Performance Evaluation of NoSQL Databases as a Service with YCSB:

Couchbase Cloud, MongoDB Atlas, and AWS DynamoDB

This 24-page report evaluates and compares the throughput and latency of Couchbase Cloud, MongoDB Atlas, and Amazon DynamoDB across four varying workloads in three different cluster configurations.

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

Q4 2020

Table of Contents

1. Executive Summary 3

2. Testing Environment 3
- 2.1 YCSB instance configuration 3
- 2.2 MongoDB Atlas cluster configuration 4
- 2.3 Couchbase Cloud cluster configuration 5
- 2.4 Amazon DynamoDB cluster configuration 6
- 2.5 Prices 6
  - 2.5.1 Couchbase costs 7
  - 2.5.2 MongoDB Atlas costs 7
  - 2.5.3 Amazon DynamoDB costs 8

3. Workloads and Tools 8
- 3.1 Workloads 8
- 3.2 Tools 8

4. YCSB Benchmark Results 10
- 4.1 Workload A: The update-heavy mode 10
  - 4.1.1 Workload definition and model details 10
  - 4.1.2 Query 10
  - 4.1.3 Evaluation results 11
  - 4.1.4 Summary 12
- 4.2 Workload E: Scanning short ranges 12
  - 4.2.1 Workload definition and model details 12
  - 4.2.3 Evaluation results 14
  - 4.2.4 Summary 15
- 4.3 Pagination Workload: Filter with OFFSET and LIMIT 15
  - 4.3.1 Workload definition and model details 15
  - 4.3.2 Query 17
  - 4.3.3 Evaluation results 17
  - 4.3.4 Summary 18
- 4.4 JOIN Workload: JOIN operations with grouping and aggregation 18
  - 4.4.1 Workload definition and model details 18
  - 4.4.2 Query 19
  - 4.4.3 Evaluation results 20
  - 4.4.4 Summary 20

5. Conclusion 21

6. About the Authors 22

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 this data is stored, accessed, and changed. In contrast, relational databases were not designed to cope with scalability and agility challenges that modern applications face. 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.

NoSQL databases have complex structures with multiple components. In this regard, it often becomes challenging for engineering teams to oversee and manage NoSQL cluster deployments. In order to avoid investing increasing amounts of time and money on cluster support, deployment, and maintenance, teams will seek Database as a Service (DBaaS) alternatives.

This report compares the performance results of three NoSQL databases as a service: Couchbase Cloud, MongoDB Atlas, and Amazon DynamoDB. 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 three different cluster configurations—6, 9, and 18 nodes—as well as under four different workloads.

The first workload performs update-heavy activity, invoking 50% reads and 50% updates of the data. The second workload performs a short-range scan that involves 95% scans and 5% updates, where short ranges of records are queried instead of 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 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. 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 four compute-optimized large instances. Each client instance of YCSB produces threads from 25 to 175 in 25-thread increments. This means the total load on the database ranged from 100 to 700 threads with increments of 100 (4 clients with 25 threads each) during each test.

Table 2.1 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 16.04 LTS
AWS region	us-east-1

2.2 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 v4.2. MongoDB employs a hierarchical cluster topology that combines router processes, configuration servers, and data shards. For each cluster size (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 a size of a replica set. The configurational server was a three-member replica set deployed automatically.

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 2.2 A detailed description of the MongoDB Atlas instance

Type	M60
vCPUs	8
Memory (GB)	64
SSD Storage (GB)	1,900
IOPS	5,700
AWS region	us-east-1

2.3 Couchbase Cloud cluster configuration

Couchbase Cloud is a fully managed database as a service. It combines a rich set of features of a key-value store to perform operations involving single documents and acts as a schemaless document store to access the documents through N1QL queries. (In a previous research paper, we performed a comparative analysis of MySQL, N1QL, and MongoDB query.) The service works by creating a virtual private cloud within a cloud provider account, so that the clusters can be securely deployed, managed, and monitored through a single user interface.

Connecting a cloud requires selecting a region and setting the customer's virtual private cloud (VPC) environment. Customers can supply an optional IP range in a classless inter-domain routing (CIDR) notation and deploy clusters using the stack template. For Amazon Web Services (AWS), the resource template was in the form of a `CloudFormation` template. Once the connected cloud is created, a cluster within a project can also be created.

The Couchbase Control Panel includes the **Cluster Sizing** page, offering customers multiple options to choose from, such as instance sizes, configurations, and quantities. Couchbase Cloud can also be tuned to deploy specific services to a single or more nodes in the cluster. The vendor calls this feature "multidimensional scaling."

For each cluster size of 6, 9, and 18 nodes, the r5.2xlarge instances were used, because they align closely with the instances running MongoDB.

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 Server. The Query service supports the querying of data by means of SQL- and N1QL-like query language and depends on both the Index and Data services.

After the cluster is deployed, data access should be configured by creating a user and granting the required access permissions. The test's bucket was created with half of the available system memory allocated for it.

