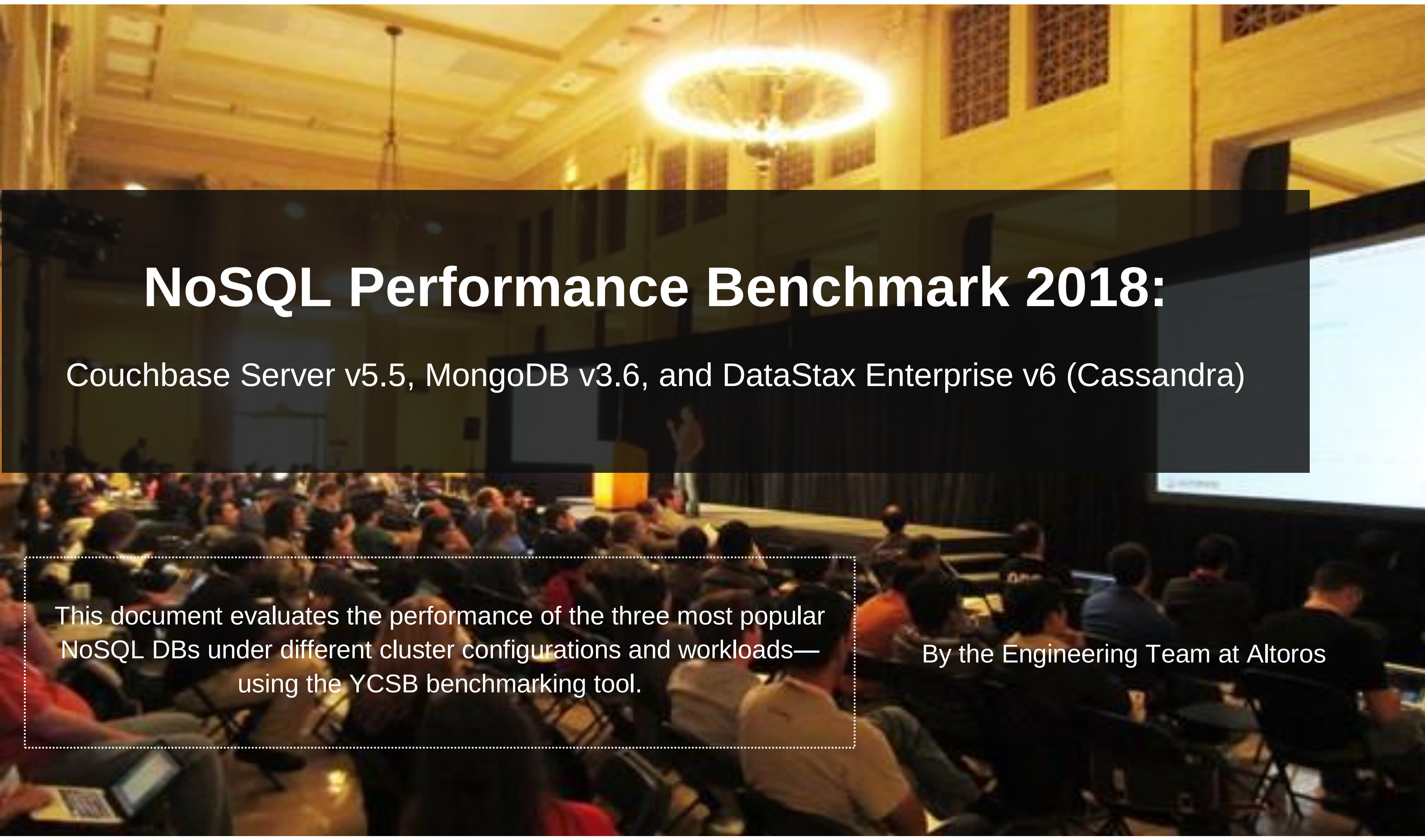

NoSQL Performance Benchmark 2018:

Couchbase Server v5.5, MongoDB v3.6, and DataStax Enterprise v6 (Cassandra)

This document evaluates the performance of the three most popular NoSQL DBs under different cluster configurations and workloads—using the YCSB benchmarking tool.

By the Engineering Team at Altoros

Q3 2018

Table of Contents

1. Introduction

NoSQL encompasses a wide variety of database technologies that were developed in response to the rise in volume of data and the frequency with which this data is accessed. In contrast, relational databases were not designed to cope with 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 solutions help to achieve the highest levels of performance and uptime for workloads.

This report compares performance results of the three popular NoSQL databases: Couchbase Server v5.5, MongoDB v3.6, and DataStax Enterprise v6 (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—namely, on 4, 10, and 20 nodes—and under four different workloads.

The first workload performs under an *update-heavy* mode, invoking 50% of reads and 50% of updates. The second one is a *short-range scan* workload, invoking 95% of scans and 5% of updates, where short ranges of records are queried instead of the individual ones. 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.

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

2. Testing Environment

2.1 Hardware configuration

Each of the NoSQL DBs was deployed on 4-, 10-, and 20-node clusters in the same geographical region. The clusters were deployed on Amazon's 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
Operating system	Ubuntu 16.04 LTS

To provide verifiable results, benchmarking was performed on Amazon Elastic Compute Cloud instances. The YCSB client was deployed to five Amazon's 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	c4.4xlarge
vCPUs	8
Memory (GiB)	15
EBS-optimized available	Yes
Network performance	High
Platform	64-bit
Operating system	Ubuntu 16.04 LTS

2.2 Couchbase Server cluster configuration

Couchbase Server is both a JSON-document and key-value distributed NoSQL database. It guarantees high performance with a built-in object-level cache, asynchronous replication, and data persistence. The database is designed to scale out or up compute-, RAM-, and storage-intensive workloads independently.

Here, we evaluated Couchbase Server Enterprise Edition 5.5.0, build 2940. Our team used a symmetric scale-out strategy, giving each node an equal share of work. Regardless of cluster size (4, 10, or 20 nodes), each node comprised Data Service, Index Service, and Query Service. Search, Analytics, and Eventing Services were turned off, and no resources were allocated for them, because the corresponding features were not the point of interest for this benchmark. Each Data Service was allocated 60% of available RAM (36,178 MB) with a Couchbase bucket type used. Each bucket had a single replica configured. Index Service was allocated about 40% of available RAM (about 24 GB) with memory optimized indexes in use. Each created index 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 puts forward a separate and incremental approach to data replication and partitioning, which happen as completely independent processes.

MongoDB v3.6 was under evaluation. 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 installation and configuration for a MongoDB sharded cluster is a fairly complicated procedure. In short, one needs to satisfy installation prerequisites, then separately configure all the data shards, configuration servers, and sharding routers to finally join those components into a cluster.

MongoDB distributes data, or shards, at the collection level. Sharding partitions the collection's data 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.

This benchmark evaluates DataStax Enterprise v6. In the table below, we detailed the changes applied to each node on 4-, 10-, and 20-node clusters.

Table 2.4 The changes applied to each node on each cluster

cassandra.yaml	
memtable_heap_space_in_mb	16,384
memtable_cleanup_threshold	0.11
memtable_flush_writers	40
row_cache_size_in_mb	20,280
commitlog_total_space_in_mb	1,969
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

3. Workloads and Tools

Database performance was defined by the speed at which a 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 module.

3.1 Workloads

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

1) *Workload A:* update heavily—50% read and 50% update, request distribution is Zipfian.
2) *Workload E:* scan short ranges—95% scan and 5% update, request distribution is Uniform.
3) *Workload Query 1:* filter with offset and limit.
4) *Workload Query 2:* `JOIN` operations with grouping and aggregation (in case of Couchbase, `ANSI JOIN` was evaluated, as well).

