WHITEPAPER



Couchbase vs. MongoDB™ for Query



INTRODUCTION



Couchbase is a distributed NoSQL document-oriented database with a core architecture that supports a flexible data model, easy scalability, consistent high performance, always-on 24x365 characteristics, and advanced security. It has been adopted across multiple industries and in the largest enterprises for their most business-critical applications. Many of those customers, including Equifax, Nuance, and LinkedIn, have gone through rigorous evaluations of Couchbase alongside MongoDB, and have chosen Couchbase based on a strong set of differentiated capabilities and architectural advantages, including:

- · Scale-out and high availability
- Global deployment
- SQL/SQL++ query for JSON
- Hybrid Operational and Analytical Processing (HOAP)
- · Full-text search
- Vector search
- Time series
- Geospatial
- · Server-side eventing and functions
- Embedded mobile database

In this paper, we will focus on how MongoDB compares to Couchbase when it comes to query. Couchbase offers a declarative query language, SQL++, based on open standards, providing users with the familiarity of SQL and the flexibility of JSON and nested data structures.

MongoDB's Proprietary Query API and Aggregation Framework

MongoDB provides a query method, find(), to query a collection with simple filtering and projection. For more complex operations, users have to use the aggregation framework to join, process, and aggregate multiple documents to return the result.

The MongoDB query method and aggregation framework are proprietary and procedural, with limited expressive power and poor performance.

- **Proprietary.** MongoDB's query method and aggregation framework is nonstandard, and for developers with RDBMS experience, it is a huge learning curve, leading to longer development cycles and complexity in maintaining the applications.
- **Procedural.** Eliot Horowitz, MongoDB co-founder, said: *"MongoDB aggregation is similar to Unix pipeline. The output of one stage goes into another...[it's] very procedural. Lets you think about in a very procedural way."* The problem with this approach is that the burden to optimize a query is now on the developers, and a pipeline optimized



for certain data distribution may not work if the data changes, and requires manual intervention again to optimize it. So this adds to the complexity of application development and ongoing maintenance.

- **Complexity.** MongoDB's aggregation framework is complex even for simple queries that you can easily express in SQL. For example, you can use **this link** to translate your favorite SQL statements into MongoDB's proprietary syntax to see the complexity of MongoDB's aggregation framework.
- **Limited expressive power.** MongoDB's proprietary query method and aggregation framework has many limitations because, unlike SQL++, it is not based on proven tuple calculus. For example, it has limited support for joins and cannot join across sharded collections. Read more about JOIN capabilities in both.
- Limited query optimization. Because the aggregation pipeline is procedural and lacks expressive power, applications need to compensate by doing complex data processing on the client side, leading to complexity and poor performance. See the YCSB-JSON benchmark comparing the performance and scalability of queries between Couchbase and MongoDB.
- Local indexes. MongoDB documentation states the limitation of having local indexes that share the same shard key as the local data as follows: "If a query does not include the shard key, the mongos must direct the query to all shards in the cluster. These scatter gather queries can be inefficient. On larger clusters, scatter gather queries are unfeasible for routine operations." Furthermore, indexes are maintained synchronously and as more indexes are added, writes slow down further and further.

SQL++: FLEXIBILITY OF JSON + EXPRESSIVE POWER OF SQL

Unlike MongoDB, Couchbase adopts open standards and extends SQL to support JSON. Couchbase Server stores JSON documents and supports a query language called SQL++.

In fact, SQL++ is the first commercial implementation of the SQL++ specification, a query language for semi-structured data based on the JSON data format that was developed by Prof. Yannis Papakonstantinou and others at University of California San Diego¹. Like its predecessor, SQL++ extends the mathematical foundation of tuple calculus to support the richness of JSON, making it a highly expressive query language that encompasses both SQL and the JSON data model. Don Chamberlin, co-inventor of SQL, wrote a **tutorial on SQL++** using SQL++ for the query examples. We have seen other vendors increase their support of SQL++ and we believe it will become the de facto query language for NoSQL systems.

Here are the key benefits of SQL++ over the proprietary MongoDB query method and aggregate framework:

• Familiarity of SQL. SQL was designed as a query language that business users can understand and express. SQL++ extends SQL to support JSON, and therefore



838

developers find SQL++ familiar and easy to learn. Expressing the same query in MongoDB requires a developer to write code using MongoDB's proprietary and complex query API and aggregation framework. See below for examples of the simplicity of SQL++'s versus the complexity of MongoDB's proprietary API.

