# Optimizing Performance of the MySQL Cluster Database

*A MySQL® White Paper*

July 2012

Table of Contents

## 1. Introduction

Data volumes are exploding – driven by increasing internet penetration rates, social networking, high-speed mobile broadband connecting ever smarter devices and new Machine to Machine (M2M) interactions.

The databases needed to support this massive growth in data have to meet new challenges, including:

- **Scaling write operations**, not just reads, across commodity hardware;
- **Low latency** for a real-time user experience;
- **24 x 7 availability** for continuous service uptime;
- **Reducing barriers to entry**, enabling developers to quickly launch and evolve new, innovative services.

Many new applications need the back-end database to meet the above challenges, while still:

- Preserving transactional integrity with ACID compliance;
- Enabling deep insight by running complex, ad-hoc queries against the data;
- Leveraging the proven benefits of industry standards and skillsets to reduce cost, risk and complexity.

If your workloads have these demands, it is time to consider MySQL Cluster.

MySQL Cluster is a write-scalable, real-time, ACID-compliant transactional database, combining 99.999% availability with the low TCO of open source. Designed around a distributed, multi-master architecture with no single point of failure, MySQL Cluster scales horizontally on commodity hardware with auto-sharding to serve read and write intensive workloads, accessed via SQL and NoSQL interfaces.

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, ATT, Shopatron, Telenor, UTStarcom, the United States Navy and Zillow.com.

It is important to recognize that as a fully-distributed, auto-sharded (partitioned) database, there are classes of application 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 by summarizing additional resources that will enable you to optimize MySQL Cluster performance.

## 2. Identifying Optimal Applications for MySQL Cluster

MySQL Cluster is implemented as an active/active, multi-master database ensuring updates made by any application or SQL node are instantly available to all of the other nodes accessing the cluster.

Tables are automatically sharded (partitioned) across a pool of low cost commodity data nodes, enabling the database to scale horizontally to serve read and write-intensive workloads, accessed both from SQL and directly via NoSQL APIs.

By automatically sharding tables at the database layer, MySQL Cluster eliminates the need to shard at the application layer, greatly simplifying application development and maintenance. Sharding is entirely transparent to the application which is able to connect to any node in the cluster and have queries automatically access the correct shards needed to satisfy a query or commit a transaction

*"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***

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.

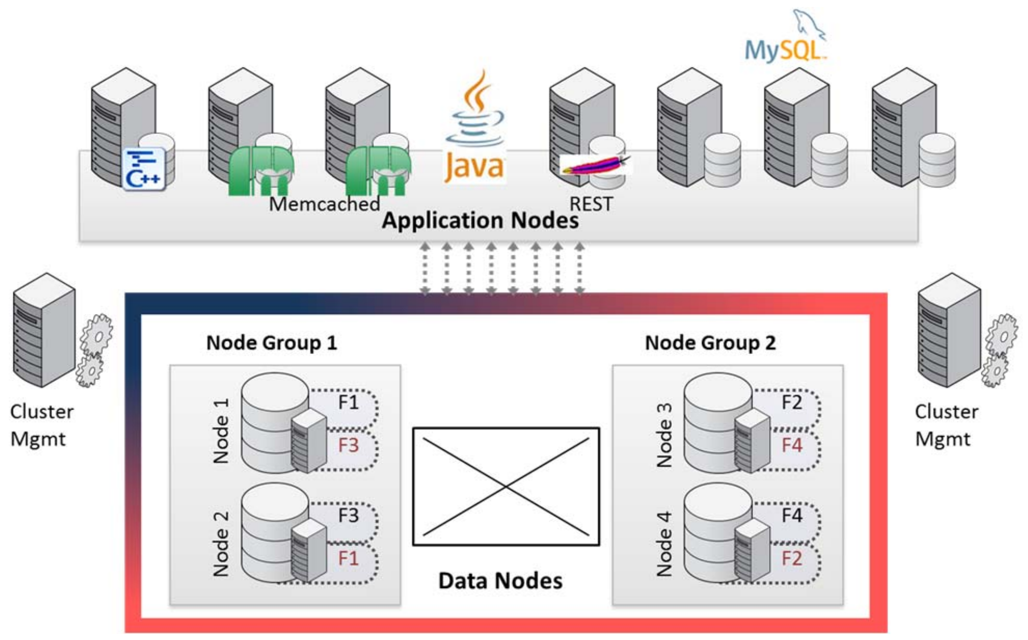


**Figure 1 MySQL Cluster provides a multi-master database with a parallel architecture**

To learn more about the MySQL Cluster architecture, refer to the "Scaling Web Databases Guide" from http://mysql.com/why-mysql/white-papers/mysql_wp_scaling_web_databases.php

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

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

- High volume OLTP
- Ecommerce, inventory management, shopping carts, payment processing, fulfillment tracking, etc;
- Payment gateways and trading systems
- Mobile and micro-payments
- Session management & caching
- Feed streaming, analysis and recommendations
- Digital content management
- Massively Multiplayer Online Games
- Subscriber (i.e. HLR & HSS) / user profile management and entitlements
- Communications and presence services
- Service Delivery Platforms
- IP Multimedia (IMS) services
- DNS / DHCP for broadband access
- VoIP, IPTV, Video on Demand, Video Conferencing

To determine whether MySQL Cluster is the right database for your new project, it is worth downloading the MySQL Cluster Evaluation Guide. This also includes best practices for conducting the evaluation:
http://mysql.com/why-mysql/white-papers/mysql_cluster_eval_guide.php

If you are considering using either InnoDB or MySQL Cluster, the comparison sections in the documentation can help define the application attributes that are best suited to each MySQL storage engine:
http://dev.mysql.com/doc/refman/5.5/en/mysql-cluster-compared.html

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.

## 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. mysqlslap is typically used to generate random traffic but to make the tests repeatable, the best approach is to create one file 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.name='Bill'  AND
  sub_name.sub_id=sub_age.sub_id;
```

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).  A trial version of MySQL Enterprise Monitor can be obtained at the following URL: https://edelivery.oracle.com/

The Query Analyzer cannot examine any database operations performed through the NDB API or any intermediary database access layer (such as MySQL Cluster's ClusterJ Java API that uses the NDB API).

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 sub_age.age FROM sub_name, sub_age WHERE sub_name.name='Bill'  AND
   sub_name.sub_id=sub_age.sub_id;
```

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. Note also that the variable is local to the MySQL Server and so it needs to be set on each MySQL Server in the cluster separately.

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!
```

Finally, MySQL Cluster provides information on what's happening within the data node – this data is exposed through a virtual database called `NDBINFO`. As an example, NDBINFO presents information on the use of Disk Page Buffer. The Disk Page Buffer is a cache on each data node which is used when using disk-based tables. Like any cache, the higher the hit rate the better the performance. Tuning the size of this cache can have a significant effect on your system. The data exposed for the Disk Page Buffer can be accessed directly from the `mysql` command line in order to calculate the cache hit ratio:

```
mysql> SELECT node_id, page_requests_direct_return AS hit,
 page_requests_wait_io AS miss,  100*page_requests_direct_return/
 (page_requests_direct_return+page_requests_wait_io) AS hit_rate
  FROM ndbinfo.diskpagebuffer;