3.2 Tools

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

- Workload executor
- The YCSB client threads
- Extensions
- Statistics module
- Database connectors

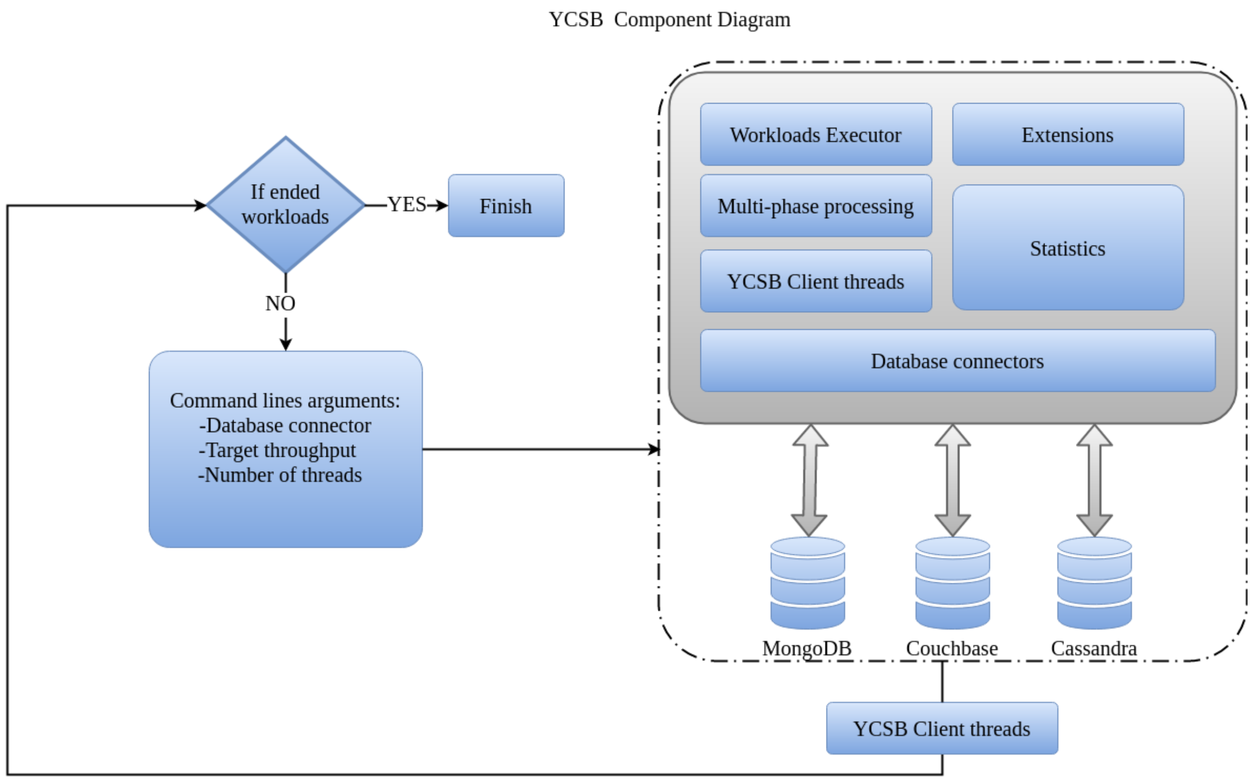

Figure 3.1 The components of the YCSB client

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.

Workloads *A* and *E* are standard workloads provided by YCSB. Default data models were used for this workloads. Workloads *Query 1* and *Query 2* 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 this scenarios on a domain level, customer-order model was introduced for this workloads:

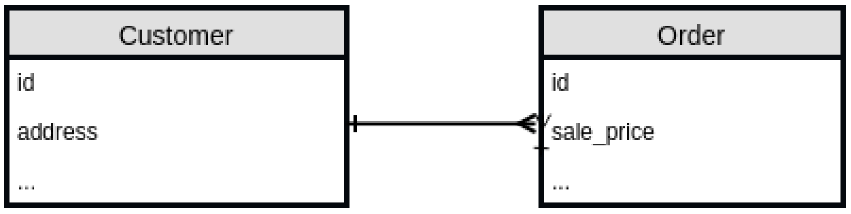

Figure 3.2 A graphic representation of the customer-order model

4. Performance 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 scaled in accordance with the cluster size: 50 million records (1 KB in size each, consisting of 10 fields and a key) on a 4-node cluster, 100 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 way, because a query request is sent directly to a proper vBucket that holds 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

We used the following queries to generate Workload A.

Table 4.1 Evaluated queries for Workload A

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. The database was able to process up to 208,000 ops/sec on a 4-node cluster, while Cassandra handled about 77,500 ops/sec and MongoDB only 41,900 ops/sec. On a 10-nodes cluster, Couchbase achieved about 302,000 ops/sec, MongoDB about 102,000 ops/sec, and Cassandra about 128,000 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 marginally improved to 320,000 ops/sec, whereas MongoDB performance grew to 177,500 ops/sec, and Cassandra to almost 200,000 ops/sec.

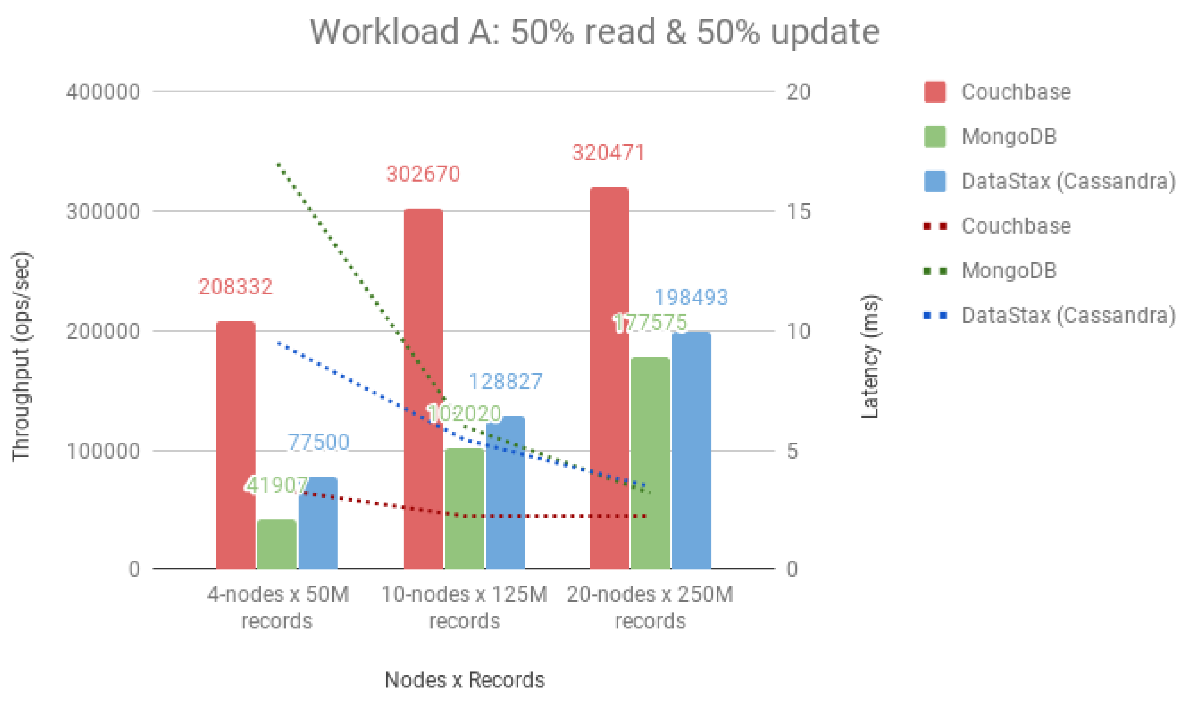

Figure 4.1.3 Performance results under Workload A on 4-, 10-, and 20-node clusters

Couchbase exhibits the consistency of request latency on a 10-node as well as on a 20-node cluster, processing 700 calling threads in 2.24 ms. MongoDB scaled well with continuously decreasing request processing time—from 6 ms on a 10-node cluster to 3.2 ms on a 20-node cluster with the same amount of calling threads.

A 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, therefore can impact performance while the procedure takes place. On a bigger clusters, the balancer has less impact on performance, because the data chunks are distributed across more nodes, therefore the migration thresholds are reached more rarely.

DataStax (Cassandra) appeared to be scaling well with the constantly decreasing request latency—from 9.5 ms on a 4-node cluster to 5.45 ms on a 10-node cluster, and 3.47 ms on a 20-node cluster. Still, it underperformed as compared to Couchbase, which exhibited 50% better throughput and lower latency on 20-node cluster.

It should be also noted that both Couchbase and Cassandra did not hit the maximum throughput at 700 client threads, but the throughput was growing constantly with the latency remaining relatively low. To evaluate the ability to handle more clients with a higher throughput, additional tests on a 10-client environment against a 4-node cluster were carried out. On a 10-client environment—each running up to 150 threads—Cassandra achieved its limit of about 83,000 ops/sec at 1,200 calling threads and stopped growing, handling less than half the requests compared to Couchbase.