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

Table 2.3 Specification of a Couchbase Cloud instance

Type	R5 Double Extra Large/r5.2xlarge
vCPUs	8
Memory (GiB)	64
EBS Storage (GiB)	200
Network bandwidth	Up to 10 Gigabit
AWS region	us-east-1

2.4 Amazon DynamoDB cluster configuration

Amazon DynamoDB is a fully managed NoSQL database service that provides fast and predictable performance with seamless scalability. All of the data is stored on solid-state drives (SSDs).

Amazon DynamoDB is provided as a service. With this product, there is no need to worry about hardware provisioning, setup and configuration, replication, software patching, or cluster scaling. The performance power of a cluster fully depends on the pricing model.

With Amazon DynamoDB, users can configure read/write capacities for their tables. Users can choose between two capacity modes for processing reads and writes:

- on-demand
- provisioned (default, free-tier eligible)

We chose the provisioned mode in order to specify the biggest number of reads and writes per second as individual settings. The read/write capacity is calculated against the cost. As long as MongoDB has the highest pricing for, we estimated the capacity against it. The provisioned capacity can automatically scale in response to traffic changes.

For evaluation purposes, auto scaling was disabled to maintain parity with other databases and to limit costs. Unfortunately, under the provisioned mode, Amazon DynamoDB throws exceptions when read/write operations exceed the predetermined capacities. This resulted in failed operations in certain workloads. It likely would not have happened if we had simply increased our investment in provisioned capacity to raise its ceiling. This means that Amazon DynamoDB is strict with its self-imposed ceiling.

2.5 Prices

In this chapter, we overview the expenses incurred during the tests. This can help you to estimate the cost of operations for benchmarking activities.

2.5.1 Couchbase costs

The monthly billing report for running Couchbase Cloud includes two types of expenditures:

- per instance-hour costs billed by Couchbase
- per infrastructural services costs billed by AWS

Per instance-hour costs depend exclusively on the sizing of a cluster or on the number of instances running in a cluster:

- subscription for Couchbase Cloud Developer Pro on reserved instances ($0.7748)
- Amazon Elastic Compute Cloud running Linux/UNIX ($0.504)

Monthly totals for per instance-hour costs are the following:

- 6 nodes—$5,524.42
- 9 nodes—$8,286.62
- 18 nodes—$16,573.25

Per infrastructural services costs are calculated based on specific workloads running in an actual cluster. The breakdown of services and costs are:

- Data Transfer
  - $0.010 per GB in/out/between EC2 availability zones or using elastic IPs or ELB
- Amazon Elastic Compute Cloud NAT Gateway
  - $0.045 per GB of data processed by NAT Gateways
  - $0.045 per NAT Gateway Hour
- Elastic Load Balancing
  - $0.008 per GB of data processed by Load Balancer
  - $0.025 per Load Balancer–hour (or partial hour)
- Simple Storage Service
  - $0.023 per GB across the first 50 TB of storage used on monthly basis

2.5.2 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—$0.96 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:

- 6 nodes amounted to around $6,105.6

- 9 nodes amounted to around $8,971.2
- 18 nodes amounted to around $17,553.6

2.5.3 Amazon DynamoDB costs

The pricing for Amazon DynamoDB under the provisioned mode is based on read/write capacities. For this report, there was no additional index or autoscaling. The read/write capacity was selected based on the monthly total for MongoDB Atlas:

- 6 nodes amounted to around $6,103.04
- 9 nodes amounted to around $8,961.11
- 18 nodes amounted to around $17,549.85

