INFO-H-415: Advanced Databases

Implementation of the YCSB benchmark for Couchbase and CouchDB

Marwah Sulaiman

Sara Saad

Otto Wantland

Sebastian Neri

Prof. Esteban Zimányi

2024

Contents

Abstract

As the project for INFO-415: Advanced Databases, we were tasked with investigating a database technology and two different tools that incorporate it. We chose NoSQL Document Stores as the database techonology and Couchbase and CouchDB as the two tools to implement. As part of the project we worked with this two tools to understand their internals, benchmark both of them and present a real world application for which they may be most suitable.

Chapter 1 consists of an overview of NoSQL databases, focusing on our chosen technology: Document Stores, specifically for the tools Couchbase and CouchDB.

Chapter 2 explains our choice of real world application, Booking.com, as well as laying out the reasons why a document store database would be an ideal fit for the application's data management needs.

Chapter 3, explains our benchmark of choice, the Yahoo! Cloud Serving Benchmark as well as the reasoning for choosing it for the project and the caveats of it's results when comparing our tools for our especific application.

Chapter 4 showcases the implementation steps for both tools and how we benchmarked them.

Chapter 5 presents the YCSB evaluation results and Analysis and a conclusion of our research.

Chapter 6 presents our additional queries tested against a real booking.com dataset in both tools.

Finally chapter 7 shows the projects conclusions and recommendations.

This report's explanations are based on the project's GitHub repository, accessible via the link below: `https://github.com/marwahmh/YCSB_DocumentDBs`

Introduction

As web services and applications have become ever more ubiquitous in modern life so too has the amount of data that these services must manage and the variety of data types that they must be able to handle. Document stores are a database technology that arose as an answer to these requirements, skewing the traditional SQL model for one composed of record objects like JSON or XML files that allow for less rigid data structures and easier integration with applications.

For this work we sought to analyze this database technology by focusing on two tools that implement it, comparing them through a benchmarking process using the Yahoo! Cloud Serving Benchmark (YCSB) and then analyzing the benchmarking process as a professional data engineer would to find the best option for a particular application.

Chapter 1 consists of an overview of NoSQL databases, focusing on our chosen technology: Document Stores. In this chapter we discuss the key characteristics of this technology such as its flexible data model, its schema-less design and why it's suited for modern distributed applications. We also delve into the specific tools we chose to investigate for this work: Couchbase and CouchDB. We delve into each tool's history as well as its most important characteristics.

Chapter 2 provides a real-world example of where these technologies may be implemented. Using the application Booking.com as an example we pose the question of how we can apply the results obtained by the benchmarking process to determine the best tool to use for the application's needs.

Chapter 3 is comprised of an introduction to our chosen benchmark, the Yahoo! Cloud Serving Benchmark, where we explain why we chose it as well as its core elements. We delve into how workloads are composed and run as well as what each of them is testing in the database.

Chapter 4 documents the steps taken to perform the benchmark. We talk about the installation process for both the chosen tools themselves as well as the benchmarking tool.

Chapter 5 showcases the results of the benchmarking process.

Chapter 6 presents our additional queries tested against a real booking.com dataset in both tools.

Chapter 7 provides our conclusions as well as parting thoughts on the project.

1 Document Store Databases

1.1 NoSQL Databases

Document store databases are a subsection of the broader **NoSQL** database technology. NoSQL databases —also known as "not only SQL" databases—are made to be scalable and adaptable, able to manage enormous volumes of unstructured or semi-structured data. In contrast to traditional relational databases, NoSQL databases provide dynamic schemas that make it simple to adapt to changing data structures without requiring migrations. Both features make them exceptional for use cases where large swaths of evolving data are being created and queried by a variety of end users, such as modern web applications.

NoSQL databases are designed to support a variety of data formats depending on the most efficient way to manage a particular type of information. Some of these technologies include graph databases, best suited for showcasing the connections within nodes, key-value pairs, the simples form of NoSQL in which the data model is organized into a collection of key-value pairs, wide-column stores, where information is stored in columns instead of rows allowing for the storing of sparse data sets, and document stores. A common aspect of all these systems is their lack of a strict schema structure like the one commonly found in relational databases. Their primary advantage over relational databases is that they are designed to handle unstructured data, such as text files, emails, and multimedia files. This lack of a schema allows the database system to scale much faster than traditional systems, favoring the accommodation of immense quantities of data over the overall structure of the system. NoSQL databases are better suited to take advantage of horizontal scaling.

Horizontal scaling refers to the practice of increasing a structure's processing power by adding additional nodes rather than by expanding a single machine's hardware. Horizontal scaling takes advantage of load distribution to handle complex data querying. This also accounts for another of NoSQL's appeals for modern web applications, the crucial importance of constant availability. By allowing for the use of distributed systems NoSQL systems help to ensure a much higher rate of failure resistance for the data structure as well as no downtime. [Sli]

NoSQL databases are not only lauded for their high availability, but they are also useful for large scale appli-

cations because they're optimized for fast data retrieval and processing. By handling data in the same format as the applications that are creating it, they allow for a much smoother implementation within modern workflows. These systems are ideal for real-time data handling and processing.
Among NoSQL databases, **document stores** are a specific type and the main focus of the present work, which will be elaborated on in the next section. Document collections are how NoSQL document databases store and arrange their data. Each document may have distinct fields and attributes, and they may lack a clear structure. Although it is not required by the database system, documents from the same collection must be comparable. Documents that lack a predetermined framework and are independent of one another are referred to as semi-structured documents. [IBM]

1.2 Document Stores Technology

Document stores are database systems that allow for the storage, retrieval, manipulation and management of document-oriented information. These documents can come in a variety of formats though they most commonly adopt a **JSON**-like structure. This allows for greater ease of development, since most developers are already used to working with JSON documents they don't need to spend as much time developing new data structures and mapping out the relationship between these structures in the same way that they would if they had to work with a traditional relational database.
Document stores are especially well suited for online applications that require management of large collections of transforming information. This can include catalogs within an e-commerce platform, or different types of content such as text, audio and video files. The lack of a central schema means that individual documents within the database can be changed without affecting the rest of the elements in the database.
Document stores are inherently a sub-concept of the key-value store with negligible differences beyond the fact that in key value stores, data is inherently non-transparent to the database, whereas document stores depend on the document's internal structure for it to extricate metadata used by the database system for manipulation and optimization. [Inca]
To query information within these databases most of them provide their own language or API that allows users to traverse the document structures and find the relevant information. All document stores implement their own **CREATE, READ and UPDATE** applications and some, such as Couchbase, even offer their own version of SQL.
To further comprehend the way this database technology works we researched two tools used for document store implementation. The tools are CouchBase and CouchDB.

1.2.1 B-Tree Structure

Both CouchDB and Couchbase implement an append-only Balanced Tree structure. Most database technologies implement Balanced Tree structures in their architectures given that it's one of the most efficient ways to store, manage and sort data in an efficient manner. [Foud]

Balanced Tree Structure

B-trees maintain a balanced structure, ensuring that all leaf nodes are at the same depth. This property guarantees consistent and predictable performance for operations like lookup, insertions, deletions, and scans.
Additionally, each node in the tree can contain multiple keys and pointers to child nodes, reducing the height of the tree compared to binary trees. When a node reaches its capacity, it splits, ensuring the tree remains balanced. This makes B-trees highly efficient for disk-based storage systems. B-trees group multiple keys into nodes (pages) that fit into a single disk block, minimizing the number of I/O operations required to traverse the tree. [Uni]

Append-Only Balanced Tree Structure

Append-only Balanced Trees is an adaptation designed to optimize write operations, particularly in scenarios where data integrity and durability are critical.
Append-only Balanced Trees work by using immutable writes. Instead of overwriting existing data, updates and new records are appended to the end of the data file. This approach avoids the risks of partial writes and corruption, as the original data remains untouched until the new version is fully written. Furthermore, when a node is updated, a copy of the node (and its parents, up to the root) is created and appended to the tree structure. This ensures that the update does not affect readers accessing the older version of the tree, thus ensuring the ability of multiple readers to access the database without being blocked by incoming writes.
This implementation also helps prioritize efficient versioning and crash recovery. Each update creates a new version of the tree, making it easy to recover from crashes or roll back to a consistent state since the old tree structure is retained until it is explicitly garbage-collected, allowing for point-in-time snapshots. This helps to ensure high availability for the system.[Unk]
Since appending data sequentially is faster than random I/O this structure is especially beneficial for systems like CouchDB and Couchbase that handle high-throughput write workloads.

1.2.2 CouchBase

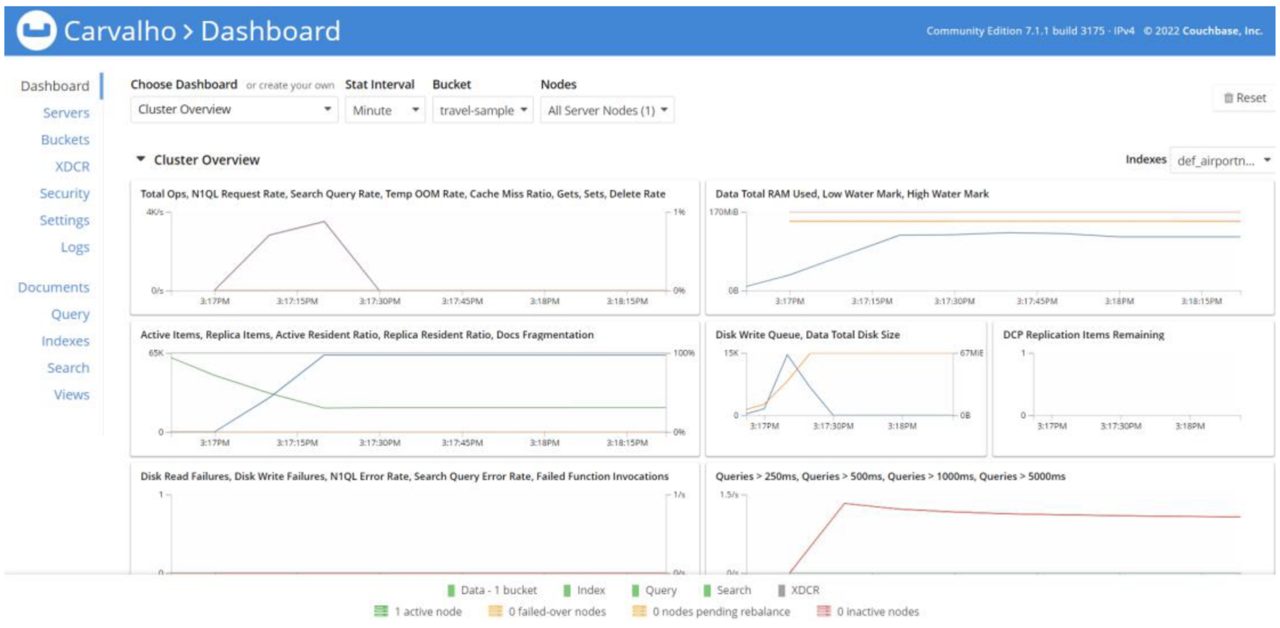
CouchBase Data Structure

Couchbase is an open-source document store database, developed in C++ by Couchbase Inc. in 2011. Couchbase is designed for web and mobile applications. It stores data in the form of JSON documents that reside in structures called **buckets**.
The CouchBase data structure is comprised of three major elements: Clusters, Nodes and Buckets. A Couch-