- **Declarative.** SQL++ is declarative. You express what you want to accomplish and the database system will figure out the most optimal way of accessing and processing the data to get the results. Couchbase builds on decades of advances in database research in query optimization and execution to deliver high performance at scale.
- **Expressive power.** Because of the strong mathematical foundation, SQL++ is a highly composable and expressive query language. Each query uses JSON documents as its input, and the output is another JSON document. SQL++ is not constrained by how data is stored physically.
- Global indexes. Couchbase supports global indexes that can be partitioned differently from the underlying data to optimize different queries for higher query throughput and lower latencies. Each index can be replicated multiple times for high availability and replicas are automatically load balanced. Each index can also be partitioned depending on the expected size and range scan needs. Couchbase can support as many indexes as needed to improve query performance without impacting write latency. This also means that the data model does not need to be changed for different query access patterns as multiple different indexes can align the data in different ways.
- **Tunable consistency.** Global indexes are updated asynchronously to the data mutations which is why write throughput is not impacted by more and more indexes being added. They are constantly updated with an advanced memory-memory database change protocol. Each query can then be made strongly consistent on a per-query basis which allows the application developer, who has context on what is most important in given parts of the application, to decide what the right priority is between consistency and latency without affecting the resources of the rest of the platform.

SIMPLICITY OF SQL++ VS. COMPLEXITY OF MONGODB'S PROPRIETARY QUERY API

Here is an example of a query that finds all the destination airports starting from San Francisco International Airport (SFO). Even for a simple JOIN, it requires a cumbersome pipeline syntax on the MongoDB query. You need another "let" stage to create the local variables for airport attributes to differentiate between the two collections. You also need an additional \$match clause to eliminate non-matching (empty) airline docs, followed by grouping and sorting. As you can see, the MongoDB query is longer and much more complex to do the same job as Couchbase SQL++.



SQL++ queries look the same as the ANSI SQL you are familiar with, while the MongoDB API becomes intractably complex with fairly simple queries. See this blog for a detailed comparison between JOIN support in Couchbase and MongoDB.

Couchbase SQL++

```
SELECT DISTINCT route.destinationairport
FROM `travel-sample` airport JOIN `travel-sample` route
ON (airport.faa = route.sourceairport AND route.type = "route")
WHERE airport.type = "airport"
AND airport.city = "San Francisco"
AND airport.country = "United States"
ORDER BY route.destinationairport
```

MongoDB



PERFORMANCE COMPARISON: COUCHBASE VS. MONGODB

Applications using SQL++ are easier to write and maintain because of the expressive power and familiarity of SQL++ to developers. The system also scales better because it automatically picks the optimal execution path, saving application developers time and removing complexity. Read more about a deep dive into Couchbase query optimization.

Because of the lack of a good optimizer and limitation in the aggregation pipeline, MongoDB performs poorly even for simple JOIN queries involving sharded partitions. As you can see from one of the tests in the YCSB-JSON benchmark, Couchbase is 6x to 18x higher in throughput than MongoDB because MongoDB has to ship the data to the client and process the joins in the application. Transferring large amounts of data between the server and client is inefficient and impacts the overall system throughput by saturating the network.



Here is a simple example from the YCSB-JSON benchmark for a pagination query that skips the first 1,000 documents (OFFSET 1000) and returns the next 10 documents (LIMIT 10):

SELECT * from YCSB
WHERE address.country = "USA"
ORDER BY address.zip
OFFSET 1000 LIMIT 10



For MongoDB, this query requires each of the shards to sort and return 1,010 documents to the query coordinator, which then has to aggregate the documents from all shards, re-sort the documents, and then skip the first 1,000 documents to return the final 10 documents to the application. This scatter-gather approach with local indexes results in processing by all shards (scatter), and returning of data from all shards to the query coordinator (gather), and then further processing by the query coordinator to finalize the results to the application.





- The mongos coordinator accepts the query from the application and distributes the query to all the shards.
- Each shard processes the query and sorts the data locally.
- Each shard then returns the first 1010 documents (OFFSET 1000, LIMIT 10) to the mongos query coordinator.
- The mongos query coordinator aggregates the documents returned from all the shards, re-sorts, and re-orders the documents.
- The mongos query coordinator then skips the first 1000 documents (OFFSET 1000), and returns the 10 documents (LIMIT 10) to the application.

