Performance Evaluation of NoSQL Databases:

Couchbase Server, MongoDB, and DataStax Enterprise (Cassandra)

This report compares the throughput and latency of three NoSQL databases across four workloads and three cluster configurations.

By **Artsiom Yudovin, Lead** Data Engineer
Uladzislau Kaminski, Senior Software Engineer
Ivan Shryma, Data Engineer
Sergey Bushik, Lead Software Engineer

Q1 2021

Table of Contents

1. Introduction

NoSQL encompasses a wide variety of database technologies that were developed in response to a rise in global volumes of data and the frequency with which this data is accessed. In contrast, relational databases were not designed to cope with the scalability and agility challenges that modern applications face, nor were they built to take advantage of the inexpensive storage and processing power available today. New-generation NoSQL systems help to achieve the highest levels of performance and uptime for workloads.

This report compares the performance results of three NoSQL databases: Couchbase Server v6.6.0, MongoDB v4.2.11 and DataStax Enterprise v6.8.3 (Cassandra). The goal of this report is to measure the relative performance in terms of latency and throughput each database can achieve. The evaluation was conducted on different cluster configurations—4, 10, and 20 nodes—as well as under four different workloads.

The first workload performs under an update-heavy mode—similar to a stock trading application—invoking 50% of reads and 50% of updates. The second workload performs a short-range scan that invokes 95% of scan and 5% of updates, where short ranges of records are queried instead of the individual ones. This way, the second workload simulates activities typical for an e-commerce application. The third workload represents a query with a single filtering option to which an offset and a limit are applied. Finally, the fourth workload is a `JOIN` query with grouping and ordering applied.

The Yahoo! Cloud Serving Benchmark (YCSB), an open-source specification and program suite for evaluating retrieval and maintenance capabilities of computer programs, was used as a default tool for evaluation consistency.

2. Key findings

2.1 Hardware configuration

Each of the NoSQL databases was deployed on 4-, 10-, and 20-node clusters in the same geographical region. The clusters were deployed on Amazon storage-optimized extra large instances.

Table 2.1 A detailed description of the Amazon EC2 instance the clusters were deployed to

Family	Storage-optimized
Type	i3.2xlarge
vCPUs	8
Memory (GiB)	61
Instance storage (GB)	1 × 1,900 (SSD)
EBS-optimized available	Yes
Network performance	Up to 10 GB
Platform	64-bit
Operational System	Ubuntu 18.04 LTS

To provide verifiable results, benchmarking was performed on Amazon Elastic Compute Cloud instances. The YCSB client was deployed to five Amazon compute-optimized large instances.

Table 2.2 A detailed description of the Amazon EC2 instance the YCSB client was deployed to

Family	Compute-optimized
Type	c4.2xlarge
vCPUs	8
Memory (GiB)	15
EBS-optimized available	Yes
Network performance	High
Platform	64-bit
Operational system	Ubuntu 18.04 LTS

2.2 Couchbase Server cluster configuration

Couchbase Server is both a JSON document and a key-value distributed NoSQL database. It guarantees high performance with a built-in object-level cache, a SQL-like query language, asynchronous replication, ACID transactions (as needed) and data persistence. The database is designed to automatically scale resources, such as CPU and RAM, depending on the workload.

For the Couchbase Server Enterprise Edition evaluation, a symmetric scale-out strategy was used giving each node equal share of work. Regardless of cluster size (4, 10, or 20 nodes), each node consists of Data, Index, and Query Services. Search, Analytics, and Eventing Services were disabled, and no resources were allocated for them as the corresponding features were not the point of interest of this benchmark. Each Data Service was allocated 60% of available RAM (36,178 MB) within its Couchbase “Bucket” (database container). Each bucket had a single replica configured. The Index Service was allocated approximately 40% of available RAM (about 24 GB) with memory-optimized indexes in use. Each index created was replicated to all Index Services.

2.3 MongoDB cluster configuration

MongoDB is a document-oriented NoSQL database. It has extensive support for a variety of secondary indexes and API-based ad-hoc queries, as well as strong features for manipulating JSON documents. The database uses a separate and incremental approach to data replication and partitioning that occur as completely independent processes.

MongoDB employs a hierarchical cluster topology that combines router processes, configuration servers, and data shards. For each cluster size (4, 10, and 20 nodes), production configuration has been used for deployment:

- A config server was deployed as a three-member replica set (a separate machine, not counted in a cluster).
- Each shard was deployed as a three-member replica set (one primary, one secondary, and one arbiter).
- Three mongos routers were deployed on each client.

Manual definition, installation, and configuration for a MongoDB sharded cluster is a fairly complicated procedure. In short, you need to satisfy installation prerequisites, then separately configure all the data shards, configuration servers, and sharding routers to finally combine those components into a cluster.

MongoDB distributes data, or shards, at the collection level, sharding partitions using the collection's data, which is defined by a shard key. Hash-based partitioning was used for all the models. To support hash-based sharding, MongoDB provides a hashed index type, which indexes the hash of a field value. With hash-based partitioning, two documents with "close" shard key values are unlikely to be part of the same chunk. This ensures a more random distribution of a collection in the cluster.

2.4 DataStax Enterprise(Cassandra) cluster configuration

DataStax Enterprise (Cassandra) is a wide column store NoSQL database management system designed to handle large amounts of data across many commodity servers, providing high availability with no single point of failure.

In the table below, the changes applied to each node on 4-, 10-, 20-node clusters are detailed.

Table 2.4 The changes applied to each node on each cluster

cassandra.yaml	
memtable_space_in_mb	16384
memtable_cleanup_threshold	0.11
memtable_flush_writers	40
row_cache_size_in_mb	20280
commitlog_total_space_in_mb	1969
cdc_total_space_in_mb	984
num_token	256
endpoint_snitch	Ec2Snitch
cassandra-env.sh	
MAX_HEAP_SIZE	20 GB
HEAP_NEWSIZE	1,800 MB
keyspace configuration	
replication _factor	2
class	SimpleStrategy
DURABILITY_WRITE	false

Row cache was also enabled for each cluster.