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

3.2 Tools

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

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

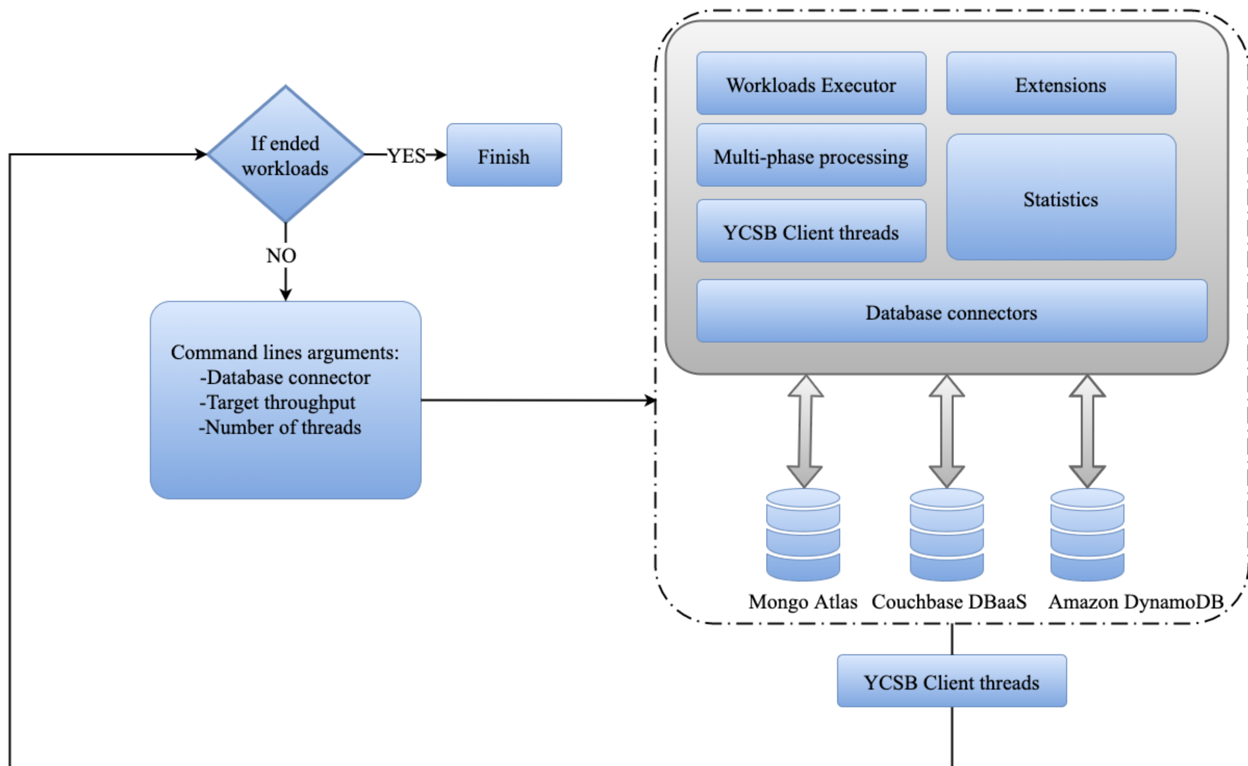


Figure 3.1 The components of the YCSB client

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 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.

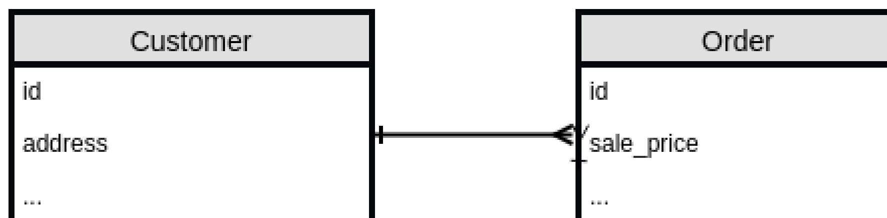


Figure 3.2 A graphic representation of the customer-order model

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 that simulates typical actions of an e-commerce solution user—50% of reading 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 6-node cluster, 100 million records on a 9-node cluster, and 200 million records on a 18-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 method as 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.

Amazon DynamoDB's read/write capacity for the workload was calculated through experiments. The chosen values have the best balance of read and write capacities based on cost. For each cluster, the following values were used:

- 6 nodes: 4,100 read and 11,800 write capacities
- 9 nodes: 5,000 read and 17,550 write capacities
- 18 nodes: 9,000 read and 34,490 write capacities

4.1.2 Query

The following queries were used to perform Workload A.

Table 4.1.2 Evaluated queries

Query name	Couchbase N1QL	MongoDB Query	Amazon DynamoDB

READ	collection.get(id, getOptions().timeout(kvTimeout))	db.ycsb.find({_id: $1})	{
			"TableName": "usertable",
			"Key": {
			"firstname": {
			"_id": "$1"
			}
			},
			"ConsistentRead": "false"
			}

UPDATE	collection.upsert(id, content, upsertOptions().timeout(kvTimeout).expiry(documentExpiry).durability(persistTo, replicateTo))	db.ycsb.update(
		{ _id: $1 },
		{
		$set: {
		fieldN: $2
		}
		})
			{
			"TableName":"usertable",
			"Key": {_id={S: $1},
			"AttributeUpdates": {
			$2={
			Value: {
			S: $3
			},
			"Action": "PUT"
			}
			}
			}

4.1.3 Evaluation results

On 6-node clusters, Couchbase Cloud and Amazon DynamoDB displayed quite similar results. Couchbase Cloud and Amazon DynamoDB had a throughput of 33,460 ops/sec and 30,400 ops/sec, respectively. Meanwhile, MongoDB Atlas had a throughput of 19,144 ops/sec, much lower than Couchbase Cloud and Amazon DynamoDB. Couchbase Cloud significantly outperformed MongoDB Atlas and Amazon DynamoDB on 9-node and 18-node clusters.

The database was able to process up to 119,000 ops/sec on a 9-node cluster and 168,908 ops/sec on an 18-node cluster, while Amazon DynamoDB managed 46,344 ops/sec on a 9-node cluster and 54,344 ops/sec on an 18-node cluster. MongoDB Atlas hit the throughput limit on a 9-node cluster with 27,544 ops/sec. The 18-node cluster throughput of MongoDB Atlas grew constantly and did nott appear to hit a throughput limit.