Base cluster is a collection of collected nodes and acts as the overarching structure for the database. Clusters handle the logging of transactions and queries, the scheduling of backups and the setting of security measures that affect the database. Clusters are also the area where Cross Data Center Replication is performed in order to ensure data availability [Incc].
Each CouchBase cluster must be composed of at least one node. Nodes are systems running an instance of CouchBase Server. The use of multiple nodes allows for easier horizontal scaling by permitting the distribution of tasks among multiple systems. Nodes can be added to a cluster directly or can be instantiated in a different system and then connected to an existing cluster. Either process requires that the cluster perform an operation called a **rebalance**. This operation is used to redistribute the buckets that were being processed by the original nodes among the newly added nodes ensuring an easy way to expand the database's capabilities. Similarly, if a node is removed from the cluster either because of maintenance or permanent deletion the same rebalance operation is performed in order to maintain proper distribution among the existing nodes.
Buckets are created within each cluster, up to a maximum of 30 buckets per cluster. Buckets are divided into two types: CouchBase buckets and Ephemeral buckets. Couchbase buckets are the most common type used. They store data persistently and in memory and allow their data to be automatically replicated and distributed among the available nodes to ensure data replication and availability. Ephemeral buckets hold information that does not require persistence. It stores data only in memory and not in disk which leads to permanent deletion if the bucket's RAM-quota is exceeded. Buckets in CouchBase can be assigned RAM-quotas which prevent an overuse of resources by any particular operation. Depending on the bucket type, if this RAM-quota is reached information will be removed from memory and either held only on disk or eliminated entirely. This allows for a high emphasis on system performance and availability. [Incb]
Buckets organize data within a hierarchical structure composed of **Scopes and Collections**. A collection is a data container that exists within a bucket. Each collection contains data items, which must all have different names. Collections offer a way to organize data items by categories such as content type or by the application that will need to access them. Collections are held within scopes. Scopes are grouping mechanisms that allow the clustering of multiple collections by categories such as their content type or by their deployment-phase. Both structures are designed to allow for a logical grouping of data items for use with replication, distribution, querying, indexing, and other essential processes within the database.[Incf]

Querying CouchBase Data

In order to access the information held within buckets users must utilize **SQL++**, CouchBase's implementation of SQL syntax for use with JSON documents. SQL++ follows the same syntax as regular SQL, it expects a query composed of **SELECT, FROM and WHERE** statements. Listing 1.1 gives an example of what an SQL++ query looks like. [Ince]


Carvalho > Dashboard

Community Edition 7.1.1 build 3175 · IPv4 © 2022 Couchbase, Inc.

Choose Dashboard or create your own: Cluster Overview
Stat Interval: Minute
Bucket: travel-sample
Nodes: All Server Nodes (1)
Indexes: def_airportn...

1 active node · 0 failed-over nodes · 0 nodes pending rebalance · 0 inactive nodes

Dashboard
Servers
Buckets
XDCR
Security
Settings
Logs
Documents
Query
Indexes
Search
Views
Reset
Cluster Overview
Total Ops, N1QL Request Rate, Search Query Rate, Temp OOM Rate, Cache Miss Ratio, Gets, Sets, Delete Rate
Data Total RAM Used, Low Water Mark, High Water Mark
Active Items, Replica Items, Active Resident Ratio, Replica Resident Ratio, Docs Fragmentation
Disk Write Queue, Data Total Disk Size
DCP Replication Items Remaining
Disk Read Failures, Disk Write Failures, N1QL Error Rate, Search Query Error Rate, Failed Function Invocations
Queries > 250ms, Queries > 500ms, Queries > 1000ms, Queries > 5000ms
Data - 1 bucket
Index
Query
Search
XDCR

Figure 1.1: Couchbase web interface showing a working cluster.

```sql
SELECT a.country FROM default:‘travel-sample‘.inventory.airline a
WHERE a.name = "Excel Airways";
```

Listing 1.1: SQL++ Example

In the sample query the user is requesting the country associated with the airline Excel Airways whose information is held in the Airline collection, that exists within the Inventory scope which exists within the travel-sample bucket. This is how scopes and collections allow for more explicit querying of information within the database.

SQL++ implements many basic elements of SQL into the CouchBase environment, allowing for simple arithmetic operations, null value handling, joins, explain statements and index creation. Additionally it provides commands for navigating the JSON data structure by looping through arrays and objects. [Incd]

Indexing on CouchBase

CouchBase allows the creation of both Primary and Secondary indexes on collections. A primary index is defined as one based on the user-generated document key that accompanies every JSON file. Since CouchBase enforces a uniqueness constraint on the document key this can be used for primary scans of the dataset.

```sql
CREATE PRIMARY INDEX ON airline;
```

Listing 1.2: Primary Index Creation

Listing 1.3 offers a look at the resulting index structure created within the cluster.

```json
[
  {
    "indexes": {
        "bucket_id": "travel-sample",
        "datastore_id": "http://127.0.0.1:8091",
        "id": "c6f4ec5d935e1626",
        "index_key": [],
        "is_primary": true,
        "keyspace_id": "airline",
        "name": "#primary",
        "namespace_id": "default",
        "scope_id": "inventory",
        "state": "online",
        "using": "gsi"
      }
    }
]
```

Listing 1.3: Primary Index Structure

In addition to primary indexes CouchBase also allows for the creation of secondary indexes based on any key-value or document-key regardless of data type. [Inch]

1.2.3 CouchDB

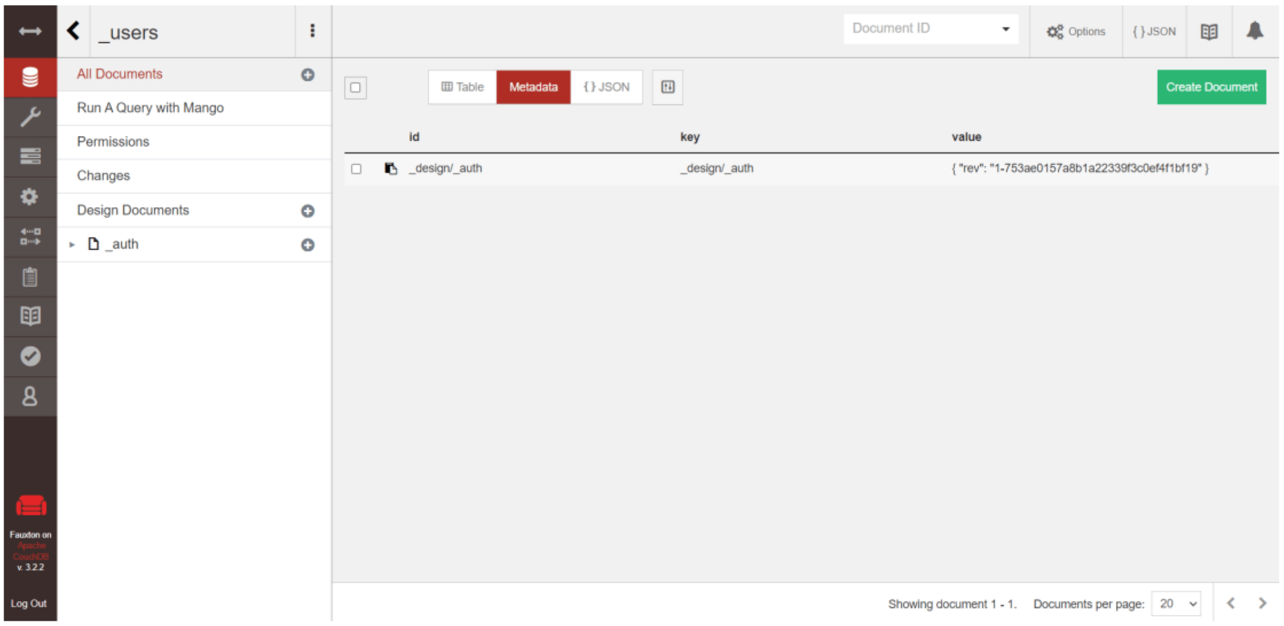
CouchDB Data Structure

CouchDB is an open-source document-oriented database developed in Erlang by Apache Software Foundation in 2005. Similarly to CouchBase, CouchDB also uses a JSON format for its documents. However, it's overall structure uses slightly different names. CouchDB stores its documents in databases, which exist within nodes that compose clusters.

While clusters in CouchDB function in a similar way to CouchBase clusters they have a key difference, CouchDB allows for the creation of singular nodes not associated to any cluster. This allows for easier onboarding while still letting the user integrate the node into a cluster should the need for scaling arise.

Nodes function similarly to CouchBase nodes, however CouchDB places a higher emphasis on document replication. Whenever a node is created the user must indicate the number of **shards and replicas**. Shards are parts of the database that can be replicated equal to the number of replicas issued. In practice this means that for a database with a shard value of 2 and a replica value of 3 every database will be split into 2 shards and copied 3 times, leading to 6 replica copies of the database existing. Each of these replica copies can exist on at most 6 nodes, fixing a limit on the cluster.[Foub]

Documents are stored within databases contained in the nodes. Similarly to buckets these databases allow documents of any type to be stored and maintain them independently of one another, allowing end users to modify them without needing to worry about problems with a wider schema. These databases can be categorized using views, in a similar fashion to Couchbase's collections views allow for a way to organize the documents within the data structure. However, CouchDB's views also serve as the way through which information is queried.[Fouc]



id	key	value
_design/_auth	_design/_auth	{ "rev": "1-753ae0157a8b1a22339f3c0ef4f1bf19" }

Figure 1.2: CouchDB web interface: Fauxton

Querying CouchDB's Data

CouchDB does not have any form of SQL implementation. Instead, data querying is handled through Javascript functions and the use of views. Views are created within a database structure similar to the way tables are created in relational models. Listing 1.4 gives an example of what this view structure looks like.

```
{
    "_id": "_design/application",
    "_rev": "1-C1687D17",
    "views": {
        "viewname": {
            "map": "function(doc) { ... }",
            "reduce": "function(keys, values) { ... }"
        }
    }
}
```

Listing 1.4: View Definition

Within the view structure multiple views can be defined. These are the equivalent of SQL functions that would need to be executed on the selected database. For example, if a user wished to perform a simple select operation for a particular key, like in Listing 1.5

```
SELECT field FROM table WHERE value="searchterm"
```

Listing 1.5: Key Lookup

Then the view function must be defined in the view document, as shown in Listing 1.6

```
{
    "_id": "_design/application",
    "_rev": "1-C1687D17",
    "views": {
        "viewname": {
            "map": "function(doc) {
                        if(doc.value) {
                            emit(doc.value, null);
                        }
                    }",
            "reduce": "function(keys, values) { ... }"
        }
    }
}
```

Listing 1.6: Implementing Key Lookup

With the proper function implemented in the view the lookup query can be performed through the operation shown in Listing 1.7 [Foua]

```
/database/_design/application/_view/viewname?key="searchterm"
```

Listing 1.7: Key Lookup Performed With Javascript

Many similar implementations can be achieved by using the view functionality of CouchDB, however, it is noted that best practices used in SQL will not translate into all of CouchDB.

