A Multi-Phase Tail Latency Analysis of Couchbase Server Using Automation and the Yahoo! Cloud Serving Benchmark

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*—As the production, storage, and analysis of data become increasingly integral to the success of modern businesses, and as data becomes increasingly kaleidoscopic due to the influx of new data sources, NoSQL (Not Only Structured Query Language) database systems are becoming a popular choice for data management because of their ability to quickly and scalably handle large volumes of structured, semi-structured, and unstructured data in a way that is friendly to developers, and more broadly, to change. One of the solutions in this arena is 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, two-phase tail latency analysis of Couchbase Server Community Edition v7.0 using a custom automated testing framework built with the Couchbase CLI and Couchbase Python 3 SDK in combination with the Yahoo! Cloud Serving Benchmark (YCSB), a robust and open source framework for analyzing the performance of different "key-value" and "cloud" serving stores like Couchbase. We analyze and report on relationships between tail latencies and dataset sizes, request distributions, ratios between various operation types (read/insert/update), and cluster architecture (homogeneous versus heterogeneous service layouts).**

***Index Terms*—Couchbase, NoSQL, YCSB, Tail Latencies, AWS, Python**

I. Introduction

Traditional relational database management systems (RDBMS), still lingering from their introduction in the 1970s era of mainframes and back-office business applications, 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.

A. A New Wave of 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].

B. Couchbase

Couchbase, an open-source, document-oriented, multi-model, distributed NoSQL database, 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 with latencies on the order of microseconds, Couchbase currently stands as one of the most widely used NoSQL solutions and is used by companies 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-commerce listings worldwide at any given moment (eBay). [Cou21c].

C. The Yahoo! Cloud Serving Benchmark (YCSB)

This innovative tsunami of new database technologies carries with it the need for testing – more specifically, the need for a method of comparing those technologies in a standardized, “apples-to-apples” kind of way. After all, there is not much advantage to a new wave of technologies if there’s no clear way of deciding which one to use for a given application. In 2010, the Yahoo! Cloud Serving Benchmark (YCSB) was introduced with the stated goal of meeting that need [Coo+10]. This project wraps standardized performance comparisons of many of these new generation cloud data serving systems into a simple framework that allows you to configure and execute specific workloads against specific target databases. The YCSB framework’s extensible design supports easy definition of custom workloads and easy extension to new systems [Coo+10]. Currently, the framework supports targeted performance testing against nearly 50 data serving systems, including but not limited to MongoDB, AWS DynamoDB, Cassandra, Redis, CouchDB, Google BigTable, and most importantly for this project, Couchbase.

D. Why Tail Latency?

For this research, we focus much of our analysis and reporting on the relationship between tail latency, or *high-percentile latency*, and various configurations of Couchbase (to be described in more detail). Tail latencies, while rare by definition, stand to have a greater impact on business relationships with customers in realistic application environments than averages and can greatly influence the user experience, thus it is important to understand those tail latencies so they can be reduced or avoided.

II. Overview of Couchbase

In this section, we’d like to introduce some key concepts underlying Couchbase Server and provide a foundation for the performance evaluation 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 Edition available for download and evaluation from the Couchbase website. In this evaluation, Couchbase Server Community Edition 7.0.0 is used. Couchbase also provides a CLI and a Python 3 software development kit (SDK) which we make heavy use of for the automation of testing.

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. More specifically, we use a set of five t2.xlarge instances, each of which come with 2 virtual CPUs, 4 gigabytes of RAM, 8 gigabytes of disk space, and low-to-moderate network performance.

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 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 bucket types is currently beyond the scope.

E. 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:

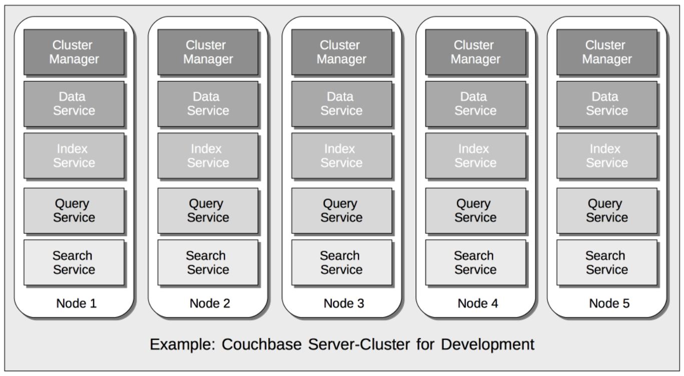
- The **data** service is used to store, set, and retrieve data items using their keys (the hashes of which map to specific vBuckets).
- The **query** service is the engine responsible for parsing N1QL queries, where N1QL is a SQL-like query language designed and optimized for Couchbase.
- The **index** 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 documentation 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 running on a node, require that their counterpart services also be running on that same node (e.g., query and index). A node, depending on its underlying hardware, may be configured to 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. Here, it is important to note that we are using the Community Edition of Couchbase for our evaluation, and this edition comes with some tighter constraints on service layouts that are outlined in more detail in subsection IV-C.

F. Architecture and Scaling

Because of the logical separation of services and the ability to assign services to different hosts, a Couchbase cluster can be architected in various ways – realistically many 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, in certain cases, performance may be optimized by dedicating several high-resource nodes to more critical services like the data service or index service. Figure 3 represents a multidimensional service layout in a Couchbase cluster.

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

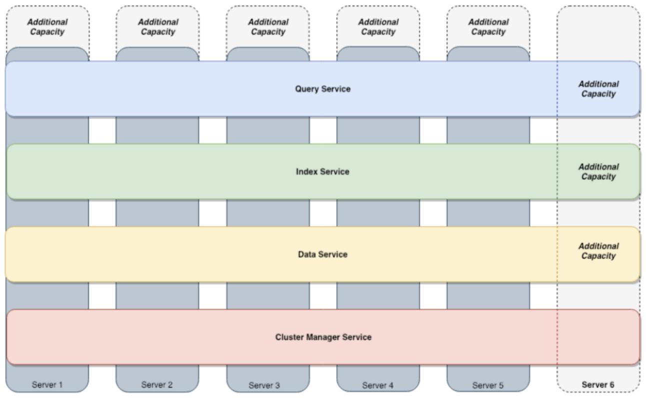
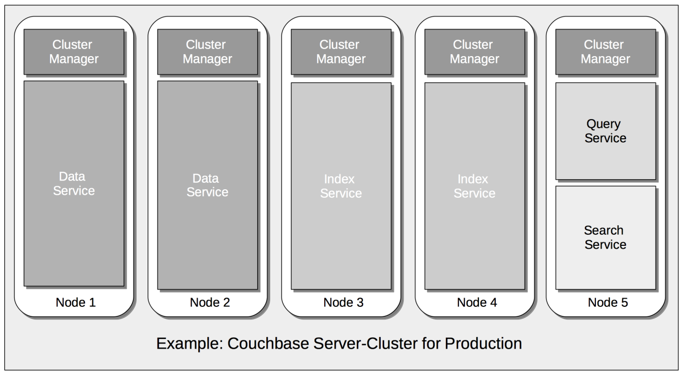


