MySQL Cluster Tutorial

O'Reilly MySQL Conference & Expo 2010, Apr. 12 2010.

Andrew Hutchings (Oracle)
Andrew Morgan (Oracle)
Geert Vanderkelen (Oracle)

This document is a handout for the MySQL Cluster Tutorial. Please also check the slides which were shown turing the tutorial.

http://en.oreilly.com/mysql2010/public/schedule/detail/12438

Introduction

MySQL Cluster is a tool which could help make your data Highly Available. This tutorial will help you run a MySQL Cluster, show how to manage it and discuss various topics such as performance, backups and schema considerations.

Before going any further we need to setup the Virtual Machine (VM) running under VirtualBox. You can install MySQL Cluster yourself following instructions found in section **Installation and Configuration**, but we strongly suggest to stick to the filesystem layout and configuration files (found on the DVD).

Virtual Machine

You have been given a DVD which contains VirtualBox and a Virtual Machine. The VM will boot Ubuntu (Linux Distribution) with all software pre-installed and configured.

To get you going, do the following:

1. Mount or open the DVD
2. Install (or upgrade) VirtualBox. The latest version is included on the DVD in the folder `software/`.
3. Copy the `clustervm/` and `config/` folder to your hard drive. Location does not matter, but make sure you copy the complete folder and all its contents.
4. Start VirtualBox: from the File-menu choose 'Import Appliance'
5. The 'Appliance Wizard' will show. Locate the `Ubuntu 9.10.ovf` file you copied from the DVD and follow the steps. No options should be changed.

Topics

Installation and Configuration
What to download, how to install and configure MySQL Cluster.

Running Nodes and Your First Table
Starting MySQL Cluster and creating your first NDB table.

Administer MySQL Cluster
Managing and monitoring MySQL Cluster.

MySQL Cluster Manager
We'll introduce a new tool to manage MySQL Cluster.

Fault Tolerance
Explains what happens when some node fails.

Online Backup
How to backup your data and meta data.

NDB Info
Getting information out of MySQL Cluster made easy.

NDBAPI
Coding for Cluster using NDB API, and 'No SQL'.

MySQL Cluster Connector for Java
Introduction and talking to Cluster directly using Java.

Schema Considerations
A few tips when planning to develop for or convert to MySQL Cluster

Scaling and Performance
How you can scale and get more performance.

Geographical Replication
Making your MySQL Cluster itself highly available.

Security
Discusses how you can secure your MySQL Cluster

Speakers

Andrew Hutchings
MySQL Support Engineer

> "Andrew Hutchings is a MySQL Support Engineer for Oracle Corporation specialising in MySQL Cluster and C/C++ APIs. He is based in the United Kingdom and has worked for MySQL/Sun since 2008. Before joining Sun he was the Technical Architect, Senior Developer and DBA for a major UK magazine publisher. In his spare time Andrew develops various bug fixes and features for MySQL and MySQL Cluster."

Andrew Morgan
MySQL Product Manager

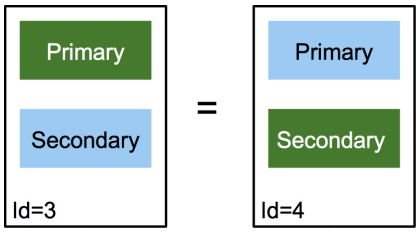
> "Andrew is the MySQL Product Manager responsible for High Availability Solutions – in particular MySQL Cluster and replication. He is based in United Kingdom and has worked for MySQL/Sun/Oracle since February 2009.Before joining MySQL he was responsible for delivering High Availability telecoms applications which is where he became exposed to MySQL Cluster – replacing proprietary and other 3rd party databases. His primary roles in MySQL are working with engineers to make sure that MySQL Cluster & replication evolve to meet the needs of their users as well as spreading the word on the what people can get from these technologies."

Geert Vanderkelen
MySQL Support Engineer

> "Geert is a member of the MySQL Support Team at Sun Microsystems. He is based in Germany and has worked for MySQL AB since April, 2005.
> Before joining MySQL he worked as developer, DBA and SysAdmin for various companies in Belgium and Germany.
> Today Geert specializes in MySQL Cluster and works together with colleagues around the world to ensure continued support for both customers and community. He's also the maintainer of Sun's MySQL Connector/Python."

MySQL Cluster in short

- MySQL Cluster is a tool to help you **keep your data available**
- It consists of various processes called Nodes which should be setup in a shared-nothing environment.
- **Data Nodes**: where data, table and index information is stored in-memory and optionally on disk (for non indexed data)
  During the tutorial you will notice `NoOfReplicas=2` in the cluster configuration. This means that data is stored 2 times with cluster. If you have 2 data nodes, the data is partitioned in two parts. Each partitions has a replica on another data node. This way, in a 2 data node setup, each data node has a copy of all the data.



- **SQL Nodes** or MySQL servers: using the NDBCluster engine, you can use SQL to update data and retrieve information from the Data Nodes through your network (using TCP/IP)
- **Management Nodes**: used for management and monitoring
- **API Nodes**: application written using NDB API (SQL Nodes are an example)
- The NDBCluster engine is **transaction-safe, ACID**. Durability through checkpointing.

Transactions

Lets have a look how data is stored inside MySQL Cluster. We'll use a table which was created using the ENGINE table option set to NDBCluster:

```
CREATE TABLE attendees (
id INT UNSIGNED NOT NULL AUTO_INCREMENT,
email VARCHAR(200) NOT NULL,
PRIMARY KEY (id)
) ENGINE=NDBCluster
```

The Request

Your web, Java or Python application talks to a MySQL server and asks to store a new attendee. It does this by first starting a transaction:

```
START TRANSACTION
```

The Transaction

The MySQL server, or SQL Node, is opening a transaction on behave of the application and gets a Transaction Coordinator assigned. This coordinator is actually a Data Node which is chosen in a Round-Robin way.

The Commit

The application is done with sending the data and issues the `COMMIT` command. The MySQL server passes on the request to the transaction coordinator and waits for the result.

The Storage

The Data Node acting as coordinator starts the 2-Phase Protocol. It first the other nodes whether they are ready to get the changes. When it gets an OK from all (and from itself), it says all to actually do the commit.

The Acknowledgment

The tuple got stored safely on the data nodes and the transaction coordinator tells the SQL Node that the transaction succeeded.

The MySQL server, still talking to the application which issued the initial request, gives an OK.

What if it fails?

When the transaction coordinator, the data node responsible for the transaction, is noticing a problem, it will do a roll back.

The application will receive an error and is responsible for taking action. If it is a temporary problem, it can, for example, to try again.

Installation and Configuration

In this section we discuss how to install MySQL Cluster and configure it.

Release Model and Versioning

MySQL Cluster is developed and maintained at a different pace than the normal MySQL server. Having both separate allows to release more often and independently.
With the new release model came a new versioning scheme. Here is an overview of MySQL Cluster versions with their full version:

MySQL Cluster	.. full version
6.3.33	mysql-5.1.44 ndb-6.3.33
7.0.14	mysql-5.1.44 ndb-7.0.14
7.1.2a	mysql-5.1.41 ndb-7.1.2beta

Download

Just like the regular MySQL releases, MySQL Cluster can be downloaded from the Developer Zone on mysql.com: http://dev.mysql.com/downloads/cluster/.
Here is an example of the tar-distribution for 32-bit Linux platforms, the one used in this tutorial:

```
mysql-cluster-gpl-7.1.2a-beta-linux-i686-glibc23.tar.gz
```

Installation

You can install MySQL Cluster the same way you install a regular MySQL server. You are free to place it where ever you like, but the more 'default' on UNIX-like systems, is in `/usr/local`.
Here are instructions for installing MySQL Cluster on a Linux 32-bit platform.

```
shell> cd /usr/local
shell> tar xzf mysql-cluster-gpl-7.0.13-linux-i686-glibc23.tar.gz
shell> ln -s mysql-cluster-gpl-7.0.13-linux-i686-glibc23 mysql
```

> If you want more details, see the section 'Installing MySQL from Generic Binaries on Unix/Linux' in the MySQL Manual:
> http://dev.mysql.com/doc/refman/5.1/en/installing-binary.html

Lets look at some files which are useful for daily usage of MySQL Cluster:

/usr/local/mysql/	
bin/mysql	MySQL command-line client tool
bin/mysqld	MySQL server Use the mysqld_safe to start the MySQL server. To see all the options available use: `shell> mysqld --help --verbose | less`
bin/mysqldump	Schema and data backup For MySQL Cluster you'll need this to backup schema, stored procedures, etc.. For data you want to use ndb_restore.
bin/mysqld_safe	MySQL server startup script Use this to start the MySQL server.
bin/ndbd	Data Node daemon
bin/ndbmtd	Data Node daemon, multi-threaded
bin/ndb_mgm	MySQL Cluster management client tool
bin/ndb_mgmd	Management Node daemon
bin/ndb_restore	Restore MySQL Cluster backup

You can use the --help option for the above application to show their options.

Locations

After the installation of MySQL Cluster we need to define where the data is going to be stored. For this tutorial we'll use some subdirectories in /opt/mysqlcluster/.

/opt/mysqlcluster	
/ndb	Data directory for MySQL Cluster
/ndb_Slave	Data Directory for Slave MySQL Cluster
/mysql_A	Data directory for first MySQL server (mysqld)
/mysql_B	Data directory for second MySQL server (mysqld)
my_A.cnf	Option files for first MySQL server
my_B.cnf	Option file for second MySQL server
config.ini	Configuration file for MySQL Cluster
/mysql_Slave	Data directory for Slave MySQL Cluster

my_Slave.cnf	Configuration file Slave MySQL server

Configuration

There are two configuration files needed: one for MySQL Cluster (config.ini) and the options file for the regular MySQL server (my.cnf). The first default location for both is /etc but it recommended to put them into a mysql subdirectory, which is also read by default: /etc/mysql/.
For this tutorial we'll be putting the configuration files under /opt/mysqlcluster.

First MySQL Server

File /opt/mysqlcluster/my_A.cnf:

```
[mysqld]
datadir = /opt/mysqlcluster/mysql_A
socket = /tmp/mysql_A
port = 3306
log_bin = master_A_binary
server_id = 1
ndbcluster
```

Second MySQL Server

File /opt/mysqlcluster/my_B.cnf:

```
[mysqld]
datadir = /opt/mysqlcluster/mysql_B
socket = /tmp/mysql_B
port = 3306
log_bin = master_B_binary
server_id = 2
ndbcluster
```

Slave MySQL Server

File /opt/mysqlcluster/my_B.cnf:

```
[mysqld]
datadir=/opt/mysqlcluster/mysql_Slave
port=3308
socket=/tmp/mysql_Slave
log_bin = master_Slave_binlog
server_id = 90
binlog_format = MIXED
ndbcluster
ndb_connectstring = localhost:1187
```


Configuration of MySQL Cluster

File /opt/mysqlcluster/config.ini:

```
[ndbd default]
NoOfReplicas = 2
DataMemory = 80M
IndexMemory = 10M
DataDir = /opt/mysqlcluster/ndb

[ndb_mgmd default]
DataDir = /opt/mysqlcluster/ndb

[tcp default]

[ndb_mgmd]

Id = 1
HostName = localhost

[ndbd]
Id = 3
HostName = localhost

[ndbd]
Id = 4
HostName = localhost

[mysqld]
Id = 20
HostName = localhost

[mysqld]
Id = 21
HostName = localhost

[mysqld]
Id = 22
HostName = localhost

[mysqld]
Id = 23
HostName = localhost
[api]
[api]
[api]
[api]
```

Configuration of MySQL Cluster

File /opt/mysqlcluster/config_Slave.ini:

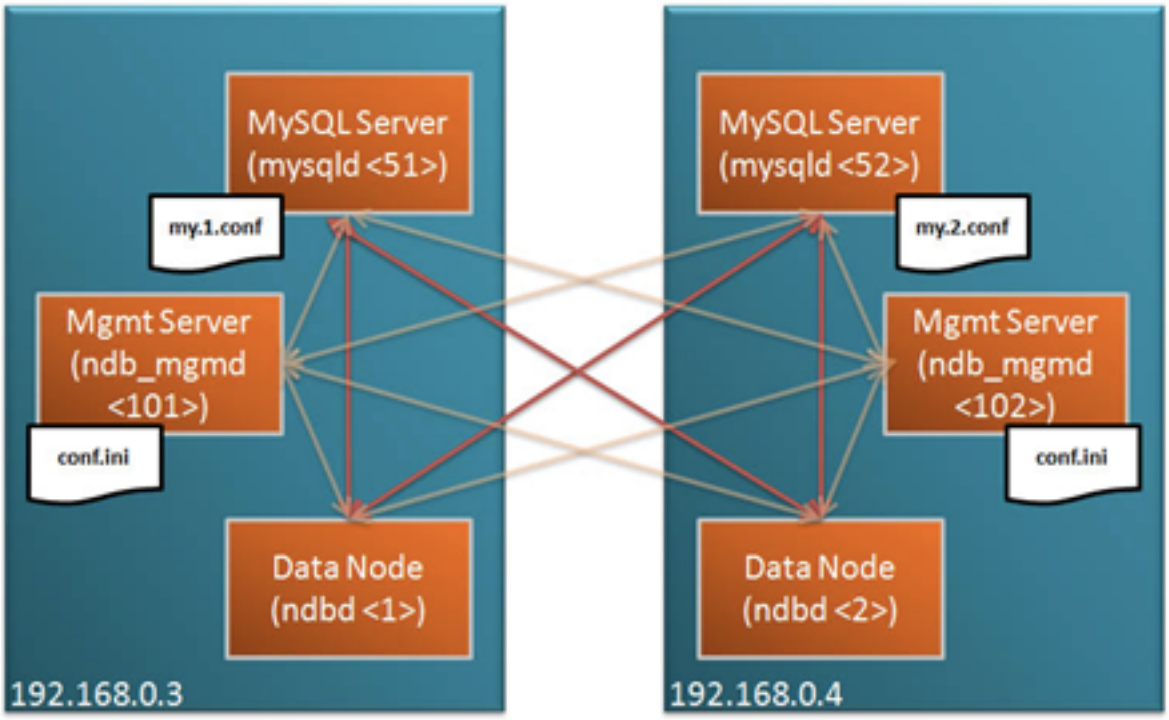
```
[ndbd default]
NoOfReplicas = 1
DataMemory = 80M
IndexMemory = 10M
DataDir = /opt/mysqlcluster/ndb_Slave

[ndb_mgmd default]
PortNumber = 1187
DataDir = /opt/mysqlcluster/ndb_Slave

[tcp default]

[ndb_mgmd]
Id = 1
HostName = localhost

[ndbd]
Id = 3
HostName = localhost

[mysqld]
Id = 20
HostName = localhost

[api]
[api]
[api]
[api]
```


Starting & Using MySQL Cluster

Starting MySQL Cluster

The nodes (processes) should be started in this order:

1. Management Node(s)
   `ndb_mgmd`
2. Data Nodes
   `ndbd` or `ndbmtd`
3. MySQL Server(s)
   `mysqld`



Start Management Node(s)

```
192.168.0.3: ndb_mgmd --initial -f conf/config.ini –configdir=/home/billy/mysql/
7_0_6/conf
192.168.0.4: ndb_mgmd --initial -f conf/config.ini –configdir=/home/billy/mysql/
7_0_6/conf
192.168.0.3: ndb_mgm
```



```
ndb_mgm> show
Connected to Management Server at: localhost:1186
Cluster Configuration
---------------------
[ndbd(NDB)] 2 node(s)
id=1 (not connected, accepting connect from 192.168.0.3)
id=2 (not connected, accepting connect from 192.168.0.4)

[ndb_mgmd(MGM)] 2 node(s)
id=101 @192.168.0.3 (mysql-5.1.34 ndb-7.0.6)
id=102 (not connected, accepting connect from 192.168.0.4)

[mysqld(API)] 2 node(s)
id=51 (not connected, accepting connect from 192.168.0.3)
id=52 (not connected, accepting connect from 192.168.0.4)
```

Start Data Nodes

```
192.168.0.3: ndbd -c localhost:1186
192.168.0.4: ndbd  -c localhost:1186
192.168.0.3: ndb_mgm

ndb_mgm> show
Connected to Management Server at: 192.168.0.4:1186
Cluster Configuration
---------------------
[ndbd(NDB)] 2 node(s)
id=1 @127.0.0.1 (mysql-5.1.34 ndb-7.0.6, Nodegroup: 0, Master)
id=2 @192.168.0.4 (mysql-5.1.34 ndb-7.0.6, Nodegroup: 0)

[ndb_mgmd(MGM)] 2 node(s)
id=101 (not connected, accepting connect from 192.168.0.3)
id=102 @192.168.0.4  (mysql-5.1.34 ndb-7.0.6)

[mysqld(API)] 2 node(s)
id=51 (not connected, accepting connect from 192.168.0.3)
id=52 (not connected, accepting connect from 192.168.0.4)
```

Note: only use the `--intial` option when starting the data nodes for the very first time as it erases any existing data in the database!

Wait until all data nodes have started before the next step.

Start MySQL Server(s)

```
192.168.0.3: mysqld --defaults-file=conf/my.1.conf &
192.168.0.4: mysqld --defaults-file=conf/my.2.conf &
192.168.0.3: ndb_mgm
```



```
ndb_mgm> show
Connected to Management Server at: localhost:1186
Cluster Configuration
---------------------
[ndbd(NDB)] 2 node(s)
id=1 @127.0.0.1 (mysql-5.1.34 ndb-7.0.6, Nodegroup: 0, Master)
id=2 @192.168.0.4 (mysql-5.1.34 ndb-7.0.6, Nodegroup: 0)

[ndb_mgmd(MGM)]   2 node(s)
id=101 @127.0.0.1 (mysql-5.1.34 ndb-7.0.6)
id=102 @127.0.0.1 (mysql-5.1.34 ndb-7.0.6)

[mysqld(API)]     2 node(s)
id=51 @192.168.0.3  (mysql-5.1.34 ndb-7.0.6)
id=52 @192.168.0.4  (mysql-5.1.34 ndb-7.0.6)
```

Create a Cluster table

```
192.168.0.3: mysql -h localhost -P 3306

mysql> use test
Database changed

mysql> create table assets (name varchar(30) not null primary key,
    -> value int) engine=ndb;
Query OK, 0 rows affected (0.99 sec)

mysql> insert into assets values ('Car','1900');
Query OK, 1 row affected (0.03 sec)

mysql> select * from assets;
+------+-------+
| name | value |
+------+-------+
| Car  |  1900 |
+------+-------+
1 row in set (0.00 sec)
```

Now check that the same data can be accessed from the second MySQL Server:

```
192.168.0.3: mysql -h 192.168.0.4 -P 3306

mysql> use test
Database changed
mysql> select * from assets;
+------+-------+
| name | value |
+------+-------+
| Car  |  1900 |
+------+-------+
1 row in set (0.00 sec)
```


Exercise:

Using the configuration files already created in previous exercise (all nodes on a single host):

1. Start the management nodes
2. Check the status
3. Start the data nodes and wait for them to start
4. Start the MySQL Servers
5. Connect to a MySQL Server, create a Cluster table (engine=ndb)
6. Add a tuple
7. Check the data is visible when using the other MySQL Server.

Administer MySQL Cluster

MySQL Cluster has a command line tool to talk directly to the management server to obtain information and execute basic functions. It also has a verbose logging system which can help diagnose many problems with the cluster.

NDB_MGM

A key tool for running basic administration tasks is `ndb_mgm`. This is a command line tool to send basic commands to the cluster. It communicates to the management nodes using an unsecured port with no authentication (take note this should be done behind a firewall or private network).
To invoke this tool simply run:

```
shell> ndb_mgm
```

If you are not running it locally to the management node then you will need to give it a connect string:

```
shell> ndb_mgm --ndb-connectstring=192.168.1.10
```

You can also execute a one-time command with this tool rather than getting a command prompt using:

```
shell> ndb_mgm -e 'SHOW'
```

Commands in ndb_mgm almost all follow a common syntax. Many commands require node IDs, some do not, so the syntax is typically as follows:

```
ndb_mgm> [node_id] {command} [parameters]
```

The `node_id` can be any single node ID or the keyword `'ALL'` to send it to all data nodes.

Common Commands

SHOW

This command gives a quick overview of the status of all the nodes, whether or not they are connected and up/down.
For example:

```
ndb_mgm> show
Cluster Configuration
---------------------
[ndbd(NDB)] 2 node(s)
id=2  @127.0.0.1  (mysql-5.1.41 ndb-7.1.2, Nodegroup: 0, Master)
id=3  @127.0.0.1  (mysql-5.1.41 ndb-7.1.2, Nodegroup: 0)

[ndb_mgmd(MGM)]   1 node(s)
id=1  @127.0.0.1  (mysql-5.1.41 ndb-7.1.2)

[mysqld(API)]     2 node(s)
id=4  @127.0.0.1  (mysql-5.1.41 ndb-7.1.2)
id=5 (not connected, accepting connect from any host)
```

If the data nodes are currently starting you will see this reflected in this status as follows:

```
[ndbd(NDB)] 2 node(s)
id=2  @127.0.0.1  (mysql-5.1.41 ndb-7.1.2, starting, Nodegroup: 0, Master)
id=3  @127.0.0.1  (mysql-5.1.41 ndb-7.1.2, starting, Nodegroup: 0)
```


It is worth noting that whilst all the data nodes are in this phase and the cluster is not officially started API nodes will not be allowed to connect, even if they are started, so these will show 'not connected'.