+---------+------+------+----------+
| node_id | hit  | miss | hit_rate |
+---------+------+------+----------+
| 3       | 6    | 3    | 66.6667  |
| 4       | 10   | 3    | 76.9231  |
+---------+------+------+----------+
```

Note that this hit-rate is the **average** over the total period since the data nodes were last restarted. When attempting to tune the size of the cache (through the `DiskPageBufferMemory` parameter[2]) what you really need to see is the **current** cache hit-ratio; you can do this manually by taking deltas of the values or MySQL Enterprise Monitor does it automatically and presents the information is a graph over time[3].

The full contents of the NDBINFO database is described in the MySQL Cluster documentation at http://dev.mysql.com/doc/refman/5.5/en/mysql-cluster-ndbinfo.html

# 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 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 2. 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.

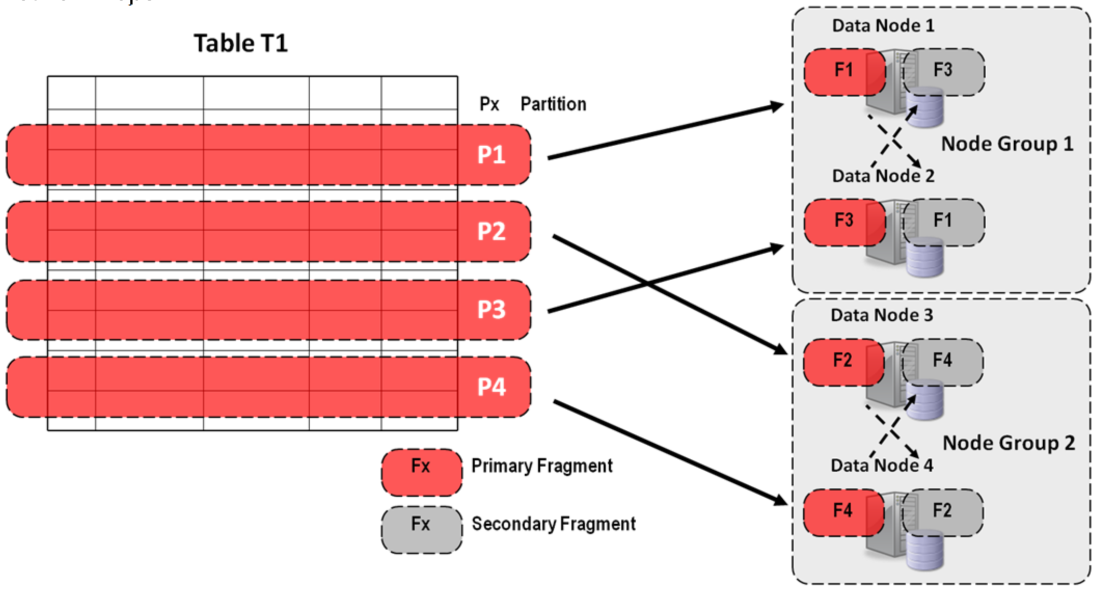


**Figure 2 Partitioning of MySQL Cluster data**

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 4.3 "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).

[2] http://dev.mysql.com/doc/refman/5.5/en/mysql-cluster-ndbd-definition.html#ndbparam-ndbd-diskpagebuffermemory
[3] http://www.clusterdb.com/mysql-cluster/further-mysql-cluster-additions-to-mysql-enterprise-monitor/

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 (µs) | Min cost (µs) | 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 4.3 "Distribution aware applications" explains how that type of partition pruning can be achieved.

MySQL Cluster supports table Joins but prior to MySQL Cluster 7.2 the performance could be poor if the depth of the join or the size of the result sets were too large. This is because joins were implemented as nested loop Joins within the MySQL Server; as the MySQL Server calculates the results at each level in the Join it fetches data one-row at a time from the data nodes – each of those network trips adds to the latency. This is illustrated in Figure 3.

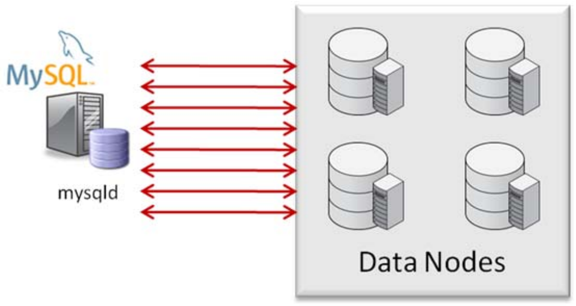


**Figure 3 Network hops from Joins**

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.

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 4 shows how this query is executed (prior to MySQL Cluster 7.2). 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 result 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).

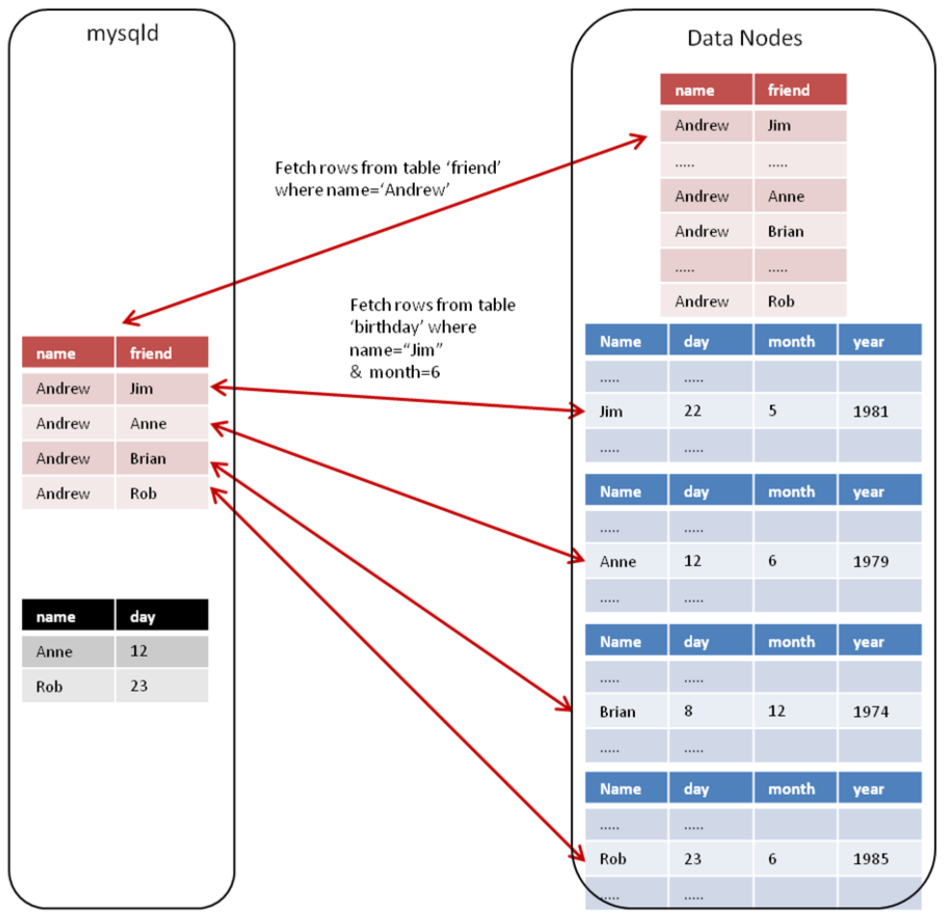


**Figure 4 Complexity of non-pushed table Joins**

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.

MySQL Cluster 7.2 introduced a much more efficient option for complex Joins – Adaptive Query Localization (AQL) and section 4.2 "Using Adaptive Query Localization for complex Joins" describes how the join should be structured in order to get the benefits. The performance improvements from using AQL can be very dramatic.

The AQL functionality ships queries from the MySQL Server to the data nodes where the query executes on local copies of the data in parallel, and then returns the merged result set back to the MySQL Server, significantly enhancing performance by reducing network trips.

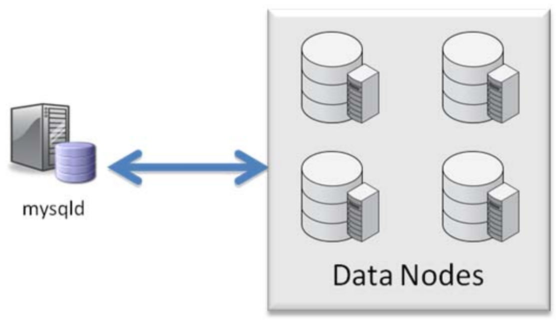


**Figure 5 AQL Reduces network trips for Joins**

AQL enables MySQL Cluster to better serve those use-cases that have the need to run complex queries along with high throughput OLTP operations in one application and database solution or where multiple applications need access to the same data but have different access/query requirements.

In addition, MySQL Cluster now provides the MySQL Server with better information on the available indexes which allows the MySQL optimizer to automatically produce better query execution plans. Previously it was up to the user to provide hints to the optimizer. To get the best results, run `OPTIMIZE TABLE <tab-name>` on one of the MySQL Servers whenever you change the tables schema, add/remove an index or make very significant changes its content.

Real world testing has demonstrated typical performance gains of 70x across a range of queries. One example comes from an online content store and management system that involved a complex query joining 11 tables. This query took over 87 seconds to complete with MySQL Cluster 7.1, and just 1.2 seconds when tested with MySQL Cluster 7.2 as show in Figure 6 (note that this test was run on very low-end hardware).

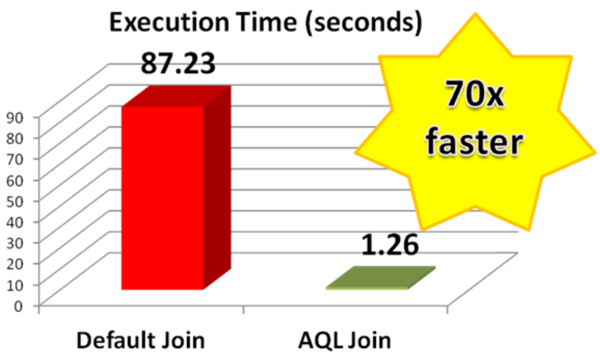


**Figure 6 Benefits of AQL Joins**

Figure 7 illustrates the real customer Join which was used to validate the 70x speed-up from AQL; more details on the actual query, test environment and the full results can be found at [http://www.clusterdb.com/mysql-cluster/70x-faster-joins-with-aql-in-mysql-cluster-7-2-dmr/](http://www.clusterdb.com/mysql-cluster/70x-faster-joins-with-aql-in-mysql-cluster-7-2-dmr/)

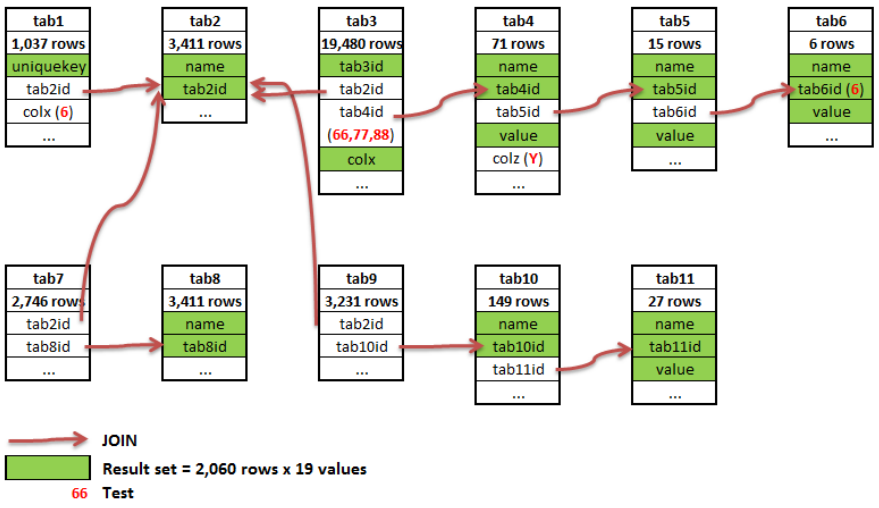


**Figure 7 Join resulting in 70x AQL speed-up**

Even with AQL there may be operations within your application that are not suited to MySQL Cluster; if they make up the majority of your high-running queries/updates then MySQL Cluster might not be the best MySQL Storage Engine for your application but there are a couple of hybrid approaches that can be adopted which enable you to play to the strengths of each different storage engines:

1. Take advantage of MySQL's pluggable Storage Architecture; create the tables that are best suited to MySQL Cluster with the `engine=ndbcluster` option but use `engine=innodb` for the data which is better suited to InnoDB.

2. Hold the real-time copy of the data in MySQL Cluster and use MySQL asynchronous Replication to keep a separate, InnoDB database up to date in near-real-time and perform the appropriate queries on that copy.

The MySQL Cluster Evaluation white paper (available from [http://www.mysql.com/why-mysql/white-papers/mysql_cluster_eval_guide.php](http://www.mysql.com/why-mysql/white-papers/mysql_cluster_eval_guide.php)) gives more guidance as to when it is better to use MySQL Cluster vs. InnoDB for your data.

## 4.2. *Using Adaptive Query Localization for complex Joins*

The use of AQL is controlled by a global variable - `ndb_join_pushdown` – which is on by default.
In order for a join to be able to exploit AQL (in other words be "pushed down" to the data nodes), it must meet the following conditions:

1. Any columns to be joined must use exactly the same data type. (For example, if an `INT` and a `BIGINT` column are joined, the join cannot be pushed down). This includes the lengths of any `VARCHAR` columns.
2. Joins referencing `BLOB` or `TEXT` columns will not be pushed down.
3. Explicit locking is not supported; however, the NDB storage engine's characteristic implicit row-based locking is enforced.
4. In order for a join to be pushed down, child tables in the Join must be accessed using one of the `ref`, `eq_ref`, or `const` access methods, or some combination of these methods. These access methods are described in [http://dev.mysql.com/doc/refman/5.5/en/explain-output.html](http://dev.mysql.com/doc/refman/5.5/en/explain-output.html)
5. Joins referencing tables explicitly partitioned by [`LINEAR`] `HASH`, `LIST`, or `RANGE` currently cannot be pushed down
6. If the query plan decide to `'Using join buffer'` for a candidate child table, that table cannot be pushed as child. However, it might be the root of another set of pushed tables.
7. If the root of the pushed Join is an `eq_ref` or `const`, only child tables joined by `eq_ref` can be appended. (A `ref` joined table will then likely become a root of another pushed Join)

These conditions should be considered when designing your schema and application queries – for example, to comply with constraint 4, attempt to make any table scan that is part of the Join be the first clause.

Where a query involves multiple levels of Joins, it is perfectly possible for some levels to be pushed down while others continue to be executed within the MySQL Server.

You can check whether a given join can be pushed down by checking it with EXPLAIN; when the join can be pushed down, you can see references to the pushed join in the Extra section of the output; for example:

```
mysql> EXPLAIN
    ->     SELECT e.first_name, e.last_name, t.title, d.dept_name
    ->         FROM employees e
    ->         JOIN dept_emp de ON e.emp_no=de.emp_no
    ->         JOIN departments d ON d.dept_no=de.dept_no
    ->         JOIN titles t ON e.emp_no=t.emp_no\G
*************************** 1. row ***************************
           id: 1
  select_type: SIMPLE
        table: d
         type: ALL
possible_keys: PRIMARY
          key: NULL
      key_len: NULL
          ref: NULL
         rows: 9
        Extra: Parent of 4 pushed join@1
*************************** 2. row ***************************
           id: 1
  select_type: SIMPLE
        table: de
         type: ref
possible_keys: PRIMARY,emp_no,dept_no
          key: dept_no
      key_len: 4
          ref: employees.d.dept_no
         rows: 5305
        Extra: Child of 'd' in pushed join@1
*************************** 3. row ***************************
           id: 1
  select_type: SIMPLE
        table: e
         type: eq_ref
possible_keys: PRIMARY
          key: PRIMARY
      key_len: 4
          ref: employees.de.emp_no
         rows: 1
        Extra: Child of 'de' in pushed join@1
*************************** 4. row ***************************
           id: 1
  select_type: SIMPLE
        table: t
         type: ref
possible_keys: PRIMARY,emp_no
          key: emp_no
      key_len: 4
          ref: employees.de.emp_no
         rows: 19
        Extra: Child of 'e' in pushed join@1
```

If `EXPLAIN` shows that your join is not being pushed down to the data nodes and you want to understand why then run `EXPLAIN EXTENDED <query>;` followed by `SHOW WARNINGS;`

To provide an aggregated summary of how often AQL is able to be exploited, a number of new entries are added to the counters table in the NDBINFO database. These counters provide an insight into how frequently the AQL functionality is able to be used and can be queried using SQL:

```
mysql> SELECT node_id, counter_name, val FROM ndbinfo.counters WHERE block_name=
  "DBSPJ";
+---------+-----------------------------------+------+
| node_id | counter_name                      | val  |
+---------+-----------------------------------+------+
|       2 | READS_RECEIVED                    |    0 |
|       2 | LOCAL_READS_SENT                  |    0 |
|       2 | REMOTE_READS_SENT                 |    0 |
|       2 | READS_NOT_FOUND                   |    0 |
|       2 | TABLE_SCANS_RECEIVED              |    0 |
|       2 | LOCAL_TABLE_SCANS_SENT            |    0 |
|       2 | RANGE_SCANS_RECEIVED              |    0 |
|       2 | LOCAL_RANGE_SCANS_SENT            |    0 |
|       2 | REMOTE_RANGE_SCANS_SENT           |    0 |
|       2 | SCAN_BATCHES_RETURNED             |    0 |
|       2 | SCAN_ROWS_RETURNED                |    0 |
|       2 | PRUNED_RANGE_SCANS_RECEIVED       |    0 |
|       2 | CONST_PRUNED_RANGE_SCANS_RECEIVED |    0 |
|       3 | READS_RECEIVED                    |    0 |
|       3 | LOCAL_READS_SENT                  |    0 |
|       3 | REMOTE_READS_SENT                 |    0 |
|        3 | READS_NOT_FOUND                  |     0 |
|        3 | TABLE_SCANS_RECEIVED             |     0 |
|        3 | LOCAL_TABLE_SCANS_SENT           |     0 |
|        3 | RANGE_SCANS_RECEIVED             |     0 |
|        3 | LOCAL_RANGE_SCANS_SENT           |     0 |
|        3 | REMOTE_RANGE_SCANS_SENT          |     0 |
|        3 | SCAN_BATCHES_RETURNED            |     0 |
|        3 | SCAN_ROWS_RETURNED               |     0 |
|        3 | PRUNED_RANGE_SCANS_RECEIVED      |     0 |
|        3 | CONST_PRUNED_RANGE_SCANS_RECEIVED |     0 |
+----------+-----------------------------------+-------+
```

Note that `ANALYZE TABLE` should be run once for each table in order to enable the MySQL optimizer to choose the best execution plan – this is especially important when using AQL. It only needs to be run on one MySQL Server but repeat whenever you change the tables schema, add/remove an index or make very significant changes its content.

## *4.3. 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 2). 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 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 many transactions 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 (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    |
+-----------------------+-------+

mysql> 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 latency 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 8 shows how partition pruning (red bars) reduces latency for smaller result sets, but can actually increase it for larger result sets. Note that shorter bars/lower latency represents faster response times.

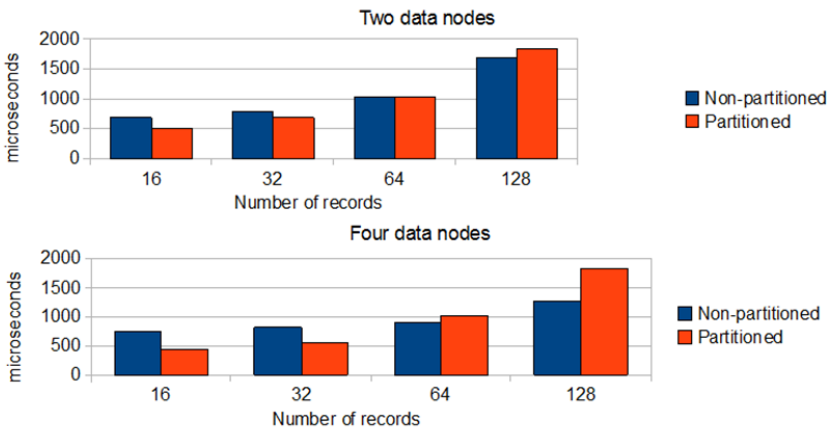


**Figure 8 Effects of index scan partition pruning**

Making your application distribution-aware in this way can be critical in enabling throughput to scale linearly as extra data nodes are added.

## *4.4. 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 |
+--------+--------------+--------------+

mysql> mysql> SELECT * FROM services WHERE sub_id=1 AND service_name='ssh';
+--------+--------------+--------------+
| sub_id | service_name | service_parm |
+--------+--------------+--------------+
|      1 | ssh          |          666 |
+--------+--------------+--------------+

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.

## 4.5. *Schema optimizations*

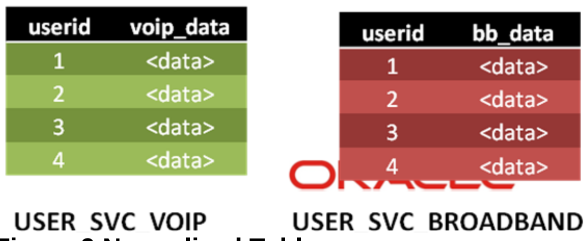
There are two main ways to modify your data schema to get the best performance: use the most efficient types and denormalize your schema where appropriate.

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 14K bytes.

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

Figure 9 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,
```



| 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 9 Normalized Tables

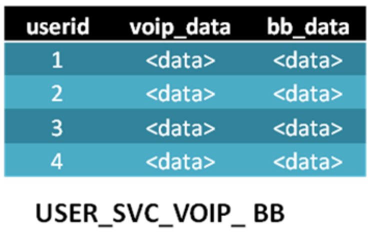
```
   USER_SVC_VOIP AS voip WHERE bb.userid=voip.userid
   AND bb.userid=1;
```