Fig. 3: Production Cluster Heterogeneous Service Architecture [The picture is from [Cou21b]]

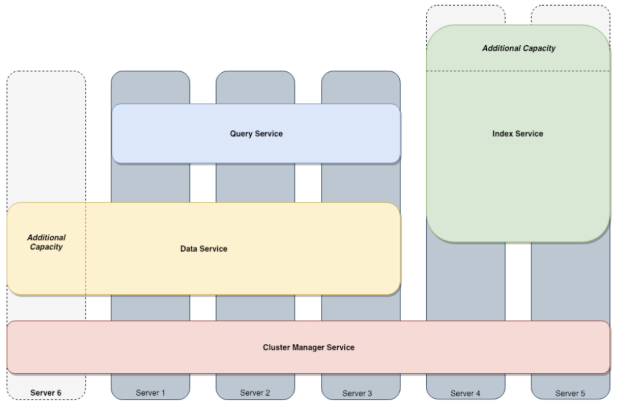


Corresponding to this architecture is the multi-dimensional scaling methodology, in which the addition of a new node to the cluster involves the intentional, calculated decision about specific services to be run on that new node for optimized application performance. This scaling approach is highlighted by Figure 4. Subsection IV-C is dedicated to an analysis of the impact of multidimensional service scaling on tail latencies of different operations.

G. 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 section III.

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



III. The Automated Testing Framework

The evaluation of Couchbase Server Community Edition 7.0.0 that we present in this paper is driven entirely by an automated testing framework built with Python 3.8. The tests on which we report are run by the framework against a Couchbase Server cluster running on AWS EC2 infrastructure – specifically five Ubuntu 20.04 “t2.xlarge” instances with public addresses enabled. 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 Couch- base’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 wrapped and abstracted in the Python framework.

A. Execution Overview

While Couchbase Server does offer a genuinely easy-to-use web interface for managing data and cluster configuration, we wanted to eliminate the need for manual interaction in this performance evaluation. The framework was built such that the only prerequisite to running it is providing your AWS access key ID and secret access key in an environment file, and then “provisioning” (with one command) a Vagrant virtual machine with an Ansible[HM17] master playbook.

The following outlines the primary steps executed when the framework is triggered to run:

1) A local virtual machine is spun up with Vagrant, and is provisioned with an Ansible[HM17] master playbook
2) The master playbook automatically provisions five Ubuntu 20.04 “t2.xlarge” EC2 instances with public addresses enabled (in addition to private VPC addresses) and installs the Couchbase Server software on each of the instances.
3) After the EC2 instances are provisioned, the playbook kicks off a framework driver script
4) Depending on arguments passed to the driver, the driver uses some combination of the DataManager, the Cluster-Manager, and the YCSB CLI to automatically configure various clusters and run workloads against them (as it does this, it also writes latency data for each operation to an output file)
5) After the workloads are finished running, the driver uses the output data files to generate plots, many of which are included in this paper

This framework’s modular architecture will enable us to extend it to other testing conditions with minimal effort, some of which are discussed in section V.

IV. Evaluation

In this evaluation, we are interested in evaluating the impact of database size (controlled by record counts, field sizes, field counts), operation type ratios (e.g., read/insert/update ratios), request distributions, and cluster architecture (homogeneous versus heterogeneous service layouts) on the tail latency of key data operations on a Couchbase cluster of five nodes. The testing is separated into the following two phases:

- In phase one, we use a fixed cluster size of five nodes with a homogeneous service architecture. With these variables holding as constants, we use YCSB to analyze the impact (on tail latency, specifically) of other variables that are easy to tune via YCSB such as record (document) counts, field sizes, field counts (document sizes), request distributions (explained in subsection IV-A2), and ratios between different operation types (i.e., read vs. write).
- In phase two, we again use a fixed cluster size of five nodes, but this time with focus on heterogeneous service architectures, with service layout and operation type acting as independent variables such that we can understand how the tail latency of each operation type changes as certain services are scaled horizontally (within the constraints of Couchbase Community Edition 7.0).

A. Phase One Test Plan: Using the YCSB Framework

As briefly mentioned in subsection I-C, the YCSB framework supports a wide range of workloads, which can either be defined ahead of time in static configuration files or at runtime by passing arguments to the YCSB executable, which provides a lot of flexibility through various parameters. In this project, the following parameters are picked to investigate their impact on the tail latency.

1) Database Size: The YCSB framework is generally separated into three steps on the client side [Coo+10]:

- The Load phase, where the data are loaded into the database system
- The Run phase, where workloads are issued to the database system
- The Report phase, where the performance statistics are printed to the screen or output to a designated location

During the load phase, the size of the documents and the number of documents can be controlled to determine the overall size of the database. By default, each document has 10 fields and each field is 100 bytes long, making each document 1KB. In this evaluation, we tune the values of the following sizing parameters to gauge their impact on the tail latency of different operations:

- Record count; this is the number of documents inserted into the database. We test values of 1K, 10K, and 100K for this parameter.
- Field count; this is the number of fields, or keys, in each document. We test values of 10 and 500 for this parameter.
- Field length; this is the size, in bytes, of each field in each document. We test values of 10 and 100 for this parameter.

Table I depicts the overall database sizes resulting from the various combinations of the above values. Note that for the YCSB tests, we use a constant memory quota of 256MB for our test bucket. This means that Couchbase will need to optimize performance by ejecting data from memory to disk based on its least recently used algorithm for four of the database sizes in the table (500MB, 100MB, 500MB, and 5GB) since they exceed the 256MB memory quota.