Contrast this with the efficient query processing of Couchbase using global indexes. The Query Service simply pushes the request down to the Index Service to filter and skip the first 1,000 documents, and return the keys to the next 10 documents. It then fetches the 10 documents from the Data Service and returns the documents to the application. This is an order of magnitude more efficient than MongoDB as shown in the **YCSB-JSON benchmark**.



- Client submits the query to the Query Service.
- 2. The Query service parses and optimizes the query, and sends the scan request to the right index.
- The index service filters filters and skips the first 1000 documents (OFFSET 1000), and returns the keys to the next 10 documents (LIMIT 10) to the Query Service.
- The Query Service fetches the 10 documents using the keys returned from the Index Service.
- 5. The Query Service returns the results to the application.

SUMMARY OF COUCHBASE SQL++ VERSUS MONGODB'S API

Query Capabilities	Couchbase SQL++	MongoDB
Declarative query	Yes	No
Based on standards	Yes	No
Scalable indexing	Yes, with global indexes that can be partitioned independently	Local indexes restricted to the same shard key as the underlying results in scatter- gather and slow queries
Query – performance at scale (see benchmarks)	High performance with global indexes and query optimization	Poor performance due to scatter-gather and JOIN limitations
SQL DML/DDL SUPPORT		
SELECT	Yes	db.collection.find()
INSERT/UPDATE/DELETE/MERGE	Yes	Partial – limited Bulk.find() expressions
CREATE/DROP INDEX	Yes	db.collection.dropindex()
JOIN Support		
INNER JOIN	Yes	Manually add a \$match stage to remove non-matching documents from \$lookup (Left Outer Join)
LEFT OUTER JOIN	Yes	\$lookup – Limited non-scalar support
RIGHT OUTER JOIN	Yes	No
Sub-Query with joins in FROM clause	Yes	No
SET OPERATORS		
UNION	Yes	Partial support with \$unionWith
INTERSECT	Yes	No
EXCEPT	Yes	No
Query Optimization		
Hash joins	Yes	No
Predicate pushdown to index	Yes	Very Limited
Aggregate pushdown to index	Yes	Very Limited
Pagination pushdown to index	Yes	Very Limited
Covered query	Yes	Does not support arrays, limited support of sharded collections
Functional indexes	Yes, can create indexes on functions and expressions	Lacks multi-expression and deep array index capabilities
Array indexing	Can index arrays nested at any level or any array expressions; filtering and unnesting exploits indexes for better performance	Can only index top-level arrays; poor performance for queries on nested arrays

In addition to its core querying capabilities, Couchbase SQL++ extends beyond traditional document queries to support a broad set of advanced capabilities. These include support for time-series data, full-text search (FTS), vector search for AI and semantic use cases, user-defined functions (UDFs), common table expressions (CTEs), graph traversals, geospatial, and AI-powered extensions for categorization, data masking, sentiment analysis, content generation, language translation, and more, all accessible through SQL++.

SQL++ throughout Couchbase:

- Couchbase Lite (the embedded Couchbase Mobile database) runs SQL++ directly on-device, enabling powerful local querying even in offline-first mobile applications.
- Couchbase Analytics provides isolated OLAP processing using the same SQL++ language, allowing complex analytical queries to run alongside operational workloads without impacting performance.

These consistent SQL++ extensions across core database services, mobile, full-text search, vector search, and analytics enable developers to write rich, advanced applications on a single platform – without having to learn or manage multiple query languages or systems.

CONCLUSION



MongoDB provides a long list of checkbox features, but many fail to work in concert with each other, leading to a database that cannot scale, perform, nor adapt to meet today's enterprise requirements. Ultimately, MongoDB is best suited for flexible data access where low latency, high throughput, multiple access patterns, geographic replication, or offline access are not required.

Couchbase, on the other hand, is routinely used for caching layers, sources of truth, and systems of record across high-scale and high-flexibility use cases, including offline-first mobile applications. By design, Couchbase is accessed and managed through a consistent set of APIs, and scaled, upgraded, and diagnosed as a single unit, making Couchbase a complete database platform that not only addresses the needs of today, but offers the flexibility to adapt to the needs of tomorrow.

To learn more, contact your Couchbase sales representative today or visit: couchbase.com | couchbase.com/downloads



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 consume 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

© 2025 Couchbase. All rights reserved.

