New Features in MySQL Cluster 5.1

A MySQL® Technical White Paper

August 2006

Copyright © 2006, MySQL AB

Table of Contents

Introduction ... 3
New Feature Overview ... 3
MySQL Cluster Architecture Overview ... 3
Disk Data ... 6
Creating a Memory-Based Table ... 6
Creating a LOGFILE GROUP and TABLESPACE ... 7
Creating a Disk-Based Table ... 7
Disk Data Administration and Maintenance ... 8
Migrating Memory-Based Tables to Disk ... 9
Parameters ... 10
Additional Notes on Disk Data ... 10
Support for Replication ... 11
Replication Channel ... 13
Replication Schema and Tables ... 14
Replication Setup ... 15
Starting Replication ... 16
Additional Notes on MySQL Cluster Replication ... 17
Faster ADD/DROP INDEX ... 17
Efficient Variable-Sized Records ... 19
Conclusion ... 20
About MySQL ... 21
Resources ... 21
White Papers ... 21
Case Studies ... 21
Press Releases, News and Events ... 21
Live Webinars ... 21
Webinars on Demand ... 22

Copyright © 2006, MySQL AB

Introduction

MySQL Cluster is a scalable, high-performance, clustered database, originally developed for some of the world's most demanding applications found in the telecommunications industry. Often these telecom applications required that the database's availability approach 99.999%. Since MySQL Cluster's introduction in 2004, new feature sets have been steadily introduced into the product. This has broadened its appeal for new application use-cases, markets and industries. MySQL Cluster is now being adopted not only as a database for traditional telecom applications like, HLR (Home Locator Registry) or SLR (Subscriber Locator Registry), it is now also being widely deployed for VOIP, internet billing, session management, eCommerce sites, search engines and even traditional back office applications. In this paper we will explore the new features which have been introduced in MySQL Cluster 5.1.

New Feature Overview

With the release of 5.1, many new features and improvements have been introduced to the already popular MySQL Cluster. These include:

- Support for Disk Data
- Row-based Replication
- Higher performance ADD/DROP INDEX
- More efficient Variable Sized Records

MySQL Cluster Architecture Overview

Before embarking on a technical examination of MySQL Cluster's new features, it makes sense to quickly review the product's architecture and how it functions.

MySQL Cluster is a high availability database built using a unique shared nothing architecture and a standard SQL interface. The system consists of multiple nodes that can be distributed across machines and regions to ensure continuous availability in the event of node or network failure. MySQL Cluster uses a storage engine, consisting of a set of data nodes to store data which can be accessed using standard SQL with MySQL Server.

MySQL Cluster tolerates failures of several data nodes and reconfigures itself on the fly to mask out these failures. The self-healing capabilities, as well as the transparency of data distribution and partitioning from the application, result in a simple programming model that enables database developers to easily include high availability in their applications without complex low-level coding.

MySQL Cluster consists of three kinds of nodes:

1. ***Data Nodes*** store all the data belonging to the MySQL Cluster. Data is replicated between these nodes to ensure it is continuously available in the event one or more nodes fail. These nodes also manage database transactions. Increasing the number of *replicas* yields additional data redundancy.

2. ***Management Server Nodes*** handle system configuration at startup and are leveraged when there is a change to the cluster. Usually only one Management Node is configured, however there is the possibility to run additional nodes in order to remove this single point of failure. Because the

Management Node is used only at startup and system re-configuration, the cluster will remain online and available regardless of the Management Node's status.

3. ***MySQL Server Nodes*** enable access to the clustered Data Nodes. This provides developers a standard SQL interface to program against. MySQL Server in turn, handles sending requests to the Data Nodes, thus eliminating the need for cluster specific, low-level programming within the application. Additional MySQL Server Nodes are typically added in order to increase performance. This arrangement naturally lends itself to an implementation of the Scale-out methodology for increasing scalability, capacity and performance.

Below in Figure 1 is an illustration of a basic MySQL Cluster configuration consisting of:

- One MySQL Server Node
- One Management Server Node
- Four Data Nodes (forming two Data Node Groups)

