Optimizing Performance of the MySQL Cluster Database

A MySQL Technical Whitepaper

Table of Contents

Page #

1. Introduction

Whether you're racing to introduce a new service, or trying to manage an avalanche of data in real time, your database has to be scalable, fast and highly available to meet ever-changing market conditions and stringent SLAs (Service Level Agreements). The MySQL Cluster database has been designed to enable users to meet these challenges.

With a "shared-nothing" distributed architecture ensuring no single point of failure, MySQL Cluster is proven to deliver 99.999% availability, allowing users to meet their most demanding mission-critical application requirements.

Its real-time design delivers consistent, millisecond response latency with the ability to service tens of thousands of transactions per second for both read and write-intensive workloads. Support for in-memory and disk based data, automatic data partitioning with load balancing and the ability to add nodes to a running cluster with zero downtime allows very high levels of database scalability to handle the most unpredictable workloads.

The benefits of MySQL Cluster have been realized in some of the most performance-demanding data management environments in the telecommunications, web, finance and government sectors, for the likes of Alcatel-Lucent, Cisco, Ericsson, Juniper, Shopatron, Telenor, UTStarcom and the United States Navy.

It is important to recognize that as a distributed, shared nothing database, there are classes of application characteristics that represent ideal candidates for the architecture of MySQL Cluster. There are also applications where the use of MySQL Cluster requires greater effort be applied to tuning and optimization in order to achieve the required levels of performance (measured as both transaction throughput and response time latency).

> *"Telenor has found MySQL Cluster to be the best performing database in the world for our applications".*
>
> **Peter Eriksson, Manager Network Provisioning Telenor**

The purpose of this whitepaper is to explore how to tune and optimize MySQL Cluster to handle diverse workload requirements. The paper discusses data access patterns and building distribution awareness into applications, before exploring schema and query optimization, tuning of parameters and how to get the best out of the latest innovations in hardware design.

The paper concludes with recent performance benchmarks performed on the MySQL Cluster database, an overview of how MySQL Cluster can be integrated with other MySQL storage engines, before summarizing additional resources that will enable you to optimize MySQL Cluster performance.

2. Identifying Optimal Applications for MySQL Cluster

> *"MySQL Cluster delivers carrier-grade levels of availability and performance with linear scalability on commodity hardware. It is a fraction of the cost of proprietary alternatives, allowing us to compete aggressively, and enabling operators to maximize their ARPU"*
>
> **Jan Martens, Managing Director SPEECH DESIGN Carrier Systems GmbH**

To achieve high availability through both redundancy and fast failover, tables managed by MySQL Cluster are divided into partitions and distributed across data nodes within the cluster.

This serves to create not just a highly available, fault resilient environment but also a multi-master database with a parallel architecture, enabling high volumes of both read AND write operations to be executed concurrently. Any updates are instantly available to all application (SQL or NDB API) nodes accessing data stored in the Data Nodes.

As write loads are distributed across all of the data nodes, MySQL Cluster can deliver very high levels of write throughput and scalability for transactional workloads. In addition, MySQL Cluster can leverage many SQL nodes running in parallel, with each node handling multiple connections, thus providing support for high-concurrency transactional applications.

Figure 1 below shows the architecture of the MySQL Cluster database.

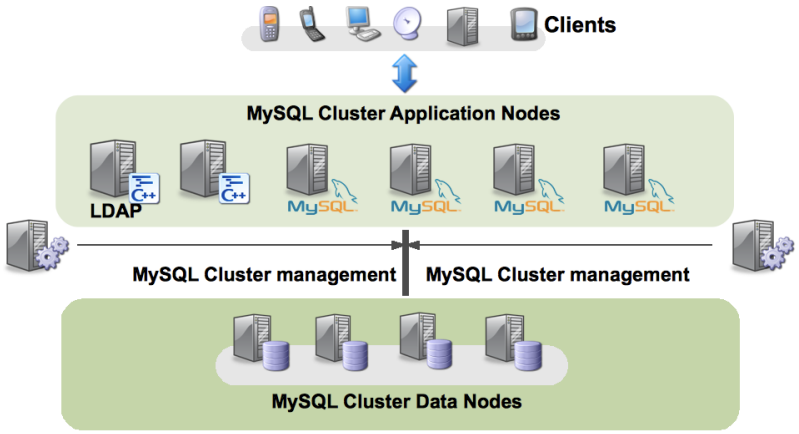


Figure 1: ***MySQL Cluster provides a multi-master database with a parallel architecture***

To learn more about the MySQL Cluster architecture, refer to the MySQL Cluster Architecture and New Features whitepaper posted at:
http://www.mysql.com/why-mysql/white-papers/mysql_wp_cluster7_architecture.php

As a result of its distributed, shared nothing architecture, it is important to understand the types of workload that are best served by MySQL Cluster. The following table summarizes the key characteristics to consider when qualifying whether MySQL Cluster will meet your own requirements.

Good Fit for MySQL Cluster	Complement With Other MySQL Storage Engines
- High write throughput with low latency response	- Database size of more than 2TB
- 99.999% availability with sub-second failover and automatic recovery	- Rows demanding more than 8KB of memory (excluding BLOBs) and 128 fields
- Unpredictable scalability demands	- Requirement to store objects over 2MB
- Mainly primary key access, or sub-components of multi-field primary key	- High volume of reporting requiring full table scans with "complex" joins
- "Simple" joins	- Foreign key support

Figure 2: ***Identifying Optimal Application Fit for MySQL Cluster***

With the above considerations in mind, the following represents a sample of applications where MySQL Cluster has been successfully deployed[1].

Authentication / Authorization / Accounting	Mobile Content Delivery
Application Servers	On-Line Application Stores and Portals
LDAP / RADIUS / Diameter Data Store	On-Line Gaming
eCommerce Services (i.e. shopping basket, payment transactions, fulfillment, etc.)	Payment Processing
IP MultiMedia Subsystem Services	Telco Service Delivery Platforms & Value-Added Services
Intelligent Network Nodes	Subscriber Databases (i.e. HLR, HSS, etc.)
Location-Based Services & Presence Management	VoIP, IPTV & Video-on-Demand Service
Message Stores / Queues	Web Session Management

The following sections of the whitepaper discuss how to measure and optimize the performance of the MySQL Cluster database itself in order to bring its benefits to a broad range of application services.

[1] Click here to see a full list of MySQL Cluster user case studies and applications: http://www.mysql.com/customers/cluster/

3. Measuring Performance and Identifying issues

Before optimizing database performance, it is best practice to use a repeatable methodology to measure performance – both transaction throughput and latency. As part of the optimization process it is necessary to consider changes to both the database itself, and how the application uses the database. Therefore, the ideal performance measurements would be conducted using the real application with a representative traffic model. Performance measurements should be made before any changes are implemented and then again after each optimization is introduced in order to check the benefits (or in some cases, degradations) of the change. In many cases, there is a specific performance requirement that needs to be met, so an iterative measure / optimize / measure process allows you to track how you are progressing towards that goal.

If it isn't possible to gather measurements using the real application, then the mysqlslap tool can be used to generate traffic using multiple connections. To make the tests repeatable, the best approach is to create files with the initial setup (creating tables and data) and then another containing the queries to be repeatedly submitted:

create.sql:

```
CREATE TABLE sub_name (sub_id INT NOT NULL PRIMARY KEY, name VARCHAR(30)) engine=ndb;
CREATE TABLE sub_age (sub_id INT NOT NULL PRIMARY KEY, age INT) engine=ndb;
INSERT INTO sub_name VALUES (1,'Bill'),(2,'Fred'),(3,'Bill'),(4,'Jane'),(5,'Andrew'),(6,'Anne'),
(7,'Juliette'),(8,'Awen'),(9,'Leo'),(10,'Bill');
INSERT INTO sub_age VALUES (1,40),(2,23),(3,33),(4,19),(5,21),(6,50),(7,31),(8,65),(9,18),
(10,101);
```

query.sql:

```
SELECT sub_age.age FROM sub_name, sub_age where sub_name.sub_id=sub_age.sub_id
AND sub_name.name='Bill' ORDER BY sub_age.age;
```

These files are then specified on the command line when running mysqlslap:

```
shell> mysqlslap --concurrency=5 --iterations=100 --query=query.sql --create=create.sql

Benchmark
    Average number of seconds to run all queries: 0.132 seconds
    Minimum number of seconds to run all queries: 0.037 seconds
    Maximum number of seconds to run all queries: 0.268 seconds
    Number of clients running queries: 5
    Average number of queries per client: 1
```

If you want to use an existing database then do not specify a 'create' file and qualify table names in the 'query' file (for example “clusterdb.sub_name”).

The list of queries to use with mysqlslap can either come from the Slow Query log (queries taking longer than a configurable time, see below for details) or from the General log (all queries).

As well as measuring the overall performance of your database application, it is also useful to look at individual database transactions. There may be specific queries that you know are taking too long to execute, or you may want to identify those queries that would deliver the greatest “bang for your buck” if they were optimized. In some cases there may be instrumentation in the application that can determine which queries are being executed most often and/or taking longest to complete.