Database Sizes			
Record Count	Field Count	Field Length	Size
1000	10	10B	100KB
1000	10	100B	1MB
1000	500	10B	5MB
1000	500	100B	50MB
10000	10	10B	1MB
10000	10	100B	10MB
10000	500	10B	50MB
10000	500	100B	500MB
100000	10	10B	10MB
100000	10	100B	100MB
100000	500	10B	500MB
100000	500	100B	5GB

TABLE I: Database sizes as the products of record count, field count, and field length

2) Request Distribution: During the run phase, the YCSB framework randomly chooses from the documents that were previously loaded during the load phase to conduct read or write operations [Coo+10]. This decision is controlled by random distributions and YCSB has the following built-in distributions that will be tested:

- Uniform, where all records in the database are equally likely to be chosen.
- Zipfian, where some records are extremely like to be chosen (also known as the head of the distribution) and the probability rapidly decreases from these records (also known as the tail of the distribution).
- Hotspot, where some percentage of the data items are accessed by some percentage of the operations

3) Read/Write Proportions: During the run phase, the YCSB framework must decide which operation to perform when, as constrained by user-defined arguments for "proportions" of the different operations. In this test, we define nine different operation proportions and analyze them in conjunction with the other discussed variables in order to study how, or if, ratios between operation types impacts overall tail latency of a Couchbase system.

- Read-heavy
  - 100% read
  - 90% read, 10% insert
  - 90% read, 10% update
- Write heavy
  - 100% insert
  - 100% update
  - 10% read, 90% insert
  - 10% read, 90% update
- Equal read-write
  - 50% read, 50% insert
  - 50% read, 50% update

4) The Test Algorithm: The test automation is structured as a nested loop, where each level of the loop represents one of the parameters.

The following pseudocode outlines, at a high level, the testing approach for this phase.

Algorithm 1 YCSB Test Automation Driver

```
1:  createHomogeneousCluster($nodeCount$=5)
2:  $opProps$ = [list of r/u/i proportions]
3:  $requestDists$ = [uniform, zipfian, hotspot]
4:  $recordCounts$ = [1K, 10K, 100K]
5:  $fieldCounts$ = [10, 500]
6:  $fieldLengths$ = [10, 100]
7:  for each $rc \in recordCounts$ do
8:      for each $fc \in fieldCounts$ do
9:          for each $fl \in fieldLengths$ do
10:             for each $rd \in requestDists$ do
11:                 for each $prop \in opProps$ do
12:                     $bucket$ = createBucket($bucketSize$)
13:                     ycsb($rc$, $fc$, $fl$, $rd$, $prop$, $bucket$)
14:                 end for
15:             end for
16:         end for
17:     end for
18: end for
19: $generatePlots()$ // build plots with written data
```

B. Phase One Test Results

In this section we provide the results obtained from the execution of the testing framework in phase two, organized by the various relationships we were intending to analyze.

1) Field Count vs. Tail Latency: As seen in Figure 5, the number of fields in each document in the Couchbase database ultimately did not really impact the tail latency of the insert operation - the same is true for the update operation. The reason for this is that when testing with YCSB, we used a constant value of False for both the *persistTo* and *replicateTo* parameters, such that YCSB would not enforce any strict data durability requirements during the write operations. That is, since we were not requiring each write operation to replicate or persist data across the 5-node cluster, our writes were fast regardless of the number of fields. However, as seen in Figure 6, there does appear to be a positive correlation between field count and read latency. This indeed makes sense simply because larger documents imply greater usage of memory, which in turn implies greater likelihood of ejecting data to disk. Couchbase's impressive performance largely comes from its automatic management of a "caching layer", which comes down to keeping as much data in memory as possible with minimal disk interactions, where the memory quota can tuned at the bucket level; we tuned our test bucket with a memory quota of 256MB. We can see from Table I that the largest database size for a field count of 10 is 100MB, which is less than 50% of our bucket memory quota, which means there should have been no need for ejection. However, for a field count of 500, the largest overall database size is 5GB, which far exceeds our memory quota for the bucket, guaranteeing ejection of some data to disk. Thus, many of the read operations on documents with a field count of 500 had to go all the way to disk to get data that wasn't available in memory, resulting in higher tail latencies.

Fig. 5: Field count impact on insert tail latency in Couchbase

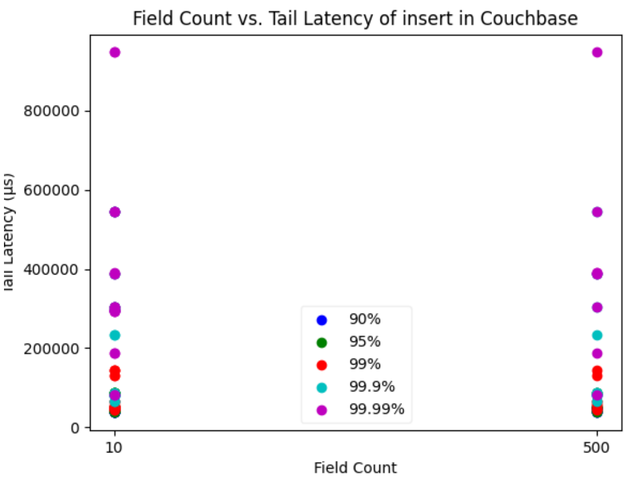
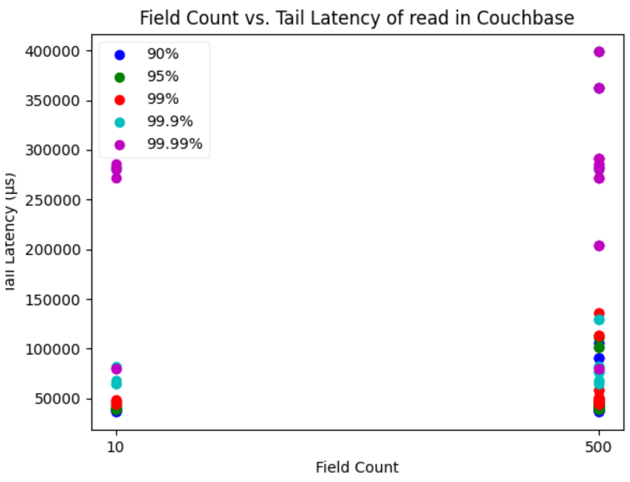
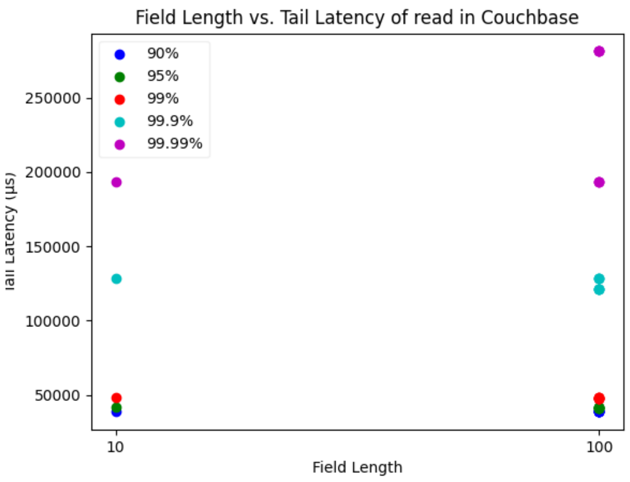