STATUS

This command gives a more detailed status report for an individual node, as such it requires a node ID (or the 'ALL' keyword).
For example, during a cluster start:

```
ndb_mgm> 2 status
Node 2: starting (Last completed phase 3) (mysql-5.1.41 ndb-7.1.2)
```

With the 'ALL' keyword on a started cluster:

```
ndb_mgm> all status
Node 2: started (mysql-5.1.41 ndb-7.1.2)
Node 3: started (mysql-5.1.41 ndb-7.1.2)
```

Note that the 'ALL' keyword only means all data nodes, not all nodes. This command can, however, be used with any node type.
For example, lets ask for the management node's status:

```
ndb_mgm> 1 status
Node 1: connected (Version 7.1.2)
```

STOP

The STOP command is used to shutdown an individual data or management node cleanly or stop all the data nodes. For example:

```
ndb_mgm> 2 stop
Node 2: Node shutdown initiated
Node 2: Node shutdown completed.
Node 2 has shutdown.
This command will not let you do an action which will terminate other nodes too
(unless 'ALL' is used).  So with node 2 down:

ndb_mgm> 3 stop
Node 3: Node shutdown aborted
Shutdown failed.
*  2002: Stop failed
*   Node shutdown would cause system crash: Permanent error: Application error
```

So with 'ALL STOP' with all data nodes running again:

```
ndb_mgm> all stop
Executing STOP on all nodes.
Node 2: Cluster shutdown initiated
Node 3: Cluster shutdown initiated
Node 2: Node shutdown completed.
Node 3: Node shutdown completed.
NDB Cluster has shutdown.
```

RESTART

The RESTART command is very much like STOP but it starts the node again. This command does not always work as expected, so in general we recommend using with caution. When it does work you should expect output similar to this:


```
ndb_mgm> 2 restart
Node 2: Node shutdown initiated
Node 2: Node shutdown completed, restarting, no start.
Node 2 is being restarted

ndb_mgm> Node 2: Started (version 7.1.2)
```

START BACKUP

This command initiates a cluster backup. This is covered in more detail in the 'Online Backup' section of this document.
It can basically be executed using:

```
ndb_mgm> start backup
Waiting for completed, this may take several minutes
Node 3: Backup 1 started from node 1
Node 3: Backup 1 started from node 1 completed
 StartGCP: 474 StopGCP: 477
 #Records: 2053 #LogRecords: 0
 Data: 50312 bytes Log: 0 bytes
```

There are additional parameters for this command too. These can change the backup number (normally sequential), whether or not the client should wait for it to complete and whether the snapshot should be consistent to the point of the start of the backup, or end of the backup. So:

```
START BACKUP [backup_id] [wait_option] [snapshot_option]
wait_option:
WAIT {STARTED | COMPLETED} | NOWAIT
snapshot_option:
SNAPSHOTSTART | SNAPSHOTEND
```

The default is:

```
START BACKUP {next_backup_id} WAIT COMPLETED SNAPSHOTEND
```

REPORT

There are currently two reports you can get using this command, `BACKUPSTATUS` and `MEMORYUSAGE`. Although shorter versions of these names can be used. For example:

```
ndb_mgm> all report memory
Node 2: Data usage is 0%(22 32K pages of total 2560)
Node 2: Index usage is 0%(16 8K pages of total 2336)
Node 3: Data usage is 0%(22 32K pages of total 2560)
Node 3: Index usage is 0%(16 8K pages of total 2336)
```

DUMP

This is probably the most dangerous command and should not be used on a production cluster. The developers added dump codes in various cluster components to get information out or trigger an action. If there are any results for the code these are sent to the management node's log file. Using the wrong number for a dump code can easily take down an entire cluster. One example is the dump code '`1000`' which does the same as `REPORT MEMORYUSAGE`, but it is recorded in the management node log instead:

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


In the management node log:

```
[MgmtSrvr] INFO     -- Node 2: Data usage is 0%(22 32K pages of total 2560)
[MgmtSrvr] INFO     -- Node 2: Index usage is 0%(16 8K pages of total 2336)
[MgmtSrvr] INFO     -- Node 2: Resource 0 min: 2602 max: 4935 curr: 1694
[MgmtSrvr] INFO     -- Node 2: Resource 3 min: 3144 max: 3144 curr: 606
[MgmtSrvr] INFO     -- Node 2: Resource 5 min: 1152 max: 1152 curr: 1088
[MgmtSrvr] INFO     -- Node 3: Data usage is 0%(22 32K pages of total 2560)
[MgmtSrvr] INFO     -- Node 3: Index usage is 0%(16 8K pages of total 2336)
[MgmtSrvr] INFO     -- Node 3: Resource 0 min: 2602 max: 4935 curr: 1694
[MgmtSrvr] INFO     -- Node 3: Resource 3 min: 3144 max: 3144 curr: 606
[MgmtSrvr] INFO     -- Node 3: Resource 5 min: 1152 max: 1152 curr: 1088
```

Cluster Log

The management node keeps a log of the state of all the nodes, such as when they connect and disconnect, when checkpointing happens and so on. It can be found on the management node's file system with the filename `ndb_{node_id}_cluster.log`, where `node_id` is the management node's ID.

The verbosity level for logging can be set using the `CLUSTERLOG` `ndb_mgm` command as follows:

```
node_id CLUSTERLOG category=threshold
```

Threshold is an integer between 0 and 15, the following table is a list of categories and their default thresholds:

Category	Default threshold
STARTUP	7
SHUTDOWN	7
STATISTICS	7
CHECKPOINT	7
NODERESTART	7
CONNECTION	7
ERROR	15
INFO	7

Data Node Logs

The data nodes all have their own logs which are mainly aimed at diagnosing faults with the nodes. There are typically 3 types of log files. The first a `stdout` log which stores basic information on things such as the node watchdog and memory allocation. Then there is the error log which stores information about node failures. Finally there is the trace files which stores signal data about each failure in the error log.

Out Log

The `stdout` log is named `ndb_{node_id}_out.log`. This normally will only show information about the node starting and stopping but can also contain information related

to load problems such as timing issues or watchdog problems. It also contains basic information about a node failure if it occurs.

Error Log

This by default records the last 25 node failures which occurred on the node. The 25 figure can be changed using `MaxNoOfSavedMessages` in config.ini. Each entry in the error log points to a corresponding trace log (or set of trace logs for `ndbmtd`). This file is named `ndb_{node_id}_error.log`.

Trace Log

The trace log contains low level details about the execution history for the node prior to the crash including a signal log. This is very complicated to read and it is recommended this information is passed to MySQL Support. These files are named `ndb_{node_id}_trace.log.{trace_id}[_t{thread_id}]`.

Data Node File System

Data nodes have a their own 'filesystem' which is a collection of files, most of which are stored in a `ndb_{node_id}_fs` directory in the data node's `DataDir` by default. Most of this is checkpoint data, which is essential so that a node failure does not mean data loss.

Local CheckPoint

Every so often a complete dump of the DataMemory is made to disk, this is called a Local CheckPoint (or LCP).
The calculation used to figure out how often this happens is a little complex. The setting is `TimeBetweenLocalCheckpoints` but it is not time based at all. It is calculated by $x = (4*2^y)$. Where x is the number of bytes of write operations since the last LCP and y is `TimeBetweenLocalCheckpoints`. In most cases the default is fine (20, which works out to 4MB).

The speed at which this is written to disk is controlled by `DiskCheckpointSpeed`. This is 10MB/sec by default and is not recommended to be increased unless you have really fast disks as this can limit time-sensitive operations such as Global CheckPoints.

Global CheckPoint

Creating an LCP is fine, but data is still being written in mean time. This data needs storing to disk too to minimize the risk of data loss if all nodes go down. This comes in the form of a redo log. Every commit is also recorded to the redo log which is committed to disk synchronously across all nodes using a Global CheckPoint (GCP). By default a GCP happens every 2 seconds using the `TimeBetweenGlobalCheckpoints` setting. If the node cannot write all the redo data to disk since the last GCP in 2 seconds then the node will intentionally fail to prevent the risk of large data loss.

Redo data is stored in the fragment log files, controlled using `NoOfFragmentLogFiles` and `FragmentLogFileSize`. There are 4 sets of fragment log files so the former setting should be multiplied by 4 when calculating how much data this will use. The fragment log files must be large enough to store all the redo between two LCPs or again, the node will fail to stop potential data loss.

Undo Log

If you use disk data there is the addition of the undo log which needs creating before the disk data files. This stores all the disk changes since the last LCP so that they can be rolled back before the redo is applied.

Online Backup

By default these is stored in the BACKUP subdirectory of the DataDir. They are covered in more detail in the 'Online Backup' section of this document.

MySQL Cluster Manager

MySQL Cluster Manager – Architecture and Use

MySQL Cluster Manager Architecture

MySQL Cluster Manager is implemented as a set of agents – one running on each physical host that will contain MySQL Cluster nodes (processes) to be managed. The administrator connects the regular mysql client to any one of these agents and then the agents each communicate with each other. By default, port 1862 is used to connect to the agents. An example consisting of 4 host machines together with the mysql client running on a laptop is shown in Figure 1.

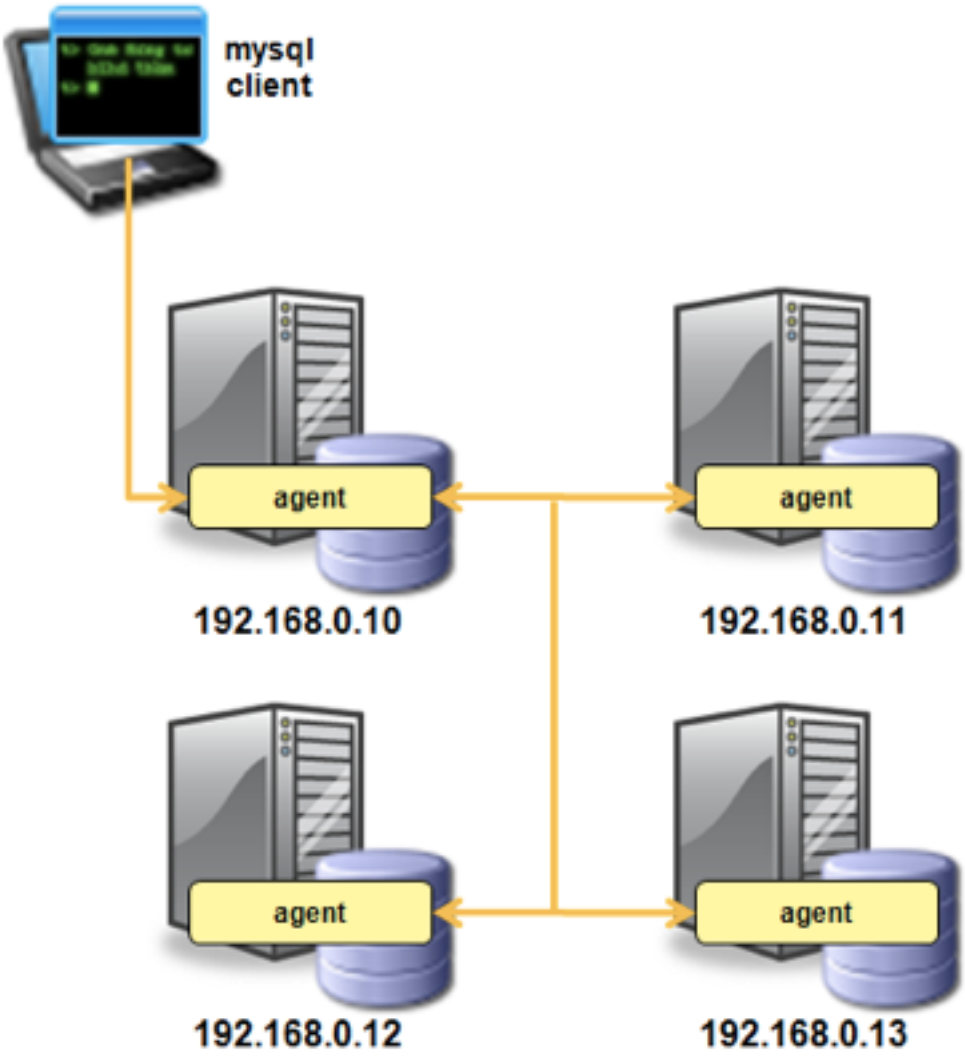


Figure 1: ***MySQL Cluster Manager Agents running on each host***

The agents work together to perform operations across the nodes making up the Cluster, each Cluster dealing with its local nodes (those processes running on the same host). For example, if a data node, management node or MySQL Server on host X fail, then the process will be restarted by the agent running on host X.

As another example, in the event of an upgrade operation the agents co-operate to ensure that each node is upgraded in the correct sequence, but it is always the local agent that will actually perform the upgrade to a specific node.

To upgrade MySQL Cluster Manager, all agents can be stopped and new ones started (using the new software version) with no impact to the continuing operation of the MySQL Cluster database.

Changes from Previous Approaches to Managing MySQL Cluster

When using MySQL Cluster Manager to manage your MySQL Cluster deployment, the administrator no longer edits the configuration files (for example `config.ini` and `my.cnf`); instead, these files are created and maintained by the agents. In fact, if those files are manually edited, the changes will be overwritten by the configuration information which is held within the agents. Each agent stores all of the cluster configuration data, but it only creates the configuration files that are required for the nodes that are configured to run on that host. An example of this is shown in Figure 2 below.

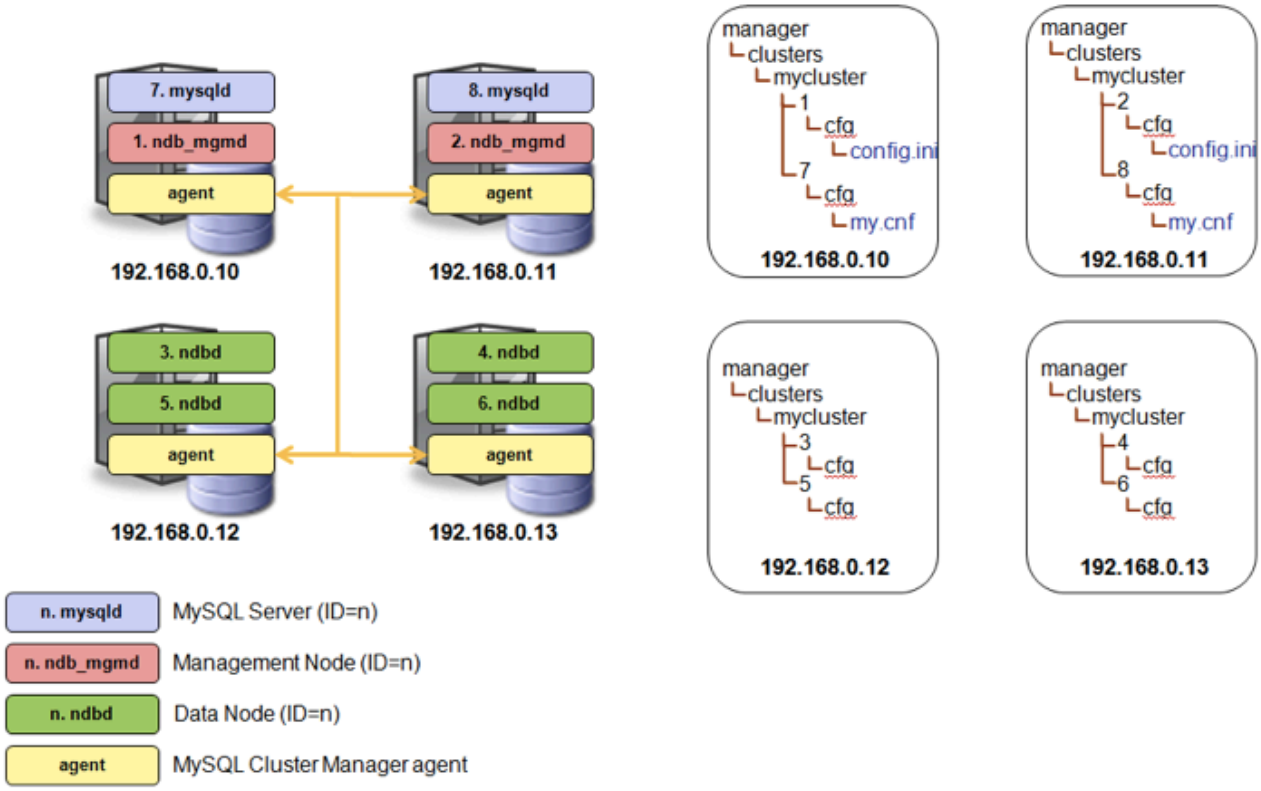


Figure 2: Configuration files created on each host

Similarly when using MySQL Cluster Manager, management actions must not be performed by the administrator using the `ndb_mgm` command (which directly connects to the management node meaning that the agents themselves would not have visibility of any operations performed with it).

The introduction of MySQL Cluster manager does not remove the need for management nodes; in particular they continue to perform a number of critical roles:

- When data nodes start up (or are restarted) they connect to the management node(s) to retrieve their configuration data (the management node in turn fetches that data from the configuration files created by the agents);
- When stopping or restarting a data node through MySQL Cluster Manager, the state change is actually performed by the management node;

- The management node(s) can continue to act as arbitrators (avoiding a split-brain scenario). For this reason, it is still important to run those processes on separate hosts from the data nodes;
- Some reporting information (for example, memory usage) is not yet available in the Cluster Manager and can still be performed using the `ndb_mgm` tool.

When using MySQL Cluster Manager, the 'angel' processes are no longer needed (or created) for the data nodes, as it becomes the responsibility of the agents to detect the failure of the data nodes and recreate them as required. Additionally, the agents extend this functionality to include the management nodes and MySQL Server nodes.

It should be noted that there is no angel process for the agents themselves and so for the highest levels of availability, the administrator may choose to use a process monitor to detect the failure of an agent and automatically restart it. When an agent is unavailable, the MySQL Cluster database continues to operate in a fault-tolerant manner but management changes cannot be made until it is restarted. When the agent process is restarted, it re-establishes communication with the other agents and has access to all current configuration data, ensuring that the management function is restored.

When using MySQL Cluster Manager, the 'angel' processes are no longer needed (or created) for the data nodes, as it becomes the responsibility of the agents to detect the failure of the data nodes and recreate them as required. Additionally, the agents extend this functionality to include the management nodes and MySQL Server nodes.

It should be noted that there is no angel process for the agents themselves and so for the highest levels of availability, the administrator may choose to use a process monitor to detect the failure of an agent and automatically restart it. When an agent is unavailable, the MySQL Cluster database continues to operate in a fault-tolerant manner but management changes cannot be made until it is restarted. When the agent process is restarted, it re-establishes communication with the other agents and has access to all current configuration data, ensuring that the management function is restored.

MySQL Cluster Manager Model & Terms

Before looking at how to install, configure and use MySQL Cluster Manager, it helps to understand a number of components and the relationships between them (as viewed by the MySQL Cluster Manager). Figure 3 illustrates a number of these entities and then there is a brief description of each.

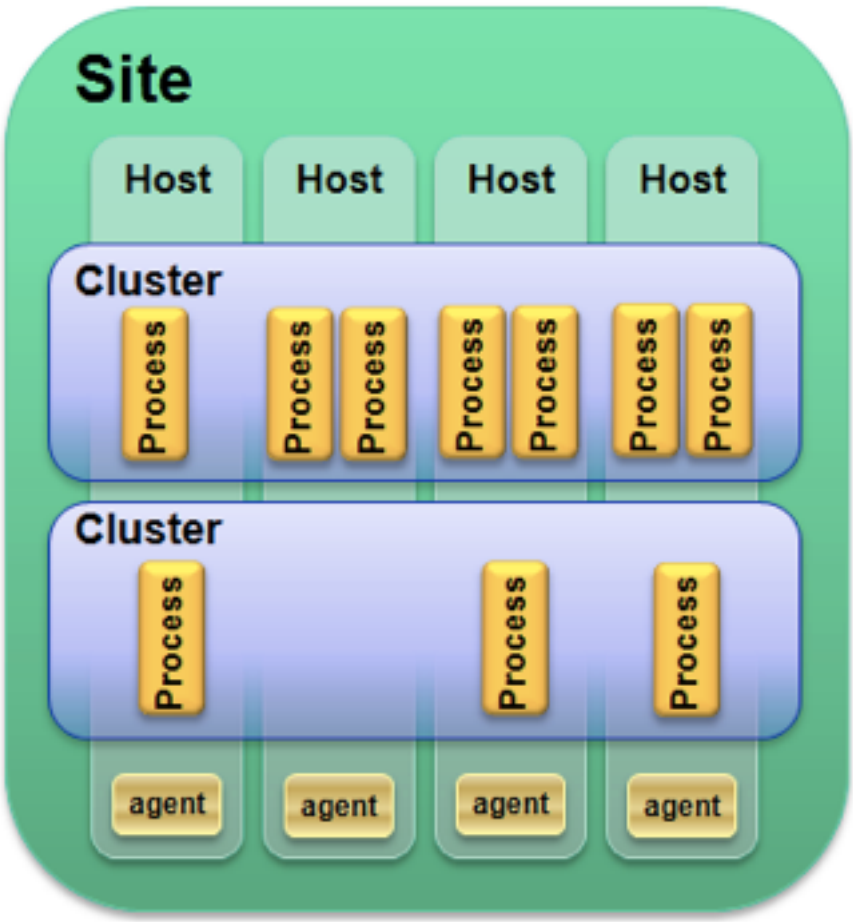


Figure 3: ***MySQL Cluster Manager model***

Site: the set of physical hosts which are used to run database cluster processes that will be managed by MySQL Cluster Manager. A site can include 1 or more clusters.

