Couchbase

WHITEPAPER

Comparing Vector Search Capabilities of MongoDB and Couchbase Using VectorDBBench

Contents

INTRODUCTION

The rapid adoption of machine learning, recommendation systems, and generative AI has driven the need for databases capable of efficiently storing and querying high-dimensional embeddings (vectors). These embeddings represent unstructured data such as images, audio, or text in numeric form and enable similarity-based retrieval through vector search operations. As database vendors increasingly integrate Approximate Nearest Neighbor (ANN) algorithms directly into their data engines, the question arises how well these emerging implementations perform in practice.

This report presents a data-driven evaluation of the vector search capabilities of Couchbase and MongoDB™. Both systems have recently introduced native support for vector indexes and ANN-based search operations, but they differ substantially in their architectural designs and index strategies. The goal of this study is to quantify and compare their performance characteristics under controlled and reproducible conditions.

The benchmarking experiments were conducted using VectorDBBench, a widely adopted open source benchmarking framework for vector databases. All workloads were executed on hardware-equivalent configurations, using both standard and large-scale datasets (100M and 1B vectors) to assess scalability and performance across different recall levels. The benchmark focuses on key performance indicators such as throughput (queries per second) and latency at different recall levels.

By systematically comparing these systems across multiple workloads and dataset sizes, this study aims to provide a transparent and reproducible analysis of how each database performs in real-world vector search scenarios, thereby offering valuable insights for practitioners evaluating vector search capabilities.

BACKGROUND

MongoDB

MongoDB is a popular NoSQL database that stores data in JSON-like documents organized into collections. Natively, MongoDB comes with its MongoDB query language (formerly known as MQL) to run operations against the database. In a nutshell, in clustered setups, MongoDB uses a primary-secondary approach where all inserts and updates have to go through the primary and are asynchronously synced with the secondaries via the oplog. Normally, a MongoDB cluster consists of one primary and two secondaries or multiples thereof (sharded setup).

For running vector workloads, users are required to use one or more additional search nodes within the clusters. The task of these search nodes is to store a vector index. According to MongoDB, scaling out the number of search nodes is the primary mechanism to achieve higher throughput.

MongoDB uses the Hierarchical Navigable Small Worlds (HNSW) algorithm and index structures for ANN. MongoDB supports two main parameters for creating the index. `maxEdges` defines the maximum number of edges that a node in the index can have in the HNSW graph. `numEdgeCandidates` controls the maximum number of nodes to evaluate to find the closest neighbors to connect to a new node. In this evaluation, we use the default `maxEdges = 32` and `numEdgeCandidates = 100`. The command we use to create the index is shown in the appendix of this document.

For querying the database, alongside the number of items to return (`limit`) and the query vector, MongoDB requires the `numCandidates` parameter. It defines the number of nearest neighbors used in the search, allowing a trade-off between latency and recall.

Couchbase

Couchbase is a distributed NoSQL database that stores data in JSON documents organized into collections and scopes. It uses SQL++ for querying and a shared-nothing architecture where all nodes can serve reads and writes, with indexing, search, and analytics as separate services.

For vector workloads, Couchbase supports vector search which builds on Couchbase's Index and Search Services and provides vector index support. Vector search supports the following three index types:

- **Hyperscale vector indexes** – Powered by the Index Service, this is optimized for pure vector searches at scale (billions of vectors). They provide high accuracy even with large dimensions, maintain a low-memory footprint, and support concurrent inserts and queries.
- **Composite vector indexes** – Combines a vector column with scalar fields. Useful when queries need to filter on scalar attributes (e.g., genre, location, supplier) before running a similarity search.
- **Search vector indexes** – Introduced in version 7.6, this index can pair vectors with full-text search (FTS) and geospatial search. It enables hybrid queries that mix semantic, text, and location filters.

In this evaluation, we use Couchbase's hyperscale vector indexes. Hyperscale vector indexes have two major parameters to control index creation: `nlist` defines the number of Voronoi cells the index uses. Each cell is represented by its centroid. For our experiments, we use `nlist = 200_000`. We also use `train_list = 100_000`, which controls the number of vectors that Couchbase considers when training for centroids in the dataset.

Quantization simplifies vector data so it consumes less space in memory and on disk. Couchbase supports product and scalar quantization. It is a parameter to be set for index creation. For this report, we use SQ4 scalar quantization which maps floating point values to low-dimensional (4-bit) integers. The command we use to create the index is shown in the appendix of this document.