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

```
   mysql> 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

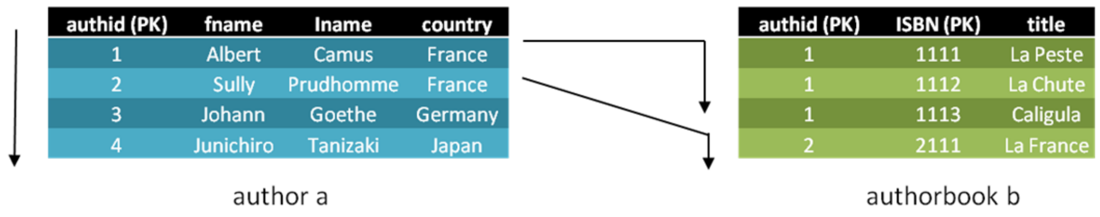
**Figure 10 De-normalized Table**

### 4.6. *Query optimization*

As explained in section 4.1, Joins can be expensive in MySQL Cluster unless you are able to take advantage of AQL (see section 4.2). This section works through an example to demonstrate how a (non-AQL'd) join can quickly become very expensive and then how a little knowledge of how the 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 11 and perform the join as shown:

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



| authid (PK) | fname | lname | country |
|---|---|---|---|
| 1 | Albert | Camus | France |
| 2 | Sully | Prudhomme | France |
| 3 | Johann | Goethe | Germany |
| 4 | Junichiro | Tanizaki | Japan |

author a

| authid (PK) | ISBN (PK) | title |
|---|---|---|
| 1 | 1111 | La Peste |
| 1 | 1112 | La Chute |
| 1 | 1113 | Caligula |
| 2 | 2111 | La France |

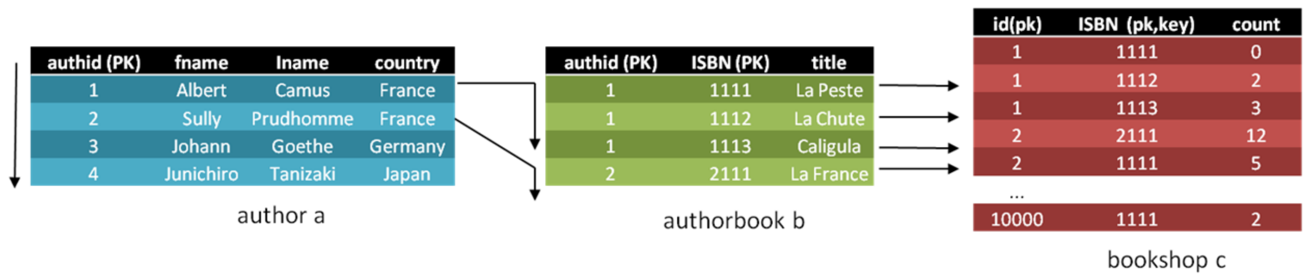
authorbook b

**Figure 11 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 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 12:

```
   SELECT c.id from author a, authorbook b, bookshop c WHERE c.ISBN=b.ISBN AND
     b.authid=a.authid AND a.country='FRANCE' AND c.count>0;
```



author a

| authid (PK) | fname | lname | country |
|---|---|---|---|
| 1 | Albert | Camus | France |
| 2 | Sully | Prudhomme | France |
| 3 | Johann | Goethe | Germany |
| 4 | Junichiro | Tanizaki | Japan |

authorbook b

| authid (PK) | ISBN (PK) | title |
|---|---|---|
| 1 | 1111 | La Peste |
| 1 | 1112 | La Chute |
| 1 | 1113 | Caligula |
| 2 | 2111 | La France |

bookshop c

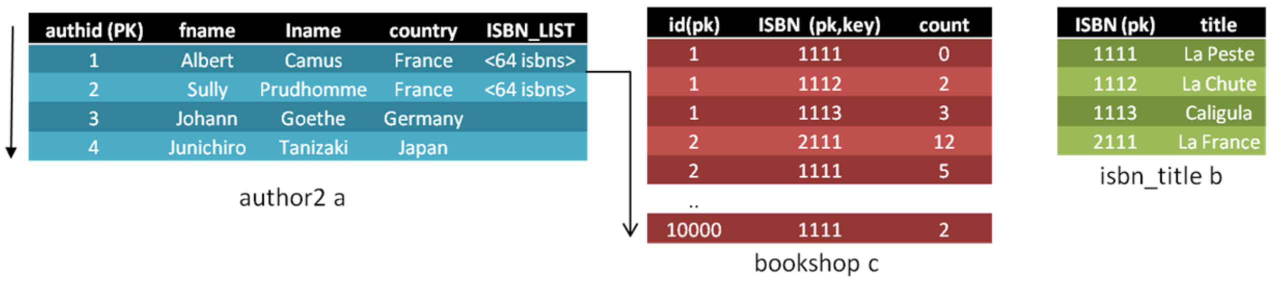
| id(pk) | ISBN (pk,key) | count |
|---|---|---|
| 1 | 1111 | 0 |
| 1 | 1112 | 2 |
| 1 | 1113 | 3 |
| 2 | 2111 | 12 |
| 2 | 1111 | 5 |
| … | | |
| 10000 | 1111 | 2 |

**Figure 12 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 13:

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



author2 a

| authid (PK) | fname | lname | country | ISBN_LIST |
|---|---|---|---|---|
| 1 | Albert | Camus | France | <64 isbns> |
| 2 | Sully | Prudhomme | France | <64 isbns> |
| 3 | Johann | Goethe | Germany | |
| 4 | Junichiro | Tanizaki | Japan | |

bookshop c

| id(pk) | ISBN (pk,key) | count |
|---|---|---|
| 1 | 1111 | 0 |
| 1 | 1112 | 2 |
| 1 | 1113 | 3 |
| 2 | 2111 | 12 |
| 2 | 1111 | 5 |
| .. | | |
| 10000 | 1111 | 2 |

isbn_title b

| ISBN (pk) | title |
|---|---|
| 1111 | La Peste |
| 1112 | La Chute |
| 1113 | Caligula |
| 2111 | La France |

**Figure 13 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 or use AQL 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. Remember to run `OPTIMIZE TABLE <tab-name>;` after adding a new index!

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
+----+-------------+-------+------+---------------+-------------+---------+-------------+------+----------------
```

Note that from MySQL Cluster 7.2 onwards, the `ANALYZE TABLE` command applies to MySQL Cluster tables and once that has been run, the MySQL optimizer will automatically make much better use of the available indexes and so there will be fewer occasions when it is necessary to force its hand.

## 4.7. 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[4].

---

[4] http://dev.mysql.com/doc/refman/5.5/en/mysql-cluster-params-overview.html

### 4.8. Connection pools

MySQL Cluster delivers the best throughput when the data nodes are served with a large numbers of operations in parallel. To that end you should use multiple MySQL Servers together with multiple application threads accessing each of those MySQL Servers.

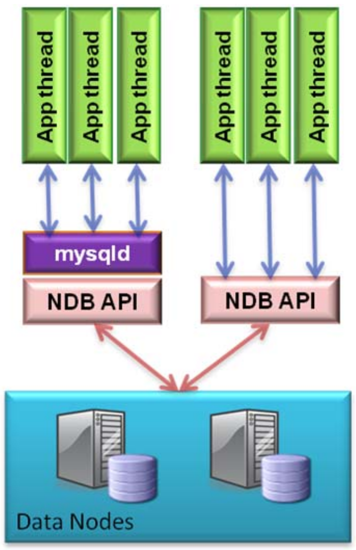


**Figure 14 API Thread Contention**

To access the data nodes, the `mysqld` process by default uses a single NDB API connection. Because there are multiple application 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 4.10 “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 and application complexity.

A more effective solution is to have multiple NDB API connections from the `mysqld` process to the data nodes as shown in Figure 15.