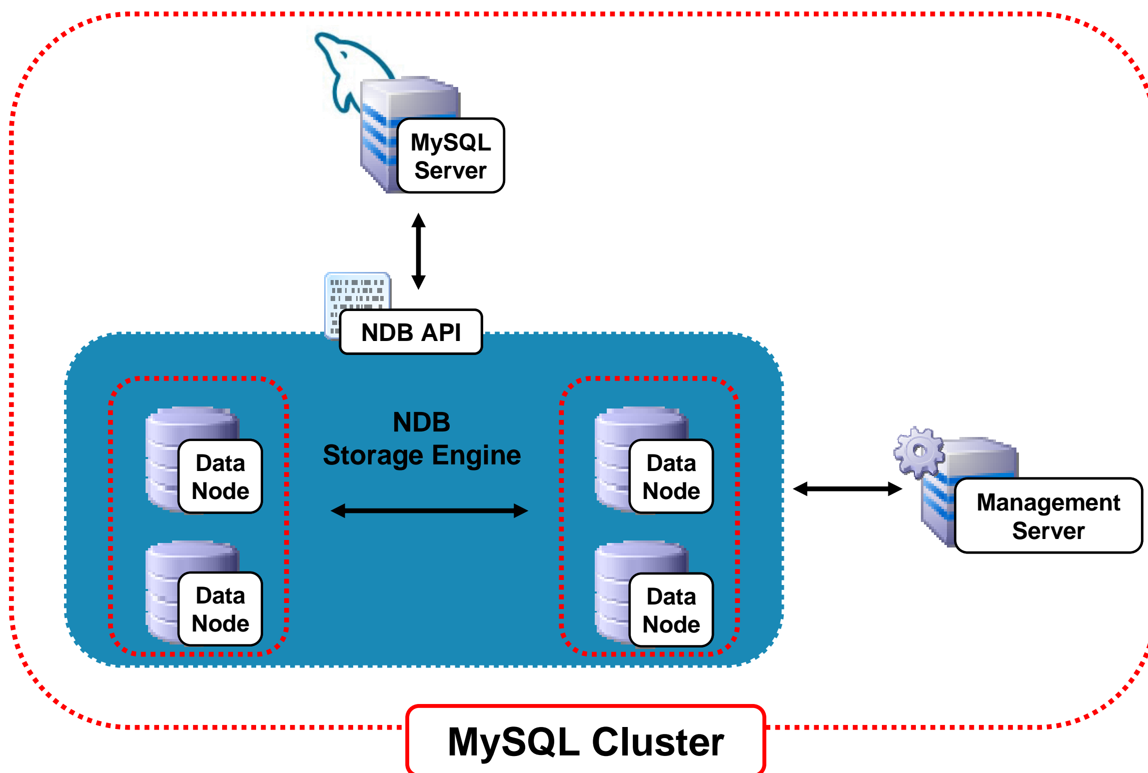

Figure 1

In Figure 2 we illustrate a MySQL Cluster configuration in which we have employed Scale-out in order to increase performance and capacity. We have done so by adding two additional MySQL Server Nodes, as well as, an additional Management Server for process redundancy. This configuration now consists of:

- Three MySQL Server Nodes
- Two Management Server Nodes
- Four Data Nodes (forming two Data Node Groups)

Copyright © 2006, MySQL AB

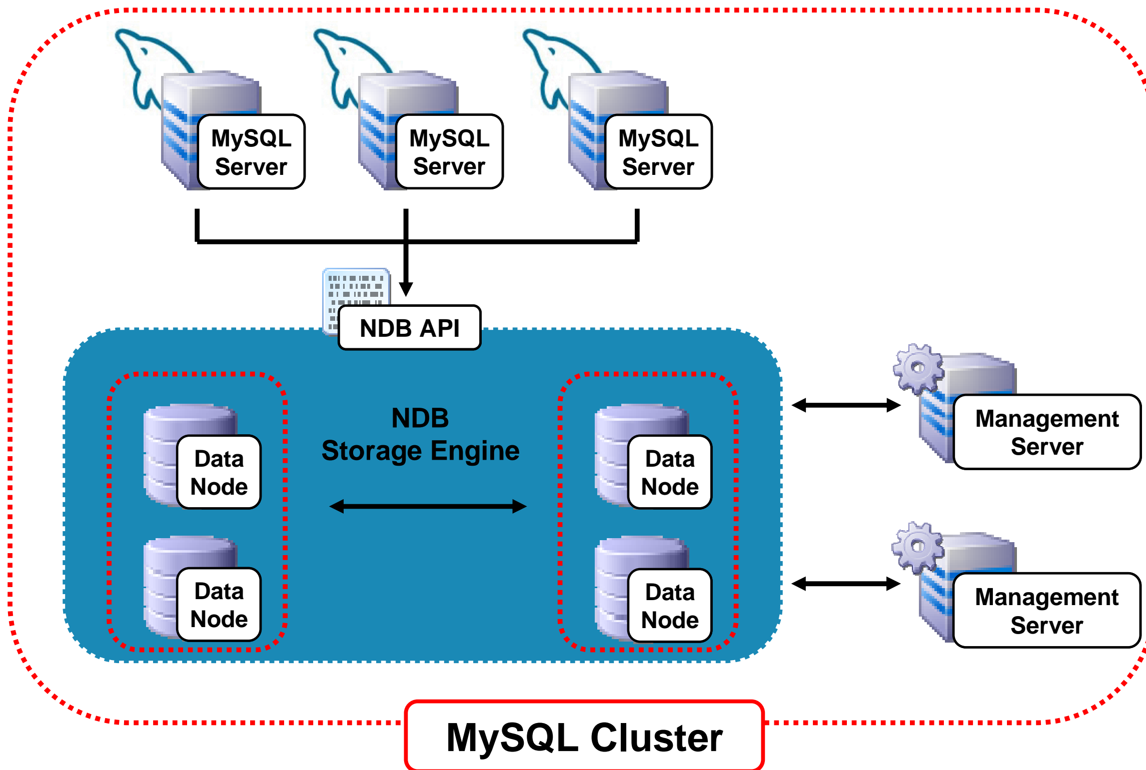

Figure 2

A MySQL Server in a MySQL Cluster is connected to all Data Nodes and there may be multiple MySQL servers in the same MySQL Cluster as illustrated in Figure 2. All transactions executed on the MySQL servers are handled by the common set of Data Nodes. This means that as soon as a transaction has been executed on one MySQL server, the result is visible through all MySQL servers connected to the MySQL Cluster.

The node-based architecture of MySQL Cluster has been carefully designed for high availability:

- If a Data Node fails, then the MySQL Server can easily use any other Data Node to execute transactions.
- The data within a Data Node is replicated on multiple nodes. If a Data Node fails, then there is always at least one other Data Node storing the same information.
- Management Server Nodes can be killed and restarted without affecting the ongoing execution of the data nodes. As mentioned previously, the Management Server Node is only leveraged at startup and when there is a reconfiguration of the cluster.

Designing the system in this way makes the system reliable and highly available since single points of failure have been minimized. Any node can be killed without it affecting the system as a whole. An application can, for example, continue executing even though a Data Node is down. Other techniques are also used to increase the reliability and availability of the database system, including:

- Data is synchronously replicated between all data nodes. This leads to very low fail-over times in case of node failures.
- Nodes execute on multiple hosts, allowing MySQL Cluster to operate even during hardware failures.

Copyright © 2006, MySQL AB

- Nodes are designed using a shared-nothing architecture. Each Data Node has its own disk and memory storage.
- Single points of failure have been minimized. Any node can be killed without any loss of data and without stopping applications using the database.