2 Real World Application

The main appeals of NoSQL systems such as CouchDB and Couchbase is that they provide easily scalable systems designed for maximum availability. Because of this they are exceptional for use cases where large swaths of evolving data are being created and queried by a variety of end users, such as with modern web applications.

Applications like Booking.com demand highly interactive and scalable solutions to meet the needs of millions of users searching for accommodations in real time. This includes handling a high number of complex data operations such as availability lookups, pricing, customer reviews, and filtering based on preferences. A data engineer in charge of handling Booking's system may find themselves with the need to update their infrastructure and could see Document stores as a viable option for their business model. Using this as a basis we examined both of our chosen tools to try to determine which of them would provide the better solution for this real-world application.

2.1 Brief Comparison of Couchbase and CouchDB

Both Couchbase and CouchDB support JSON document and key-value data models, making them suitable for applications with dynamic schemas like Booking.com. In terms of consistency and replication Couchbase offers strong consistency with distributed ACID transactions and master-master replication. Both characteristics help to ensure constant data availability across multiple nodes. On the other hand, CouchDB provides eventual consistency, which might require more cautious handling for real-time updates. It also supports master-master replication with quorum options for reliability.

Similarly, Couchbase supports automatic failover and cross-data-center replication whereas CouchDB supports cross-data-center replication but lacks built-in automatic failover. This, in combination with Couchbase's caching integration means it's not only better prepared for handling outages but also better at performing under high-traffic scenarios that may appear, for example during peak holiday season when millions of users are searching for holiday accommodations.

In terms of SQL implementation Couchbase offers SQL++, a query language for JSON files that is based on SQL though not a direct implementation. CouchDB, however, does not have any implementation of SQL

and instead defines its own functions for data querying and management. As we've seen CouchDB's query implementation relies on the use of views to allow for data querying. This process may prove troublesome for a team already used to working with SQL as their primary language.
Both technologies support secondary indexes, but Couchbase also provides eventing services, enabling real-time notifications for data changes, crucial for interactive applications like availability updates. Additionally, Couchbase offers a wide array of services such as full-text search, analytics, and mobile sync, making it an overall more comprehensive solution than CouchDB.

2.1.1 Modelling the Application

Aside from the general differences in both tools we also need to understand what exactly each one would provide our chosen application.

CouchBase

One of CouchBase's main selling points is it's priority in delivering scalability and speed. Couchbase integrates caching at multiple levels, including an in-memory managed cache, which reduces the need for disk I/O during read and write operations. This could ensure lightning-fast responses for operations like room availability lookup or pricing checks. The append-only B-tree storage further optimizes performance by minimizing disk fragmentation, making it possible to handle millions of concurrent users with minimal latency, which would prove invaluable during peak seasons such as holidays.
Couchbase is also feature rich. Its SQL++ query language allows developers to run complex queries similar to traditional SQL while supporting JSON's hierarchical data structure making it easy for experienced data scientist and engineers to pick up and use. Furthermore, CouchBase offers Mobile Sync, which is the ability to sync data with mobile devices that may be offline at the moment. Travelers can search or confirm bookings even in areas with limited connectivity, and the system synchronizes seamlessly once the device reconnects. Additionally, CouchBae offers advanced features like full-text search which enable users to perform natural-language queries (e.g., "hotels near Eiffel Tower with free WiFi"), while analytics services help generate insights, such as trends in room availability or user preferences.[Incg]
CouchBase's structure is also designed to offer a high degree of reliability. By offering strong data consistency, updates to bookings or inventory are immediately visible to all users, avoiding double bookings or outdated availability. Similarly, Automatic Failover ensures that if one node in the cluster fails, the system quickly reroutes requests to another node without downtime. This is essential for maintaining trust with users and reducing losses during high-demand periods. Finally, Cross-Data-Center Replication (XDCR) allows Booking.com to replicate data across global data centers, ensuring low-latency access for users in different regions and robust disaster recovery.

CouchDB

While CouchBase presents its scalability and speed as its main features CouchDB focuses on simplicity and flexibility. CouchDB's architecture is straightforward, making it easy to set up and maintain. This means less initial commitment for its implementation. Its eventual consistency model is well-suited for operations where slight delays are acceptable, such as updating user activity logs or recommendations based on browsing history. The simplicity of its JSON-centric approach ensures rapid development and adaptation to changing requirements, ideal for non-critical data pipelines like analytics dashboards. Furthermore, CouchDB can provide real-time notifications of document updates. For Booking.com, this can support asynchronous processes, such as alerting users about price drops, newly added accommodations, or last-minute availability. These updates allow the system to engage users proactively, improving customer experience and potentially driving more bookings.
Another aspect of CouchDB to consider is its cost-effectiveness. Since CouchDB does not include features like integrated caching or advanced services, which reduces resource usage and operational overhead, it can be a lightweight and economical choice for less resource-intensive components. Such services as activity monitoring, non-critical data archival, or experimental features can benefit from this implementation.
After analyzing the benefits offered by document store databases for our chosen application we needed a way to properly compare both of our chosen tools in order to select the correct one to implement. To be able to do this we chose to use the YCSB benchmark tool to see which tool would perform better under working conditions.

3 YCSB Benchmarking Framework

3.1 Introduction to YCSB

In this chapter, we will introduce the Yahoo! Cloud Serving Benchmark (YCSB), which is a standard benchmarking framework for evaluating the performance of NoSQL databases.It plays an essential role in the process of developing and testing data management systems.
The main appeal of the YCSB is it's widespread implementation. The client itself contains a variety of workloads designed to give a well-rounded view of a database's performance in key areas. However, the tool is also designed to be extensible, allowing users to implement their own workloads as well as create the proper interfaces to test out new databases.

3.2 Benchmark Architecture

Figure 2.1 depicts the YCSB architecture, which includes the YCSB Client (a workload executor), client threads, and the statistics module. YCSB Client is a Java program that generates data for loading into target databases as well as activities that describe workloads. In fundamental operations, the workload executor initiates many client threads. Each thread performs a series of actions, including calls to the database interface layer to load the database (the load phase) and execute workloads (the transaction phase). The database interface layer translates client threads' simple requests, such as read requests, into database calls. Threads measure the latency and throughput of their activities and report these results to the statistics module. At the end of the workload execution, the statistics module collects these measurements and delivers the execution duration, average latency, latency percentage, and a histogram or time series of latencies.

3.3 Benchmark Workloads

YCSB includes a set of predefined workloads that simulate common application scenarios. While the YCSB provides the ability to define custom workloads for particular applications the core set of workloads can help to give an overall picture of a system's performance.

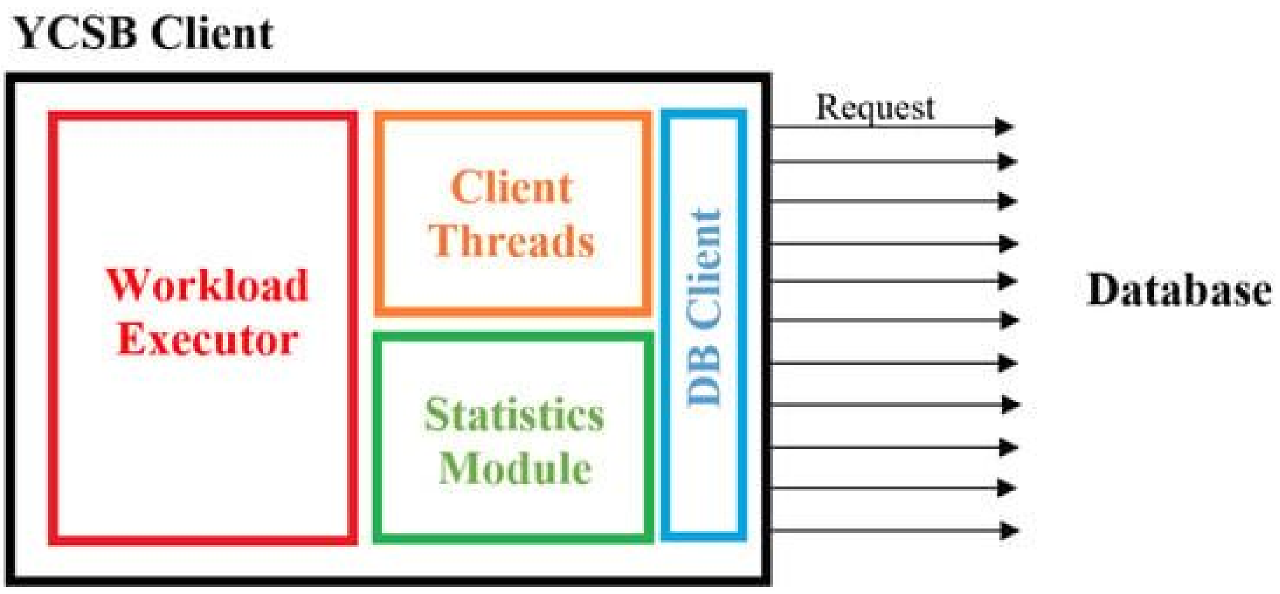


Figure 3.1: YCSB Architecture

This core is composed of six different workloads: [Bus]

- **Workload A: Update heavy workload**

  A mix of 50% reads and 50% writes. Meant to simulate a heavy load of update operations performed on the database.

- **Workload B: Read mostly workload**

  A mix of 95% reads and 5% writes workload. Meant to simulate an application where users are more likely to need to view information in the database rather than actively change it.

- **Workload C: Read only**

  A mix of 100% read operation.

- **Workload D: Read latest workload**

  A workload that consists of 95% reads and 5% inserts. Focuses on inserting new records and only querying those new records.

- **Workload E: Short ranges**

  A workload of reads where only short ranges of individual documents are queried instead of the whole document.

- **Workload F: Read-modify-write**

  A workload comprised of an operation where a record is read, modified, then written back into the database.

3.4 Benchmarking for Our Application

As was alluded before the YCSB benchmark offers a comprehensive tool for benchmarking a variety of different database technologies. Nonetheless, it was important to analyze wether the use of the core workloads would provide the full picture needed to properly decide on a tool for our application or if there were gaps that may require a different benchmarking process.