Fig. 6: Field count impact on read tail latency in Couchbase



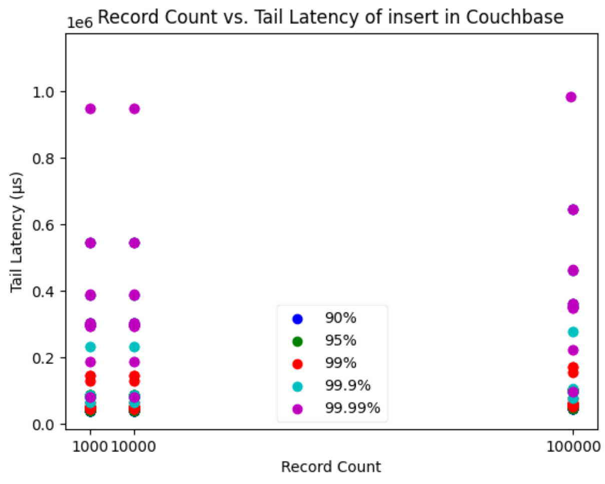
2) Field Length vs. Tail Latency: As was the case in subsection IV-B1, field length did not impact the tail latencies of the write operations because without durability requirements of replication and persistence, the speed of write operations are largely indifferent to document size variations. As seen in Figure 7, the read operation tail latency was heavily impacted by the field length. Of course, field length is yet another way of changing the document size, and thus the overall database size, so it is guaranteed to have a positive correlation with the likelihood of ejecting data to disk due to memory quota consumption, which implies a positive correlation with read latency.

Fig. 7: Field length (bytes) impact on read tail latency in Couchbase



3) Record Count vs. Tail Latency: Similar to the case in subsections IV-B1 and IV-B2, the tail latency of the write operations (update and insert) were not significantly impacted by modifying the record (document) count, for the same reason: changing the database size does not change the durability requirements enforced in write operations. This lack of correlation is depicted in Figures 8 and 9. The tail latency of the read operation, however, was significantly positively correlated with the increasing record count. As the number of documents in the database grows, so does the usage of the available memory quota, which increases the likelihood of ejecting data to disk; because of this, a greater proportion of read operations must go all the way to disk to read data since it is no longer in memory. The natural result is higher tail latencies for the read operation with a larger number of stored documents.

Fig. 8: Document/Record count impact on insert tail latency in Couchbase



4) Request Distribution vs. Tail Latency: While figures 11, 12, and 13 hint that there's not a significant relationship between request distribution and tail latency regardless of operation type, it is worth noting that YCSB's "zipfian" request distribution does seem to offer around 40% lower latency for the update operation specifically. With the zipfian request distribution, some data items (called a "hotspot") have a greater probability to be targeted by operations than other items. We can make sense of the lower 99.99th percentile for zipfian if we consider that a random "hotspot" could, by chance, align with the "active set" – that is, the set of data items currently in the Couchbase Cluster memory as opposed to disk. In other words, zipfian could randomly choose the ideal set of data items to operate on such that few or none of them have been ejected to disk.

5) Operation Proportion vs. Tail Latency: This particular test yielded results that are both interesting and perhaps the least insightful. As shown in Figures 14 and 15 in which the X axis labels are formatted as *read ratio - update ratio - insert ratio*, the average 95th and 99th percentile latency across all

Fig. 9: Document/Record count impact on update tail latency in Couchbase

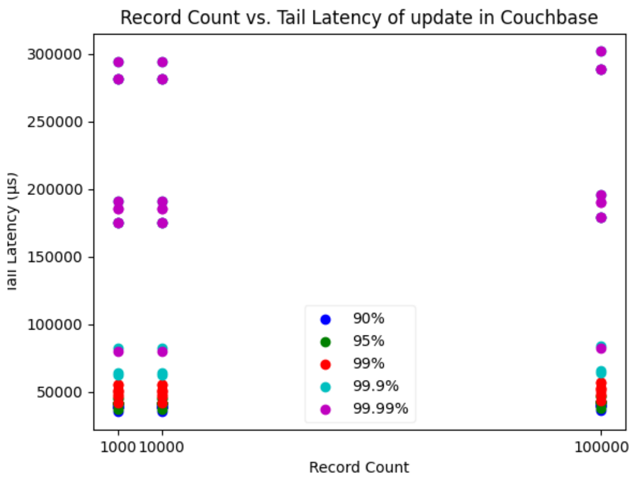
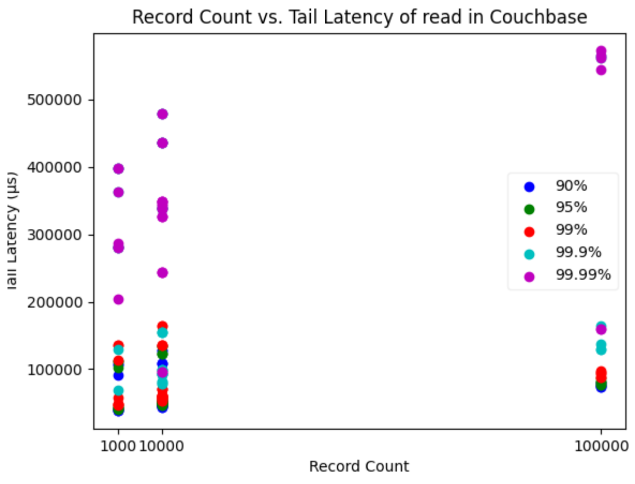


Fig. 10: Document/Record count impact on read tail latency in Couchbase



operations appear disconnected from whether a workload is read-intensive or write intensive. First, the local minima appear at 90% read-10% insert, 10% read-90% insert, and 100% read; thus, it cannot be concluded from this data that either a read or write-intensive workload provides lower overall tail latencies for all operations. Moreover, the local maxima appear at 100% update, 50% read-50% insert, 50% read-50% update, and 90% read-10% insert; thus it cannot be concluded from the maxima that either a read or write-intensive workload leads to higher overall tail latencies for all operations.

