MYSQL NDB

Scale-up MySQL™ NDB Cluster 8.0.26 to +1.5M QPS with the AMD EPYC™ 7742

Introduction

MySQL™ celebrated the release of MySQL NDB Cluster[1] 1 8.0.26 on July 20th, 2021. MySQL NDB Cluster (NDB) is part of the MySQL family of open-source products that provides an in-memory, distributed, shared-nothing, high-availability storage engine for use in either a standalone configuration or with MySQL servers as front-ends. For the complete set of changes, please see Changes in MySQL NDB Cluster 8.0.26 (2021-07-20, General Availability).* You can download the Windows (x86, 64-bit) MSI installer of Zip archive from here.*

Choosing a database can seem like an overwhelming task because it requires the customer to consider factors such as performance (throughput and latency), high availability, data volume, scalability, ease of use, and operations. These considerations are affected by where the database runs, whether that is:

- In a cloud provider such as Oracle Cloud Infrastructure[1] that offers a broad range of infrastructure from small virtual machines[1] (VMs) to large bare metal[1] (BM) instances, and High-Performance Computing (HPC) servers.[1]
- One's own on-premise hardware.

Tuning a database for optimal performance can understanding and experimenting with hundreds of different parameters. Going one level deeper into the operating system and tuning kernel settings to best match the database requirements further complicates this task. Finally, database tuning occurs for a specific workload; applying the same tuning settings to a different workload might result in sub-optimal performance. This performance brief will help you strike the best balance when deploying NDB on AMD EPYC™ 7742 processor-based platforms.

AMD EPYC 7002 for HPC

2nd Gen AMD EPYC CPUs deliver high per-core performance to the industry by taking advantage of fast CPU frequencies, low latency memory, and a unified cache structure. AMD EPYC empowers the HPC community to propel innovations and insights with ground-breaking high-performance computing and advanced security features to deliver excellent results.

"Zen2" Core & Security

Support for up to:

- 64 physical cores, 128 threads
- 256MB of L3 cache per CPU
- 32MB of L3 cache per core
- 4TB of DDR4-3200 memory
- 128 PCIe® Gen 4 lanes
- Infinity Guard security[2]
- Secure Boot
- Encrypted memory with SME

Scale Out & Scale Up

Scaling is critical for HPC applications. AMD EPYC 7002 processors provide high bandwidth between nodes with support for PCIe® Gen 4 enabled network devices and accelerators. Each node can take advantage of up to 64 cores, support for 8 memory channels of DDR4-3200, and up to 256 MB of L3 cache per CPU.

MySQL™ NDB Cluster

MySQL NDB Cluster is a high-availability, high-redundancy version of MySQL adapted for the distributed computing environment. The most recent NDB Cluster release series uses version 8 of the NDB storage engine (also known as NDBCLUSTER) to enable running several computers with MySQL servers and other software in a cluster

Achieving High Performance and High Availability

The server configuration shown in Table 1 was used to set up a high-performance cluster with high availability protection in a single system using the recently-released DB Cluster 8.0.26. The Sysbench OLTP point select benchmark was then used to achieve a constant throughput of over 1.5M primary key lookups per second with a two-data-node cluster. Each data node is configured with 32-core CPUs using a total of 16 MySQL servers and 1024 clients (SysBench threads).

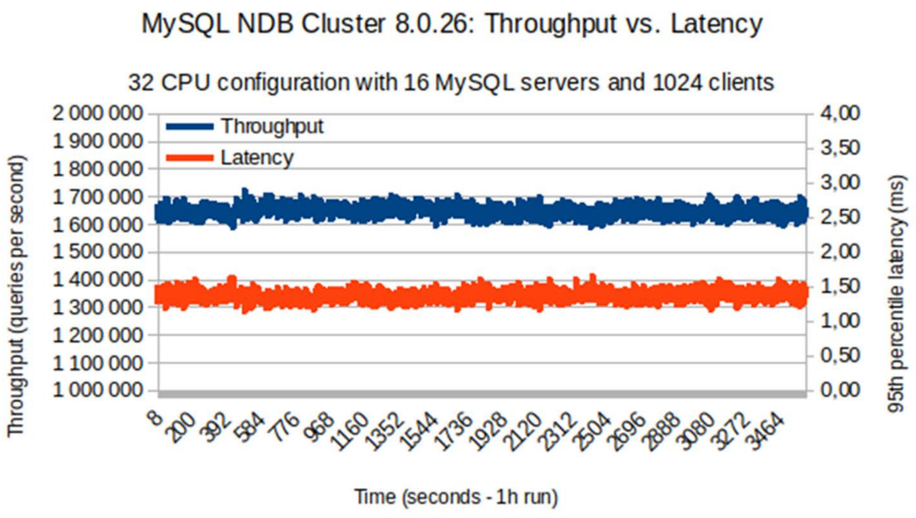


Figure 1: MySQL NDB Cluster 8.0.26: Throughput vs. Latency[4]

Figure 1 shows the results of a 1-hour throughput vs. latency test run by Oracle MySQL NDB engineers that achieved a constant **Throughput** (blue line) in the range of 1.6–1.7 million queries per second (primary-key lookups) with a maximum recorded throughput of 1,716,700 primary key lookups per second. The red line represents the 95th percentile **Latency**, which is in the range of 1.1–1.6 milliseconds with an average of 1.35 milliseconds.[4]

The following sections detail the hardware, NDB configuration, benchmark setup, and the analyses of intermediate results that lead to these performance numbers.

