A Comprehensive Performance Evaluation of Couchbase as a Distributed Datastore

Austin Hunt
Department of Computer Science
Vanderbilt University
Nashville, TN, USA
austin.j.hunt@vanderbilt.edu

Guoliang Ding
Department of Computer Science
Vanderbilt University
Nashville, TN, USA
guoliang.ding@vanderbilt.edu

***Abstract*—The increasing prevalence and complexity of data in the digital era has welcomed an innovative tsunami of NoSQL (Not Only Structured Query Language) database systems designed and built from the ground up to be fast, powerful, flexible and scalable to suit modern business needs. Among these systems lives an increasingly popular industry solution called Couchbase Server, an open source database software package for building and managing distributed, document-oriented, multi-model NoSQL databases optimized for modern interactive applications. In this paper, we provide a detailed quantitative evaluation of the performance of Couchbase Server along various metrics by leveraging AWS EC2 infrastructure in combination with a custom-built automated testing framework that utilizes Couchbase's CLI tools and its Python 3 SDK. In our evaluation, we include analyses of the relationships between data operation latencies and various aspects of cluster configuration available through Couchbase Server.**

***Index Terms*—Couchbase, NoSQL, distributed systems, performance analysis, AWS, Python**

I. Introduction

Born back in the 1970s in the era of mainframes and back-office business applications, traditional relational database management systems (RDBMS) are struggling to keep up with the requirements of the current digital era in which data grows continuously in both volume and complexity. Designed and engineered to run on a single server where the bigger, the better, an RDBMS is generally scaled vertically by adding more processors, memory and storage but both physics and nonlinear price increases often make vertical scaling impractical – sometimes even impossible. In addition, the strict consistency requirements of RDBMS make it difficult to scale them horizontally, and inflexible RDBMS data models, despite their provision of safety and consistency with data management, make changing the data model a difficult process, which is problematic in a world of continuous change. In the digital era, the ever-changing structure of data that needs to be stored challenges the idea of pre-planned, fixed schemas when building solutions that meet long term running requirements. Consequently, the huge amount of data generated from the internet, cell phones, social media, and the growing Internet of Things (IoT) demands a more modern, flexible, and scalable solution than is offered by traditional data stores.

Document database systems, which store information as documents in formats like JSON, XML, and BSON, have been developed to address the limitation of relational database systems and provide the flexibility, performance, and scalability required by the modern Internet and mobile applications we are now accustomed to using. From the mid-2000s to 2020, a steady rise in the adoption of document-oriented database technology by small startups, IT shops, and established Fortune 500 companies, is a consequence of its capacity to simplify data management by making it a flexible part of continuous application development [Mon21].

As an open-source, document-oriented, multi-model, distributed NoSQL database, Couchbase lives among the new wave of modern document database systems belonging to the NoSQL family. By offering strong consistency guarantees, various options for simple horizontal scaling, and impressively high performance, Couchbase currently stands as one of the most widely used NoSQL solutions, with big-name customers like Wells Fargo, Tesco, eBay, and PayPal. These companies use Couchbase to guarantee high service availability while managing massively high-throughput processes like 1) real-time fraud monitoring for over 50 million daily transactions (Wells Fargo), 2) catalog scaling and inventory management for millions of products (Tesco), 3) processing millions of user analytics updates every minute (PayPal), and supporting about 1.3 billion live e-commerse listings worldwide at any given moment (eBay) [Cou21c].

II. Overview of Couchbase

In this section, we'd like to introduce some key concepts underlying Couchbase Server and provide a foundation to understand the performance evaluation described in section IV.

A. Couchbase Server

Couchbase Server refers to the actual database software package. The package comes as two different versions: the paid Couchbase Server Enterprise version and the free Couchbase Server Community version available for download and evaluation from the Couchbase website. In this evaluation, Couchbase Server Community Version 7.0.0 is used.

B. Node

A Couchbase node is a physical or virtual machine that hosts a single instance of Couchbase Server [Cou21b]. A node can only have one single instance of Couchbase Sever running on it, and Couchbase keeps node management simple by providing only a single node type, which greatly helps with the installation, configuration, management and troubleshooting of the cluster as a whole. In this evaluation, we will use AWS EC2 instances to host the Couchbase Server and serve as the nodes.

C. Cluster

A cluster contains one or more nodes, which can be added or removed from a cluster on the fly without impacting the performance of the cluster. Much of this evaluation revolves around the automatic adjustment of cluster configuration (especially cluster size) to measure its relationship with latencies of data operations.

D. Bucket

A bucket can be roughly compared to a database in traditional RDBMS terms, but instead of housing a set of tables with columns and rows, a bucket simply houses a collection of documents, specifically in the JSON format. There are three types of buckets in Couchbase [Cou21b]:

- Standard, or "Couchbase", buckets, which are the default and store data both in memory and persistently
- Ephemeral buckets, which are designed for usage by applications that do not require data persistence
- Memcached buckets, which are now deprecated – these were originally designed for use alongside other database platforms, even RDBMS platforms, to serve a caching function

In this evaluation, the Standard Couchbase buckets are used and the performance difference of buckets types is currently beyond the scope.