If you have access to MySQL Enterprise Monitor and are using SQL to query the database, you can use the powerful Query Analyzer (which can track down expensive transactions). The Query Analyzer works by having a proxy sit between the application and the MySQL Server nodes and so there is a performance impact when it is used. A trial version of MySQL Enterprise Monitor can be obtained at the following URL: http://www.mysql.com/trials/

The Query Analyzer cannot examine any database operations performed through the NDB API or any intermediary database access layer (such as MySQL Cluster's JPA interface or the back-ndb driver for OpenLDAP that uses the

NDB API). Note that, the usefulness of MySQL Enterprise Monitor for MySQL Cluster is currently limited as it can only monitor the MySQL Server nodes and not the data nodes, which is where the real resources of interest lie.

If MySQL Enterprise Monitor is not available then MySQL's slow query log can be used to identify the transactions that take the longest time to execute. Note that identifying the slowest query isn't by itself always sufficient to identify the best queries to optimize – it is also necessary to look at the frequency of different queries (more benefit could be derived from optimizing a query that takes 20ms but is executed thousands of times per hour than one that takes 10 seconds but only executes once per day).

You can specify how slow a query must be before it is included in the slow query log using the long_query_time variable, the value is in seconds and setting it to 0 will cause all queries to be logged. The following example causes all queries that take more than 0.5 second to be captured in `/data1/mysql/mysqld-slow.log`:

```
mysql> set global slow_query_log=1; // Turns on the logging
mysql> set global long_query_time=0.5 // 500 ms
mysql> show global variables like 'slow_query_log_file';
      +---------------------+-------------------------------+
      | Variable_name       | Value                         |
      +---------------------+-------------------------------+
      | slow_query_log_file | /data1/mysql/mysqld-slow.log |
      +---------------------+-------------------------------+
      1 row in set (0.00 sec)
```

This is an example of an entry from the log file:

```
# Time: 091125 15:05:39
# User@Host: root[root] @ localhost []
# Query_time: 0.017187  Lock_time: 0.000078 Rows_sent: 6  Rows_examined: 22
SET timestamp=1259161539;
select * from sub_name, sub_age where sub_name.sub_id=sub_age.sub_id order by sub_age.age;
```

Note: any changes to the long_query_time variable (including setting it for the first time – which is actually changing it from the default of 10 seconds) do not effect active client connections to the MySQL Server – those connections must be dropped and then reestablished.

A tool – mysqldumpslow – is provided to help browse the contents of the log file but you may need to view the log file directly to get sufficient resolution for the times.

As a first step to understanding why a query might be taking too long, the EXPLAIN command can be used to see some of the details as to how the MySQL Server executes the query (for example, what indexes – if any – are used):

```
mysql> explain select * from sub_name, sub_age where sub_name.sub_id=sub_age.sub_id order by
sub_age.age;

+----+-------------+----------+--------+----------------+---------+---------+--------------------------+------+----------------+
| id | select_type | table    | type   | possible_keys  | key     | key_len | ref                      | rows | Extra          |
+----+-------------+----------+--------+----------------+---------+---------+--------------------------+------+----------------+
|  1 | SIMPLE      | sub_age  | ALL    | PRIMARY,index2 | NULL    | NULL    | NULL                     |    8 | Using filesort |
|  1 | SIMPLE      | sub_name | eq_ref | PRIMARY,index1 | PRIMARY | 4       | clusterdb.sub_age.sub_id |    1 |                |
+----+-------------+----------+--------+----------------+---------+---------+--------------------------+------+----------------+
2 rows in set (0.07 sec)
```

The slow query log and the EXPLAIN are not unique to MySQL Cluster and they can be used with all MySQL storage engines.

If the slow query is not sufficient, then the General Log can be enabled to generate a complete view of all queries executed on a MySQL server. The general log is enabled with:

```
mysql> set global general_log=1; // Turns on the logging of all queries - only use for a short
period of time as it is expensive!
```

4. Optimizing MySQL Cluster Performance

4.1. Access patterns

When looking to get the maximum performance out of a MySQL Cluster deployment, it is very important to understand the database architecture. Unlike most other MySQL storage engines, the data for MySQL Cluster tables is not stored in the MySQL Server – instead it is partitioned across a pool of data nodes as shown in Figure 3. The rows for a table are divided into partitions with each data node holding the primary fragment for one partition and a secondary (backup) fragment for another. If satisfying a query requires lots of network hops from the MySQL Server to the data nodes or between data nodes in order to gather the necessary data, then performance will degrade and scalability will be impacted.

Achieving the best performance from a MySQL Cluster database typically involves minimizing the number of network hops.

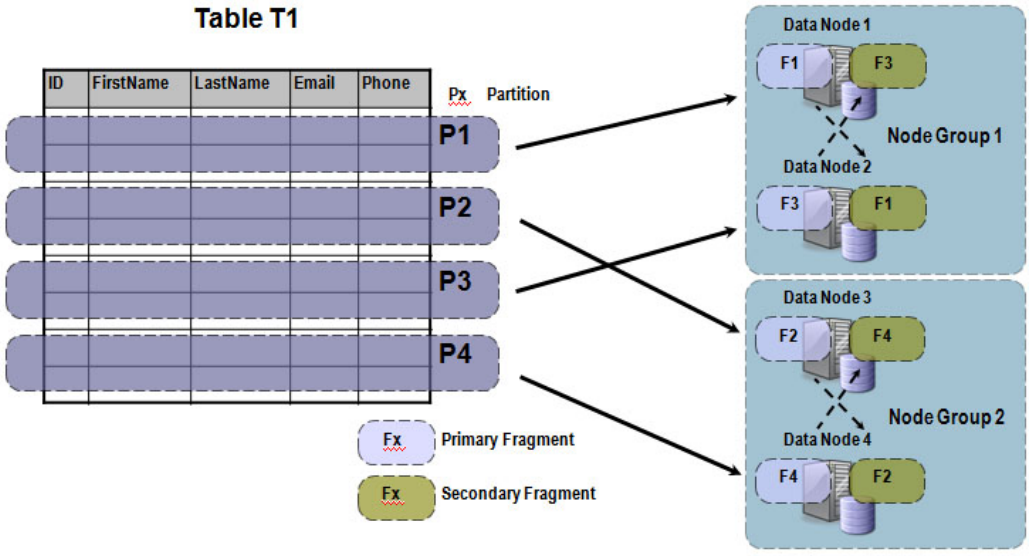


Figure 3: ***Partitioning of MySQL Cluster data***

By default, partitioning is based on a hashing of the primary key for a table, but that can be over-ridden to improve performance as described in section 4.2.- Distribution aware applications.

Simple access patterns are key to building scalable and high performing solutions (this is not unique to the MySQL Cluster storage engine but its distributed architecture makes it even more important).

The best performance will be seen when using Primary Key lookups which can be done in constant time (independent of the size of the database and the number of data nodes). Table 1 shows the cost (time in µs) of typical Primary Key operations using 8 application threads connecting to a single MySQL Server node.

PK Operations	Avg cost (us)	Min cost (us)	Normalized (scalar)
Insert 4B + 64B	768	580	2.74
read 64B	280	178	1
update 64B	676	491	2.41
Insert 4B + 255B	826	600	2.82
read 255B	293	174	1
update 255B	697	505	2.38
delete	636	202	2.17

Table 1: ***Cost of Primary Key operations***

The key things to note from these results are:

1. The amount of data being read or written makes little difference to the time that an operation takes
2. Operations that write (insert, update or delete) data take longer than reads but not massively so (2.2 – 2.8x). The reason that writes take longer is the 2-phase commit protocol that is used to ensure that the data is updated in both the primary and secondary fragments before the transaction commits.

Index searches take longer as the number of tuples (n) in the tables increase and you can expect to see O(log n) latency.

As well as the size of the tables, the performance of an index scan can be influenced by how many data nodes are involved in the scan. If the result set is very large then applying the processing power of all of the data nodes can deliver the quickest results (and this is the default behavior). However, if the result set is comparatively small then limiting the scan to a single data node can reduce latency as there are less network hops required – Section 4.2. - "Distribution aware applications" explains how that type of partition pruning can be achieved.

MySQL Cluster supports joins, but performance can be lower than for other MySQL storage engines if the depth of the join (number of tables involved) and the data sets are too large.

As shown in Figure 4, joins are processed in the MySQL Server process (mysqld) but the MySQL Server must then fetch the data from the data nodes. If this involves just a small number of network trips, then performance will be high but if lots of trips are needed then the latency for the query will increase.

The number of network trips required is influenced by the size of the results set from each stage of the join, as well as the depth of each join. In addition, if the tables involved in the join are large then more time will be consumed for each of the trips to the data nodes.

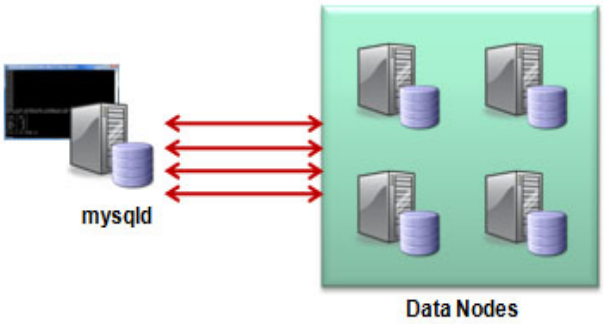


Figure 4: ***Network hops from joins***

This is best illustrated using an example.