Amazon DynamoDB had unstable results, because it produced a great number of failed operations. On each type of cluster, Amazon DynamoDB had 40% of failed update operations and almost 1% of failed read operations.

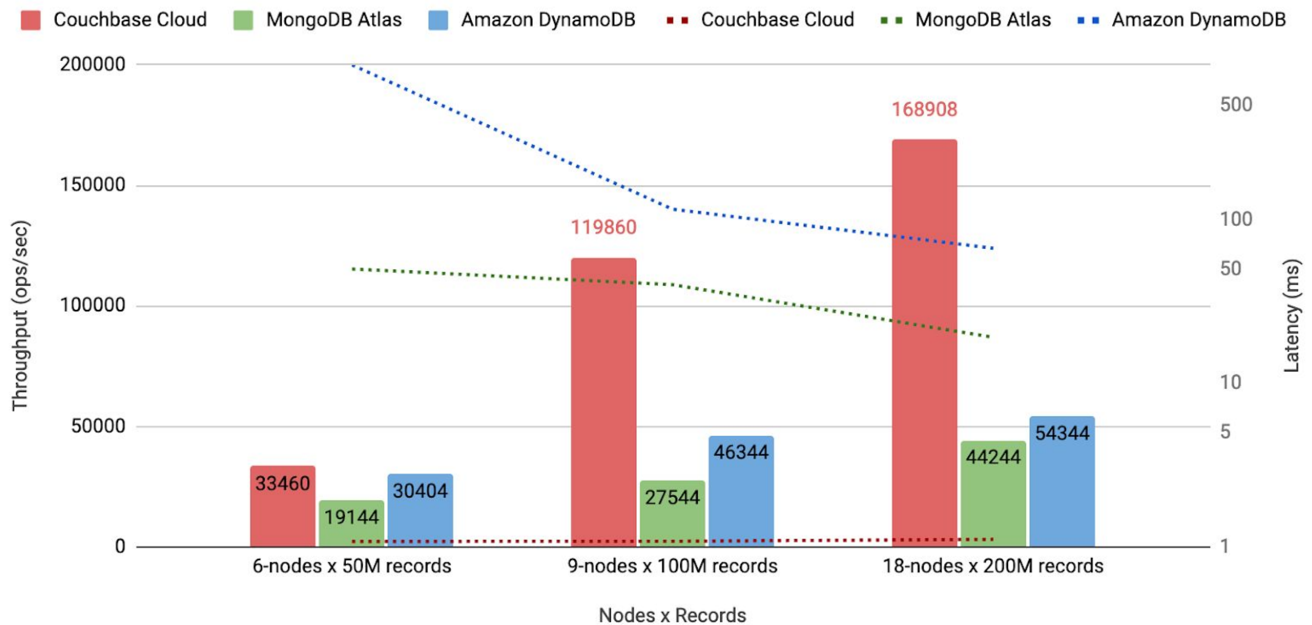


Figure 4.1.3 Performance results under Workload A on 6-, 9-, and 18-node clusters

4.1.4 Summary

The throughput of each database grew constantly depending on the type of a cluster. All databases achieved the throughput limit for each cluster type, except MongoDB Atlas on a 18-node cluster. Couchbase Cloud showed high throughput growth and clearly outperformed MongoDB Atlas and Amazon DynamoDB on 9-node and 18-node clusters.

While all three databases had low latency, Couchbase Cloud stood out in this aspect with a latency of a millisecond. The latency of MongoDB Atlas was around 50 ms and Amazon DynamoDB was 100 ms. MongoDB Atlas and Couchbase Cloud showed stable results without failed operations in comparison to Amazon DynamoDB. While MongoDB Atlas demonstrated weak results, they were also the most predictable and expected ones.

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 are clustered by ID). The scenario has been 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: 50 million records (each 1 KB in size, consisting of 10 fields and a key) on a 6-node cluster, 100 million records on a 9-node cluster, and 250 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 routes 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.

In Amazon DynamoDB, the scan operation is required to use read capacity. There are no special tricks to speed up scan operation. You can try using parallel scan, but read capacity will not change anyway. As long as read capacity is cheap, we were able to increase the default maximum count to 40,000 read operations. The capacities were chosen after a few experiments to get the best result. For each cluster, the following values were used:

- 6 nodes had 40,000 read and 4,620 write capacities
- 9 nodes had 80,000 read and 8,971 write capacities
- 18 nodes had 176,900 read and 900 write capacities

The workload for Amazon DynamoDB performed differently, because the start key is unique (i.e., Amazon DynamoDB cannot hit the `Item` that `ExclusiveStartKey` represents by your query condition), so we need to `getItem(startKey)` and then use scan for the response to retrieve the value, but it influenced the throughput. There are two requests in SCAN row in Amazon DynamoDB.