E. vBucket

A vBucket, short for "virtual bucket", is part of Couchbase's larger system of auto-sharding, replicating, and distributing data across a cluster, or *clusters*. As discussed in subsection II-D, all data (that is, all documents, which are composed of key-value pairs of data) is housed in buckets. Every bucket B is broken into 1024 vBuckets, and Couchbase intelligently sprinkles those 1024 vBuckets evenly across all nodes in a cluster using the combination of a hash algorithm and a cluster map, which is part of the cluster configuration (not managed by the user, but by the software). In short, any data operation on some key K contained in a document in bucket B is mapped via a hash function to one of those 1024 vBuckets, and then the cluster map is used to map the resulting vBucket to one of the nodes. This ensures that every piece of data in the bucket is *active* only on one of the nodes determined by that cluster map, and allows automatic load balancing as well as consistency, since a write operation on key K will only commit if that node from the cluster map is available. [Cou21b].

F. Data Replication

Two forms of data replication are supported by Couchbase Server [Cou21b]:

- Intra-cluster replication: for each bucket in the cluster, up to 3 replicas can be configured. Each replica itself is also implemented as 1024 vBuckets, as discussed in subsection II-E. In general, only active buckets are accessed for read and write operations and the replica buckets will receive a continuous stream of mutations from the active bucket with Couchbase's Database Change Protocol (DCP), a high performance, low-level protocol for streaming replication of mutation operations across the cluster so that every node immediately reads what is written to one of the other nodes. This is significant in that it provides a valuable level of consistency from a distributed systems standpoint.
- Cross Data Center Replication (XDCR): in cases where multiple clusters are used, perhaps in two or more different data centers, this type of replication is used for replicating data either unidirectionally (from a source bucket B1 in cluster C1 to a target bucket B2 in cluster C2), or bidirectionally (buckets B1 and B2 simultaneously stream their replications to each other while using sequence numbers or timestamps for conflict resolution)

For this evaluation, the multi-cluster replication is beyond the scope of the study. During our testing, when we create the bucket, the number of replicas is set to 1. The impact of the number of replicas on the performance is not tested in this report but would be a good topic for future work.

G. Services

A service in Couchbase is an isolated set of processes dedicated to a particular task or set of tasks [Cou21b]. Couchbase Server breaks its functionality into a set of seven core services, only 4 of which were relevant to and are actively used in this evaluation:

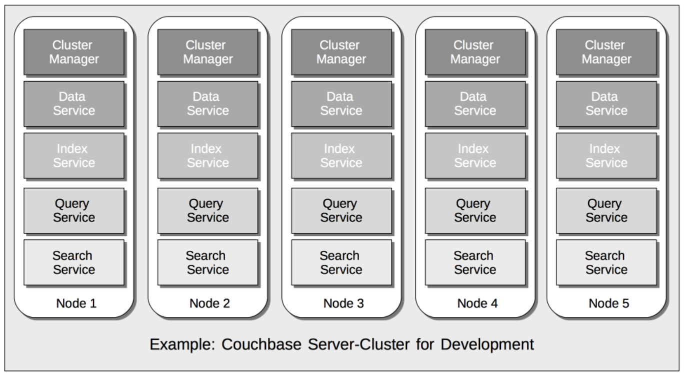
- The **data** service is used to store, set, and retrieve data items using their keys (the hashes of which map to specific vBuckets).
- The **query** service is the engine responsible for parsing N1QL queries, where N1QL is a SQL-like query language designed and optimized for Couchbase.
- The **index** service handles the creation of indexes for use by the query service; indexes can be created to improve query performance. This evaluation creates a default index for each bucket which is used when testing query operation latency.
- The **search** service is responsible for creating the indexes specifically for Full Text Search (FTS) and handles language-aware searching. The Full Text Search is one of the specific operation types whose performance is tested in this evaluation.

By design, these services can be deployed, maintained, and provisioned independently of one another (for the most part) using a framework that Couchbase aliases "multi-dimensional scaling". There are only a few exceptional requirements, namely 1) you must have the data service running on at least one node, and 2) certain services are interdependent and if used, require that their counterpart services also be running (e.g. query and index). A node, depending on its underlying hardware, may have one or all services running on it; this can be decided upon by the development team for the client application, which allows for optimal use of resources based on the known requirements of the application.

H. Architecture and Scaling

A Couchbase cluster can be architected in various ways – realistically infinite ways, actually, considering the combinatorics behind multidimensional scaling possibilities as cluster size increases. For development purposes, the simplest cluster architecture is a homogeneous one in which all nodes are running the same service quotas, as seen in Figure 1.

Fig. 1: Development Cluster Homogeneous Service Architecture [The picture is from [Cou21b]]



Corresponding to this cluster architecture is the homogeneous scaling methodology in which horizontal scaling simply entails adding one or more additional nodes that run the exact same set of services already running on each of the existing nodes. Figure 2 illustrates this scaling methodology clearly.

A more advanced cluster architecture that is more appropriate to production environments is the heterogeneous service architecture where different nodes will have different services running such that each node carries unique responsibilities. For example, for the more critical services like Data Service or Index Service, there can be several nodes with better and stronger configurations dedicated to these services only for optimized performance. Figure 3 represents a multidimensional service layout in a Couchbase cluster.

Corresponding to this architecture is the multi-dimensional scaling methodology, in which the addition of a new node to the cluster involves the intentional, calculated decision about specific services to be run on that new node for optimized application performance. This scaling approach is highlighted by figure 4.