Consider a very simple social networking application - I want to find all of my friends who have their birthday this month. To achieve this, a query will be executed that uses data from two existing tables:

```
mysql> describe friends; describe birthday;
+--------+-------------+------+-----+---------+-------+
| Field  | Type        | Null | Key | Default | Extra |
+--------+-------------+------+-----+---------+-------+
| name   | varchar(30) | NO   | PRI | NULL    |       |
| friend | varchar(30) | NO   | PRI |         |       |
+--------+-------------+------+-----+---------+-------+

+-------+-------------+------+-----+---------+-------+
| Field | Type        | Null | Key | Default | Extra |
+-------+-------------+------+-----+---------+-------+
| name  | varchar(30) | NO   | PRI | NULL    |       |
| day   | int(11)     | YES  |     | NULL    |       |
| month | int(11)     | YES  |     | NULL    |       |
| year  | int(11)     | YES  |     | NULL    |       |
+-------+-------------+------+-----+---------+-------+
```

The query creates a join between these 2 tables, first of all finding all of the rows of interest from the friends table (where the name="Andrew") and then looks up all of those friends in the birthday table to check the month of their birthday to see if it is June:

```
mysql> SELECT birthday.name, birthday.day FROM friends, birthday WHERE friend.name='Andrew'
AND friends.friend=birthday.name AND birthday.month=6;
+------+------+
| name | day  |
+------+------+
| Anne |   12 |
```

```
| Rob   |    23 |
+-------+-------+
```

Figure 5 shows how this query is executed. The MySQL Server first fetches all rows from the friends table where the name field matches “Andrew”. The MySQL server then loops through each row in the results set and sends a new request to the data nodes for each one, asking for any matching rows where the name matches a specific friend and the birthday month is June (6th month).

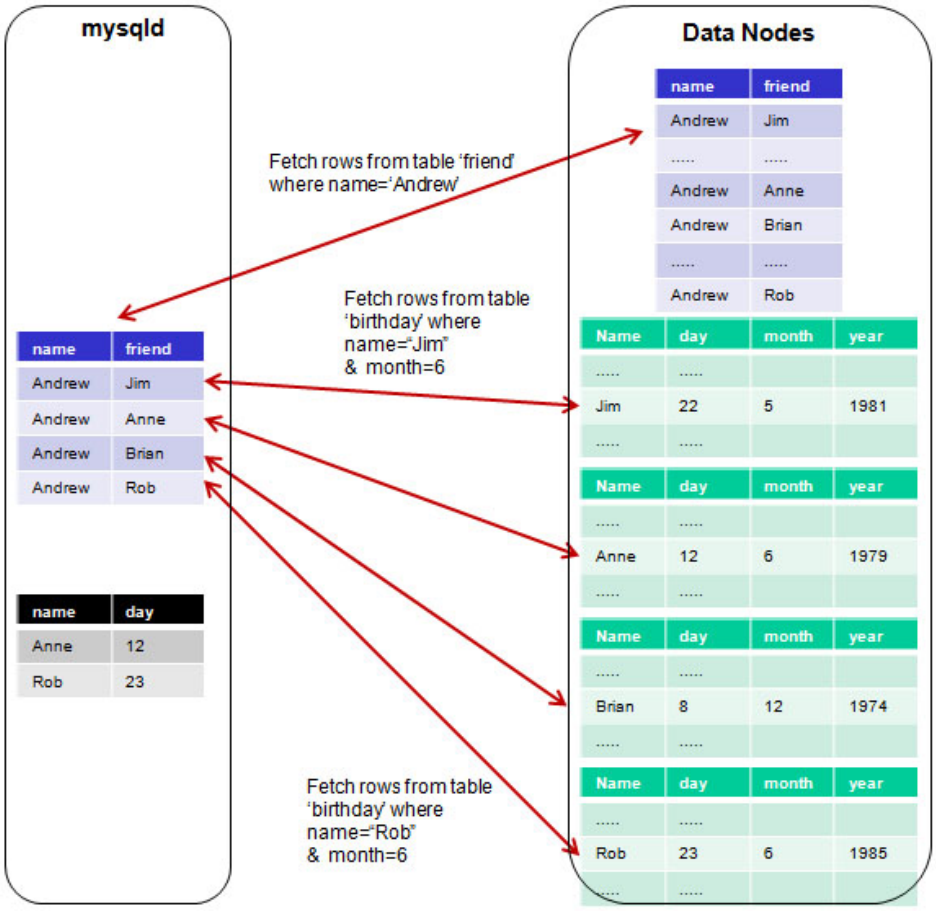


Figure 5: ***Complexity of table joins***

The performance of this join should be good as there are only 4 rows (friends) in the first result set and so there are only 5 trips to the data nodes in total. Consider though if I was more popular (!) and there were thousands of results from the first part of the query. Additionally, if the join was extended another level deep by looking up the birthday for each of the matching friends (birthday in June) in a star-sign table – that would also add more network trips.

If your application needs to perform read-only joins that would be too expensive on your primary (MySQL Cluster) database then you could consider replicating that data (using MySQL asynchronous replication) to another database using a more appropriate storage engine.

Join performance is an active area of development and improvements are planned to be available in future releases.

4.2. Distribution aware applications

As discussed above, when adding rows to a table that uses MySQL Cluster as the storage engine, each row is assigned to a partition where that partition is mastered by a particular data node in the Cluster (as shown in Figure

3). The best performance is often achieved when all of the data required to satisfy a transaction is held within a single partition. This means that it can be satisfied within a single data node rather than being bounced back and forth between multiple nodes, resulting in extra latency.

By default, MySQL Cluster partitions the data by hashing the primary key. As demonstrated below, this is not always optimal. For example, if there are two tables, the first using a single-column primary key (sub_id) and the second using a composite key (sub_id, service_name):

```
mysql> describe names;
+---------+-------------+------+-----+---------+-------+
| Field   | Type        | Null | Key | Default | Extra |
+---------+-------------+------+-----+---------+-------+
| sub_id  | int(11)     | NO   | PRI | NULL    |       |
| name    | varchar(30) | YES  |     | NULL    |       |
+---------+-------------+------+-----+---------+-------+

mysql> describe services;
+--------------+-------------+------+-----+---------+-------+
| Field        | Type        | Null | Key | Default | Extra |
+--------------+-------------+------+-----+---------+-------+
| sub_id       | int(11)     | NO   | PRI | 0       |       |
| service_name | varchar(30) | NO   | PRI |         |       |
| service_parm | int(11)     | YES  |     | NULL    |       |
+--------------+-------------+------+-----+---------+-------+
```

If we then add data to these (initially empty) tables, we can then use the EXPLAIN command to see which partitions (and hence physical hosts) are used to store the data for this single subscriber:

```
mysql> insert into names values (1,'Billy');

mysql> insert into services values (1,'VoIP',20),(1,'Video',654),(1,'IM',878),(1,'ssh',666);

mysql> explain partitions select * from names where sub_id=1;
+----+-------------+-------+------------+-------+---------------+---------+---------+-------+------+-------+
| id | select_type | table | partitions | type  | possible_keys | key     | key_len | ref   | rows | Extra |
+----+-------------+-------+------------+-------+---------------+---------+---------+-------+------+-------+
|  1 | SIMPLE      | names | p3         | const | PRIMARY       | PRIMARY | 4       | const |    1 |       |
+----+-------------+-------+------------+-------+---------------+---------+---------+-------+------+-------+

mysql> explain partitions select * from services where sub_id=1;
+----+-------------+----------+-------------+------+---------------+---------+---------+-------+------+-------+
| id | select_type | table    | partitions  | type | possible_keys | key     | key_len | ref   | rows | Extra |
+----+-------------+----------+-------------+------+---------------+---------+---------+-------+------+-------+
|  1 | SIMPLE      | services | p0,p1,p2,p3 | ref  | PRIMARY       | PRIMARY | 4       | const |   10 |       |
+----+-------------+----------+-------------+------+---------------+---------+---------+-------+------+-------+
```

The service records for the same subscriber (sub_id = 1) are split across 4 different partitions (p0, p1, p2 & p3). This means that the query results in messages being passed backwards and forwards between the 4 different data nodes which consumes extra CPU (Central Processing Unit) time and incurs additional network latency.