3. Workloads and Tools

Database performance was defined by the speed at which the database processed basic operations. A basic operation is an action performed by a workload executor, which drives multiple client threads. Each thread executes a sequential series of operations by making calls to a database interface layer both to load a database (the load phase) and to execute a workload (the transaction phase).

The threads throttle the rate at which they generate requests, so that we may directly control the offered load against the database. In addition, the threads measure latency and the achieved throughput of their operations and report these measurements to the statistics collection module.

3.1 Workloads

The performance of each database was evaluated under the following workloads:

- **Workload A.** Update heavily: 50% read and 50% update, request distribution is Zipfian.

- **Workload E.** Scan short ranges: 95% scan and 5% update, request distribution is Uniform.
- **Pagination Workload.** Filter with offset and limit.
- **JOIN Workload.** `JOIN` operations with grouping and aggregation (in the case of Couchbase, `ANSI JOIN` was evaluated, as well).

3.2 Tools

We used the YCSB client as a worker, which consists of the following components:

- workload executor
- the YCSB client threads
- extensions
- statistics module
- database connectors

The workloads were tested under the following conditions:

- Data fits the memory.
- Durability is false.
- Replication is set to "1" signifying that just a single replica is available for each data set.

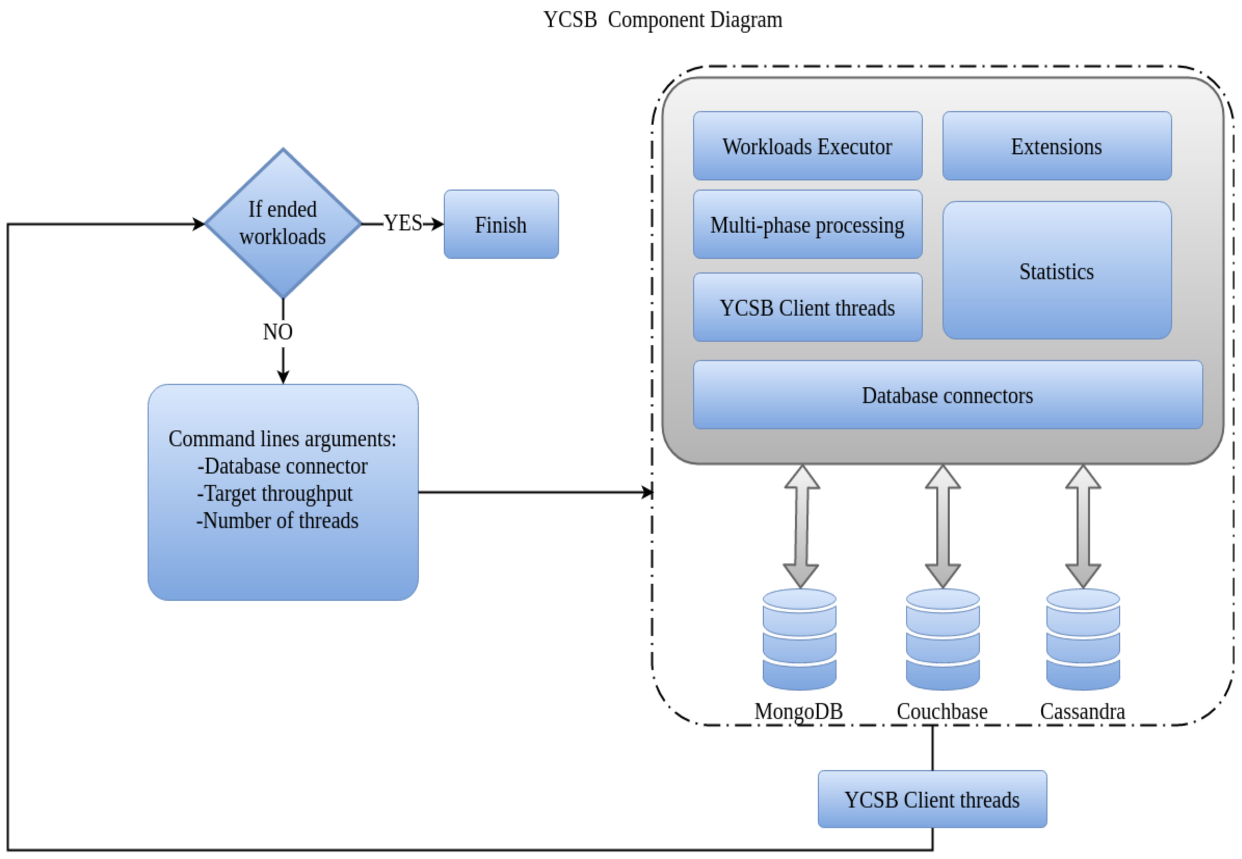

Figure 3.1 The components of the YCSB client

Workloads A and E are standard workloads provided by YCSB. Default data models were used for these workloads. Pagination Workload and JOIN Workload represent scenarios from real-life domains: finance (server-side pagination for listing filtered transactions) and e-commerce (series of reports on various products and services utilized by customers). To emulate these scenarios on a domain level, a customer–order model was introduced for these workloads.

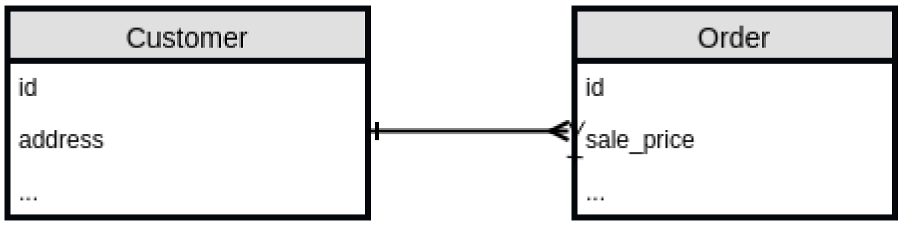

Figure 3.2 A graphic representation of the customer–order model

4. YCSB Benchmark Results

4.1 Workload A: The update-heavy mode

4.1.1 Workload definition and model details

Workload A is an update-heavy workload, which simulates typical actions of an e-commerce solution user—50% of read operations and 50% of updates. This is a basic key-value workload.

