- a. True
- b. False

Question 8:

Which of the following WHERE statements can be evaluated using a hashed primary key?

- a. WHERE city = "Los Angeles"
- b. WHERE founding\_date BETWEEN "1776-07-04" AND "2000-01-01"
- c. WHERE population > 40000000
- d. WHERE pastry LIKE "apple%"

Question 9:

An example of condition pushdown is:

- a. An SQL node evaluating a condition on every single row of a table
- b. Each data node evaluating a condition on every single row of a table
- c. A management node evaluating a condition on every single row of a table

## 8.6. Performance and Tuning — Exercises

---

- d. Failure to evaluate a condition, which is then noted in the event log

Question 10:

In MySQL Cluster, a batched insert is less efficient than using individual insert statements:

- a. True
- b. False

Question 11:

A hash index is not usable for range queries:

- a. True
- b. False

Question 12:

What effect does increasing the number of replicas (`NoOfReplicas`) have on transactions involving NDB tables? [*check all that apply*]

- a. There is lower network latency within the cluster
- b. Total network traffic is reduced dramatically
- c. Transactions take less time to commit
- d. Transactions take longer to commit

*Answers to Exercises*

Answer 1:

*The SQL node (a).*

Answer 2:

*Network traffic is required to transfer the data from one data node to another (a), Delegation itself requires some time (b), and The data node that handles an NDB API call might need to delegate the call to another data node that actually contains the row (d) are all correct.*

Answer c (Operations on partitions cannot be evaluated in parallel) is incorrect because one of the strengths of MySQL cluster is the potential parallel processing that can occur with certain queries, under certain setups.

Answer 3:

*Some operations can be performed in parallel (b) and A given data node contains only a portion of the*

## 8.6. Performance and Tuning — Exercises

---

*data, so there is less data to process (c)* are both correct.

Answer **a** (All operations are conducted singly, one after another) is incorrect because some operations can indeed be performed in parallel under the right circumstances.

Answer 4:

*Physically separating redundant components minimizes the chance that a disaster such as a fire, a plane crash or an earthquake results in failure of all redundant components at once (c)* is correct.

Answers **a** (Transactions are logged more frequently in the management node) and **b** (Transactions are logged more frequently in the management node) are incorrect because neither the management nodes or data nodes "log transactions". Answer **d** (Distant, isolated nodes have lower memory requirements) is also incorrect, because neither distance nor latency has any effect on the amount of memory required by an individual node.

Answer 5:

Bandwidth represents the quantity of data that can be transferred across a network during a given unit of time; throughput is the amount of work that can be performed by the system during a given amount of time.

Bandwidth affects MySQL Cluster performance due to the fact that data processing and retrieval is dependent on communications between cluster data nodes and SQL nodes.

Cluster throughput is dependent upon the processing power and memory available on each cluster data node.

Answer 6:

*False*. NDB tables always have a primary key. It is true that an NDB table need not have a primary key explicitly defined; however, in cases where no primary is explicitly created, the NDB storage engine creates one.

Answer 7:

*False*. NDB table primary keys are hash indexes.

Answer 8:

*WHERE city = "Los Angeles" (a)* is the only correct answer. Answers **b** (*WHERE founding\_date BETWEEN "1776-07-04" AND "2000-01-01"*) and **c** (*WHERE population gt; 40000000*) are incorrect because hashed keys cannot evaluate ranged queries — they can be used only for equality comparison. Answer **d** (*WHERE pastry LIKE "apple%"*) is incorrect because, once again, the primary key lookup method can be used for equality comparison only, and is incompatible with the *LIKE* keyword, which allows for a range of possible results.



## 8.6. Performance and Tuning — Exercises

---

Answer 9:

*Each data node evaluating a condition on every single row of a table (b)* is correct.

Answer **a** (An SQL node evaluating a condition on every single row of a table) is incorrect because this describes a classic table scan, and represents one of the worst kinds of performance hits in database administration. Answer **c** (A management node evaluating a condition on every single row of a table) is incorrect because management nodes are not involved in evaluating data. Answer **d** (Failure to evaluate a condition, which is then noted in the event log) is incorrect because while such events occasionally occur due to invalid syntax, this is not within the purview of condition pushdown.

Answer 10:

*False.* In fact, MySQL Cluster is optimized for batch insert operations.

Answer 11:

*True.* Hash indexes execute a hashing function on column values, placing them in “buckets” — that is, in specific locations in memory directly associated with the hash values. When searching for a specific value, that value is likewise hashed, and then checked to see if such a value exists at the corresponding memory location. Therefore, range queries cannot be used with hash indexes, because a range in theory could be an infinite number of values that would need to be checked.

Answer 12:

*Transactions take longer to commit (d)* is correct.

Answer **a** (There is lower network latency within the cluster) is incorrect because network latency is not a function of the number of replicas within the cluster. Answer **b** (Total network traffic is reduced dramatically) is also incorrect because having more replicas (and therefore, more servers) is actually likely to *increase* network traffic. Answer **c** (Transactions take less time to commit) is incorrect because more replicas often means more nodes that need to be involved during a COMMIT operation.