In this evaluation, we are using the homogeneous service architecture where each node within the cluster runs an equal

Fig. 2: Homogeneous Scaling [The picture is from [YN19]]

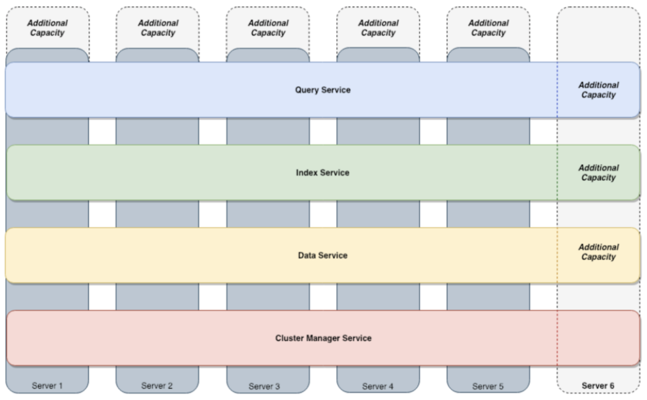
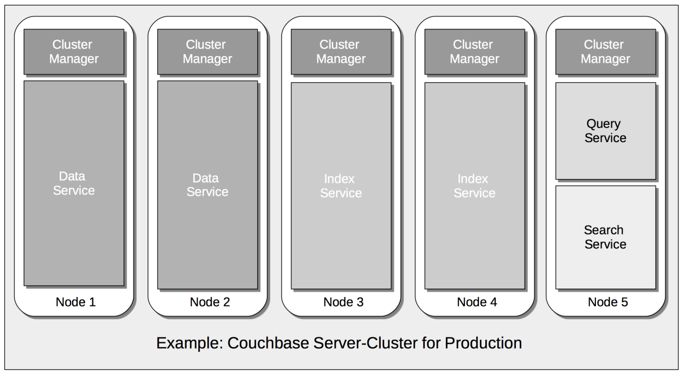


Fig. 3: Production Cluster Heterogeneous Service Architecture [The picture is from [Cou21b]]

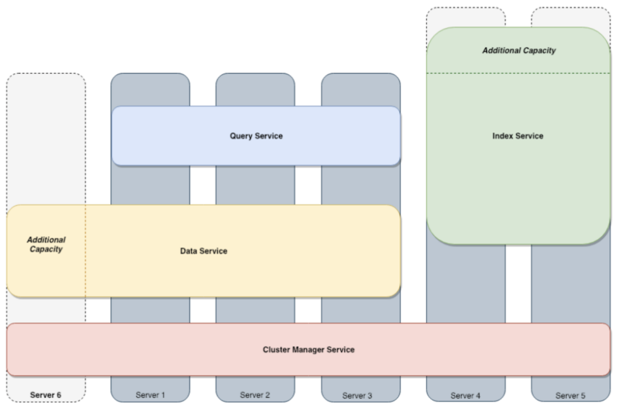


service quota. The work we present here still leaves room for additional tests on the impact of different service layouts (heterogeneous service architectures) on operation latencies. More details are provided in section V.

I. Durability

Regardless of whether a write operation is initiated from the Python 3 SDK or elsewhere, it is always first written to a single node – the *active* node for the data being operated upon, as discussed in subsection II-E. After that, Couchbase Server automatically and transparently handles replicating the mutation to any configured replicas (and to disk). For a given mutation operation (e.g. insert, update, and delete), one may optionally define specific durability requirements for the operation which tell Couchbase, essentially, how fully the data should be replicated throughout the cluster before committing the mutation; the higher the durability requirements (i.e. the higher the number of memory and/or disk locations to which data needs to be replicated before committing), the longer Couchbase Server will wait before that mutation commits. There are currently three available configurations for durability

Fig. 4: Multi-Dimensional Scaling [The picture is from [YN19]]



requirements that can be optionally passed when executing a mutation operation via the SDK [Cou21b]:

- The **majority** configuration tells Couchbase Server that the mutation should be replicated to memory on a majority of the cluster nodes before committing.
- The **majorityAndPersistToActive** configuration tells Couchbase Server that the *majority* requirement should be satisfied in addition to persisting the mutation to disk on the active node before committing the mutation.
- The **persistToMajority** configuration tells Couchbase Server not to commit a mutation operation until it has been written to disk on a majority of the cluster nodes (in addition to the memory requirements of the *majority* configuration).

As one transitions from the **majority** requirements up to the **persistToMajority** requirements, the system durability and its resilience to failure definitely increase, but there is no free lunch. Writing every change to disk is much slower than storing it in memory, and that fact is multiplied if you require writing each change to a majority of the nodes in a sizable cluster before committing; luckily, Couchbase enables the application developer to decide when to pay that performance price. In this evaluation, we will test the impact of these three options on the performance latency of various operation types.

J. CLI, SDK and Data Access

Couchbase provides a diverse arrangement of command-line interface (CLI) tools to handle a bulk of the management and monitoring of cluster infrastructure, from the cluster configuration itself down to the nodes and vBuckets. Couchbase also provides a multitude of language-specific software development kits (SDKs) that enable developers to easily integrate application code and logic with a Couchbase datastore.

Both the CLI [Cou21a] and Python SDK [Cou21d] are leveraged in this evaluation to develop an automated testing framework. Details are provided in subsection IV-A.