4.1.4 Summary

In the above configurations, Couchbase v5.5 exhibits much better (up to 4x better latency and up to 3x better throughput) performance at scale than MongoDB v3.6 and DataStax Enterprise v6 (Cassandra). Meanwhile, MongoDB reaches its limit at about 500–700 threads and does not scale further. Both MongoDB and DataStax Enterprise (Cassandra) show consistent improvement in the overall throughput proportionally to the cluster size growth. For larger cluster sizes, we observed that five client nodes (that we kept consistent throughout our tests) were not enough to fully saturate a 20-node Couchbase, MongoDB, and Cassandra cluster, therefore we got a marginal performance improvement than the cluster is capable of delivering.

For more detailed comparison results, please refer to *Figures A.1.1–A.1.3* in the "*A.1 Workload A: The update-heavy mode*" section of *Appendix A*.

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 is for 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 scaled in accordance with the cluster size: 50 million records (1 KB in size each, 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.

Because 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 a 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 records to 125 million records, the search time increases by 5%. This issues 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 this database: 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 the *token* function. The return result of a scan operation depends on the selected partitioner. For this benchmark, the default Murmur3Partitioner was used. However, Murmur3Partitioner simply calculates a key hash, but does not preserve ordering—which may result in an unexpected return of a scan operation. Additionally, Cassandra does not support any ordering by the partitionary key.

4.2.2 Query

The following queries were used to perform Workload E.

Table 4.2 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 proportionally to the number of cluster nodes: from 10,500 ops/sec on a 4-node cluster to 23,500 ops/sec on a 10-node cluster. On a 20-node cluster, the throughput reached 32,000 ops/sec, which is about 36% more than on a 10-node cluster, with the request latency remaining about 10 millisecond due to the usage of a primary index and the replication of Index Service.

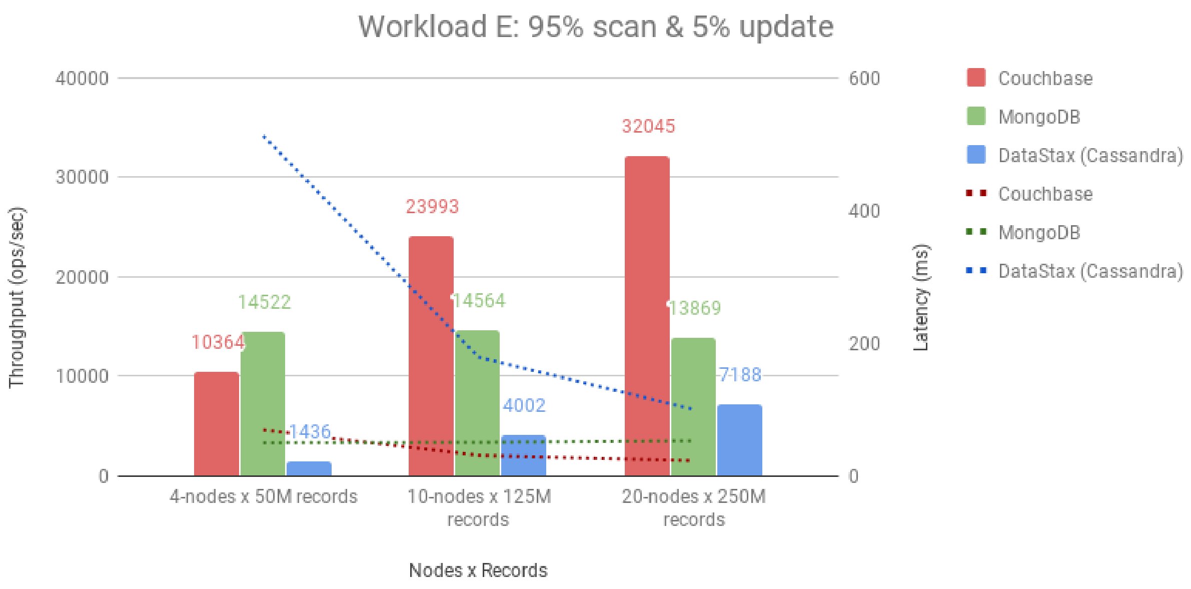

Figure 4.2.3 Performance results under Workload E on 4-, 10-, and 20-node clusters

In contrast, MongoDB was able to process about 14,000 ops/sec regardless of cluster and data set sizes. It is more than Couchbase was able to handle on a 4-node cluster and less than on 10- and 20-node clusters.

Cassandra showed rather low performance with the scan operation: around 1,400 ops/sec on a 4-node cluster, 4,000 ops/sec on a 10-node cluster, and around 7,000 ops/sec on a 20-node cluster. However, Cassandra was able to achieve a linear growth of performance across all clusters and data sets. This can be explained by the fact that the coordinator node sends scan requests to other nodes in a cluster responsible for specific token ranges. The more nodes the cluster has, the less data falls in the target range on each node, thus the less data each node has to return. It results in reduced per-node request processing time. As the coordinator sends the requests in parallel, the overall request processing time depends on a request latency of each cluster node—such a latency decreases with cluster growth. This is proved by the following brief analysis: the latency decreases from 511 ms on a 4-node cluster to 179 ms on a 10-node cluster, and to 100 ms on a 20-node cluster.

4.2.4 Summary

MongoDB performed around 30% better than Couchbase on relatively small sized clusters and data sets (4 nodes and 50 million of records each 1 KB in size) but remained flat irrespective of the cluster size. At the same time, Couchbase demonstrated much better scaling capabilities and outperformed MongoDB on bigger clusters by 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 provided better scalability in comparison to MongoDB and Couchbase, preserving linear performance growth, but still being way behind Couchbase and MongoDB in terms of overall operation performance.

For more detailed comparison results, please refer to *Figures A.2.1–A.2.3* in the “*A.2 Workload E: Scanning short ranges*” section of *Appendix A*.

4.3 The 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 scaled in accordance with the cluster size: 5 million *customers* (4 KB in size each) 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 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 to maximize utilization of 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 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 amount of 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 the 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, 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 following two data filtering strategies were evaluated:

- **Using a secondary index**. It is a default way of querying by a non-partitioning key in Cassandra. The index table for a secondary index is stored on each node in a cluster. So, querying via a secondary index can have an adverse impact on performance due to poor scalability.
- **Using synthetic sharding.** Under this approach, an extra field (e.g., `shard_id - [1-n]`) should be added to the primary key, which consists of `shard_id` and a filtering field. Using `shard_id`, we can control distribution between nodes. In such a case, any queries should be repeated `n` times across each `shard_id` to further aggregated a final result.

Also, Cassandra does not support the `OFFSET` operation, so it was emulated in the code.

4.3.2 Query

The following queries were used to perform *Workload Q1*.

Table 4.3 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)`
		`SELECT id`
		`FROM table`
		`WHERE`
		`address_country = $1`
		`LIMIT $2`

4.3.3 Evaluation results

For Couchbase, the use of memory optimized indexes resulted in pretty high performance of the filter operation with offset and limit applied. This way, Couchbase significantly outperformed MongoDB. On a 4-node cluster, Couchbase showed an average 36,700 ops/sec throughput with a latency around 8–10 milliseconds versus 11,000 ops/sec for MongoDB. In addition to that, Couchbase showed great scalability with 83,500 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.)

On a 20-node cluster with a data set of 100 million records, the throughput reached up to 95,000 ops/sec. It is 14% more than on a 10-node cluster (keeping the workload clients to 5 nodes and number of threads unchanged to 700 throughout all the tests) with a data set of 50 million. MongoDB displayed same performance regardless of data set and cluster sizes.

