Couchbase

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 Staples, 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**
- **Full-Text Search**
- **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 (N1QL) 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 non-standard, and for developers with RDBMS experience, it is a huge learning curve, leading to a longer development cycle and complexity in maintaining the applications.
- **Procedural.** Eliot Horowitz, MongoDB CTO, 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, use http://www.querymongo.com/ 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 N1QL, it is not based on proven tuple calculus. For example, it has limited support for JOINs and cannot join across sharded collections. See the blog comparing JOIN support in Couchbase and MongoDB.
- **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.

N1QL: 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 N1QL. The name N1QL suggests "not first normal form," a reference to the fact that JSON documents, unlike relational data, may contain nested data structures.

In fact, N1QL is the first commercial implementation of SQL++, 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 Diego[1]. 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 an excellent tutorial on SQL++ using N1QL for the query examples. We expect other vendors to support SQL++ and we believe it will become the de-facto query language for NoSQL systems.

Here are the key benefits of N1QL 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. N1QL extends SQL to support JSON, and therefore developers find N1QL familiar and easy to learn. See the table below that shows the similarity between N1QL and Oracle SQL. 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 N1QL's simplicity vs the complexity of MongoDB's proprietary API.
- **Declarative.** N1QL 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, N1QL is a highly composable and expressive query language. Each query uses JSON documents as its input, and the output is another JSON document. N1QL 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

[1] https://arxiv.org/abs/1405.3631

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 N1QL vs Complexity of MongoDB's Proprietary Query API

Here is an example of a query that finds all the destination airports starting from 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 visually, the MongoDB query is longer and much more complex to do the same job as Couchbase N1QL.

N1QL 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 N1QL

```
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:

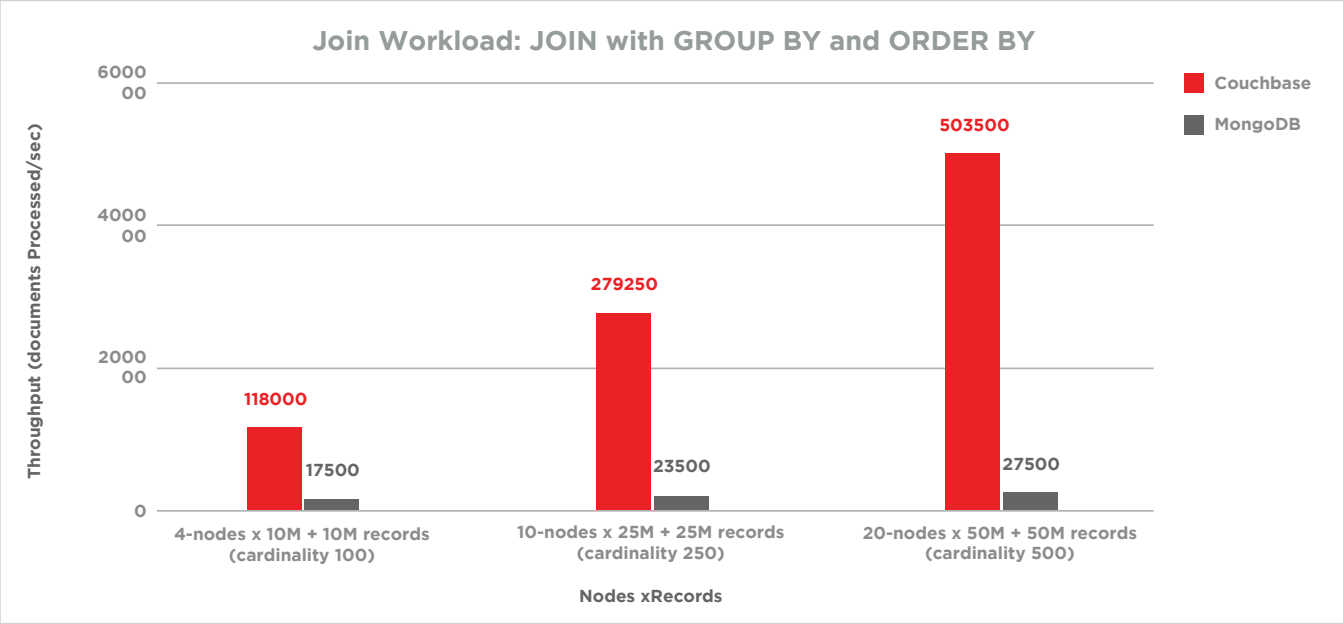
```
db.airport.aggregate([
   {
      $match: {
          $and: [
            {"type": "airport"},
            { city: "San Francisco"},
            { "country": "United States"}
          ]
      }
   },
   {
     $lookup:
       {
         from:"route",
         let: { rfaa : "$faa"},
         pipeline: [
           { $match:
               { $expr:
                 { $and:
                   [
                     { $eq: ["$sourceairport", "$$rfaa"]} ,
                     { $eq: ["$type", "route"] }
                   ]