Benchmarking Setup

SysBench is a well-known, simple to use benchmark for evaluating database performance under different load scenarios. All benchmarks were run using SysBench 1.1.0, available from https://github.com/akopytov/sysbench*[1] with no changes made to the benchmark code to help ensure transparency and reproducibility.

This dataset uses 8 tables and 10M rows per table using around 60GB of memory. This configuration is a common starting point for many benchmarks performed by the Oracle MySQL team for both InnoDB and NDB Cluster storage engines. This dataset is large enough to barely exceed the size of the CPU cache but not so large as to cause too much I/O activity, such as long-duration node restarts or dataset initialization.

The test used the server configuration shown in Table 1 and an OLTP point-select workload consisting of primary-key lookup queries returning a constant string value. This workload tests the full database stack (MySQL servers and NDB storage engine) for overall code efficiency and the best possible query execution latency. Key generation is done using the default uniform distribution algorithm. SysBench is run in the same machine as the database and connects to MySQL servers via Unix sockets.

Hardware Setup

AMD System Configuration	
System	Dell EMC PowerEdge R7525 Server
CPU	2 x AMD EPYC 7742 | SP3 Package
Frequency: Base | Boost[3]	2.25GHz | 3.4GHz
Cores	64
L3 Cache	256MB
Memory	32 x 64 GB DDR4 DIMMs (SK Hynix), 3200 MT/s Speed, 2TB RAM Total
Storage: OS | Data	4 x 3.2TB Dell Express Flash PM1725b NVMe SSDs
BIOS and Settings	SMT=off, X2APIC=on, IOMMU=off, APBDIS=1, Fixed SOC P-state=0, Determinism=power, NPS=4, DF C-states=off, PIO, EPIO, TSME=off, PCIe 10-bit tag=on
OS Settings	Clear caches before every run, NUMA balancing 0, randomize_va_space 0, cc6 disabled, Governor=Performance

Table 1: AMD System Configuration

Note: Similar specification servers are available by choosing the BM.Standard.E3.128[1] shape currently offered in Oracle Cloud Infrastructure (OCI). You can find a list of available shapes and their specifications here.[1]

Software Setup

Software	
Software	MySQL MySQL NDB Cluster 8.0.26
OS	Oracle Linux® 8.3 with Unbreakable Enterprise Kernel (UEK) 5.4.17-5.4.17-2011.7.4

Table 2: Table 2: Software Configuration

MySQL NDB Cluster Setup

A minimal recommended high-availability scenario requires 4 hosts: 2 hosts running data nodes, and 2 hosts running management nodes and MySQL servers or applications (see the FAQ*). This scenario allows any of the hosts to be unavailable without impacting the service. Software-level redundancy can be supported by running two data nodes and multiple MySQL servers or applications using a single-box setup. In this scenario allows performing online operations such as online upgrades without service impact.

Equally splitting machine resources for each data node and set of MySQL servers takes advantage of the server configuration described in Table 1.

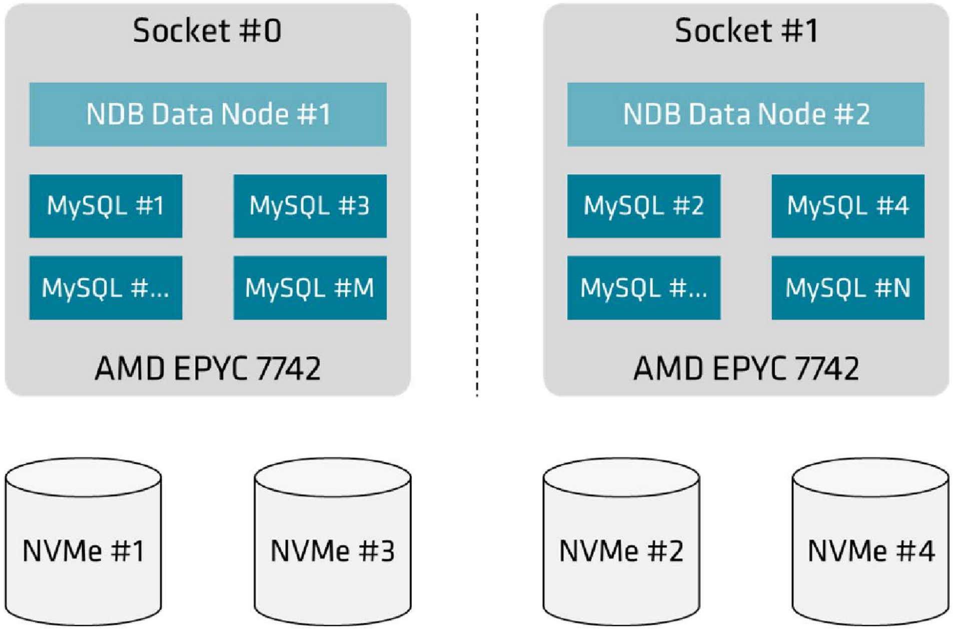


Figure 2: MySQL NDB Cluster Setup

MySQL NDB Cluster "Cluster-in-a-Box" Setup Using a Dual-Socket Server

This setup uses a single NUMA node per socket (physical CPU). The server supports configuring up to 4 NUMA nodes per socket for a total of 8 NUMA nodes , as described in the [AMD Tuning Guide](). Each NUMA node runs a single data node and a balanced number of MySQL servers accessing half of the available memory. MySQL NDB is an in-memory database; however, disk-checkpointing is both enabled by default and a recommended setting. In our setup, all NVMe disks are available from a single NUMA node only (ideally we would have half of the disks per NUMA node).