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 1 in `my.cnf`:

**config.ini**:

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

[ndbd]
hostname=localhost
NodeId=3

[ndbd]
hostname=localhost
NodeId=4

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

[mysqld]
hostname=localhost
NodeId=101
#optional to define the id

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

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

[api]
# do not specify NodeId!
```

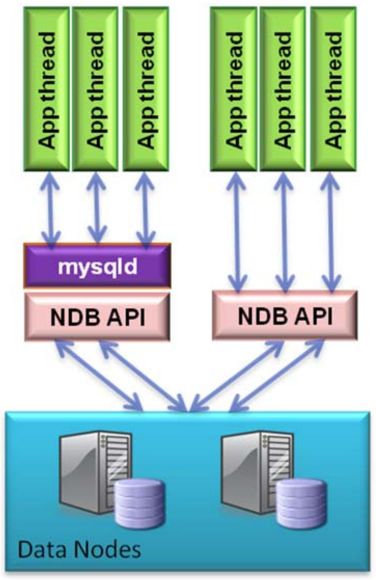


**Figure 15 Multiple Connection pool**

```
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.

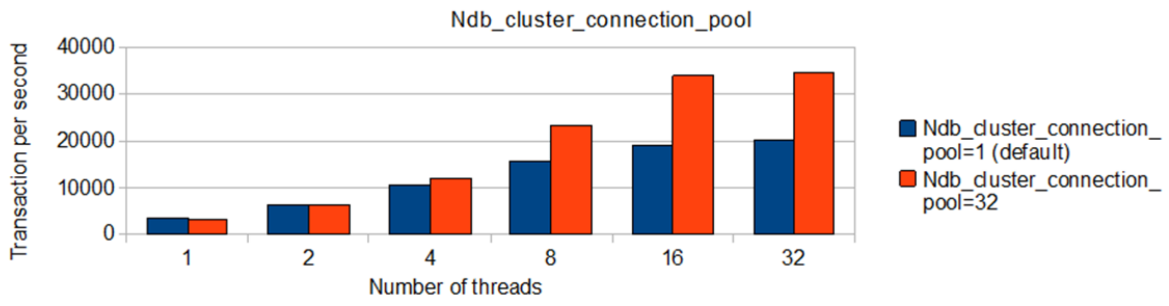


**Figure 16 Benefits of connection pooling**

Figure 16 shows the performance benefits of using connection pooling. Applications would typically experience a 70% improvement but in some cases it can exceed 150%.

## *4.9. 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. 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.

`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 (as an aside – that process can be automated if you are using MySQL Cluster Manager[5]). This all makes it extremely simple for developers or administrators to switch to the multi-threaded version.

**Optimal thread configuration for Multi-Threaded Data Nodes**

The configuration of data node threads can be managed in two ways via the config.ini file:

[5] http://www.mysql.com/products/cluster/mcm/

- Simply set `MaxNoOfExecutionThreads` to the appropriate number of threads to be run in the data node, based on the number of threads presented by the processors used in the host or VM.
- Use the new `ThreadConfig` variable that enables users to configure both the number of each thread type to use and also which CPUs to bind them too.

The flexible configuration afforded by the multi-threaded data node enhancements means that it is possible to optimize data nodes to use anything from a single CPU/thread up to a 48 CPU/thread server. Co-locating the MySQL Server with a single data node can fully utilize servers with 64 – 80 CPU/threads. It is also possible to co-locate multiple data nodes per server, but this is now only required for very large servers with 4+ CPU sockets equipped with dense multi-core processors.

An example of how to make best use of a 24 CPU/thread server box is to configure the following:

- 8 `ldm` threads
- 4 `tc` threads
- 3 `recv` threads
- 3 `send` threads
- 1 `rep` thread for asynchronous replication.

Each of those threads should be bound to a CPU. It is possible to bind the main thread (schema management domain) and the IO threads to the same CPU in most installations.

In the configuration above, we have bound threads to 20 different CPUs. We should also protect these 20 CPUs from interrupts by using the `IRQBALANCE_BANNED_CPUS` configuration variable in `/etc/sysconfig/irqbalance` and setting it to `0x0FFFFF`.

The reason for doing this is that MySQL Cluster generates a lot of interrupt and OS kernel processing, and so it is recommended to separate activity across CPUs to ensure conflicts with the MySQL Cluster threads are eliminated.

When booting a Linux kernel it is also possible to provide an option `isolcpus=0-19` in `grub.conf`. The result is that the Linux scheduler won't use these CPUs for any task. Only by using CPU affinity syscalls can a process be made to run on those CPUs.

By using this approach, together with binding MySQL Cluster threads to specific CPUs and banning CPU's IRQ processing on these tasks, a very stable performance environment is created for the MySQL Cluster data node.

On a 32 CPU/Thread server:

- Increase the number of `ldm` threads to 12
- Increase `tc` threads to 6
- Provide 2 more CPUs for the OS and interrupts.
- The number of `send` and `recv` threads should, in most cases, still be sufficient.

On a 40 CPU/Thread server, increase `ldm` threads to 16, `tc` threads to 8 and increment `send` and `recv` threads to 4.

On a 48 CPU/Thread server it is possible to optimize further by using:

- 12 `tc` threads
- 2 more CPUs for the OS and interrupts
- Avoid using `IO` threads and `main` thread on same CPU
- Add 1 more `recv` thread.

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.

### 4.10. 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

In addition, there are a growing number of native drivers that provide access to the data held in the data nodes without needing to go through the SQL layer (i.e. the drivers directly access the NDB API).

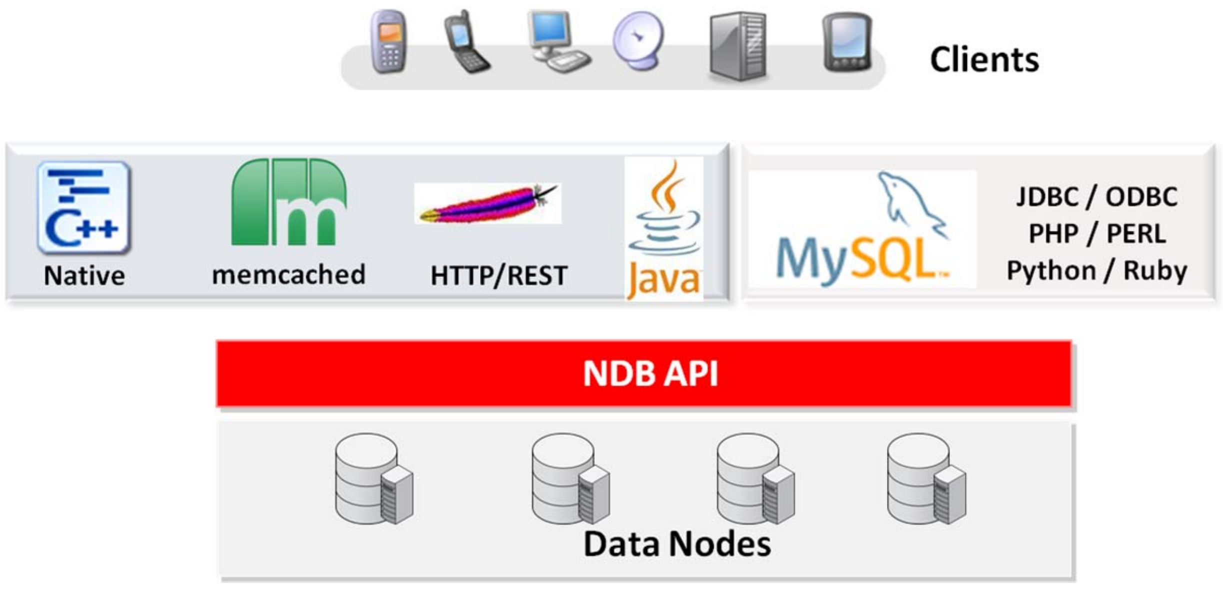


**Figure 17 Access to data through multiple APIs**

MySQL Cluster 7.2 adds a native Memcached protocol to the rich diversity of APIs available to access the database. Using the Memcached API, web services can directly access MySQL Cluster without transformations to SQL, ensuring low latency and high throughput for read/write queries. Operations such as SQL parsing are eliminated and more of the server's hardware resources (CPU, memory and I/O) are dedicated to servicing the query within the storage engine itself. These benefits are discussed in more detail in the Developer Zone article posted as follows:
http://dev.mysql.com/tech-resources/articles/nosql-to-mysql-with-memcached.html

For more information on how to make use of the Memcached API, refer to the "MySQL Cluster 7.2" white paper which can be found at http://mysql.com/why-mysql/white-papers/mysql_wp_cluster7_architecture.php and for a worked example, refer to http://www.clusterdb.com/mysql-cluster/scalabale-persistent-ha-nosql-memcache-storage-using-mysql-cluster/

In a similar way, ClusterJ provides a native Java API for enterprise applications – delivering ease of development and improved performance. More details on ClusterJ can be found in the “MySQL Cluster Connector for Java”[6] white paper.