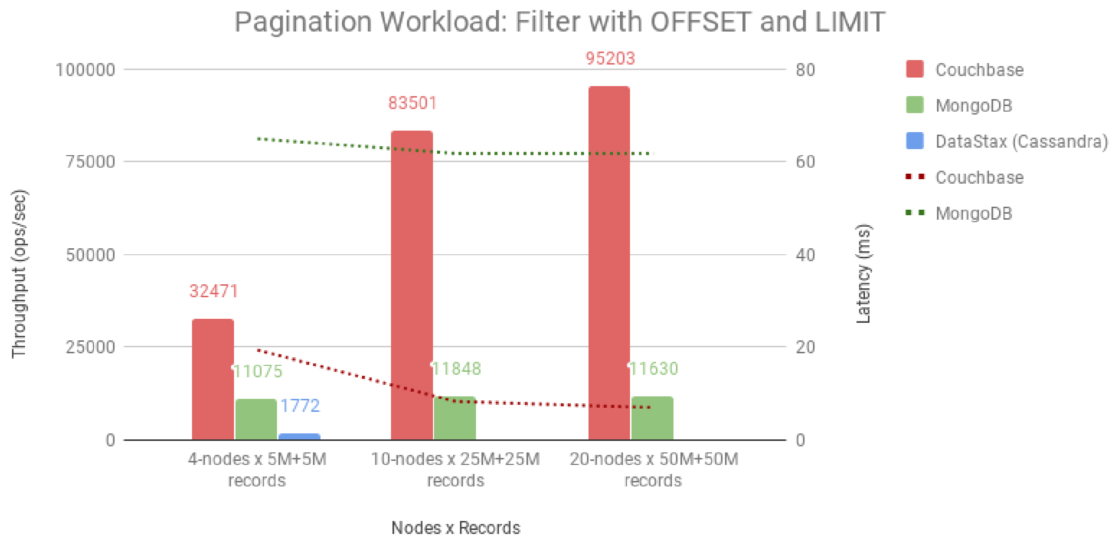

Figure 4.3.3 Performance results under the Pagination Workload on 4-, 10-, and 20-node clusters

Two filtering strategies were evaluated for Cassandra. The secondary indexes solution achieved up to 1,700 ops/sec on a 4-node cluster. On a 10-node cluster, the latency increased dramatically, and requests started failing with a timeout. In this case, further workload evaluation on a 20-node cluster was meaningless because of the cluster's inability to handle it. The second approach—synthetic sharding—resulted in around 50 ops/sec throughput on a 4-node cluster, so there was no sense in further evaluation on 10- and 20-node clusters.

4.3.4 Summary

Couchbase exhibits multiple times (8x to 9x) higher throughput and relatively low latencies for the filter operation compared to MongoDB and Cassandra. 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. Cassandra does not seem to be a good choice when a business scenario requires data filtering not using a primary key, as the database is not able to handle increasing loads.

For more detailed comparison results, please refer to *Figures A.3.1–A.3.3* in the "*A.3 Pagination Workload: Filter with OFFSET and LIMIT*" section of *Appendix A*.

4.4 The Join Workload: JOIN operations with grouping and aggregation

4.4.1 Workload definition and model details

Workload Query 2 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 scaled in accordance with the cluster size: 5 million *customers* and 5 million *orders* (4.5 KB in size each) 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 out-of-the-box with the N1QL query engine in Couchbase:

- `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`, and `RIGHT OUTER` joins.
- `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 provide performance optimization under suitable conditions.

Only the first two types—`Index JOIN` and `ANSI JOIN`—were evaluated under this benchmark. In addition to that, a dedicated covering index was used, because 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 indexes 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. One way to work with joins 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 a dedicated “joining table” and then query its elements by a partition key, which generally becomes identical to *read by key*. This approach leads to data duplication and 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, too. However, the two-phase read approach does not fit the Cassandra paradigm and appears to be non-scalable and non-performant, because it requires the use 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. Because 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 simulate the *Join Workload*.

Table 4.4 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”`
	`  }`
	`}}])`
		`SELECT month, zip,`
		`SUM(sale_price) FROM`
		`customer_orders_join`
		`WHERE zip = $1 AND`
		`month = $2`

4.4.3 Evaluation results

Couchbase indexes and ANSI JOINs 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 1,180 ops/sec on a 4-node cluster (with a data set of cardinality 100) at an average request latency of about 590 milliseconds (using 700 client threads).

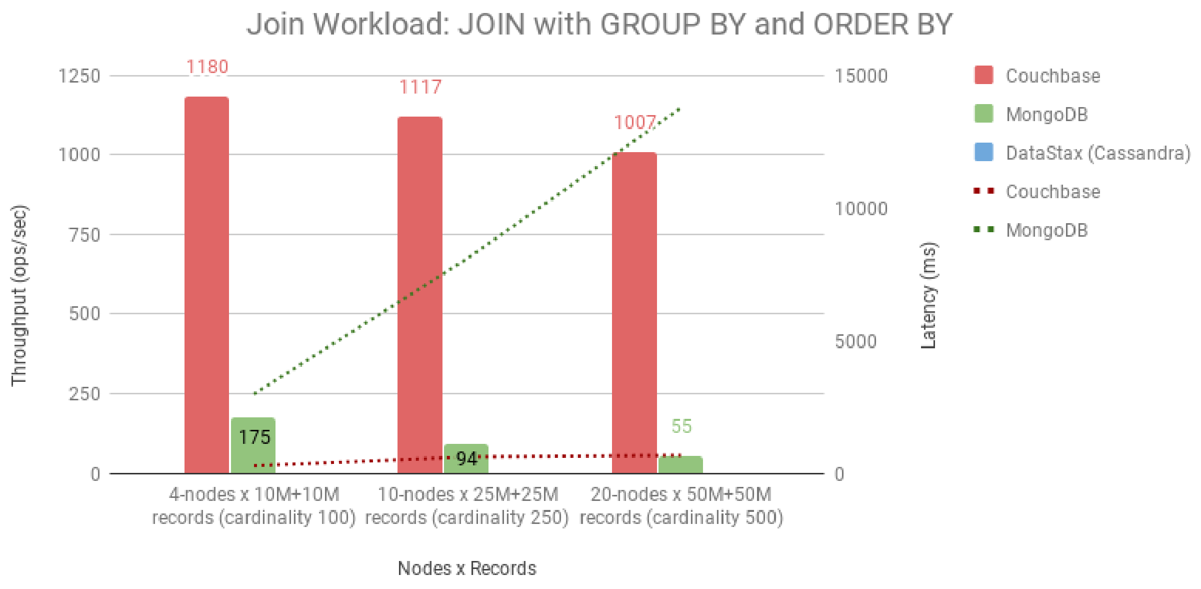

Figure 4.4.3.1 Performance results under Join Workload on 4-, 10-, and 20-node clusters

On a 10-node cluster (with a data set of cardinality 250), Couchbase performed at about 1,100 ops/sec at an average request latency of 650 milliseconds (using 700 client threads). Finally, the database reached about 1,007 ops/sec with a latency around 700 milliseconds (using 700 client threads) on a 20-node cluster (with a data set of cardinality 500).

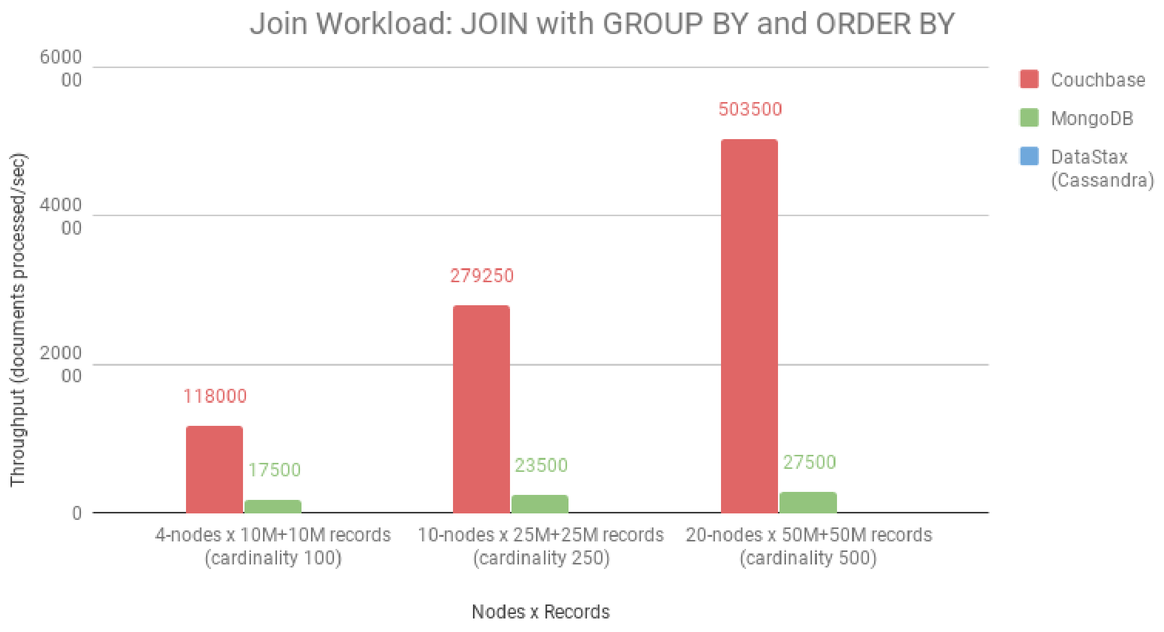

Figure 4.4.3.2 Number of documents processed under Join Workload on 4-, 10-, and 20-node clusters