Table 4.2.1 Evaluated queries

Query name	Couchbase N1QL	MongoDB Atlas	Amazon DynamoDB

SCAN	SELECT meta().id
	FROM `bucket`
	WHERE meta().id >=
	$1
	ORDER BY meta().id
	LIMIT $2
		db.ycsb.find({
		id: {
		$gte: $1
		}, {
		id: 1
		}).sort({
		id: 1
		}).limit($2)
			{
			"TableName":
			"usertable",
			"Key": {
			"firstname": {
			"_id":
			"$1"
			}
			},
			"ConsistentRead":
			"false"
			}
			
			{TableName:
			usertable,
			AttributesToGet:
			[id]}

UPDATE	collection.replace(
	id, content,
	replaceOptions().ti
	meout(kvTimeout).ex
	piry(documentExpiry
	).durability(persis
	tTo, replicateTo))
		db.ycsb.update(
		{ _id: $1 },
		{
		$set: {
		fieldN: $2
		}
		})
			{
			"TableName":"userta
			ble",
			"Key":
			{_id={S: $1},
			"AttributeUpdates":
			{
			$2={
			Value:
			{
			S: $3
			},
			"Action": "PUT"
			}
			}
			}

4.2.3 Evaluation results

On 6-node clusters, Amazon DynamoDB had the lowest result with 8,378 ops/sec. Couchbase Cloud and MongoDB Atlas had a throughput of 12,296 ops/sec and 6,636 ops/sec, correspondingly. MongoDB Atlas and Amazon DynamoDB managed the same amount of operations on 9-node clusters as on 6-node clusters with 16,346 ops/sec and 8,464 ops/sec, respectively. Couchbase Cloud database was able to process 23,993 ops/sec—the best result on 9-node clusters. Furthermore, Couchbase Cloud continued to increase throughput up to 32,045 ops/sec on 18-node clusters, unlike Amazon DynamoDB that had virtually similar results on all clusters. MongoDB Atlas processed 31,408 ops/sec on an 18-node cluster.

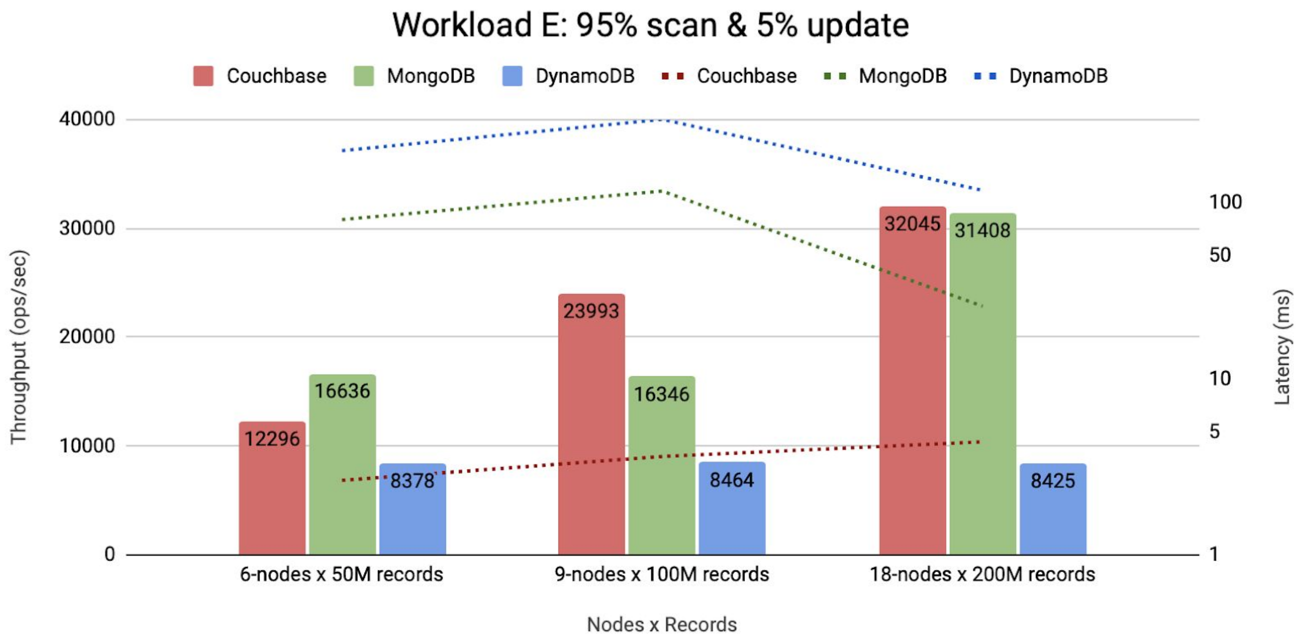


Figure 4.2.3 Performance results under Workload E on 6-, 9-, and 18-node clusters

4.2.4 Summary

Under Workload E, Couchbase Cloud demonstrated the best results with a gradual increase in operations per second, as well as the lowest and most stable latency. Increasing the amount of shards positively affected MongoDB Atlas performance as scan operations became more efficient with additional shards. On the other hand, the results seem to be close to the throughput limit, since increasing count of records per shard leads to lower amount of operations and higher latency.