Even though Couchbase is considered a document database and is not exactly a key-value database (which is a separate type of NoSQL database), at the core of its flexible JSON-based data model is, in fact, a distributed key-value store, which is where much of its impressive performance derives from. This key-value store can be accessed and updated very quickly and simply using basic CRUD APIs available via the language-specific SDKs. By using the IDs of the JSON documents stored in a Couchbase bucket, the documents can be easily accessed with extremely low (sub-millisecond) latency [Cou21e]. The following is a high level view of some sample Python snippets executing basic CRUD operations against a Couchbase API [Cou21d] :

```python
# Create/Insert document
doc = {"foo": "bar", "bar": "foo"}
result = collection.insert("doc-key", doc)

# Read/Retrieve document
result = collection.get("doc-key")

# Update document
new_doc = {"foo": "tea", "tea": "foo"}
result = collection.upsert("doc-key", new_doc)

# Delete/Remove document
result = collection.remove("doc-key")
```

All these operations can be conducted with different operation settings and in this evaluation, the latency of these operations will be investigated and compared across different cluster settings and operation settings. More details about the test plan are provided in subsection IV-B.

III. Related Work

André Calçada1 and Jorge Bernardino, in their 2019 paper [CB19] presented a three-part evaluation of MongoDB, CouchDB, and Couchbase using the OSSpal methodology: a methodology key to a larger OSSpal project that has succeeded the Business Readiness Rating (BRR) with the goal of helping to find and evaluate free and open source software that meets modern business needs. The goal of their work was to provide a comparison between Couchbase and the other two datastores along weighted features such as stability, security, robustness, scalability, and others. Based on their qualitative and quantitative evaluation of Couchbase (following the OSSpal methodology guidelines), they ranked Couchbase second among the three, behind MongoDB and ahead of CouchDB – this is no surprise since Couchbase is a newer combination of the CouchDB and Membase datastores. In another 2019 paper by George Yildiz and Fredrik Wallström [YW19], Couchbase, juxtaposed with Google's tile-based mapping system, is evaluated as a tool to solve a specific problem: maintaining system scalability when sharing massive amounts of data between many mobile devices. They determined that Couchbase, together with Google's solution, solved the problem well, but the results of their tests were dependent on the test data. In our paper, we focus on exclusively evaluating Couchbase along various metrics via a custom-built Python testing framework, offering a different, narrower perspective on the distributed datastore than is presented in other papers.

IV. Evaluation

In this section, we present a detailed breakdown of our multi-tiered evaluation of Couchbase Server. We start with a description of the automated testing framework built to perform the evaluation, then we provide the test plan that guided us in developing the driving test algorithm within the framework. We finish the section with a detailed outline, with visuals, of the results obtained from the execution of the framework.

A. The Automated Testing Framework

The evaluation of Couchbase Server Community Version 7.0.0 that we present in this paper is driven entirely by an automated testing framework built with Python 3.8. The tests in the paper are run by the framework against a Couchbase Server cluster provisioned with AWS EC2 infrastructure – specifically five Ubuntu 20.04 "t2.medium" instances with public addresses enabled (in addition to private VPC addresses). The framework uses the combined power of the Couchbase Server Python 3 software development kit (SDK) and the Couchbase command line interface (CLI) to automatically manage, respectively, both 1) data operations and 2) cluster configuration. These two types of management are handled, respectively, with a Data Manager class and a Cluster Manager class, each written with Python. The Couchbase Server Python 3 SDK (version 3.1.3) allows the framework to perform data operations rapidly using a high-performance C library called 'libcouchbase' which communicates to the cluster over Couchbase's binary protocols. The Couchbase Server command line interface (CLI) is provided to manage and monitor clusters, servers, vBuckets, XDCR (cross datacenter replication), and so on; while the CLI was not designed specifically for use with Python, we choose to use Python to trigger the shell CLI commands such that all data and configuration operations are centralized in the Python framework.

While Couchbase Server does offer a genuinely easy-to-use web interface for managing data and cluster configuration (which is great for such management in production environments), we wanted to completely eliminate the need for manual interaction for the performance testing execution. The framework was built such that the only prerequisites to running it are 1) creating a set of virtual machines (preferably AWS EC2 instances) reachable at all of the officially documented Couchbase Server ports, and 2) providing the addresses of those machines in a **hosts.ini** configuration file at the root of the project. From there, the framework uses Ansible[HM17] to automatically install and start the Couchbase service on each available host, and then runs a series of automated Couchbase Server performance tests (as outlined in subsection IV-B) using those prepared hosts.

B. Test Plan

We produced our test plan with particular interest in identifying relationships between a few key variables, outlined below.

1) Operation Type: An important question when evaluating any data store, distributed or not, is "What does the performance look like for the main data operations?". With this paper, we dive into the latencies involved with both mutation and non-mutation operations, from document insertion, document updates, and document deletion, to N1QL "SELECT" queries and Full Text Search operations.

2) Durability configuration: Since one of the sought-after features of Couchbase among its large customers is the ability to tune the system durability based on application-specific needs, we take a look in this paper at how the various (three in total) durability configurations impact data operation latencies. Note that when we refer to "low", "medium", and "high" durability, we respectively mean the *majority*, *majority and persist active*, and *persist to majority* durability configurations offered by Couchbase Server, which define different requirements for mutation operations to be committed. In short, the higher the configured durability, the more persistent the data, and the greater the expected latency. Also note that this is particularly relevant when buckets of data are replicated; if data is not configured to replicate, then durability configurations are irrelevant. We only explore cases where replication is used.

3) Cluster Size: Perhaps the most obvious variable to analyze is the impact of cluster size on data operation latency. That is, as nodes are added or removed, how does the latency of each type of operation respond? Note that the maximum number of nodes for our testing is 5 due to AWS EC2 cost constraints. The testing framework runs tests against clusters with sizes 1 to 5, as seen in the pseudocode in this section.