Having defined the cluster topology using two data nodes and several MySQL servers, the next step is to define how many CPU resources to allocate to NDB and MySQL server processes. A 25/75 CPU allocation provides a good starting point.

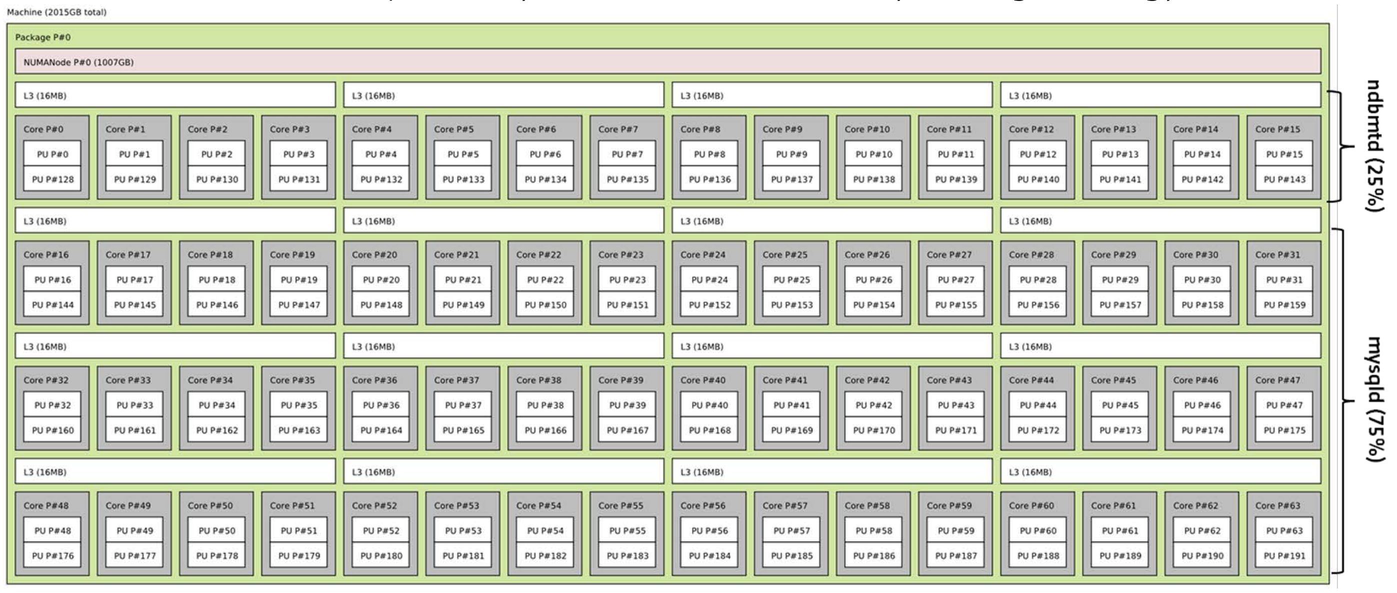


Figure 3: Recommended ndbmtd (25%) and mysqid (75%) CPU allocation

MySQL NDB Cluster is designed to be very efficient and requires fewer resources than MySQL server. The actual division of resources will depend on the workloads. Cases where queries can be pushed to the data nodes benefit from reserving more CPU for NDB. Cases where SQL-level aggregations or functions are performed benefit from reserving more CPU for MySQL server.

The above resource allocation reserves 16 cores (32 threads) for NDB data (`ndbmtd`) processes and 48 cores (96 threads) for MySQL server (`mysqld`) processes on each socket (physical CPU).

The main NDB Cluster configuration is:

```
[ndbd default]
NoOfReplicas = 2
DataMemory = 128G

# Auto configures NDB to use 16 cores/32 threads per data node
AutomaticThreadConfig = 1
NumCPUs = 32
NoOfFragmentLogParts = 8

# Prevents disk-swapping
LockPagesInMainMemory = 1

# Enables Shared-Memory Transporters (20% performance gain)
UseShm=1

# Allocates sufficient REDO log to cope with sysbench prepare step
RedoBuffer=256M
FragmentLogFileSize=1G
NoOfFragmentLogFiles=256
```

The key elements of this configuration are:

- **NoOfReplicas:** Defines the number of fragment replicas for each table stored in the cluster. With two data nodes, each node will contain all the data, thereby ensuring redundancy in case any of the data nodes goes down.
- **DataMemory:** The amount of memory used to store in-memory data. This was set it to 128G for this benchmark. Each data node has 1TB of available RAM, and this could be increased to 768G while still leaving a big margin for the operating system.
- **AutomaticThreadConfig:** When enabled, allows the data node to define which NDB-specific threads to run.
- **NumCPUs:** Restricts the number of logical CPUs to use. We set it to 32, which means that we are expecting NDB to take advantage of the 16 cores / 32 threads available.
- **NoOfFragmentLogParts:** Optional configuration; sets the number of parallel REDO logs per node. We set this to 8 because there will be 8 LDM threads when using NumCPUs=32. This enables each LDM thread to access REDO log fragments without using mutexes, thereby slightly improving performance.
- **LockPagesInMainMemory:** Prevent swapping to disk, ensuring best performance. We set this to 1, which locks the memory after allocating memory for the process.
- **UseShm:** Enables a shared memory connection between the data nodes and the MySQL servers. This is a must when co-locating MySQL servers with data nodes, as it provides a 20% performance improvement.