In case of Amazon DynamoDB, the client had to make two requests to the databases instead of a single one to perform under Workload E. Due to the increased load, the client hit its throughput limit. We found it out when checking Amazon DynamoDB metrics via a database console. Furthermore, Amazon DynamoDB got 10% of failed operations on a 6-node cluster, but on 9-node and 18-node clusters failed operations amounted to less than 1%.

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 was 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 6-node cluster, 25 million customers on a 9-node cluster, and 50 million customers on a 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 a 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 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 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 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 `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 } );
```

Amazon DynamoDB could not compete in the Pagination Workload. Amazon DynamoDB requires every attribute in the index key schema to be a top-level attribute of a string, number, or binary type. Nested attributes and multivalued sets are not allowed in indexes. This means an index on the `address.country` nested field cannot be created and employed. Instead, a full table scan will be performed. Complexity of full table scan depends linearly on the table size requiring tens of seconds on millionish data sets to complete. Practically, a scan request cannot be fulfilled in a configured time interval and results in a time-out error.

Given that the request execution pipeline in Amazon DynamoDB consists of **scan**, **limit (for pagination)**, **and filter** stages, this additionally puts a restriction on the limit clause. The limit parameter in `DynamoDBQueryExpression` is used for the pagination purpose only, so it can limit the number of items per page and not the number of pages requested, and could not help to speed up the workload.

4.3.2 Query

The following queries were used to perform Pagination Workload.

Table 4.3.1 Evaluated queries

Couchbase N1QL	MongoDB Query	Amazon DynamoDB

`SELECT meta().id`
`FROM `bucket``
`WHERE address.country='$1'`
`OFFSET $2`
`LIMIT $3`
	`db.customer.find({`
	`address.country: $1`
	`}, {`
	`id: 1`
	`})`
	`.skip($2)`
	`.limit($3)`
		`{TableName: customer,`
		`Limit: $1,`
		`ProjectionExpression:`
		`#keyid,FilterExpression:`
		`#f1.#f2 = :countrykey,`
		`ExpressionAttributeNames:`
		`{#keyid=_id,`
		`#f1=address,`
		`#f2=country},`
		
		`ExpressionAttributeValues:`
		`{:countrykey={S: $1}}`
		`}`

4.3.3 Evaluation results

On a 6-node cluster, MongoDB Atlas had the lowest throughput of 16,340.34 ops/sec, while Couchbase Cloud had 30,728 ops/sec. MongoDB Atlas and Couchbase Cloud showed the lowest performance on 9-node clusters rather than on 6-node clusters. This happened due to the amount of records per shard. Couchbase Cloud database performed the best on an 18-node cluster with 36,612.42 ops/sec, and MongoDB Atlas managed 20,152 ops/sec on the same cluster.

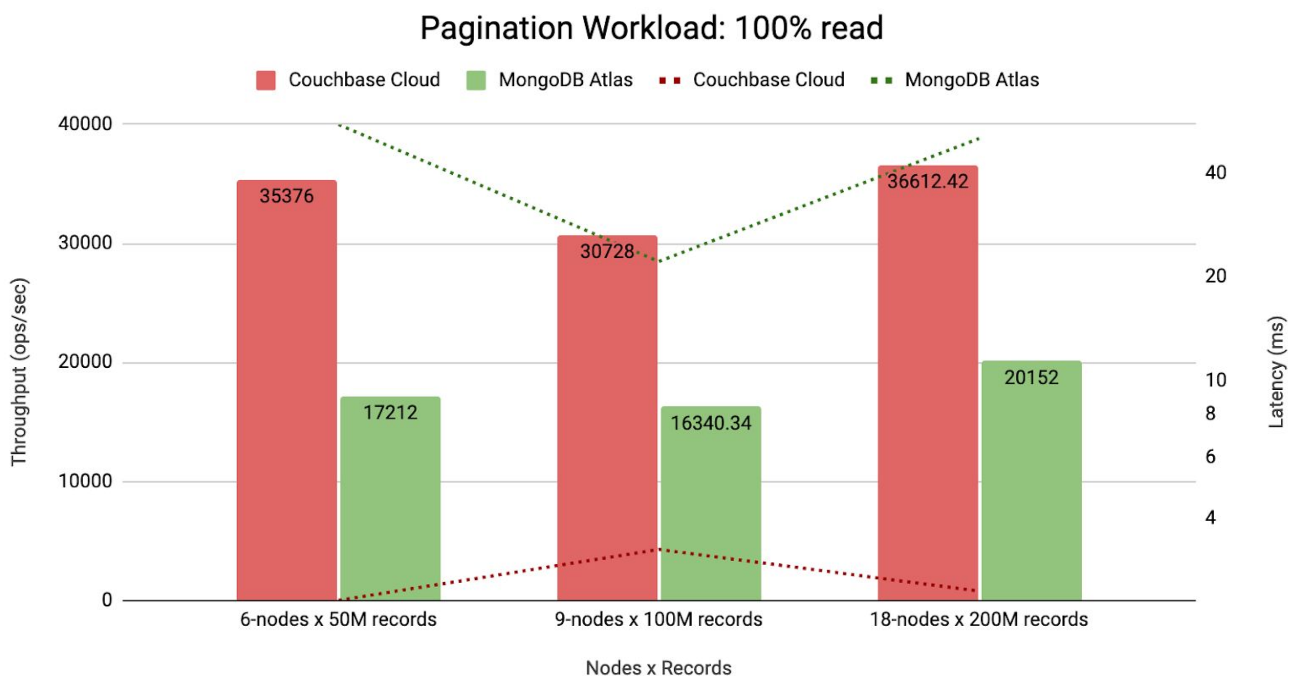


Figure 4.3.2 Performance results under Pagination Workload on 6-, 9-, and 18-node clusters

4.3.4 Summary

Couchbase Cloud demonstrated good results for Pagination Workload. It performed worse on a 9-node cluster compared to the 6-node one, since the size of a data set and indexes doubled, while the cluster size increased by only 50%. For 18-node clusters, Couchbase Cloud and MongoDB performed the best. However, Couchbase Cloud outperformed MongoDB for all cluster sizes.

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. 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: 25 million customers and 25 million orders (each 4.5 KB in size) on a 6-node cluster, 50 million customers and 50 million orders on a 9-node cluster, and 50 million customers and 50 million orders on a 18-node cluster.
- The maximum of a query length reached 100 records.
- Uniform distribution was used for a query length and query offset selection.
- The maximum of a query offset reached 5 records.

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

- `Index JOIN` is used when one of the two tables represents a document key(s) employing the `ON KEYS` statement.
- `ANSI JOIN` is applicable to arbitrary expressions on any field in a document, standart `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 for one of the tables in the `JOIN` (usually, the smaller one) used by the other table to find matches. Performance can be optimized under suitable conditions.

Only the first two types—`Index JOIN` and `ANSI JOIN`—were evaluated under this benchmark. In addition, 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 consists of only index resolutions without time-consuming document retrieval over the network, which results in a significant query performance boost.

The following secondary index was created for Couchbase.

