



2023 NoSQL DBaaS Performance: Couchbase Capella vs. MongoDB Atlas

Using Yahoo! Cloud Serving Benchmark, this report compares the throughput and latency of the popular databases as a service (DBaaS) across four business scenarios and four different cluster configurations.

By Ivan Shryma, Data Engineer, Altoros

Q2 2023

[Table of contents](#)

1. Executive summary	3
2. A testing environment	3
2.1 YCSB instance configuration	3
2.2 Couchbase Capella cluster configuration	4
2.3 MongoDB Atlas cluster configuration	5
2.4 Operating costs	6
2.4.1 Couchbase Capella costs	6
2.4.2 MongoDB Atlas costs	7
3. Workloads and tools	7
3.1 Workloads	7
3.2 Tools	8
4. YCSB benchmark results	9
4.1 Workload A: the update-heavy mode	9
4.1.1 Workload definition and model details	9
4.1.2 Query	9
4.1.3 Evaluation results	10
4.1.4 Summary	11
4.2 Workload C: read-only	11
4.2.1 Workload definition and model details	11
4.2.2 Query	12
4.2.3 Evaluation results	12
4.2.4 Summary	13
4.3 Workload E: scanning short ranges	13
4.3.1 Workload definition and model details	13
4.3.2 Query	14
4.3.3 Evaluation results	14
4.3.4 Summary	16
4.4 Pagination Workload: filter with OFFSET and LIMIT	16
4.4.1 Workload definition and model details	16
4.4.2 Query	18
4.4.3 Evaluation results	18
4.4.4 Summary	19
5. Conclusion	19
6. Appendix	20
6.1 MongoDB Atlas shard collection and indexes	20
MongoDB Atlas shard collection	20
MongoDB Atlas index	20
6.2 Indexes for the scan query	20
Couchbase indexes	20
6.3 Indexes for Pagination Workload	20
Couchbase indexes	20
MongoDB Atlas indexes	20
6.4 Pricing	21
7. About the author	23

1. Executive summary

NoSQL encompasses a wide variety of database technologies that were developed in response to a rise in the volume of data and the frequency with which information is stored, accessed, and updated. In contrast, relational databases were not designed to cope with scalability and agility challenges that modern applications require. Furthermore, relational databases cannot take advantage of the affordable storage and processing power available in today's cloud environments. Meanwhile, new-generation NoSQL solutions help to achieve the highest levels of performance and uptime for modern application workloads. Finally, teams are more regularly seeking Database-as-a-Service (DBaaS) options to avoid having to invest increasing amounts of time and money in cluster support, deployment, and maintenance.

This report compares the performance results of four NoSQL databases as a service: **Couchbase Capella** and **MongoDB Atlas**. The goal of this report is to measure the relative performance in terms of latency and throughput that each database can achieve. The evaluation was conducted on four different cluster configurations—3, 6, 9, and 18 nodes—as well as under four different workloads.

The first workload performed *update-heavy* activity, involving 50% reads and 50% updates of the data. The second workload was *read-only*, with 100% read operations. The third workload performed a *short-range scan* that involved 95% scans and 5% updates, where short ranges of records were queried instead of individual ones. Finally, the fourth workload was a query with a single filtering option to which an offset and a limit were applied.

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

2. A testing environment

2.1 YCSB instance configuration

To provide verifiable results, the benchmark was performed on easily obtained Amazon Elastic Compute Cloud (EC2) instances. The YCSB client was deployed to 10 compute-optimized large instances. Each client instance of YCSB produced 40 threads. This means the total load on the database was 400 threads for each test.

Table 2.1 A description of the Amazon EC2 instance deployed to the YCSB client

Family	Compute-optimized
Type	c4.2xlarge
vCPUs	8
Memory (GiB)	15

(continued in the next page)

Table 2.1 A description of the Amazon EC2 instance deployed to the YCSB client (continued)

EBS-optimized available	Yes
Network performance	High
Platform	64-bit
Operating system	Ubuntu 18.04 LTS
AWS region	us-east-1

2.2 Couchbase Capella cluster configuration

Couchbase Capella is a fully managed Database as a Service. It combines the features of a key-value store allowing operations on single documents. The database also acts as a schemaless document store to access the documents by querying through SQL++ (SQL for JSON).

The Capella Control Panel includes a cluster sizing page, offering customers multiple options to choose from—such as instance sizes, configurations, and quantities. Couchbase Capella can also be tuned to deploy specific services to a single or several nodes in the cluster. The vendor calls this feature “Multi-Dimensional Scaling.”

Each node was configured to run the Data, Index, and Query services. The Data service is the most fundamental of all Couchbase services, providing access to data in memory and on disk. The Index service supports the creation of primary and global secondary indexes on items stored within Couchbase. The Query service supports the querying of data by means of SQL and relies on both the Index and Data services. Figure 2.2 shows the architecture of an example Capella cluster.

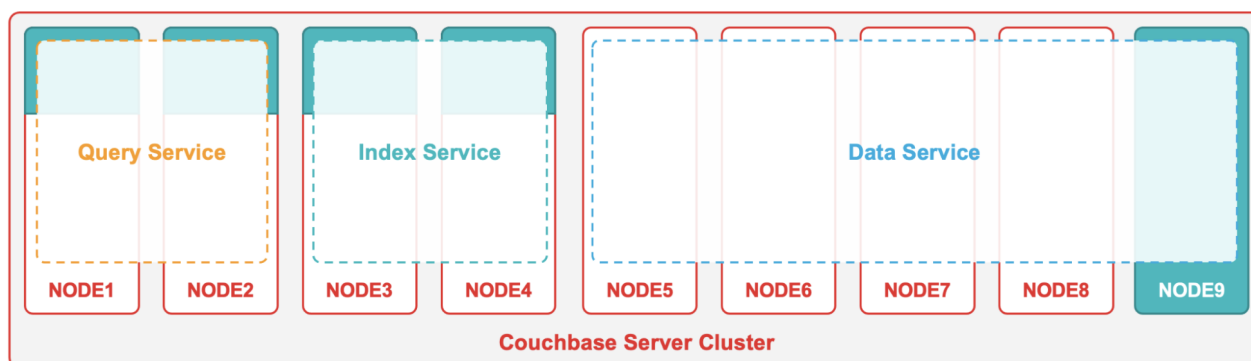


Figure 2.2 The architecture of a Couchbase Capella cluster ([image credit](#))

After the cluster is deployed, data access should be configured by creating database credentials and granting the required access permissions. The test’s bucket was created with half of the available system memory allocated for it. In the report, we have used a new storage engine called [Magma](#), which is designed to be highly performant for very large data sets that do not fit in memory.

The final step is to configure a list of allowed IPs on the control panel's Connect tab, since Couchbase Capella allows clusters to connect to trusted IP addresses only.

Table 2.2 Specification of a Couchbase Capella instance

vCPUs	8
Memory (GB)	64
EBS storage (GB)	200
IOPS	5,700
AWS region	us-east-1

2.3 MongoDB Atlas cluster configuration

MongoDB Atlas 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 that happen as completely independent processes.

In this evaluation, we utilized MongoDB Atlas v5.1. MongoDB employs a hierarchical cluster topology that combines router processes, configuration servers, and data shards. For each cluster size (3, 6, 9, and 18 nodes), the following production-grade configurations were 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, two secondaries).
- MongoDB's routers were deployed on each node for each shard.

Automatic installation and configuration for a MongoDB sharded cluster is a simple procedure. Users can choose their preferred cloud provider, region and type of nodes, count of shards, as well as the size of a replica set. The configurational server was a three-member replica set deployed automatically. Figure 2.3 shows the architecture of an example MongoDB cluster.