When using hyperscale vector indexes, vector search first identifies the centroid vector closest to the query vector. It then uses the `nprobe` parameter to search neighboring Voronoi cells closest to the centroid and returns the top `k` number of vectors. In our experiments, we vary `nprobe` to trade off between latency and recall.

METHODOLOGY

The following sections describe the methodology we used for running the experiments. This includes the hardware and software configurations, the applied benchmark, and the selected datasets used for this evaluation.

Hardware Setup

At the time of the evaluation, Couchbase's vector support was not enabled in Couchbase Capella™. For this reason, the Couchbase clusters used in this evaluation are deployed on AWS EC2 virtual machines. We use Couchbase Enterprise version 8.0.0-3444, which is a prerelease. For MongoDB, we rely on MongoDB's Atlas cloud service to ensure we have access to a solid MongoDB installation with the most recent enterprise features. We use MongoDB version 8.0 for the evaluation at hand.

The hardware setup for both competitors is equivilent. The exact hardware specifications are shown in Table 1. In addition to the hardware required to run the database systems, we use one further AWS EC2 virtual machine of type r7i.8xlarge (32 vCores, 256GB RAM) to run the client-side workload.

Table 1. Hardware Set-up for MongoDB and Couchbase installations

	Data Nodes	Index/Query Nodes	Version
MongoDB	M80 replica set Three nodes, each with: 32 vCores 128GB RAM 750GB storage	4 x high-CPU S80 each with: 64 vCores 128GB RAM 3.2GB NVMe storage	v8.0 on MongoDB Atlas @AWS
Couchbase	3xm6gd.8xlarge each with: 32 Graviton2 cores 128GB RAM 1.9TB NVMe storage	4xc8gd.16xlarge each with: 64 Graviton2 cores 128GB RAM 2x1.9TB NVMe storage	Couchbase Enterprise 8.0.0-3444 @AWS EC2

In the case of Couchbase, all nodes are deployed in the same availability zone, including the machine running the client-side workload. In the case of MongoDB, the distribution of cluster nodes across availability zones can neither be controlled nor retrieved. Hence, we cannot place the workload machine in specific availability zone. The option for the closest possible placement is to use the same region as the database cluster, which is what we did.

For the 1 billion vector data set, Couchbase delivers 500,000+ vector queries per dollar, while MongoDB provides just under 300 queries per dollar, a value gap of roughly 2,000x in real throughput per dollar spent on the same underlying hardware and recall levels. This metric is calculated by converting queries-per-second into queries-per-minute and dividing by the per-minute database costs.

Workload: VectorDBBench

For measuring vector search performance, a benchmark suite that simulates realistic vector search use cases is required. With the rise of vector databases, several open source **vector search benchmark suites** have been released. **VectorDBBench** has gained much traction over the past 18 months and is considered the current de facto standard in vector database benchmarking. This study uses the VectorDBBench version 1.0.6.[1]

VectorDBBench supports three modes of operation: (i) The *capacity mode* iteratively loads an ever-increasing number of data points/rows into the database until a predefined insert timeout threshold is violated. (ii) The *search mode* evaluates four different operation types, each of which produces its own set of metrics/results. The majority of operation types query the database content for test vectors in order to measure the performance of the database system. Some also measure the quality of the results. (iii) The *filtering search mode* resembles the search mode, but uses additional filters (i.e., `WHERE` clause) in the queries. (iv) Finally, the newly introduced *streaming mode* measures the search performance while maintaining a constant insertion workload.

For the evaluation at hand, we make use of the search mode.

DATASETS

VectorDBBench ships with different datasets that are available to its users out of the box. More precisely, the documentation distinguishes between four types of datasets ranging from xlarge (up to 100M data points) to small (up to 100K data points).

[1] At the time of writing, the current version was VectorDBBench 1.0.6.

In addition, VectorDBBench allows users to define a custom dataset as long as the data is available in a set of files that follow a specific format and naming schema. VectorDBBench uses three different files that all need to be available in a Parquet format:

- The `train.parquet` file contains the dataset to be loaded into the database. The entries in the file contain an identifier (`id`) and an embedding (`emb`) with the latter being the actual vector.
- The `test.parquet` file contains the (test) queries to issue against the database. Each entry comes with an identifier (`id`) and the search vector (`emb`). VectorDBBench issues all vectors from that file against the database. All datasets that ship with VectorDBBench contain 1,000 entries in that file.
- The `neighbor.parquet` file contains the sample solution for each vector. Each entry references the query identifier (`id`) and contains the ids of the vectors closest to the search query (`neighbors_id`) in descending order. For example, `neighbor_id[0]` is closer to the search query than `neighbor_id[20]`.