The other configuration options are required only to run the SysBench prepare command used to fill data in the database. They have no impact when running an OLTP point select workload but might have in other workloads.

The management-node and data-node specific options are:

```
[ndb_mgmd]
NodeId = 1
HostName = localhost
DataDir = /nvme/1/ndb_mgmd.1

[ndbd]
NodeId = 2
HostName = localhost
DataDir = /nvme/1/ndbd.1

[ndbd]
NodeId = 3
HostName = localhost
DataDir = /nvme/2/ndbd.2
```

These options define one management node and two data nodes. Each node has a unique identifier (`NodeId`), the hosts from where it will be running (`HostName`, set to `localhost`), and the path where to store required files (`DataDir`).

The final required configuration for NDB processes is adding API nodes to allow MySQL servers and NDB tools to connect to the cluster. Some of these configurations include:

```
[mysqld]
NodeId = 11
HostName = localhost

...

[api]
NodeId = 245

...
```

Click here* for a complete list of data node configuration parameters.

The MySQL server configuration is:

```
mysqld]
ndbcluster
ndb-connectstring=localhost
max_connections=8200

# Below three options are for testing purposes only
user=root
default_authentication_plugin=mysql_native_password
mysqlx=0

[mysqld.1]
ndb-nodeid=11
port=3306
socket=/tmp/mysql.1.sock
basedir=/nvme/3/mysqld.1
datadir=/nvme/3/mysqld.1/data

[mysqld.2]
ndb-nodeid=12
port=3307
socket=/tmp/mysql.2.sock
basedir=/nvme/4/mysqld.2
```

```
datadir=/nvme/4/mysqld.2/data

...
```

All MySQL servers or instances have three important settings specified under [`mysqld`]:

- **ndbcluster:** Enables the NDB Cluster storage engine.
- **ndb-connectstring:** Explicitly sets the address and port of all management nodes used to connect to NDB Cluster. This setting is optional if the management node is run locally;
- **max_connections:** Only required when running benchmarks with a large number of clients.

Each MySQL server requires individual configurations for at least the port, socket, basedir, and datadir. The complete configuration files and instructions are available from https://github.com/tiagomlalves/epyc7742-ndbcluster-setup*.

Running MySQL NDB Cluster, MySQL Server, and SysBench

Running MySQL NDB Cluster and MySQL Server requires installing all packages in the system and making them available in the path. This section assumes that SysBench has been compiled and installed in the system.

We use `numactl` to set process affinity to specific CPUs / NUMA nodes according to the setup described in <add cross-reference> that reserves 25% of CPU capacity for NDB data nodes and 75% CPU capacity for other processes.

To run the management node:

```
$ numactl -C 60-63,188-191,124-127,252-255 \
  ndb_mgmd \
    --ndb-nodeid=1 \
    --configdir="/nvme/1/ndb_mgmd.1" \
    -f mgmt_config.ini
```

Notice that the management node (`ndb_mgmd`) consumes very few resources and can be run from any logical CPU. The above `numactl` settings allow `ndb_mgmd` to run from any CPU in any NUMA node except for those reserved for data nodes.

To run data nodes:

- The first data node (`nodeid=2`) runs in the first 16 cores of the first NUMA node (NUMA #0), and affinity is thus set to CPUs 0–15 and 128–143.

```
$ numactl -C 0-15,128-143 \
  ndbmtd --ndb-nodeid=2
```

- The second data node (`nodeid=3`) runs in the first 16 cores of the second NUMA node (NUMA #1), and affinity is thus set to CPUs 64–79 and 192–207.

```
$ numactl -C 64-79,192-207 \
  ndbmtd --ndb-nodeid=3
```

This test ran multiple MySQL servers per NUMA node:

- All odd-numbered MySQL servers ran in NUMA #0 (CPUs 16–63 and 144–191

```
$ numactl -C 16-63,144-191 \
  mysqld \
    --defaults-file=my.cnf \
    --defaults-group-suffix=.1
```

- All even-numbered MySQL servers ran in NUMA #1 (CPUs 80–127 and 208–255).

```
$ numactl -C 80-127,208-255 \
  mysqld \
    --defaults-file=my.cnf \
    --defaults-group-suffix=.2
```

Running multiple MySQL servers in the same NUMA node shares all CPUs except those reserved for the data nodes. It is possible to have each MySQL server process running in a dedicated set of CPUs and thereby preventing shared CPU resources between processes. This approach requires careful validation, as discussed later.