To capture the true performance of this benchmark in the midst of varying cardinality, we also plotted a separate graph (*Figure 4.4.3.2*) that estimates the number of documents processed (throughput × cardinality) per second under each benchmark. This graph clearly highlights that Couchbase performance scaled linearly and—compared to MongoDB—its throughput grew from over 5x (on 4-node cluster) to 18x (on a 20-node cluster), which is pretty significant.

4.4.4 Summary

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

In its turn, 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.

For more detailed comparison results, please refer to *Figures A.4.1–A.4.3* in the “*A.4 The Join Workload: JOIN operations with grouping and aggregation*” section of *Appendix A*.

5. Conclusion

Hardly any NoSQL database can perfectly fit all the requirements of any given use case. Every solution has its advantages and disadvantages that become more or less important depending on specific criteria to meet.

First of all, it should be noted that all the workloads were executed with the assumption that a data set fits the available memory. With that in mind, all the reads from Data Service and Index Service for Couchbase were from RAM and thus performed on in-memory speeds.

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

Couchbase demonstrated good performance across all the evaluated workloads and appears to be a good choice, providing out-of-the-box functionality sufficient to handle the deployed workloads 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 one issue we observed was that MongoDB did not support `JOIN` operations on sharded collections out of the box. This way, dedicated data modeling provided a way out—however, with a negative impact on performance.

Cassandra provided rather 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 joins or aggregates.

6. About the Authors

This benchmark was performed by **Altoros**, a 300+ people strong consultancy that helps Global 2000 organizations with a methodology, training, technology building blocks, and end-to-end solution development. The company turns cloud-native app development, customer analytics, blockchain, and AI into products with a sustainable competitive advantage. Assisting enterprises on their way to digital transformation, Altoros stands behind some of the world's largest Cloud Foundry and NoSQL deployments. For more, please visit www.altoros.com.

To download more NoSQL guides and tutorials:

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

Appendix A

This section provides additional details for performance evaluation under the described workloads.