SEARCH MODE OPERATIONS AND METRICS

The search mode performs four operations types which produce the following metrics:

1. During the *load operation*, all data from a selected dataset is ingested into the database. In all bindings provided by VectorDBBench, this is done with a single client (1 thread/process) in batches of a configurable size. For this evaluation, we updated the bindings for MongoDB and Couchbase to support parallel inserts. The primary metric VectorDBBench reports for the load operation is the load time.
2. In the *optimize operation*, the binding is allowed to perform an arbitrary set of steps to optimize the database for later query phases. This may include flushing segments, (re-)building an index, and other database-specific steps. Obviously, different bindings (databases) apply different means of optimization during this step. The primary metric VectorDBBench reports for the optimize step is the optimization time.
3. In the *concurrent search operation*, a certain number of clients (threads/processes) concurrently query the database for vectors leading to kNN or ANN queries. The number of parallel clients to use is configurable and we vary it in this evaluation. Similarly, `k`, the number of nearest neighbors to be returned by a query can be configured. We use the default, `k = 10`. The primary metric VectorDBBench reports for the concurrent query step is throughput (queries per second) per concurrency level. Recently, it also began reporting different latency metrics, including P95, P99, and average.
4. In the *serial search operation,* a single process queries the database for a set of test vectors. In addition to measuring latencies, this step also evaluates the query results and compares them to the ideal result set provided through the `neighbor.parquet` file. For serial search, VectorDBBench reports different latency metrics as well as recall, which captures the quality of the results based on the comparison with the ideal result set.

Evaluation Datasets

For the evaluation in this paper we use two different datasets: Case A uses the LAION 100M dataset that ships with VectorDBBench. The dataset contains 100 million data points with 768 dimensions each. Case B aims at a larger dataset that contains 1 billion data points. Unfortunately, VectorDBBench does not ship datasets of this size. Their largest dataset category, xlarge, covers datasets with 100 million (100M) vectors.

A dataset with 1 billion elements was released in 2010 by Laurent Amsaleg and Hervé Jégou from CNRS/IRISA in France. It is called ANN_SIFT1B and its vectors have 128 dimensions. The ANN_SIFT1B dataset contains a train file (called base set), a test set (called query set), and a neighbor set (called groundtruth), so that the dataset contains all information required by VectorDBBench. Yet, ANN_SIFT1B is only available in `bvecs` and `ivecs` formats, which is a binary format not compatible with VectorDBBench.

In order to use this dataset for our evaluation, we converted the `bvecs` and `ivecs` files into Parquet format. We split the train data into 1,000 different files, each with 1M entries. Once the dataset is loaded into the database, we run a validation script which compares the entries of the `bvecs` file and the content from the database to make sure the conversion worked and no vector got corrupted in the entire process.

Vector Indexes

Vector search is the capability of a database system to find vectors stored in the database that are closest (nearest neighbors) to the vector a user searches for (search vector). Due to the computational complexity of doing exact nearest neighbor search, most database systems rely on AANN search, which is a lot faster than exact search, but introduces some non-determinism in the search.

For implementing ANN search, specialized index types exist. The commands used for creating the vector indexes of the respective database systems are listed in the appendix of this document.

Workload Parameters

Vector search performance can be looked at from various angles. Obviously, the classical database performance metrics of throughput (queries per second, QPS) and latency still exist. Yet, because ANN search is non-deterministic, these metrics should be considered in the context of the query's recall,[2] since low latency and high throughput is easier to achieve with less precise results.

[2] Recall is the fraction of relevant instances that were retrieved from a query. VectorDBBench measures recall by counting how many of the `k` results returned from a query are contained in the first `k` elements of the neighbor list. It does not consider the order.

In the scope of this study, the primary parameters affecting recall are `nProbes` for Couchbase and `numCandidates` for MongoDB. Both parameters determine how intensely the index (and hence, the search space) shall be sampled to find vectors close to the search vector.

In our experiments, we apply several values to each of the parameters, e.g., `10`, `50`, and `100`; and for each of the values, we search for the maximum sustainable throughput by varying the number of concurrent clients. We start with a concurrency level of 32 clients and increase it to 64, 128, 256, and 386. If a setup pushes the index nodes beyond an average CPU utilization of 90%, we stop and do not evaluate the next higher configuration.