C. Phase Two Test Plan: Heterogeneous Service Scaling

In this section we provide our phase two approach taken to test the effect of multidimensional scaling on the tail latency

Fig. 11: Request distribution impact on insert tail latency in Couchbase

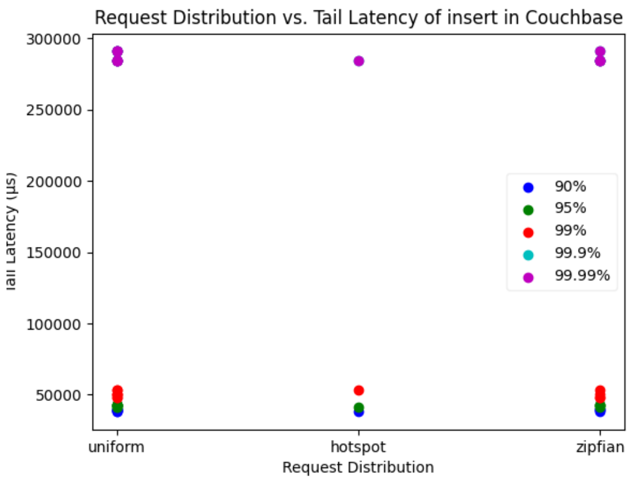


Fig. 12: Request distribution impact on update tail latency in Couchbase

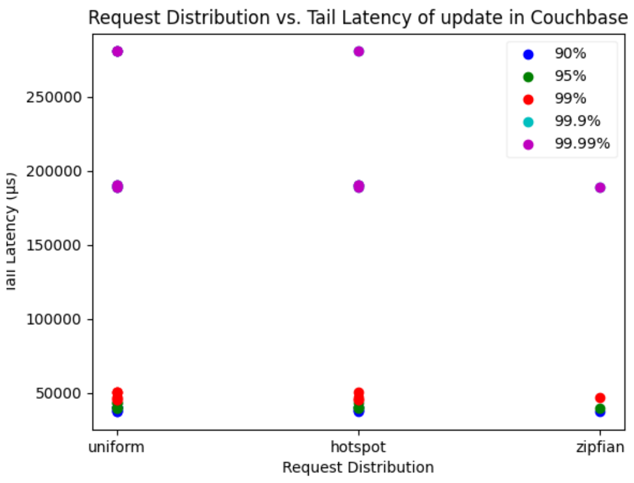
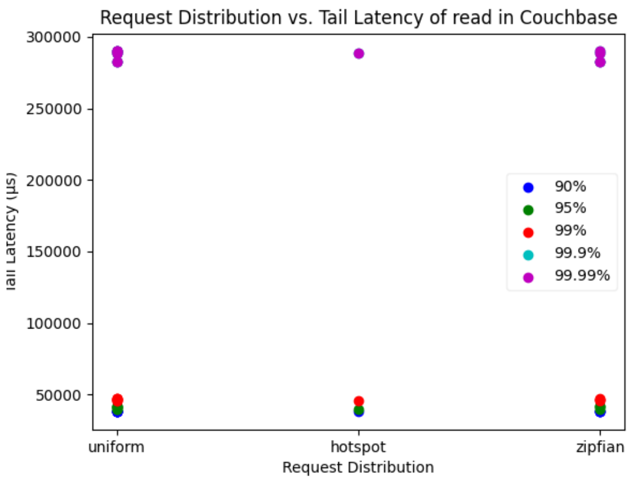


Fig. 13: Request distribution impact on read tail latency in Couchbase



of various operations in Couchbase. Unfortunately, we were constrained to a limited set of test cases by the Community edition of Couchbase, which requires that every node in a given cluster runs one of three possible combinations of services, listed below:

- data
- query, index, data
- FTS, query, index, data

Based on those requirements, the first thing to note is that every node must run the data service. This means that the data service could not be independently scaled from one to many nodes. Also, since the query service cannot run independently, in order to test the impact of scaling the query service, we needed to scale not just the query service from one to many nodes, but the query, index, and data service grouped together from one to many nodes, with the remaining nodes for each iteration only running the data service. This same constraint applies to the index service. As a result, the query and index services could not be tested independently and thus the results of the query service scaling tests are the same results for the index service scaling test. Lastly, since the FTS (full text search) service can not run independently, in order to test the scaling of FTS, we needed to scale not just the FTS service from one to many nodes, but the FTS, query, index, and data services grouped together, with the remaining nodes for each iteration only running query, index, and data. This in essence means that all nodes are running query, index, and data for all iterations, but only the FTS service is scaling horizontally even though it is tied to the other services. In short, we are only able to test the impact of scaling FTS, query, and index, where the testing for query and index is essentially duplicated.

Our driving test algorithm for testing the impact of multidimensional scaling on the tail latency of each operation is shown in Algorithm 2. Unlike the tests in the previous phase, this driving algorithm is only one flat loop rather than a nested one.

D. Phase Two Test Results

1) FTS Scaling vs. Operation Latency: As seen in Figures 16, 17, and 18 scaling the FTS service beyond one node to two significantly reduced the top percentile latencies for the insert, update, and FTS operations. However, there appears to be a slight spike (local maximum) in tail latency for the update, delete, and FTS operations when FTS is running on three of the five nodes. Since three happens to be the first number allowing for the existence of a quorum, this could be related

Fig. 14: Operation Proportion vs Average 95th Percentile Latency Across All Operations