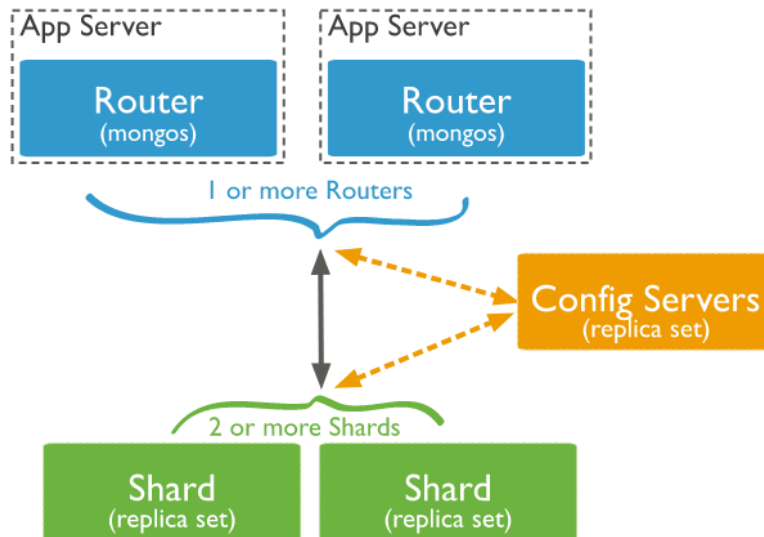


Figure 2.3 The architecture of a MongoDB cluster ([image credit](#))

MongoDB distributes shards at the collection level. MongoDB’s sharding feature partitions the collections’ data using a shard key. Hash-based partitioning was used for all the models. To support hash-based sharding, MongoDB provides a hashed index type that 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 more random distribution of collections in the cluster.

Table 2.3 A detailed description of a MongoDB Atlas instance

Type	M60
vCPUs	8
Memory (GB)	64
SSD storage (GB)	200
IOPS	5,700
AWS region	us-east-1

2.4 Operating costs

2.4.1 Couchbase Capella costs

The monthly billing report for running Couchbase Capella includes per instance–hour costs billed by the provider. Approximate monthly total for supporting a Capella cluster of specified configuration:

- 3 nodes amounted to around \$2,642
- 6 nodes amounted to around \$5,284
- 9 nodes amounted to around \$7,926
- 18 nodes amounted to around \$15,851

Note that charges in Couchbase Capella are billed in [Couchbase Capella Credits](#).

2.4.2 MongoDB Atlas costs

The pricing for MongoDB Atlas database is calculated based on the services that are used for cluster configuration. For this report, the following services were used:

- Atlas Instance—\$1.00 per server per hour
- Atlas Data Storage—\$0.000182 per GB per hour
- Atlas Data Transfer—\$0.01 per GB

Approximate monthly total for supporting a cluster of specified configuration:

- 3 nodes amounted to \$5,026
- 6 nodes amounted to around \$9,655
- 9 nodes amounted to around \$14,292
- 18 nodes amounted to around \$28,203

3. Workloads and tools

Database performance is defined by the speed at which a database processes basic operations. A basic operation is an action performed by a workload executor that 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, making it possible to directly control the load against the database. In addition, the threads measure latency, as well as the achieved throughput of their operations, and then 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 C.** Read only: 100% read, request distribution is Zipfian.
- 3) **Workload E.** Scan short ranges: 95% scan and 5% update, request distribution is Uniform.
- 4) **Pagination Workload.** Filter with `OFFSET` and `LIMIT`.

3.2 Tools

The YCSB client was used as a worker, consisting of the following components:

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

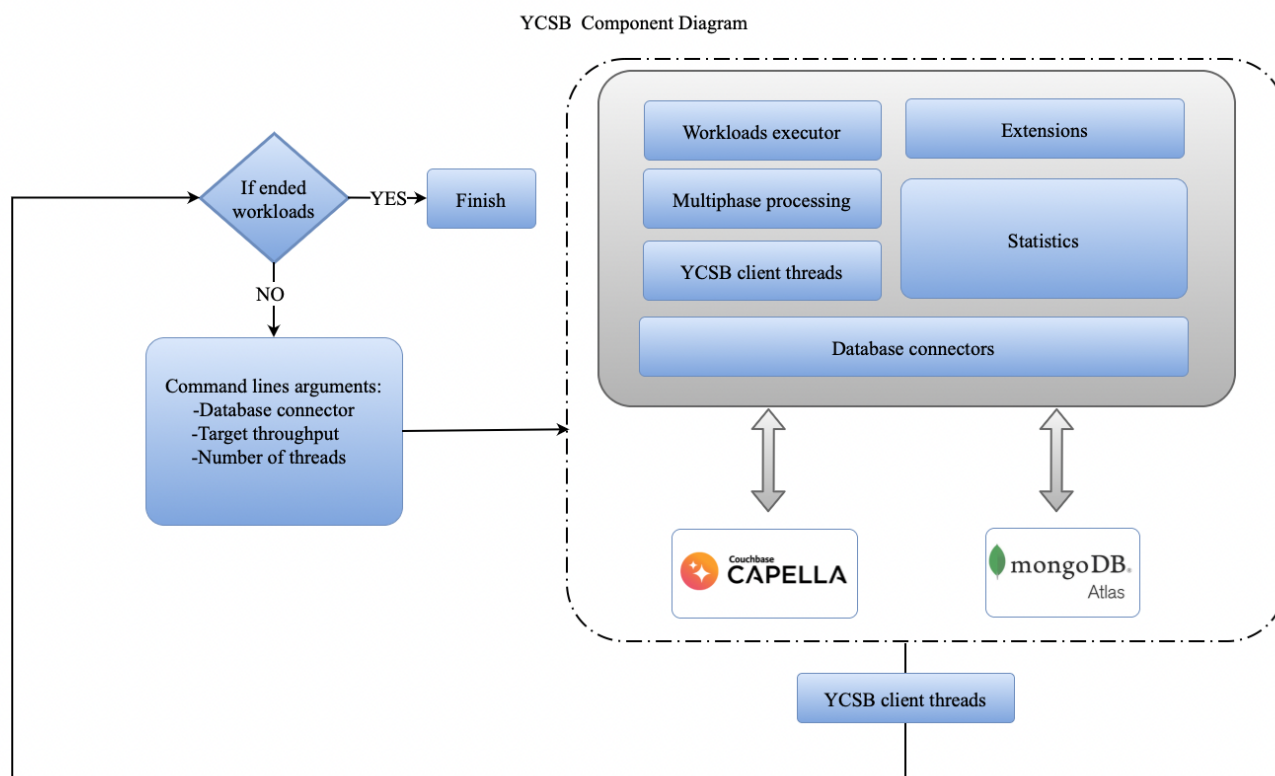


Figure 3.2.1 The components of the YCSB client

The workloads were tested under the following conditions:

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

Workloads A, C, and E are standard workloads provided by YCSB. Default data models were used for these workloads. Pagination Workload represents scenarios from real-life domains, such as finance (server-side pagination for listing filtered transactions). To emulate these scenarios on a domain level, a customer model was introduced for that workload:

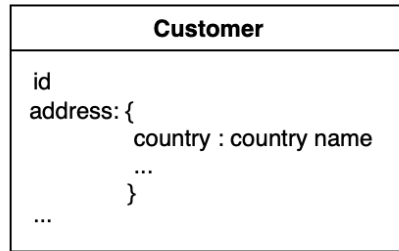


Figure 3.2.2 A graphic representation of the customer model

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 that simulates typical actions of an e-commerce solution. 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: 25 million records (each 1 KB in size, consisting of 10 fields and a key) on a 3-node cluster, 50 million records on a 6-node cluster, 100 million records on a 9-node cluster, and 200 million records on a 18-node cluster.

Couchbase Capella stores data in buckets and collections, which are the logical groups of items—key–value pairs. vBuckets are physical partitions of the bucket data. By default, Capella creates a number of master vBuckets per bucket to store bucket data and evenly distribute vBuckets across all cluster nodes.

Querying with document keys is the most efficient method, since 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.

For MongoDB Atlas, we have sharded the collection by `_id` (see [Appendix 6.1](#), “*MongoDB Atlas shard collection and indexes*”).

4.1.2 Query

The following queries were used to perform Workload A.