In this document, QPS_{max} denotes the highest throughput achieved for a specific value of `nProbes/numCandidates`. For each QPS_{max}, we also report the concurrency level that achieves it, the average query latency, and the 95th percentile of the query latency.

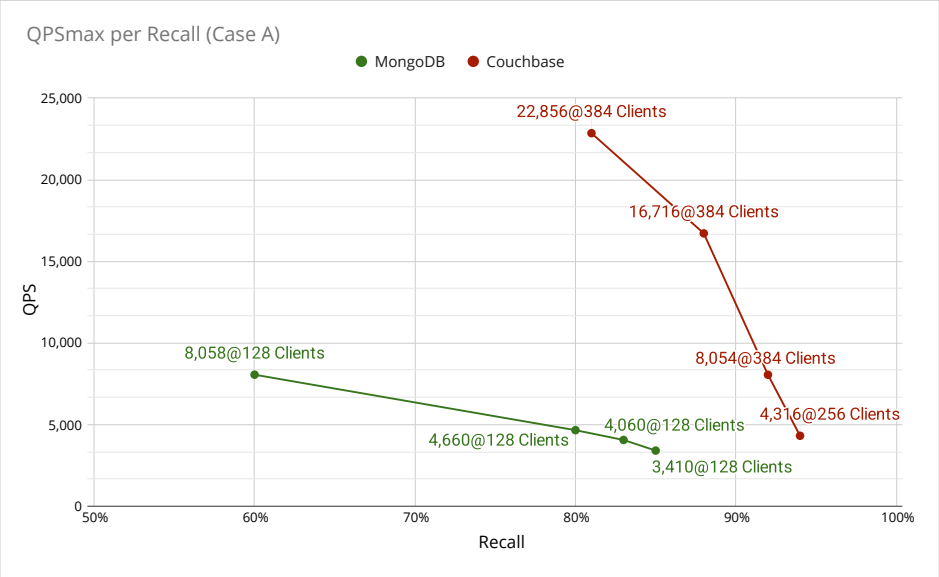
RESULTS

Case A: 100M with 768 Dimensions

For the 100M LAION dataset, we measured QPS_{max} and recall for different values of `nProbes` and `numCandidates`, respectively. Figure 1 illustrates this evaluation by showing the results for `nProbes` and `numCandidates` set as `10`, `50`, `70`, and `100`. As expected, the recall increases monotonically with higher values for `nProbes` and `numCandidates` (not shown in the plot).

Figure 1 also shows that QPS_{max} decreases when recall increases. This is to be expected because the amount of computational resources stays constant, while the computational effort required to produce a response increases. For Couchbase, this results in a decrease in QPS from 22,856 QPS for a recall of 81% to 8,054 QPS for a recall of 92% (-65%) and 4,316 QPS for a recall of 94% (-46% compared to a recall of 92%).

Figure 1. QPS_{max} per Recall for 100M Vectors



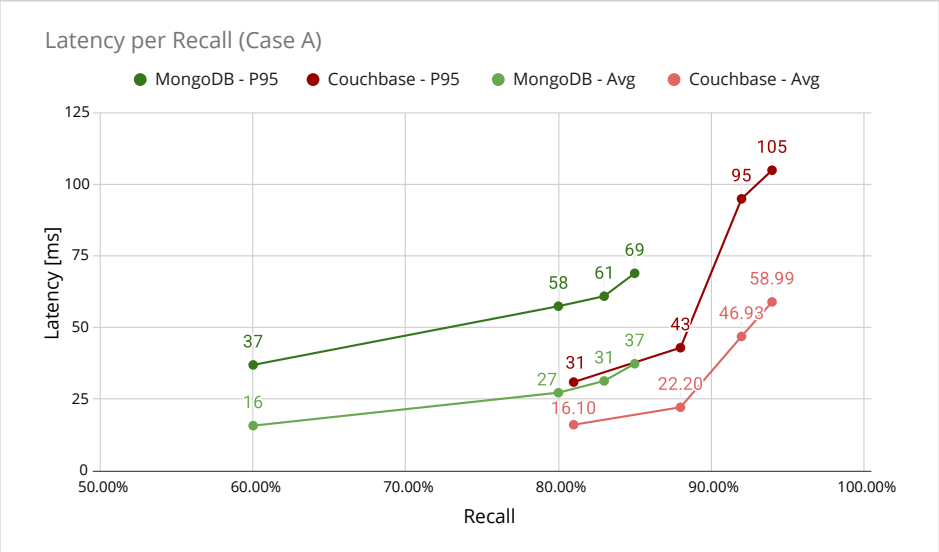
For MongoDB, we see a similar decrease, yet with a lower baseline. For a recall of 60%, we can achieve 8,058 QPS. This further decreases to 4,660 QPS for a recall of 80% and 3,410 QPS for a recall of 85% (-27% compared to a recall of 80%). 85% is the greatest recall we could achieve for MongoDB in our experiments.