Applications connecting to the MySQL Cluster using a MySQL Server gain the following benefits:

- Data independence, meaning that the application can be written without requiring knowledge of the physical storage of the data. Data is stored in a storage engine which handles all of the low level details such as data replication and automatic fail over.
- Network and distribution transparency, which means that the application program depends on neither the operational details of the network, nor the distribution of the data on the data nodes,
- Replication and partition transparency, meaning that applications can be written in the same way regardless of whether the data is replicated or not, and independent of how the data is partitioned.
- A standard SQL interface that is easy for developers and DBAs to use without requiring any low level programming to achieve high availability.

These features make it possible for MySQL Cluster to dynamically reconfigure itself in case of failures without the need for custom coding in the application program.

For more information and an in-depth overview of MySQL Cluster's architecture and how it works, please refer to the *MySQL Cluster Architecture Overview White Paper*, available at: http://www.mysql.com/why-mysql/white-papers/cluster-technical.php .

Disk Data

One of the most anticipated features of MySQL Cluster 5.1 is the support for disk-based data. For those already familiar with MySQL Cluster, you know that the NDB storage engine – which powers the actual MySQL Cluster - was previously a 100% in-memory database engine. While this was excellent for smaller databases that could be held in RAM, support for disk data now allows MySQL Cluster to expand its database footprint, giving you the ability to create even larger database clusters and still manage them effectively.

Data that must be highly available, but does not necessarily demand the very high performance characteristics of RAM-based data, will be the best candidates for leveraging disk-data. Also, for those who have run into "storage ceilings" because of the hard limits imposed by the operating system or hardware, you can now look at migrating the in-memory data to disk or developing your new applications with MySQL Cluster's data on disk support in mind.

In this section we'll take a look at creating, migrating and maintaining disk-based data tables for MySQL Cluster. Please note that support for disk-based data was formally introduced in version 5.1.6 Beta.

Creating a Memory-Based Table

Ordinarily, when creating an NDB table one simply issued a statement like the one below and the table and indexes were created in main memory and ready for use.

```
CREATE TABLE table1 (col1 int, col2 int, col3 int, col4 int,
```

Copyright © 2006, MySQL AB

```
PRIMARY KEY(col1), index(col1,col2))
ENGINE=ndb;
```

The properties of this table are as follows:

- Columns ‘col1’ and ‘col2’ reside in memory because they are part of an index. (This remains a restriction in MySQL 5.1 even when using disk-based data, as NDB indexes still must fully reside in RAM.)
- Columns ‘col3’ and ‘col4’ reside in memory as well, however we can change this in 5.1 to disk, by defining a ‘tablespace’ within MySQL Cluster and assigning these table columns to it.

Creating a LOGFILE GROUP and TABLESPACE

As mentioned, to take advantage of MySQL Cluster’s new data on disk functionality, we must first perform a few preliminary steps. These steps include creating a `LOGFILE GROUP` and a `TABLESPACE`.

The creation of a `LOGFILE GROUP` creates a file on each data node for storing `UNDO` logs. (Please note in version 5.1, only one `LOGFILE GROUP` is supported, theoretically an unlimited amount of `UNDOFILE`’s are supported however. The default value for the `UNDO` buffer is 8 MB.)

```
CREATE LOGFILE GROUP logfg1
ADD UNDOFILE 'undofile1.dat'
INITIAL_SIZE 16M
UNDO_BUFFER_SIZE = 1M
ENGINE=ndb;
```

Next we’ll create a file on each data node for storing partitions of the disk-based table. Note that tables we create in a `TABLESPACE` are associated with a `LOGFILE GROUP`.

```
CREATE TABLESPACE tsp1
ADD DATAFILE 'datafile1.dat'
USE LOGFILE GROUP logfg1
INITIAL_SIZE 12M
ENGINE=ndb;
```

Creating a Disk-Based Table

Now that we’ve created our `LOGFILE GROUP` and `TABLESPACE`, we are ready to create our first disk-based table.

```
CREATE TABLE table2 (pk_col1 int, fname_col2 VARCHAR(24), lname_col3
VARCHAR(255), ssno_col4 int ,
PRIMARY KEY (pk_col1), INDEX (pk_col1, ssno_col4))
TABLESPACE tsp1 STORAGE DISK
ENGINE=ndb;
```

The properties of this table are as follows:

- Columns ‘pk_col1’ and ‘ssno_col4’ reside in memory because they are part of an index. As was already mentioned, this is due to the fact that in version 5.1 indexes must reside in memory.

- Columns ‘fname_col2’ and ‘lname_col3’ will reside on disk as they are not part of an index.

Disk Data Administration and Maintenance

If at a later point it is necessary to add an additional `DATAFILE` to the `TABLESPACE`, we can use the following command:

```
ALTER TABLESPACE tsp1
ADD DATAFILE 'datafile2.dat'
INITIAL_SIZE 16M
ENGINE=ndb;
```

Adding an additional `UNDOFILE` to the `LOGFILE GROUP` is done in the following manner:

```
ALTER LOGFILE GROUP logfg1
ADD UNDOFILE 'undofile2.dat'
INITIAL_SIZE 16M
ENGINE=ndb;
```

You may also like to add an `INDEX` to a table after you have defined it on disk, for example:

```
CREATE TABLE table3 (col1 int, col2 int, col3 int, col4 int,
PRIMARY KEY (col1))
TABLESPACE tsp1 STORAGE DISK
ENGINE=ndb;

ALTER TABLE table3
ADD INDEX col1and2_index (col1, col2),
ENGINE=ndb;
```

This will create the appropriate index and move the dependent columns into memory, but leave the other columns on disk.

Something to note about a `TABLESPACE` and `LOGFILE GROUP`, is that the objects which reside within them must be dropped before the actual `TABLESPACE` and `LOGFILE GROUP` can be dropped. Not doing so will produce results similar to the following:

```
DROP TABLESPACE tsp1
ENGINE=ndb;
ERROR 1507 (HY000): Failed to drop TABLESPACE

DROP LOGFILE GROUP logfg1
ENGINE=ndb;
ERROR 1507 (HY000): Failed to drop LOGFILE GROUP
```

Managing disk-data structures can be summarized at a glance:

- A `LOGFILE GROUP` cannot be dropped if there is a dependent `TABLESPACE`.
- A `TABLESPACE` cannot be dropped if there is a dependent `DATAFILE`.
- A `DATAFILE` cannot be dropped from a `TABLESPACE` if there are dependent tables in the `TABLESPACE`.