4) Bucket Size: With every piece of data in Couchbase being stored in a bucket, we were interested in analyzing the relationship between the sizes of these buckets and the latencies of operations on the data that they store. We expected to see larger latencies for larger bucket sizes, particularly for the updating, querying, full text searching, and deleting operations that depend on finding something in that data. Note that when we refer to bucket sizes "small", "medium", and "large", this terminology refers to the number of documents stored in the bucket. Small, medium, and large buckets respectively hold 1000, 3000, and 5000 JSON documents generated with a custom-built Vanderbilt-University-themed random JSON data generator. Also note that all buckets, regardless of size, were allocated 256MB of RAM when initialized, and for clusters with more than one node, buckets were configured to use one replica (these are the cases where durability configuration is relevant).

The following pseudocode outlines, at a high level, the approach taken to address these relationships in the test execution.

Algorithm 1 Couchbase Evaluation Test Driver

```
1: for each $durability \in [low, medium, high]$ do
2:     for $clusterSize := 1...numHosts$ do
3:         setupCluster($clusterSize$) // homogeneous services
4:         for $bucketSize \in [small, medium, large]$ do
5:             $bucket$ = createBucket($bucketSize$) // homogeneous services
6:             flushBucket($bucket$) // clean slate
7:             runInserts($bucket$,...) // write latencies
8:             runN1QLSelects($bucket$,...) // write latencies
9:             runFTS($bucket$,...) // write latencies
10:            runUpdates($bucket$,...) // write latencies
11:            runDeletes($bucket$,...) // write latencies
12:            flushBucket($buckete$,...) // clean slate
13:        end for
14:    end for
15: end for
16: $generatePlots()$ // build plots with written data
```

C. Test Results

In this section we provide the results obtained from the execution of the testing framework, organized by the various relationships we were intending to analyze when starting the project as indicated in section IV-B.

1) Durability Configuration vs. Operation Latency: Here we present an analysis of the relationship between durability configuration (where one of three configurations are provided with Couchbase Server, which we alias "low", "medium", and "high" as discussed previously) and operation latency, broken down by the various operation types. Note that the only operations affected by durability configuration are mutation operations, so here we specifically focus on the insert and update operations, but we also include the other operation types for comparison. Table I shows the average latency in seconds for the various operation types for the low, medium and high durability configurations in a cluster of 5 nodes. From the table, we can see:

- The latency never exceeds three hundredths of a second, regardless of operation.
- The write operations, insert and update, exhibit consistently higher latencies.
- Only the write operations are significantly affected by the change in durability configuration; the insert operation is affected the most.
- It is unclear whether durability configuration affects the delete operation; deletion is a mutation operation, but the deletion latency does not consistently trend upward as durability requirements increase.

2) Cluster Size vs. Operation Latency: In this section we provide an analysis, again on a per-operation-type basis, of the relationship between Couchbase Server cluster size (ranging from one to five) and operation latency. Table II displays the average latency in seconds for each operation type, for each

	Durability Level (Cluster Size=5)		
Operation	low	medium	high
Write Operations			
insert	0.01035	0.02092	0.02232
update	0.01012	0.01350	0.01372
Non-Write Operations			
delete	0.00957	0.00983	0.00948
N1QL SELECT	3.2e-5	3.4e-5	3.2e-5
Full Text Search	7.6e-5	8.1e-5	8.1e-5

TABLE I: Durability level impact on write latencies (seconds) in a cluster of 5 nodes, compared with non-write operations which are unaffected

cluster size, using the lowest durability configuration for all tests to eliminate that as a variable. From this table, we can gather a few insights:

- The insert, update, and delete operations are fairly on par with each other with regard to latency, while the full text search and N1QL "SELECT" query operations are orders of magnitude faster (on the order of 10^{-5} seconds for each cluster size).
- The performance of the read operations (FTS and N1QL "SELECT") does not exhibit impact by cluster size changes.
- The mutation operations (delete, update, and insert) do exhibit a mostly positive correlation between their latencies and the cluster size, but
- The 3-node cluster seems to have caused the formation of a local maximum for the latencies of the insert, delete, update, and full text search operations, which contradicts a purely positive correlation

The upward trend in latencies for the mutation operations with increasing cluster size, in combination with the presence of the local maxima of operation latencies in the cluster of three nodes, reveals that while there is correlation between mutation operation latencies and cluster size, it is not purely positive. There are some points at which latencies do, in fact, drop when a new node is added. This raises curiosity about what this table would look like for a larger range of cluster sizes: at what thresholds would these local maxima appear and why? Figures 5, 6, and 7 are provided in addition to the table for the insert, update, and delete operations to further highlight the cluster size versus latency correlation as well as the local maxima discussed here.

	Cluster Size (low durability)				
Operation	1	2	3	4	5
delete	0.0075	0.0086	0.0092	0.0089	0.0096
update	0.00781	0.00800	0.00954	0.00914	0.0101
Full Text Search	7.9e-5	7.7e-5	8.3e-5	7.9e-5	7.7e-5
N1QL SELECT	3.5e-5	3.4e-5	3.3e-5	3.4e-5	3.2e-5
insert	0.0079	0.0083	0.0098	0.0095	0.0104

TABLE II: Cluster size impact on operation latencies (seconds) using "low" durability configuration for each cluster

3) Bucket Size vs. Operation Latency: Here, we take a look at the relationship between the latencies of the various

Fig. 5: Cluster size impact on latency of delete operation with durability = low; average exhibits upward trend and slight local maxima at cluster-size-3