With respect to concurrent workloads and overall throughput, MongoDB cannot achieve parallelism beyond 128 concurrent clients. Couchbase is able to get up to 384 clients for most data points, and supports 256 with a recall of 94%.

Overall, the data clearly shows that the Couchbase curve dominates the MongoDB curve. With a recall around 80%, Couchbase outperforms MongoDB by almost a factor of five (22,856 QPS vs. 4,660 QPS). With a recall of 94%, Couchbase achieves 25% higher QPS than MongoDB, which has a recall of 85%.

Figure 2 shows the average latency per query and the 95th percentile (P95) of the query latency. The experiments represented by the data points are the same ones as shown in Figure 1. Hence, the MongoDB latency metrics at 60% recall were measured in the same run that led to the data point at 60% recall and 8,058 QPS with 128 clients in Figure 1. While the primary goal of the evaluation is to find the maximum QPS, both database systems expose an acceptable latency, even at 85% recall, MongoDB's average latency is only 37ms and its P95, 69ms. Yet, at 88% recall, Couchbase's latency barely exceeds 20ms and its P95 is only at 43ms. For larger recall levels, latency increases, but even at 94% recall, Couchbase remains well below 60ms, while P95 barely exceeds 100ms. Overall, the results in Figure 2 align with the earlier discussion on QPS.

Figure 2. P95 and Average Latency per Recall for 100M Vectors



Case B: 1B with 128 Dimensions

For the SIFT dataset of 1 billion data points, we measure QPS_{max} and recall for `nProbes` and `numCandidates` set to `10`, `20`, `50`, and `100`, respectively. Figure 3 illustrates the results of this evaluation.

For Couchbase, the results show a QPS_{max} of 19,057 QPS at a recall of 66%. With a recall of 77%, the achievable throughput is still 9,877 QPS (-48%), and 3,153 QPS with 88% recall (-68% compared to 77% recall). With 93% recall, 703 QPS can still be achieved (-78% compared to 88% recall). For MongoDB, the results are null since it is unable to reach two-digit throughput for any of the configurations. The best throughput we can achieve is 6 QPS with 256 parallel clients at a 57% recall. With an increase in the recall level, the throughput decreases even more: at 89% recall, it is only 2 QPS with 64 parallel clients. This is the maximum recall that could be achieved with our methodology for MongoDB.