Table 4.1.2 Evaluated queries for Workload A

	Read	Update
Couchbase Key-Value API	<code>collection.get(id, \$2, getOptions().timeout(kvTimeout))</code>	<code>collection.upsert(id, content, upsertOptions().timeout(kvTimeout).expiry(documentExpiry).durability(persistTo, replicateTo))</code>
MongoDB Query	<code>db.ycsb.find({_id: \$1})</code>	<code>db.ycsb.update({_id: \$1}, { \$set: { fieldN: \$2 } })</code>

4.1.3 Evaluation results

On each type of a cluster, Couchbase Capella significantly outperformed MongoDB Atlas. On a 3-node cluster, it had a throughput of 232,050 ops/sec with a 2.67 ms latency. Couchbase Capella’s performance improved all the way to an 18-node cluster, where it had a throughput of 423,580 ops/sec with less than a 1 ms latency.

As the cluster size increased, MongoDB also demonstrated better performance. And the best results were on an 18-node cluster. MongoDB Atlas achieved 58,290 ops/sec with a 8.13 ms latency.

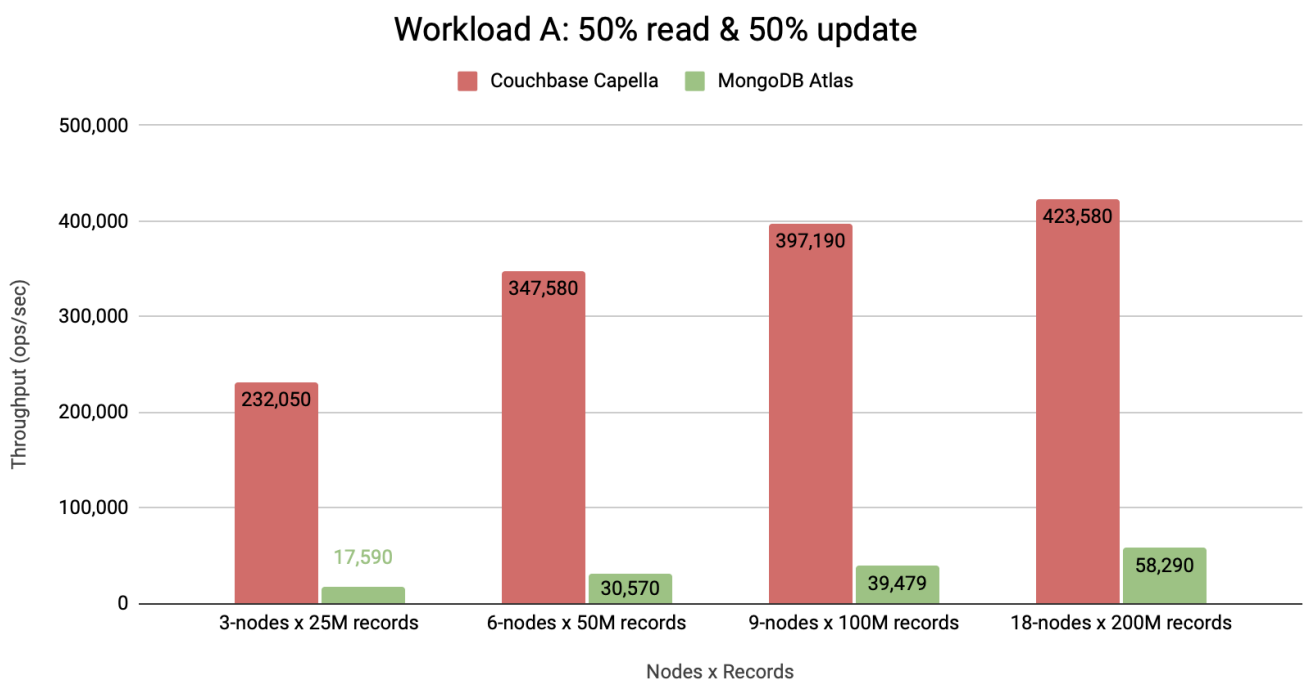


Figure 4.1.3.1 Throughput results under Workload A on 3-, 6-, 9-, and 18-node clusters

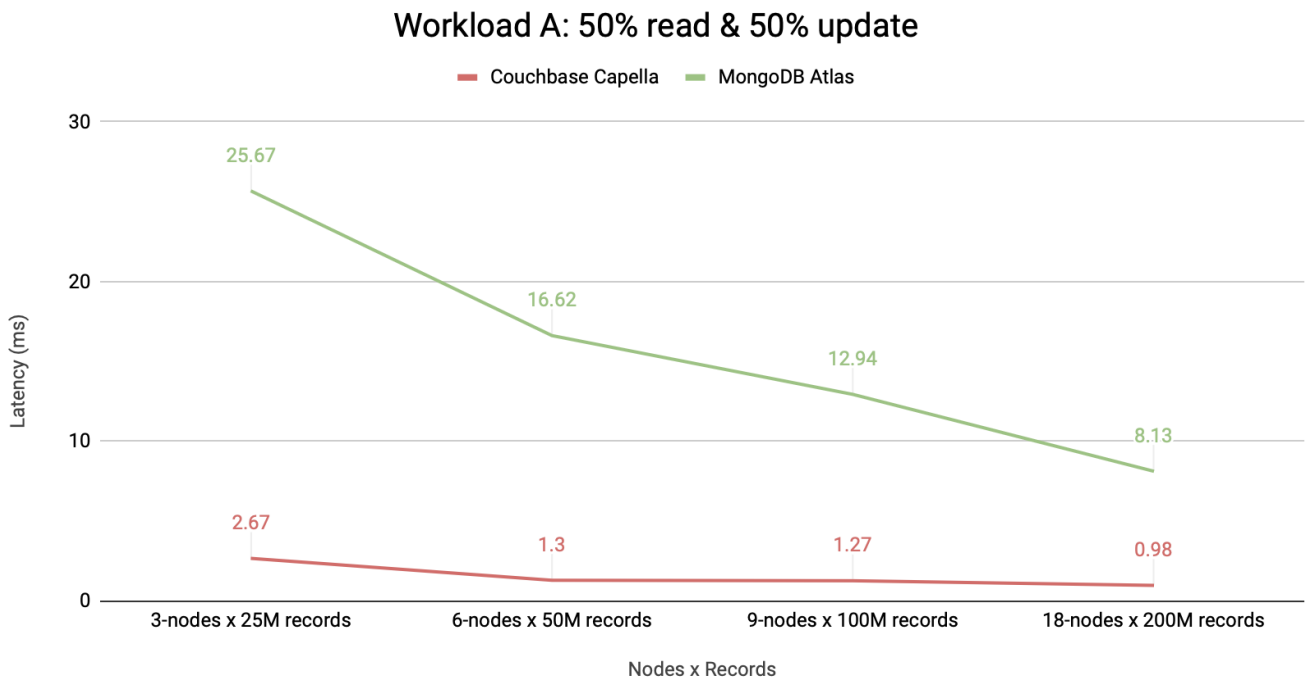


Figure 4.1.3.2 Latency results under Workload A on 3-, 6-, 9-, and 18-node clusters

4.1.4 Summary

Couchbase Capella demonstrated high throughput growth and clearly outperformed MongoDB Atlas on each type of a cluster.

Couchbase Capella stood out with a latency of about 1 ms on 6-, 9-, and 18-node clusters, and a latency of 2.67 ms on 3 nodes. The latency of MongoDB Atlas decreased continuously from 25.67 ms on 3 nodes to 8 ms on an 18-node cluster. MongoDB Atlas had worse results, nonetheless, it demonstrated the most growth with a cluster’s expansion from 3 to 18 nodes, with a 3x reduction in latency and a 3x increase in throughput.

4.2 Workload C: read-only

4.2.1 Workload definition and model details

Workload C is 100% read. The workload simulates user profile cache. The scenario was executed under the following settings:

- The read ratio was 100%.
- The Zipfian request distribution was used.
- The size of a data set was scaled in accordance with the cluster size: 25 million records (each 1 KB in size, consisting of 10 fields and a key) on a 3-node cluster, 50 million records on a 6-node cluster, 100 million records on a 9-node cluster, and 200 million records on a 18-node cluster.