The scenario was executed with the following settings:

- The read/update ratio was 50%–50%.
- The Zipfian request distribution was used.
- The size of a data set was scaled in accordance with the cluster size: 50 million records (each 1 KB in size, consisting of 10 fields and a key) on a 4-node cluster, 125 million records on a 10-node cluster, and 200 million records on a 20-node cluster.

Couchbase Server stores data in buckets, which are the logical groups of items—key-value pairs. vBuckets are physical partitions of the bucket data. By default, Couchbase Server creates a number of master vBuckets per bucket (typically 1,024) to store bucket data and evenly distribute vBuckets across all cluster nodes. Querying with document keys is the most efficient because a query request is sent directly to a proper vBucket holding target documents. This approach does not require any index creation and is the fastest way to retrieve a document due to the key-value storage nature. The workload was executed without any index creation.

DataStax Enterprise (Cassandra) cluster has been preliminary warmed up to cache the results in memory (20 GB of RAM has been allocated for cache), which resulted in a hit rate up to 60%.

4.1.2 Query

The following queries were used to perform Workload A.

Table 4.1.1 Evaluated queries

Couchbase N1QL	MongoDB Query	Cassandra CQL
`bucket.get(docId, RawJsonDocument.class)`	`db.ycsb.find({_id: $1})`	`SELECT * FROM table WHERE id = $1 LIMIT 1`

4.1.3 Evaluation results

Under an in-memory data set with no disk hits, Couchbase significantly outperformed both MongoDB and Cassandra across all cluster topologies. Couchbase processed up to 105,900 ops/sec on a 4-node cluster, while Cassandra handled 97,500 ops/sec, and MongoDB only 23,700 ops/sec. On a 10-node cluster, Couchbase achieved 187,000 ops/sec, MongoDB 43,300 ops/sec, and Cassandra 142,200 ops/sec. On a 20-node cluster, it was observed that five workload clients (with 700 threads) were not enough to saturate the Couchbase cluster any further, therefore performance significantly improved to 330,000 ops/sec, whereas MongoDB performance grew to only 37,300 ops/sec, and Cassandra increased to 191,300 ops/sec.

Couchbase exhibited latency consistency with 3.4 ms on a 4-node cluster with 700 calling threads and 1.4 ms on a 20-node cluster with 3,500 threads. MongoDB scaled well with a request processing time from 18 ms on a 4-node cluster to 14 ms on a 10-node cluster with the same amount of calling threads. On a 20-node cluster, MongoDB latency increased to 19 ms.

The request latency spike on a 4-node cluster for MongoDB was caused by the Sharded Cluster Balancer. The balancer is a background process that monitors the number of chunks on each shard. When the number of chunks on a given shard reaches specific migration thresholds, the balancer attempts to automatically migrate chunks between shards and reach an equal number of chunks per shard. This can impact performance while the procedure takes place. On a bigger cluster, the balancer has less impact on performance, because the data chunks are distributed across more nodes, therefore the migration thresholds are infrequently reached.

DataStax Enterprise (Cassandra) appeared to be scaling well with the constantly decreasing request latency from 6.8 ms on a 4-node cluster to 4.8 ms on a 10-node cluster, and increased to 13.3 ms on a 20-node cluster. Cassandra got a few failed operations on a 20-node cluster, due to connection issues. It still underperformed compared to Couchbase, which exhibited 50% better throughput and lower latency on 4- and 10-node clusters.

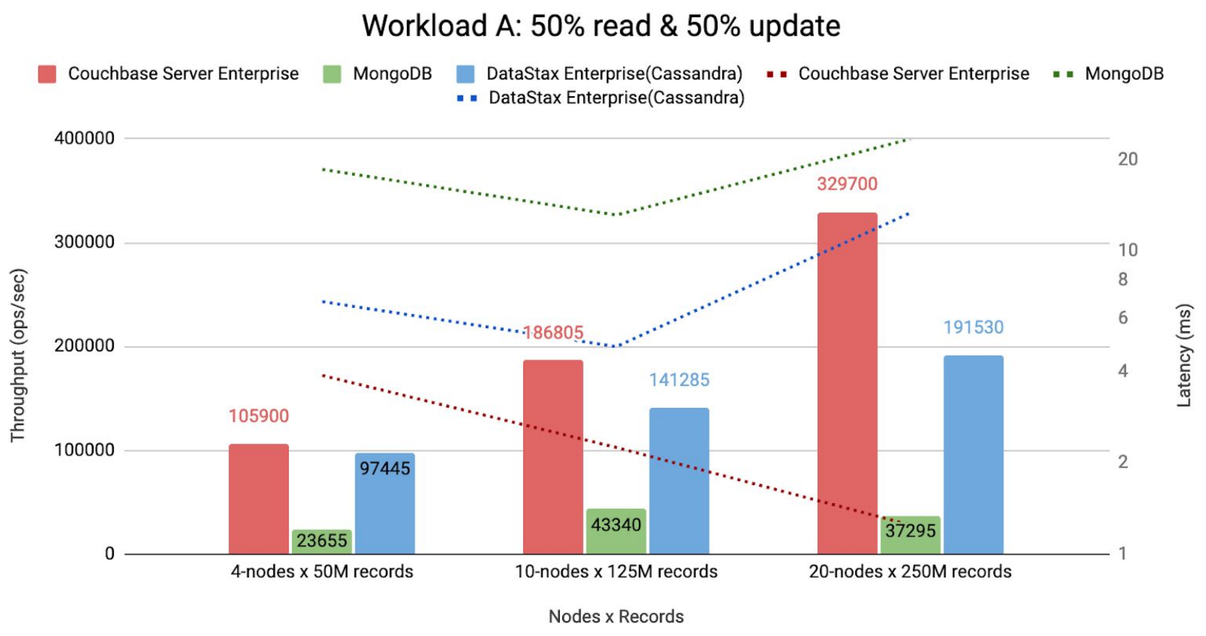

Figure 4.1.3 Performance results under Workload A on 4-, 10-, and 20-node clusters