Figure 3. QPS_{max} per Recall for 1B Vectors

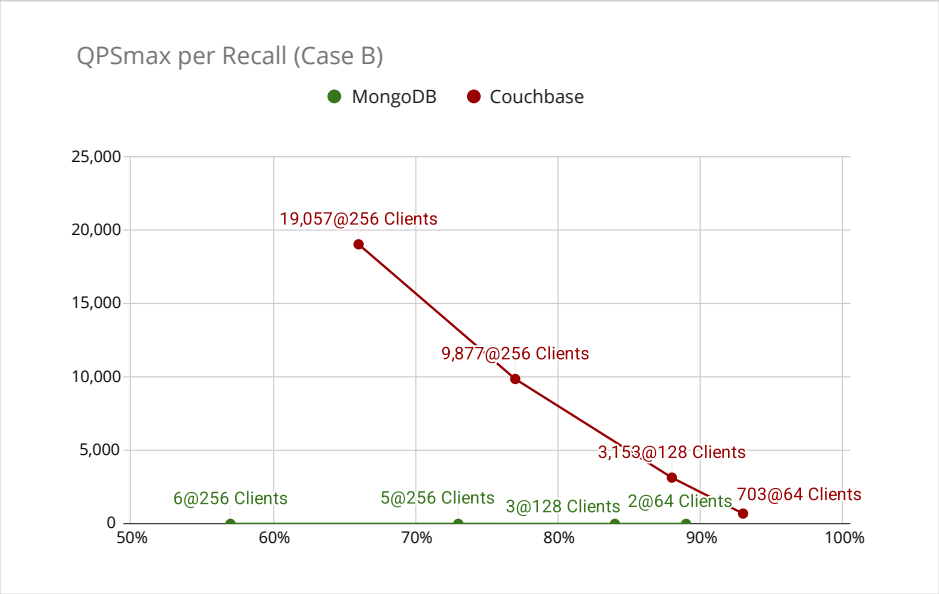
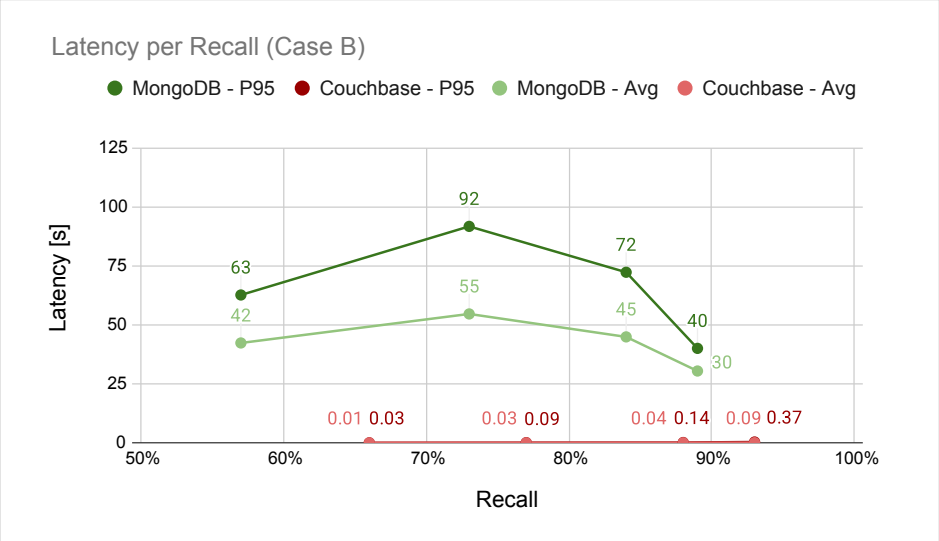


Figure 4 shows the relationship between recall and latency (average and P95). It confirms the results from Figure 3, showing that the average query latency for MongoDB is beyond 30 seconds for all levels of recall considered in this evaluation. The P95 for MongoDB is above one minute in all cases except the one with the highest recall, where it is 40 seconds. When going from a recall of 73% to 84% to 89%, the drop in latency for MongoDB is a consequence of using less concurrent clients for higher recall levels, namely 256, 128, and 64, respectively.

Figure 4. Latency P95 and Average Latency per Recall for 1B Vectors



For Couchbase, both average latency and P95 latency increase when increasing the recall level. Yet, in all evaluations for Case B, the average latencies remain below 100ms, which is more than 300 times lower than MongoDB's best result. The P95 latency increases up to 370ms for a recall of 94%, more than 100 times lower than MongoDB's best result.

Discussion

The evaluations for both Case A and Case B leave Couchbase as the clear winner of our comparison. The following discussion offers insights and trends that we can derive from comparing the results of both cases related to a single system's performance between Case A and Case B.

COUCHBASE

Recall: For both cases, we experience an increasing recall metric when increasing the `nProbes` parameter. While for Case A, the lowest recall we measured is 81% (`nProbes = 10`), it is only 66% for Case B. This behavior can be explained with the difference sizes of the dataset; overall, sampling with `nProbes = 10` is insufficient for 1 billion data points. For `nProbes = 100`, the recall is aligned and both cases show similar results (94% vs. 93%).

Latency: The latencies for Couchbase develop unevenly between Case A and Case B. For the respective lower recall level (`nProbes = 10`), latencies are similar for both cases. For higher recalls (and values of `nProbes` respectively), there is an increase of more than 50% for average latency and more than a factor of two for latency P95. This is a consequence of using 10 times as many data points in Case B, which increases the index size. For Case A, large parts of the index can reside in memory, while the larger index for Case B only fits partially into memory. Therefore, search queries over the larger dataset need to access disk more often for Case B, particularly, when more sampling is used (larger values of `nProbes`).