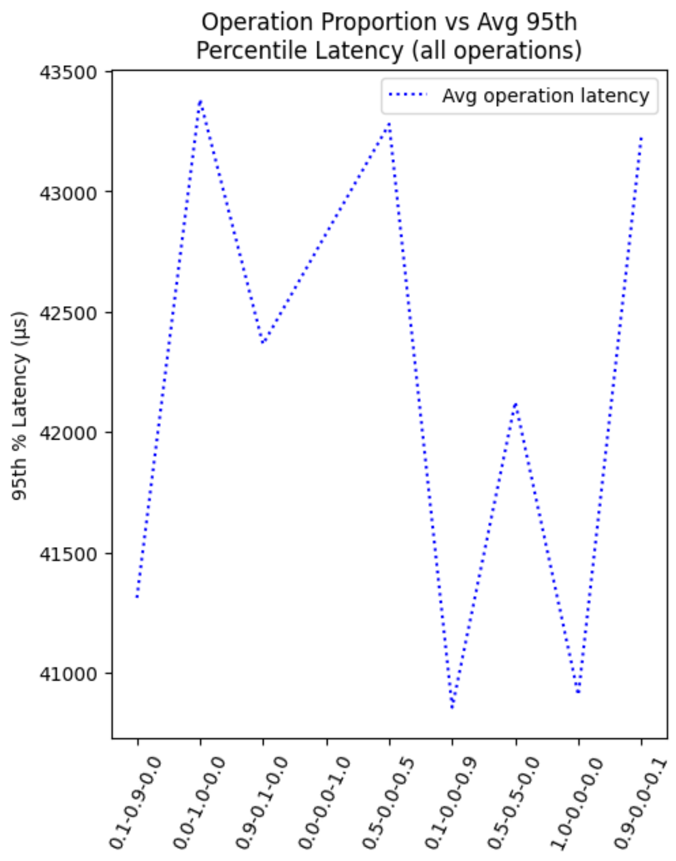
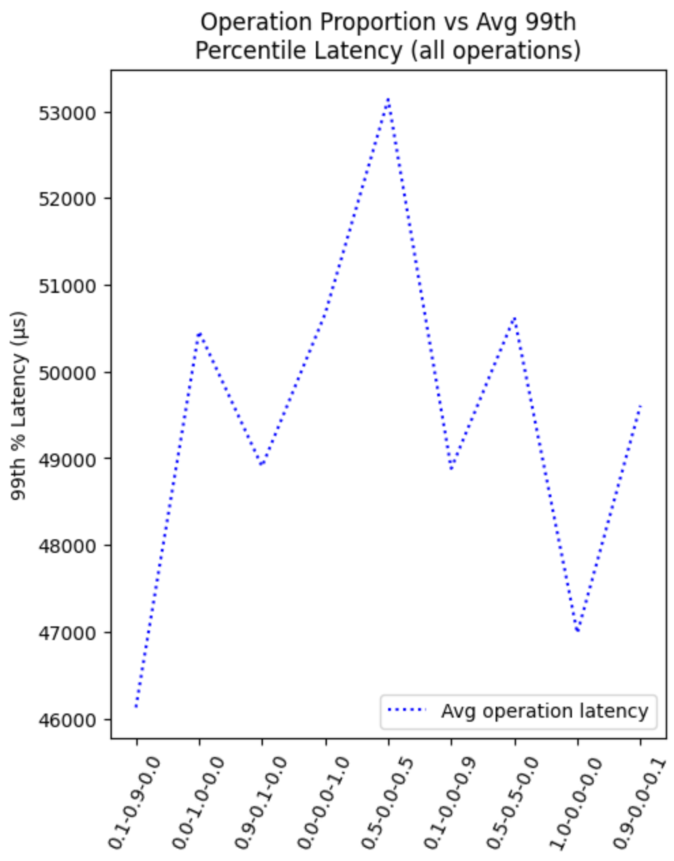


Fig. 15: Operation Proportion vs Average 99th Percentile Latency Across All Operations



to the way Couchbase handles cluster metadata management using the Chronicle consensus algorithm.

2) *Query & Index Scaling vs. Tail Latency:* As mentioned in subsection IV-C, the impact of scaling the query and index services could not be tested independently due to their interdependence in Couchbase Server. So, this subsection should be interpreted as a discussion of the impact of scaling both the query and index service together. In Figures 23 and 25, we see a similar maximum in top-percentile latencies for the update and N1QL query operations when the query and index services are running on three of the five nodes. For the FTS operation in Figure 21, we see a consistent, gradual decline in tail latency as the query and index services scale horizontally, and we see an even sharper decline in the tail latency of the delete operation, which essentially bottoms out once the query and index services are running on two nodes. It appears from Figure 22 that there is the the least amount of correlation between the scaling of query and index and the tail latency of the insert operation. When one considers that the primary service responsible for the insertion of new documents by key into the database is the data service and not the query or index service, this lack of correlation can be expected. Moreover, the consistent decline in the FTS tail latency with the scaling of the query and index service aligns with the documentation that states that the query service is responsible for performing scan operations on relevant search indexes in order to serve FTS queries; adding more of what's serving the queries should indeed have a positive impact on query latency.

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) *Replication Count Impact On Operation Latency:* In this evaluation, the number of replications is not treated as a variable. It is expected as data replication requirements change, the tail latency of write operations would change as well since all mutations would need to be streamed to the replicas. Coupled with different durability settings (i.e., Couchbase's *majority*, *majorityAndPersistToActive*, and *persistToMajority* durability configurations), the impact could be even more substantial. For example, for a cluster of a given size, how does changing the number of required replications from one to three affect the tail latency of the insert operation for each of the three main durability configurations? With more replication

Algorithm 2 Service Scaling Test Driver

```
1:  $CLUSTERSIZE$ = 5
2:  $DURABILITY$ = ‘medium’
3:  $BUCKETSIZE$ = 200 // *num docs*
4:  $bucket$ = createBucket($BUCKETSIZE$)
5:  $serviceLayouts$ = getServiceLayouts()
6:  **for each** $slayout \in serviceLayouts$ **do**
        setupHeterogeneousCluster($slayout$)
7:      flushBucket($bucket$) // clean slate
8:      runInserts($bucket$,...) // *write latencies*
9:      runN1QLSelects($bucket$,...) // *write latencies*
10:     runFTS($bucket$,...) // *write latencies*
11:     runUpdates($bucket$,...) // *write latencies*
12:     runDeletes($bucket$,...) // *write latencies*
13:     flushBucket($buckete$,...) // clean slate
14: **end for**
15: $generatePlots()$ // *build plots with written data*
```