We can override the default behavior by telling MySQL Cluster which fields from the Primary Key should be fed into the hash algorithm. For this example, it is reasonable to expect a transaction to access multiple records for the same subscriber (identified by their sub_id) and so the application will perform best if all of the rows for that sub_id are held in the same partition:

```
mysql> drop table services;

mysql> create table services (sub_id int, service_name varchar (30), service_parm int, primary key
(sub_id, service_name)) engine = ndb
-> partition by key (sub_id);

mysql> insert into services values (1,'VoIP',20),(1,'Video',654),(1,'IM',878),(1,'ssh',666);

mysql> explain partitions select * from services where sub_id=1;
+----+-------------+----------+------------+------+---------------+---------+---------+-------+------+-------+
| id | select_type | table    | partitions | type | possible_keys | key     | key_len | ref   | rows | Extra |
+----+-------------+----------+------------+------+---------------+---------+---------+-------+------+-------+
|  1 | SIMPLE      | services | p3         | ref  | PRIMARY       | PRIMARY | 4       | const |   10 |       |
+----+-------------+----------+------------+------+---------------+---------+---------+-------+------+-------+
```

Now all of the rows for sub_id=1 from the services table are now held within a single partition (p3) which is the same as that holding the row for the same sub_id in the names table. Note that it wasn't necessary to drop, recreate and re-provision the services table, the following command would have had the same effect:

```
mysql> alter table services partition by key (sub_id);
```

This process of allowing an index scan to be performed by a single data node is referred to as partition pruning. Another way to verify that partition pruning has been implemented is to look at the ndb_pruned_scan_count status variable:

```
mysql> show global status like 'ndb_pruned_scan_count';
+------------------------+-------+
| Variable_name          | Value |
+------------------------+-------+
| Ndb_pruned_scan_count | 12    |
+------------------------+-------+
select * from services where sub_id=1;
+--------+--------------+--------------+
| sub_id | service_name | service_parm |
+--------+--------------+--------------+
|      1 | IM           |          878 |
|      1 | ssh          |          666 |
|      1 | Video        |          654 |
|      1 | VoIP         |           20 |
+--------+--------------+--------------+

mysql> show global status like 'ndb_pruned_scan_count';
+------------------------+-------+
| Variable_name          | Value |
+------------------------+-------+
| Ndb_pruned_scan_count | 13    |
+------------------------+-------+
```

The fact that the count increased by 1 confirms that partition pruning was possible.

The benefits from partition pruning are dependent on the number of data nodes in the Cluster and the size of the result set – with the best improvements achieved with more data nodes and less records to be retrieved.

Figure 6 shows how partition pruning (orange bars) reduces latency for smaller result sets, but can actually increase it for larger result sets. Note that shorter bars/lower latency represents better performance.

Figure 6: ***Effects of index scan partition pruning***

4.3. Batching operations

Batching can be used to read or write multiple rows with a single trip from the MySQL Server to the data node(s), greatly reducing the time taken to complete the transaction. It is possible to batch inserts, index scans (when not part of a join) and Primary Key reads, deletes and most updates.

Batching can be as simple as inserting multiple rows using one SQL statement, for example, rather than executing multiple statements such as:

```
mysql> insert into services values (1,'VoIP',20);
mysql> insert into services values (1,'Video',654);
mysql> insert into services values (1,'IM',878);
mysql> insert into services values (1,'ssh',666);
```

you can instead use a single statement:

```
mysql> insert into services values (1,'VoIP',20),(1,'Video',654),(1,'IM',878),(1,'ssh',666);
```

In a test where 1M insert statements were replaced with 62.5K statement where each statement inserted 16 rows, the total time was reduced from 765 to 50 seconds – a 15.3x improvement! For obvious reasons, batching should be used whenever practical.

Similarly, multiple select statements can be replaced with one:

```
mysql> select * from services where sub_id=1 AND service_name='IM';
+--------+--------------+--------------+
| sub_id | service_name | service_parm |
+--------+--------------+--------------+
|      1 | IM           |          878 |
+--------+--------------+--------------+
1 row in set (0.00 sec)

mysql> mysql> select * from services where sub_id=1 AND service_name='ssh';
+--------+--------------+--------------+
| sub_id | service_name | service_parm |
+--------+--------------+--------------+
|      1 | ssh          |          666 |
+--------+--------------+--------------+
1 row in set (0.00 sec)

mysql> select * from services where sub_id=1 AND service_name='VoIP';
+--------+--------------+--------------+
| sub_id | service_name | service_parm |
+--------+--------------+--------------+
|      1 | VoIP         |           20 |
+--------+--------------+--------------+
```

replaced with:

```
mysql> select * from services where sub_id=1 AND service_name IN ('IM','ssh','VoIP');
+--------+--------------+--------------+
| sub_id | service_name | service_parm |
+--------+--------------+--------------+
|      1 | IM           |          878 |
|      1 | ssh          |          666 |
|      1 | VoIP         |           20 |
+--------+--------------+--------------+
```

It is also possible to batch operations on multiple tables by turning on the transaction_allow_batching parameter and then including multiple operations between BEGIN and END statements:

```
mysql> SET transaction_allow_batching=1;
mysql> BEGIN;
mysql> INSERT INTO names values (101, 'Andrew');
mysql> INSERT INTO services VALUES (101, 'VoIP', 33);
mysql> UPDATE services SET service_parm=667 WHERE service_name='ssh';
mysql> COMMIT;
```

transaction_allow_batching does not work with SELECT statements or UPDATEs which manipulate variables.

4.4. Schema optimizations

There are two main ways to modify your data schema to get the best performance: use the most efficient types and denormalize your schema where appropriate.

Only the first 255 bytes of BLOB and TEXT columns are stored in the main table with the rest stored in a hidden table. This means that what appears to your application as a single read is actually executed as two reads. For this reason, the best performance can be achieved using the VARBINARY and VARCHAR types instead. Note that you will still need to use BLOB or TEXT columns if the overall size of a row would otherwise exceed 8,052 bytes.

The number of reads and writes can be reduced by denormalizing your tables.

Figure 7 shows voice and broadband data split over two tables which means that two reads are needed to return all of the user's data using this join:

```
mysql> SELECT * FROM USER_SVC_BROADBAND AS bb,
USER_SVC_VOIP AS voip WHERE bb.userid=voip.userid
AND bb.userid=1;
```

userid	voip_data
1	<data>
2	<data>
3	<data>
4	<data>

USER_SVC_VOIP

userid	bb_data
1	<data>
2	<data>
3	<data>
4	<data>

USER_SVC_BROADBAND

Figure 7: *Normalized Tables*

which results in a total throughput of 12,623 transactions per second (average response time of 658 µs) on a regular Linux-based two data node cluster (2.33GHz, Intel E5345 Quad Core – Dual CPU).

In Figure 8, those two tables are combined into one which means that the MySQL Server only needs to read from the data nodes once:

```
SELECT * FROM USER_SVC_VOIP_BB WHERE userid=1;
```

which results in a total throughput of 21,592 transactions per second (average response time of 371 µs) – a 1.71x improvement.

userid	voip_data	bb_data
1	<data>	<data>
2	<data>	<data>
3	<data>	<data>
4	<data>	<data>

USER_SVC_VOIP_ BB

Figure 8: ***De normalized Table***

4.5. Query optimization

As explained in section 4.1., joins can be expensive in MySQL Cluster. This section works through an example to demonstrate how a join can quickly become very expensive and then how a little knowledge of how application will actually use the data can be applied to significantly boost performance.

For the example, we start with 2 tables as shown in Figure 9 and perform the join as shown:

```
SELECT fname, lname, title FROM a,b WHERE b.id=a.id AND a.country='France';
```

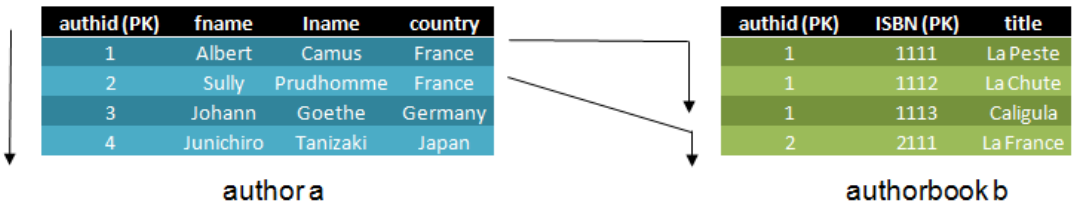


Figure 9: ***Nested Loop Join***

This query performs an index scan of the left table to find matches and then for each match in 'a' it finds matches in 'b'. This could result in a very expensive query in cases where there there are many records matching `a.country='France'`. In this example, both tables have the same primary key and so if possible it would be advantageous to de-normalize and combine them into a single table.

The number of network requests from the MySQL Server to the data nodes also grows as we add an extra table to the join – in this example, checking which book retailers stock books from France using the tables in Figure 10:

```
SELECT c.id FROM a,b,c WHERE c.ISBN=b.ISBN b.id=a.id AND a.country='France' AND c.count>0;
```

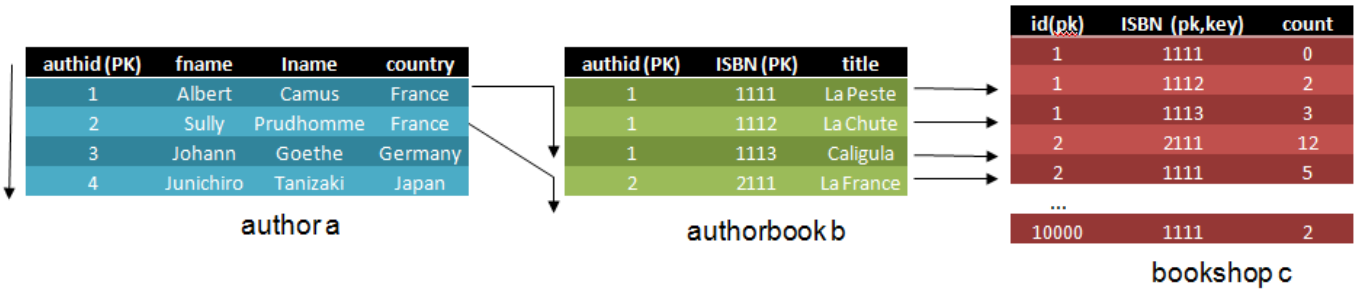


Figure 10: ***Three level join***

Now, for each ISBN in 'authorbook' we have to join on ISBN in bookshop. In this example, there are 4 matches in authorbook which means 4 index scans on bookshop – another 4 network hops. If the application can tolerate this resulting latency, then this design is acceptable.