So, to begin our maintenance operations we will first we’ll drop the tables:

Copyright © 2006, MySQL AB

```
DROP TABLE table2;
DROP TABLE table3;
```

Next, we drop the `DATAFILE`s associated with the `TABLESPACE`:

```
ALTER TABLESPACE tsp1
DROP DATAFILE 'datafile1.dat'
ENGINE=ndb;

ALTER TABLESPACE tsp1
DROP DATAFILE 'datafile2.dat'
ENGINE=ndb;
```

Then we drop the `TABLESPACE` and `LOGFILE GROUP`:

```
DROP TABLESPACE tsp1
ENGINE=ndb;

DROP LOGFILE GROUP logfg1
ENGINE=ndb;
```

Migrating Memory-Based Tables to Disk

Now let's take a look at how to migrate an existing memory-based table to a disk-based table. First, we'll create a sample in-memory table:

```
CREATE TABLE table4 (col1 int(5), col2 int(5), col3 int(5), col4
int(5),
PRIMARY KEY(col1), INDEX(col1,col2))
ENGINE=ndb;
```

The properties of this table as follows:

- Columns 'col1' and 'col2' reside in memory because they are part of an index.
- Columns 'col3' and 'col4' reside in memory as well, because we have not defined a `TABLESPACE` assignment for them.

As previously noted, disk data requires a `TABLESPACE` and `LOGFILE GROUP`, so we'll define those next:

```
CREATE LOGFILE GROUP logfg1
ADD UNDOFILE 'undofile1.dat'
INITIAL_SIZE 16M
UNDO_BUFFER_SIZE = 1M
ENGINE=ndb;

CREATE TABLESPACE tsp1
ADD DATAFILE 'datafile1.dat'
USE LOGFILE GROUP logfg1
INITIAL_SIZE 12M
ENGINE ndb;
```

Copyright © 2006, MySQL AB

Now, we'll issue the following ALTER command to migrate the table disk using the previously defined `TABLESPACE`.

```
ALTER TABLE table4
TABLESPACE tsp1 STORAGE DISK
ENGINE=ndb;
```

The properties of this table are now as follows:

- Columns 'col1' and 'col2' reside in memory because they are part of an index.
- Columns 'col3' and 'col4' will reside on disk as they are not part of an index.

Parameters

Below are a few parameters you should add to the *[NDBD]* section of your cluster's config.ini file when working with disk-based data.

- `DiskPageBufferMemory= X`

This parameter specifies the number of bytes used for the "caching" of disk pages. Each page is allocated 32k.

- `SharedPoolMemory= X`

This parameter specifies the number of bytes that can be used by various resources. Currently, the following resources using SharedPoolMemory include:

- TABLESPACES
- LOGFILE GROUPS
- DATAFILES
- UNDO FILES
- EXTENT INFO
- LOG BUFFER WAITER
- LOG SYNC WAITER
- PAGE REQUEST
- UNDO BUFFER

Additional Notes on Disk Data

- Indexes on disk are not supported at this time.
- Variable size attributes consume fixed space in the `DATAFILE` and page buffer cache.
- Extents are not released to a `TABLESPACE` when all the pages in the extent are freed. For example when you make random insert or delete, the space allocated to the table (from the `TABLESPACE`) will be maximum space used. It should be noted that this limitation also affects memory-based data.
- Relative to the potential for the fragmentation of disk-based data, there is currently no support for `OPTIMIZE TABLE` as it exists for MyISAM, InnoDB, and the BDB storage engines respectively.
- Only one `LOGFILE` group can be specified, this places a limit on the ability to split the load over different tables.
- `AUTO EXTEND` is currently not supported.

Copyright © 2006, MySQL AB

- You must include an `ENGINE` clause in any `CREATE LOGFILE GROUP`, `ALTER LOGFILE GROUP`, `CREATE TABLESPACE` or `ALTER TABLESPACE` statements. The accepted values for `ENGINE` are `NDB` and `NDBCLUSTER`.

Support for Replication

MySQL Server 5.1 introduces support for row-based replication. Previously, MySQL Cluster was unable to take advantage of the default statement-based replication available in versions 4.1 and 5.0. Now, with support for row-based replication, there is the ability to replicate from a MySQL Cluster to another MySQL Cluster or to other MySQL Servers. It is possible to create the following master/slave configurations:

- MySQL Cluster to MySQL Cluster
- MySQL Server (MyISAM, InnoDB, etc) to MySQL Cluster
- MySQL Cluster to MySQL Server

Obviously, if the goal is to achieve the highest possible availability, a Cluster to Cluster replication configuration with redundant replication channels, will be the ideal.

In all the scenarios where MySQL Replication has been traditionally leveraged, MySQL Cluster can now be as well. Some popular reasons for implementing replication include:

- Enabling Scale-out for capacity and performance
- A replicated database for fail-over
- Increasing read and write scalability
- A replicated database for use in maintenance operations such as, upgrades, testing or backups
- Replication to achieve higher availability within the data center or across a geographic WAN

First, let's review some basics about MySQL replication regardless if you use MySQL Cluster or not. Replication includes a ***master*** server and a ***slave*** server, the master being the source of the operations and data to be replicated and the slave being the recipient of these. This configuration can be illustrated as shown in Figure 3 below:

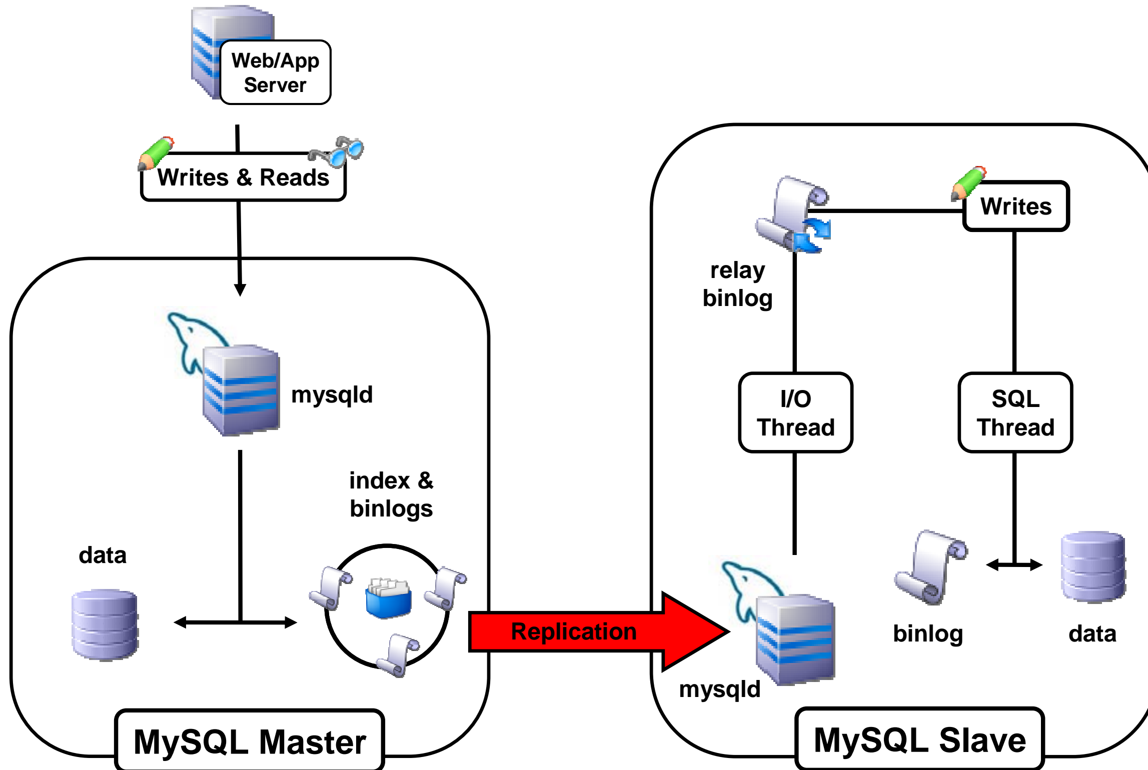

Figure 3

Although the implementation of replication within MySQL Cluster is architecturally similar, there are a few differences which should be explained. For example, a portion of MySQL Cluster's internals depend on the NDB storage engine, *which does not explicitly support standard SQL*. A MySQL Cluster to MySQL Cluster replication configuration is illustrated below in Figure 4:

Copyright © 2006, MySQL AB

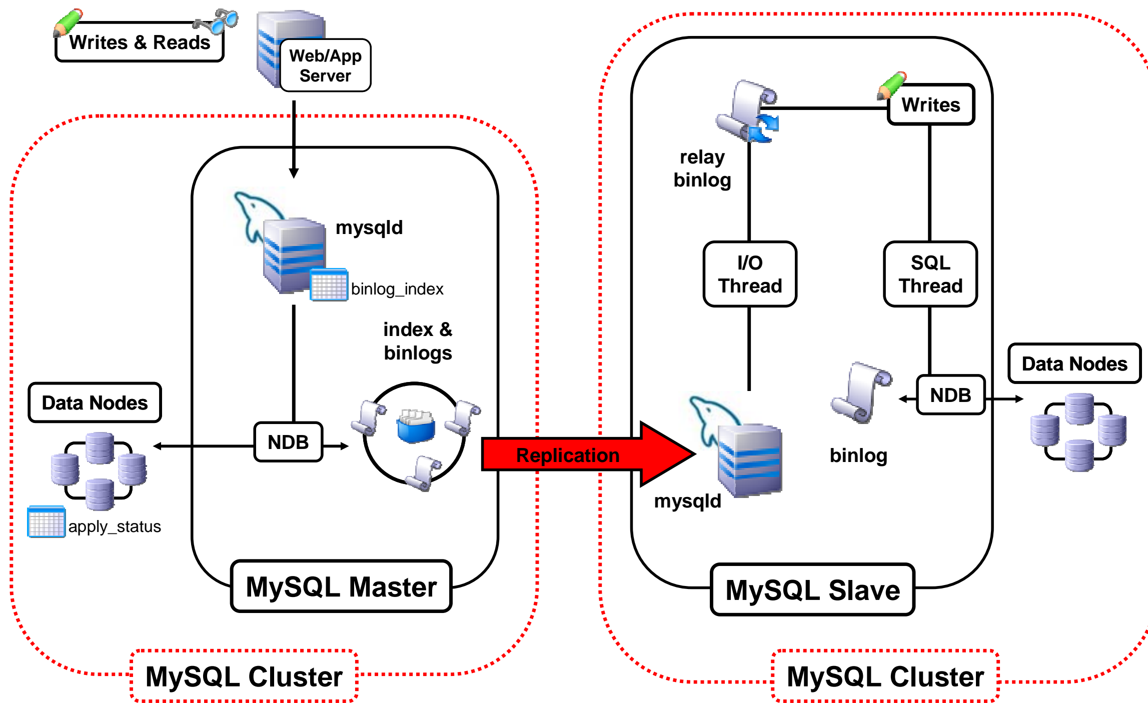

Figure 4

In the above configuration, the replication process is one in which consecutive states of a master cluster are logged and saved to a slave cluster. This process is achieved by a thread known as the *NDB binlog injector thread*, which runs on each MySQL server and creates a binary log (also known as a binlog). The NDB binlog injector thread guarantees that all changes in the cluster producing the binary log — and not just those changes that are effected via the MySQL Server — are inserted into the binary log in the correct order. This is important because the NDB storage engine supports the NDB API which allows you to write applications that interface directly with the NDB kernel, bypassing the MySQL Server and normal SQL syntax.