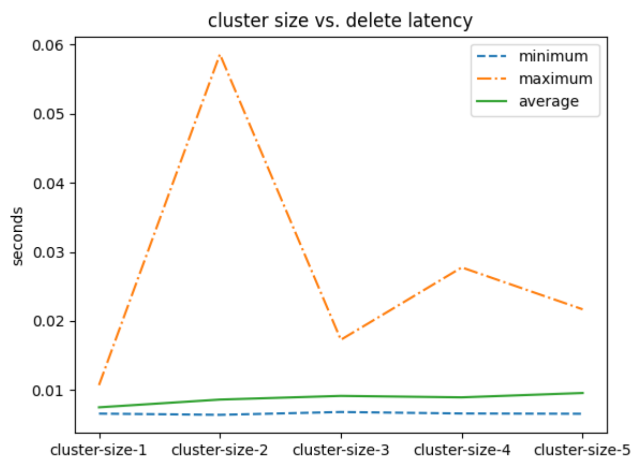
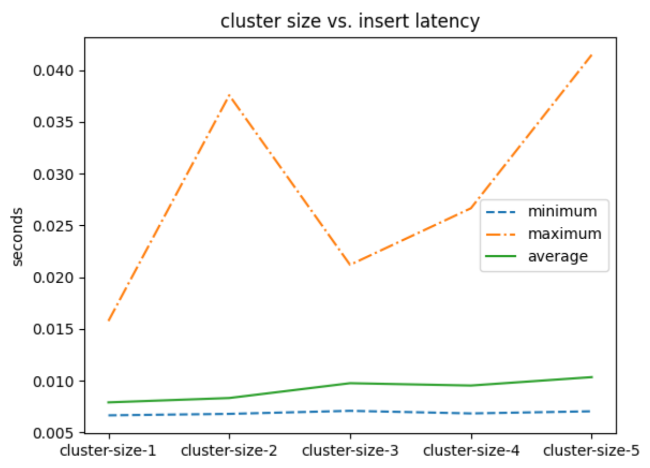
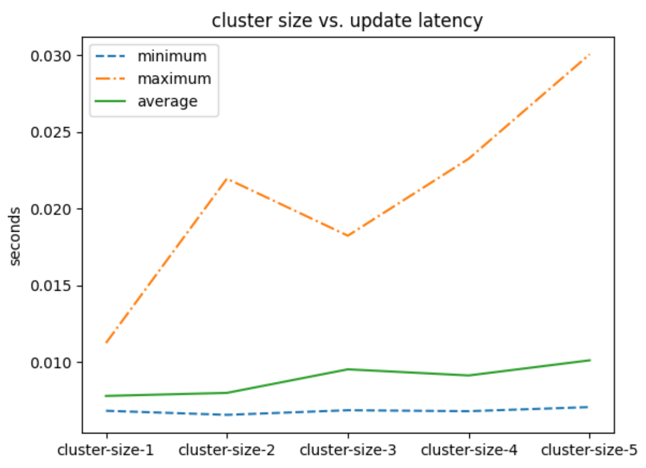


Fig. 6: Cluster size impact on latency of insert operation with durability = low; average exhibits upward trend and slight local maxima at cluster-size-3



operation types and the size of the buckets within a Couchbase cluster consisting of five nodes. Note that this bucket size relationship was automatically tested for all cluster sizes between one and five nodes but we felt the five-node cluster would offer the most valuable insights for this discussion. Table III displays the average latency for each operation type, for each of three main bucket sizes (small, medium, and large), within a cluster of five nodes configured for low durability. The small, medium, and large bucket sizes respectively correspond to buckets containing 1000, 3000, and 5000 documents. From this table, we can gather:

Fig. 7: Cluster size impact on latency of update operation with durability = low; average exhibits upward trend and noticeable local maxima at cluster-size-3



- There is generally no significant impact on average operation latency by a change in the number of documents in the “bucket”, or database.
- The only operations that exhibit a slight increase in latency as bucket size increases are the update and delete operations.

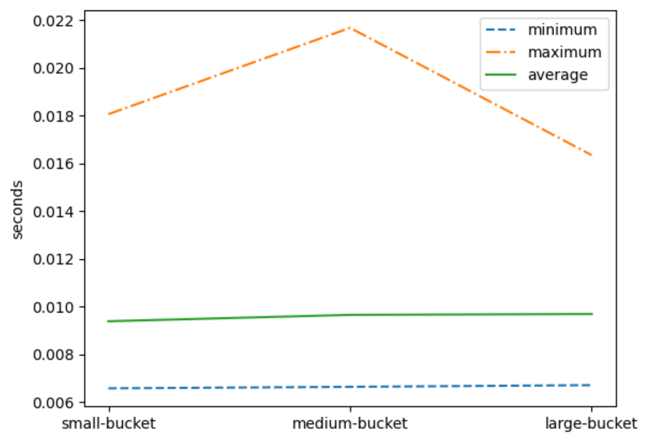
If we take into consideration that Couchbase provides atomicity at the single document level for mutation operations and that Couchbase uses a constant time hash function to map operations on a key K to a specific *active* node for that key, we can understand why the latencies for these operations are essentially static for each bucket size. First, since inserting a single document doesn’t depend on first finding anything in the bucket before inserting data, the insertion performance is independent of the number of documents already stored in the bucket; this is shown in the table, as there is no clear trend upward or downward for the insert operation latency. Moreover, since any update or delete operation on a document is mapped directly to the vBucket and the associated cluster node holding that document, the performance of these operations is also independent of the bucket size. As an additional note, Couchbase’s white paper [Cou21e] describes its underlying key-value implementation as similar to a hash table, where constant operation time is expected regardless of the bucket size. Table III, supplemented with Figures 8, 9, and 10, provides a visualization of the general lack of correlation between operation latency and bucket size.