4.1.4 Summary

Couchbase exhibited much better performance at scale (up to 4x latency and 3x throughput) than MongoDB and DataStax Enterprise (Cassandra). MongoDB reached its limit at about 500–700 threads and did not scale further. Both MongoDB and DataStax Enterprise (Cassandra) showed consistent improvement in the overall throughput proportionally to the cluster size growth. For larger cluster sizes, we observed that five client nodes, which we kept consistent throughout the tests, were not enough to fully saturate 20-node clusters of Couchbase, MongoDB, and Cassandra. Therefore, we got a marginal performance improvement in comparison to what a cluster is typically capable of delivering.

4.2 Workload E: Scanning short ranges

4.2.1 Workload definition and model details

Workload E is a short-range scan workload in which short ranges of records are queried, instead of individual ones. This workload simulates threaded conversations, where each scan goes through the posts in a given thread (assuming the entries to be clustered by ID). The scenario has been executed under the following settings.

- The read/update ratio was 95%–5%.
- The Zipfian request distribution was used.
- The size of a data set was scaled in accordance with the cluster size: 50 million records (each 1 KB in size, consisting of 10 fields and a key) on a 4-node cluster, 100 million records on a 10-node cluster, and 250 million records on a 20-node cluster.
- The maximum scan length reached 100 records.
- Uniform was used as a scan length distribution.

Given the fact that the scan operation is performed over the primary key in Couchbase, the following primary index has been created:

```
   CREATE PRIMARY INDEX `ycsb_primary` ON `ycsb`
   USING GSI WITH {"nodes": [...]}
```

The primary index is simply an index of the document key on the entire bucket. The primary index contains a full set of keys in a given keyspace. It is widely used for full bucket scans (primary scans), when the query does not have any filters (predicates) or when no other index or access path can be used. From the data structure point of view, the primary index is a skip list, containing the document IDs with binary search complexity.

Due to the cluster topology where each cluster node comprises Data and Query Services, primary indexes are scaled in accordance with cluster size and provide linear growth of throughput proportionally to the number of nodes. If we take in mind the complexity of a binary search by an index, when a data set grows from 50 million to 125 million records, the search time increases by 5%. This issue is mitigated by increasing a cluster size by two times. After a cluster doubles in size, about 90% of throughput growth is expected. This is explained by a double growth of Query Services divided by the expected 5% slowdown of scan operation per node.

MongoDB distributes data using a shard key. There are two types of shard keys supported by the system: range-based and hash-based. The range-based partitioning supports more efficient range queries. Given a range query on a shard key, a query router can easily determine which chunks overlap this range and route the query to only those shards that contain these chunks. However, the range-based partitioning can result in an uneven data distribution, which may negate some of the benefits of sharding. The hash-based partitioning ensures an even distribution of data at the expense of efficient range queries. Hashed key-value results in random distribution of data across chunks and, therefore, shards. However, random distribution makes it more likely that a range query on a shard key will not be able to target a few shards, but would more likely query every shard in order to return a result. The hash-based partitioning was used for all partitioning, so some performance degradation is expected here.

The scan operation implemented in YCSB for Cassandra is based on a *token* function. The return result of a scan operation depends on the selected partition. For this benchmark, the default Murmur3Partitioner was used. However, Murmur3Partitioner simply calculates a key hash, but does not preserve ordering—which may result in the unexpected return of a scan operation. Additionally, Cassandra does not support any ordering by partitionary key.

4.2.2 Query

The following queries were used to perform Workload E.

Table 4.2.1 Evaluated queries

Couchbase N1QL	MongoDB Query	Cassandra CQL

`SELECT RAW meta().id`
`FROM `ycsb``
`WHERE meta().id >= $1`
`ORDER BY meta().id`
`LIMIT $2`
	`db.ycsb.find({`
	`id: {`
	`$gte: $1`
	`}, {`
	`id: 1`
	`}).sort({`
	`id: 1`
	`}).limit($2)`
		`SELECT id`
		`FROM table`
		`WHERE token(id) >=`
		`token($1)`
		`LIMIT $2`

4.2.3 Evaluation results

Couchbase demonstrated great scalability with the linear growth of throughput that was proportional to the number of cluster nodes: from 9,625 ops/sec on a 4-node cluster to 22,580 ops/sec on a 10-node cluster. On a 20-node cluster, the throughput reached 33,095 ops/sec, which is about 46% more than on a 10-node cluster, with the request latency decreasing from 34 ms to about 13 ms due to usage of the primary index and the replication of the Index Service.

MongoDB had similar results from 18,255 ops/sec to 21,440 ops/sec. The results were comparatively the same regardless of cluster and data set sizes. MongoDB performed better than Couchbase on a 4-node cluster, but lower on 10- and 20-node clusters.