To run SysBench:

```
$ THREADS=1024 ; \
  MYSQL_SOCKET=/tmp/mysql.1.sock,/tmp/mysql.2.sock,... ; \
  numactl -C 16-63,144-191,80-127,208-255 \
  sysbench \
    --db-driver=mysql \
    --mysql-storage-engine=ndbcluster \
    --mysql-socket="${MYSQL_SOCKET}" \
    --mysql-user=root \
    --tables=8 \
    --table-size=10000000 \
    --threads="${THREADS}" \
    --time=300 \
    --warmup-time=120 \
    --report-interval=1 \
    oltp_point_select run
```

This test runs SysBench in the same machine as MySQL NDB Cluster and MySQL servers using Unix sockets. Typical scenarios run applications and database from different servers and requiring using the TCP/IP network stack instead. These scenarios exhibit an inferior performance compared to that reported in this brief.

Each run has a duration of 300 seconds (5 minutes) with a warm-up period of 120 seconds (2 minutes). In practice, a 2-minute warm-up duration suffices to run the benchmark for 1–2 minutes. This benchmark was validated for periods longer than one hour with no significant variation in the mean throughput recorded.[4]

Scaling up a Single MySQL Server

The first step when benchmarking MySQL NDB Cluster was to run a single MySQL server to obtain a base understanding of the specific workload being tested and understand know how to allocate resources and fine tune further parameters.

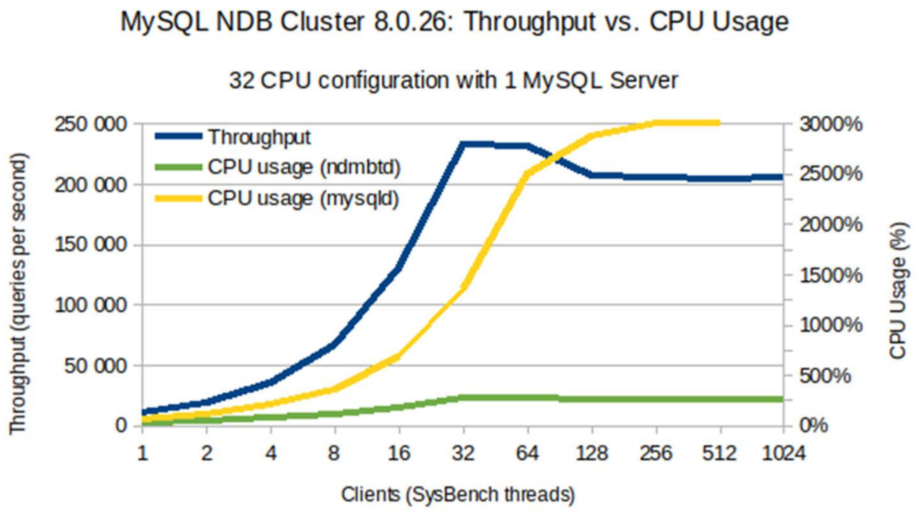


Figure 4: Scaling up a Single MySQL Server Results4

Figure 4 depicts:

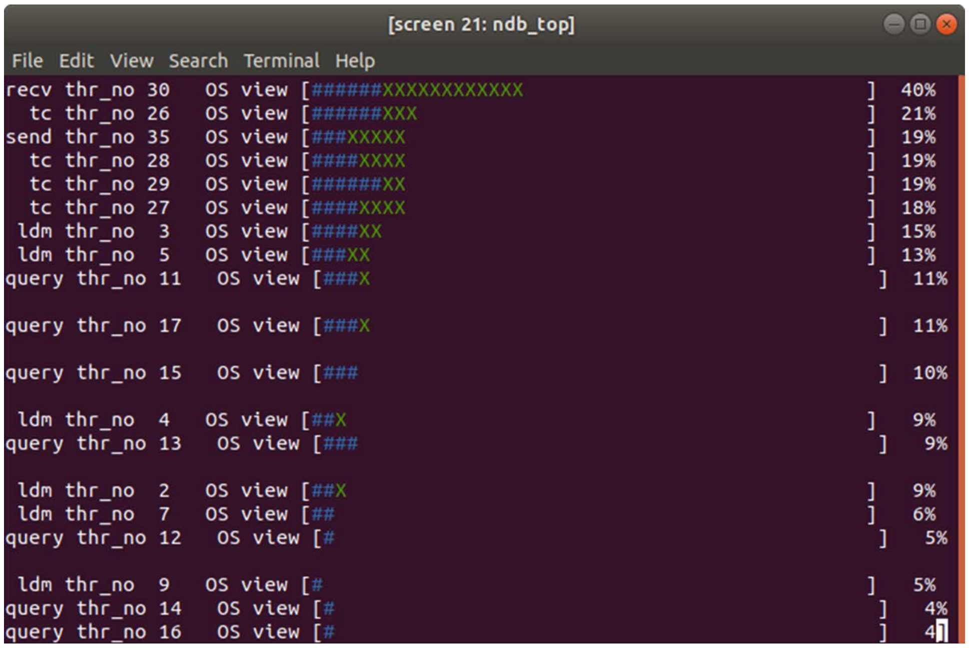
- The Throughput measured in queries per second, depicted in blue.
- NDB data node (`ndbmtd`) CPU utilization, depicted in green
- MySQL server (`mysqld`) CPU utilization, depicted in yellow.
- Throughput scale (left Y-axis)
- CPU utilization (right Y-axis)
- Number of clients (SysBench threads) used for each run (X-axis).