If performance needs to be improved then the schema can be modified by exploiting knowledge of the real data sets – for example if we know that none of the authors we are interested in have written more than 64 books then we can include a list of ISBN numbers directly into a new version of the author table and reduce the number of network hops to two using this pseudo-SQL as illustrated in Figure 11:

```
SELECT ISBN_LIST FROM a WHERE a.country='France'
SELECT id FROM c WHERE a.isbn in (<ISBN_LIST>);
```

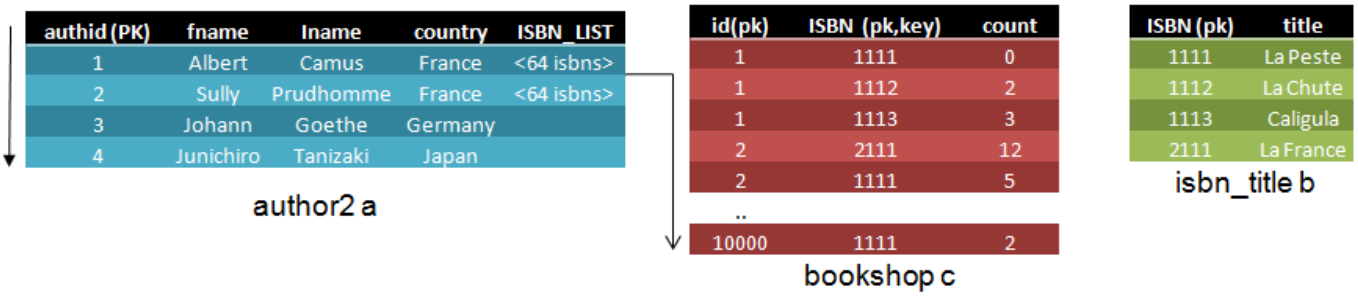


Figure 11: ***Query reduced to 2 network hops***

In this way, one index scan identifies all ISBN values for French books and that list is then used for a single scan of the bookshop table to find matching stores.

The next step is to implement the pseudo-code as a stored procedure:

```
CREATE PROCEDURE ab(IN c VARCHAR(255))
BEGIN
  SELECT @list:=GROUP_CONCAT(isbn) FROM author2 WHERE country=c;
  SET @s = CONCAT("SELECT DISTINCT id FROM bookshop WHERE count>0 AND ISBN IN (", @list, ");");
  PREPARE stmt FROM @s;
  EXECUTE stmt;
END
```

This stored procedure can then be used to implement the join by passing in the string “France”:

```
mysql> call ab('France');
```

This produces the same results as the original join, but is 4.5x faster. Clearly this is a more complex solution with extra development costs and so you would likely stick with the original design if its performance was acceptable.

To confirm whether you are actually reducing the number of requests sent by the MySQL Server to the data nodes, you can check how much the Ndb_execute_count status variable is incremented (this variable counts the number of requests sent from the MySQL Server to the data nodes):

```
mysql> show global status like 'Ndb_execute_count';
mysql> show global status like 'Ndb_execute_count';
+--------------------+-------+
| Variable_name      | Value |
+--------------------+-------+
| Ndb_execute_count | 9     |
+--------------------+-------+
```

In addition to simplifying or removing joins, queries can be made faster by the correct use of indexes; when adding indexes, measure the benefit to see whether they are justified. You can use the EXPLAIN command to check that the correct index is being used.

For example, a table is created with two indexes involving the 'a' column and when we check how a SELECT statement would be executed then both indexes are given as options; as we know which index should give the best performance for this query, we can force its use:

```
mysql> CREATE TABLE t1(id int(11) NOT NULL AUTO_INCREMENT,
                       a bigint(20) DEFAULT NULL,
                       ts timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP,
                       PRIMARY KEY (id),
                       KEY idx_t1_a (a),
                       KEY idx_t1_a_ts (a,ts)) ENGINE=ndbcluster DEFAULT CHARSET=latin1;
mysql> explain select * from t1 where a=2 and ts='2009-10-05 14:21:34';
+----+-------------+-------+------+--------------------+----------+---------+-------+------+-------------+
```

```
| id | select_type | table | type | possible_keys        | key      | key_len | ref   | rows | Extra       |
+----+-------------+-------+------+----------------------+----------+---------+-------+------+-------------+
|  1 | SIMPLE      | t1    | ref  | idx_t1_a,idx_t1_a_ts | idx_t1_a | 9       | const |   10 | Using where |
+----+-------------+-------+------+----------------------+----------+---------+-------+------+-------------+

mysql> explain select * from t1 FORCE INDEX (idx_t1_a_ts) where a=2 and ts='2009-10-05 14:21:34';
+----+-------------+-------+------+---------------+-------------+---------+-------------+------+----------------
| id | select_type | table | type | possible_keys | key         | key_len | ref         | rows | Extra
+----+-------------+-------+------+---------------+-------------+---------+-------------+------+----------------
|  1 | SIMPLE      | t1    | ref  | idx_t1_a_ts   | idx_t1_a_ts | 13      | const,const |   10 | Using where wit
+----+-------------+-------+------+---------------+-------------+---------+-------------+------+----------------
```

4.6. Parameter tuning

There are numerous parameters that can be set to achieve the best performance from a particular configuration. The MySQL Cluster documentation provides a description of each of them, but as a starting point, use the tool at www.severalnines.com/config to get a set of values that are known to work well and then tailor them for your specific circumstances.

4.7. Connection pools

To scale your application, you may have multiple threads, each connecting to a MySQL Server process (mysqld) as shown in Figure 12. To access the data nodes, the mysqld process by default uses a single NDB API connection. Because there are multiple threads all competing for that one connection, there is contention on a mutex that prevents the throughput from scaling. The same problem can occur when using the NDB API directly (see section 4.9. - Alternative APIs).

One workaround would be to have each application thread use a dedicated mysqld process, but that would be wasteful of resources and increase management overhead (for example, user privelidges need to be set in each mysqld separately).

A more effective solution is to have multiple NDB API connections from the mysqld process to the data nodes as shown in Figure 13.

To use connection pooling, each connection (from the mysqld) needs to have its own [mysqld] or [api] section in the config.ini file and then set ndb-cluster-connection-pool to a value greater than one in my.cnf:

```
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
#optional to define the id
```

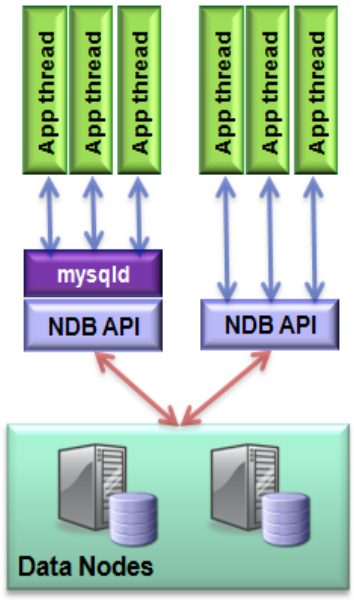


Figure 12: *API Thread Contention*

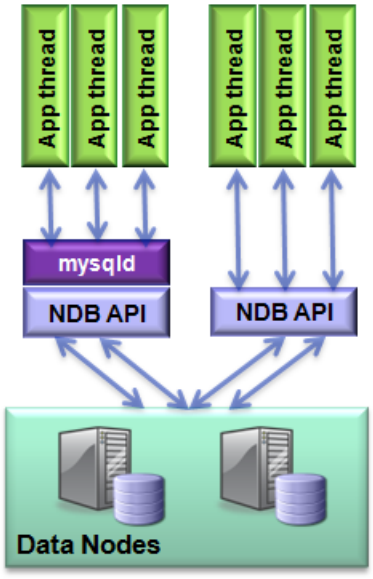


Figure 13: *Multiple Connection pool*

```
[api]
# do not specify id!
hostname=localhost

[api]
# do not specify id!
hostname=localhost

[api]
# do not specify id!
hostname=localhost

my.cnf:
======

[mysqld]
ndbcluster
datadir=/home/billy/mysql/my_cluster/data
basedir=/usr/local/mysql
ndb-cluster-connection-pool=4
# do not specify an ndb-node-id!
```

In addition, do not specify an ndb-node-id as a command line option when starting the mysqld process. In this example, the single mysqld process will have 4 NDB API connections.

Figure 14 shows the performance benefits of using connection pooling. Applications would typically experience a 70% improvement but in some cases it can exceed 150%

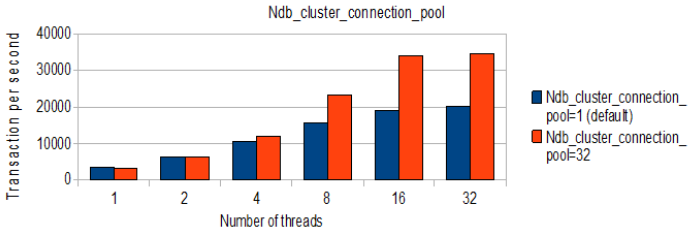


Figure 14: *Benefits of connection pooling*

4.8. Multi-Threaded Data Nodes

Multi-threaded data nodes add the option of “scaling-up” on larger computer hardware by allowing the data nodes to better exploit multiple CPU cores/threads in a single server host. Each of these data nodes is able to make effective use of up to 8 CPU cores/threads. Data is partitioned among a number of threads within the data node process; in effect duplicating the MySQL Cluster partitioned architecture within the data node. This design maximizes the amount of work that can be performed independently by threads and minimizes their communication.