4) Operation Type vs. Operation Latency: Here, we use Table IV to present a higher-level view of the relationship between latency and the type of operation being executed. We acknowledge that this relationship may be different depending on other variables such as cluster size or service layout but here we simply discuss averages for each type of operation,

	Bucket Size (low durability, 5 node cluster)		
Operation	small (n=1000)	medium (n=3000)	large(n=5000)
delete	0.0094	0.0097	0.0097
update	0.0100	0.0101	0.0103
Full Text Search	7.8e-5	7.5e-5	7.7e-5
N1QL SELECT	3.2e-5	3.1e-5	3.2e-5
insert	0.0106	0.0101	0.0103

TABLE III: Bucket size impact on operation latencies (seconds) using "low" durability configuration and a cluster of 5 nodes

Fig. 8: Bucket size impact on delete latency (durability=low, cluster size = 5)



where the averages encapsulate the latency records for all durability configurations, all cluster sizes, and all bucket sizes to emphasize the operation type versus operation latency relationship. From this table, we can obtain some key insights:

- The most obvious item is that the query operations (full text search and N1QL SELECT queries) are significantly faster than the mutation operations, as one would expect
- The operations ranked lowest to highest by average latency are N1QL SELECT, FTS, delete, update, and insert
- The latency ranges (expressed as maximum divided by minimum) for the delete, update, FTS, N1QL SELECT, and insert operations, respectively, are: 38.875, 12.22, 27.94, 7.4, 137.08
- The significant range in the latencies for each operation is as expected since this table includes data from the lowest durability level, lowest cluster size, and lowest bucket size, as well as from the highest of each. The difference between those ranges, however, further reveal that certain operations like insert are impacted more heavily by cluster configuration changes than other operations.

5) Durability and Cluster Size vs. Operation Latency: In this section, we compare the insert operation latency across the dimension of durability and cluster size for large (5000

Fig. 9: Bucket size impact on insert latency (durability=low, cluster size = 5)

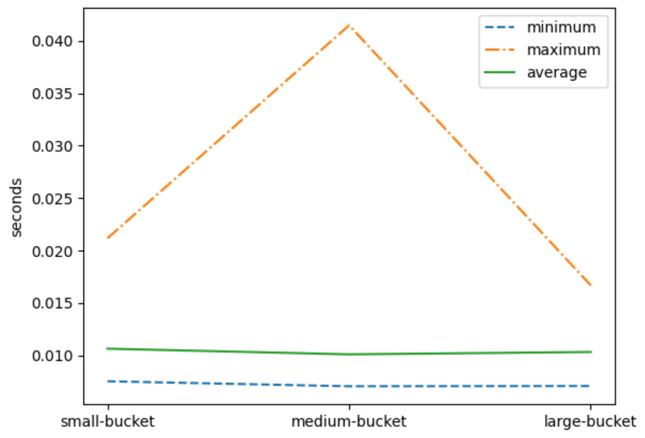
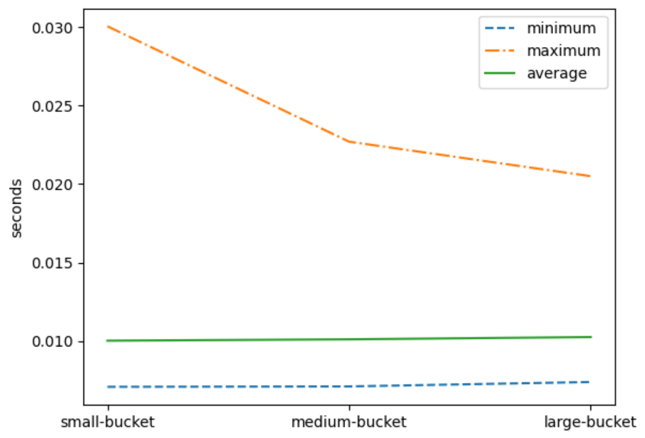


Fig. 10: Bucket size impact on update latency (durability=low, cluster size = 5)



	Operation				
Measure	delete	update	FTS	N1QL SELECT	insert
min	0.0064	0.0066	6.8e-5	2.7e-5	0.0067
max	0.2488	0.0807	0.0019	0.0002	0.9185
avg	0.0098	0.0124	7.9e-5	3.3e-5	0.0182

TABLE IV: Comparison of min, max, and average latencies between operation types across all durability levels, all cluster sizes, all bucket sizes

documents) buckets. Table V displays the results. In general, for a given cluster size, increasing the durability increases the document insertion latency, which is similar to what we have observed in Section IV-C1. However, when analyzing the impact of cluster size within a specific durability level, the results fluctuate. For low durability, the latency increases as the cluster size increases. However, for medium or high durability, a cluster of size one results in the highest latency while the minimum latency occurs with a larger cluster size; for medium durability the minimum insertion latency occurs in a cluster of five nodes while for high durability it occurs in a cluster of three nodes. The result seems counter-intuitive initially. However, after checking the Couchbase documentation on durability, the reason seems to lie within Couchbase's definition of **majority** and the implementation of vBuckets. With medium and high durability configurations, the mutation must be written to disk and this is monitored on the vBucket level. When there are multiple nodes, the vBucket is automatically balanced across all the nodes, and several nodes can write the mutation replication in parallel and quickly reach the criteria set for majority. However, when there is only one node in the cluster, it seems that all the changes must be written to the disk in sequence, which results in a longer execution time and a higher latency. The update operation is also tested, and similar patterns and observations are made.