Figure 4 shows that:

- Increasing from 1 to 32 clients increases both throughput and `mysqld` CPU utilization accordingly, while the same range shows only a slight `ndbmtd` CPU utilization increase.[4]
- Maximum throughput of ~230K queries per second occurs at 32 clients. MySQL server (`mysqld`) CPU utilization is about ~1400%, meaning that a total of 14 logical CPUs are in use.
- `mysqld` CPU utilization almost doubles (2500% – 25 logical CPUs) causing a slight throughput degradation from 32-63 clients.
- Further throughput degradation and a flattened `mysqld` CPU utilization curve occurs from 64 to 128 clients, meaning that the MySQL server is saturated. At this stage, `mysqld` is using ~3000% of CPU (30 logical CPUs) out of the 96 logical CPUs available (48 cores / 96 threads).
- Increasing the client count beyond 128 clients onward yields no further increase in throughput or CPU utilization.

Optimal throughput conditions using a single MySQL server for OLTP point select workload happens at 32 clients, with maximum MySQL server capacity occurring at around 64 clients.

The MySQL server is the bottleneck because NDB CPU utilization is fairly low. You can see this using the ndb_top tool:



```
recv thr_no 30    OS view [######XXXXXXXXXXXX                          ]  40%
  tc thr_no 26    OS view [######XXX                                   ]  21%
send thr_no 35    OS view [###XXXXX                                    ]  19%
  tc thr_no 28    OS view [####XXXX                                    ]  19%
  tc thr_no 29    OS view [######XX                                    ]  19%
  tc thr_no 27    OS view [####XXXX                                    ]  18%
 ldm thr_no  3    OS view [####XX                                      ]  15%
 ldm thr_no  5    OS view [###XX                                       ]  13%
query thr_no 11    OS view [###X                                        ]  11%

query thr_no 17    OS view [###X                                        ]  11%

query thr_no 15    OS view [###                                         ]  10%

 ldm thr_no  4    OS view [##X                                         ]   9%
query thr_no 13    OS view [###                                         ]   9%

 ldm thr_no  2    OS view [##X                                         ]   9%
 ldm thr_no  7    OS view [##                                          ]   6%
query thr_no 12    OS view [#                                           ]   5%

 ldm thr_no  9    OS view [#                                           ]   5%
query thr_no 14    OS view [#                                           ]   4%
query thr_no 16    OS view [#                                           ]   4
```

Figure 5: ndb_top showing bottleneck in the MySQL server

Figure 5 shows the CPU utilization of the different NDB threads (`ldm`, `query`, `tc`, `send`, `recv`, and `main`). The test configuration includes 3 `recv` threads, of which one is only at 40% and the remainder are idle. Most other threads have a utilization below 80%. This confirms that the bottleneck is in the MySQL server side and not in NDB. Please click here* for information about the NDB internals.[1]

Addressing the MySQL bottleneck is as simple as scaling up the number of MySQL servers. Figure 4 shows that optimal throughput conditions happen when using 32 clients with 14 logical CPUs being used by MySQL server. Per Oracle: "Considering that we have 48 cores / 98 threads per socket, we can have ~98/14 = 7 MySQL servers per socket. Rounding up this gives us around 8 MySQL servers per socket."[4]

Scaling up Multiple MySQL Servers

More MySQL servers can be added when a single MySQL server becomes saturated. We previously estimated that we could use up to 8 MySQL servers per socket, or 16 MySQL servers in total.

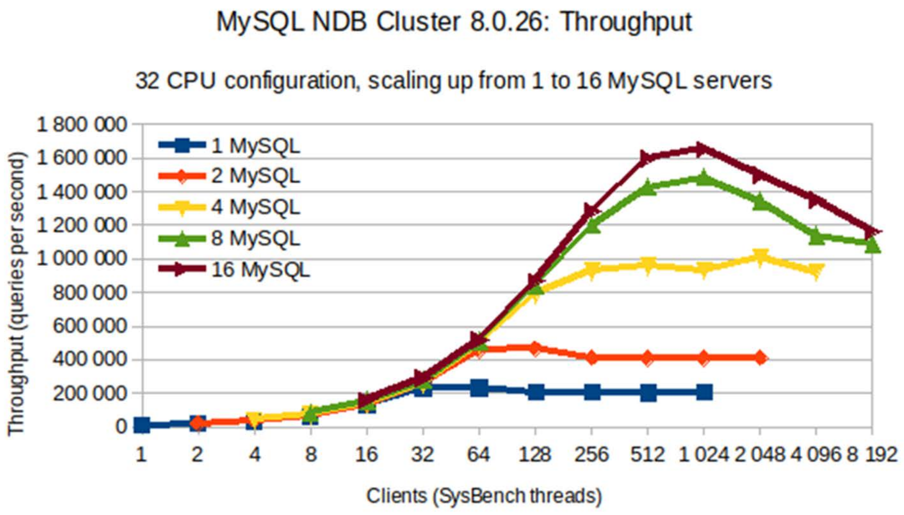


Figure 6: Scaling up Multiple SQL Servers4

Figure 6 shows a series of tests done with an increasing number of clients (SysBench threads) for 1 to 16 MySQL servers. This chart records the average queries per second over a 300-second period. As expected, the number of MySQL servers must double after 32 clients to sustain an increasing throughput with more clients (SysBench threads).[4] However, scaling from 8 to 16 MySQL servers does not double the throughput. Maximum throughput is reached with 16 MySQL servers using 1024 clients (SysBench threads). Adding extra clients degrades throughput as the system becomes saturated.