Fig. 16: Effect of Scaling FTS Service on FTS Operation

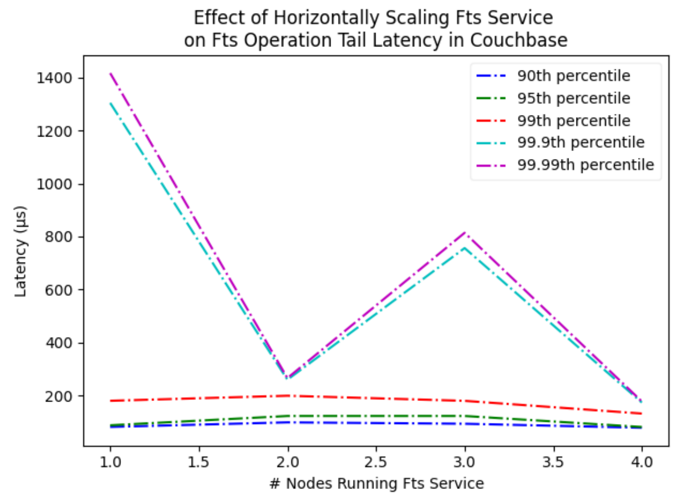


Fig. 17: Effect of Scaling FTS Service on Insert Operation

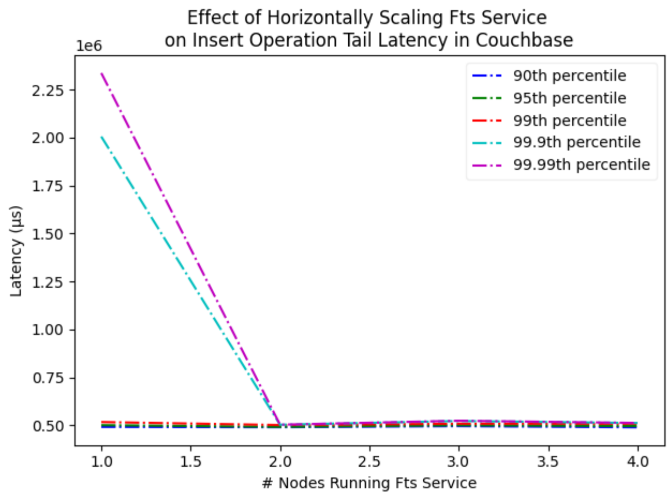
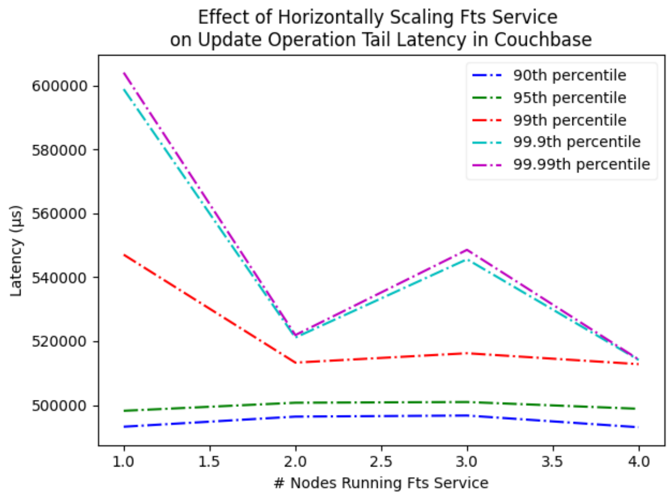


Fig. 18: Effect of Scaling FTS Service on Update Operation



and stronger durability settings, the cluster is more resilient in case of failures, but is it worth the expected increase in write tail latencies that could impact the customer experience?

2) Node Failover Impact on Operation Latency: Another item we did not 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 section III, it currently offers the capacity to run this kind of test, but it has not yet been implemented.

3) Work Load Condition of the Server: In this evaluation, the EC2 instances run only the Couchbase Server software. However, in a real-world application, cluster nodes may run a number of applications, which may introduce a series of background activities that may execute irregularly and cause resource contentions on the server, resulting in performance degradation for one or all applications involved. Therefore, we

Fig. 19: Effect of Scaling FTS Service on Delete Operation

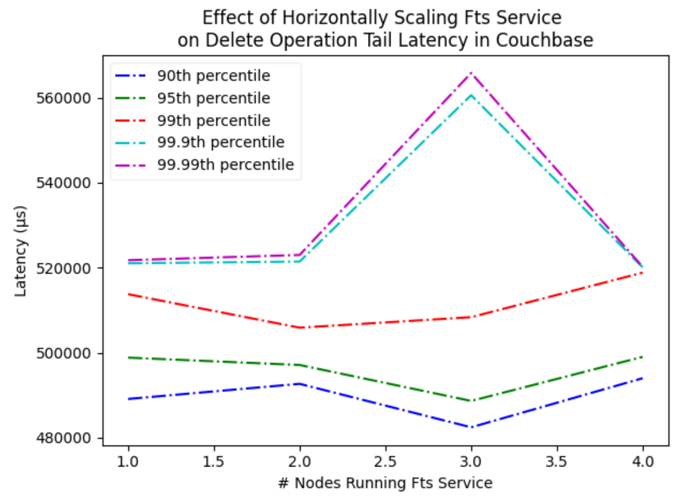


Fig. 20: Effect of Scaling FTS Service on N1QL Query Operation

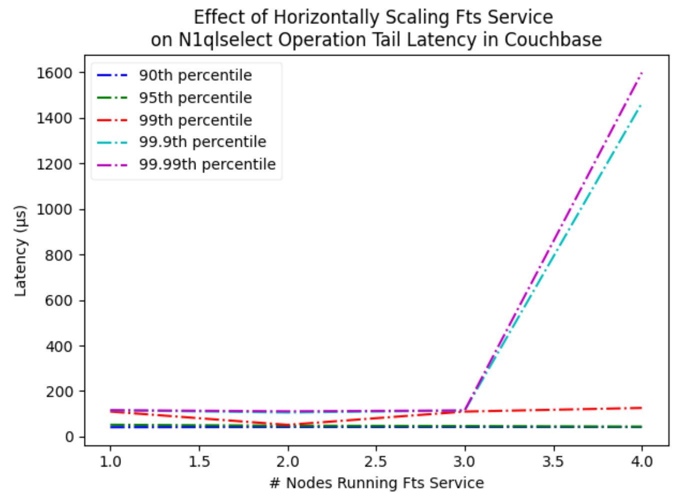


Fig. 21: Effect of Scaling Query & Index Services on FTS Operation

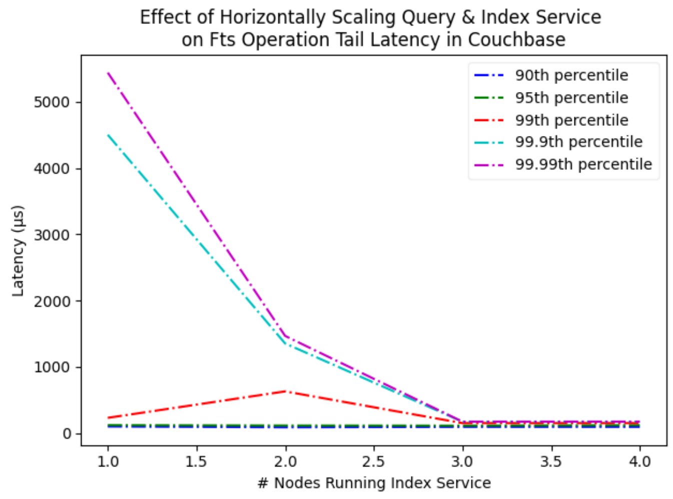
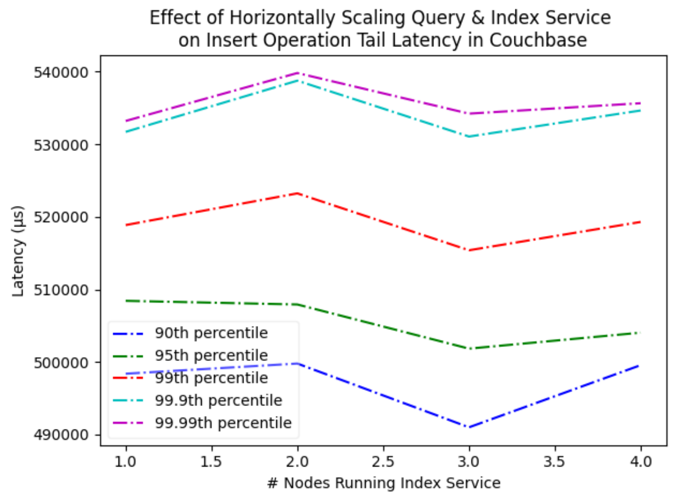