A.1 Workload A: The update-heavy mode

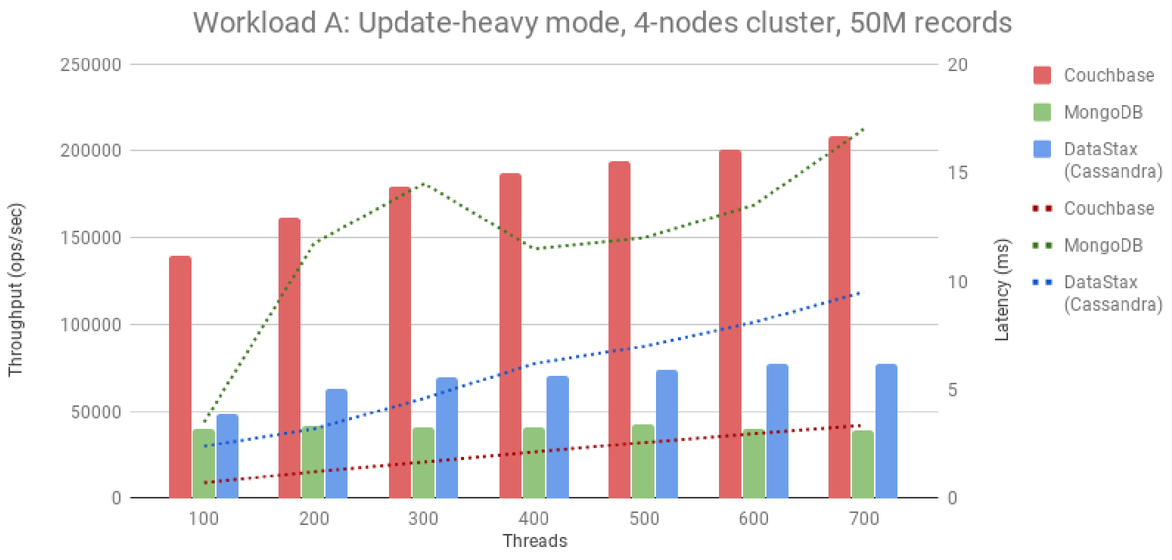

Figure A.1.1 Evaluation results under Workload A on a 4-node cluster

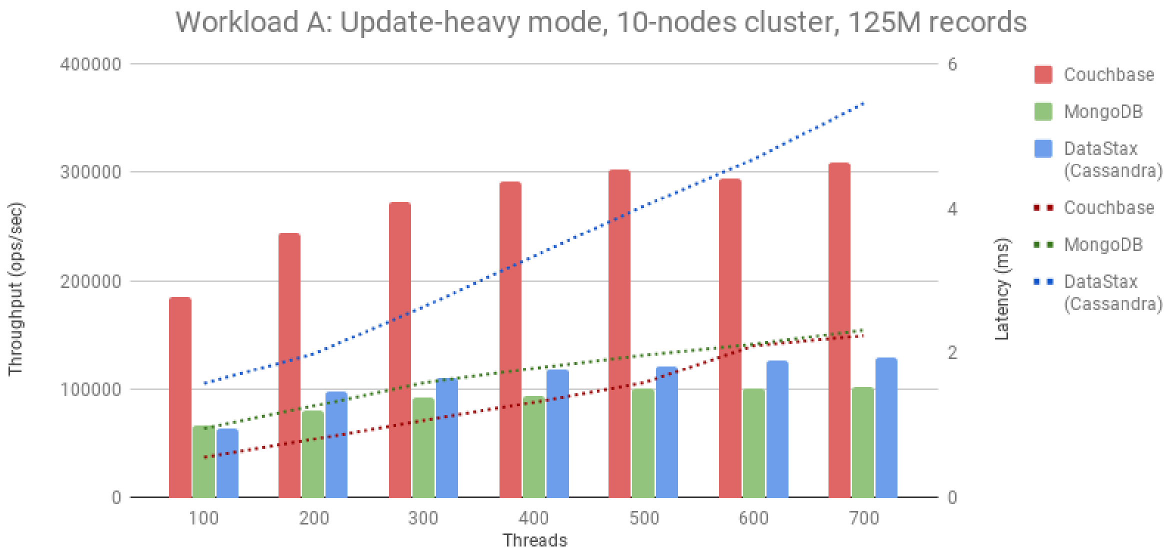

Figure A.1.2 Evaluation results under Workload A on a 10-node cluster

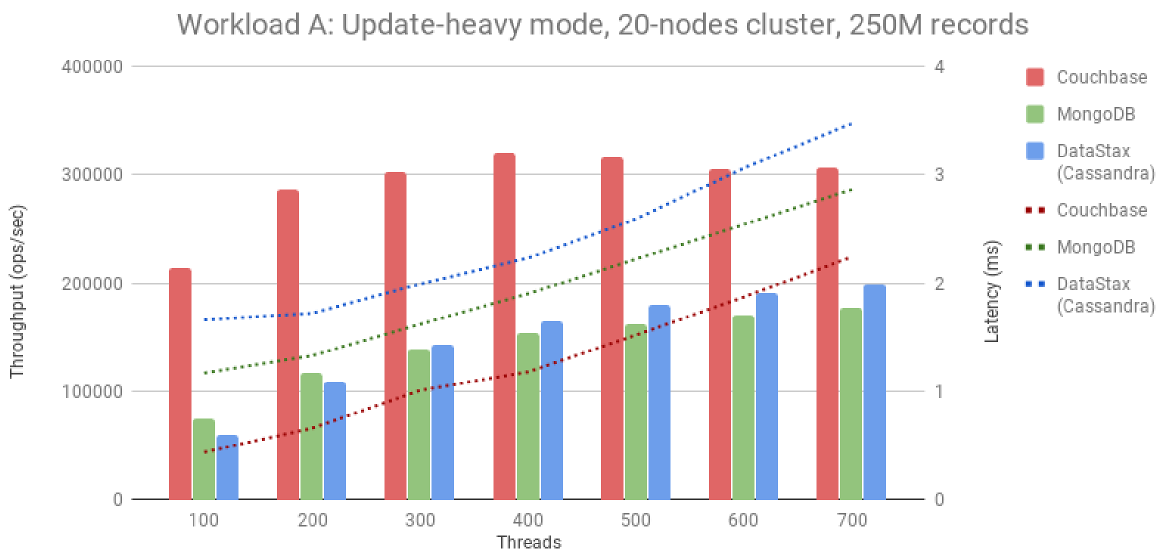

Figure A.1.3 Evaluation results under Workload A on a 20-node cluster

A.2 Workload E: Scanning short ranges

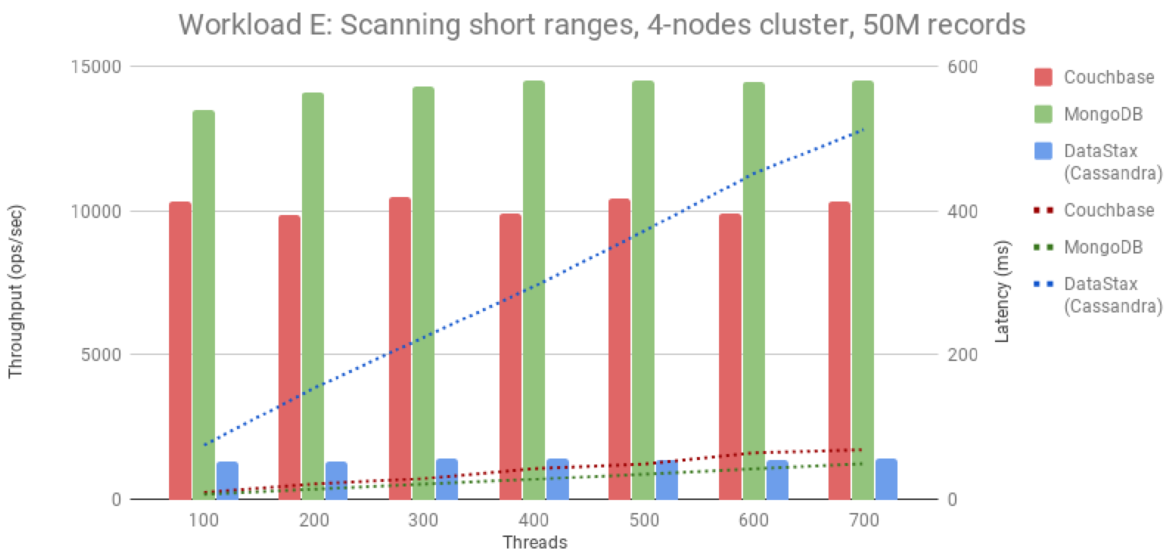

Figure A.2.1 Evaluation results under Workload E on a 4-node cluster

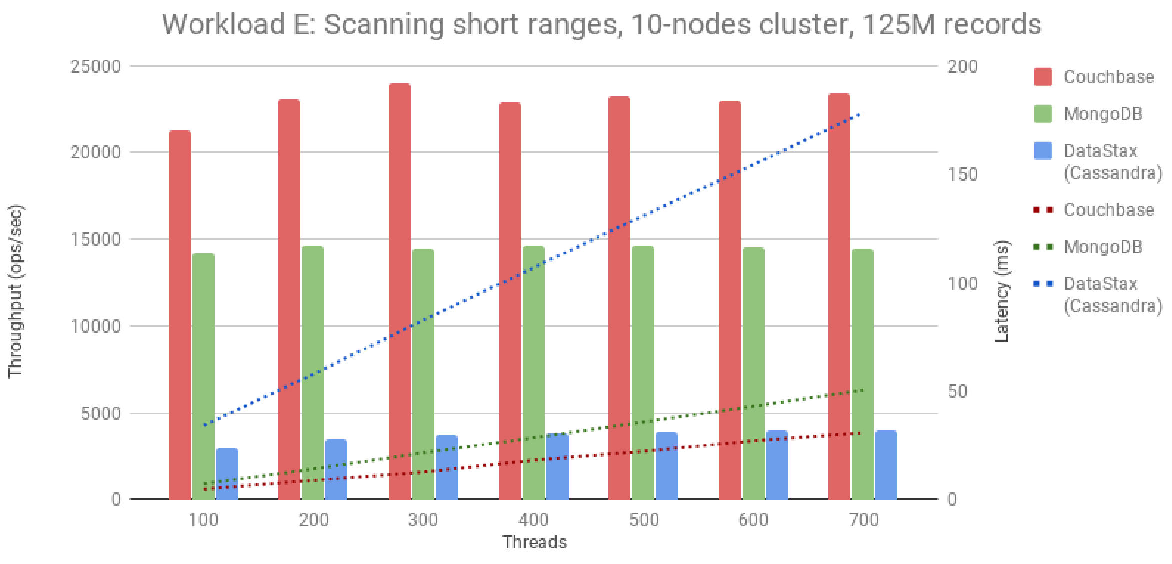

Figure A.2.2 Evaluation results under Workload E on a 10-node cluster

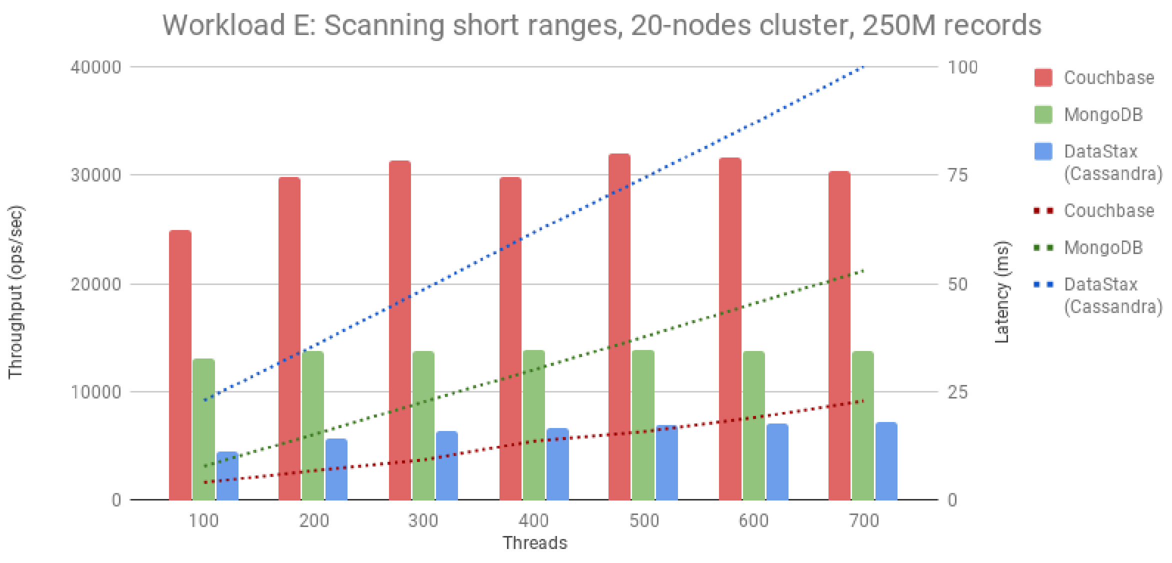

Figure A.2.3 Evaluation results under Workload E on 20-node cluster

A.3 The Pagination Workload: Filter with OFFSET and LIMIT

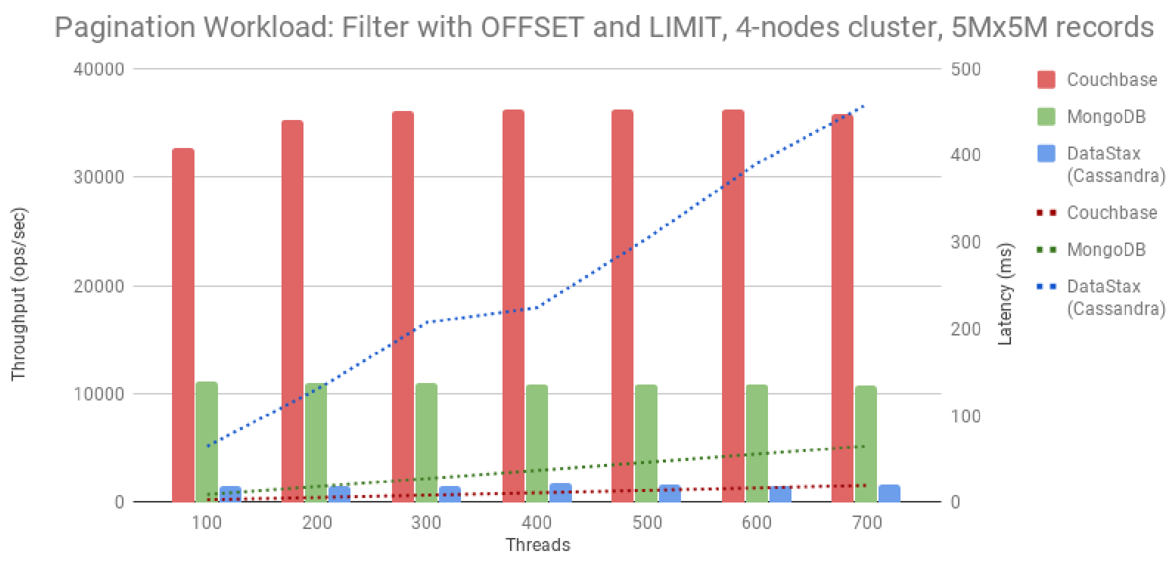

Figure A.3.1 Evaluation results under Pagination Workload on a 4-node cluster

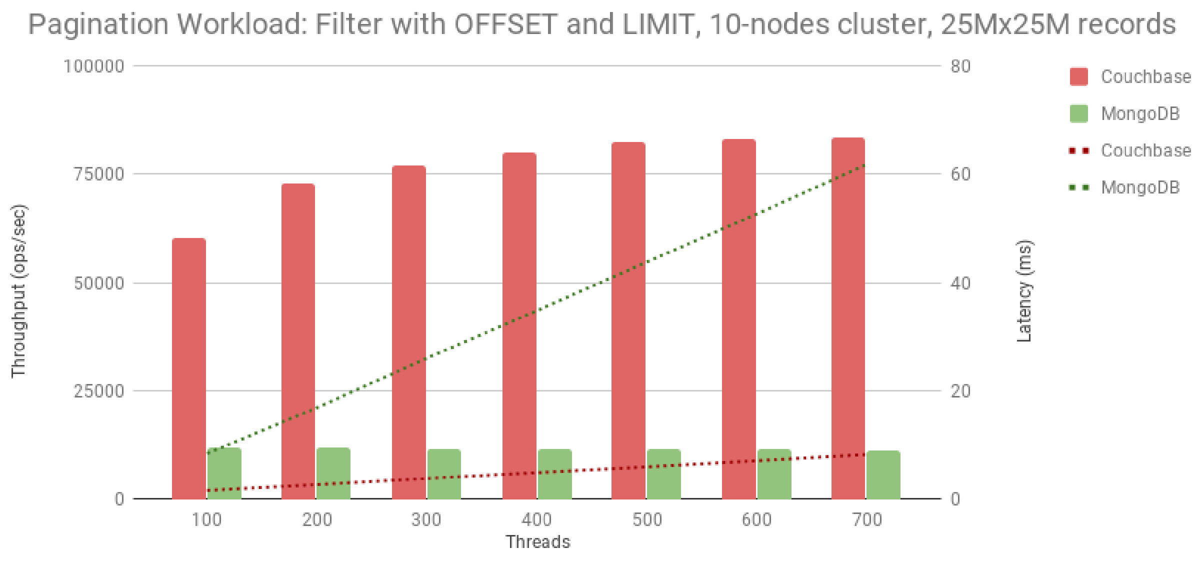

Figure A.3.2 Evaluation results under Pagination Workload on a 10-node cluster

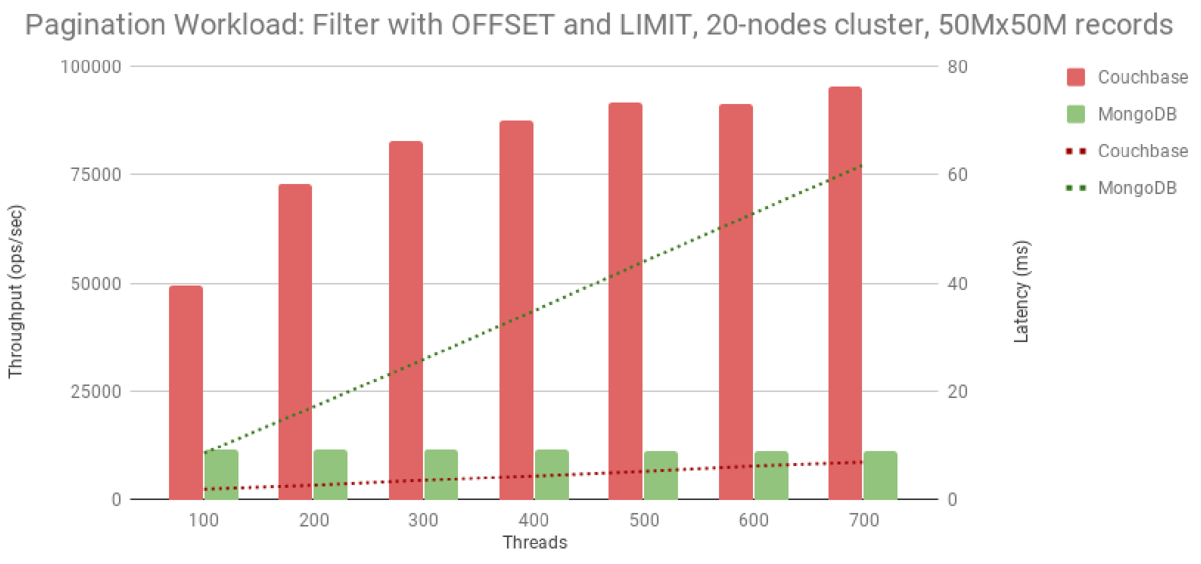

Figure A.3.3 Evaluation results under Pagination Workload on a 20-node cluster

A.4 The Join Workload: JOIN operations with grouping and aggregation

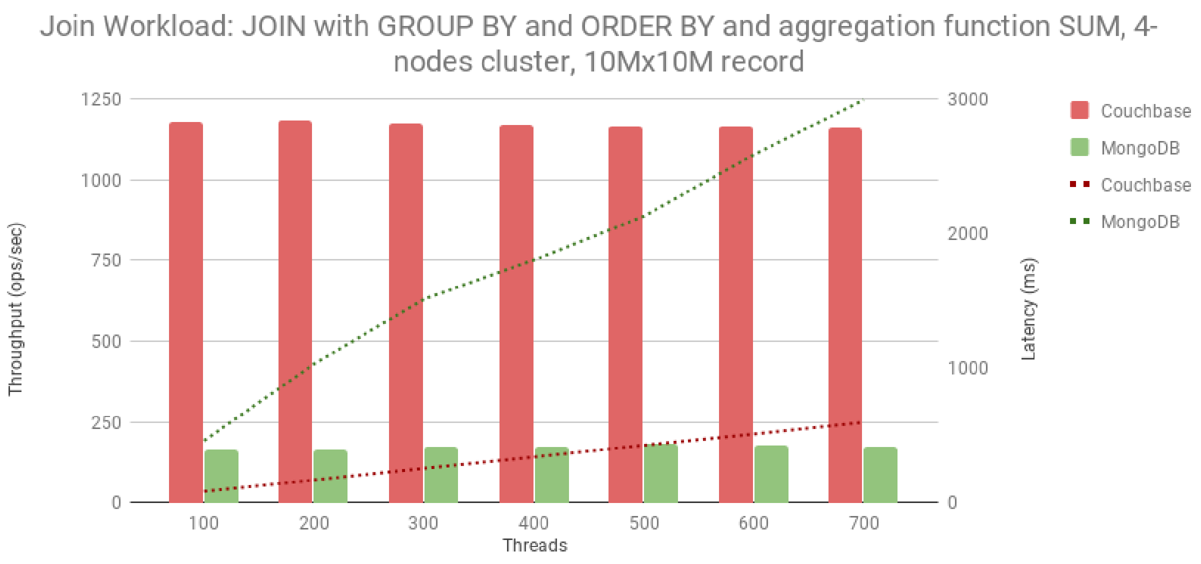

Figure A.4.1 Evaluation results under Join Workload on a 4-node cluster

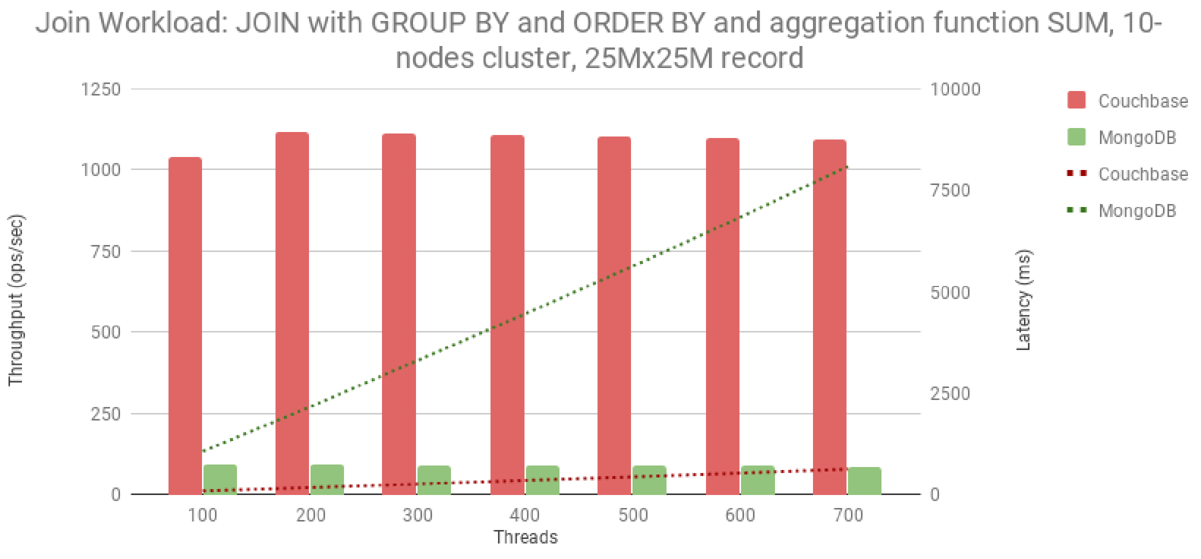