	Durability		
Cluster size	low	medium	high
1	0.0081	0.0291	0.0297
2	0.0087	0.0212	0.0218
3	0.0095	0.022	0.0202
4	0.0093	0.0255	0.0206
5	0.0103	0.0196	0.0245

TABLE V: The impact of Durability and Cluster Size on average insert operation latency (seconds) with large bucket (n=5000)

V. Unresolved Problems & Future Work

While we were able to capture a number of key metrics in this evaluation of Couchbase, we have a couple of items we were hoping to include in the evaluation but were unable to due to time constraints. We outline them below.

1) Service Layout vs. Operation Latency: Since one of the key advantageous features of Couchbase is the ability to scale services independently ("multidimensional scaling"), we hoped to include an evaluation of how different service layouts impacted the latencies of different operation types. For example, in a five node cluster, how does running *only* the query service on the first two nodes compare to running *all* services homogeneously across all nodes when looking at N1QL SELECT query latencies? The expected result would be to see the query latency decrease if the query service is scaled independently higher than the other services in the cluster. Our repository does currently include the foundation for this service layout impact evaluation but it is not fully implemented or actively used yet.

2) Document Size Impact On Operation Latency: While we did analyze the relationship between Couchbase Server performance and bucket size (and found a general lack of correlation for the operations studied), we did not run tests using the size of the JSON documents as an independent variable. It is likely, especially considering that Couchbase Server emphasizes the provision of atomicity, consistency, and isolation at the single document level, that varying the document size (i.e. the number of key-value pairs in a given document) is much more likely to affect operation latency than changing the bucket size, particularly for mutation operations like inserting and updating, which have to fully write new documents before committing mutations. Our framework already uses document size as a variable and thus has the capacity to run this kind of test, but currently all tests simply pass a default value of 25 as the document size argument.

3) Replication Count Impact On Operation Latency: In this evaluation, the number of replication for the bucket is set as one. With a different number of replication, it is expected that the latency on different operations will be different since all the mutations will be streamed to the replicas. Coupled with different durability settings, the impact could be even more substantial. For example, for a cluster of a given size, with the number of replications changing from one to three, how will the latency change when the durability setting is at low, medium and high?

4) Node Failover Impact on Operation Latency: Another item we did not quite address in the evaluation is the impact of node failover on the latency of the different operation types. More specifically, we wanted to generate a line graph (for each operation type) of the operation latencies (for, say, 100 operations) over time, during which a node was "gracefully" failed over (one of the two main node failover types offered by Couchbase Server, the other being "hard") while those operations were executing. The expected result would be a spike in the latency at the time of node failover. However, a key consideration with a test like this is that due to the vBucket partitioning within Couchbase as discussed in section II, an operation will only be affected if the node being failed over holds the active data on which the that operation is executing. That is, if a data operation is executed on some key K, the hash of that key points to a vBucket which belongs to some node N based on the cluster map, and the latency of that operation should only be affected if N is the one being intentionally failed over. Since our framework already includes both a Cluster Manager component and a Data Manager component as described in subsection IV-A, it currently offers the capacity to run this kind of test, but it has not yet been implemented.

VI. Conclusions and Expected Results

With this project, our goal was to create an automated test framework that would allow us to provide a deep-dive quantitative evaluation of Couchbase Server along various metrics relevant to a larger distributed systems context. While there are a few key evaluation metrics we were unable to fully include, the framework nonetheless offers the capacity to pursue additional research questions outlined in section V, and has been valuable in generating many insightful statistics highlighting the relationships we set out to understand. We would like to emphasize that AWS cloud infrastructure through the EC2 service and Couchbase Server's default utilities for data and cluster management played critical roles in this evaluation.

References

[HM17] Lorin Hochstein and Rene Moser. *Ansible: Up and Running: Automating configuration management and deployment the easy way.* " O'Reilly Media, Inc.", 2017.

[CB19] André Calçada and Jorge Bernardino. "Evaluation of Couchbase, CouchDB and MongoDB using OSSpal." In: *KDIR*. 2019, pp. 427–433.

[YW19] George Yildiz and Fredrik Wallström. *Evaluation of Couchbase As a Tool to Solve a Scalability Problem with Shared Geographical Objects*. 2019.

[YN19] Artsiom Yudovin and Yauheniya Novikova. *Technical NoSQL Comparison Report:Couchbase Server v6.0, MongoDB v4.0, and Cassandra v6.7 (DataStax)*. 2019.

[Cou21a] Couchbase. *Couchbase CLI Reference*. https://docs.couchbase.com/server/current/cli/cli-intro.html. 2021.

[Cou21b] Couchbase. *Couchbase Documentation*. https://docs.couchbase.com/server/current/introduction/intro.html. 2021.

[Cou21c] Couchbase. *Couchbase Featured Customers*. https://www.couchbase.com/customers. 2021.

[Cou21d] Couchbase. *Couchbase Python SDK Reference*. https://docs.couchbase.com/python-sdk/current/hello-world/start-using-sdk.html. 2021.

[Cou21e] Couchbase. *Couchbase Under the Hood: An Architectural Overview*. 2021.

[Mon21] MongoDB. *When to Use a NoSQL Database*. https://www.mongodb.com/nosql-explained/when-to-use-nosql. 2021.