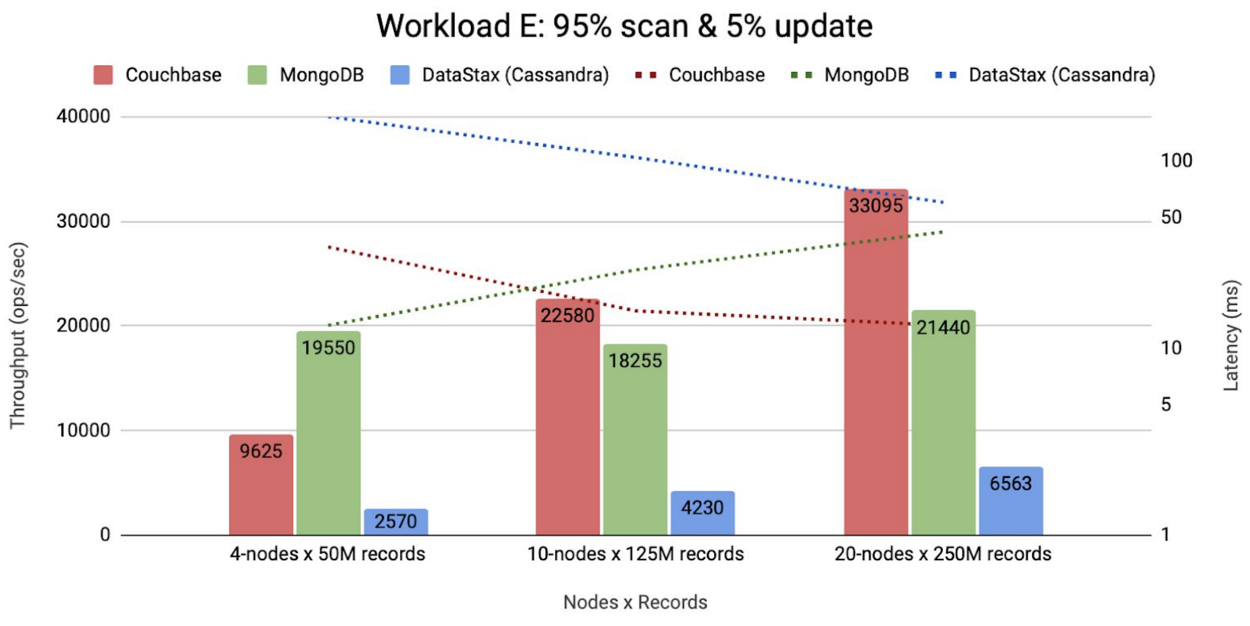

Figure 4.2.3 Performance results under Workload E on 4-, 10-, and 20-node clusters

Cassandra showed rather low performance on a scan operation: around 2,570 ops/sec on a 4-node cluster, 4,230 ops/sec on a 10-node cluster, and around 6,563 ops/sec on a 20-node cluster. However, Cassandra was able to achieve a linear performance increase performance across all clusters and data sets. This can be explained by the fact that coordinator nodes send scan requests to other nodes in the cluster responsible for specific token ranges. The more nodes a cluster has, the less data falls in the target range on each node, thus the less data each node has to return. This resulted in reduced per-node request processing time. As the coordinator sends the requests in parallel, the overall request processing time depends on each cluster node request latency which decreases with cluster growth. This is proven by the gradual decrease of request latencies from 173 ms on a 4-node cluster to 104 ms on a 10-node cluster and 63 ms on a 20-node cluster.

4.2.4 Summary

MongoDB performed better than Couchbase on relatively small-sized clusters and data sets (4 nodes and 50 million records, each 1 KB in size), but remained flat irrespective of the cluster size. On the other hand, Couchbase outscaled and outperformed MongoDB on bigger clusters showing linear throughput growth on 10- and 20-node clusters with data sets of 125 and 250 million records correspondingly. MongoDB showed the ability to handle the increasing amount of data with the throughput remaining the same. Cassandra displayed greater scalability in comparison to MongoDB and Couchbase, preserving the linear performance growth, but still lagging behind Couchbase and MongoDB in terms of overall operation performance.

4.3 Pagination Workload: Filter with OFFSET and LIMIT

4.3.1 Workload definition and model details

Pagination Workload is a query with a single filtering option, an offset, and a limit. The workload simulates a selection by field with pagination. The scenario was executed under the following settings.

- The read ratio is 100%.
- The size of a data set was scaled in accordance with the cluster size: 5 million *customers* (each 4 KB in size) on a 4-node cluster, 25 million *customers* on a 10-node cluster, and 50 million *customers* on a 20-node cluster.
- The maximum of a query length reached 100 records.
- Uniform was used as a query length distribution.
- The maximum query offset reached 5 records.
- Uniform was used as a query offset distribution, as well.

The primary index of Couchbase allows it to query any field of a document; however, this type of querying is rather slow. For the sake of fast query execution, secondary indexes are created for specific fields by which data is filtered. Couchbase provides two index storage modes—memory-optimized and disk-optimized (standard) ones.

Memory-optimized indexes use an in-memory database with a lock-free skip list, which has a probabilistic ordered data structure and, thus, performs at in-memory speeds. The search is similar to a binary search over linked lists with the `O(log n)` complexity. The lock-free skip list is used to provide non-blocking reads/writes and maximize utilization of the CPU cores. On top of a lock-free skip list, there is a multi-version manager responsible for regular snapshotting in the background. Memory-optimized indexes reside in memory and thus require the amount of RAM available to fit all the data inside of it. The indexes on a given node will stop processing further mutations, if a node runs out of index RAM quota. The index maintenance is paused until sufficient memory becomes available on the node. Since the data set was required to fit the available memory, memory-optimized indexes fit the requirements well.

Memory-optimized global secondary indexes were created for filtering fields with index replication on each cluster node.

```
CREATE INDEX `ycsb_address_country` ON `ycsb` (address.country)
USING GSI WITH {"nodes": [...]}
```

MongoDB uses mongos instances to route queries and operations to shards in a sharded cluster. If the result of the query is not sorted, the mongos instance opens a result cursor that “round robins” results from all cursors on the shards. If a query limits the size of the result set using the `limit() cursor` method, the mongos instance passes that limit to the shards and then reapplies the limit to the result before returning it to the client. If a query specifies a number of records to skip using the `skip() cursor` method, the mongos cannot pass the skip to the shards. Instead, the mongos retrieves unskipped results from the shards and skips the appropriate number of documents when assembling the complete result. However, when used in conjunction with `limit()`, the mongos will pass the limit plus the value of `skip()` to the shards to improve the efficiency of these operations.