Throughput: For both cases, Couchbase starts off at around 20k QPS for `nProbes = 10` and decreases for larger values of `nProbes` / higher recalls. Yet, the decrease is more significant for Case B for which the system can only sustain 700 QPS for `nProbes = 100`. In contrast, in Case A, Couchbase can sustain around 4.3k QPS. This is an immediate consequence of the fact that the level of concurrency had to be lowered for Case B compared to Case A, which is a consequence of the larger dataset and more frequent disk access.

MONGODB

Recall: For MongoDB, the recall increases for both cases when increasing `numCandidates`. Yet, the tendency is different compared to Couchbase. For MongoDB, Case B produces higher recall rates in all cases except for `numCandidates = 10`. For `numCandidates = 100` in Case B, it reaches 89% – the highest value measured for MongoDB in all of our experiments. In Case A, it only achieves 85% for this `numCandidates`.

Other metrics: A fair comparison of the other metrics is not possible. MongoDB's results are significantly worse for Case B. Throughput drops from several thousand QPS to a handful QPS, and latency metrics skyrocket from sub-seconds to minutes. We suspect that this significant drop in performance occurs because MongoDB's index becomes so large with 1 billion data points that much of it does not fit into memory. This, in turn, requires search operations to frequently go to disk, thereby increasing latency and bringing throughput to a halt.

CONCLUSION

This paper compared the vector search performance of Couchbase 8.0 with MongoDB 8.0. We used VectorDBBench, the de facto standard benchmarking suite for vector workloads, and ran it with two different datasets. The first dataset had 100 million vectors with 768 dimensions each and the second dataset had 1 billion vectors with 128 dimensions each.

In the experiments, we varied the computational complexity for search queries, which impacted recall and the quality of the query results. For each recall level, we identified the number of parallel clients that achieved the highest throughput.

Based on the 100M dataset, Couchbase achieves higher throughput across all recall levels while maintaining both higher recall and lower latency compared to MongoDB. For the 1B dataset, Couchbase performance only slightly degrades compared to the 100M dataset, while MongoDB is no longer able to achieve any production-grade performance.

APPENDIX

Commands to Create Vector Indexes

COUCHBASE

```
CREATE VECTOR INDEX `vector_index` on `database`(`emb`
VECTOR)

WITH {

     "nodes":[

           "<node1>",

           "<node2>",

           "<node 3>",

           "<node4>",
```

```
        ],
        "num_replica":3,
        "dimension":<data set dimensions>,
        "similarity":"L2",
        "description":"IVF200000,SQ4",
        "train_list":1000000
}
```

<data set dimensions>

- 768 for Case A
- 128 for Case B

Other parameters:

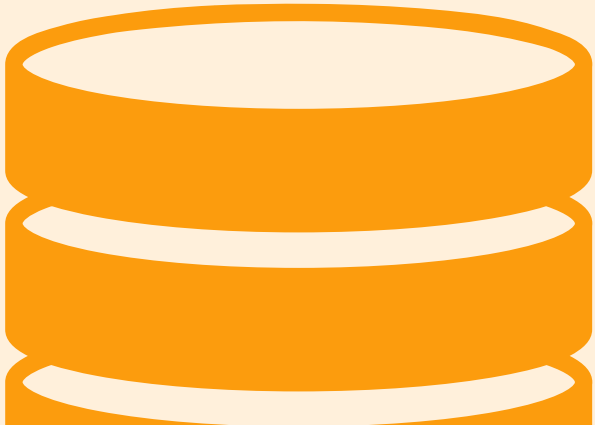
index memory quota was set to 100GB

MONGODB

```
{
      'type': 'vectorSearch',
      'fields': [
         {
             'type': 'vector',
             'similarity': 'euclidean',
             'numDimensions': <data set dimensions>,
             'path': 'vector',
             'quantization': 'scalar',
             'hnswOptions': {'maxEdges': 32}
}]}
```

<data set dimensions>

- 768 for Case A
- 128 for Case B



Modern customer experiences need a flexible database platform that can power applications spanning from cloud to edge and everything in between. Couchbase's mission is to simplify how developers and architects develop, deploy and run modern applications wherever they are. We have reimagined the database with our fast, flexible and affordable cloud database platform Capella, allowing organizations to quickly build applications that deliver premium experiences to their customers – all with best-in-class price performance. More than 30% of the Fortune 100 trust Couchbase to power their modern applications.

For more information, visit www.couchbase.com and follow us on X (formerly Twitter) @couchbase.