This is achieved by allowing the Local Query Handler to run in a multi-threaded mode while other key activities (Transaction Coordination, replication, IO and transporter) are run in separate threads.

Each of the LQH threads is responsible for one primary sub-partition (a fraction of the partition that a particular data node is responsible for) and one replica sub-partition (assuming that NoOfReplicas is set to 2). The Transaction Coordinator for a data node is responsible for routing requests to the correct LQH thread within the data node as well as to other data nodes if needed.

Figure 15 illustrates this for an 8 core system where 4 threads are used for running instances of the Local Query Handler (LQH).

The Transaction Coordinator handles co-ordination of transactions and timeouts; it serves as the interface to the NDB API for indexes and scan operations.

The Access Manager handles hash indexes of primary keys providing speedy access to the records.

The Tuple Manager handles storage of tuples (records) and contains the filtering engine used to filter out records and attributes.

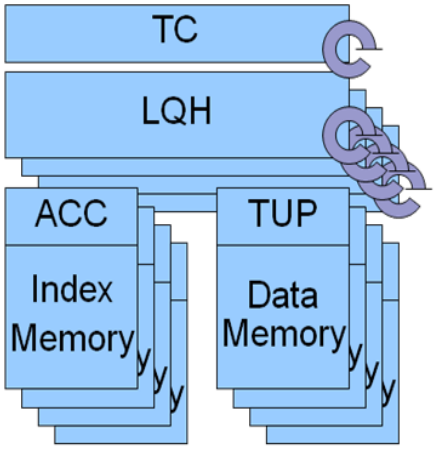


Figure 15: *Multi-threaded data node*

The four LQH threads are in turn allocated work by the single Transaction Coordinator (TC) thread. The Access Manager (ACC) and the Tuple Manager (TUP) are run within each of the LQH threads.

Performance can be increased by more than 4x for 8 core hosts.

ndbmtd is the equivalent of ndbd but for multi-threaded data nodes, the process that is used to handle all the data in tables using the MySQL Cluster NDB data node. ndbmtd is intended for use on host computers having multiple CPU cores/threads.

In almost every way, ndbmtd functions in the same manner as ndbd; and the application therefore does not need to be aware of the change. Command-line options and configuration parameters used with ndbd also apply to ndbmtd. ndbmtd is also file system-compatible with ndbd. In other words, a data node running ndbd can be stopped, the binary replaced with ndbmtd, and then restarted without any loss of data. This all makes it extremely simple for developers or administrators to switch to the multi-threaded version.

Using ndbmtd differs from using ndbd in two key respects:

- You must set an appropriate value for the MaxNoOfExecutionThreads configuration parameter in the config.ini file. If you do not do so, ndbmtd runs in single-threaded mode — that is, it behaves like ndbd.
- Trace files are generated for critical errors in ndbmtd processes in a somewhat different fashion from the way these are generated by ndbd failures.

The MaxNoOfExecutionThreads configuration parameter is used to determine the number of execution threads spawned by ndbmtd. Although this parameter is set in the [ndbd] or [ndbd default] sections of the config.ini file, it is exclusive to ndbmtd and is ignored for ndbd. This parameter takes an integer value from 2 to 8 inclusive. Generally, you should set this to the number of CPU cores/threads on the data node host, as shown in Table 2.

Number of cores	Recommended MaxNoOfExecutionThreads Value
2	2
4	4
8	8

Table 2: *Appropriate settings for MaxNoOfExecutionThreads*

Clearly, the performance improvement realized by this capability will be partly dependent on the application. As an extreme example, if there is only a single application instance which is single threaded, sending simple read or write transactions to the Cluster over a single connection, then no amount of parallelism in the data nodes can accelerate it. So while there is no requirement to change the application, some re-engineering may help to achieve the best improvement.

4.9. Alternative APIs

This white paper has focused on performance when accessing MySQL Cluster data using SQL through the MySQL Server. The best performance can be achieved by accessing the data nodes directly using the C++ NDB API. This requires a greater investment in the development and maintenance of your application, but the rewards can be a leap in throughput and greatly reduced latency. It is also possible to mix-and-match, using SQL for much of your application but the NDB API for specific, performance-critical aspects of your workload. Note that MySQL Server uses the NDB API internally to access the data nodes.

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

To get the best of both worlds – improved performance and easy maintenance – there are growing number of middleware components that go directly to the NDB API. An example of this is the back-ndb driver for OpenLDAP – allowing you to read and write data in MySQL Cluster using the LDAP protocol. MySQL will be releasing a JPA-compliant Java interface which will be the ideal way for high performance Java applications to use MySQL Cluster.

4.10. Hardware enhancements

Much of this white paper has focused on reducing the number of messages being sent between the nodes making up the Cluster; a complementary approach is to use higher bandwidth, lower latency network interconnects. The DX range of SCI products provided by Dolphin Interconnect Solutions (www.dolphinics.com) are known to work very well with MySQL Cluster and can improve performance on network-intensive workloads by more than 2x without making any other optimizations.

Faster CPUs or CPUs with more cores or threads can improve performance. If deploying data nodes on hosts using multiple CPUs/cores/threads, then consider moving to the multi-threaded data node available with MySQL Cluster 7.0 and higher. Disk performance can always be a limiting factor (even when using in-memory tables) – so the faster the disks the better and very promissing results have been acheived using Solid State Device (SSD) storage.

4.11. Miscellaneous

Using the MySQL Query cache is rarely useful for MySQL Cluster and it is recommended to disable it.

Make sure that the data nodes do not use SWAP space rather than real memory – this impacts both performance and system stability.

When using Ethernet for the interconnects, lock data node threads to CPUs that are not handling the network interrupts. When using Sun CMT (Chip Multi-Threading) hardware, create processor sets and bind interrupt handling to particular CPUs. Threads can be locked to individual CPUs using the LockExecuteThreadToCPU and the LockMaintThreadsToCPU parameters in the [ndbd] section of config.ini.

5. Scaling MySQL Cluster by Adding Nodes

MySQL Cluster is designed to be scaled horizontally. Simply adding additional MySQL Servers or data nodes will often increase performance – in particular, raising the transaction throughput of the system..

To add extra MySQL Server nodes, it is simply a matter of adding an extra [mysqld] section to the config.ini file, perform a rolling restart and then start the new mysqld process. Any data that is not held in the MySQL Cluster storage engine (for example stored procedures and user priveliges) will have to be recreated on the new server. The other servers continue to provide service while the new ones are added.

It is possible to add new node groups (and thus new data nodes) to a running MySQL Cluster without shutting down and reloading the cluster. Additionally, the existing table data can be repartitioned across all of the data nodes (moving a subset of the data from the existing node groups to the new one).

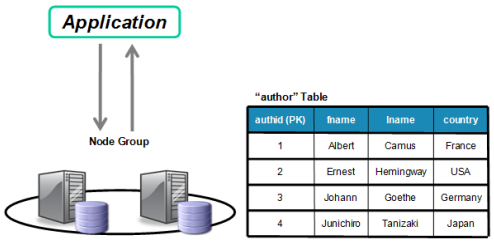
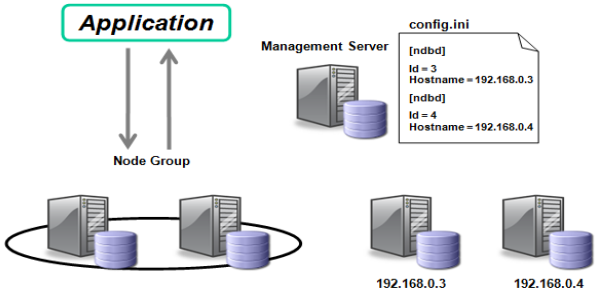


Figure 16: *Starting Configuration*

This section will step through an example of extending a single node group Cluster by adding a second node group and repartitioning the data across the 2 node groups. Figure 16 shows the original system.

This example assumes that the 2 servers making up the new node group have already been commissioned and focuses on the steps required to introduce them into the Cluster.



Step 1: Edit config.ini on all of the management servers as shown in Figure 17.

Figure 17: *Update config.ini on all management servers*

Step 2: Restart the management server(s), existing data nodes and MySQL Servers, as illustrated in Figure 18. Note that you will be made to wait for each restart to be reported as complete before restarting the next node so that service is not interrupted.

In addition, all of the MySQL Server nodes that access the cluster should be restarted.