## 4.11. 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.

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 promising results have been achieved using Solid State Device (SSD) storage, especially for the table spaces used for disk-based tables.

Refer to the MySQL Cluster Evaluation Guide[7]

## 4.12. 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 by using the `LockPagesInMainMemory`[8] configuration parameter – 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 Oracle Sun T-Series 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` or from MySQL Cluster 7.2, use the `ThreadConfig` parameter for multi-threaded data nodes.

When using auto-increment columns, it is possible to reduce the frequency of requests from a MySQL Server to the data nodes for more values by increasing the value of `ndb-auto-increment-prefetch-sz`.

As a general rule, do not enable the Query Cache in the MySQL Server. The reason for this is that if the data is modified through one MySQL Server then the caches of all of the others must be purged – this can quickly lead to a reduction in performance. Note that even for disk-based table data, the data nodes cache the most recently accessed data in memory. The exception would be if you have tables that are read-only or change very rarely; in this case add this to the `my.cnf` file:

```
query_cache_size=1000000
query_cache_type=2 # On-Demand
```

and then insert `SQL_CACHE` between the `SELECT` keyword and the rest of any query for which you wish to use the Query Cache.

By default, in-memory table data is actually logged to disk in order to provide durability – if for example you lost power to the data center then when the cluster is brought back up this data can be recovered from the disk copy. There are certain applications (such as the storing of session data) that do not require this

[6] http://www.mysql.com/why-mysql/white-papers/mysql_wp_cluster_connector_for_java.php
[7] http://www.mysql.com/why-mysql/white-papers/mysql_cluster_eval_guide.php
[8] http://dev.mysql.com/doc/refman/5.5/en/mysql-cluster-ndbd-definition.html#ndbparam-ndbd-lockpagesinmainmemory

persistence  as they can afford for the data to be lost after a complete system shutdown. For that data, you can disable the logging to disk (and hence improve performance):

```
mysql> set ndb_table_no_logging=1;
mysql> create table session_table(..) engine=ndb;
mysql> set ndb_table_no_logging=0;
```

## 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.

It is possible to add new MySQL Servers or 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).

The simplest and safest (less scope for user-error) is to use MySQL Cluster Manager (available for download and free 30 day trial from https://edelivery.oracle.com/). MySQL Cluster Manager automates the process of adding additional nodes to a running Cluster – helping ensure that there is no loss of service during the process. In this example a second MySQL Server process is added to two of the existing hosts and two new data nodes are added to each of two new hosts to produce a configuration as shown in Figure 18 below.

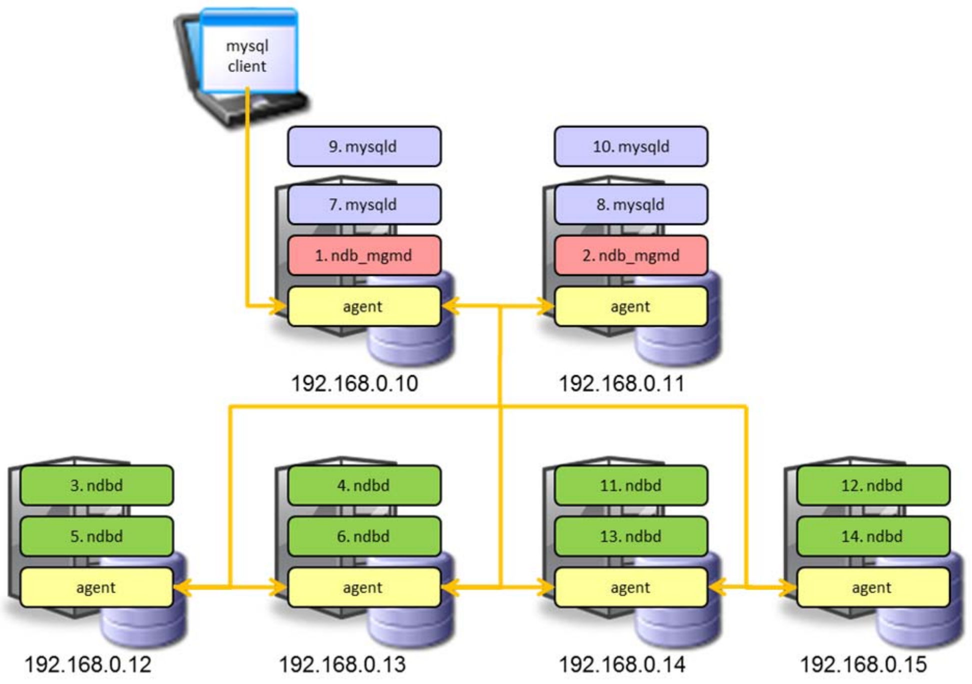
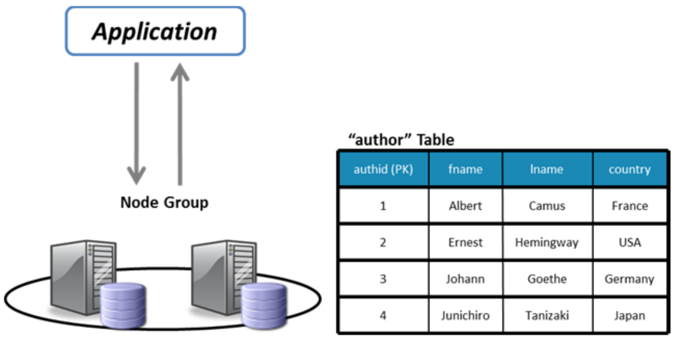


**Figure 18 On-line scaling automated by MySQL Cluster Manager**

If any of the new nodes are going to run on a new server then the first action is to install and run the MySQL Cluster Manager agent on those nodes. Once the agent is running on all of the hosts, from the MySQL Cluster Manager command line, the following steps should then be run:


Application
Node Group
"author" Table
authid (PK) | fname | lname | country
1 | Albert | Camus | France
2 | Ernest | Hemingway | USA
3 | Johann | Goethe | Germany
4 | Junichiro | Tanizaki | Japan
**Figure 19 Starting configuration**

1. Add the new hosts to the site being managed:

```
mcm> add hosts --hosts=192.168.0.14,192.168.0.15 mysite;
```

2. Register the MySQL Cluster load (package) against these new hosts:

```
mcm> add package --basedir=/usr/local/mysql_7_0_9 --hosts=192.168.0.14,192.168.0.15
   7.0.9;
```

3. Add the new nodes to the Cluster (as part of this, MySQL Cluster Manager will perform a rolling restart of the original nodes so that they become aware of the configuration change):

```
mcm> add process --
   processhosts=mysqld@192.168.0.10,mysqld@192.168.0.11,ndbd@192.168.0.14,ndbd@192.1
   68.0.15,ndbd@192.168.0.14,ndbd@192.168.0.15 -s
   port:mysqld:52=3307,port:mysqld:53=3307 mycluster;
```