Figure 7 looks at latency when running an increasing number of clients for a different number of MySQL servers.

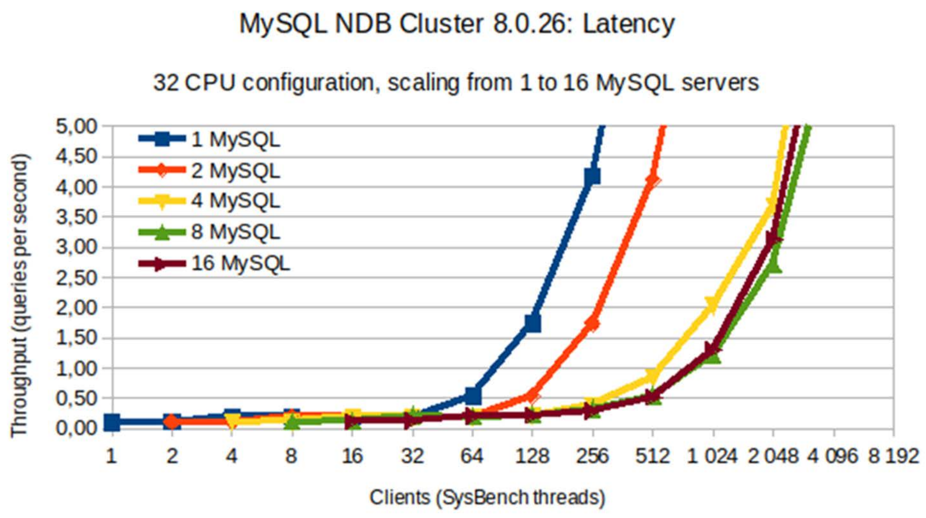


Figure 7: MySQL NDB Cluster 8.0.26 Latency4

When using a single MySQL server, the 95th percentile latency is below 0.5ms up to 32 clients (SysBench threads) and then grows exponentially when more clients are used. Doubling the number of MySQL servers allows doubling the number of clients (SysBench threads), which keeps the same low latency below 0.5ms. However, using 8 or more MySQL servers can no longer keep latencies below 0.5ms when using 512 clients or more because of system saturation. There is also no significant latency difference between using a total of 8 or 16 MySQL servers, although latency does worsen at 2048 clients and above.

Figures 6 and 7 show the average throughput and 95th percentile latency for a full run.[4] Figure 8 shows throughput and latency stability during the run:

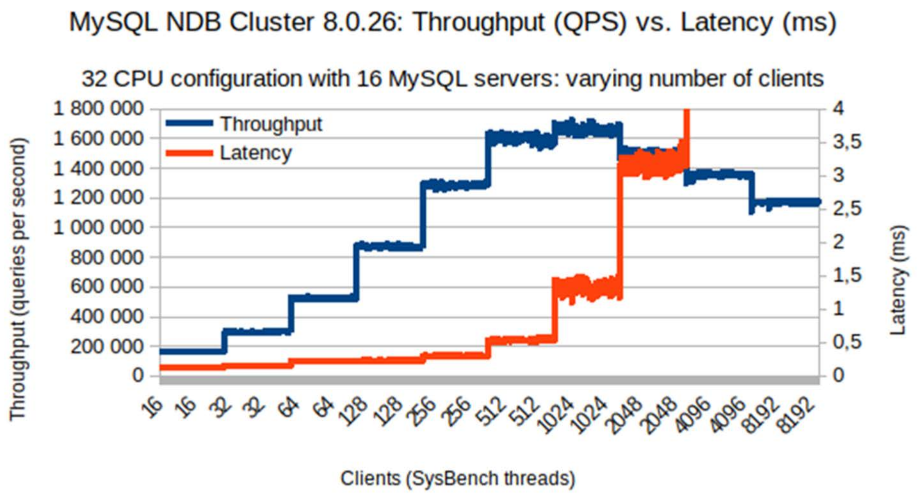
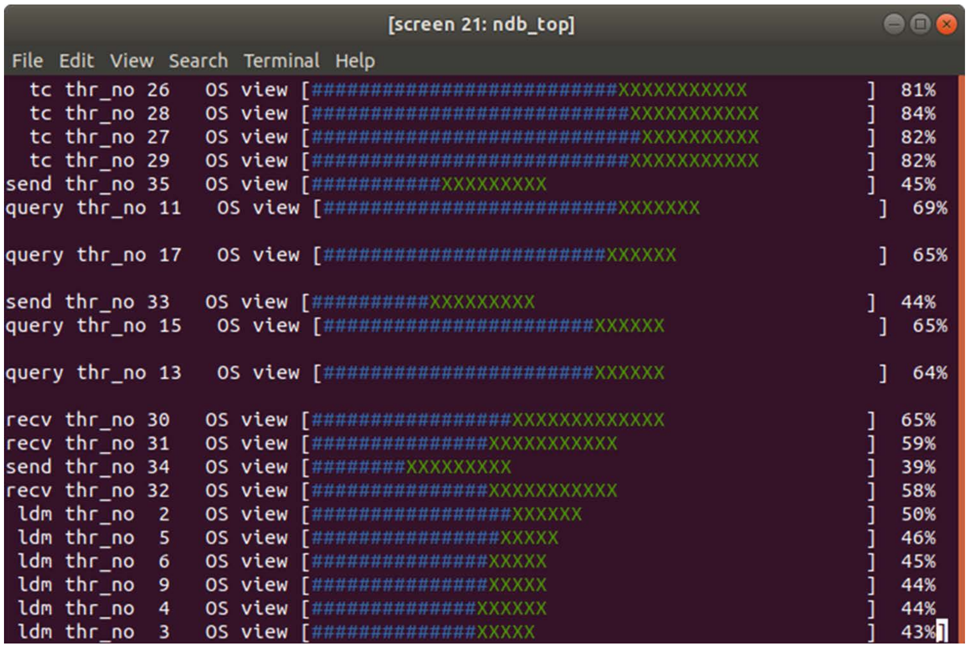