Cluster: represents a MySQL Cluster deployment. A Cluster contains 1 or more processes running on 1 or more hosts.

Host: a physical machine (i.e. a server), running the MySQL Cluster Manager agent.

Agent: the MySQL Cluster Manager process running on each host. Responsible for all management actions performed on the processes.

Process: an individual MySQL Cluster node; one of: `ndb_mgmd`, `ndbd`, `ndbmtd`, `mysqld` & `ndbapi`.

Package: a copy of a MySQL Cluster installation directory as downloaded from mysql.com, stored on each host.

Using MySQL Cluster Manager – a worked example

This section walks through the steps required to install, configure and run MySQL Cluster Manager. It goes on to demonstrate how it is used to create and manage a MySQL Cluster deployment.

Example configuration
Throughout the example, the MySQL Cluster configuration shown in Figure 4 is used.

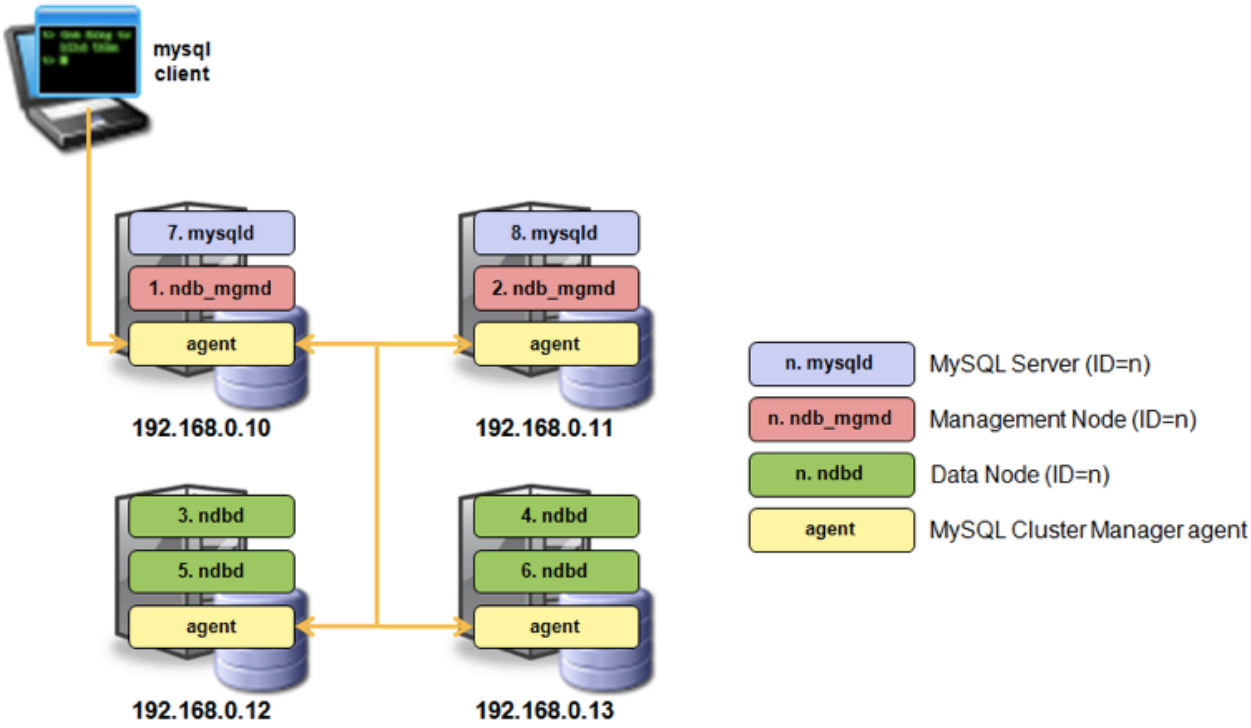


Figure 4: *MySQL Cluster configuration used in example*

Installing, configuring, running & accessing MySQL Cluster Manager
The agent must be installed and run on each host in the Cluster:

1. Expand the downloaded tar-ball into a known directory (in this example we use `/usr/local/mcm`)
2. Copy the `/usr/local/mcm/etc/mysql-cluster-manager.ini` to `/home/billy/mcm/` and edit it:

```
[mysql-proxy]
plugins=manager
log-file = mysql-manager-agent.log
log-level = message
manager-directory=/home/billy/mcm/
```

3. Launch the agent process:

```
$ /usr/local/mcm/bin/mysql-cluster-manager --defaults-file=/home/billy/mcm/
mysql-cluster-manager.ini &
```

4. Access any of the agents from any machine (that has the `mysql` client installed):

```
$ mysql -h 192.168.0.10 -P 1862 -u admin -p --prompt='mcm> '
Enter password: super
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 1
Server version: 1.0.1-agent-manager MySQL Cluster Manager
Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.
mcm>
```


After repeating steps 1 through 3 on each of the physical hosts that will contain MySQL Cluster nodes to be managed, the system looks like Figure 5 below.

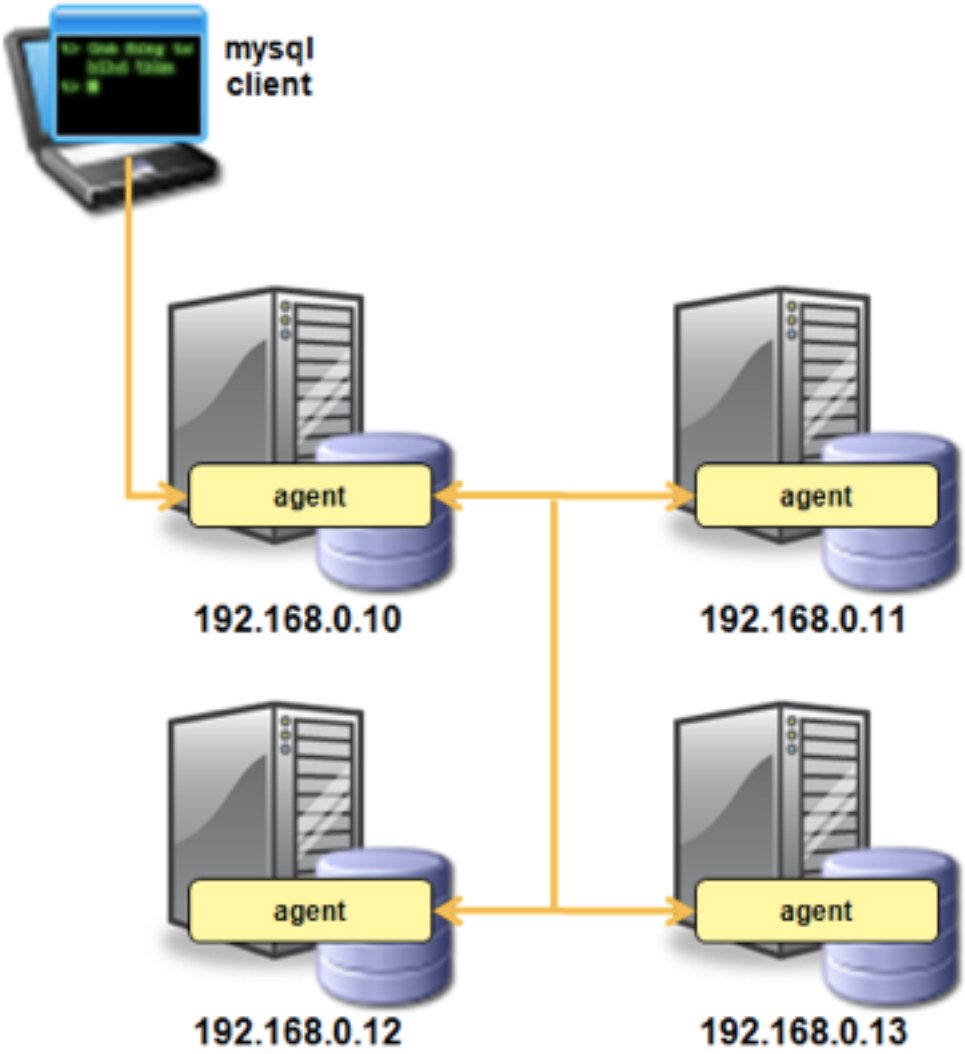


Figure 5: ***MySQL Cluster Manager Agents running on each host***

Creating & Starting a Cluster

Once the agents have been installed and started, the next step is to set up the entities that will make up the managed Cluster(s). As a prerequisite, the installation files for the version (s) of MySQL Cluster that are to be managed should be stored in a known location – in this example, `/usr/local/mysql_X_Y_Z`.

As in step 4 from section 1.3.2 connect to the agent process running on any of the physical hosts – from there the whole process of creating all entities needed for the Cluster (site, hosts, packages and Cluster) and then starting the Cluster can be performed.

1. Define the site – that is, the set of hosts that will run MySQL Cluster processes that need to be managed:

```
mcm> create site --hosts=192.168.0.10,192.168.0.11,192.168.0.12,192.168.0.13
mysite;
```

2. Define the package(s) that will be used for the Cluster; further packages can be added later (for example when upgrading to the latest MySQL Cluster release). In this

example, the MySQL Cluster installation directories are in the same location on every host – if that isn't the case then the `--hosts` option can be used to qualify which hosts the directory name applies to:

```
mcm> add package --basedir=/usr/local/mysql_6_3_27a 6.3.27a;
mcm> add package --basedir=/usr/local/mysql_7_0_9 7.0.9;
```

3. Create the Cluster, specifying the package (version of MySQL Cluster) and the set of nodes/processes and which host each should run on:

```
mcm> create cluster --package=6.3.26
--processhosts=ndb_mgmd@192.168.0.10,ndb_mgmd@192.168.0.11,
ndbd@192.168.0.12,ndbd@192.168.0.13,ndbd@192.168.0.12,
ndbd@192.168.0.13,mysqld@192.168.0.10,mysqld@192.168.0.11
mycluster;
```

4. The Cluster has now been defined – all that remains is to start the processes for the database to become available:

```
mcm> start cluster mycluster;
```

Note that the node-ids are allocated automatically, starting with 1 for the first in the list of processes provided, and then incrementing by 1 for each subsequent process.

The MySQL Cluster is now up and running as shown in Figure 7 above.

Check the status of the Cluster

Using MySQL Cluster Manager, you can check the status at a number of levels:

- The overall state of the Cluster
- The state of each of the hosts (specifically whether the agent on that host is running and accessible)
- The state of each of the nodes (processes) making up the Cluster

```
mcm> show status --cluster mycluster;
+-----------+-------------------+
| Cluster   | Status            |
+-----------+-------------------+
| mycluster | fully-operational |
+-----------+-------------------+

mcm> list hosts mysite;
+--------------+-----------+---------+
|     Host     | Status    | Version |
+--------------+-----------+---------+
| 192.168.0.10 | Available | 1.0.1   |
| 192.168.0.11 | Available | 1.0.1   |
| 192.168.0.12 | Available | 1.0.1   |
| 192.168.0.13 | Available | 1.0.1   |
+--------------+-----------+---------+
```



```
mcm> show status --process mycluster;
+------+----------+--------------+---------+-----------+
| Id   | Process  | Host         | Status  | Nodegroup |
+------+----------+--------------+---------+-----------+
| 1    | ndb_mgmd | 192.168.0.10 | running |           |
| 2    | ndb_mgmd | 192.168.0.11 | running |           |
| 3    | ndbd     | 192.168.0.12 | running | 0         |
| 4    | ndbd     | 192.168.0.13 | running | 0         |
| 5    | ndbd     | 192.168.0.12 | running | 1         |
| 6    | ndbd     | 192.168.0.13 | running | 1         |
| 7    | mysqld   | 192.168.0.10 | running |           |
| 8    | mysqld   | 192.168.0.11 | running |           |
+------+----------+--------------+---------+-----------+
```

Checking and setting MySQL Cluster parameters

When using MySQL Cluster Manager, the administrator reviews and changes all configuration parameters using the `get` and `set` commands rather than editing the configuration files directly. For both `get` and `set` you can control the scope of the attributes being read or updated. For example, you can specify that the attribute applies to all nodes, all nodes of a particular class (such as all data nodes) or to a specific node. Additionally, when reading attributes you can provide a specific attribute name or ask for all applicable attributes as well as indicating whether you wish to see the attributes still with their default values or just those that have been overwritten.

There follow some examples of getting and setting attributes.

1. Fetch all parameters that apply to all data nodes, including defaults

```
mcm> get -d :ndbd mycluster;
+----------------------+-------+----------+------+----------+------+---------+----------+
| Name                 | Value | Process1 | Id1  | Process2 | Id2  | Level   | Comment  |
+----------------------+-------+----------+------+----------+------+---------+----------+
| __ndbmt_lqh_threads  | NULL  | ndbd     | 3    |          |      | Default |          |
| __ndbmt_lqh_workers  | NULL  | ndbd     | 3    |          |      | Default |          |
| Arbitration          | NULL  | ndbd     | 3    |          |      | Default |          |
........               :       :          :      :          :      :         :          :
| __ndbmt_lqh_threads  | NULL  | ndbd     | 4    |          |      | Default |          |
| __ndbmt_lqh_workers  | NULL  | ndbd     | 4    |          |      | Default |          |
| Arbitration          | NULL  | ndbd     | 4    |          |      | Default |          |
| ArbitrationTimeout   | 3000  | ndbd     | 4    |          |      | Default |          |
........               :       :          :      :          :      :         :          :
| __ndbmt_lqh_threads  | NULL  | ndbd     | 5    |          |      | Default |          |
........               :       :          :      :          :      :         :          :
| __ndbmt_lqh_threads  | NULL  | ndbd     | 6    |          |      | Default |          |
........               :       :          :      :          :      :         :          :
+----------------------+-------+----------+------+----------+------+---------+----------+
```

2. Fetch the values of parameters (excluding defaults) for `mysqld` with `ID=7`:

```
mcm> get :mysqld:7 mycluster;
+------------+-------------------------------------------------+----------+------+-...
| Name       | Value                                           | Process1 | Id1  |
+------------+-------------------------------------------------+----------+------+-...
| datadir    | /usr/local/mcm/manager/clusters/mycluster/7/data | mysqld   | 7    | ...
| HostName   | ws1                                             | mysqld   | 7    | ...
| ndb-nodeid | 7                                               | mysqld   | 7    | ...
| ndbcluster |                                                 | mysqld   | 7    | ...
| NodeId     | 7                                               | mysqld   | 7    | ...
+------------+-------------------------------------------------+----------+------+-...
```


3. Fetch the port parameter to connect to `mysqld` with `ID=7`:

```
mcm> get -d port:mysqld:7 mycluster;
+------+-------+---------+------+---------+------+---------+---------+
| Name | Value | Process1 | Id1  | Process2 | Id2  | Level   | Comment |
+------+-------+---------+------+---------+------+---------+---------+
| port | 3306  | mysqld   | 7    |          |      | Default |         |
+------+-------+---------+------+---------+------+---------+---------+
```

4. Turn off privilege checking for all MySQL Servers and change the port for connecting to the mysqld with `ID = 8` to 3307. Allow data nodes to be automatically restarted after they fail:

```
mcm> set skip-grant-tables:mysqld=true,port:mysqld:8=3307,StopOnError:ndbd=false
mycluster;
```

   MySQL Cluster Manager automatically determines which nodes (processes) need to be restarted and in which order to make the change take effect but avoid loss of service. In this example, this results in each management, data and MySQL Server node being restarted sequentially as shown in Figure 6 below.

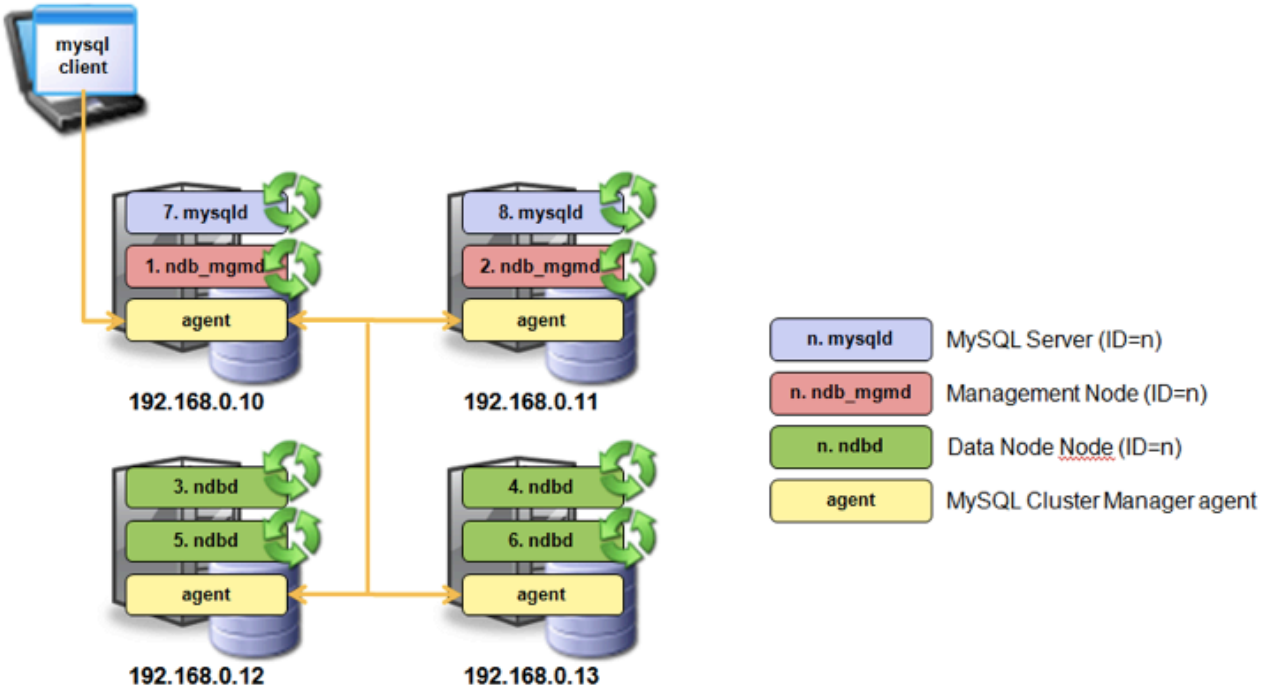


Figure 6: ***MySQL Cluster nodes automatically restarted after configuration change***

Upgrading MySQL Cluster

Upgrading MySQL Cluster when using MySQL Cluster Manager is extremely simple. The amount of administrator time needed and the opportunities for human error are massively reduced.

There are three prerequisites before performing the upgrade:

1. Ensure that the upgrade path that you are attempting to perform is supported; by referring to http://dev.mysql.com/doc/mysql-cluster-excerpt/5.1/en/mysql-cluster-upgrade-downgrade-compatibility.html.
   If not then you will need to perform an upgrade to an intermediate version of MySQL Cluster first;

2. Ensure that the version of MySQL Cluster Manager you are using supports both the original and new versions of MySQL Cluster; if not then you must first upgrade to an appropriate version of MySQL Cluster Manager as described in section 1.3.7.
3. Ensure that the install tar ball for the new version of MySQL Cluster has been copied to each host and the associated packages defined.

The upgrade itself is performed using a single command which will upgrade each node in sequence such that database service is not interrupted to connected applications and clients:

```
mcm> upgrade cluster --package=7.0.9 mycluster;
+------------------------------+
| Command result               |
+------------------------------+
| Cluster upgraded successfully |
+------------------------------+
1 row in set (1 min 41.54 sec)
```

Upgrading MySQL Cluster Manager

Upgrading MySQL Cluster Manager is very straightforward. Simply stop the agents on each host and then start up the new version of the agents. The same `mysql-cluster-manager.ini` file should be used for the new version; in particular the `manager-directory` attribute should be the same as used with the previous version, so that the configuration information is not lost.

MySQL Cluster continues to run when the MySQL Cluster Manager agents are stopped, and so there is no service impact during the upgrade. If a MySQL Cluster node fails during the upgrade, the database continues to function but the node is not recovered until the new agents have started – for this reason, the administrator may choose to perform the upgrade when demands on the database are at their lowest.

Shutting down the Cluster

An entire Cluster can be shutdown using a single command:

```
mcm> stop cluster mycluster;
+------------------------------+
| Command result               |
+------------------------------+
| Cluster stopped successfully |
+------------------------------+
1 row in set (22.96 sec)
```

Unlike the `shutdown` command from `ndb_mgm` that would be used if not using MySQL Cluster Manager, `stop cluster` will also safely stop all MySQL Server processes.

HA Provided by MySQL Cluster Manager

If MySQL Cluster detects that a MySQL Cluster process has failed, then it will automatically restart it (for data nodes this only happens if `StopOnError` attribute the has been set to `FALSE`). For data nodes, the agents take over this role from the existing MySQL Cluster data node angel processes, but for Management and MySQL Server nodes this is new functionality.