4. At this point the new nodes are part of the cluster but the processes have not started; this step fixes that:

```
mcm> start process --added mycluster;
```

5. The new data nodes are now running and part of the cluster but to make full use of them, the existing Cluster tables should be repartitioned (run from the regular MySQL client connected to any of the MySQL Servers):

```
mysql> ALTER ONLINE TABLE <table-name> REORGANIZE PARTITION;
mysql> OPTIMIZE TABLE <table-name>;
```

More information on MySQL Cluster Manager can be found in "MySQL Cluster Manager - Reducing Complexity & Risk while Improving Management Efficiency" which is available from http://www.mysql.com/why-mysql/white-papers/mysql_wp_cluster_manager.php

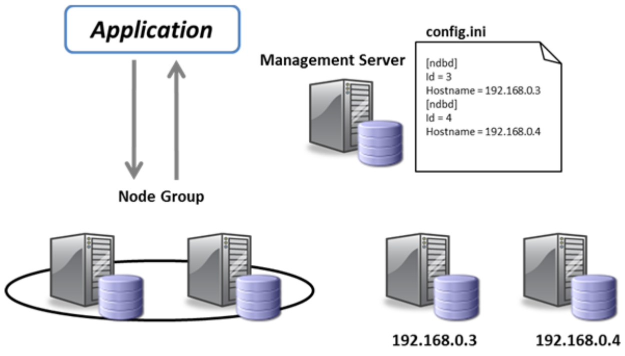
If not using MySQL Cluster Manager then the same results can be achieved but with a little more manual effort as described below.

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) will have to be recreated on the new server. The other servers continue to provide service while the new ones are added.

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 19 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.

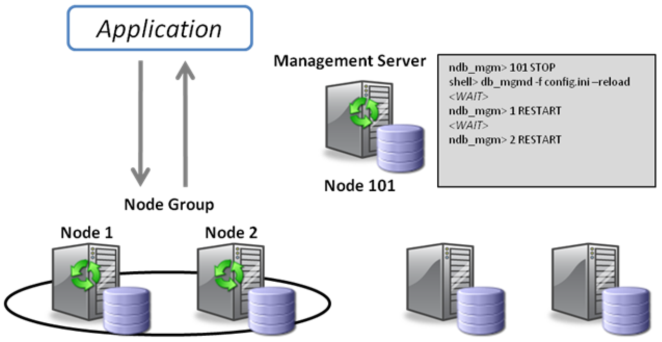
**Step 1**: Edit `config.ini` on all of the management servers as shown in Figure 20.



```
[ndbd]
Id = 3
Hostname = 192.168.0.3
[ndbd]
Id = 4
Hostname = 192.168.0.4
```

**Figure 20 Update config on all management servers**

**Step 2**: Restart the management server(s), existing data nodes and MySQL Servers, as illustrated in Figure 21. 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.

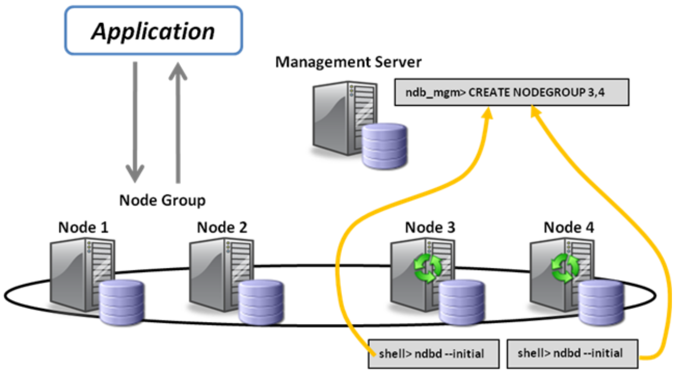


```
ndb_mgm> 101 STOP
shell> db_mgmd -f config.ini –reload
<WAIT>
ndb_mgm> 1 RESTART
<WAIT>
ndb_mgm> 2 RESTART
```

**Figure 21 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 22.

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.



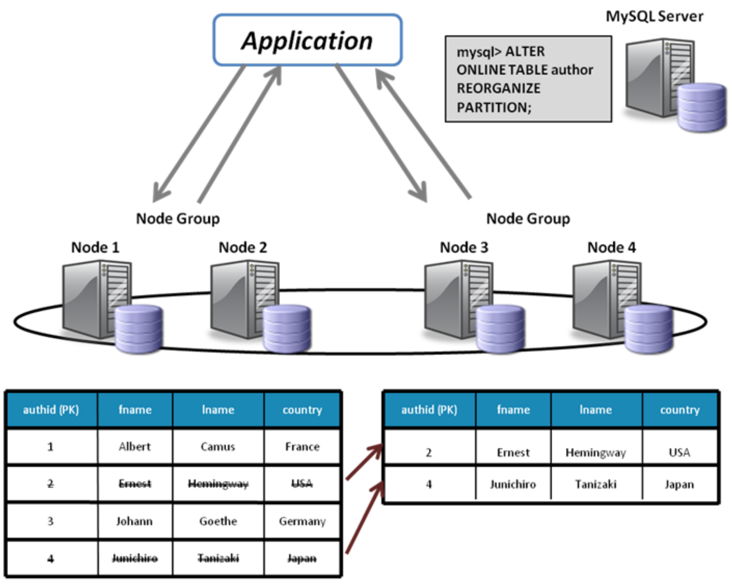
```
ndb_mgm> CREATE NODEGROUP 3,4
shell> ndbd --initial
shell> ndbd --initial
```

**Figure 22 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 23 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 TABLE` SQL command can be used to recover the unused spaces from the repartitioned tables.

Note that this is identical to Step 5 of the MySQL Cluster Manager instructions.



| authid (PK) | fname | lname | country |
|---|---|---|---|
| 1 | Albert | Camus | France |
| ~~2~~ | ~~Ernest~~ | ~~Hemingway~~ | ~~USA~~ |
| 3 | Johann | Goethe | Germany |
| ~~4~~ | ~~Junichiro~~ | ~~Tanizaki~~ | ~~Japan~~ |

| authid (PK) | fname | lname | country |
|---|---|---|---|
| 2 | Ernest | Hemingway | USA |
| 4 | Junichiro | Tanizaki | Japan |

**Figure 23 Data Repartitioned across node groups**

## 6. Further resources

The consulting 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 is a fast developing database and there are a number of resources 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/index-cluster.html.

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.

To get your first Cluster up and running in just a few minutes refer to the MySQL Cluster Quick Start guides from http://www.mysql.com/downloads/cluster/.

See how to use MySQL Cluster as part of different web-scale architectures in the "MySQL Reference Architectures for Massively Scalable Web Infrastructure" white paper (available from http://www.mysql.com/why-mysql/white-papers/mysql_wp_high-availability_webrefarchs.php).

You can learn more about all of the new features introduced as part of the latest MySQL Cluster release from this whitepaper:
http://mysql.com/why-mysql/white-papers/mysql_wp_cluster7_architecture.php

For more information about getting the best from MySQL Cluster (including tips on selecting hardware for improving performance) refer to the MySQL Cluster Evaluation guide: http://www.mysql.com/why-mysql/white-papers/mysql_cluster_eval_guide.php

View details of MySQL Cluster benchmark results – in particular how to acheive 1 Billion updates per minute: http://www.mysql.com/why-mysql/white-papers/mysql-cluster-benchmarks.php

## 7. 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, using Adaptive Query Localization, 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.