3.4.1 Key Metrics for Evaluation

While the benchmark provides a standardized way to evaluate NoSQL databases we also identified key metrics that are of great importance for the application. These are: high throughput, low latency, and scalability for search, booking, and user interaction.

Throughput consists of the number of operations (read/write) per second the database can handle under peak loads. This is critical for handling real-time queries like room availability or pricing during peak traffic.

Latency refers to the time taken to complete individual operations (reads, writes, updates). It is essential for providing a seamless user experience when searching for listings or completing bookings.

The application also places a premium on **scalability** to ensure consistent response times as Booking.com expands its inventory and user base.

Finally, **consistency** refers to wether the database provides immediate (strong consistency) or eventual consistency after updates. Consistency is critical for ensuring data accuracy in bookings to prevent double-booking scenarios.

3.4.2 Expected Workload Outcomes and Insights

Given our key metrics we believe that all six core workloads will provide relevant information for our tool comparison. Given what we know about this systems we also provided some hypothesis for each tool's performance with every workload.

- **Workload A: Update heavy workload**
  - **Couchbase:** Higher throughput and lower latency due to integrated caching and strong consistency.
  - **CouchDB:** Slower updates because of the append-only model and eventual consistency, though suitable for non-critical updates.
- **Workload B: Read mostly workload**
  - **Couchbase:** Lower latency due to in-memory caching and optimized storage for read-heavy workloads.
  - **CouchDB:** Acceptable performance, but slower for frequent, concurrent reads due to reliance on disk I/O.

- **Workload C: Read only**
  - **Couchbase:** Superior performance due to caching and full-text search capabilities.
  - **CouchDB:** Decent performance but less efficient under high read concurrency.
- **Workload D: Read latest workload**
  - **Couchbase:** Consistently strong performance because of its ability to serve recent data quickly via caching.
  - **CouchDB:** Performance could degrade as data grows due to lack of managed caching.
- **Workload E: Short ranges**
  - **Couchbase:** Efficient scans due to advanced indexing and SQL++ support.
  - **CouchDB:** Slower scans as it lacks rich indexing options and optimized query languages.
- **Workload F: Read-modify-write**
  - **Couchbase:** High throughput for concurrent read-write operations due to its distributed ACID transactions.
  - **CouchDB:** Limited throughput under write-heavy operations due to eventual consistency.

Table 3.1 showcases the expected results for the core workload benchmark test of our chosen tools.

Metric	CouchBase	CouchDB
Throughput	High	Low
Latency	Low	High
Consistency	High	Low
Scalability	High	High

Table 3.1: Expected Result for Key Metrics

3.4.3 Other Required Operations

While the core workloads provided by the YCSB tool allow us to validate key metrics for our application they do not consider the whole range of operations that the application would entail.

Basing ourselves on the work done by Keshav Murthy in his example of extending the YCSB databse for benchmarking JSON documents [Mur] we analyzed what other key metrics we would need to measure as well as the queries that could be implemented into the workloads to test these metrics.

Aside from the four basic operations bench-marked by the core YCSB workload, **INSERT, UPDATE, READ and SCAN** the extended benchmark would also need to test for **SEARCH** operations. These are essential for queries filtering accommodations by price range, rating, or availability.

PAGE is another operation that would require a new workload. This would be composed of a paginated display of accommodations, supporting OFFSET and LIMIT.

NestScan would also require implementation within the workload. This operation is necessary for querying reviews or amenities stored as nested fields within the document.

Aggregate functions are another necessary part of the application's functions that should be bench-marked.

Operations such as grouping accommodations by city or calculating average nightly price.

Finally a way to benchmark the **Report** operation would be necessary as well. This operation is required to generate detailed summaries of bookings within specific time frames or locations.

4 Set-Up and Implementations

4.1 Running YCSB with CouchBase

First, we installed couchbase server using thr guide in the official website.

For the benchmarking, we used the existing couchbase binding for YCSB, available here. It is compatible with the latest version of the benchmark.

To run a workload, use a terminal command like:

```
bin/run.sh -w workloads/workloada -R $record_count -O $record_count -G $thread_count
```

We created a shell script to run all our workloads with the specific number of records (1k, 10k, 100k, and 1M), each with 3 and 6 threads, and save our results.

```
#!/bin/bash

# Define variables
WORKLOADS=("workloada" "workloadb" "workloadc" "workloadd" "workloade" "workloadf")

RECORD_COUNTS=(1000 10000 100000 1000000)
THREAD_COUNTS=(3 6)
OUTPUT_DIR="output"
SCRIPT="bin/run.sh"

# Ensure output directory exists
mkdir -p "$OUTPUT_DIR"

# Main loop to iterate over workloads, record counts, and thread counts
for workload in "${WORKLOADS[@]}"; do
  for record_count in "${RECORD_COUNTS[@]}"; do
    for thread_count in "${THREAD_COUNTS[@]}"; do
      # Construct file names for this configuration
      LOAD_FILE="${OUTPUT_DIR}/${workload}-load-${record_count}-${thread_count}.dat"
      RUN_FILE="${OUTPUT_DIR}/${workload}-run-${record_count}-${thread_count}.dat"
      LOG_FILE="${OUTPUT_DIR}/${workload}-log-${record_count}-${thread_count}.txt"

      # Run the script with the current configuration
      echo "Running workload: $workload, Record count: $record_count, Threads: $thread_count"
      $SCRIPT -w workloads/$workload -R $record_count -O $record_count -G $thread_count > "$LOG_FILE" 2>&1

      # Move the generated result files to uniquely named files
      mv "${OUTPUT_DIR}/${workload}-load.dat" "$LOAD_FILE"
      mv "${OUTPUT_DIR}/${workload}-run.dat" "$RUN_FILE"
    done
  done
done

echo "All runs completed. Results are saved in $OUTPUT_DIR."
```

4.2 Running YCSB with CouchDB

The current official YCSB benchmark implementation does not have support for CouchDB, and therefore, we have implemented the binding. We build on this, which uses a very old version of YCSB. Changes in the code have been applied, as well as resolving many dependencies issues to allow support for the latest YCSB version.

Prerequisites

Before starting, ensure the following tools are installed:

- A Unix-like system (Linux/MacOS) or WSL on Windows.
- `curl`: For downloading files.
- `tar`: For extracting compressed files.
- Java Development Kit (JDK) 8 or later.
- CouchDB: Obtainable from CouchDB's official website.

4.3 Installing Maven

To install Apache Maven, run the following commands:

```
curl -O https://archive.apache.org/dist/maven/maven-3/3.6.3/binaries/apache-maven-3.6.3-bin.tar.gz
tar -xvzf apache-maven-3.6.3-bin.tar.gz
sudo mv apache-maven-3.6.3 /usr/local/apache-maven-3.6.3

```

```
# Set up environment variables
export M2_HOME=/usr/local/apache-maven-3.6.3
export PATH=$M2_HOME/bin:$PATH

# Apply changes (for Zsh users)
source ~/.zshrc

# Verify installation
mvn -version
```

Listing 4.1: Installing Maven

The above steps download Maven, extract it, configure the environment, and verify its installation.

4.4 Building the YCSB Project

Navigate to the YCSB repository directory and build the project:

```
cd YCSB
mvn clean package -DskipTests -X -Dcheckstyle.skip
```

Listing 4.2: Building YCSB

The `-DskipTests` flag skips tests during the build process, and `-Dcheckstyle.skip` bypasses style checks.

4.5 Installing CouchDB

Download and install CouchDB from its official website. During setup, configure an administrator account with the following credentials:

- Username: `Admin`
- Password: `password`

Ensure CouchDB is running on `127.0.0.1:5984`.

4.6 Loading Data into CouchDB

Run the following command to load 10,000 records into CouchDB:

```
./bin/ycsb load couchdb -P workloads/workloada \
  -p hosts="127.0.0.1" \
  -p url="http://Admin:password@127.0.0.1:5984" \
  -p recordcount=10000 \
  -p threadcount=1 \
  -s > LA1.txt
```

Listing 4.3: Loading data into CouchDB

Explanation of the parameters:

- `-P workloads/workloada`: Specifies the workload configuration file.
- `-p hosts`: Host address for CouchDB.
- `-p url`: URL with admin credentials for CouchDB.
- `-p recordcount`: Number of records to load.
- `-p threadcount`: Number of threads to use.
- `-s > LA1.txt`: Outputs results to `LA1.txt`.

4.7 Running Benchmark Tests

Execute the benchmark tests with the following command:

```
./bin/ycsb run couchdb -P workloads/workloada \
  -p hosts="127.0.0.1" \
  -p url="http://Admin:password@127.0.0.1:5984" \
  -p recordcount=10000 \
  -p threadcount=1 \
  -p operationcount=10000 \
  -s > LA1.txt
```

Listing 4.4: Running tests on CouchDB

Explanation of the additional parameter:

- `-p operationcount`: Specifies the number of operations to perform during the test.

Also, we have created a shell script to run all our desired workloads and configurations:

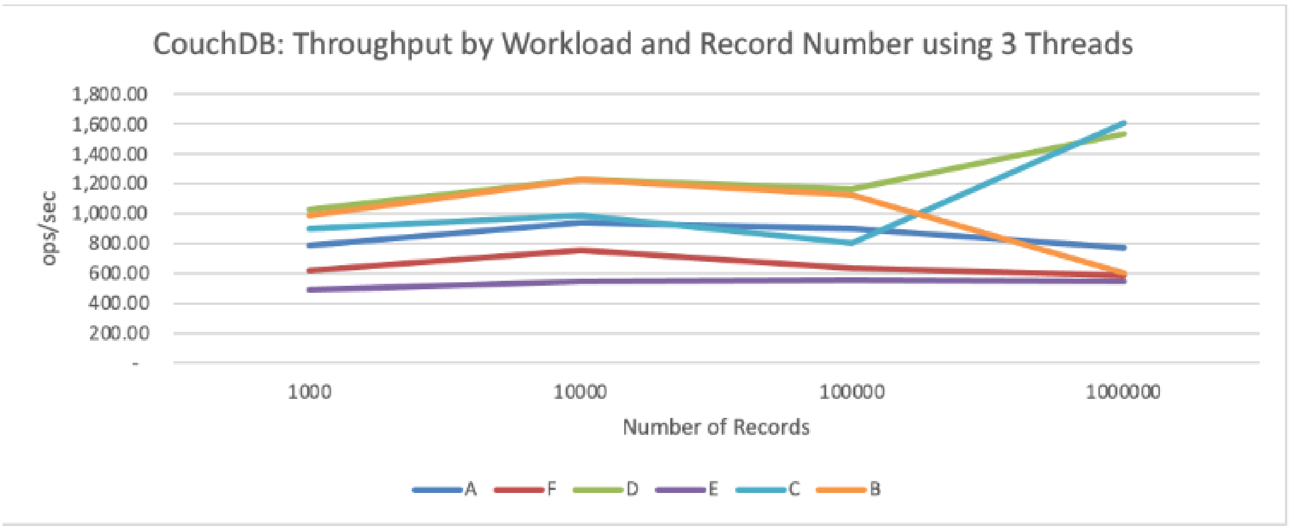
```
#!/bin/bash

# Define variables
WORKLOADS=("workloada" "workloadb" "workloadc" "workloadd" "workloade" "workloadf")
RECORD_COUNTS=(1000 10000 100000 1000000)
THREAD_COUNTS=(3 6)
OUTPUT_DIR="Results"
SCRIPT="./bin/ycsb"
SLEEP_DURATION=10

# Ensure output directory exists
mkdir -p "$OUTPUT_DIR"

# Main loop to iterate over workloads, record counts, and thread counts
for record_count in "${RECORD_COUNTS[@]}"; do
  for workload in "${WORKLOADS[@]}"; do

    LOAD_FILE="${OUTPUT_DIR}/${workload}-load-${record_count}-1.dat"
    echo "Load ${record_count} records for ${workload}"
    $SCRIPT load couchdb -P workloads/$workload -p hosts="127.0.0.1" -p url="http://
   Admin:password@127.0.0.1:5984" -p recordcount=$record_count -p threadcount=1 -s >
    "$LOAD_FILE" 2>&1

    for thread_count in "${THREAD_COUNTS[@]}"; do
      # Construct file names for this configuration
      RUN_FILE="${OUTPUT_DIR}/${workload}-run-${record_count}-${thread_count}.dat"
      # LOG_FILE="${OUTPUT_DIR}/${workload}-log-${record_count}-${thread_count}.txt"

      # Run the script with the current configuration
      echo "Running workload: $workload, Record count: $record_count, Threads:
   $thread_count"
      $SCRIPT run couchdb -P workloads/$workload -p hosts="127.0.0.1" -p url="http://
   Admin:password@127.0.0.1:5984" -p recordcount=$record_count -p operationcount=
   $record_count -p threadcount=$thread_count  -s > "$RUN_FILE"  2>&1


    done
    echo "Delete DB of ${record_count} records for ${workload}"
    curl -X DELETE http://Admin:password@127.0.0.1:5984/usertable

    echo "Sleeping for $SLEEP_DURATION seconds..."
    sleep "$SLEEP_DURATION"
    echo "Done."
  done
done

echo "All runs completed. Results are saved in $OUTPUT_DIR."
```

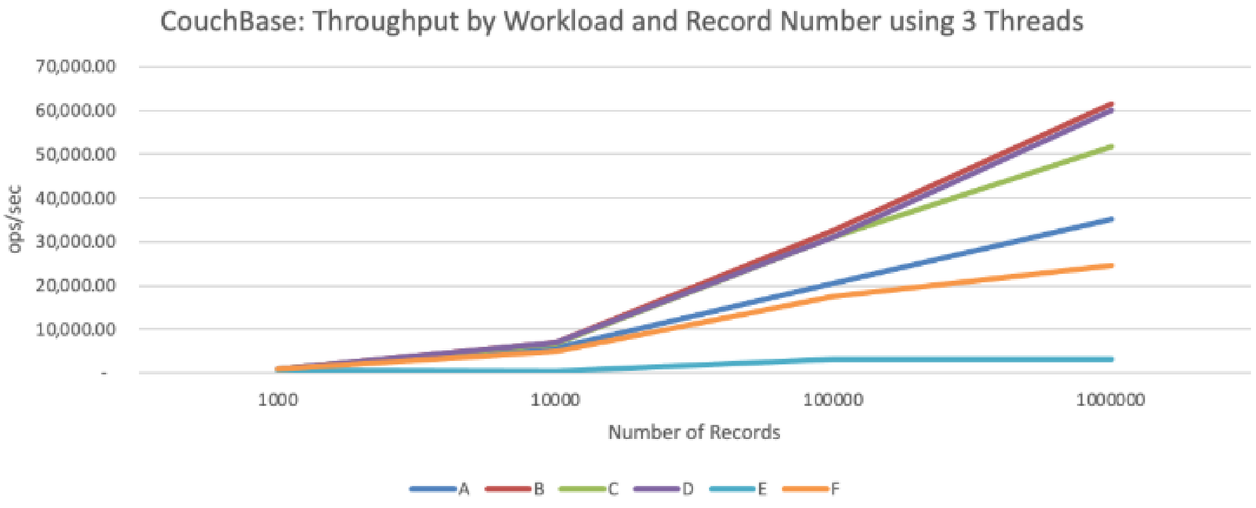
4.8 Running YCSB with Relational Database (PostgreSQL)

We used the other team's results to benchmark with the relational model. Kindly Refer to their full report for more details. A summary of their results is also attached in the results section.

5 Benchmarking Results and Analysis

5.1 Results Analysis: CouchBase vs. CouchDB

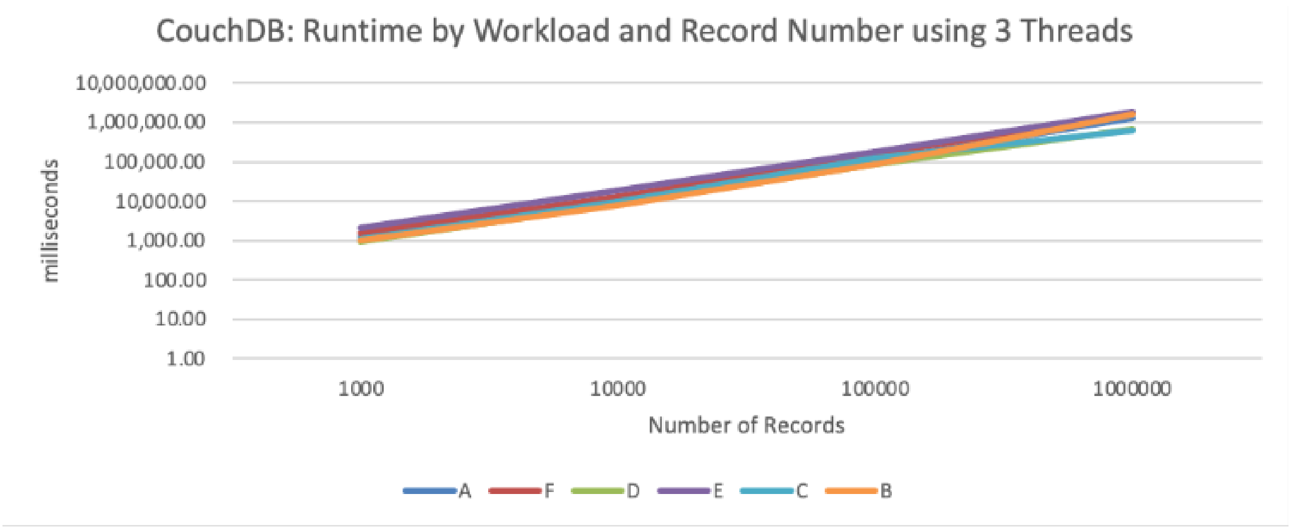


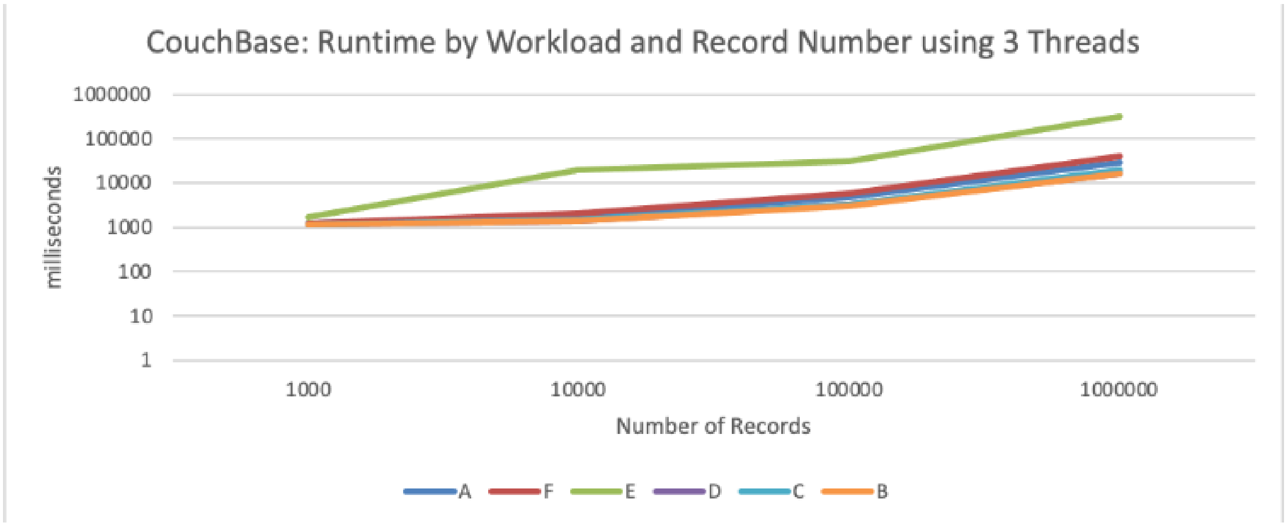


From the charts above, it is evident that CouchBase significantly outperforms CouchDB in terms of throughput, demonstrating an exponential growth trend as the scale factors increase. For CouchBase, the exponential scalability is particularly noticeable, with throughput reaching approximately 60,000 operations per second for workloads B and D, which are among the most read-intensive operations. Additionally, workload C—focused entirely on reads—achieved 50,000 operations per second, showcasing CouchBase's efficiency in handling read-heavy scenarios. Mixed workloads A and F, which include 50% read operations, also exhibited impressive scalability, with throughput reaching 35,000 and 24,000 operations per second, respectively. Conversely, for workload E, which is the most scan-intensive, CouchBase achieved a modest throughput of under 4,000 operations per second, yet this still doubles CouchDB's performance on the same workload.

In contrast, CouchDB demonstrated a relatively constant throughput, averaging 887 operations per second. Workloads C and D stood out with nearly twice the average throughput, as they are highly read-intensive. However, an unexpected 50% decrease in throughput was observed for workload B, despite its read-heavy nature, which diverges from expected behavior. This anomaly highlights potential inefficiencies in CouchDB's handling of concurrent or high-volume read operations.

Overall, CouchBase exhibits exceptional scalability for read-intensive operations as dataset sizes grow, emphasizing its ability to manage large-scale data and high-concurrency demands effectively. For scan-intensive operations, both CouchBase and CouchDB deliver relatively modest throughput, although CouchBase consistently outperforms CouchDB in terms of performance and scalability, further solidifying its advantage.