It is the responsibility of the administrator or application to restart the MySQL Cluster Manager agents in the event that they fail; for example by creating a script in `/etc/init.d`

Introducing MySQL Cluster Manager to an existing Cluster

In MySQL Cluster 1.0, there is no automated method to bring an existing MySQL Cluster deployment under the management of MySQL Cluster Manager; this capability will be added in a future release. In the mean time, a documented procedure is provided.

Single host exercise

1. The MySQL Cluster Manager software has been installed in /usr/local/mcm
2. Create a directory called 'mcm' in your home directory
3. Copy `/usr/local/mcm/etc/mysql-cluster-manager.ini` to your local mcm folder and edit it to set manager-directory to a folder called 'manager' within your mcm folder (no need to create the folder)
4. Launch the agent:
   $ /usr/local/mcm/bin/mysql-cluster-manager --defaults-file=/home/billy/mcm/mysql-cluster-manager.ini&
5. Connect to the agent:

```
$ mysql -h 127.0.0.1 -P 1862 -u admin -p --prompt='mcm> '
```

6. Define the site:

```
mcm> create site --hosts=127.0.0.1 mysite;
```

7. Define the site packages:

```
mcm> add package --basedir=/usr/local/mysql_7_X_Y my_package;
```

8. Define the Cluster:

```
mcm> create cluster --package=my_package
--processhosts=ndb_mgmd@127.0.0.1,ndbd@127.0.0.1,ndbd@127.0.0.1,mysqld@127.0.0.1
mycluster;
```

9. Start the Cluster:

```
mcm> start cluster mycluster;
```

10. Stop the Cluster:

```
mcm> stop cluster mycluster;
```


Fault tolerance

MySQL Cluster is designed to be very fault tolerant. Although there are issues you should be aware of. This section will cover how cluster is fault tolerant and how your application should work with this.

MySQL Server

When using MySQL in general temporary errors can happen, that is to say errors that may not occur if the transaction is retried. In MySQL Cluster this can happen more often than other storage engines for a variety of reasons, such as timeouts, node failures or redo log problems. With MySQL Cluster when this happens the transaction will be implicitly rolled-back and the error will be returned to the client. It is up to the client application to retry the transaction.

Sometimes the error message returned will not give enough details to find the true cause of the failure, so it is a good idea to run 'SHOW WARNINGS' after an error to see other messages from cluster. For example, if there is a transaction exclusive locking some rows this could happen:

```
mysql> select * from t1;
ERROR 1205 (HY000): Lock wait timeout exceeded; try restarting transaction
```

This is kind of useful, but MySQL Cluster has 5 different things that can cause this error. So:

```
mysql> show warnings\G
*************************** 1. row ***************************
  Level: Error
   Code: 1297
Message: Got temporary error 274 'Time-out in NDB, probably caused by deadlock'
from NDB
*************************** 2. row ***************************
  Level: Error
   Code: 1297
Message: Got temporary error 274 'Time-out in NDB, probably caused by deadlock'
from NDB
*************************** 3. row ***************************
  Level: Error
   Code: 1205
Message: Lock wait timeout exceeded; try restarting transaction
*************************** 4. row ***************************
  Level: Error
   Code: 1622
Message: Storage engine NDB does not support rollback for this statement.
Transaction rolled back and must be restarted
4 rows in set (0.00 sec)
```

So, the error here is NDB error code 274, it is now possible for the MySQL cluster team to find out the cause of this (274 is triggered by TransactionDeadLockDetectionTimeOut exceeded during a scan operation).

Heartbeats

The data nodes use a system of heartbeats in a ring to make sure that a node's neighbour is still up and running. The heartbeat interval is set using the HeartbeatIntervalDBDB setting which is 1500ms by default. A cluster node will be declared dead if it is not heard from

within 4 heartbeat intervals. This can happen if the data node is overloaded and too busy to send heartbeats on-time or if there are network problems.

If any heartbeats are missed it will be logged in the cluster log on the management node. A node's neighbours are decided when a node starts and this information is logged in the node's `stdout` log.

Online Backup

MySQL Cluster allows you to take online backups from your production database without interrupting any transaction. There are two parts: backing up the meta data (schema, stored procedures, etc..) and backing up the data itself.

Tools

To take an online backup, we'll use the following client tools:

- `mysqldump`: for backing up the meta data (table definition, stored procedures, etc..)
- `ndb_mgm`: the management client tool in which we'll issue the START BACKUP command
- `mysql`: to restore the meta data
- `ndb_restore`: reading data from the backup, and writing it back

Backing up the data, online

First, lets look how to make a backup for the data stored in MySQL Cluster using the `ndb_mgm` management tool. Important: it will only backup tables created with the NDBCluster storage engine.

```
ndb_mgm> START BACKUP
Waiting for completed, this may take several minutes
Node 3: Backup 2 started from node 1
Node 3: Backup 2 started from node 1 completed
 StartGCP: 284 StopGCP: 287
 #Records: 2059 #LogRecords: 0
 Data: 51016 bytes Log: 0 bytes
```

The `START BACKUP` asks the Data Nodes to dump the data for which they are responsible for to disk. It will also dump the meta data: table definition and index information.

When the command is done, a subdirectory `BACKUP-<backup-id>/` will be created in the file system subdirectory `BACKUP/` of each data node.

For example, the `/opt/mysqlcluster/ndb/BACKUP/BACKUP-2/` will contain the following files when `START BACKUP` finished:

```
BACKUP-2-0.3.Data
BACKUP-2-0.4.Data
BACKUP-2.3.ctl
BACKUP-2.3.log
BACKUP-2.4.ctl
BACKUP-2.4.log
```

The *.Data files contain the actual records. *.ctl files contain the meta information. The *.log contains the Transaction Log, which are the transaction committed while the backup was being made.

Important: backups made by data nodes need to be moved to a secure place.

Backing up meta data

Although the backup done using the management tool is including the meta data, it is advised to dump your table structures using `mysqldump`.

- Other tables not using the NDBCluster engine can only be backed up using `mysqldump`.

- Stored routines and other none table metadata is not stored in Cluster and consequently not backed up the online backup tool.
- Going from one version of MySQL Cluster to another might bring incompatible table formats.

For example:

```
shell> mysqldump --no-data --routines --all-tablespaces --all-databases >
meta_backup.sql
```

Restoring using ndb_restore

Backups made using ndb_mgm's START BACKUP command can be restored using ndb_restore. For this to work, you'll need to configure your MySQL Cluster so there is always a free [API] slot, which can connect from a dedicated machine which will do the restore.

Note that MySQL Cluster needs to be empty before restoring data. There is a possibility using ndb_restore to only recover a selection of tables. This could be used for restoring a table which was dropped by mistake.

Since backups are moved away from the data nodes ones they are made, `ndb_restore` will not necessary run from the data nodes. For simplicity, this tutorial assumes they were kept in place.

Restoring meta data

Meta data can be restored by both using `ndb_restore` and the `mysql` client tool. However, it is advised to restore the backups done with the `mysqldump` tool.

For example using an SQL dump of the meta data:

```
shell> mysql -uroot < meta_backup.sql
```

However, it's also important to know how to do it using `ndb_restore`. Suppose you have the following backup with ID 2 and need to restore Node 3 and Node 4, which were started empty.

```
BACKUP-2-0.3.Data
BACKUP-2-0.4.Data
BACKUP-2.3.ctl
BACKUP-2.3.log
BACKUP-2.4.ctl
BACKUP-2.4.log
```

When you use ndb_restore, you only need restore the meta once. Every backup of each node contains this information, so it doesn't matter which one you take.
Lets restore the meta data using the backup of Node 3:

```
shell> ndb_restore -m -b 2 -n 3 /path/to/BACKUP-2/

-m = restore meta
-b = backup ID
-n = node ID
```

The tables should now already be available.

Restoring the data

Similar to restoring meta data, we can restore the records. This time you'll need to it for each data node, or you will end up with half the data.

```
shell> ndb_restore -r -b 2 -n 3 /path/to/BACKUP-2/
shell> ndb_restore -r -b 2 -n 4 /path/to/BACKUP-2/
```

```
-m = restore meta
-r = restore records
-b = backup ID
-n = node ID
```

ndb_restore can do more

There are quite a few options for `ndb_restore`. For example `--print-meta` and `--print-data` allow you to get the table and records information out of a backup without actually restoring it.

You can also selective restore using `--include-*` and `--exclude-*` options.

Check the full list using `ndb_restore --help`.

NDB Info

ndbinfo Data Node Statistics

ndbinfo is a read-only database presenting information on what is happening within the cluster – in particular, within the data nodes. This database contains a number of views, each providing data about MySQL Cluster node status, resource usage and operations.

Because this information is presented through regular SQL views, there is no special syntax required to access the data either in its raw form ("`SELECT * FROM ndbinfo.counters`") or manipulating it by filtering on rows and columns or combining (joining) it with data held in other tables. This also makes it simple to integrate this information into existing monitoring tools (for example MySQL Enterprise Monitor).

The ndbinfo views include the node-id for the data node that each piece of data is associated with and in some cases the values will be different for each data node.

ndbinfo.counters

This view presents information on the number of events since the last restart for each data node:

```
mysql> select * from ndbinfo.counters;
+---------+------------+----------------+------------+--------------+------+
| node_id | block_name | block_instance | counter_id | counter_name | val  |
+---------+------------+----------------+------------+--------------+------+
|       3 | DBLQH      |              1 |         10 | OPERATIONS   | 2069 |
|       3 | DBLQH      |              2 |         10 | OPERATIONS   |   28 |
|       4 | DBLQH      |              1 |         10 | OPERATIONS   | 2066 |
|       4 | DBLQH      |              2 |         10 | OPERATIONS   |   27 |
|       3 | DBTC       |              0 |          1 | ATTRINFO     |  140 |
|       3 | DBTC       |              0 |          2 | TRANSACTIONS |   19 |
|       3 | DBTC       |              0 |          3 | COMMITS      |   19 |
|       3 | DBTC       |              0 |          4 | READS        |   19 |
|       3 | DBTC       |              0 |          5 | SIMPLE_READS |    0 |
|       3 | DBTC       |              0 |          6 | WRITES       |    0 |
|       3 | DBTC       |              0 |          7 | ABORTS       |    0 |
|       3 | DBTC       |              0 |          8 | TABLE_SCANS  |    0 |
|       3 | DBTC       |              0 |          9 | RANGE_SCANS  |    0 |
|       4 | DBTC       |              0 |          1 | ATTRINFO     |    2 |
|       4 | DBTC       |              0 |          2 | TRANSACTIONS |    1 |
|       4 | DBTC       |              0 |          3 | COMMITS      |    1 |
|       4 | DBTC       |              0 |          4 | READS        |    1 |
|       4 | DBTC       |              0 |          5 | SIMPLE_READS |    0 |
|       4 | DBTC       |              0 |          6 | WRITES       |    0 |
|       4 | DBTC       |              0 |          7 | ABORTS       |    0 |
|       4 | DBTC       |              0 |          8 | TABLE_SCANS  |    1 |
|       4 | DBTC       |              0 |          9 | RANGE_SCANS  |    0 |
+---------+------------+----------------+------------+--------------+------+
```

The `block_name` refers to the software component that the counter is associated with – this gives some extra context to the information:

- **DBLQH**: Local, low-level query handler block, which manages data and transactions local to the cluster's data nodes, and acts as a coordinator of 2-phase commits.
- **DBTC**: Handles distributed transactions and other data operations on a global level

The `block_instance` field is used to distinguish between the different Local Query Handles threads when using the multi-threaded data node.

The counter_name indicates what events are being counted and the val field provides the current count (since the last restart for the node).

This information can be used to understand how an application's SQL queries (or NDB API method calls) are mapped into data node operations and can be invaluable when tuning the database and application.

ndbinfo.logbuffers

This view presents information on the size of the redo and undo log buffers as well as how much of that space is being used:

```
mysql> select * from ndbinfo.logbuffers;
+---------+----------+--------+----------+----------+--------+
| node_id | log_type | log_id | log_part | total    | used   |
+---------+----------+--------+----------+----------+--------+
|       3 | REDO     |      0 |        1 | 67108864 | 131072 |
|       3 | REDO     |      0 |        2 | 67108864 | 131072 |
|       3 | DD-UNDO  |      4 |        0 |  2096128 |      0 |
|       4 | REDO     |      0 |        1 | 67108864 | 131072 |
|       4 | REDO     |      0 |        2 | 67108864 | 131072 |
|       4 | DD-UNDO  |      4 |        0 |  2096128 |      0 |
+---------+----------+--------+----------+----------+--------+
```

This information can be used to identify when the buffers are almost full so that the administrator can increase their size – preempting potential issues. If the redo log buffers are exhausted then applications will see 1221 "REDO log buffers overloaded" errors. In extreme cases, if the undo log buffers fill too quickly then the database may be halted.

When using the multi-threaded data node (ndbmtd), the log_part column is used to distinguish between different LQH threads.

The sizes of the redo buffer can be increased using the RedoBuffer parameter. The size of the undo buffer is specified in the undo_buffer_size attribute when creating the log file.

ndbinfo.logspaces

This view provides information on the disk space that has been configured for the log spaces for redo and undo logs:

```
mysql> select * from logspaces;
+---------+----------+--------+----------+-----------+--------+
| node_id | log_type | log_id | log_part | total     | used   |
+---------+----------+--------+----------+-----------+--------+
|       3 | REDO     |      0 |        0 | 536870912 |      0 |
|       3 | REDO     |      0 |        1 | 536870912 |      0 |
|       3 | REDO     |      0 |        2 | 536870912 |      0 |
|       3 | REDO     |      0 |        3 | 536870912 |      0 |
|       3 | DD-UNDO  |      4 |        0 |  78643200 | 169408 |
|       4 | REDO     |      0 |        0 | 536870912 |      0 |
|       4 | REDO     |      0 |        1 | 536870912 |      0 |
|       4 | REDO     |      0 |        2 | 536870912 |      0 |
|       4 | REDO     |      0 |        3 | 536870912 |      0 |
|       4 | DD-UNDO  |      4 |        0 |  78643200 | 169408 |
+---------+----------+--------+----------+-----------+--------+
```


For the redo log space, there is 1 row for each of the 4 file sets for each data node and the `total` colum is equal to the `NoOfFragmentLogFiles` configuration parameter multiplied by the `FragmentLogFileSize` parameter while the used column is the amount actually used. If the files fill up before a local checkpoint can complete then error code 410 (Out of log file space temporarily) will be observed. That error can now be avoided by increasing `NoOfFragmentLogFiles` and/or `FragmentLogFileSize` if used approaches `total`.

For the undo log space, the `total` column represents the cumulative size of all of the undo log files assigned to the log group, as added using create/alter logfile group or the `InitialLogFileGroup` configuration parameter. Adding extra undo files if used approaches total can be done to avoid getting 1501 errors.

ndbinfo.memoryusage

This view compares the amount of memory and index used to the amount configured for each data node:

```
mysql> select * from memoryusage;
+---------+--------------+------+------+
| node_id | memory_type  | used | max  |
+---------+--------------+------+------+
|       3 | DATA_MEMORY  |  632 | 3144 |
|       4 | DATA_MEMORY  |  632 | 3144 |
|       3 | INDEX_MEMORY |   38 | 2336 |
|       4 | INDEX_MEMORY |   38 | 2336 |
+---------+--------------+------+------+
```

For `data memory`, the `max` column represents the amount of configured memory for the data node – set using the `DataMemory` configuration parameter and `used` represents the amount currently in use by the Cluster tables.

For index memory, the `max` column represents the amount of configured index memory for the data node – set using the `IndexMemory` configuration parameter and `used` represents the amount currently in use by the Cluster tables.

To avoid exhausting the memory, monitor this table and if used approaches max then either:

- Delete obsolete table rows and run `OPTMIMIZE TABLE`
- Increase the value of `IndexMemory` and/or `DataMemory`
- Consider whether it is practical to alter some tables or columns to be stored on disk rather than in memory

ndbinfo.nodes

This view shows status information for every data node in the cluster:

```
mysql> select * from ndbinfo.nodes;
+---------+--------+---------+-------------+
| node_id | uptime | status  | start_phase |
+---------+--------+---------+-------------+
|       3 |  16548 | STARTED |           0 |
|       4 |     17 | STARTED |           0 |
+---------+--------+---------+-------------+
```

The `uptime` column shows the time in seconds since the data node was started or last restarted.

The status is one of NOTHING, CMVMI, STARTING, STARTED, SINGLEUSER, STOPPING_1, STOPPING_2, STOPPING_3, or STOPPING_4. If the status is set to STARTING then the start phase is also displayed.

ndbinfo.transporters

This is a view showing the status of the connection between each data node and each of the other nodes in the cluster (data nodes, management nodes, MySQL Server nodes and any NDB API applications):

```
mysql> select * from transporters;
+---------+----------------+--------------+
| node_id | remote_node_id | status       |
+---------+----------------+--------------+
|       3 |              1 | CONNECTED    |
|       3 |              3 | DISCONNECTED |
|       3 |              4 | CONNECTED    |
|       3 |            102 | CONNECTED    |
|       3 |            103 | CONNECTING   |
|       3 |            104 | CONNECTING   |
|       4 |              1 | CONNECTED    |
|       4 |              3 | CONNECTED    |
|       4 |              4 | DISCONNECTED |
|       4 |            102 | CONNECTED    |
|       4 |            103 | CONNECTING   |
|       4 |            104 | CONNECTING   |
+---------+----------------+--------------+
```

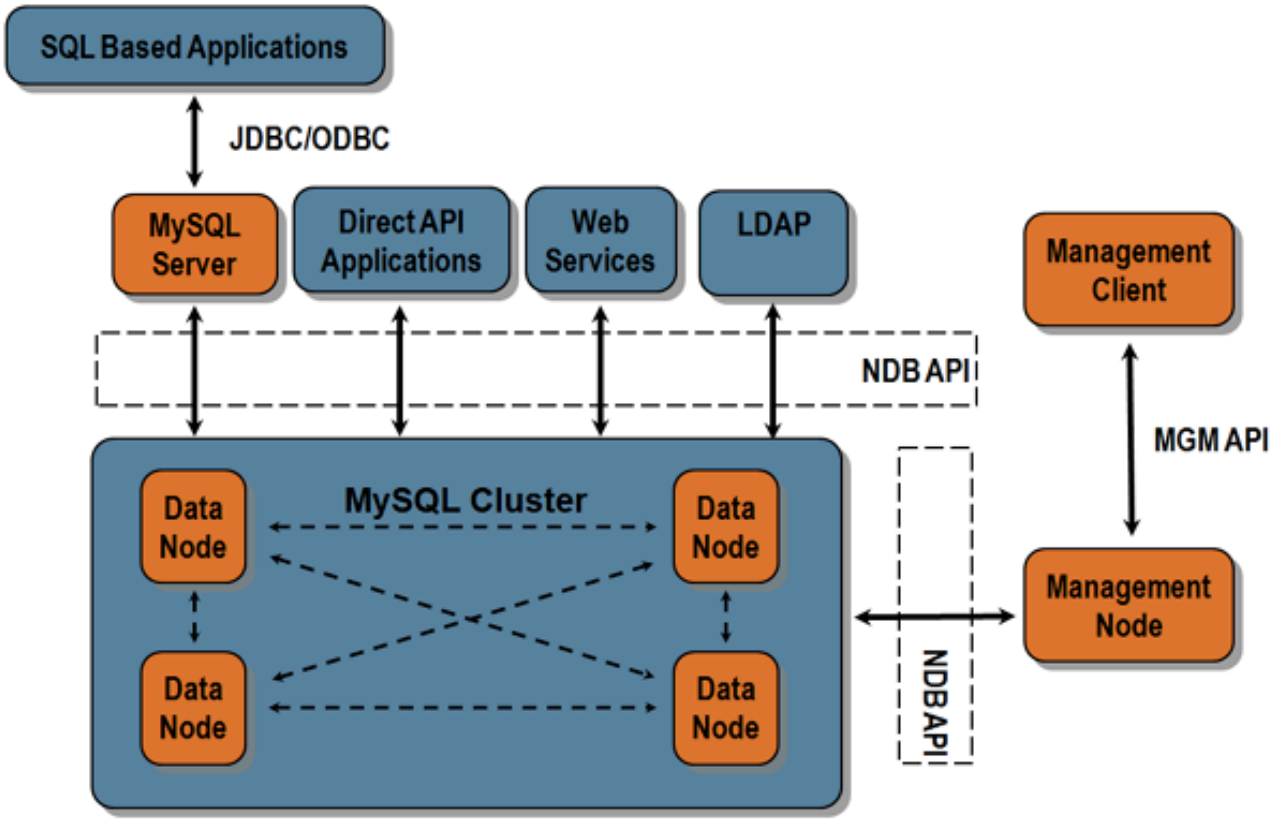
Exercise

1. Connect to Cluster using mysql client
2. open the ndbinfo database:
   `mysql> use ndbinfo;`
3. Check what tables are visible:
4. `mysql> show tables;`
5. Query the memory usage table:
6. `mysql> select * from memoryusage;`
7. Increase DataMemory in config.ini by 10% and restart 1 data node
8. Check memoryusage again
   `mysql> select * from memoryusage;`