Due to the workload’s simplicity, no additional configuration or indexes for the databases were necessary—the MongoDB Atlas cluster was just shared for workloads A and E.

4.2.2 Query

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

Table 4.2.2 Evaluated queries for Workload C

	Read
Couchbase Key-Value API	<code>collection.get(id, \$2, getOptions().timeout(kvTimeout))</code>
MongoDB Query	<code>db.ycsb.find({_id: \$1})</code>

4.2.3 Evaluation results

Couchbase Capella outperformed MongoDB Atlas with 578,590 ops/sec and a latency of 0.63 ms.

On a 3-node cluster, MongoDB Atlas had the lowest throughput of 42,810 ops/sec, which increased to 174,070 ops/sec on an 18-node cluster with a minimum latency of 2.4 ms. This was due to the amount of records per shard.

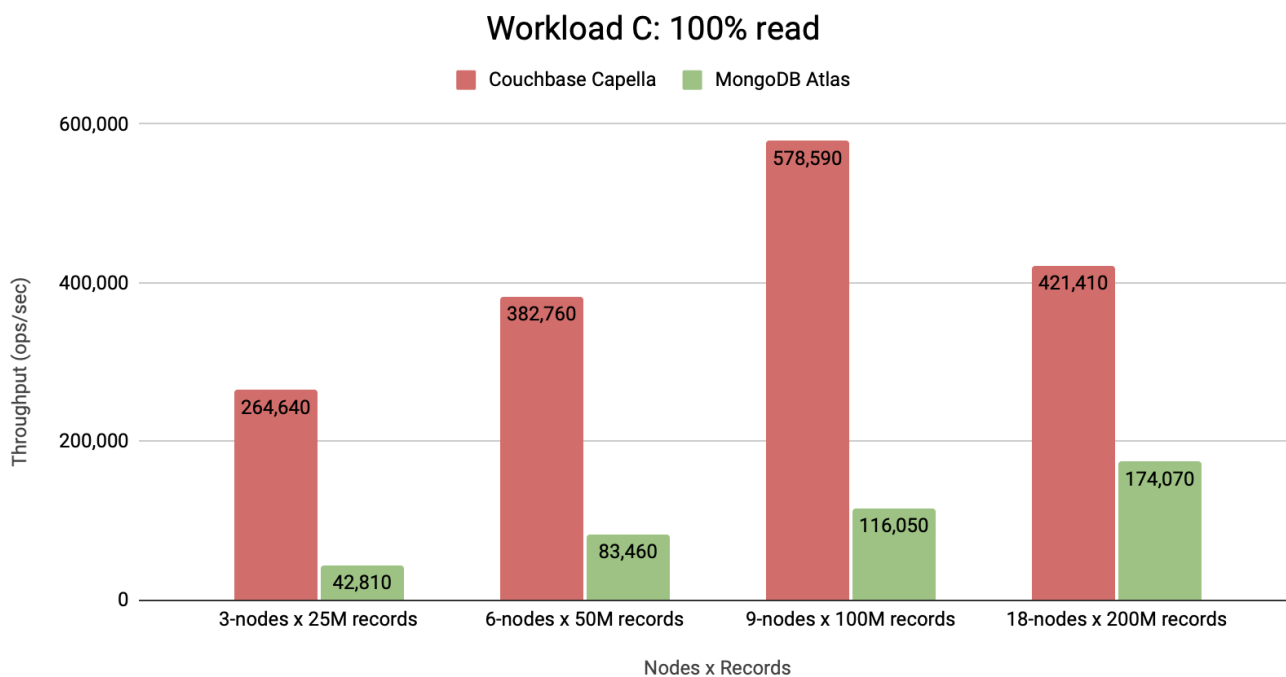


Figure 4.2.3.1 Throughput results under Workload C on 3-, 6-, 9-, and 18-node clusters

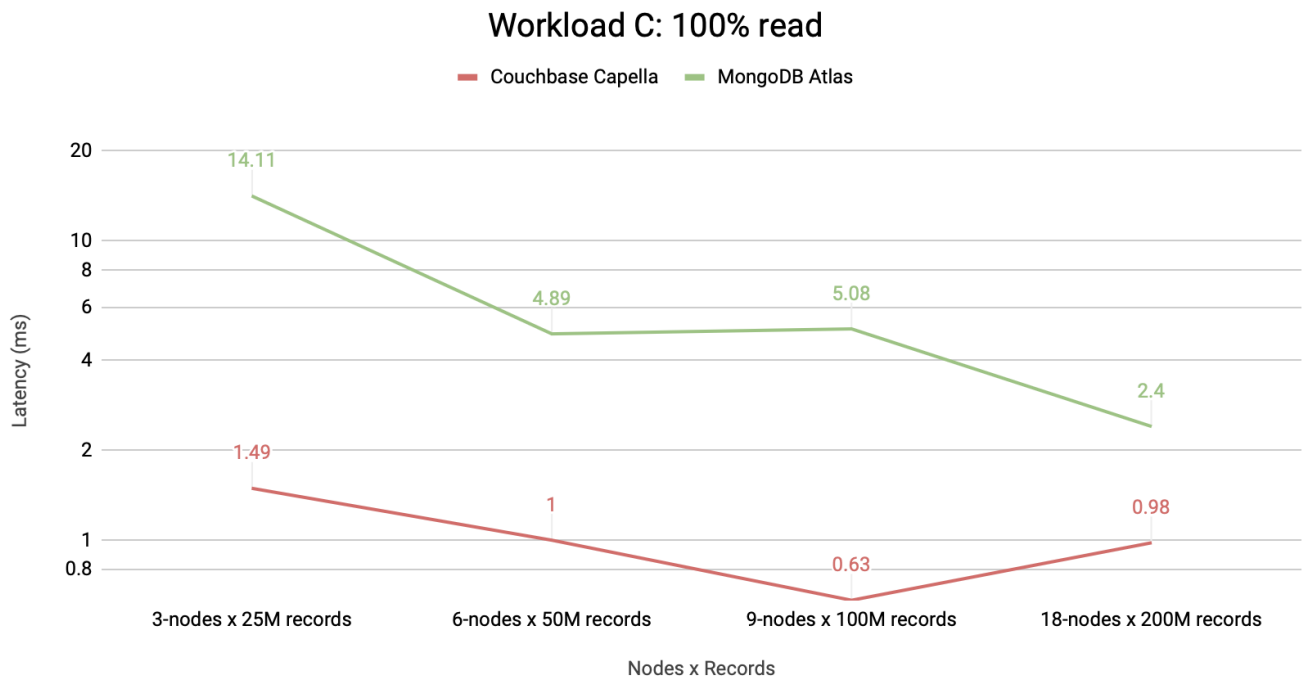


Figure 4.2.3.2 Latency results under Workload C on 3-, 6-, 9-, and 18-node clusters

4.2.4 Summary

Workload C, which consisted of simple read operations, produced interesting results for all the databases. Couchbase Capella showed a steady and significant growth in performance up to the 18-node cluster, with a small drop in throughput and latency—from 578,590 ops/sec with a latency of 0.63 ms to 487,387 ops/sec with a latency of 0.98 ms. Nonetheless, this performance was still very good.

On the other hand, MongoDB Atlas demonstrated stable results and showed growth on each type of cluster. Its worse performance was on a 3-node with only 42,810 ops/sec with a latency of 14.11 ms, while it showed the maximum capabilities—on an 18-node cluster, achieving 174,070 ops/sec with a latency of 2.4 ms.

4.3 Workload E: scanning short ranges

4.3.1 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 are clustered by ID). The scenario was executed under the following settings:

- The scan/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: 25 million records (each 1 KB in size, consisting of 10 fields and a key) on a 3-node cluster, 50 million

records on a 6-node cluster, 100 million records on a 9-node cluster, and 200 million records on a 18-node cluster.

- The maximum scan length reached 100 records.
- Uniform was used as a scan length distribution.