                }
              }
           }
         ],
         as: "airline_docs"
       }
   },
   { $match: {"airline_docs": {$ne: []}} },
   {  $unwind: { path: "$airline_docs", preserveNullAndEmptyArrays: true }},
   { $project: { _id:0,  "airline_docs.destinationairport" : 1 }},
   { $group: {
      _id : "$airline_docs.destinationairport"
     }
   },
   { $sort: { _id : 1 }},
 ]);
```

Performance Comparison: Couchbase vs. MongoDB

Applications using N1QL are easier to write and maintain because of the expressive power and familiarity of N1QL to developers. The system also scales better because it will automatically pick the optimal execution path.

The database should automatically optimize and select the optimal execution plan to get the results, instead of application developers spending time figuring out how to lay out and manually optimize the aggregation pipeline as required in MongoDB. See this blog for a deep dive into the N1QL query optimization.

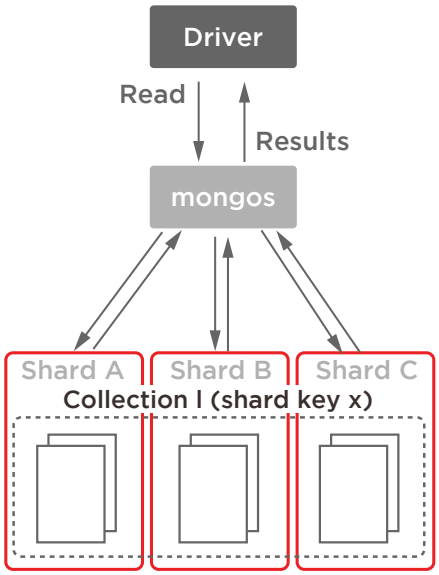
Because of the lack of a good optimizer and limitation in the aggregation pipeline, MongoDB performs poorly even for simple JOIN queries involving sharded partitions. As you can see from one of the tests in the YCSB-JSON benchmark, Couchbase is 6x to 18x higher in throughput than MongoDB because MongoDB 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 1000 documents (OFFSET 1000) and returns the next 10 documents (LIMIT 10):

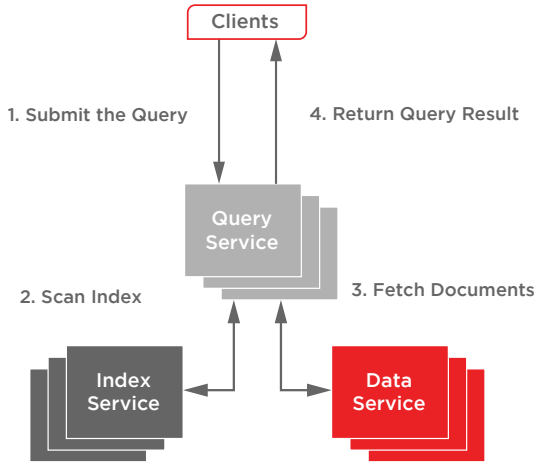
```
SELECT * from YCSB
WHERE address.country = "USA"
ORDER BY address.zip
OFFSET 1000 LIMIT 10
```

For MongoDB, this query will require each of the shards to sort and return 1010 documents to the query coordinator, which then has to aggregate the documents from all shards, re-sort the documents, and then skip the first 1000 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.



1. The mongos coordinator accepts the query from the application and distributes the query to all the shards
2. Each shard processes the query and sorts the data locally
3. Each shard then returns the first 1010 documents (OFFSET 1000, LIMIT 10) to the mongos query coordinator
4. The mongos query coordinator aggregates the documents returned from all the shards, re-sorts, and re-orders the documents
5. 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 down the request to the index service to filter and skip the first 1000 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 then MongoDB as shown in the YCSB-JSON benchmark.



1. 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
3. The Index Service filters and skips the first 1000 documents (OFFSET 1000), and returns the keys to the next 10 documents (LIMIT 10) to the Query Service
4. 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

Similarity between Couchbase N1QL to Oracle SQL

Language Capabilities	Couchbase N1QL	Oracle SQL