NDB API

NDB API Overview

The NDB API is the real-time, C++ interface that provides access to the data held in the data nodes. Typically, applications access this through another layer (e.g. via a MySQL Server). The NDB APU is used extremely successfully by many mission-critical, real-time customer applications. You should consider using the NDB API if the following criteria are met:



- C++ application
- You need maximum real-time performance (latency & throughput)
- Willing to accept added complexity in your application and lock-in to a proprietary interface

The API is documented at: http://dev.mysql.com/doc/ndbapi/en/index.html

There are a number of alternate ways to more easily exploit many of the performance benefits of the NDB API while making life easier for the developer:

- back-ndb plugin for OpenLDAP
- MySQL Cluster Connector for Java (including OpenJPA)
- http plugin for Apache

Example NDB API Code

```
#include <mysql.h>
#include <NdbApi.hpp>
#include <stdio.h>
#include <iostream>
#include <cstdlib>
#include <string.h>

static void run_application(MYSQL &, Ndb_cluster_connection &);

#define PRINT_ERROR(code,msg) \
 std::cout << "Error in " << __FILE__ << ", line: " << __LINE__ \
     << ", code: " << code \
     << ", msg: " << msg << "." << std::endl
#define MYSQLERROR(mysql) { \
 PRINT_ERROR(mysql_errno(&mysql),mysql_error(&mysql)); \
 exit(-1); }
#define APIERROR(error) { \
 PRINT_ERROR(error.code,error.message); \
 exit(-1); }

int main(int argc, char** argv)
{
 if (argc != 3)
  {
  std::cout << "Arguments are <mysqld port> <connect_string>.\n";
  exit(-1);
  }
 // ndb_init must be called first
 ndb_init();

 // connect to mysql server and cluster and run application
  {
  int mysqld_port  = atoi (argv[1]);
  const char *connectstring = argv[2];

  // Object representing the cluster
  Ndb_cluster_connection cluster_connection(connectstring);

  // Connect to cluster management server (ndb_mgmd)
  if (cluster_connection.connect(4 /* retries               */,
                                 5 /* delay between retries */,
                                 1 /* verbose               */))
    {
   std::cout << "Cluster management server was not ready within 30 secs.\n";
   exit(-1);
    }

  // Optionally connect and wait for the storage nodes (ndbd's)
  if (cluster_connection.wait_until_ready(30,0) < 0)
    {
   std::cout << "Cluster was not ready within 30 secs.\n";
   exit(-1);
    }

  // connect to mysql server
  MYSQL mysql;
  if ( !mysql_init(&mysql) ) {
   std::cout << "mysql_init failed\n";
   exit(-1);
    }
  if ( !mysql_real_connect(&mysql, "localhost", "root", "", "",
                           mysqld_port, NULL, 0) )
   MYSQLERROR(mysql);

  // run the application code
  run_application(mysql, cluster_connection);
  }

 ndb_end(0);

 return 0;
}

static void create_table(MYSQL &);
static void drop_table(MYSQL &);
static void do_insert(Ndb &);
static void do_indexScan(Ndb &);
```



```
static void run_application(MYSQL &mysql,
                            Ndb_cluster_connection &cluster_connection)
{
 /****************************************
  * Connect to database via mysql-c          *
  ****************************************/
 mysql_query(&mysql, "CREATE DATABASE TEST_DB_1");
 if (mysql_query(&mysql, "USE TEST_DB_1") != 0) MYSQLERROR(mysql);
 create_table(mysql);

 /****************************************
  * Connect to database via NdbApi           *
  ****************************************/
 // Object representing the database
 Ndb myNdb( &cluster_connection, "TEST_DB_1" );
 if (myNdb.init()) APIERROR(myNdb.getNdbError());

 do_insert(myNdb);
 do_indexScan(myNdb);

 drop_table(mysql);

 mysql_query(&mysql, "DROP DATABASE TEST_DB_1");
}

/*********************************************************
 * Create a table named COUNTRY if it does not exist     *
 *********************************************************/
static void create_table(MYSQL &mysql)
{
 if (mysql_query(&mysql,
                 "DROP TABLE IF EXISTS"
                 "  COUNTRY"))
  MYSQLERROR(mysql);

 if (mysql_query(&mysql,
                 "CREATE TABLE"
                 "  COUNTRY"
                 "    (SUB_ID INT UNSIGNED NOT NULL PRIMARY KEY,"
                 "     COUNTRY_CODE INT UNSIGNED NOT NULL)"
                 "  ENGINE=NDB"))
  MYSQLERROR(mysql);

}

/*********************************
 * Drop a table named MYTABLENAME
 *********************************/
static void drop_table(MYSQL &mysql)
{
 if (mysql_query(&mysql,
                 "DROP TABLE"
                 "  COUNTRY"))
  MYSQLERROR(mysql);

}

static void do_insert(Ndb &myNdb)
{
 NdbDictionary::Dictionary* myDict= myNdb.getDictionary();
 const NdbDictionary::Table *myTable= myDict->getTable("COUNTRY");

 if (myTable == NULL)
  APIERROR(myDict->getNdbError());

 NdbTransaction *myTransaction= myNdb.startTransaction();
 if (myTransaction == NULL) APIERROR(myNdb.getNdbError());

 NdbOperation *myOperation= myTransaction->getNdbOperation(myTable);
 if (myOperation == NULL) APIERROR(myTransaction->getNdbError());

 myOperation->insertTuple();
 myOperation->equal("SUB_ID", 1);
 myOperation->setValue("COUNTRY_CODE", 44);

 myOperation= myTransaction->getNdbOperation(myTable);
 if (myOperation == NULL) APIERROR(myTransaction->getNdbError());

 myOperation->insertTuple();
```



```
 myOperation->equal("SUB_ID", 2);
 myOperation->setValue("COUNTRY_CODE", 1);

 myOperation= myTransaction->getNdbOperation(myTable);
 if (myOperation == NULL) APIERROR(myTransaction->getNdbError());

 myOperation->insertTuple();
 myOperation->equal("SUB_ID", 4);
 myOperation->setValue("COUNTRY_CODE", 61);

 myOperation= myTransaction->getNdbOperation(myTable);
 if (myOperation == NULL) APIERROR(myTransaction->getNdbError());

 myOperation->insertTuple();
 myOperation->equal("SUB_ID", 5);
 myOperation->setValue("COUNTRY_CODE", 33);

 myOperation= myTransaction->getNdbOperation(myTable);
 if (myOperation == NULL) APIERROR(myTransaction->getNdbError());

 myOperation->insertTuple();
 myOperation->equal("SUB_ID", 7);
 myOperation->setValue("COUNTRY_CODE", 46);

 myOperation= myTransaction->getNdbOperation(myTable);
 if (myOperation == NULL) APIERROR(myTransaction->getNdbError());

 myOperation->insertTuple();
 myOperation->equal("SUB_ID", 8);
 myOperation->setValue("COUNTRY_CODE", 1);

 myOperation= myTransaction->getNdbOperation(myTable);
 if (myOperation == NULL) APIERROR(myTransaction->getNdbError());

 myOperation->insertTuple();
 myOperation->equal("SUB_ID", 9);
 myOperation->setValue("COUNTRY_CODE", 44);

 myOperation= myTransaction->getNdbOperation(myTable);
 if (myOperation == NULL) APIERROR(myTransaction->getNdbError());

 myOperation->insertTuple();
 myOperation->equal("SUB_ID", 10);
 myOperation->setValue("COUNTRY_CODE", 33);

 myOperation= myTransaction->getNdbOperation(myTable);
 if (myOperation == NULL) APIERROR(myTransaction->getNdbError());

 myOperation->insertTuple();
 myOperation->equal("SUB_ID", 12);
 myOperation->setValue("COUNTRY_CODE", 44);

 myOperation= myTransaction->getNdbOperation(myTable);
 if (myOperation == NULL) APIERROR(myTransaction->getNdbError());

 myOperation->insertTuple();
 myOperation->equal("SUB_ID", 13);
 myOperation->setValue("COUNTRY_CODE", 1);

 myOperation= myTransaction->getNdbOperation(myTable);
 if (myOperation == NULL) APIERROR(myTransaction->getNdbError());

 myOperation->insertTuple();
 myOperation->equal("SUB_ID", 14);
 myOperation->setValue("COUNTRY_CODE", 61);

 if (myTransaction->execute( NdbTransaction::Commit ) == -1)
  APIERROR(myTransaction->getNdbError());

 myNdb.closeTransaction(myTransaction);

}


/*****************************************************
 * Read and print all tuples via primary ordered index scan *
 *****************************************************/
static void do_indexScan(Ndb &myNdb)
{
 NdbDictionary::Dictionary* myDict= myNdb.getDictionary();
```



```
const NdbDictionary::Index *myPIndex= myDict->getIndex("PRIMARY", "COUNTRY");

if (myPIndex == NULL)
 APIERROR(myDict->getNdbError());

std::cout << "SUB_ID    COUNTRY_CODE" << std::endl;

NdbTransaction *myTransaction=myNdb.startTransaction();
if (myTransaction == NULL) APIERROR(myNdb.getNdbError());

NdbIndexScanOperation *psop;

/* RecAttrs for NdbRecAttr Api */
NdbRecAttr *recAttrAttr1;
NdbRecAttr *recAttrAttr2;

psop=myTransaction->getNdbIndexScanOperation(myPIndex);

if (psop == NULL) APIERROR(myTransaction->getNdbError());

Uint32 scanFlags=
 NdbScanOperation::SF_OrderBy |
 NdbScanOperation::SF_MultiRange |
 NdbScanOperation::SF_ReadRangeNo;

if (psop->readTuples(NdbOperation::LM_Read,
                     scanFlags,
                     (Uint32) 0,          // batch
                     (Uint32) 0) != 0)    // parallel
 APIERROR (myTransaction->getNdbError());

/* Add a bound
* Tuples where SUB_ID >=2 and < 4
  */
Uint32 low=2;
Uint32 high=4;
Uint32 match=13;

if (psop->setBound("SUB_ID", NdbIndexScanOperation::BoundLE, (char*)&low))
 APIERROR(myTransaction->getNdbError());
if (psop->setBound("SUB_ID", NdbIndexScanOperation::BoundGT, (char*)&high))
 APIERROR(myTransaction->getNdbError());

if (psop->end_of_bound(0))
 APIERROR(psop->getNdbError());

/* Second bound
* Tuples where SUB_ID > 5 and <=9
  */
low=5;
high=9;
if (psop->setBound("SUB_ID", NdbIndexScanOperation::BoundLT, (char*)&low))
 APIERROR(myTransaction->getNdbError());
if (psop->setBound("SUB_ID", NdbIndexScanOperation::BoundGE, (char*)&high))
 APIERROR(myTransaction->getNdbError());

if (psop->end_of_bound(1))
 APIERROR(psop->getNdbError());

/* Third bound
* Tuples where SUB_ID == 13
      */

if (psop->setBound("SUB_ID", NdbIndexScanOperation::BoundEQ, (char*)&match))
 APIERROR(myTransaction->getNdbError());

if (psop->end_of_bound(2))
 APIERROR(psop->getNdbError());


/* Read all columns */
recAttrAttr1=psop->getValue("SUB_ID");
recAttrAttr2=psop->getValue("COUNTRY_CODE");

if(myTransaction->execute( NdbTransaction::Commit ) != 0)
 APIERROR(myTransaction->getNdbError());

if (myTransaction->getNdbError().code != 0)
 APIERROR(myTransaction->getNdbError());
```



```
 while (psop->nextResult(true) == 0)
    {
  printf(" %8d    %8d   Range no : %2d\n",
             recAttrAttr1->u_32_value(),
             recAttrAttr2->u_32_value(),
             psop->get_range_no());
    }
 psop->close();

 myNdb.closeTransaction(myTransaction);

}
```


MySQL Cluster Connector for Java

Technical Overview

Prior to MySQL Cluster 7.1, the choices for Java developers were somewhat limited. The only supported method was JDBC – either directly or through a 3rd party layer such as Java Persistence API (JPA) compliant middleware. If Java users wanted greater performance then they were responsible for writing their own Java Native Interface (JNI) layer sitting between their (Java) application and the (C++) NDB API.

With the introduction of MySQL Cluster 7.1, the options are much wider as shown in Figure 1. Working from left to right the options are:

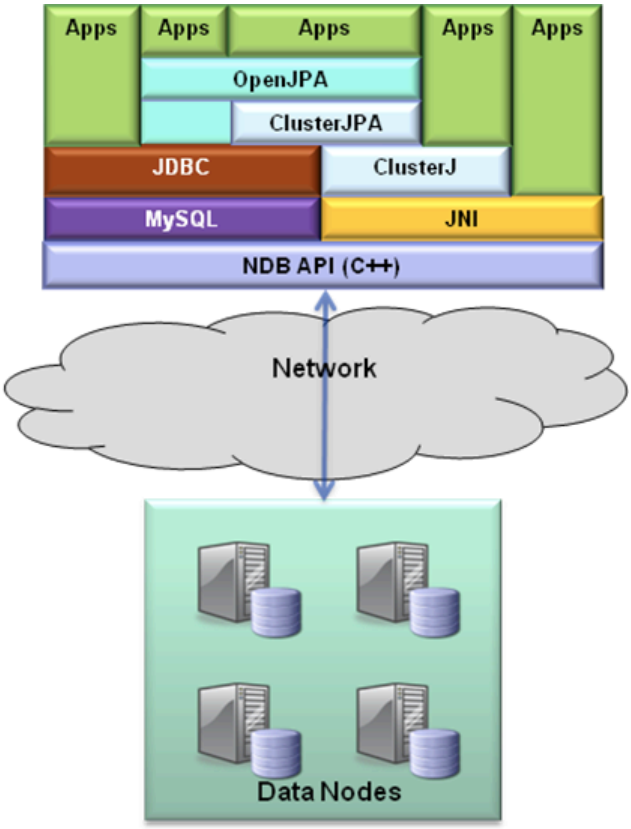


Figure 1: Options for Java Applications

- Java application makes JDBC calls through "JDBC Driver for MySQL (Connector/J)". This is a technology that many Java developers are comfortable and it is very flexible. However it requires that the application developer maps from their Java objects to relational, SQL statements. In addition, performance can be compromised as Connector/J accesses the data through a MySQL Server rather than directly accessing the data nodes through the NDB API.
- Applications can shift responsibility for the Object-Relational-Mapping (ORM) to a 3rd party JPA solution (for example, Hibernate, Toplink or OpenJPA). While this frees the Java developer from having to work with a relational data model, performance can still be limited if the JPA layer is having to access the data via JDBC.
- MySQL Cluster 7.1 introduces a plug-in for OpenJPA (ClusterJPA_ which allows most JPA operations to be performed through the NDB API (via a new "ClusterJ" layer) with the remainder using JDBC. This gives the Java developer the luxury of working purely with objects while still getting much of the performance benefits of using the NDB API.

- Application developers can choose to bypass the JPA layer and instead go directly to the ClusterJ layer. This introduces some limitations but it may be appropriate for developers wanting to get the best possible performance and/or wanting to avoid introducing additional 3rd party components (OpenJPA). ClusterJ is introduced by MySQL Cluster 7.1.
- Finally, Java application developers still have the option to implement their own wrapper (typically using JNI) to act as the Java/C++ mediator.

ClusterJ

ClusterJ is designed to provide a high performance method for Java applications to store and access data in a MySQL Cluster database. It is also designed to be easy for Java developers to use and is "in the style of" Hibernate/Java Data Objects (JDO) and JPA. It uses the Domain Object Model DataMapper pattern:

- Data is represented as domain objects
- Domain objects are separate from business logic
- Domain objects are mapped to database tables

The purpose of ClusterJ is to provide a mapping from the relational view of the data stored in MySQL Cluster to the Java objects used by the application. This is achieved by annotating interfaces representing the Java objects; where each persistent interface is mapped to a table and each property in that interface to a column. By default, the table name will match the interface name and the column names match the property names but this can be overwritten using the annotations.

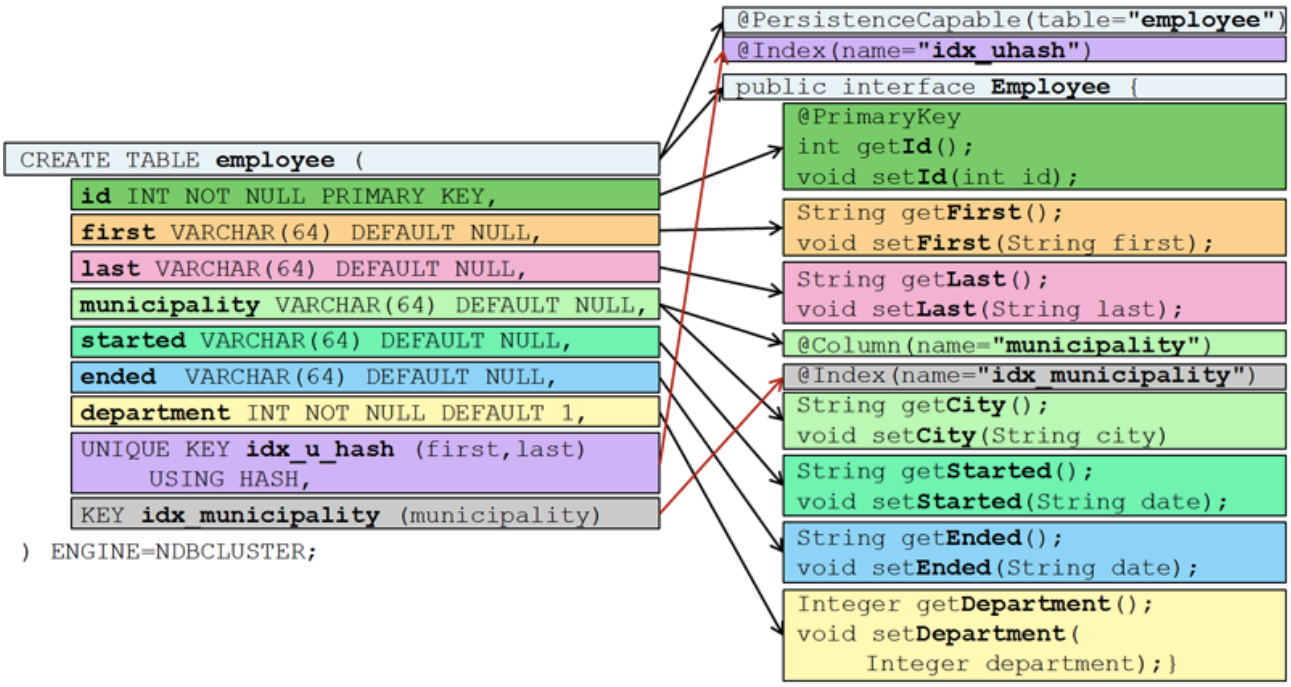


Figure 2: ClusterJ Annotation example

If the table does not already exist (for example, this is a brand new application with new data) then the table must be created manually - unlike OpenJPA, ClusterJ will not create the table automatically.
Figure 2 shows an example of an interface that has been created in order to represent the data held in the 'employee' table.

ClusterJ uses the following concepts:

- **SessionFactory**: There is one instance per MySQL Cluster instance for each Java Virtual Machine (JVM). The SessionFactory object is used by the application to get hold of sessions. The configuration details for the ClusterJ instance are defined in the Configuration properties which is an artifact associated with the SessionFactory.
- **Session**: There is one instance per user (per Cluster, per JVM) and represents a Cluster connection
- **Domain Object**: Objects representing the data from a table. The domain objects (and their relationships to the Cluster tables) are defined by annotated interfaces (as shown in the right-hand side of Figure 2.

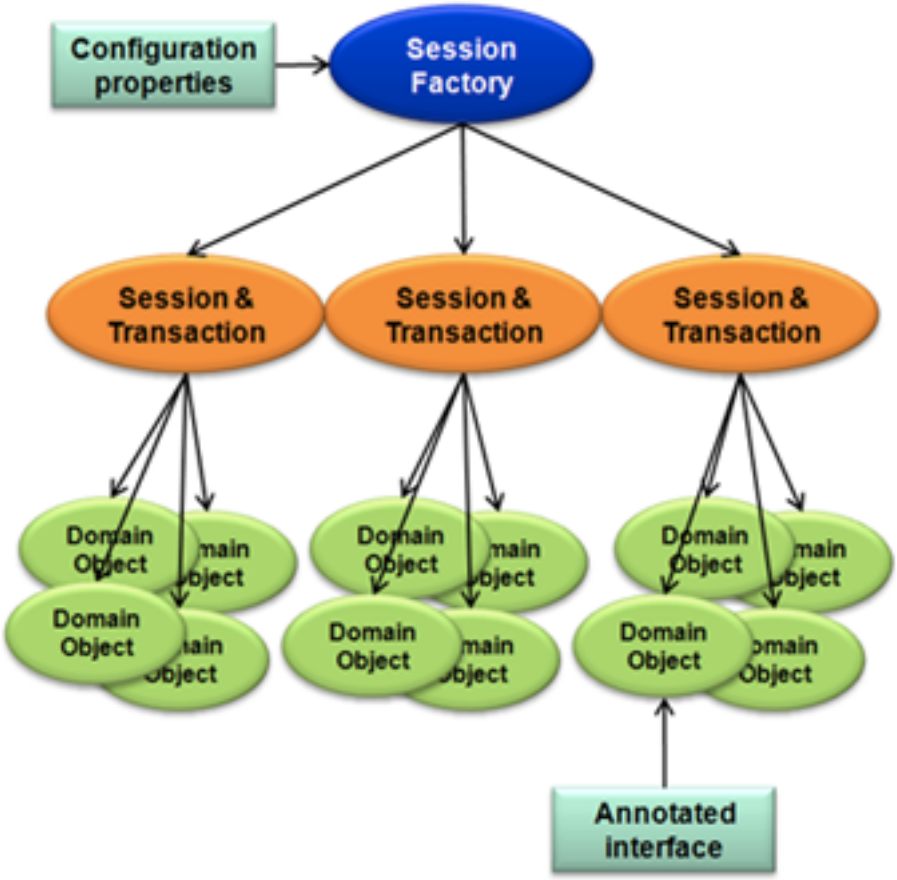


Figure 3: ClusterJ terminology

- **Transaction**: There is one transaction per session at any point in time. By default, each query is run under its own transaction.