It also bears mentioning that the data flow or line of communication between a master and a slave is often referred to as the "replication channel". The replication channel is illustrated above in Figure 4 with a red arrow.

Replication Channel

A replication channel requires two MySQL servers acting as replication servers (one master and one slave).

Each MySQL server used for replication in either cluster must be uniquely identified among all the MySQL replication servers participating in either cluster (you cannot have replication servers on both the master and slave clusters sharing the same ID). This can be done by starting each MySQL Server node using the `--server-id=id` option, where `id` is a unique integer.

Servers involved in replication must be compatible with one another with respect to both the version of the replication protocol used and the SQL feature sets which they support. The simplest and easiest way to assure that this is the case is to use the same MySQL version for all servers involved.

Copyright © 2006, MySQL AB

For the sake of the following sections we assume that the slave server or cluster is dedicated to the replication of the master, and that there are no other transactions being committed against it.

Replication Schema and Tables

A MySQL Cluster leveraging replication has to make use of a number of dedicated tables in a separate database named `cluster`, which is local to each MySQL Server node in both the replication master and the replication slave (whether the slave is a single server or a cluster). This database, which is created during the MySQL installation process, contains a table for storing the binary log's indexing data. As the `binlog_index` table is local to each MySQL server and does not participate in clustering, it uses the MyISAM storage engine. This means it must be created on all MySQL Servers belonging to the master cluster. This table is defined as follows:

```
CREATE TABLE `binlog_index` (
            `Position`  BIGINT(20) UNSIGNED NOT NULL,
            `File`      VARCHAR(255) NOT NULL,
            `epoch`     BIGINT(20) UNSIGNED NOT NULL,
            `inserts`   BIGINT(20) UNSIGNED NOT NULL,
            `updates`   BIGINT(20) UNSIGNED NOT NULL,
            `deletes`   BIGINT(20) UNSIGNED NOT NULL,
            `schemaops` BIGNINT(20) UNSIGNED NOT NULL,
            PRIMARY KEY (`epoch`))
            ENGINE=MYISAM  DEFAULT CHARSET=latin1;
```

Please note that prior to MySQL 5.1.8, the `cluster` database was known as the `cluster_replication` database.

An additional table, named `apply_status`, is used to keep a record of the operations that have been replicated from the master to the slave. The data within `apply_status` is not specific to any one SQL node in the (slave) cluster, and so `apply_status` can use the NDB Cluster storage engine, as shown here:

```
CREATE TABLE `apply_status` (
      `server_id` INT(10) UNSIGNED NOT NULL,
      `epoch`     BIGINT(20) UNSIGNED NOT NULL,
      PRIMARY KEY  USING HASH (`server_id`))
      ENGINE=NDBCLUSTER
      DEFAULT CHARSET=latin1;
```

The `binlog_index` and `apply_status` tables are created in a separate database because they should not be replicated. No user intervention is normally required to create or maintain either of them. Both the `binlog_index` and the `apply_status` tables are maintained by the NDB injector thread. This keeps the master MySQL Server updated to changes performed by the NDB storage engine. The NDB binlog injector thread receives events directly from the NDB storage engine. The NDB injector is responsible for capturing all the data events within the cluster, and ensures that all events changing, inserting, or deleting data are recorded in the `binlog_index` table. The slave I/O thread will transfer the master's binary log to the slave's relay log.

Replication Setup

Preparing the MySQL Cluster for replication consists of the following steps:

1. Create a slave account on the master Cluster with the appropriate privileges:

```
GRANT REPLICATION SLAVE
ON *.* TO 'slave_user'@'slave_host'
IDENTIFIED BY 'slave_password';
```

where *slave_user* is the slave account username, *slave_host* is the hostname or IP address of the replication slave, and *slave_password* is the password to assign to this account.

For security reasons, it is preferable to use a unique user account — not employed for any other purpose — for the replication slave account.

2. Configure the slave to use the master. Using the MySQL client on the slave, submit the `CHANGE MASTER TO` statement:

```
CHANGE MASTER TO
MASTER_HOST='master_host',
MASTER_PORT='master_port,
MASTER_USER='slave_user',
MASTER_PASSWORD='slave_password';
```

where *master_host* is the hostname or IP address of the replication master, *master_port* is the port for the slave to use for connecting to the master, *slave_user* is the username set up for the slave on the master, and *slave_password* is the password set for that user account in the previous step.

You can also configure the slave to use the master by setting the corresponding startup options in the slave server's my.cnf file. To configure the slave in the same way as the preceding example `CHANGE MASTER TO` statement, the following information would need to be included in the slave's `my.cnf` file:

```
[mysqld]
master-host=master_host
master-port=master_port
master-user=slave_user
master-password=slave_password
```

3. In order to provide replication backup capability, you will also need to add an `ndb-connectstring` option to the slave's `my.cnf` file prior to starting the replication process.

```
ndb-connectstring=management_host[:port]
```

4. If the master cluster is already in use, you can create a backup of the master and load this onto the slave to cut down on the amount of time required for the slave to synchronize itself with the master.

5. In the event that you are not using MySQL Cluster on the replication slave, you can create a backup with this command at a shell on the replication master:

```
mysqldump --master-data=1
```

6. Then import the resulting data dump onto the slave by copying the dump file over to the slave. After this, you can use the MySQL client to import the data from the dump file into the slave database as shown here, where *dump_file* is the name of the file that was generated using *mysqldump* on the master, and *db_name* is the name of the database to be replicated:

```
mysql -u root -p db_name < dump_file
```

If you copy the data to the slave in this fashion, you should make sure that the slave is started with the `--skip-slave-start` option on the command line, or else include `skip-slave-start` in the slave's `my.cnf` file to keep it from trying to connect to the master to begin replicating before all the data has been loaded. Once the loading of data has completed, follow the additional steps outlined in the next two sections.

Ensure that each MySQL server acting as a replication master is configured with a unique server ID, and with binary logging enabled, using the row format. These options can be set either in the master server's `my.cnf` file, or on the command line when starting the master MySQL Server process.