SELECT with Aggregation	`SELECT name, age, COUNT(name)`
	`FROM customers`
	`GROUP BY name, age`
		`SELECT name, age, COUNT(name)`
		`FROM customers`
		`GROUP BY name, age`

SELECT with Ordering	`SELECT id, name, age`
	`FROM customers`
	`ORDER BY age ASC, balance DESC`
		`SELECT id, name, age`
		`FROM customers`
		`ORDER BY age ASC, balance DESC`

SELECT with Pagination	`SELECT id, name, age`
	`FROM customers`
	`OFFSET 400 LIMIT 50`
		`SELECT id, name, age`
		`FROM customers`
		`OFFSET 400 ROWS`
		`FETCH NEXT 50 ROWS ONLY`

SELECT with JOINs	`SELECT id, name, age, balance`
	`FROM customers, accounts`
	`INNER | LEFT OUTER | RIGHT OUTER ...`
		`SELECT id, name, age, balance`
		`FROM customers, accounts`
		`INNER | LEFT OUTER | RIGHT OUTER |`
		`FULL OUTER JOIN ...`

INSERT	`INSERT INTO customers(KEY,VALUE)`
	`VALUES(' c123', {'id':'c123',`
	`'name':'Jan', 'age':23} )`
		`INSERT INTO customers(id, name, age)`
		`VALUES ('c123', 'Jan', 23)`

UPDATE	`UPDATE | UPSERT customers`
	`SET age = 32`
	`WHERE id = 'c123'`
		`UPDATE customers`
		`SET age = 32`
		`WHERE id = 'c123'`

DELETE	`DELETE`
	`FROM customers`
	`WHERE id = 'c123'`
		`DELETE`
		`FROM customers`
		`WHERE id = 'c123'`

GRANT	`GRANT query_select`
	`ON orders, customers`
	`TO bill, linda`
		`GRANT SELECT`
		`ON oe.customers_seq`
		`TO hr`

REVOKE	`REVOKE query_update`
	`ON customers`
	`FROM debby`
		`REVOKE UPDATE`
		`ON hr.employees`
		`FROM oe`

Summary of Couchbase N1QL versus MongoDB's API

Query Capabilities	Couchbase N1QL	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 - Join on scalars only. Cannot join on two sharded collections.
RIGHT OUTER JOIN	Yes	No
Sub-Query with JOINs in FROM Clause	Yes	No
Set Operators		
UNION	Yes	No
INTERSECT	Yes	No
EXCEPT	Yes	No

Query Optimization		
Hash JOINs	Yes	No
Predicate Pushdown to Index	Yes	No
Aggregate Pushdown to Index	Yes	No
Pagination Pushdown to Index	Yes	No
Covered Query	Yes	Does not support arrays or sharded collections
Functional Indexes	Yes, can create indexes on functions and expressions	Support indexes on built-in hash function only
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.

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.

Learn more

To learn more, contact your Couchbase sales representative today or visit:

couchbase.com | couchbase.com/downloads

Couchbase's mission is to be the data platform that revolutionizes digital innovation. To make this possible, Couchbase created the world's first Engagement Database to help deliver ever-richer and ever-more-personalized customer and employee experiences. Built with the most powerful NoSQL technology, the Couchbase Data Platform was architected on top of an open source foundation for the massively interactive enterprise. Our geo-distributed Engagement Database provides unmatched developer agility and manageability, as well as unparalleled performance at any scale, from any cloud to the edge.