ClusterJ will be suitable for many Java developers but it has some restrictions which may make OpenJPA with the ClusterJPA plug-in more appropriate. These ClusterJ restrictions are:

- Persistent Interfaces rather than persistent classes. The developer provides the signatures for the getter/setter methods rather than the properties and no extra methods can be added.
- No Relationships between properties or between objects can be defined in the domain objects. Properties are primitive types.
- No Multi-table inheritance; there is a single table per persistent interface
- No joins in queries (all data being queried must be in the same table/interface)
- No Table creation - user needs to create tables and indexes
- No Lazy Loading - entire record is loaded at one time, including large object (LOBs).

ClusterJPA

JPA is the Java standard for persistence and different vendors can implement their own implementation of this API and they can (and do) add proprietary extensions. Three of the most common implementations are OpenJPA, Hibernate and Toplink. JPA can be used within server containers or outside of them (i.e. with either J2EE or J2SE).

OpenJPA > ClusterJ > NDB API	OpenJPA > Connector/J > MySQL > NDB API
Insert	
Delete	
Find (Primary Key reads)	
Update	
	Other queries

Typically a JPA implementation would access the database (for example, MySQL Cluster) using JDBC. JDBC gives a great deal of flexibility to the JPA implementer but it cannot give the best performance when using MySQL Cluster as there is an internal conversion to SQL by Connector/J and a subsequent translation from SQL to the C++ NDB API by the MySQL Server. As of MySQL Cluster 7.1, OpenJPA can be configured to use the high performance NDB API (via ClusterJ) for most operations but fall back on JDBC for more complex queries.

The first implementation of ClusterJPA is as an OpenJPA BrokerFactory but in the future, it may be extended to work with other JPA implementations.

ClusterJPA overcomes ClusterJ limitations, notably:

- Persistent classes
- Relationships
- Joins in queries
- Lazy loading
- Table and index creation from object model

Typically users base their selection of a JPA solution on factors such as proprietary extensions, what existing applications already use and (increasingly with ClusterJPA) performance.

The performance of ClusterJPA (OpenJPA using ClusterJ) has been compared with OpenJPA using JDBC in Figure 4. It should be noted that the performance is significantly better when using ClusterJPA (the yellow bar). It is hoped that in the future the performance can be improved even further for finds, updates and deletes.

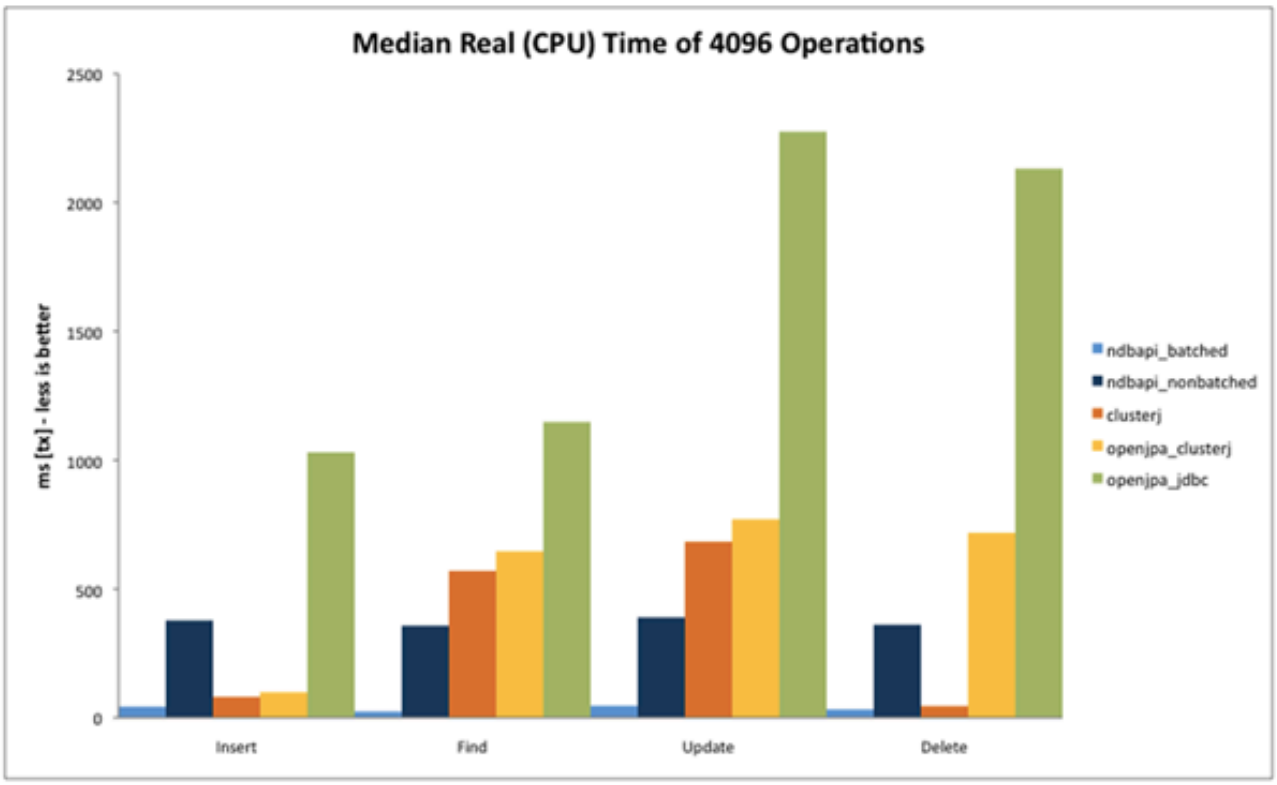


Figure 4: Performance of OpenJPA

Adapting an OpenJPA based application to use ClusterJPA with MySQL Cluster should be fairly straight-forward with the main change being in the definition of the persistence unit in persistence.xml:

```
<persistence xmlns=http://java.sun.com/xml/ns/persistence xmlns:xsi="http://www.w3.org/2001/
XMLSchema-instance" version="1.0">
  <persistence-unit name="clusterdb" transaction-type="RESOURCE_LOCAL“>
    <provider> org.apache.openjpa.persistence.PersistenceProviderImpl </provider>
    <class>Employee</class>
    <class>Department</class>
    <properties>
      <property name="openjpa.jdbc.SynchronizeMappings" value="buildSchema" />
      <property name="openjpa.ConnectionDriverName"
             value="com.mysql.jdbc.Driver" />
      <property name="openjpa.ConnectionURL" value="jdbc:mysql://localhost:3306/clusterdb" />
      <property name="openjpa.ConnectionUserName" value="root" />
      <property name="openjpa.ConnectionPassword" value="" />
      <property name="openjpa.BrokerFactory" value="ndb" />
      <property name="openjpa.ndb.connectString" value="localhost:1186" />
      <property name="openjpa.jdbc.DBDictionary" value="TableType=ndb"/>
      <property name="openjpa.ndb.database" value="clusterdb" /
    </properties>
  </persistence-unit>
</persistence>
```

Defining the object to table mappings is performed by annotating the persistent class for the domain object. If not already in existence, OpenJPA will create the table (although the user is responsible for changing the storage engine for the table to NDB so that ClusterJPA can perform correctly).

This paper does not go into the use of JPA in great depth – focusing instead on the specifics of using OpenJPA with MySQL Cluster/ClusterJPA. For more information on the use of JPA and OpenJPA, refer to http://openjpa.apache.org/ .

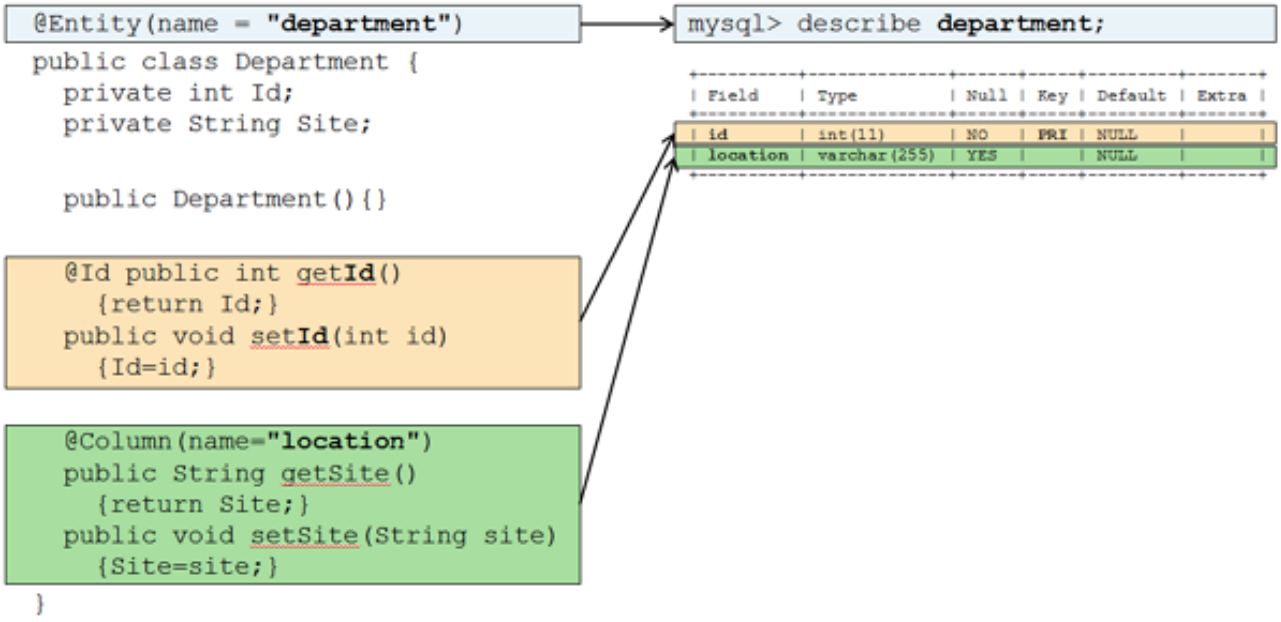


Figure 5: ClusterJPA Class annotation.

Pre-requisites for Tutorial

The tutorials are using MySQL Cluster 7.1.2 on Fedora 12. If using more recent versions of MySQL Cluster then you may need to change the class-paths as explained in http://dev.mysql.com/doc/ndbapi/en/mccj-using-clusterj.html and http://dev.mysql.com/doc/ndbapi/en/mccj-using-jpa.html

For both of these tutorials, it is necessary to have MySQL Cluster up and running. For simplicity all of the nodes (processes) making up the Cluster will be run on the same physical host, along with the application.

Although most of the database access is performed through the NDB API, the Cluster includes a MySQL Server process for OpenJPA to use for complex queries and to allow the user to check the contents of the database manually.

These are the MySQL Cluster configuration files being used (during the tutorial, you can use the Cluster that has already been created):

config.ini:

```
[ndbd default]noofreplicas=2
datadir=/home/billy/mysql/my_cluster/data

[ndbd]
hostname=localhost
id=3

[ndbd]
hostname=localhost
id=4

[ndb_mgmd]
id = 1
hostname=localhost
datadir=/home/billy/mysql/my_cluster/data

[mysqld]
hostname=localhost
id=101

[api]
hostname=localhost
```

my.cnf:

```
[mysqld]
ndbcluster
datadir=/home/billy/mysql/my_cluster/data
basedir=/usr/local/mysql
```

These tutorials focus on ClusterJ and ClusterJPA rather than on running MySQL Cluster; if you are new to MySQL Cluster then refer to http://www.clusterdb.com/mysql-cluster/creating-a-simple-cluster-on-a-single-linux-host/ before trying these tutorials.

ClusterJ Tutorial

ClusterJ needs to be told how to connect to our MySQL Cluster database; including the connect string (the address/port for the management node), the database to use, the user to login as and attributes for the connection such as the timeout values. If these parameters aren't defined then ClusterJ will fail with run-time exceptions. This information represents the "configuration properties" shown in Figure 3.
These parameters can be hard coded in the application code but it is more maintainable to create a clusterj.properties file that will be imported by the application. This file should be stored in the same directory as your application source code.

clusterj.properties:

```
com.mysql.clusterj.connectstring=localhost:1186
com.mysql.clusterj.database=clusterdb
com.mysql.clusterj.connect.retries=4
com.mysql.clusterj.connect.delay=5
com.mysql.clusterj.connect.verbose=1
com.mysql.clusterj.connect.timeout.before=30
com.mysql.clusterj.connect.timeout.after=20
com.mysql.clusterj.max.transactions=1024
```

As ClusterJ will not create tables automatically, the next step is to create 'clusterdb' database (referred to in clusterj.properties) and the 'employee' table:


```
[billy@ws1 ~]$ mysql -u root -h 127.0.0.1 -P 3306 -u root
mysql>  create database clusterdb;use clusterdb;
mysql> CREATE TABLE employee (
  ->     id INT NOT NULL PRIMARY KEY,
  ->     first VARCHAR(64) DEFAULT NULL,
  ->     last VARCHAR(64) DEFAULT NULL,
  ->     municipality VARCHAR(64) DEFAULT NULL,
  ->     started VARCHAR(64) DEFAULT NULL,
  ->     ended  VARCHAR(64) DEFAULT NULL,
  ->     department INT NOT NULL DEFAULT 1,
  ->     UNIQUE KEY idx_u_hash (first,last) USING HASH,
  ->     KEY idx_municipality (municipality)
  -> ) ENGINE=NDBCLUSTER;
```

The next step is to create the annotated interface:

***Employee.java**:*

```
import com.mysql.clusterj.annotation.Column;
import com.mysql.clusterj.annotation.Index;
import com.mysql.clusterj.annotation.PersistenceCapable;
import com.mysql.clusterj.annotation.PrimaryKey;

@PersistenceCapable(table="employee")
@Index(name="idx_uhash")
public interface Employee {

  @PrimaryKey
  int getId();
  void setId(int id);

  String getFirst();
  void setFirst(String first);
  String getLast();
  void setLast(String last);

  @Column(name="municipality")
  @Index(name="idx_municipality")
  String getCity();
  void setCity(String city);

  String getStarted();
  void setStarted(String date);

  String getEnded();
  void setEnded(String date);

  Integer getDepartment();
  void setDepartment(Integer department);
}
```

The name of the table is specified in the annotation `@PersistenceCapable (table="employee")` and then each column from the employee table has an associated getter and setter method defined in the interface. By default, the property name in the interface is the same as the column name in the table – that has been overridden for the City property by explicitly including the `@Column(name="municipality")` annotation just before the associated getter/setter methods. The `@PrimaryKey` annotation is used to identify the property who's associated column is the Primary Key in the table. ClusterJ is made aware of the existence of indexes in the database using the `@Index` annotation.

The next step is to write the application code which we step through here block by block; the first of which simply contains the import statements and then loads the contents of the clusterj.properties defined above (**Note** – refer to section 3.1.1 for details on compiling and running the tutorial code):

***Main.java (part 1)**:*

```
import com.mysql.clusterj.ClusterJHelper;
import com.mysql.clusterj.SessionFactory;
import com.mysql.clusterj.Session;
import com.mysql.clusterj.Query;
import com.mysql.clusterj.query.QueryBuilder;
import com.mysql.clusterj.query.QueryDomainType;

import java.io.File;
import java.io.InputStream;
import java.io.FileInputStream;
import java.io.*;

import java.util.Properties;
import java.util.List;

public class Main {

public static void main (String[] args) throws
java.io.FileNotFoundException,java.io.IOException {

 // Load the properties from the clusterj.properties file

 File propsFile = new File("clusterj.properties");
 InputStream inStream = new FileInputStream(propsFile);
 Properties props = new Properties();
 props.load(inStream);

 //Used later to get userinput
 BufferedReader br = new BufferedReader(new
      InputStreamReader(System.in));
```

The next step is to get a handle for a `SessionFactory` from the `ClusterJHelper` class and then use that factory to create a session (based on the properties imported from clusterj.properties file.

***Main.java (part 2)**:*

```
 // Create a session (connection to the database)
 SessionFactory factory = ClusterJHelper.getSessionFactory(props);
 Session session = factory.getSession();
```

Now that we have a session, it is possible to instantiate new Employee objects and then persist them to the database. Where there are no transaction `begin()` or `commit()` statements, each operation involving the database is treated as a separate transaction.

***Main.java (part 3)**:*

```
  // Create and initialise an Employee
              Employee newEmployee = session.newInstance(Employee.class);
      newEmployee.setId(988);
      newEmployee.setFirst("John");
  newEmployee.setLast("Jones");
 newEmployee.setStarted("1 February 2009");
 newEmployee.setDepartment(666);

 // Write the Employee to the database
 session.persist(newEmployee);
```

At this point, a row will have been added to the 'employee' table. To verify this, a new `Employee` object is created and used to read the data back from the 'employee' table using the primary key (`Id`) value of `998`:

***Main.java (part 4)**:*

```
 // Fetch the Employee from the database
 Employee theEmployee = session.find(Employee.class, 988);

 if (theEmployee == null)
  {System.out.println("Could not find employee");}
 else
  {System.out.println ("ID: " + theEmployee.getId() + "; Name: " +
      theEmployee.getFirst() + " " + theEmployee.getLast());
      System.out.println ("Location: " + theEmployee.getCity());
      System.out.println ("Department: " + theEmployee.getDepartment());
      System.out.println ("Started: " + theEmployee.getStarted());
      System.out.println ("Left: " + theEmployee.getEnded());
  }
```

This is the output seen at this point:

```
ID: 988; Name: John Jones
Location: null
Department: 666
Started: 1 February 2009
Left: null
Check the database before I change the Employee - hit return when you are done
```

The next step is to modify this data but it **does not** write it back to the database yet:

***Main.java (part 5)**:*

```
 // Make some changes to the Employee & write back to the database
 theEmployee.setDepartment(777);
 theEmployee.setCity("London");

 System.out.println("Check the database before I change the Employee -
    hit return when you are done");
 String ignore = br.readLine();
```

The application will pause at this point and give you chance to check the database to confirm that the original data has been added as a new row **but** the changes have not been written back yet:


```
mysql> select * from clusterdb.employee;
+-----+-------+-------+-------------+----------------+-------+------------+
| id  | first | last  | municipality | started        | ended | department |
+-----+-------+-------+-------------+----------------+-------+------------+
| 988 | John  | Jones | NULL         | 1 February 2009 | NULL  |        666 |
+-----+-------+-------+-------------+----------------+-------+------------+
```

After hitting return, the application will continue and write the changes to the table.