Starting Replication

This section outlines the procedure for starting MySQL Cluster replication using a single replication channel.

1. Start the MySQL replication master server by issuing this command from a shell on the master:

```
mysqld --nbdcluster --server-id=id \ --log-bin --binlog-format=row &
```

where *id* is this server's unique ID. This starts the master server's `mysqld` process with binary logging enabled, using the proper row-based logging format.

2. Start the MySQL replication slave server from a shell as shown here:

```
mysqld --ndbcluster --server-id=id &
```

where `id` is the slave server's unique ID. It is not necessary to enable logging on the replication slave.

Be aware that you should use the `--skip-slave-start` option with this command or else you should include `skip-slave-start` in the slave server's `my.cnf` file, unless you want replication to begin immediately. With the use of this option, the start of replication is delayed until the appropriate `START SLAVE` statement has been issued, as explained in Step 4 below.

3. It is necessary to synchronize the slave server with the master server's replication binlog. If binary logging has not previously been running on the master, run the following statement on the slave:

```
CHANGE MASTER TO
MASTER_LOG_FILE='',
MASTER_LOG_POS=4;
```

This instructs the slave to begin reading the master's binary log from the log's starting point.

4. Finally, you must instruct the slave to begin applying replication by issuing this command from the MySQL client on the replication slave:

```
START SLAVE;
```

This also initiates the transmission of replication data from the master to the slave.

Additional Notes on MySQL Cluster Replication

The following are known limitations when using replication with MySQL Cluster in MySQL 5.1:

- The use of data definition statements (DDL), such as `CREATE TABLE`, `DROP TABLE`, and `ALTER TABLE`, are recorded in the binary log for only the MySQL server on which they are issued.
- A MySQL server involved in replication should be started or restarted after using `ndb_restore` to discover and setup replication of NDB Cluster tables. Alternatively, you can issue a `SHOW TABLES` statement on all databases in the cluster.
- Similarly, when using `CREATE SCHEMA`, the new database is not automatically discoverable by the MySQL server. Thus, this statement must be issued on each MySQL server participating in the cluster when creating a new database.
- Restarting the cluster with the `--initial` option will cause the sequence of GCI and epoch numbers to start over from 0. (This is generally true of MySQL Cluster and not limited to replication scenarios involving Cluster.) The MySQL servers involved in replication should in this case be replicated. After this, you should use the `RESET MASTER` and `RESET SLAVE` statements to clear the invalid `binlog_index` and `apply_status` tables.

For information concerning the use of two replication channels, implementing fail over or how to configure backups on MySQL Clusters that make use of replication, please refer to the MySQL 5.1 Reference Manual, http://dev.mysql.com/doc/refman/5.1/en/mysql-cluster-replication.html .

Faster ADD/DROP INDEX

MySQL Cluster 5.1 introduces the ability to add an index without disruption to active transactions or causing a resource bottleneck as was observable in previous versions. Let's review how adding an index was performed in versions 4.1 and 5.0.

1. We start with a table (*table_1*) with two associated indexes (*index 1* & *index 2*)
2. Next a temporary table (*temp table*) is created
3. Existing data from *table_1*, *index 1*, *index 2* ***and*** the new index (*index 3*) are created
4. Once completed, the original table is deleted and the temporary table is renamed

In Figure 5 is a graphical depiction of index creation in MySQL Cluster versions 4.1 and 5.0.

Copyright © 2006, MySQL AB

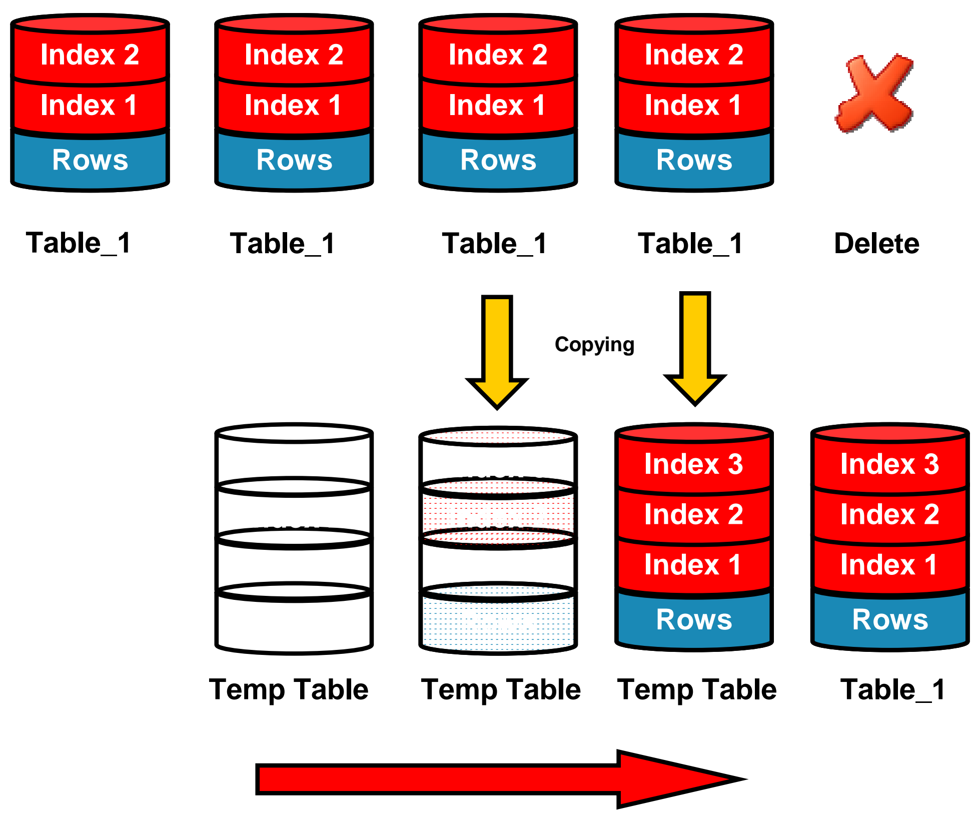

Figure 5

Adding an index in 5.1, as illustrated in Figure 6, is now a much faster operation. This means that there is no longer any temporary table creation, no recreation of data or deletion required. The advantages of a faster `ADD INDEX` are better performing table maintenance operations, less memory and disk requirements, as you no longer have to have enough storage to hold a duplicate copy of the table during the maintenance operation, and a Cluster that is more adaptive to the changing needs of the applications it supports.