```
CREATE INDEX `query2` ON `bucket`(address.zip, month, order_list,
sale_price) USING GSI;
```

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 JOIN Workload, an alternative solution was employed. One way to work with `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 modelling approach can be applied to all the databases under evaluation, but it leads to dramatically varying results. Due to this, we considered a similar business case with two different models available: *customers* and *orders*. In this case, a `JOIN` operation was a simple two-phase read with filtering, which had a significant impact on the overall `JOIN` operation performance.

Amazon DynamoDB could not perform the query properly for the same reason as in Pagination Workload, because of the scan by internal attribute. Furthermore, there is no `GROUP BY` operator in Amazon DynamoDB, as well as there is no option to `JOIN` tables. So, to evaluate Amazon DynamoDB under Pagination Workload, we need to change the data model. This would mean putting the three databases in different conditions, affecting the fairness of the results. For this reason, we did not run JOIN Workload.

4.4.2 Query

The following queries were used to perform JOIN Workload.

Table 4.4.1 Evaluated queries

Couchbase N1QL	MongoDB Query	DynamoDB

SELECT o2.month,
c2.address.zip,
SUM(o2.sale_price) as
sale_price
FROM `bucket` c2
INNER JOIN `bucket` 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: {
		index se

	`id: null,`
	`sum: {`
	`$sum: "$sale_price"`
	`}`
	`}}])`	

4.4.3 Evaluation results

Amazon DynamoDB is not included in the evaluation for JOIN Workload as indexes on nested attributes, and multivalued sets are not supported. Couchbase is ahead in throughput with 750 ops/sec, while MongoDB has 100 ops/sec on 6-node clusters.

The performance of both databases slightly decreased on 9-node clusters, since the data set doubled in size, while the cluster size increased by 50% only. In other words, less workers were available to fetch documents for a nested JOIN loop on larger data sets.

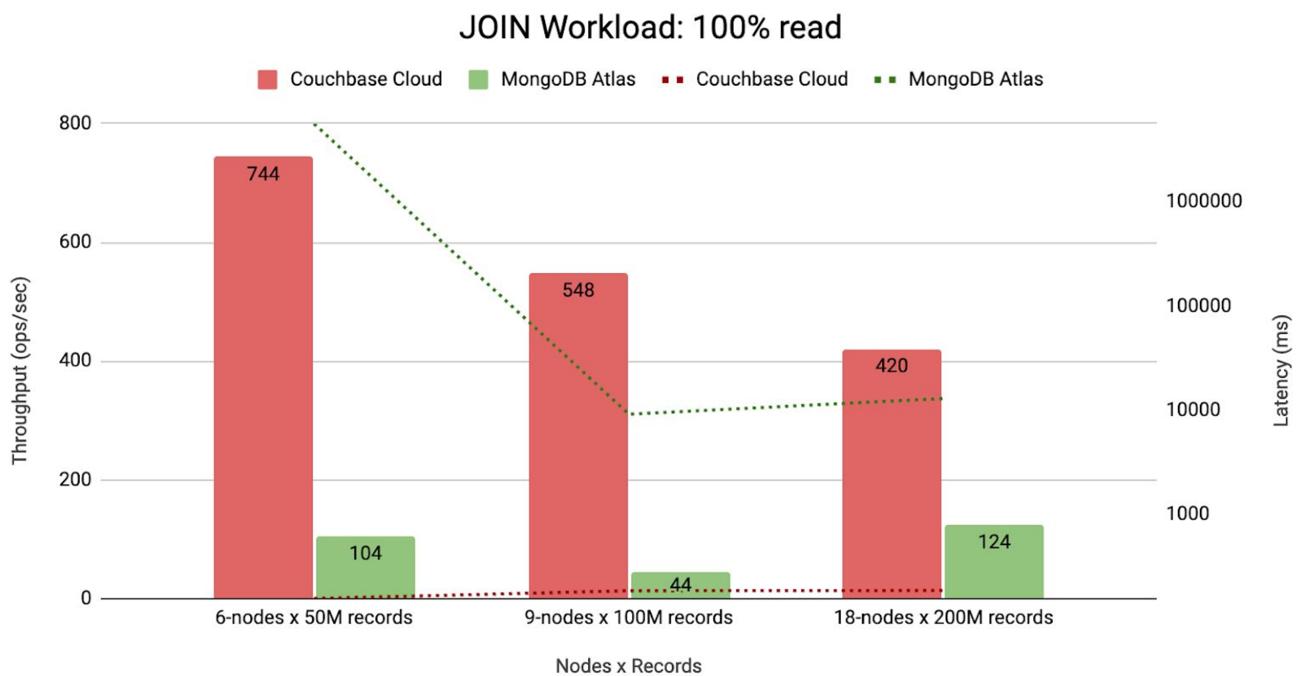


Figure 4.4.3 Performance results under JOIN Workload on 6-, 9-, and 18-node clusters

4.4.4 Summary

Pagination Workload can be processed by searching index in `O(log n)` time, while JOIN Workload is much more expensive consisting of multiple steps: index search, looking up values of the other index for `JOIN` matching, and then sort-based aggregation using `ORDER BY`. This results in at least `O((n*m) log(n*m))` complexity and, thereafter, shows lower numbers compared to Pagination Workload.

Couchbase Cloud outperforms MongoDB in JOIN Workload by demonstrating at least a three-fold higher throughput and lower latencies on all cluster sizes.

5. Conclusion

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