MongoDB Atlas distributes data using a shard key. There are two types of shard keys supported by this database: range- 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 such 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.

4.3.2 Query

The following queries were used to perform Workload E.

Table 4.3.2 Evaluated queries for Workload E

	Scan	Update
Couchbase SQL++ / Key-Value API	<pre>SELECT meta().id FROM `bucket` WHERE meta().id >= \$1 ORDER BY meta().id LIMIT \$2</pre>	<pre>collection.upsert(id, content, upsertOptions().timeout(kv Timeout).expiry(documentEx piry).durability(persistTo , replicateTo))</pre>
MongoDB Query	<pre>db.ycsb.find({ _id: {\$gte: \$1},{_id: 1} }) .sort({_id: 1}) .limit(\$2)</pre>	<pre>db.ycsb.update({_id: \$1}, {\$set: {fieldN: \$2}})</pre>

4.3.3 Evaluation results

On a 3-node cluster, MongoDB Atlas had the best throughput—compared to other types of clusters—with 27,940 ops/sec. However, MongoDB had the highest latency on a 3-node cluster with 53.8 ms, while Capella showed a lower latency of 16.45 ms.

On 18 nodes, Couchbase Capella showed the huge growth and there was the best with 3.84 ms, while having a throughput of 71,170 ops/sec.

Workload E: 95% scan & 5% update

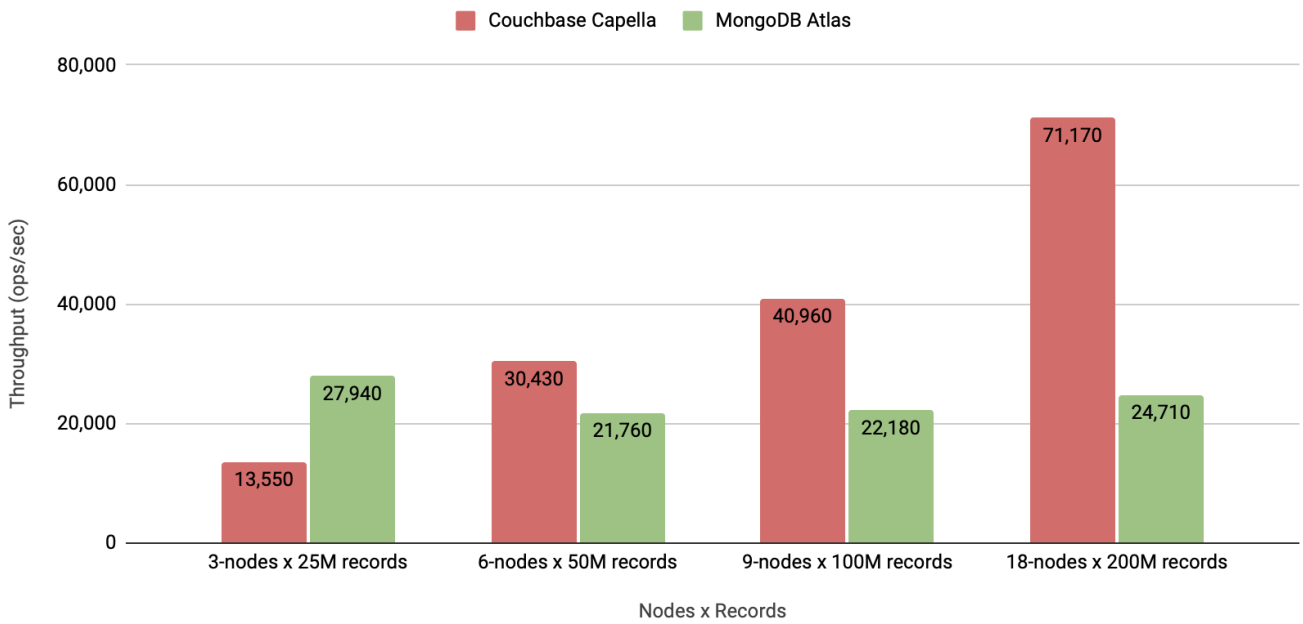


Figure 4.3.3.1 Throughput results under Workload E on 3-, 6-, 9-, and 18-node clusters

Workload E: 95% scan & 5% update

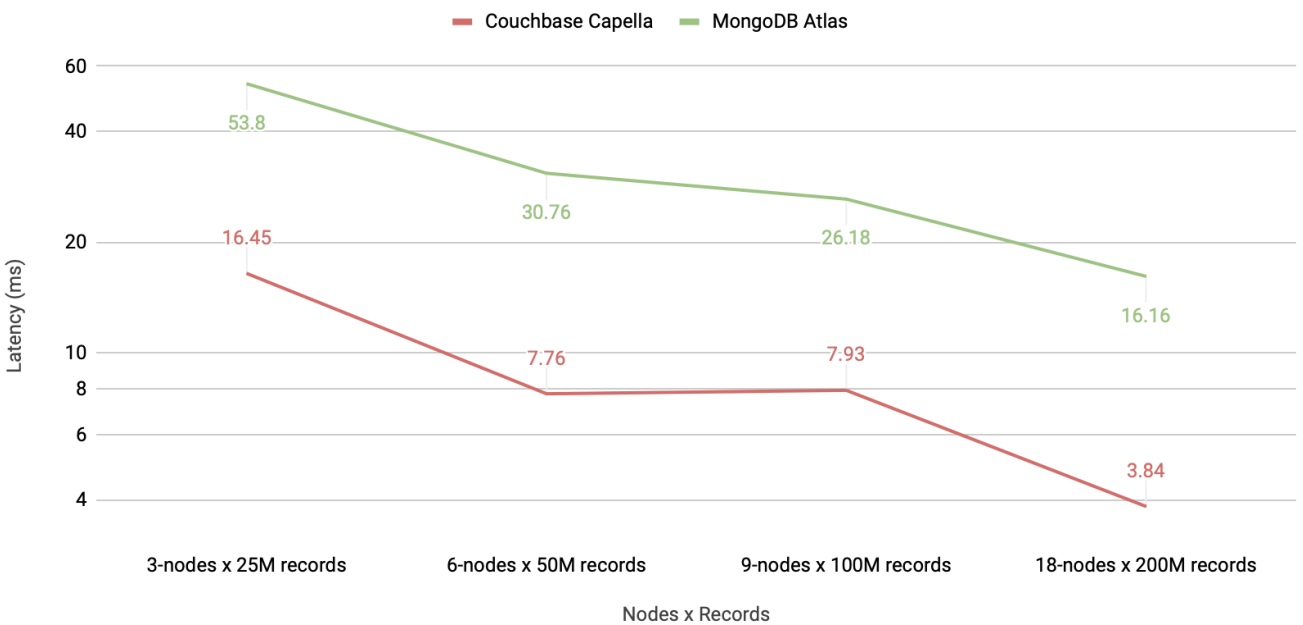


Figure 4.3.3.2 Latency results under Workload E on 3-, 6-, 9-, and 18-node clusters

4.3.4 Summary

Under *Workload E*, DynamoDB demonstrated the best throughput results with a gradual increase in operations per second, but produced a high number of failed requests. The latency of DynamoDB increased from 4.89 ms on a 3-node cluster to 6.72 ms on 18 nodes.

An increase in the amount of shards did not significantly impact MongoDB Atlas's performance in scan operations. The throughput remained relatively stable across different cluster types. However, the latency on 18 nodes was considerably lower at 16 ms compared to 53.8 ms on a 3-node cluster.

As the number of nodes grew, Couchbase Capella exhibited good results in both latency and throughput without throwing errors. The latency was the lowest and most predictable across all the databases, decreasing from 16 ms on 3 nodes to 3.84 ms on an 18-node cluster. Additionally, the size of the cluster significantly impacted Couchbase Capella's throughput, which increased from 13,550 to 71,170 ops/sec, demonstrating more than a 5x improvement.