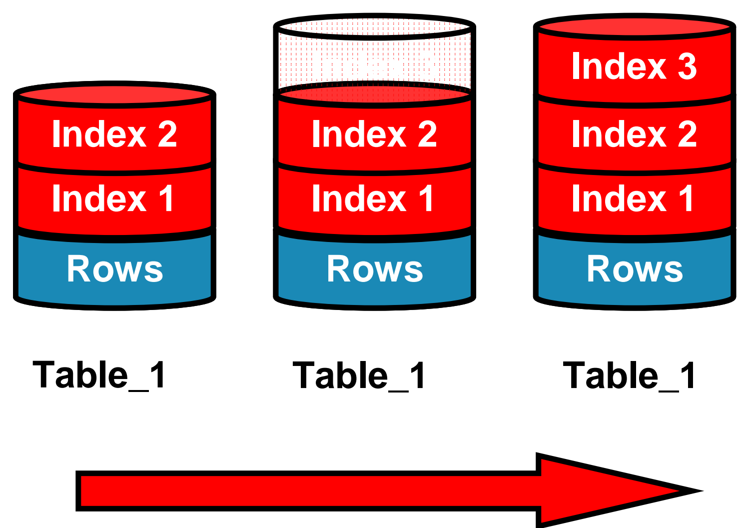

Figure 6

Copyright © 2006, MySQL AB

Below is an example of the performance differences between creating and dropping an index using version 5.0 vs. 5.1. As you can see, version 5.1 is roughly four times faster in the test.

- **Version 5.0**

```
mysql> CREATE INDEX b ON t1(b);

Query OK, 1356 rows affected (2.20 sec)
Records: 1356 Duplicates: 0 Warnings: 0

mysql> DROP INDEX b ON t1;

Query OK, 1356 rows affected (2.03 sec)
Records: 1356 Duplicates: 0 Warnings: 0
```

- **Version 5.1**

```
mysql> CREATE INDEX b ON t1(b);

Query OK, 1356 rows affected (0.58 sec)
Records: 0 Duplicates: 0 Warnings: 0

mysql> DROP INDEX b ON t1;

Query OK, 1356 rows affected (0.46 sec)
Records: 1356 Duplicates: 0 Warnings: 0
```

Similar results can be observed when deleting indexes as well.

Efficient Variable-Sized Records

Previously in versions 4.1 and 5.0 of MySQL Cluster, the use of variable-sized records within a table consumed more space then was necessary. Now, in version 5.1, variable-sized records are dealt with in a much more efficient manner. This means that only the data that is actually resident within the row consumes memory. The advantages are that there is no more wasted memory, more rows per gigabyte can be stored and larger datasets can be leveraged for in-memory data. An example of the differences between the two is depicted below in Figure 6. Please note, as mentioned previously, disk-based data does not support variable-sized records in the same efficient manner as MySQL Cluster does for in-memory data.

Version 4.1 and 5.0

int	int	varchar(255)	
1	2	the quick brown fox	**wasted space**
3	4	the quick brown fox	**wasted space**
5	6	the quick brown fox	**wasted space**
7	8	the quick brown fox	**wasted space**
9	9	the quick brown fox	**wasted space**

Version 5.1

int	int	varchar(255)
1	2	the quick brown fox
3	4	the quick brown fox
5	6	the quick brown fox
7	8	the quick brown fox
9	9	the quick brown fox

Figure 6

Conclusion

In this paper we explored in some detail the new features which have been introduced in MySQL Cluster 5.1.

- Support for Disk Data
- Row-based Replication
- Higher performance ADD/DROP INDEX
- More efficient Variable Sized Records

With the introduction of disk-based data, MySQL Cluster is now able to support much larger datasets then were previously possible because of operating system or physical RAM limitations.

Copyright © 2006, MySQL AB

Support for row-based replication now allows the MySQL Cluster to achieve an even higher level of availability and scalability. This functionality can be leveraged to enable Scale-out in order to increase scalability and performance, create replicated databases for fail-over or for use in maintenance operations such as, upgrades or backups. Replication of a MySQL Cluster can also be used to create redundancy within the data center or across a geographic WAN.

Improvements in the performance of index operations coupled with a more efficient manner in supporting variable-sized records, make MySQL Cluster a faster and more efficient database server.

MySQL Cluster continues down a development path focused on delivering a dramatically lower TCO for clustered databases and at the same time facilitating the ability to leverage a Scale-out methodology using commodity hardware and open source components.

About MySQL

MySQL AB develops, markets, and supports a family of high performance, affordable database servers and tools.

The company's flagship product is MySQL, the world's most popular open source database, with more than six million active installations. Many of the world's largest organizations, including Google, Sabre Holdings, The Associated Press, Suzuki and NASA, are realizing significant cost savings by using MySQL to power web sites, business-critical enterprise applications and packaged software. MySQL AB is a second generation open source company, and supports both open source values and corporate customers' needs in a profitable, sustainable business. For more information about MySQL, please go to http://www.mysql.com/

Resources

Below are links to additional high availability resources from MySQL.

White Papers

http://www.mysql.com/why-mysql/white-papers/

Case Studies

http://www.mysql.com/why-mysql/case-studies/

Press Releases, News and Events

http://www.mysql.com/news-and-events/

Live Webinars

http://www.mysql.com/news-and-events/web-seminars/

Webinars on Demand

http://www.mysql.com/news-and-events/on-demand-webinars/

Copyright © 2006, MySQL AB