For better performance, an additional secondary index was added to a filtered field:

```
db.customer.ensureIndex( { "address.country": 1 } );
```

Data filtering is not a typical case for Cassandra, as the database is designed to be queried by a primary key. The data model of Cassandra should be transformed for competitive results. In this case, this result cannot be compared with the other databases. Based on the above, it was decided that Cassanda would not be part of this workload.

4.3.2 Query

The following queries were used to perform Pagination Workload.

Table 4.3.1 Evaluated queries

Couchbase N1QL	MongoDB Query	Cassandra CQL

`SELECT RAW meta().id`
`FROM `ycsb``
`WHERE address.country='$1'`
`OFFSET $2`
`LIMIT $3`
	`db.customer.find({`
	`  address.country: $1`
	` }, {`
	`   id: 1`
	` })`
	` .skip($2)`
	` .limit($3)`
		Not applicable

4.3.3 Evaluation results

MongoDB and Couchbase Server performed similarly on the first two cluster types. On a 4-node cluster, Couchbase reached an average 29,840 ops/sec throughput with a latency around 8–10 ms versus 19,450 ops/sec for MongoDB. In addition to that, Couchbase had great scalability with 61,505 ops/sec throughput on a 10-node cluster. (The throughput increased with a cluster size growth due to the Index Service replication and load balancing.) MonogDB had 57,570 ops/sec with a latency of 10–13 ms on a 10-node cluster. For Couchbase, the use of memory-optimized indexes resulted in pretty high performance of filter operation with offset and limit applied. Couchbase significantly outperformed MongoDB on a 20-node cluster. On a

20-node cluster with a data set of 100 million records, Couchbase's throughput reached up to 96,075 ops/sec. It is 40% more than on a 10-node cluster (keeping the workload clients to five nodes and number of threads unchanged to 700 throughout all the tests) with a data set of 50 million. Meanwhile, MongoDB had 50,375 ops/sec with a latency of 10 ms.

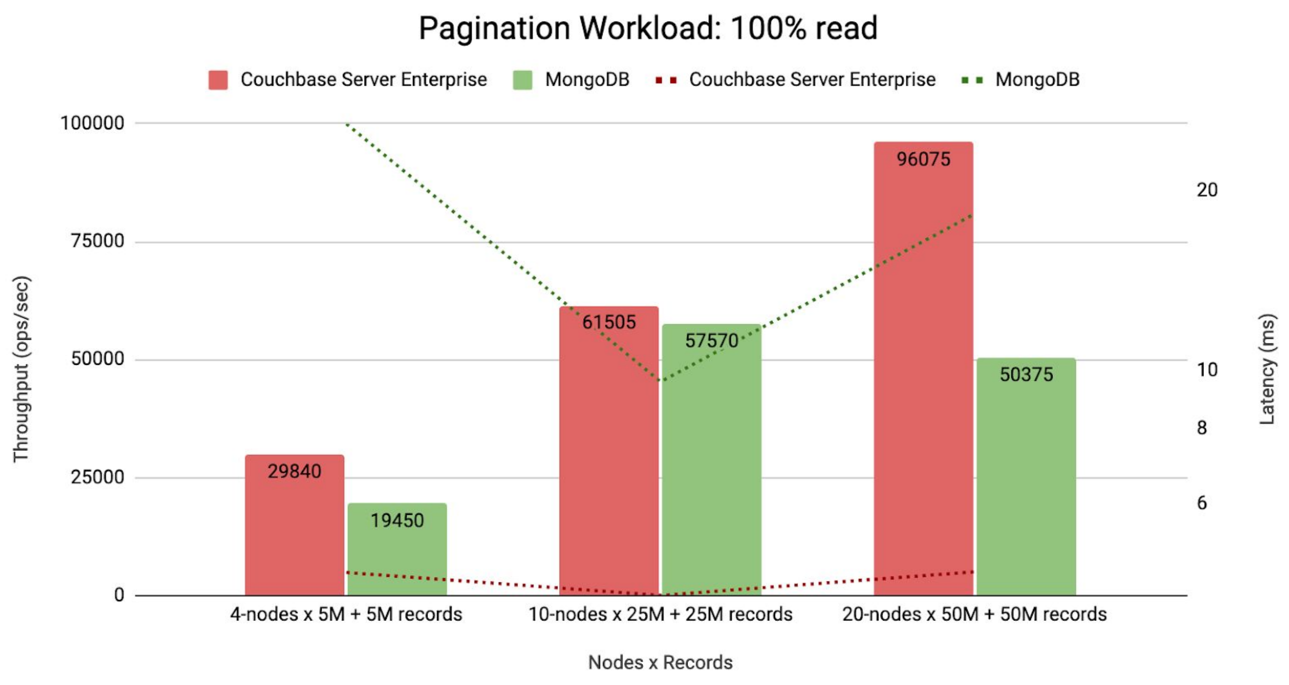

Figure 4.3.2 Performance results under Pagination Workload on 4-, 10-, and 20-node clusters

4.3.4 Summary

Couchbase exhibits twice the throughput and relatively low latencies for filter operation compared to MongoDB in the largest cluster. MongoDB remained flat with its performance even when the cluster size scaled. Meanwhile, Couchbase scaled linearly due to memory-optimized indexes and an out-of-the-box load balancing and scaling of Query and Index Services.

4.4 JOIN Workload: JOIN operations with grouping and aggregation

4.4.1 Workload definition and model details

JOIN Workload is a `JOIN` query with grouping and ordering applied. The workload simulates a selection of complex child-parent relationships with categorization employed. The scenario was executed under the following settings.

- The read ratio was 100%.
- The size of a data set was scaled in accordance with the cluster size: 5 million *customers* and 5 million *orders* (each 4.5 KB in size) on a 4-node cluster, 25 million *customers* and 25 million *orders* on a 10-node cluster, and 50 million *customers* and 50 million *orders* on a 20-node cluster.
- The maximum of a query length reached 100 records.

- Uniform was used as a query length distribution.
- The maximum of a query offset reached 5 records.
- Uniform was used as a query offset distribution, as well.