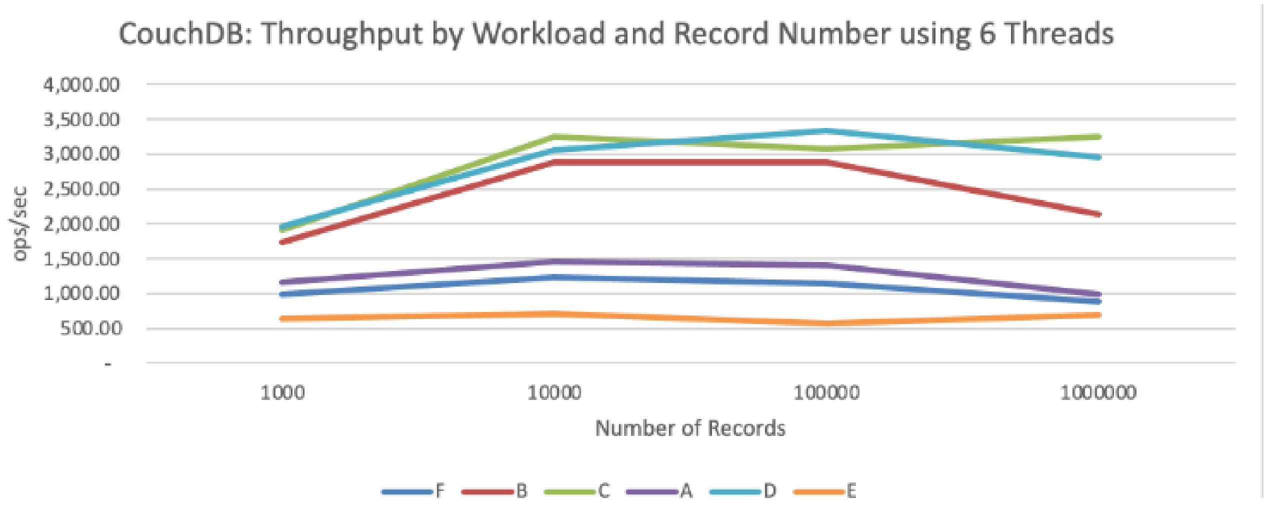


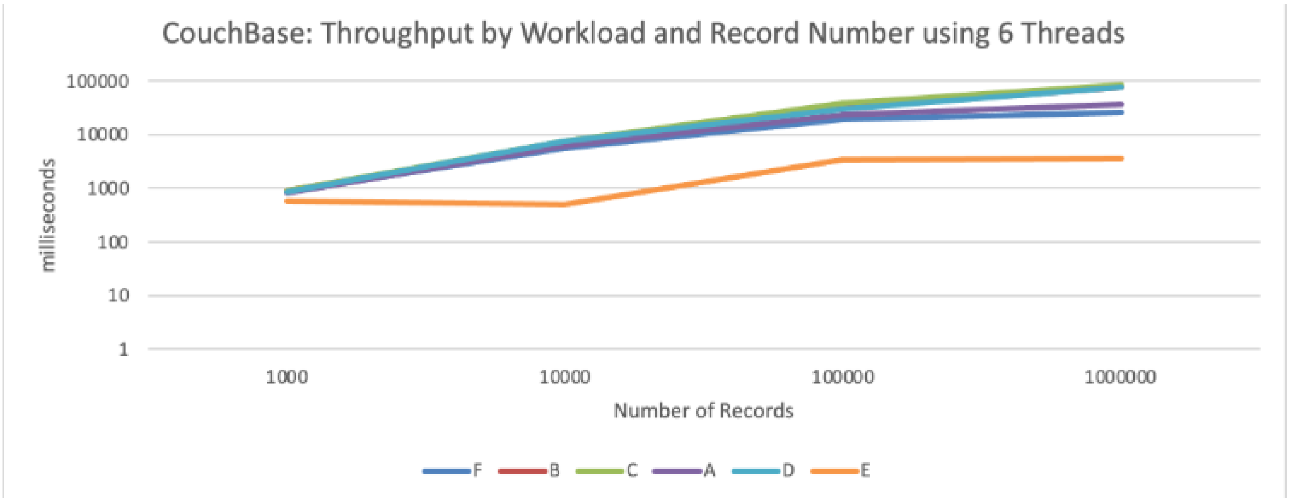


Runtime Performance

CouchBase significantly outperforms CouchDB in terms of runtime, benefiting from its exceptional throughput scalability. For CouchBase, most workloads demonstrate low runtime requirements, typically around 25,000 milliseconds. The exception is workload E, the most scan-intensive workload, which shows an exponential growth trend in runtime, peaking at 306,133 milliseconds.

On the other hand, CouchDB exhibits a linear growth trend in runtime as the number of records increases exponentially. This results in runtimes reaching around 1,000,000 milliseconds, primarily due to CouchDB's static throughput capabilities, which limit its efficiency in handling operations at scale.





Thread Scaling Performance

With 6 threads, the throughput results align with earlier trends. CouchBase continues to handle the exponentially growing number of records with excellent scalability, achieving up to 86,000 operations per second. In contrast, CouchDB maintains a steady but significantly lower throughput, capped at approximately 3,500 operations per second. For CouchDB, this doubling of resources led to a 2.1x improvement in overall throughput, rising from an average of 887 operations per second with 3 threads to 1,848 operations per second with 6 threads.

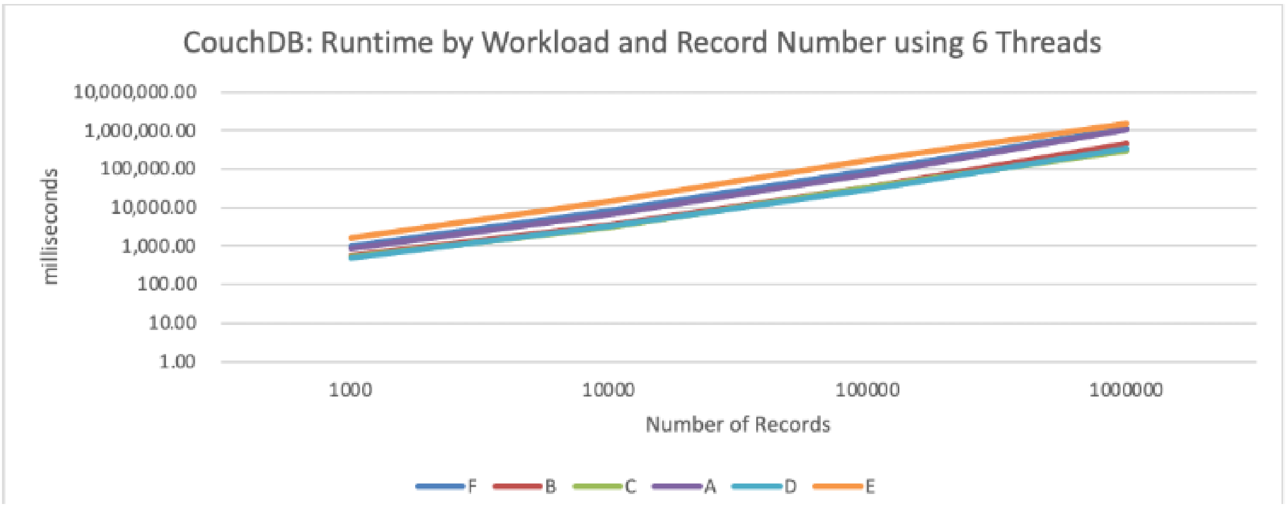
The doubling of resources for CouchBase resulted in only a 20% overall throughput increase, with averages rising from 17,049 operations per second at 3 threads to 20,522 operations per second at 6 threads. Notably, for CouchBase, workload E exhibits similar behavior to the 3-thread scenario, reaching up to 3,600 operations per second, indicating that doubling the resources had no impact on the performance of this scan-intensive workload. However, for all other workloads in CouchBase, a noticeable improvement is observed between 3 and 6 threads. Despite this improvement, CouchDB's performance remains far below CouchBase's.

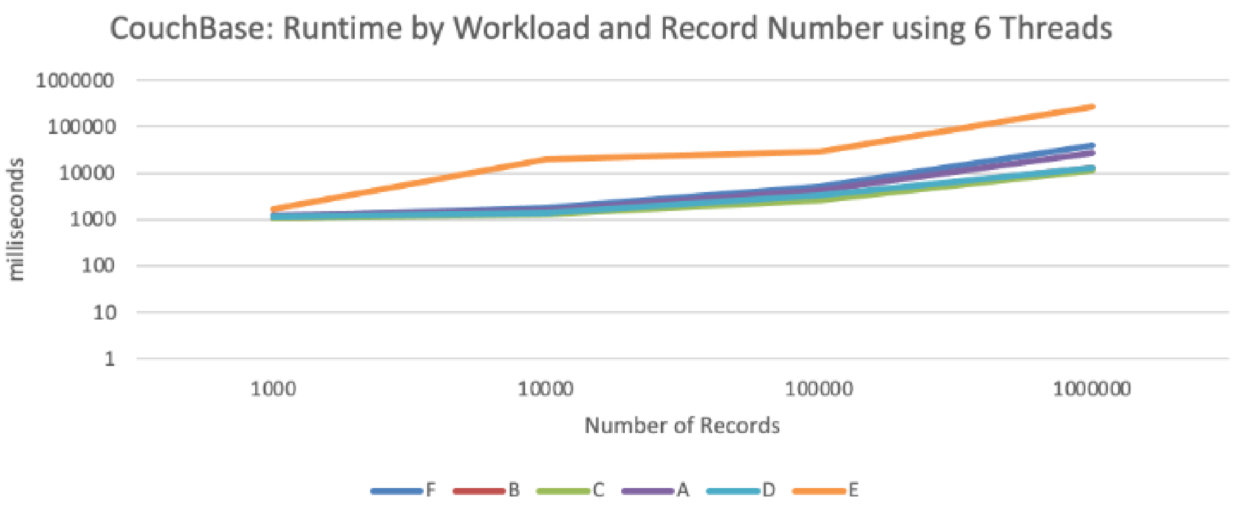
Runtime Trends

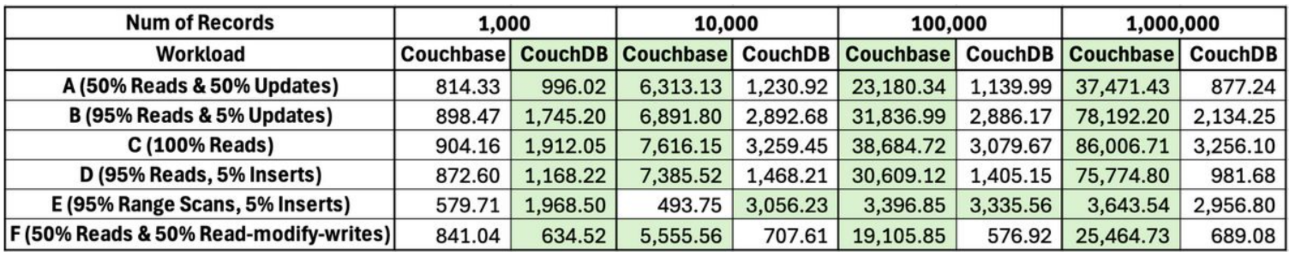
The runtime trends are a direct reflection of throughput performance. Although the resources were doubled with the use of 6 threads, the average runtime of CouchBase decreased by only 10%, while CouchDB achieved a more significant 40% reduction in average runtime.