4.4 Pagination Workload: filter with OFFSET and LIMIT

4.4.1 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 was 100%.
- The size of a data set was scaled in accordance with the cluster size: 1 million customers (each 4 KB in size) on a 3-node cluster, 5 million customers on a 6-node cluster, 25 million customers on a 9-node cluster, and 100 million customers on an 18-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.

The primary index of Couchbase allows for querying any field of a document. However, this type of querying is rather slow, since it retrieves all the documents of all types in the bucket, whether or not a query eventually returns them to the user. 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- and disk-optimized. The latter is the default mode.

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 one over linked lists with the $O(\log n)$ complexity. The lock-free skip list is used to provide nonblocking reads/writes and maximize utilization of the CPU cores. On top of

a lock-free skip list, there is a multiversion manager responsible for regular snapshotting in the background.

Memory-optimized indexes reside in memory and 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 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 `query1` ON `bucket`(`address`.`country`) USING GSI;
```

MongoDB Atlas 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 from

all cursors on the shards using a round robin method. 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 the records to skip using the `skip() cursor` method, the mongos cannot pass the skip to the shards. Instead, the mongos instance 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 instance 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 as shown below.

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

For sharding with two or more shards, we need to shard by `address.country`. However, since the field has low cardinality, we have opted to shard by an additional field, `id`. This has greatly improved performance and ensured proper distribution of data between shards.

```
sh.shardCollection( "ycsb.customer", { "address.country": 1, _id:  
1 }, false )
```

4.4.2 Query

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

Table 4.4.2 Evaluated queries for *Pagination Workload*

Couchbase SQL++	<pre>SELECT meta().id FROM `bucket` WHERE address.country='\$1' OFFSET \$2 LIMIT \$3</pre>
MongoDB Query	<pre>db.customer.find({address.country: \$1}, {_id: 1}) .skip(\$2) .limit(\$3)</pre>

4.4.3 Evaluation results

Couchbase Capella outperformed MongoDB Atlas on almost all types of clusters, achieving the highest throughput of 163,640 ops/sec on an 18-node cluster with a latency of 2.52 ms. Only on 3 nodes, Capella was the second by throughput—at the same time, it clearly won the latency comparison with 1.49 ms on this configuration.

MongoDB demonstrated stable performance, with the best result of 56,700 ops/sec and an 8.9 ms latency on a 9-node cluster. On 3 nodes, MongoDB had the highest throughput with about 46,160 ops/sec.

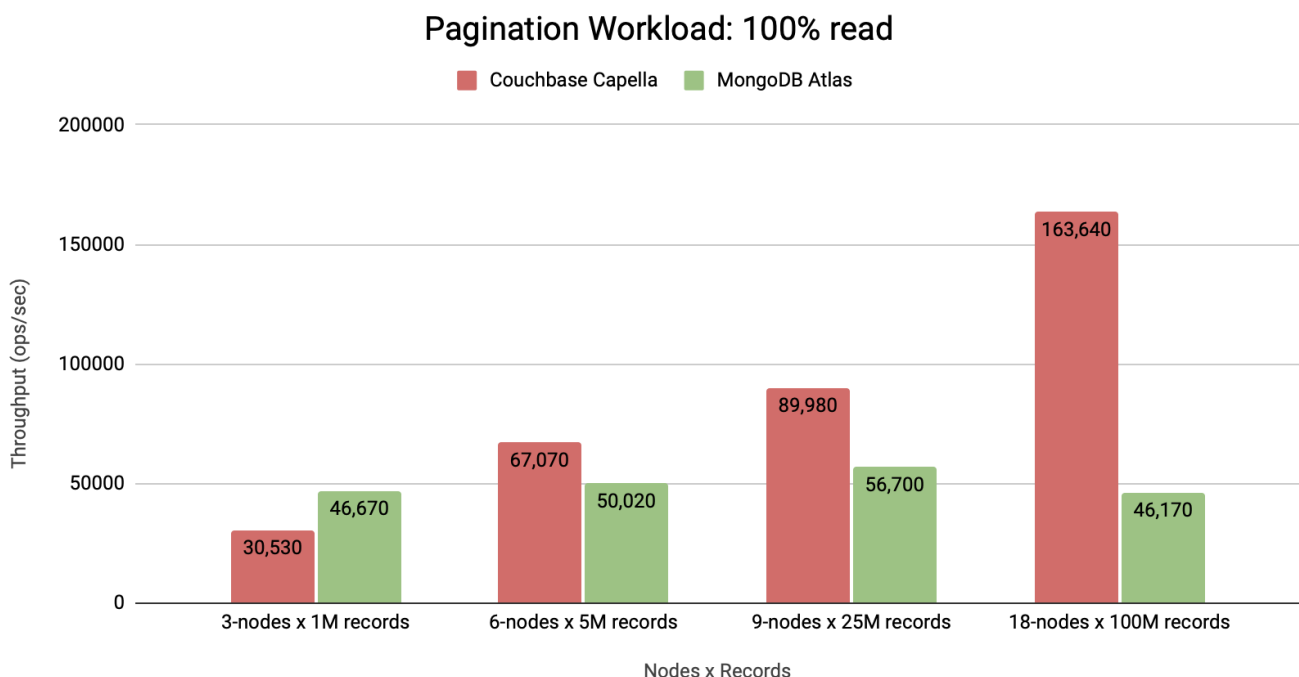


Figure 4.4.3.1 Throughput results under *Pagination Workload* on 3-, 6-, 9-, and 18-node clusters

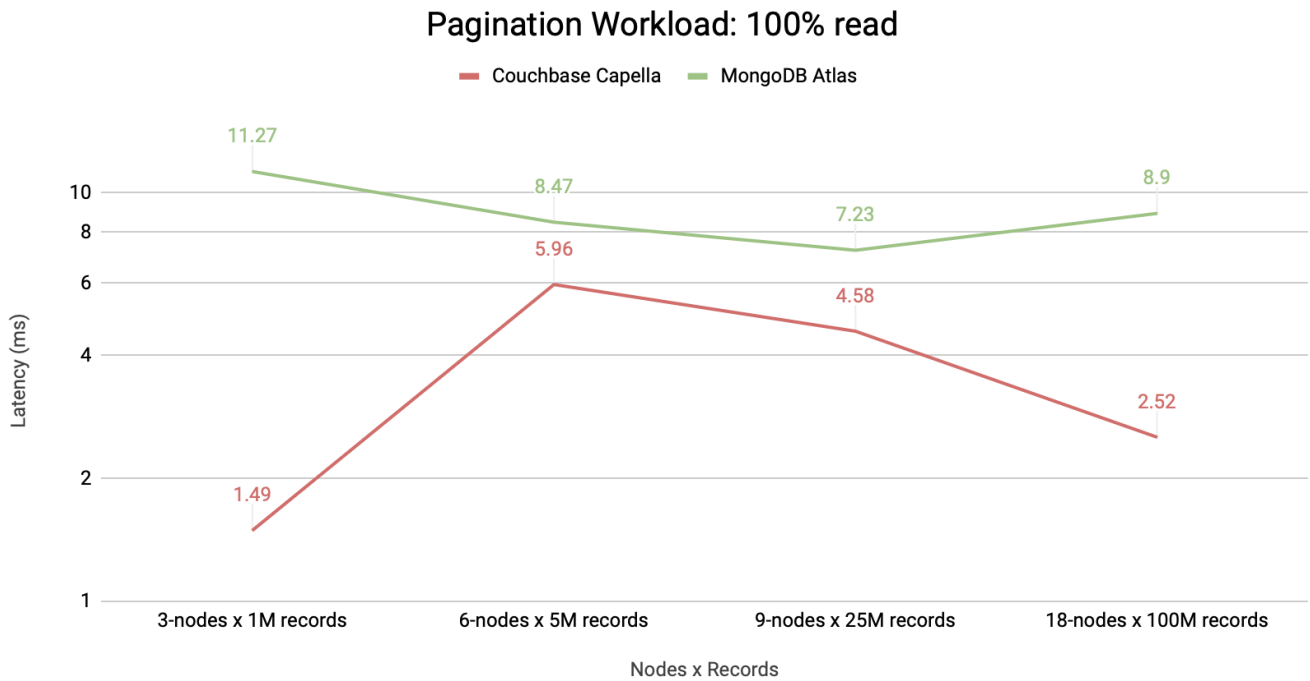


Figure 4.4.3.2 Latency results under Pagination Workload on 3-, 6-, 9-, and 18-node clusters

4.4.4 Summary

In *Pagination Workload*, Couchbase Capella outperformed MongoDB, with throughput increasing as the number of nodes grew. However, latency increased from 1.49 ms on 3 nodes to 6 ms on 6 nodes before dropping to 2.52 ms on 18 nodes. At the same time, the workload’s throughput increased by over 5 times from 3 to 18 node clusters.

MongoDB Atlas demonstrated a performance increase from 3 nodes to 9 nodes, achieving 56,700 ops/sec with a latency of 7.23 ms. However, the performance dropped on 18 nodes to the level of a 3-node cluster, with 46,170 ops/sec and a latency of 8.9 ms. Nevertheless, these results were pretty stable and did not change with different amounts of nodes and records.

5. Conclusion

Typically, no single database as a service is perfect for meeting all the requirements of a given scenario. Each solution has its advantages and disadvantages, which may become more or less important depending on the specific criteria. Despite this, DBaaS can help engineers to reduce the time needed for deployment, configuration, and support.

Though DBaaS solutions do not offer broad system tools for configurations, the databases have been optimally tuned for each workload. Therefore, configurations can be changed based on workloads.

Couchbase Capella performed better than the other databases in each workload. The query engine of Couchbase Capella supports aggregation, filtering, and other operations on large data sets. As clusters and data sets grow in size, Couchbase Capella ensures a high level of

scalability across these operations. Capella was good overall and showed that it is capable of performing any type of query with good performance.

MongoDB Atlas produced relatively decent and predictable results and was scalable enough to handle increasing amounts of data and cluster expansion. In addition, MongoDB showcased improved performance with a growing number of shards in almost any workload.

6. Appendix

6.1 MongoDB Atlas shard collection and indexes

MongoDB Atlas shard collection