There are different types of `JOIN` operations available in the N1QL query engine in Couchbase out of the box:

- `Index JOIN` is used when one side of `JOIN` has to be document key(s) employing the `ON KEYS` statement.
- `ANSI JOIN` is applicable to arbitrary expressions on any field in a document, standard `JOIN` statement, with a nested loop under the hood. N1QL supports the standard `INNER`, `LEFT OUTER`, `RIGHT OUTER` `JOIN`s.
- `ANSI HASH JOIN` creates an in-memory hash table for one side of the `JOIN` operation (usually, the smaller one) used by the other side to find matches. It can be a performance optimization under suitable conditions.

Only the first two types—`Index JOIN` and `ANSI JOIN`—were evaluated during this benchmark. In addition to that, a dedicated covering index was used as it contained all the fields required by the query. This way, a query engine skips the whole document retrieval from data nodes after the index selection is made. Therefore, the query execution plan only consists of the index's resolution without a time-consuming document retrieval over the network, which results in a significant query performance boost.

The following covering index has been created:

```
CREATE INDEX `ycsb__address_month_orders_price` ON `ycsb`
(address.zip, month, order_list, sale_price)
USING GSI WITH {"nodes": [...]}
```

MongoDB ensures the `$lookup` aggregation out of the box to apply a left outer `JOIN` over an unsharded collection in the same database. It helps to filter document keys from the "joined" collection for further processing. Unfortunately, MongoDB v3.6 did not support the `$lookup` aggregation on sharded collections when the evaluation was carried out. So, in order to evaluate the JOIN Workload, an alternative solution was employed. The one way to work with multiple `JOIN` operations on a non-relational database is to denormalize a data model, embed the elements into the parent objects, and perform a regular query. Still, this approach invokes additional redundancy and extra storage costs, as well as impacts the read/write performance.

Another way is to model the dedicated "joining table" and query its elements by a partition key, which generally becomes identical to *read by key*. This approach leads to data duplication and an increase in write complexity through the necessity to support consistency between models, which also causes a significant write-performance downgrade. Furthermore, the approach brings along additional storage costs. The same specific data modeling approach can be applied to all the databases under evaluation, but it drives to dramatically varying results. This is the reason why we were considering a similar business case with two different models available: *customers* and *orders*. In this case, the `JOIN` operation was a simple two-phase read with filtering, which had a significant impact on the overall `JOIN` operation performance.

Cassandra also does not have an out-of-the-box `JOIN` operation support. The alternative solutions provided for MongoDB are applicable to Cassandra, as well. However, the two-phase read approach does not fit the Cassandra paradigm and appears to be non-scalable and non-performant, as it requires the usage of secondary indexes and, therefore, does not work on large data sets. For this reason, the second approach—with modeling an extra joining table—was evaluated keeping in mind all the drawbacks and side effects it brings.

As it resulted in querying by a partitioning key with the `SUM` aggregation and the corresponding read performance, the approach was excluded from the further comparison, because we were not evaluating the partition-key reads only. In terms of the read data, the performance under the dedicated joining table approach reached about 59,000 ops/sec on a 4-node cluster, about 159,000 ops/sec on a 10-node cluster, and up to 253,000 ops/sec on a 20-node cluster.

4.4.2 Query

The following queries were used to perform the *JOIN Workload*.

Table 4.4.1 Evaluated queries

Couchbase N1QL	MongoDB Query	Cassandra CQL
`SELECT o2.month, c2.address.zip, SUM(o2.sale_price) FROM `ycsb` c2 INNER JOIN `ycsb` o2 ON (META(o2).id IN c2.order_list) WHERE c2.address.zip = $1 AND o2.month = $2 GROUP BY o2.month, c2.address.zip ORDER BY SUM(o2.sale_price)`	`$r1 = db.customer.find({ address.zip: $1 }, { address.zip: 1, order_list: 1 }) $r2 = db.order.aggregate([ { $match: { $and: [{ id: { $in: $r1.order_list } }, { month: $2 }] }}, { $group: { id: null, sum: { $sum: "$sale_price" }}}])`	Not applicable

4.4.3 Evaluation results

Couchbase indexes and `ANSI JOIN` operations showed consistent performance on all cluster topologies as the number of documents scaled and the cardinality grew from 100 to 500 qualified documents per query. In general, Couchbase significantly outperformed MongoDB regardless of data set and cluster sizes thanks to different types of `JOIN` operations available out of the box. Couchbase was able to execute around 1,850 ops/sec on a 4-node cluster (with a data set of cardinality 100) at an average request latency of about 100 ms (using 700 client threads). On a 10-node cluster (with a data set of cardinality 250), Couchbase performed at around 250 ops/sec at an average request latency of 3,000 ms (using 700 client threads). Finally, the database reached about 995 ops/sec with a latency around 500 ms (using 700 client threads) on a 20-node cluster (with a data set of cardinality 500).

MongoDB demonstrated modest results in comparison to Couchbase Server. The 4-node cluster had 145 ops/sec with 10,000 ms latency, the 10-node cluster—65 ops/sec with 9,000 ms, and the 20-node cluster—45 ops/sec with 15,000 ms.