Figure 8: Throughput (QPS) vs Latency (ms)4

Figure 8 shows the average throughput (in blue) and 95th percentile latency (in red) sampled every second using 16 MySQL servers for an increasing number of clients (SysBench threads).[4] Both throughput and latency are fairly stable, except when nearing system saturation. Throughput holds steady even when latency exceeds 0.5ms, which is normal for an in-memory database.

Going Above 1.7M QPS

This cluster configuration and workload can exceed above 1.7M queries per second. It is also possible to further reduce the variation in measurements by fine-tuning operating system settings. However, this is no longer an easy task, because it requires experimenting with other configuration parameters and is thus outside the scope of this performance brief. Even so, you can see a hint about the next possible steps, starting by looking at the output of `ndb_top` shown in Figure 9:



```
  tc thr_no 26   OS view [##########################XXXXXXXXXXX          ]  81%
  tc thr_no 28   OS view [###########################XXXXXXXXXXX         ]  84%
  tc thr_no 27   OS view [############################XXXXXXXXXX         ]  82%
  tc thr_no 29   OS view [###########################XXXXXXXXXXX         ]  82%
send thr_no 35   OS view [###########XXXXXXXXX                           ]  45%
query thr_no 11   OS view [#########################XXXXXXX               ]  69%

query thr_no 17   OS view [########################XXXXXX                 ]  65%

send thr_no 33   OS view [##########XXXXXXXXX                            ]  44%
query thr_no 15   OS view [#######################XXXXXX                  ]  65%

query thr_no 13   OS view [#######################XXXXXX                  ]  64%

recv thr_no 30   OS view [#################XXXXXXXXXXXXX                 ]  65%
recv thr_no 31   OS view [###############XXXXXXXXXXX                     ]  59%
send thr_no 34   OS view [########XXXXXXXXX                              ]  39%
recv thr_no 32   OS view [###############XXXXXXXXXXX                     ]  58%
 ldm thr_no  2   OS view [#################XXXXXX                        ]  50%
 ldm thr_no  5   OS view [################XXXXX                          ]  46%
 ldm thr_no  6   OS view [###############XXXXX                           ]  45%
 ldm thr_no  9   OS view [###############XXXXX                           ]  44%
 ldm thr_no  4   OS view [##############XXXXXX                           ]  44%
 ldm thr_no  3   OS view [##############XXXXX                            ]  43%
```

Figure 9: NDB Cluster Bottleneck

This confirms that the NDB cluster is now the bottleneck of the overall system because the `tc` threads are at 80% CPU utilization and have reached saturation. Meanwhile, the other threads are far from saturation, which leaves space for further optimization.

Enabling AutomaticThreadConfig and configuring NumCPUs=32 means that NDB will use the following threads:

- 8 `ldm`
- 8 `query`
- 4 `tc`
- 3 `send`
- 3 `recv`
- 1 `main`

Given that the `tc` threads are saturated but the `ldm+query` threads are still not being fully utilized, you could try improving query execution, by manually reducing the number of `ldm+query` threads and adding a few more `tc` threads.

Conclusion

MySQL NDB Cluster was developed to enable horizontal scaling; however, continuous high-end hardware improvement makes it important to have a simple way to scale up a database. MySQL NDB Cluster is an open-source distributed in-memory database that combines predictable scalability with high availability and provides real-time in-memory access with transactional consistency across partitioned and distributed datasets. It was developed to support scenarios requiring high-availability (99.999% or more) and predictable query times. Customers who follow this introductory performance brief can easily scale up MySQL NDB Cluster 8.0.26 to report over 1.7M primary key lookups per second on AMD EPYC 7742 processors.

References

1. For a complete list of world records see https://amd.com/worldrecords. EPYC-22
2. AMD Infinity Guard features vary by EPYC™ Processor generations. Infinity Guard security features must be enabled by server OEMs and/or Cloud Service Providers to operate. Check with your OEM or provider to confirm support of these features. Learn more about Infinity Guard at https://www.amd.com/en/technologies/infinity-guard.
3. For AMD EPYC processors this is the maximum frequency achievable by any single core on the processor under normal operating conditions for server systems.
4. https://blogs.oracle.com/mysql/scale-up-mysql-ndb-cluster-8026-to-%2b15m-qps-the-easy-way-with-amd-epyc-7742

RELATED LINKS

- For more information about AMD's EPYC line of processors visit: https://www.amd.com/epyc.
- For more information about AMD EPYC™ 7742 CPUs visit: https://www.amd.com/en/products/cpu/amd-epyc-7742.
- Source code and binaries for MySQL NDB Cluster 8.0.26: https://www.mysql.com/products/cluster.*