```
sh.shardCollection( "ycsb.usertable", { _id: 1 }, false )
```

MongoDB Atlas index

```
db.usertable.ensureIndex({_id: "hashed"})
```

6.2 Indexes for the scan query

Couchbase indexes

```
CREATE PRIMARY INDEX ON `bucket` WITH {"num_replica":  
NUMBER_OF_INDEX_NODES - 1}
```

6.3 Indexes for Pagination Workload

Couchbase indexes

```
CREATE PRIMARY INDEX ON `bucket` WITH {"num_replica":  
NUMBER_OF_INDEX_NODES - 1};  
  
CREATE INDEX `query1` ON `bucket`(`address`.`country`) WITH  
{"num_replica": NUMBER_OF_INDEX_NODES - 1};
```

MongoDB Atlas indexes

```
sh.shardCollection( "ycsb.customer", { "address.country": 1, _id:  
1 }, false )
```

6.4 Pricing

The following figures display the extrapolated cost per billion operations for each workload. However, in the case of MongoDB, to compare it fairly with other databases, we needed to make additional adjustments to the cluster configuration, such as increasing IOPS from 3,000 to 5,700, which would result in slightly higher storage costs. In the charts below, the lower numbers indicate better results.

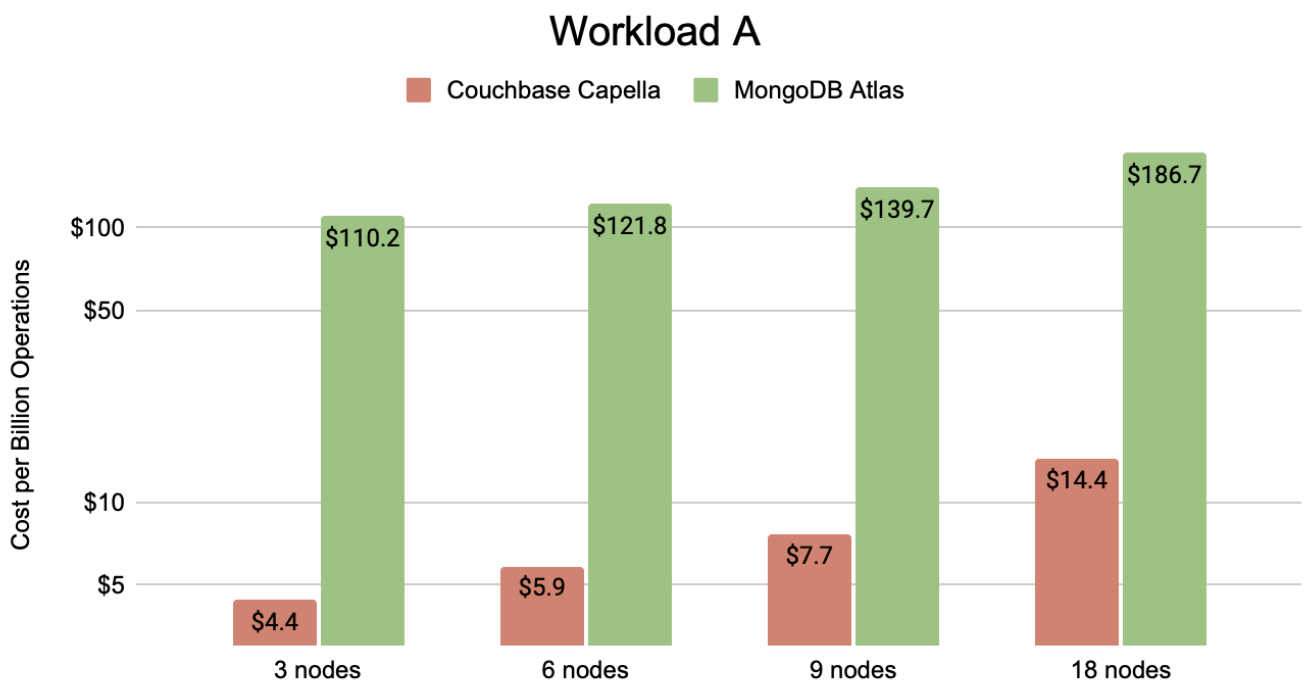


Figure 6.4.1 Cost per billion operations for Workload A

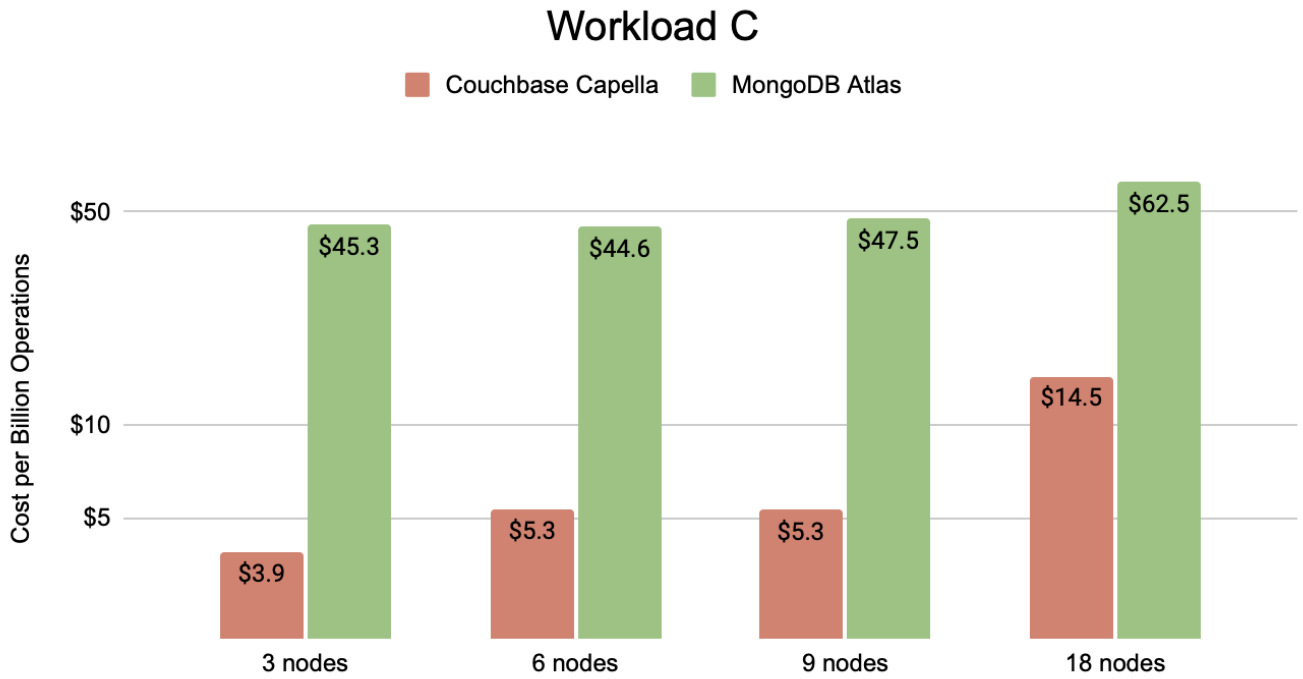


Figure 6.4.2 Cost per billion operations for Workload C

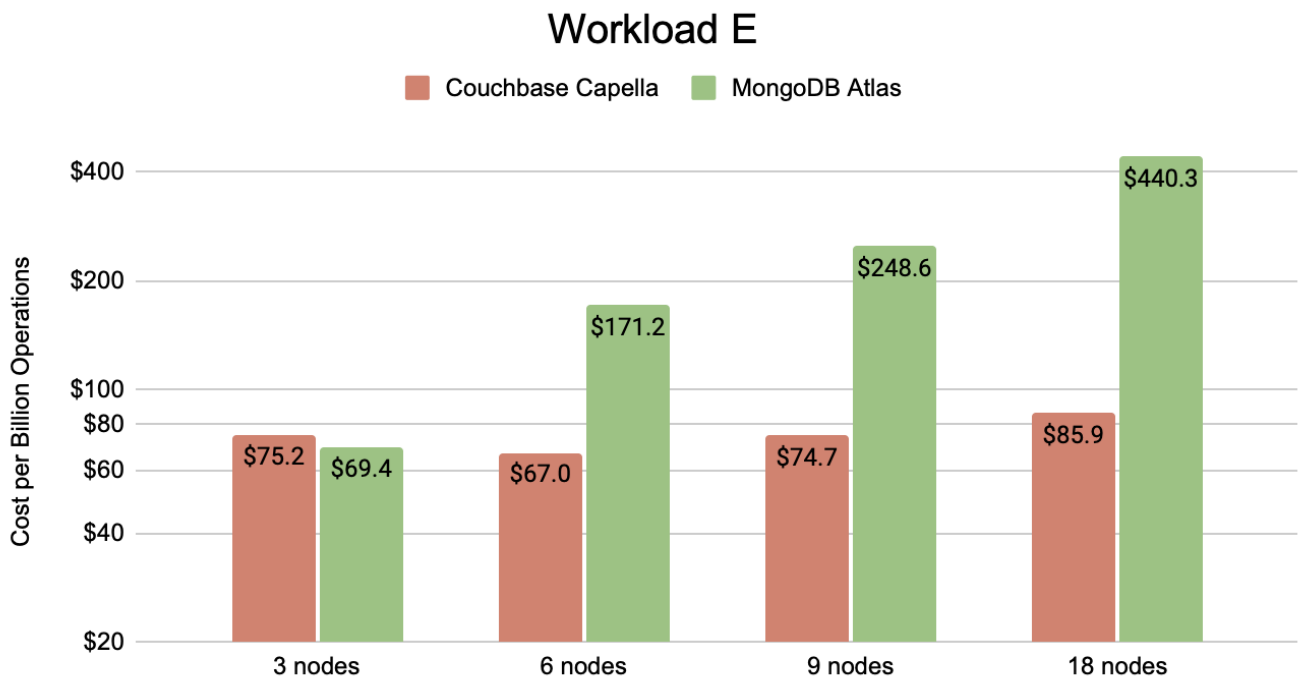


Figure 6.4.3 Cost per billion operations for Workload E

Workload Pagination

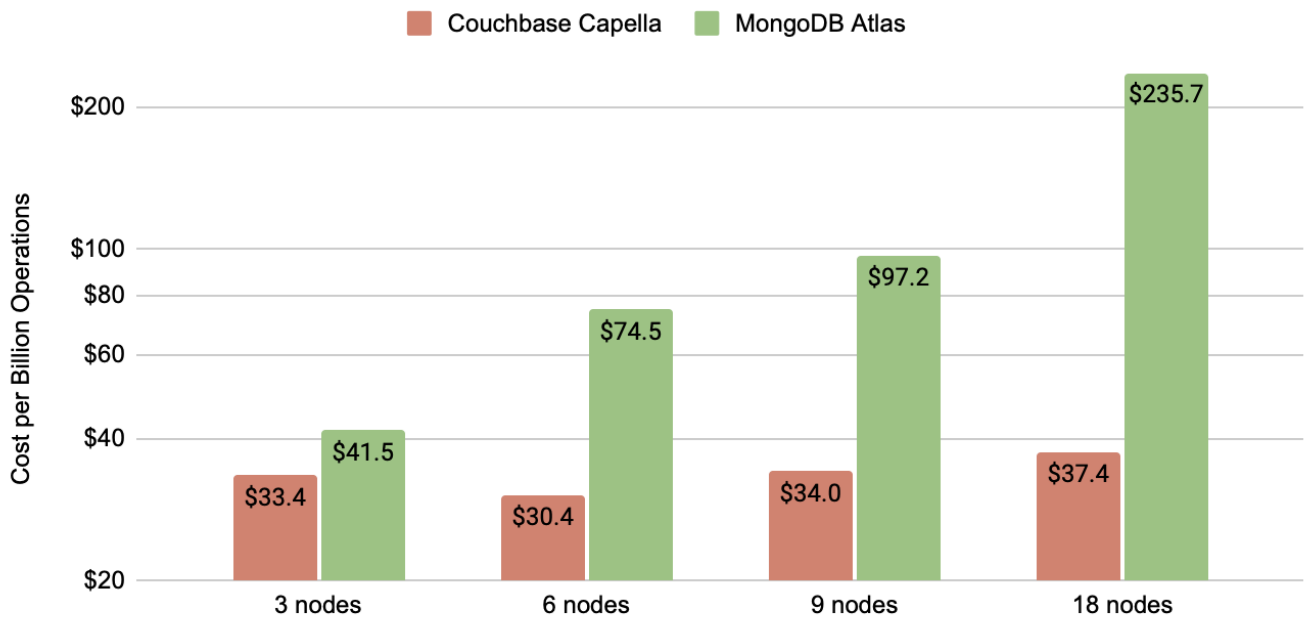
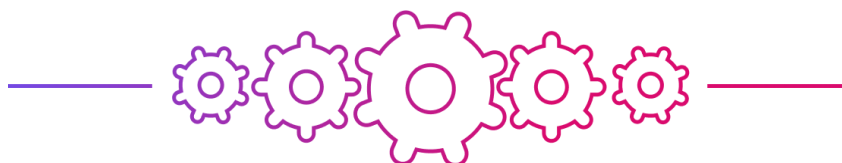
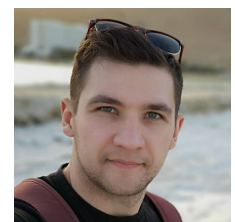


Figure 6.4.4 Cost per billion operations for Pagination Workload

7. About the author

Ivan Shyrma is Data Engineer at Altoros with 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.



Altoros is an experienced IT services provider that helps enterprises to increase operational efficiency and accelerate the delivery of innovative products by shortening time to market. Relying on the power of cloud automation, microservices, AI/ML, and industry knowledge, our customers are able to get a sustainable competitive advantage. For more, please visit www.althoros.com.

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.