***Main.java (part 6)**:*

```
session.updatePersistent(theEmployee);

System.out.println("Check the change in the table before I bulk add
    Employees - hit return when you are done");
ignore = br.readLine();
```

The application will again pause so that we can now check that the change has been written back (persisted) to the database:

```
mysql> select * from clusterdb.employee;
+-----+-------+-------+-------------+----------------+-------+------------+
| id  | first | last  | municipality | started        | ended | department |
+-----+-------+-------+-------------+----------------+-------+------------+
| 988 | John  | Jones | London       | 1 February 2009 | NULL  |        777 |
+-----+-------+-------+-------------+----------------+-------+------------+
```

The application then goes onto create and persist 100 new employees. To improve performance, a single transaction is used to that all of the changes can be written to the database at once when the `commit()` statement is run:

***Main.java (part 7)**:*

```
// Add 100 new Employees - all as part of a single transaction
newEmployee.setFirst("Billy");
newEmployee.setStarted("28 February 2009");

session.currentTransaction().begin();

 for (int i=700;i<800;i++) {
  newEmployee.setLast("No-Mates"+i);
      newEmployee.setId(i+1000);
        newEmployee.setDepartment(i);
                  session.persist(newEmployee);
   }

session.currentTransaction().commit();
```

The 100 new employees will now have been persisted to the database. The next step is to create and execute a query that will search the database for all employees in department 777 by using a `QueryBuilder` and using that to build a `QueryDomain` that compares the 'department' column with a parameter. After creating the, the department parameter is set to 777 (the query could subsequently be reused with different department numbers). The application then runs the query and iterates through and displays each of employees in the result set:

Main.java (part 8):

```
// Retrieve the set all of Employees in department 777
QueryBuilder builder = session.getQueryBuilder();
QueryDomainType<Employee> domain =
     builder.createQueryDefinition(Employee.class);
domain.where(domain.get("department").equal(domain.param(
     "department")));
Query<Employee> query = session.createQuery(domain);
query.setParameter("department",777);

List<Employee> results = query.getResultList();
for (Employee deptEmployee: results) {
 System.out.println ("ID: " + deptEmployee.getId() + "; Name: " +
       deptEmployee.getFirst() + " " + deptEmployee.getLast());
 System.out.println ("Location: " + deptEmployee.getCity());
 System.out.println ("Department: " + deptEmployee.getDepartment());
     System.out.println ("Started: " + deptEmployee.getStarted());
   System.out.println ("Left: " + deptEmployee.getEnded());
  }

System.out.println("Last chance to check database before emptying table
 - hit return when you are done");
ignore = br.readLine();
```

At this point, the application will display the following and prompt the user to allow it to continue:

```
ID: 988; Name: John Jones
Location: London
Department: 777
Started: 1 February 2009
Left: null
ID: 1777; Name: Billy No-Mates777
Location: null
Department: 777
Started: 28 February 2009
Left: null
```

We can compare that output with an SQL query performed on the database:

```
mysql> select * from employee where department=777;
+------+-------+-------------+--------------+------------------+-------+------------+
| id   | first | last        | municipality | started          | ended | department |
+------+-------+-------------+--------------+------------------+-------+------------+
|  988 | John  | Jones       | London       | 1 February 2009  | NULL  |        777 |
| 1777 | Billy | No-Mates777 | NULL         | 28 February 2009 | NULL  |        777 |
+------+-------+-------------+--------------+------------------+-------+------------+
```

Finally, after pressing return again, the application will remove all employees:

Main.java (part 8):

```
session.deletePersistentAll(Employee.class);
           }
         }
```

As a final check, an SQL query confirms that all of the rows have been deleted from the 'employee' table.

```
mysql> select * from employee;
Empty set (0.00 sec)
```


Compiling and running the ClusterJ tutorial code

```
cd /usr/local/clusterj_example_wp_7_1_2a

javac -classpath /usr/local/mysql/share/mysql/java/clusterj-api.jar:. Main.java
Employee.java

java -classpath /usr/local/mysql/share/mysql/java/clusterj.jar:. -
Djava.library.path=/usr/local/mysql/lib Main
```

OpenJPA/ClusterJPA Tutorial

JPA/OpenJPA/ClusterJPA can be used within or outside a container (i.e. it can be used with J2EE or J2SE) – for simplicity, this tutorial does not use a container (i.e. it is written using J2SE).

Before being able to run any ClusterJPA code, you first need to download and install OpenJPA from http://openjpa.apache.org/ - this tutorial uses OpenJPA 1.2.1. Simply extract the contents of the binary tar ball to the host you want to run your application on; for this tutorial, I use /usr/local/openjpa.

Additionally, ClusterJPA must sometimes use JDBC to satisfy certain queries and so "JDBC Driver for MySQL (Connector/J)" should also be installed – this can be downloaded from http://dev.mysql.com/downloads/connector/j/ Again, simply extract the contents of the tar ball, for this tutorial the files are stored in /usr/local/connectorj and version 5.1.12 is used.

If the ClusterJ tutorial has already been run on this MySQL Cluster database then drop the tables from the cluster so that you can observe them being created automatically – though in a real application, you may prefer to create them manually.

A configuration file is required to indicate how persistence is to be handled for the application. Create a new directory called META-INF in the application source directory and within there create a file called persistence.xml:

```
<persistence xmlns="http://java.sun.com/xml/ns/persistence"
      xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" version="1.0">
  <persistence-unit name="clusterdb" transaction-type="RESOURCE_LOCAL">
    <provider>
      org.apache.openjpa.persistence.PersistenceProviderImpl
    </provider>
    <class>Employee</class>
    <class>Department</class>
    <properties>
      <property name="openjpa.jdbc.SynchronizeMappings" value="buildSchema" />
      <property name="openjpa.ConnectionDriverName"
          value="com.mysql.jdbc.Driver" />
      <property name="openjpa.ConnectionURL"
          value="jdbc:mysql://localhost:3306/clusterdb" />
      <property name="openjpa.ConnectionUserName" value="root" />
      <property name="openjpa.ConnectionPassword" value="" />
      <property name="openjpa.BrokerFactory" value="ndb" />
      <property name="openjpa.jdbc.DBDictionary" value="TableType=ndb"/>
      <property name="openjpa.ndb.connectString" value="localhost:1186" />
      <property name="openjpa.ndb.database" value="clusterdb" />
    </properties>
  </persistence-unit>
</persistence>
```


A persistence unit called 'clusterdb' is created; the provider (implementation for the persistence) is set to openjpa (as opposed for example to hibernate). Two classes are specified – 'Employee' and 'Department' which relate to the persistent classes that the application will define. Connector/J is defined as the JDBC connection (together with the host and the port of the MySQL Server to be used). The key to having OpenJPA use ClusterJPA is to set the BrokerFactory to ndb and specify the connect string (host:port) for the MySQL Cluster management node. The database is defined to be 'clusterdb' for both the JDBC and ClusterJ connections.

If not already done so, create the 'clusterdb' database (if it already contains tables from the ClusterJ tutorial then drop them):

```
mysql> create database clusterdb;
```

The next step is to create the persistent class definitions for the Department and Employee Entities:

Department.java:

```
  import javax.persistence.*;

 @Entity(name = "department")
 public class Department {

  private int Id;
  private String Site;

  public Department(){}

  @Id public int getId() {return Id;}
  public void setId(int id) {Id=id;}

  @Column(name="location")
  public String getSite() {return Site;}
  public void setSite(String site) {Site=site;}

  public String toString() {
      return "Department: " + getId() + " based in " + getSite();
    }
  }
```

Using the @Entity tag, the table name is specified to be 'department'. Note that unlike ClusterJ, ClusterJPA uses persistent classes (rather than interfaces) and so it is necessary to define the properties as well as the getter/setter methods. The primary key is defined using the @Id tag and we specify that the column associated with the Site property should be called 'location' using the @Column tag.

As this is a class, it is possible to add other useful methods – in this case toString().

Employee.java:

```
import javax.persistence.*;

@Entity(name = "employee") //Name of the table

public class Employee {

 private int Id;
 private String First;
 private String Last;
 private String City;
 private String Started;
 private String Ended;
 private int Department;

 public Employee(){}

 @Id public int getId() {return Id;}
 public void setId(int id) {Id=id;}

 public String getFirst() {return First;}
 public void setFirst(String first) {First=first;}

 public String getLast() {return Last;}
 public void setLast(String last) {Last=last;}

 @Column(name="municipality")
 public String getCity() {return City;}
 public void setCity(String city) {City=city;}

 public String getStarted() {return Started;}
 public void setStarted(String date) {Started=date;}

 public String getEnded() {return Ended;}
 public void setEnded(String date) {Ended=date;}

 public int getDepartment() {return Department;}
 public void setDepartment(int department) {Department=department;}

 public String toString() {
     return getFirst() + " " + getLast() + " (Dept " +
     getDepartment()+ ") from " + getCity() +
     " started on " + getStarted() + " & left on " + getEnded();
   }
 }
```

The next step is to write the application code which we step through here block by block; the first of which simply contains the import statements and then (**Note** – refer to section 3.1.2 for details on compiling and running the tutorial code):

Main.java (part 1):

```
 import java.util.List;
 import javax.persistence.EntityManager;
 import javax.persistence.EntityManagerFactory;
 import javax.persistence.EntityTransaction;
 import javax.persistence.Persistence;
 import javax.persistence.Query;
 import java.io.*;

 public class Main {

  public static void main (String[] args) throws java.io.IOException {

  EntityManagerFactory entityManagerFactory =
Persistence.createEntityManagerFactory("clusterdb");
  EntityManager em = entityManagerFactory.createEntityManager();
   EntityTransaction userTransaction = em.getTransaction();

  BufferedReader br = new BufferedReader(new InputStreamReader(System.in));

  System.out.println("The tables will now have been created - check through
SQL.");
  System.out.println("mysql> use clusterdb;");
  System.out.println("mysql> show tables;");
  System.out.println("Hit return when you are done");
  String ignore = br.readLine();
```

As part of creating the EntityManagerFactory and EntityManager, OpenJPA creates the tables for the two classes specified for the '`clusterdb`' persistence unit. While the application waits for the user to press return, this can be checked:

```
mysql> use clusterdb
mysql> show tables;
+---------------------+
| Tables_in_clusterdb |
+---------------------+
| department          |
| employee            |
+---------------------+
```

After hitting return, the application can create an Employee object and then persist it – at which point it will be stored in the 'employee' table. A second Employee object is then created and populated with the data read back from the database (using a primary key look up on the `Id` property with a value of `1`):

Main.java (part 2):

```
    userTransaction.begin();
   Employee emp = new Employee();
   emp.setId(1);
   emp.setDepartment(666);
   emp.setFirst("Billy");
   emp.setLast("Fish");
   emp.setStarted("1st February 2009");
   em.persist(emp);
  userTransaction.commit();

  userTransaction.begin();
   Employee theEmployee = em.find(Employee.class, 1);
  userTransaction.commit();

  System.out.println(theEmployee.toString());

  System.out.println("Chance to check the database before City is set");
  System.out.println("Hit return when you are done");
  ignore = br.readLine();
```

The Employee object read back from the database is displayed:

```
Billy Fish (Dept 666) from null started on 1st February 2009 & left on null
Chance to check the database before City is set
Hit return when you are done
```

At this point, the application waits to give the user a chance to confirm that the Employee really has been written to the database:

```
mysql> select * from employee;
+----+--------------+------------+-------+-------+------+-------------------+
| id | municipality | department | ended | first | last | started           |
+----+--------------+------------+-------+-------+------+-------------------+
|  1 | NULL         |        666 | NULL  | Billy | Fish | 1st February 2009 |
+----+--------------+------------+-------+-------+------+-------------------+
```

After hitting return, the application continues and an update is made to the persisted `Employee` object – note that there is no need to explicitly ask for the changes to be persisted, this happens automatically when the transaction is committed:

Main.java (part 3):

```
  userTransaction.begin();
   theEmployee.setCity("London");
   theEmployee.setDepartment(777);
  userTransaction.commit();

  System.out.println("Chance to check the City is set in the database");
  System.out.println("Hit return when you are done");
  ignore = br.readLine();
```

At this point, the application again waits while the user has a chance to confirm that the changes did indeed get written through to the database:


```
mysql> select * from employee;
+----+--------------+------------+-------+-------+------+-------------------+
| id | municipality | department | ended | first | last | started           |
+----+--------------+------------+-------+-------+------+-------------------+
|  1 | London       |        777 | NULL  | Billy | Fish | 1st February 2009 |
+----+--------------+------------+-------+-------+------+-------------------+
```

When allowed to continue, the application creates and persists an additional 100 Employee & Department entities. It then goes on to create and execute a query to find all employees with a department number of 777 and then looks up the location of the site for that department.

```
  Department dept;

  userTransaction.begin();
   for (int i=700;i<800;i++) {
       emp = new Employee();
       dept = new Department();
       emp.setId(i+1000);
       emp.setDepartment(i);
       emp.setFirst("Billy");
       emp.setLast("No-Mates-"+i);
       emp.setStarted("1st February 2009");
       em.persist(emp);
       dept.setId(i);
       dept.setSite("Building-"+i);
       em.persist(dept);
       }

  userTransaction.commit();

  userTransaction.begin();

  Query q = em.createQuery("select x from Employee x where x.department=777");
  Query qd;

  for (Employee m : (List<Employee>) q.getResultList()) {
       System.out.println(m.toString());
       qd = em.createQuery("select x from Department x where x.id=777");
       for (Department d : (List<Department>) qd.getResultList()) {
           System.out.println(d.toString());
       }
     }

  userTransaction.commit();
```

These are the results displayed:

```
Billy No-Mates-777 (Dept 777) from null started on 1st February 2009 & left on
null
Department: 777 based in Building-777
Billy Fish (Dept 777) from London started on 1st February 2009 & left on null
Department: 777 based in Building-777
```

Note that joins between tables are possible with JPA but that is beyond the scope of this tutorial.

Finaly, the EntityManager and EntityManagerFactory are closed:

Main.java (part 4):

```
    em.close();
  entityManagerFactory.close();
   }
}
```

Compiling and running the ClusterJPA tutorial code

```
javac -classpath /usr/local/mysql/share/mysql/java/clusterjpa.jar:/usr/local/
openjpa/openjpa-1.2.1.jar:/usr/local/openjpa/lib/geronimo-jpa_3.0_spec-1.0.jar:.
Main.java Employee.java Department.java

java -Djava.library.path=/usr/local/mysql/lib/ -classpath /usr/local/mysql/
share/mysql/java/clusterjpa.jar:/usr/local/openjpa/openjpa-1.2.1.jar:/usr/local/
openjpa/lib/*:/usr/local/connectorj/mysql-connector-java-5.1.12-bin.jar:. Main
```

Exercise

1. Move to the folder containing the ClusterJPA example:
   `cd /usr/local/clusterj_example_wp_7_1_2a`
2. Compile the software:
   ```
   javac -classpath /usr/local/mysql/share/mysql/java/clusterj-api.jar:.
   Main.java Employee.java
   ```
3. Run the software:
   ```
   java -classpath /usr/local/mysql/share/mysql/java/clusterj.jar:. -
   Djava.library.path=/usr/local/mysql/lib Main
   ```
4. Check the contents of the database at each opportunity

Schema considerations

Develop for MySQL Cluster

Going from one storage engine is relatively easy with MySQL, but it usually isn't the best way forward. If you go for example from MyISAM to InnoDB, you'll need to tune a completely different set of options to make run smoothly. The same is true when you alter tables to use the NDBCluster storage engine:

```
ALTER TABLE employees ENGINE=NDBCluster
```

One example where it can go wrong are with some of the restrictions and limitations found in MySQL Cluster. The following table compares a few differences between NDBCluster, InnoDB and MyISAM:

	NDBCluster	InnoDB	MyISAM
Columns per table	128	1000	4096
Row size	8052 bytes	±8kB*	64kB

(*Excluding variable-lenght columns)

If one of your tables would have 200 columns, it would not be possible to store it in MySQL Cluster.

> More on limitations within MySQL and MySQL Cluster in the MySQL Manual:
> http://dev.mysql.com/doc/refman/5.1/en/restrictions.html
> http://dev.mysql.com/doc/refman/5.1/en/mysql-cluster-limitations.html
> http://dev.mysql.com/doc/refman/5.1/en/innodb-restrictions.html
> http://dev.mysql.com/doc/refman/5.1/en/myisam-storage-engine.html

Re-normalization

When you have an existing project, you probably did normalize your data. If you want to get your project into MySQL Cluster, you'll have to check if your schema still fits.

The Row Size is a common pitfall. As we seen earlier, it is limited to 8052 bytes. Your InnoDB tables will not work.
To demonstrate, lets create a table with two rather big VARCHAR fields:

```
CREATE TABLE t1 (
 c1 INT NOT NULL KEY,
 c2 VARCHAR(4000),
 c3 VARCHAR(4000)
) ENGINE=NDBCluster DEFAULT CHARSET=Latin1
```

This works, as we keep under the 8052 bytes. Notice also that we use a 1-byte character set, latin1. If we would use UTF-8 as character set, it would fail:

```
CREATE TABLE t1 (
 c1 INT NOT NULL KEY,
 c2 VARCHAR(4000),
 c3 VARCHAR(4000)
) ENGINE=NDBCluster DEFAULT CHARSET=UTF-8
```


This fails with an error saying the maximum row size could be reached. In MySQL a UTF-8 character can take up to 3 bytes. If we do the calculation for only the c2 VARCHAR column, an application would be allowed to store data as big as 12000 bytes, or 4000 x 3.

When you have such tables, you'll probably need to redo some normalization steps. You might need to split up bigger table, putting data you rarely need in another table.

A solution for the table in the example above would be to use TEXT, but it is good to limit the usuage of LOB-fields as much as possible. We'll say why later.

Denormalization

Normalization is good, but with MySQL Cluster you might want to denormalize.

One reasons you would denormalize is limiting queries joining tables. Lets look at some typical (simplistic) normalized situation where we store the location of a subscriber in a separate table:

uid	username	fullname
1	andrewh	Andrew Hutchings
2	andrewm	Andrew Morgan
3	geertv	Geert Vanderkelen

uid	location
1	UK
2	UK
3	BE

To get the subscribers username and location of Geert Vanderkelen, you'll do the following join:

```
SELECT s.uid, l.location
FROM subscribers AS s LEFT JOIN locations AS l
WHERE s.uid = 3
```

This works fine with MySQL Cluster, but in a busy system and with more data involved, it might cause to much network traffic. When you always need the location for the subscriber, you better put the location back into the subscriber table:

uid	username	fullname	location
1	andrewh	Andrew Hutchings	UK
2	andrewm	Andrew Morgan	UK
3	geertv	Geert Vanderkelen	BE

This would allow us to make a much simpler query:

```
SELECT uid, location FROM subscribers WHERE uid = 3
```

Primary Keys and Unique Indexes

You should define a Primary Key for every MySQL Cluster table. If you don't, one will be created automatically, but that's considered bad practice.

Both the Primary Key and Unique Indexes will use data and index memory, unless you give the `USE HASH` option when you create the table. This way, only the hash part will be created and no data memory will be consumed. Updating data will be faster, less data memory will be used, but queries doing range scans might be slower.

Historical Data

Remove redundant even quicker within MySQL. Only the data that is hot when you have lots of changes and new records coming.

Disk-Based Tables

Tables can be created so non-indexed fields are going to disk instead of memory. This will allow you to store much more data in MySQL Cluster, but you have to also consider some possible performance penalty.
One of the reasons why MySQL Cluster is storing in memory is to compensate for the possible network latency. Hard drives could possibly add more latency.

You will need to create a 'logfile group', and 'tablespaces'. We are not covering this in the tutorial, but the MySQL Manual has a nice example will get you going. Note that you might need to configure your MySQL Cluster to get it actually working well.

> Some external links:
> http://dev.mysql.com/doc/refman/5.1/en/mysql-cluster-disk-data.html
> http://johanandersson.blogspot.com/2008/12/disk-data-summary.html

Scaling and Performance

MySQL Nodes

MySQL Cluster is not designed for the single thread / single query performance. In fact you can be almost certain that it will perform worse in this scenario than any other storage engine type. It is designed for multiple simultaneous queries.
By default a `mysqld` server only has a single connection to the cluster to process queries which can be a bottleneck. There are two ways around this. First, you could add more mysqld nodes and have your application load-balance between them.
Or alternatively you could use the `--ndb-cluster-connection-pool` setting to add more connections between mysqld and the cluster. Please note that to do this you will need to add more `[mysqld]` slots to your `my.cnf` to account for each connection in the pool.

NDBAPI

Of course, direct NDBAPI is going to give much better performance than using SQL.
Not only do you eliminate the SQL overhead but there are some things in NDBAPI which cannot be defined using SQL.

Data Nodes

MySQL Cluster was originally designed for primary key equality lookups, and it excels at this. When running an SQL query with this kind of lookup mysqld will try to go directly to the node with the data and the data will be retrieved very quickly.

Ordered index lookups are, in general, a lot slower. This is because the query has to be sent to all nodes to be processed simultaneous and then the results are collected and sent back. When adding more nodes there is more network traffic and round trips required for this. So performance can degrade with more nodes when ordered indexes are used.

Other Issues

Blobs

Blobs (text columns are blob columns too) in MySQL Cluster are stored using a hidden table.
Each blob is split up into many rows for this table (depending on the size of the blob). This not only causes performance problems but can cause locking problems too because MySQL needs to keep the blob table consistent with the main table.
Where possible a large `VARCHAR` or `VARBINARY` may be a better option.

Joins

At the moment joins require a lot of excessive overhead to process. The second table in the join is effectively queried for every matching row in the first table. This is a lot of network overhead.
There is a work-in-progress solution to this called pushed-down joins. This is where the joins are processed directly in the data nodes. This is currently an in-development feature with no scheduled release date (or any guarantee it will ever be released).

Online Add Node

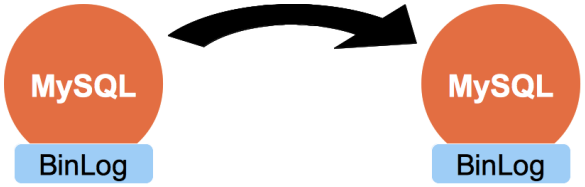
One of the newer features in MySQL Cluster is online add node. This lets you add data nodes to an running cluster without having to rebuild it from scratch.

To do this the following procedure must be followed:

1. Add the necessary `[ndbd]` sections to `config.ini`
2. Restart all management nodes with `--initial` or `--reload`
3. Restart all data nodes
4. Restart all MySQL or API nodes
5. Start the new data nodes with `--initial`
6. Run `CREATE NODEGROUP x,y` (where x and y are new nodes) to create a new data node group with the new nodes.
7. Run `ALTER TABLE … REORGANIZE PARTITION` in MySQL to reorganize the data into the new nodes
8. Run `OPTIMZE TABLE … in MySQL` to reclaim space after the reorganization

Geographical Replication

The MySQL Server comes with Replication built in. It allows you to duplicate your data from one machine to another for failover or backup purposes.



This section we will show how MySQL Cluster takes advantage of this feat.

First, lets look at the binary logging of the MySQL Server, which is a prerequisite for replication.