Analyzing the results for 6 threads reveals that CouchDB demonstrated superior throughput for the smallest dataset, achieving an average throughput 71% higher than Couchbase. However, as the dataset size increased, Couchbase exhibited significantly higher throughput compared to CouchDB, highlighting its scalability advantage.

In the case of CouchBase, scalability is impressive as the number of records increases exponentially, with an average runtime of 19,219 milliseconds. However, despite this strong performance, workload E remains







Num of Records	1,000		10,000		100,000		1,000,000	
Workload	Couchbase	CouchDB	Couchbase	CouchDB	Couchbase	CouchDB	Couchbase	CouchDB
A (50% Reads & 50% Updates)	814.33	996.02	6,313.13	1,230.92	23,180.34	1,139.99	37,471.43	877.24
B (95% Reads & 5% Updates)	898.47	1,745.20	6,891.80	2,892.68	31,836.99	2,886.17	78,192.20	2,134.25
C (100% Reads)	904.16	1,912.05	7,616.15	3,259.45	38,684.72	3,079.67	86,006.71	3,256.10
D (95% Reads, 5% Inserts)	872.60	1,168.22	7,385.52	1,468.21	30,609.12	1,405.15	75,774.80	981.68
E (95% Range Scans, 5% Inserts)	579.71	1,968.50	493.75	3,056.23	3,396.85	3,335.56	3,643.54	2,956.80
F (50% Reads & 50% Read-modify-writes)	841.04	634.52	5,555.56	707.61	19,105.85	576.92	25,464.73	689.08

the slowest, reaching up to 274,458 milliseconds in the 1-million-record test.

For CouchDB, a linear growth trend is observed, similar to the results with 3 threads, as the number of records grows exponentially, with runtime peaking at 1.1 million milliseconds. Despite the 40% overall decrease in average runtime, the performance remains too high for large datasets.

5.2 Workload Analysis for Booking.com Use Case

To understand how both databases perform in the context of booking.com, we analyze how the empirical results apply to real-world operations, which involve frequent updates to room availability, search queries for hotels or flights, dynamic content, range scans, and read-modify-write operations.

Workload A: Update Heavy (50% Reads, 50% Updates)

CouchBase: CouchBase shows superior performance in Workload A with a throughput of up to 37,471.43 operations per second (at 1,000,000 records). As the number of operations increases, the performance remains high, indicating that CouchBase efficiently handles both read and update operations. The runtime of 26,687 ms for 1 million records is relatively low, making CouchBase well-suited for scenarios requiring a balance of frequent reads and updates, like updating room availability in a booking system.

CouchDB: CouchDB's performance significantly drops for larger datasets, with a throughput of only 877.24 operations per second (at 1,000,000 records) and a runtime of 1,018,666 ms, indicating that it struggles to handle large-scale update operations efficiently.

Workload B: Read Heavy (95% Reads, 5% Updates)

CouchBase: CouchBase excels in read-heavy workloads, with a throughput of 78,192.20 operations per second (at 1,000,000 records) and a runtime of 12,789 ms. It handles large-scale read operations efficiently, which makes it ideal for search operations like browsing hotel availability.

CouchDB: While CouchDB can handle read-heavy workloads, its performance is significantly lower than CouchBase's, with a throughput of only 2,134.25 operations per second at 1 million records and runtime of 468,548 ms, which is much higher than CouchBase's performance.

Workload C: Read Only (100% Reads)

CouchBase: CouchBase shows exceptional performance with 100% read workloads, with a throughput of 86,006.71 operations per second (at 1,000,000 records) and a runtime of 11,627 ms. This is ideal for read-only operations, such as displaying product recommendations or user reviews.

CouchDB: CouchDB's throughput significantly lags behind CouchBase's in larger datasets, with a throughput of only 3,079.67 operations per second and a runtime of 307,116 ms for 1 million records, which indicates that it becomes inefficient at larger scales.

Workload D: Read Latest (95% Reads, 5% Inserts Targeting Recently Inserted Data)

CouchBase: CouchBase performs exceptionally well in read-latest scenarios, with a throughput of 75,774.80 operations per second (at 1,000,000 records) and a runtime of 13,197 ms. It efficiently manages both read operations and recent data inserts, making it suitable for dynamic content like new hotel listings or price updates.

CouchDB: CouchDB's performance drops for large datasets, with a throughput of only 2,956.80 operations per second and a runtime of 1,451,216 ms at 1 million operations, making it much slower than CouchBase.

Workload E: Short Range Scans (95% Range Scans, 5% Inserts)

CouchBase: While CouchBase's throughput for short-range scans is significantly lower than any other workload, with 3,643.54 operations per second (at 1,000,000 records) and a very slow runtime of 274,458 ms, it still outperforms CouchDB in terms of scalability. It can handle large datasets with moderate efficiency, suitable for browsing hotels by price or rating.

CouchDB: Despite its better performance on range scans than CouchBase, CouchDB's throughput of 689.08 operations per second for 1,000,000 records and runtime of 1,451,216 ms for 1 million records make it unsuitable for large-scale scan-heavy operations.

Workload F: Read-Modify-Write (50% Reads, 50% Read-Modify-Writes)

CouchBase: CouchBase performs well in mixed read-modify-write workloads with a throughput of 25,464.73 operations per second (at 1,000,000 records) and a runtime of 39,270 ms. It efficiently handles both read and write operations, making it suitable for scenarios where booking details or customer preferences need frequent updates.

CouchDB: CouchDB's throughput significantly lags behind CouchBase's as its throughput of 887.24 operations per second and runtime of 1,139,940 ms for 1 million records indicates poor scalability for large datasets and mixed workloads.

Benchmarking Additional Operations Using YCSB

While the YCSB benchmark does not specifically include operations such as Scan, Search, Page, NestScan, Aggregate, or Report required for the booking.com case study, we can leverage its results to gain insights into the performance of these operations. YCSB's focus on workloads like read-heavy and read-only operations provides a useful baseline for understanding how databases handle large-scale data retrieval. By analyzing the results from Workload C: Read-Only (100

Workload C focuses purely on read-heavy operations, where the system performs only retrieval tasks without any write or update operations. In this scenario, the database's ability to handle large volumes of read requests is crucial, as it simulates the performance of databases during high-traffic conditions such as searching, scanning, or aggregating large datasets.

- **Couchbase results from Workload C** demonstrate a throughput of 86,006.71 operations per second and a runtime of 11,627 ms for 1 million records. These results suggest that Couchbase excels at handling large-scale, read-heavy workloads. This high throughput and low runtime indicate that Couchbase is well-optimized for executing complex queries across vast datasets, making it suitable for applications requiring quick access to large volumes of data, such as accommodation searches or product listings.

- **On the other hand, CouchDB** shows a significantly lower throughput of 3,079.67 operations per second with a much higher runtime of 307,116 ms for 1 million records. These results highlight that CouchDB struggles with large-scale read operations, becoming increasingly inefficient as the dataset grows.
- Since Workload C represents purely reading data, it provides a useful reference for the performance of other operations involving substantial data retrieval, such as Scan, Search, Page, NestScan, Aggregate, and Report. These operations typically rely on the database's ability to efficiently retrieve and process data. Therefore, Workload C is a valuable reference for evaluating their performance in real-world scenarios:
  - **Scan**: Browsing all accommodations in a location with LIMIT for pagination involves retrieving large sets of data in a specific order. Since this operation requires many reads, Couchbase will be more efficient, processing higher throughput with lower latency even for large datasets.
  - **Search**: Advanced filtering, such as querying by price range, rating, or availability, involves a mix of read-heavy operations with additional conditions. Couchbase's high throughput indicates that it can handle these complex queries efficiently, whereas CouchDB's lower throughput suggests slower performance for these kinds of queries, especially as datasets scale.
  - **Page**: Paginated results, especially those requiring OFFSET and LIMIT, rely on the database's ability to perform efficient reads. Couchbase excels in this regard, handling pagination well even with large datasets, while CouchDB may experience slower performance as the dataset size increases.
  - **NestScan**: Querying nested fields, such as reviews or amenities, involves read-heavy operations. The results from Workload C show that Couchbase can efficiently process these types of queries, even when the data is nested within documents. In contrast, CouchDB struggles with nested queries, as its performance degrades with increasing dataset size.
  - **Aggregate**: Grouping data (e.g., by city or calculating averages) often involves complex aggregation queries that require the database to scan large portions of the dataset. Given Couchbase's strong performance in read-heavy scenarios, it would perform well on aggregate queries, handling large datasets efficiently. CouchDB, however, will be slower for such operations due to its lower throughput and higher runtime.
  - **Report**: Generating reports based on time frames or locations involves filtering large datasets and summarizing the results. Couchbase's high throughput for reads ensures that it can generate

detailed reports quickly, while CouchDB may struggle to provide fast report generation as the dataset grows.

Based on the empirical results from Workload C, it is evident that Couchbase consistently outperforms CouchDB in read-heavy operations, especially when dealing with large datasets. Couchbase's high throughput and low latency make it an ideal choice for operations that rely on reading large volumes of data quickly, such as scan, search, pagination, NestScan, aggregation, and reporting. While CouchDB may still be suitable for smaller, less complex workloads, Couchbase is the superior option for large-scale applications requiring efficient read operations. If your use case involves large datasets and frequent read-heavy operations, Couchbase is the better choice due to its scalability and performance in handling large-scale, data-intensive queries.

Postgres Performance Test

The performance tests run by our colleagues using PostgreSQL on the DBLP dataset reveal some interesting insights about PostgreSQL's capabilities as a document store:

- **Query 6:** `add_publication_and_related_data` demonstrates stable execution times across scales, maintaining an O(1) insertion time regardless of the dataset size. This indicates that PostgreSQL handles simple data insertions efficiently without significant performance degradation.
- The similarity in execution times across different scales suggests that PostgreSQL's relational model can handle document-oriented tasks without a drastic increase in execution time, even with growing dataset sizes.
- PostgreSQL's performance remains stable as the scale increases, making it a reliable choice for document storage, especially for operations that involve inserting and appending data to the database.
- PostgreSQL's referenced model showcases superior scalability and efficiency for complex queries, benefiting from its mature query optimization and index ing capabilities. Their testing on the DBLP dataset reaffirmed the benchmarking results, showing PostgreSQL's steady performance for structured data PostgreSQL is ideal for applications demanding high data integrity, complex relationships, and analytical querying, such as transactional systems or data warehouses

6 Additional Queries

To cover the gaps in benchmarking the two DBs with YCSB, we managed to implement and run additional queries, using the data of Travel and Hotel Listing from Booking.com 2020 available here.