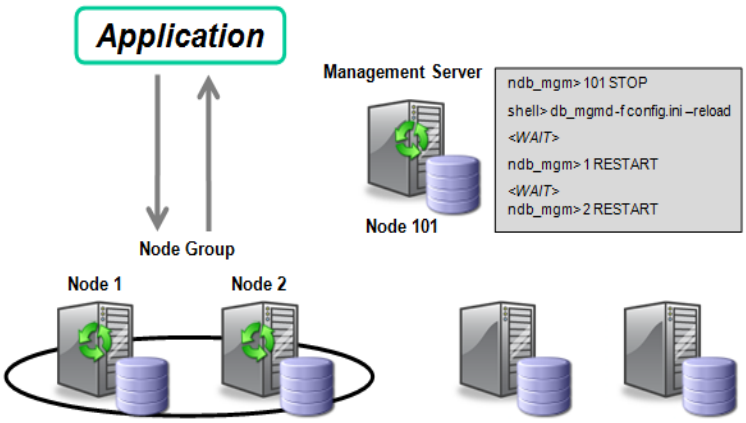


Figure 18: ***Perform a rolling restart***

Step 3: Create the new node group

You are now ready to start the 2 new data nodes. You can then create the node group from them and so include them in the Cluster, as shown in Figure 19.

Note that in this case, there is no need to wait for Node 3 to start before starting Node 4 but you must wait for both to complete before creating the node group.

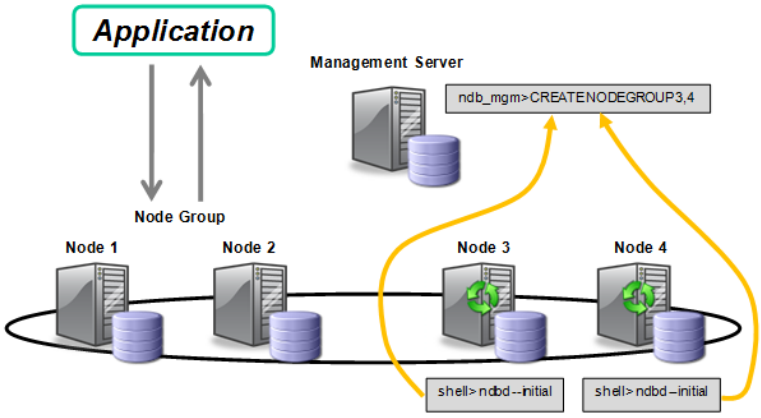


Figure 19: ***Start the new data nodes and add to Cluster***

Step 4: Repartition Data

At this point the new data nodes are part of the Cluster, but all of the data is still held in the original node group (Node 1 and Node 2). Note that once the new nodes are added as part of a new node group, new tables will automatically be partitioned across all nodes.

Figure 20 illustrates the repartitioning of the table data (disk or memory) when the command is issued from a MySQL Server. Note that the command should be repeated for each table that you want to be repartitioned. As a final step, the OPTIMIZE SQL command can be used to recover the unused spaces from the repartitioned tables.

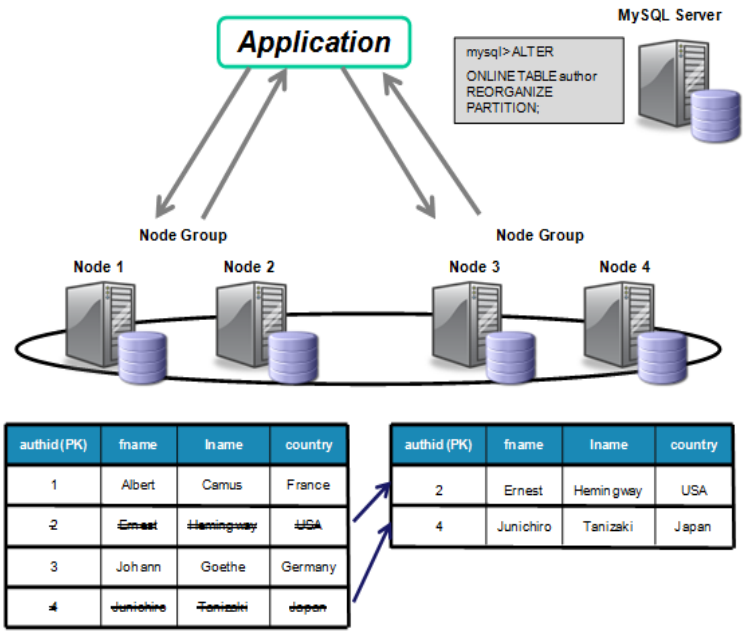


Figure 20: ***Data Repartitioned across node groups***

6. MySQL Cluster DBT2 Performance Benchmark

As discussed earlier in this paper, MySQL Cluster employs a parallel server architecture with multiple active master nodes. This ensures transactions (both reads and writes) can be load balanced and automatically scaled across multiple SQL servers simultaneously, with each SQL node able to access and update data across any node in the cluster.

MySQL Cluster also offers a flexible architecture with the ability to store both indexes and data in-memory or data on disk. As a result of this in-memory characteristic, MySQL Cluster is able to limit disk-based I/O bottlenecks by asynchronously writing transaction logs to disk, therefore maintaining real-time performance.

To demonstrate the capabilities of its parallel, real-time design, MySQL Cluster was recently benchmarked using the DBT2 test suite.

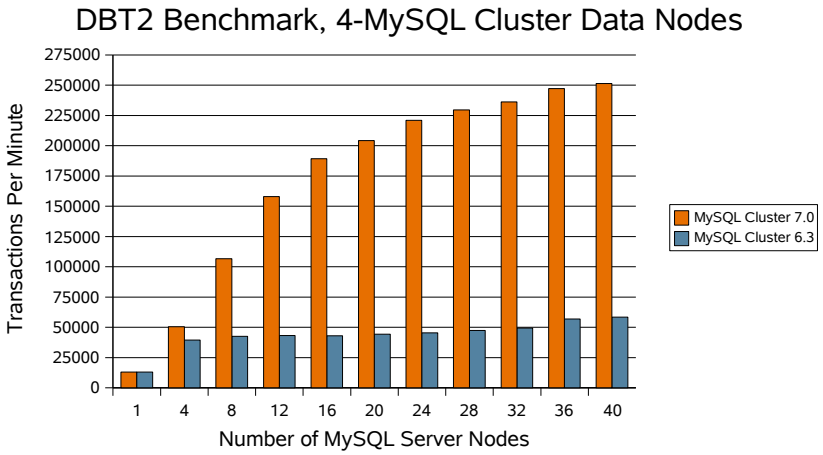


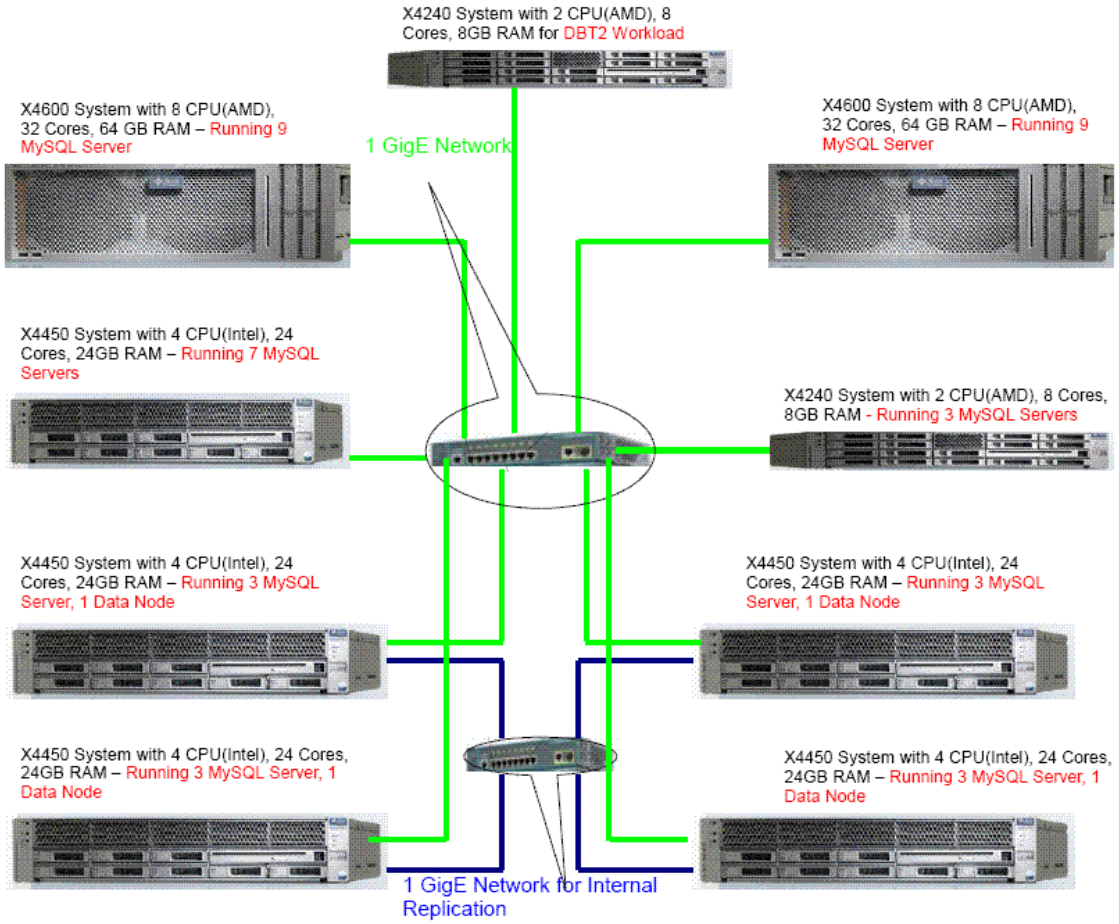
Figure 21: ***MySQL Cluster achieves over 250k TPM, or 125k Operations per Second with Average Latency of just 1.5ms***

Running DBT2, MySQL Cluster achieved 251,000 transactions per minute with just four data nodes[2]. Each transaction involved around 30 database operations, and so MySQL Cluster was able to sustain around 125,000 operations per second, with an average response time of just 1.5 milliseconds. This delivered performance represents a 4x improvement in scalability over previous versions of MySQL Cluster.