Although DBaaS does 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 Cloud showed better performance across all the evaluated workloads in comparison to other databases. In case of queries, Couchbase Cloud provides sufficient functionality to handle the deployed workloads. Furthermore, the query engine of Couchbase Cloud 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 Cloud ensures a satisfactory level of scalability across these operations.

MongoDB Atlas produced comparatively decent results. MongoDB is scalable enough to handle increasing amounts of data and cluster extension. Under this benchmark, the only issue we observed was that MongoDB did not support `JOIN` operations on sharded collections out of the box. This resulted in a negative impact and poor performance in JOIN Workload.

Amazon DynamoDB is significantly different from the other databases, because it looks like a pure service without proper tuning. Only two parameters can be changed:read and write capacities. In this case, read and write capacities have been calculated depending on the cost of other databases for each workload. Unfortunately, Amazon DynamoDB did not provide competitive results. It produces a large volume of failed requests. Additionally, Amazon DynamoDB did not take part in several workloads, because the data model had to be changed to get competitive results and, thus, cannot be compared to other databases.

Overall, each database performed worse than the results from our previous report, where each database was deployed manually. However, while manual deployment provides more configuration options and better tuning, it also takes longer and costs more.

6. About the Authors

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

His primary skills are software architecture and system design. He took part in numerous projects connected with processing and distributing huge amounts of dataarrays. 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 at Altoros.

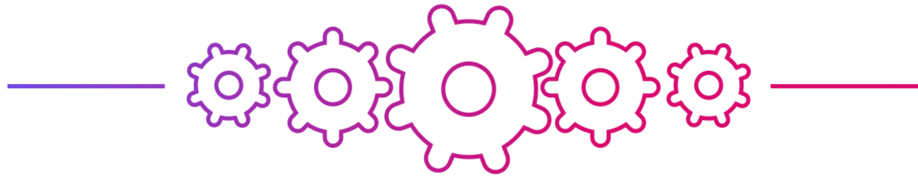
He has a solid software development background. He is focused on maintaining, designing, customizing, upgrading, and implementing complex software architectures, including data-intensive and distributed systems. Artsiom dedicates much of his spare time to these activities, and now he is one of the contributors 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. He has worked for several years as a full-stack engineer and has designed durable distributed systems. Ivan is able to create complex architecture solutions, adopt systems for production usage, and is keen on resolving any engineering problems.

Sergey Bushik is a Lead Software Engineer.

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



Altoros is 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. For more, please visit www.altoros.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.