Figure A.4.2 Evaluation results under Join Workload on a 10-node cluster

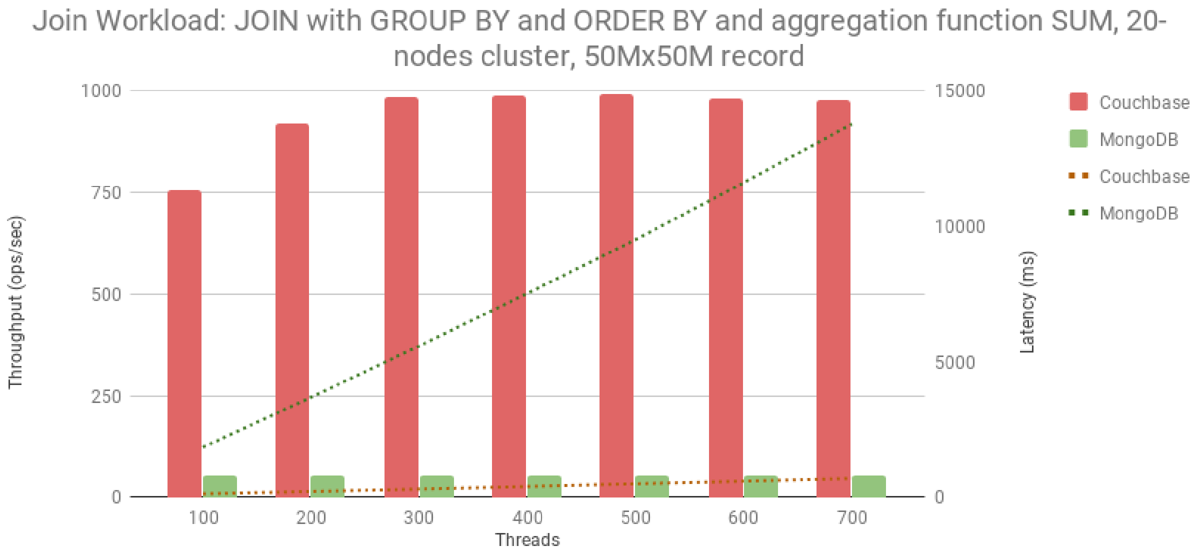

Figure A.4.3 Evaluation results under Join Workload on a 20-node cluster

Appendix B

In the comparative tables below, you will find performance results of the evaluated databases. The results demonstrate the amount of operations per second handled by each database in different percentiles. (A *percentile* is a measure that indicates the value below which a given percentage of observations in a group of observations fall.) Each workload iteration was executed for 10 minutes, aggregating statistics every second. Thus, each observation consisted of 600 measures.

Table B.1 99-, 95-, 50-, and 5-percentile for Workload A

(ops/sec)	Database		
	Couchbase	MongoDB	Cassandra
4 nodes			
99 percentile	271,255	96,700	80,380
95 percentile	238,470	84,030	76,900
50 percentile	206,515	39,945	57,905
5 percentile	190,880	1,630	42,085
10 nodes			
99 percentile	380,410	164,200	178,370
95 percentile	368,130	158,170	169,370
50 percentile	307,585	110,275	137,400
5 percentile	263,555	14,055	67,840
20 nodes			
99 percentile	395,725	206,590	266,625
95 percentile	388,955	203,250	257,325
50 percentile	307,520	182,550	213,170
5 percentile	263,555	139,280	112,535

Table B.2 99-, 95-, 50-, and 5-percentile for Workload E

(ops/sec)	Database		
	Couchbase	MongoDB	Cassandra
4 nodes			
99 percentile	14,495	15,555	2,040
95 percentile	14,035	15,200	1,790