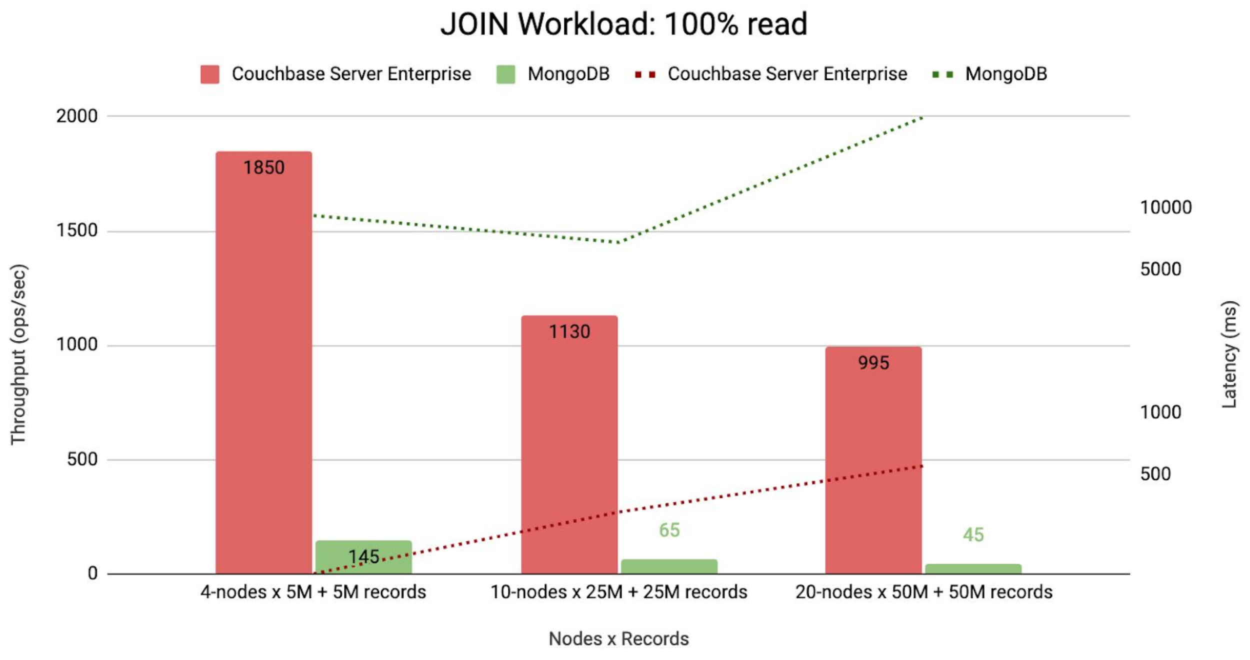

Figure 4.4.3.1 Performance results under JOIN Workload on 4-, 10-, and 20-node clusters

4.4.4 Summary

Couchbase is the only system under evaluation to support `JOIN` operations (provided by the query engine) out of the box. Both `Index JOIN` and `ANSI JOIN` scaled almost linearly and demonstrated an ability to handle increasing amounts of data at scale. The disproportional data cardinality made the throughput of JOIN Workload appear flat even when the number of documents processed per second scaled almost linearly. This indicated that proper data cardinality is also vital when it comes to generating data randomly for these benchmarks.

MongoDB provides the `$lookup` aggregation stage, which is a `LEFT OUTER JOIN` equivalent. However, the `$lookup` aggregation was available only for unsharded collections when this benchmark was conducted, so this option was not evaluated. The "read parent–read dependencies" solution performed rather poorly and appeared to be non-scalable. Thus, for `JOIN` queries there seem to be no alternatives other than Couchbase.

5. Conclusion

No single NoSQL database can perfectly fit all the requirements of any given use case. Every system has its advantages and disadvantages that become more or less important depending on the specific criteria to meet.

First of all, it should be noted that all the workloads were executed with the assumption of the data set fitting the available memory. Taking this into account, all the reads from Data and Index Services for Couchbase were from RAM, thus, performed at in-memory speeds.

With the same amount of available RAM, DataStax Enterprise (Cassandra) did not allow storing everything in cache. Therefore, the majority of the reads were made from disk.

Couchbase showed good performance across all the evaluated workloads, providing functionality sufficient to handle the deployed workloads out of the box and requiring no in-depth knowledge of the database's architecture. Furthermore, the query engine of Couchbase supports aggregation, filtering, and `JOIN` operations on large data sets without the need to model data for each specific query. As clusters and data sets grow in size, Couchbase ensures a satisfactory level of scalability across these operations.

MongoDB produced comparatively decent results on relatively small clusters. MongoDB is scalable enough to handle increasing amounts of data and cluster extension. Under this benchmark, the only major issue we observed was that MongoDB did not support `JOIN` operations on sharded collections. Nevertheless, dedicated data modeling provided an alternate solution, though, with a negative impact on performance.

DataStax Enterprise (Cassandra) demonstrated good performance for intensive parallel writes and reads by a partition key and, as expected, failed on non-clustering key-read operations. In general, we proved that Cassandra is able to show great performance for write-intensive operations and reads by a partition key. Still, Cassandra is operations-agnostic and behaves well only if multiple conditions are satisfied. For instance, reads are processed by a known primary key only, data is evenly distributed across multiple cluster nodes, and finally there is no need for `JOIN` operations or aggregates.

6. About the Authors

Uladzislau Kaminski is a Senior Software Engineer and Cloud-Native Development Consultant.

His primary skills are software architecture and system design. He took part in numerous projects dealing with processing and distributing huge amounts of data arrays. Uladzislau has a durable background in building systems from scratch and adapting existing solutions, as well as designing, analyzing, and testing them.

Artsiom Yudovin is a Tech Lead of Data Engineers.

He has a solid software development background. He is focused on maintaining, designing, customizing, upgrading, and implementing complex software architectures, including data-intensive and distributed systems. Artsiom dedicates much of his spare time to these activities, and now he is one of the contributors to well-known open-source projects.

Ivan Shyrma is a Data Engineer.

He has extensive hands-on experience in high-load, scalable applications and web services development. Ivan has worked as a full-stack engineer for several years designing durable distributed systems. He is able to create complex architecture solutions, adopt systems for production use, and is keen on resolving any engineering problems.

Sergey Bushik is a Lead Software Engineer.

He has extensive experience in multilayered application architecture and high-level design. Sergey is an expert in relational databases with experience in J2EE technologies and Java frameworks. He also has a background working with NoSQL and NewSQL storage systems, as well as with stream processing frameworks.

Want more?

To download other research papers and articles like that:

- check out our resources page
- subscribe to the blog
- or follow @altoros for daily updates

Feel free to contact us if you'd like to discuss your project.