Fig. 22: Effect of Scaling Query & Index Services on Insert Operation



have planned but not yet implemented an approach for testing the effect of daemon processes on Couchbase operations. The plan is to spawn up a certain docker container that will consume a certain percentage of available CPU while making all other variables like cluster size, database size, and service layout constant, and analyze the daemon process's impact on the tail latency of different operation types.

4) Instance Type: In this evaluation, we are hosting our Couchbase cluster on a set of virtual machines provisioned with AWS EC2, a service that provides many configuration options around networking, volume storage, resource allocation via selection of instance types, and more. In this analysis we use the "t2.xlarge" instance type, as it meets the hardware requirements specified by Couchbase documentation. However, we remain interested in plotting the relationship between EC2 instance types and tail latencies of Couchbase data operations; we know horizontal scalability is a big advantage of Couchbase, but how does it respond to vertical scaling? Of course, the general expectation is that the more powerful the instance type, the smaller the tail latency (not to mention the guaranteed EC2 price change). We would also expect the existence of an inflection point at which vertical scaling no longer results in a significant reduction in tail latency. Also, that expected inflection point is likely different for different workloads. For each real-world application, there must exist some optimal set of instance types for a Couchbase cluster. If you can plot cost increase and tail latency decrease along a

Fig. 23: Effect of Scaling Query & Index Services on Update Operation

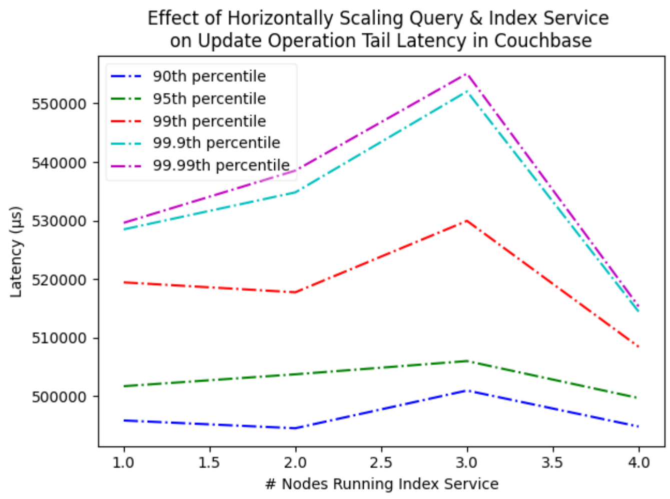
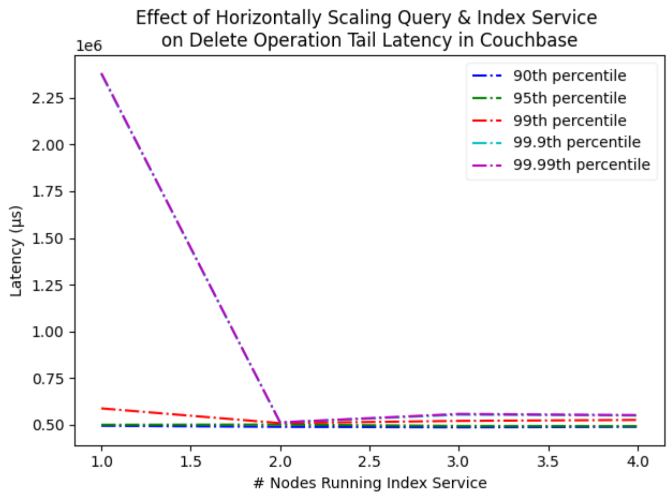


Fig. 24: Effect of Scaling Query & Index Services on Delete Operation

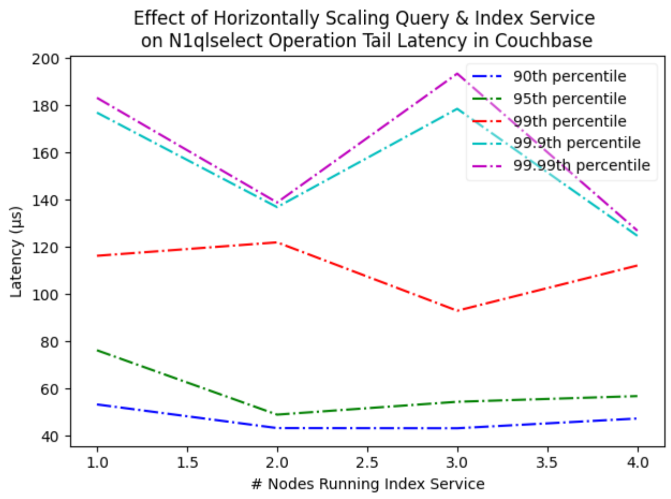


change in instance type configurations for a cluster, then you can identify the point at which the tail latency's negative slope is no longer steeper (if it ever is, that is) than the cost increase's positive slope; that is the inflection point where vertical scaling is not worth the money.

VI. Conclusion

Our goal with this study was to take Couchbase Server – an open-source, document-oriented, multi-model, distributed NoSQL database riding the wave of next-gen database systems – and gain an understanding of its multitude of configurable properties and how they interact with each other, and more importantly, how they combine to impact the tail latencies of key operations such as reads, insertions, updates, and deletions.

Fig. 25: Effect of Scaling Query & Index Services on N1QL Query Operation



We approached this goal with a strong focus on automation such that we could fine-tune and gauge a number of different settings without much overhead. This project, which makes heavy use of orchestration tools like Ansible and Vagrant in conjunction with cloud infrastructure via AWS EC2, combines experiences and perspectives from the rich arenas of both distributed systems and cloud computing, and rests on the shoulders of other open source giants who contributed frameworks like YCSB that allowed entire phases of this evaluation to be executed.

References

[Coo+10] Brian F Cooper et al. "Benchmarking cloud serving systems with YCSB". In: *Proceedings of the 1st ACM symposium on Cloud computing*. 2010, pp. 143–154.

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

[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.

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