50 percentile	12,005	14,505	1,415

5 percentile	10,155	13,785	1,095
10 nodes			
99 percentile	27,240	15,810	5,115
95 percentile	26,745	15,175	4,675
50 percentile	23,795	14,480	4,020
5 percentile	18,975	13,750	3,400
20 nodes			
99 percentile	39,620	15,445	8,335
95 percentile	37,180	14,765	7,900
50 percentile	31,985	13,830	7,210
5 percentile	26,040	12,735	6,565

Table B.3 99-, 95-, 50-, and 5-percentile for the Pagination Workload

(ops/sec)	**Database**		
	Couchbase	**MongoDB**	**Cassandra**
4 nodes			
99 percentile	37,960	11,590	2,245
95 percentile	37,420	11,310	2,055
50 percentile	36,150	10,780	1,520
5 percentile	34,055	10,230	1,140
10 nodes			
99 percentile	93,145	12,140	-
95 percentile	89,845	11,840	-
50 percentile	84,690	11,320	-
5 percentile	76,615	10,170	-
20 nodes			
99 percentile	161,145	12,390	-
95 percentile	141,980	12,020	-

50 percentile	97,755	11,330	-
5 percentile	54,369	9,900	-

Table B.4 99-, 95-, 50-, and 5-percentile for the Join Workload

(ops/sec)	Database		
	Couchbase	**MongoDB**	**Cassandra**
4 nodes			
99 percentile	1,125	225	69,570
95 percentile	1,200	205	62,735
50 percentile	1,155	175	56,790
5 percentile	1,080	150	48,365
10 nodes			
99 percentile	1,755	150	195,665
95 percentile	1,620	125	185,620
50 percentile	1,070	85	161,340
5 percentile	690	55	132,495
20 nodes			
99 percentile	1,360	180	285,705
95 percentile	1,239	85	278,405
50 percentile	1,000	50	253,345
5 percentile	714	25	238,280