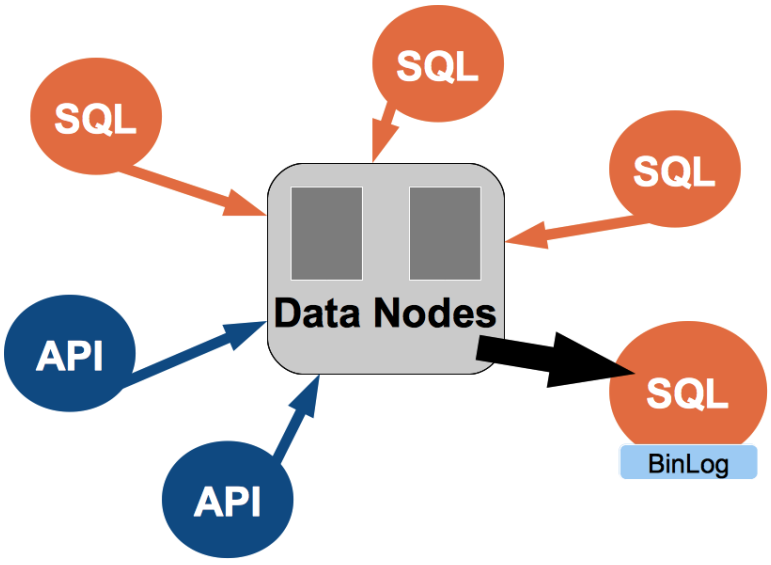
Binary Log Format

Before MySQL 5.1, the only available option for binary logging was Statement Based. This means that each successful SQL statement that changes data gets logged.
Another MySQL server, the Slave, will read these statements and apply them.

MySQL 5.1 brings 2 new formats in addition to Statement-Based binary logging: Row-Based and Mixed. The latter is a format where the storage engine will be able to choose whether it does Statement or Row-Based binary logging. Mixed is also default for MySQL Cluster installations. (Note that for the 'normal' MySQL 5.1 server, Statement-Based logging is the default.)

Row-Based binary logging was needed to allow replication of one cluster to another. The reason is that there can be multiple SQL and API nodes doing changes simultaneously.

Since API Nodes don't talk SQL, statements can't be logged. With Row-Based logging, the data that changed goes to the binary logs, from all SQL and API Nodes.



The above picture shows one MySQL server doing binary logging. It does does by listening for events through NDB Injector-Thread.
The above shown setup has one flaw, we'll see later what.

Enabling Binary Logging

You can enable the binary logging in any SQL Node. Here is an example from the configuration files used in this tutorial:

```
[mysqld]
ndbcluster
log_bin = master_A_binlog
server_id = 1
binlog_format = MIXED
```

The `--log-bin` option enables the binary logging into files named `master_A_binlog`. The `--binlog-format` is not really needed, but it's good to be explicit. Here is how the files look like:

```
master_A_binlog.000011
master_A_binlog.000012
master_A_binlog.index
```

Don't forget to set the `--server-id` option to a value unique to each SQL Node participating in the MySQL Cluster. If you don't do this, replication will not work.

You can check the content of a binary log using the `mysqlbinlog` tool. Reading Row-Based or Mixed binary logs might be a bit difficult. Take for example the following table:

```
CREATE TABLE attendees (
id INT UNSIGNED NOT NULL AUTO_INCREMENT,
 email VARCHAR(200),
PRIMARY KEY (id)
) ENGINE=NDBCluster;
```

Lets insert some data:

```
INSERT INTO attendees (email) VALUES ('geert@localhost')
```

Reading the binary log, you don't get the SQL statement:

```
shell> mysqlbinlog master_A_binlog.00001
..
BINLOG '
Xz/CSxMBAAAAMwAAAO8DAAAAACUAAAAAAAEABHRlc3QACWF0dGVuZGVlcwACAw8CyAAC
Xz/CSxMBAAAAPgAAAC0EAAAAABAAAAAAAAEABW15c3FsABBuZGJfYXBwbHlfc3RhdHVzAAUDCA8I
CAL/AAA=
Xz/CSxcBAAAAOwAAAGgEAAAAABAAAAAAAAAAAR/gAQAAAA8AAACiAAAAAAAAAAAAAAAAAAAAAAAA
AAA=
Xz/CSxcBAAAAMgAAAJoEAAAQACUAAAAAAAEAAgP8AQAAAA9nZWVydEBsb2NhbGhvc3Q=
'/*!*/;
..
```


You can, however, have a more readable output using the --verbose option:

```
shell> mysqlbinlog -vv master_A_binlog.00001
BINLOG '
Xz/CSxMBAAAAMwAAAO8DAAAAACUAAAAAAAEABHRlc3QACWF0dGVuZGVlcwACAw8CyAAC
Xz/CSxMBAAAAPgAAAC0EAAAAABAAAAAAAAEABW15c3FsABBuZGJfYXBwbHlfc3RhdHVzAAUDCA8I
CAL/AAA=
Xz/CSxcBAAAAOwAAAGgEAAAABAAAAAAAAAABR/gAQAAAA8AAACiAAAAAAAAAAAAAAAAAAAAAAAAAA
AAA=
### INSERT INTO mysql.ndb_apply_status
### SET
###   @1=1 /* INT meta=0 nullable=0 is_null=0 */
###   @2=695784701967 /* LONGINT meta=0 nullable=0 is_null=0 */
###   @3='' /* VARSTRING(255) meta=255 nullable=0 is_null=0 */
###   @4=0 /* LONGINT meta=0 nullable=0 is_null=0 */
###   @5=0 /* LONGINT meta=0 nullable=0 is_null=0 */
Xz/CSxcBAAAAMgAAAJoEAAAQACUAAAAAAAEAAgP8AQAAAA9nZWVydEBsb2NhbGhvc3Q=
'/*!*/;
### INSERT INTO test.attendees
### SET
###   @1=1 /* INT meta=0 nullable=0 is_null=0 */
###   @2='geert@localhost' /* VARSTRING(200) meta=200 nullable=1 is_null=0 */
```

The `mysqlbinlog` tool will try to make a more readable SQL statement out of the data that was logged.

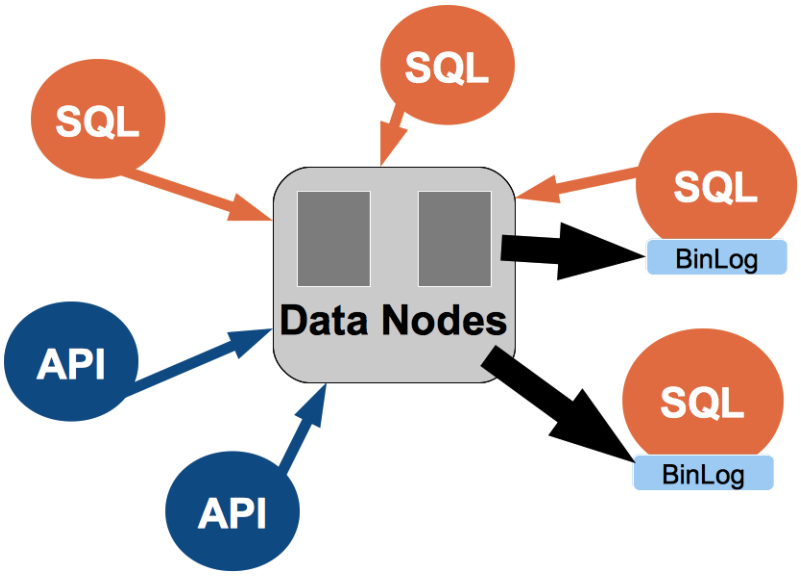
The LOST_EVENT incident & solution

We saw previously a setup which had one MySQL Server (or SQL Node) doing binary logging. This is a problem. Imagine that the machine where this node writing the binlogs is suddenly failing. All other SQL and API nodes still continue to do changes, but there is no MySQL Server able record them.

When a MySQL Server starts and it finds out it did do binary logging earlier, it will report the LOST_EVENTS incident in the binary logs. Slaves reading from this binary log will stop replicating and show an error. They will report that the Master was down and might miss data. If a Slave would continue, your data will probably be inconsistent. You do not want that.

```
shell> mysqlbinlog master_A_binlog.00001
..
#020411 22:48:47 server id 1  end_log_pos 143
# Incident: LOST_EVENTS
RELOAD DATABASE; # Shall generate syntax error
..
```


Luckily, the solution is rather simple. Just configure another SQL Node to do binary logging.



Setting up Replication between Clusters

Setting up Geographical Replication between two Cluster is as simple as setting it up between two 'normal' MySQL servers.

We'll need to start a second Cluster for this.

```
shell> ndb_mgmd --initial -f /opt/mysqlcluster/config_Slave.ini --configdir=/
opt/mysqlcluster/ndb_Slave
shell> ndbd -c localhost:1187
shell> mysqld_safe --defaults-file=/opt/mysqlcluster/my_Slave.cnf &
```

To access the Slave MySQL Cluster using the `ndb_mgm` and `mysql` tool, use the following:

```
shell> ndb_mgm -c localhost:1187
shell> mysql -S /tmp/mysql_Slave
```

On the Master Cluster you create a MySQL user which has the `REPLICATION USER` privilege. You have to do this on every SQL Node which you are doing binary logging:

```
shell> mysql
mysql_A> GRANT REPLICATION SLAVE ON *.* TO repl@127.0.0.1;

shell> mysql -S /tmp/mysql_B
mysql_B> GRANT REPLICATION SLAVE ON *.* TO repl@127.0.0.1;
```

The reason you need to do it on both is because the grant tables in MySQL are local to each node. They are MyISAM and are not part of Cluster.

Now we need to the binary log position from which we are going to start the replication. We will use the SQL Node A to start with.


```
shell> mysql
mysql_A:(none)> SHOW MASTER STATUS;
+------------------------+----------+-..
| File                   | Position | ..
+------------------------+----------+-..
| master_A_binlog.000012 |      214 | ..
+------------------------+----------+-..
```

Using this information we can go to the Slave and issue the `CHANGE MASTER` command.

```
mysql_Slave> CHANGE MASTER TO
      MASTER_HOST='127.0.0.1',MASTER_PORT=3306,
      MASTER_USER='repl',
      MASTER_LOG_FILE='master_A_binlog.000012',
      MASTER_LOG_POS=214;
```

Then start the slave, and use `SHOW SLAVE STATUS` to see if it is working:

```
mysql_Slave> START SLAVE;
mysql_Slave> SHOW SLAVE STATUS\G
..
               Slave_IO_State: Waiting for master to send event
..
             Slave_IO_Running: Yes
            Slave_SQL_Running: Yes
..
             Master_Server_Id: 1
```

You create the table on the Master side, using the NDBCluster engine. It should get created on the Slave as well:

```
mysql_A> CREATE TABLE attendees (
         id INT UNSIGNED NOT NULL AUTO_INCREMENT,
         email VARCHAR(200),
         PRIMARY KEY (id)
         ) ENGINE=NDBCluster;
```

Now you can simply insert data and see it replicating.

Handling LOST_EVENTS

Lets see what happens when we shutdown the mysql_A server and start again.

```
shell> mysqladmin -S /tmp/mysql shutdown
shell> mysqld_safe --defaults-file=/opt/mysqlcluster/my_A.cnf &
```

Now, check the Slave:

```
shell> mysql -S /tmp/mysql_Slave
mysql_Slave> SHOW SLAVE STATUS\G
..
           Last_Errno: 1590
           Last_Error: The incident LOST_EVENTS occured on the master.
                       Message: mysqld startup
..
```

It is now up to you do decide whether you can continue skipping the error, knowing that you might have lost data or switch to the other replication channel.
Skipping can be done doing this:


```
mysql_Slave> SET GLOBAL SQL_SLAVE_SKIP_COUNTER = 1;
mysql_Slave> START SLAVE;
```

Exercise: Switching Replication Channel

Switching replication channels is needed when the MySQL Server from which the Slave is reading changes, went down. It is also useful when doing a rolling upgrade of Cluster.

For the tutorial we will be starting a second MySQL Cluster which will be reading changes from the first (Master) MySQL Cluster. We will be switching the MySQL Server from which the Slave Cluster is reading from.

Starting Master and Slave Cluster

Start the Master MySQL Cluster. We will start it empty for this exercise:

```
shell> cd /opt/mysqlcluster
shell> ndb_mgmd -f config.ini
shell> ndbd -c localhost:1186 --initial
shell> ndbd -c localhost:1186 --initial
shell> mysqld_safe --defaults-file=my_A.cnf &
shell> mysqld_safe --defaults-file=my_B.cnf &
```

Lets start the Slave MySQL Cluster, note that it has only 1 data node:

```
shell> cd /opt/mysqlcluster
shell> ndb_mgmd -f config_Slave.ini --configdir=/opt/mysqlcluster/ndb_Slave
shell> ndbd -c localhost:1187
shell> mysqld_safe --defaults-file=my_Slave.cnf &
```

You can check the Slave Cluster and connect to the MySQL Server using following:

```
shell> ndb_mgm -c localhost:1187
shell> mysql -S /tmp/mysql_Slave
```

Setting up the Replication Users on the Master

We have to create the replication users on both MySQL Servers. The grant tables are local to each MySQL Server and are not stored in the Data Nodes:

```
shell> mysql
mysql_A> GRANT REPLICATION SLAVE ON *.* TO repl@127.0.0.1;
shell> mysql -S /tmp/mysql_B
mysql_B> GRANT REPLICATION SLAVE ON *.* TO repl@127.0.0.1;
```

Starting Replication

To start the replication we will need the actual position of the binary log. Initially the Slave will be reading from the first MySQL Server of the master, mysql_A:

```
shell> mysql
mysql_A:(none)> SHOW MASTER STATUS;
+------------------------+----------+-..
| File                   | Position | ..
+------------------------+----------+-..
| master_A_binlog.000012 |      214 | ..
+------------------------+----------+-..
```


On the Slave we now issue the CHANGE MASTER TO command using the information we got from the Master MySQL Server above:

```
mysql_Slave> CHANGE MASTER TO
MASTER_HOST='127.0.0.1',MASTER_PORT=3306,
MASTER_USER='repl',
MASTER_LOG_FILE='master_A_binlog.000012',
MASTER_LOG_POS=214;
```

Start the Slave and check the status for any errors. The IO_Thread and SQL_Thread should be running, saying 'Yes':

```
mysql_Slave> START SLAVE;
mysql_Slave> SHOW SLAVE STATUS\G
```

Creating table, inserting data, checking:

```
mysql_A> CREATE TABLE attendees (
id INT UNSIGNED NOT NULL AUTO_INCREMENT,
email VARCHAR(200),
PRIMARY KEY (id)
) ENGINE=NDBCluster;

mysql_A> INSERT INTO attendees (email) VALUES ('geertv@localhost'),
('andrewh@localhost'),('andrewm@localhost');

Check the data in the table attendees on mysql_A and compare it with the content
of using mysql_Slave.
```

Shut down MySQL from Master Cluster

For this exercise we just shutdown the mysql_A MySQL Server. This means that the Slave will not be able to read from the Master anymore:

```
shell> mysqladmin -S /tmp/mysql shutdown
```

We are not starting it again, but if you would, you'll notice that the Slave side will have a LOST_EVENTS error.

After the MySQL Server mysql_A is down, you can insert some data using mysql_B. This is a control. The Slave will not get it, but will when we switched it.

```
mysql_B> insert into attendees (email) values ('marta@localhost');

mysql_Slave> SELECT * FROM attendees;
+----+-------------------+
| id | email             |
+----+-------------------+
|  3 | andrewm@localhost |
|  1 | geertv@localhost  |
|  2 | andrewh@localhost |
+----+-------------------+
```


The Switch

We now have to tell the Slave MySQL Server it has to use the other MySQL server of the Master to read changes from. We have to find out what was the last 'epoch' or event from the Master that it applied. You do this with the following query:

```
mysql_Slave> SELECT MAX(epoch) FROM mysql.ndb_apply_status;
+---------------+
| MAX(epoch)    |
+---------------+
| 1915555414023 |
+---------------+

mysql_B> SELECT
 SUBSTRING_INDEX(File, '/', -1) AS BinLog,
 Position
 FROM mysql.ndb_binlog_index
 WHERE epoch >= 1915555414023
 ORDER BY epoch ASC LIMIT 1;
+-----------------------+----------+
| BinLog                | Position |
+-----------------------+----------+
| master_B_binlog.000006 |      405 |
+-----------------------+----------+
```

Back to the Slave, we're using the `CHANGE MASTER TO` command to point it to the mysql_B server:

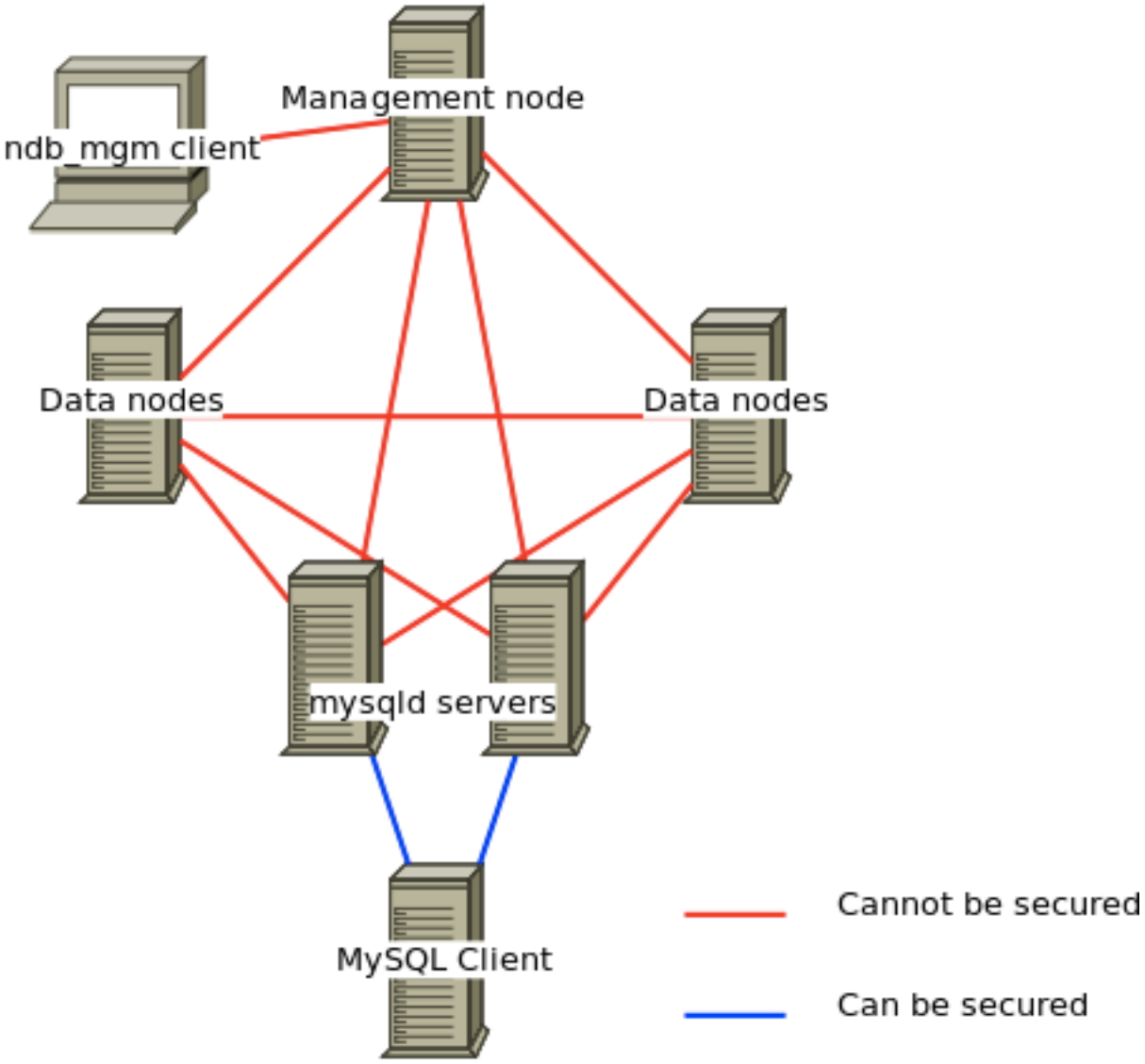
```
mysql_Slave> STOP SLAVE;
mysql_Slave> CHANGE MASTER TO
       MASTER_LOG_FILE = 'master_B_binlog.000006',
       MASTER_LOG_POS = 405,
       MASTER_HOST='127.0.0.1',
       MASTER_PORT=3307;
```

You should now be able to start the Slave and previously done changes should get replicated:

```
mysql_Slave> START SLAVE;
mysql_Slave> SELECT * FROM attendees;
+----+------------------+
| id | email            |
+----+------------------+
|  3 | andrewm@localhost |
|  1 | geertv@localhost  |
|  2 | andrewh@localhost |
|  4 | marta@localhost   |
+----+------------------+
```


Security

It should be noted right from the start that MySQL Cluster nodes have no security in their communications. This is because the overhead required would cause a very large performance hit. So all nodes should be on a private LAN. The added advantage is having the cluster on a private LAN can stop any other network chatter from delaying node communications.



MySQL Authentication

The privileges system used by MySQL is stored using MyISAM tables. Due to some internal restrictions changing these to the NDB engine will cause many complications, therefore all privileges should be synchronized externally (using a script for instance).