Note: *More than one MySQL Server node was installed on each physical server, with a number of the server instances used as load generators to the MySQL Cluster database. An alternative deployment could use multiple connections from each MySQL Server, which would have resulted in fewer MySQL Server nodes being used while sustaining the same performance levels*

[2] Read more about the benchmark here: http://blogs.sun.com/hasham/entry/mysql_cluster_7_performance_benchmark

6.1. Benchmark Topology



MySQL Cluster Benchmark Topology – 4 Data Nodes, GigE

Hardware Configuration:
x4450 Systems - 4 * Intel Xeon(106D1 family 6 model 29 step 1) Processor at 2.66GHz with total 24 cores, 24 GB RAM, 4 * 1GB GigE, 4 * 146 GB internal HDD, OpenSolaris

X4600 Systems: 8 * AMD (AuthenticAMD 100F42 family 16 model 4 step 2) processor at 2.7 GHz with total 32 cores, 64GB RAM, 4 * 1GB GigE, 4 * 146 GB internal HDD, OpenSolaris

X4240 Systems: 2 * AMD Opteron (100F23 family 16 model 2 step 3) quad core at 2.3 GHz, 8GB RAM, 4 * 1GB Gigabit Ethernet (GigE), 4 * 146 GB internal Hard Disk Drive (HDD), OpenSolaris

Notes: Each MySQL Server is running within a processor set of 3 cores. Total 40 MySQL Servers used.

Figure 22: *MySQL Cluster Benchmark Topology*

6.2. Database Test 2 (DBT-2)

DBT2 is an open source benchmark developed by OSDL (Open Source Development Labs). Significant updates to the benchmark were made to simplify its ability to run with a clustered database such as MySQL Cluster Carrier Grade Edition. DBT2 simulates a typical OLTP (Online Transaction Processing) application that performs five distinct transaction types.

For the benchmark, DBT2 and MySQL Cluster were configured as an “in memory” database to simulate typical “real-time” database configurations. It should be noted that performance results are being measured as new-order transactions per minute (TPM). The changes made to the DBT2 benchmark are documented and can be found on the SourceForge page for DBT2 downloads. It can also be downloaded from www.iclaustron.com.

7. Using the MySQL Pluggable Storage Engine Architecture to Meet Diverse Application Needs

Many web-based or communications services are highly modular, with each module providing specific functionality. For example, a Location Based Service (LBS) will include a range of modules to handle specific tasks:

- Management of subscriber data & contact lists
- Processing of location updates and presence status
- Mapping data management and proximity processing
- Mobile advertising and commerce, including recommendation engines based on preferences
- Activity logging for CDRs

Some of the modules described above are ideal for database engines that are optimized for primary key access, while others demand range queries, scans and complex, multi-table joins. In some instances, using MySQL database storage engines that are designed for each of these requirements provides the optimum solution to meet the performance and scaling expectations demanded by the application.

MySQL users have long benefited from the MySQL pluggable storage engine architecture that allows a developer or DBA to select a specialized storage engine for a particular application need. A MySQL storage engine is a low-level engine that manages the storing and retrieval of data, and can be accessed through a MySQL Server instance, and in some situations, directly by an application. For example, MySQL Cluster is implemented as a MySQL storage engine. The use of the storage engine is transparent to both the application and the user.

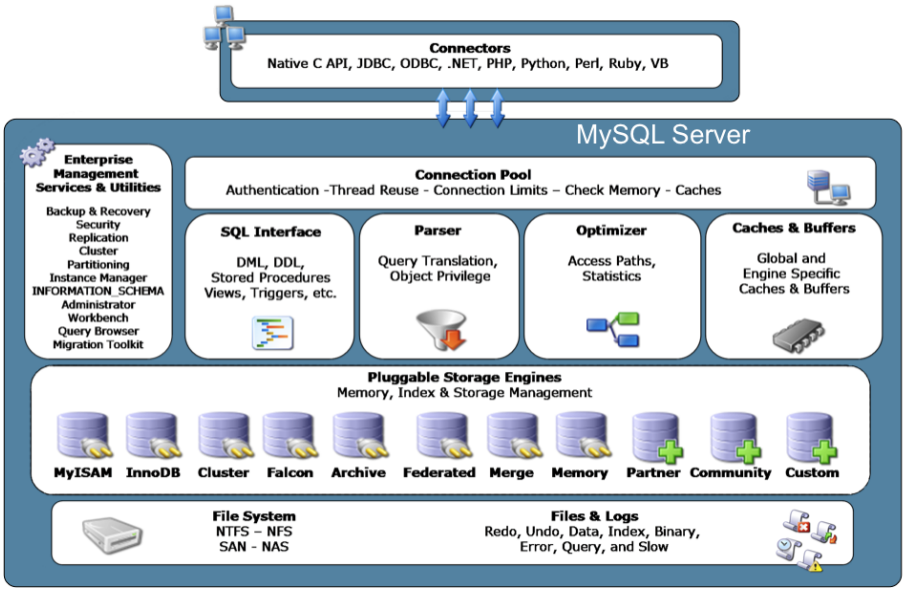


Figure 23: ***MySQL's Pluggable Storage Engine Architecture Enables Unrivaled Database Flexibility***

The flexibility of MySQL extends to being able to select a specific storage engine for each table accessed by an application. Therefore, using the LBS example above, MySQL Cluster can serve as the master database used for the management of user data, location updates, activity logging, etc. while the processing of proximity requests and recommendations reporting can be offloaded to MySQL Server slaves using another MySQL storage engine, such as MyISAM.

MySQL Cluster's Geographic Replication capability can be used to replicate real-time data, typically within a second of commit, between tables managed by MySQL Cluster to MyISAM tables (possibly running on a different host or site). In the LBS example, location and presence data is replicated by the MySQL Server from the MySQL Cluster storage engine to an InnoDB or MyISAM storage engine acting as a slave, against which complex queries involving range scans and multi-level joins can be run.[3]

[3]Read the blog to learn how to deploy this type of replication scenario

Using this type of functionality, MySQL Cluster can be used to handle write-intensive workloads demanding the highest levels of scalability and real-time performance, even though modules of the service itself demand the ability to run complex queries involving operations that may not be suitable for MySQL Cluster.

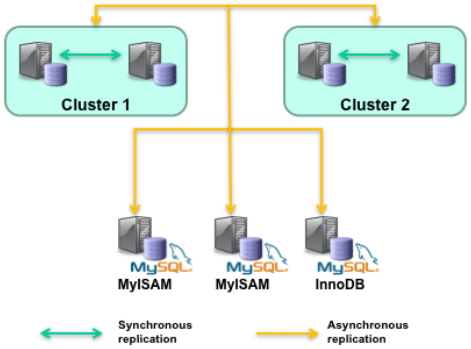


Figure 24: ***Geographic Replication Enables Data to be Replicated to Alternative Storage Engines, and to Remote Sites for Disaster Recovery***

8. Resources to Learn More

The Professional Services team at MySQL has extensive experience in achieving the best possible performance from MySQL Cluster. They can be engaged through a packaged consulting service or as a custom consulting project. Details can be found at http://www.mysql.com/consulting/

MySQL Cluster Gold support includes performance tuning, see http://www.mysql.com/products/database/cluster/support.html for details.

MySQL Cluster is a fast developing database and there are a number of Blogs that provide information on these in real-time. A list is provided at http://www.mysql.com/products/database/cluster/resources.html

Full documentation for MySQL Cluster is published at http://dev.mysql.com/doc/#cluster

There is an active MySQL Cluster forum where you can seek advice from the experts at http://forums.mysql.com/list.php?25. Alternatively you can use the cluster@lists.mysql.com mailing list.

9. Conclusion

Since its release in 2004, MySQL Cluster has continued to evolve to meet the demands of workloads demanding extreme levels of performance, scalability and 99.999% uptime.

With its distributed design, MySQL Cluster is optimized for workloads which comprise mainly primary key access. As this paper demonstrates, however, optimizing schemas and queries, tuning parameters and exploiting distribution awareness in applications enables MySQL Cluster to serve an ever-broader portfolio of application requirements.

Continual enhancements in both the core database technology along with real-time monitoring and reporting capabilities will enable database developers and administrators to extend their use-cases for MySQL Cluster.

> *"MySQL Cluster has enabled us to meet our demands for scalability, performance and continuous uptime, at a much lower cost than proprietary technologies. We would be dead in the water without it".*
>
> **Richard McCluskey, Senior Engineer, go2 Media**

To learn more about the MySQL Cluster database, check out the resources on www.mysql.com/cluster