6.1 Dataset

The dataset contains around 30,000 documents for hotel listings, and a sample record is as following:

```
{
    "pageurl": "https://www.booking.com/hotel/in/treebo-trip-daisey-dee.en-gb.html?
   label=gen173nr-1
   DCAQoggJCDHNlYXJjaF9jb29yZ0gJWARolQKIAQGYAQm4ARjIAQ_YAQPoAQH4AQOIAgGoAgS4ArzlsvMFwAIB
   &sid=b369ec9535ceb9bef18e93d6d1377489&all_sr_blocks=513689101_179845172_2_1_0&
   checkin=2020-03-16&checkout=2020-03-17&dest_id=7467&dest_type=region&group_adults
   =2&group_children=0&hapos=27&highlighted_blocks=513689101_179845172_2_1_0&hpos=2&
   no_rooms=1&sr_order=popularity&sr_pri_blocks=513689101_179845172_2_1_0__119461&
   srepoch=1584181959&srpvid=ce584a239277011d&ucfs=1&from=searchresults;
   highlight_room=&;selected_currency=INR;changed_currency=1;top_currency=1",
    "record": {
        "uniq_id": "0ef15af30235dc2a267ec37ff9c5c0ee",
        "hotel_id": "5136891",
        "hotel_name": "Treebo Trip Daisey Dee",
        "review_count": "27",
        "rating_count": "8.8",
        "default_rank": "30",
        "price_rank": "37",
        "ota": "booking.com",
        "room_type": [
            {
                "room_type_name": "Standard Double Room",
                "room_type_price": 1338,
                "room_type_occupancy": 2,
                "room_type_breakfast": "breakfast",
                "room_type_cancellation": "free_cancellation",
                "availability": [
                    {
                        "from": "2025-01-20",
                        "to": "2025-01-25"
                    },
                    {
                        "from": "2025-01-12",
                        "to": "2025-01-14"
                    },
                    {
                        "from": "2025-01-01",
                        "to": "2025-01-07"
                    }
                ]
            },
            {
                "room_type_name": "Standard Double Room",
                "room_type_price": 1136,
                "room_type_occupancy": 1,
                "room_type_breakfast": "breakfast",
                "room_type_cancellation": "free_cancellation",
                "availability": [
                    {
                        "from": "2025-02-06",
                        "to": "2025-02-11"
                    },
                    {
                        "from": "2025-02-04",
                        "to": "2025-02-09"
                    },
                    {
                        "from": "2025-02-02",
                        "to": "2025-02-05"
                    }
                ]
            }
        ],
        "checkin_date": "2020-03-16",
        "crawled_date": "2020-03-14 10:59:45 +0000",
        "city": "Paris"
    }
}
```

6.2 Business Questions and Queries

1. Search

Business Question 1: Retrieve all accommodations with a nightly price between 100 **and** 200**.**

CouchDB Query:

```
{
  "selector": {
    "record.room_type": {
      "$elemMatch": {
        "room_type_price": {
          "$gte": 100,
          "$lte": 200
        }}}},
  "limit": 1000
}

Runtime: 7.691s
```

Couchbase Query:

```
SELECT *
FROM accommodations
WHERE ANY room IN record.room_type SATISFIES room.room_type_price BETWEEN 100 AND 200
     END;

Runtime: 1.951s
```

Business Question 2: Find accommodations with an average rating of 4.5 or higher.

CouchDB Query:

```
{
  "selector": {
    "record.rating_count": { "$gte": 4.5 }
  },
  "limit": 20000
}

Runtime: 12.55s
```

Couchbase Query:

```
SELECT *
FROM _default
```

```
WHERE TO_NUMBER(record.rating_count) >= 4.5;

Runtime:6.497s
```

Business Question 3: Search for accommodations that are available between "2024-01-01" and "2024-01-15".

CouchDB Query:

```
{
  "selector": {
    "record.room_type": {
      "$elemMatch": {
        "availability": {
          "$elemMatch": {
            "from": {
              "$lte": "2025-01-15"
            },
            "to": {
              "$gte": "2025-01-01"
            }}}}}}}

Runtime:11.55s
```

Couchbase Query:

```
SELECT *
FROM _default
WHERE ANY room IN record.room_type SATISFIES
      ANY avail IN room.availability
      SATISFIES avail.‘from‘ <= "2025-01-15" AND avail.‘to‘ >= "2025-01-01"
      END
END;

Runtime: 9.288 s
```

2. Page

Business Question 1: Show the first 10 accommodations sorted by price.

CouchDB Query:

```
{
  "index": {
    "fields": ["record.room_type.0.room_type_price"]
  },
  "name": "room_type_price_index",
  "type": "json"
}

{
  "selector": {},
  "sort": [
    { "record.room_type.0.room_type_price": "asc" }
  ],
  "limit": 10
}

Runtime: 7.89s
```

Couchbase Query:

```
SELECT *
FROM _default
ORDER BY record.room_type[0].room_type_price ASC
LIMIT 10;

Runtime: 3.2634s
```

Business Question 2: Display the second page of accommodations with 5 results per page.

CouchDB Query:

```
{
  "selector": {},
  "sort": [
    {
      "record.room_type.0.room_type_price": "asc"
    }
  ],
  "limit": 5,
  "skip": 5
}

Runtime:  6.56s
```

Couchbase Query:

```
SELECT *
FROM _default
ORDER BY record.room_type[0].room_type_price ASC
LIMIT 5 OFFSET 5;
```

```

Runtime: 3.25110s
```

3. Aggregate

Business Question 1: Calculate the minimum nightly price of accommodations by city.

CouchDB Query: (Requires creating a view)

```
First, create map function and view

curl -X PUT "http://Admin:password@localhost:5984/booking/_design/aggregations" \
-H "Content-Type: application/json" \
-d '{
  "_id": "_design/aggregations",
  "views": {
    "min_price_by_city": {
      "map": "function (doc) { if (doc.record && doc.record.room_type && doc.record.
    city) { doc.record.room_type.forEach(function(room) { if (room.room_type_price &&
     !isNaN(parseFloat(room.room_type_price))) { emit(doc.record.city, parseFloat(
    room.room_type_price)); } }); } }",
      "reduce": "_stats"
    }
  }
}'

Then, get city aggregations

curl -X GET "http://Admin:password@localhost:5984/booking/_design/aggregations/_view/
    min_price_by_city?reduce=true&group=true"

Runtime: 7.99s
```

Couchbase Query:

```
SELECT city, MIN(room_type_price) AS min_price
FROM _default
GROUP BY city;

Runtime: 5.7688s
```

Business Question 2: Count the number of accommodations available in each city

CouchDB Query:

Requires creating a view

```
time curl -X PUT "http://Admin:password@localhost:5984/booking/_design/city_counts" \
-H "Content-Type: application/json" \
-d '{
  "_id": "_design/city_counts",
  "views": {
    "count_by_city": {
      "map": "function(doc) { if (doc.record && doc.record.city) { emit(doc.record.city, 1); } }",
      "reduce": "_count"
    }
  }
}'

Then
time curl -X GET "http://Admin:password@localhost:5984/booking/_design/city_counts/_view/count_by_city?reduce=true&group=true"

Runtime: 9.04s
```

Couchbase Query:

```
SELECT record.city, COUNT(*) AS accommodation_count
FROM _default
GROUP BY record.city;

Runtime: 5.81386s
```

Business Question 3: Group accommodations by rating and count the total in each group.

CouchDB Query:

Requires creating a view

```
time curl -X PUT "http://Admin:password@localhost:5984/booking/_design/rating_counts" \
-H "Content-Type: application/json" \
-d '{
  "_id": "_design/rating_counts",
  "views": {
    "count_by_rating": {
      "map": "function(doc) { var rating = doc.record && doc.record.rating_count ? parseFloat(doc.record.rating_count) : null; emit(rating, 1); }",
      "reduce": "_count"
    }
  }
}'

```

```
then

time curl -X GET "http://Admin:password@localhost:5984/booking/_design/rating_counts/
    _view/count_by_rating?reduce=true&group=true"

Runtime: 8.47s
```

Couchbase Query:

```

SELECT TO_NUMBER(record.rating_count) AS rating, COUNT(*) AS rating_count
FROM _default
GROUP BY TO_NUMBER(record.rating_count);

Runtime: 5.741s
```

5. Report

Business Question 1: Generate a report of bookings made in December 2024.

CouchDB Query:

```
{
  "selector": {
    "record.checkin_date": {
      "$gte": "2024-12-01",
      "$lte": "2024-12-31"
    }
  }
}

Runtime: 4.8s
```

Couchbase Query:

```
 SELECT *
FROM _default
WHERE record.checkin_date BETWEEN "2024-12-01" AND "2024-12-31";

Runtime: 2.95s
```

7 Conclusion

In chapter 3 we provided our hypothesis for the expected performance of both tools regarding the core workload.

Metric	CouchBase	CouchDB
Throughput	High	Low
Latency	Low	High
Consistency	High	Low
Scalability	High	High

Table 7.1: Expected Result for Key Metrics

After performing the tests the actual results are as follows:

Metric	CouchBase	CouchDB
Throughput	High	Low
Latency	Low	Mid
Consistency	High	High
Scalability	High	Low

Table 7.2: Actual Results for Key Metrics

CouchBase's ability to scale effectively across various workloads highlights its suitability for enterprise-level applications like Booking.com. Its impressive performance in read-heavy, read-modify-write, and update-heavy scenarios makes it a reliable choice for high-concurrency environments requiring large dataset management. In contrast, CouchDB's consistent but significantly lower performance makes it more appropriate for smaller-scale applications with lower concurrency demands. CouchBase's superior performance metrics, scalability, and runtime efficiency establish it as the optimal choice for dynamic and data-intensive use cases. Moreover, it achieved better runtime for the additional queries covering the other areas like Search, and Aggregate, which also proves its suitability for applications like booking.com.

Whole bibliography

[Bus] Busbey, Sean (n.d.). *Core Workloads.* URL: https://github.com/brianfrankcooper/YCSB/wiki/Core-Workloads. (accessed: 28.11.2024).

[Foua] Foundation, The Apache Software (n.d.[a]). *3.2.4. View Cookbook for SQL Jockeys¶.* URL: https://docs.couchdb.org/en/stable/ddocs/views/nosql.html. (accessed: 28.11.2024).

[Foub] — (n.d.[b]). *Cluster Management.* URL: https://docs.couchdb.org/en/stable/cluster/theory.html. (accessed: 27.11.2024).

[Fouc] — (n.d.[c]). *Introduction.* URL: https://cwiki.apache.org/confluence/display/COUCHDB/Introduction. (accessed: 27.11.2024).

[Foud] — (n.d.[d]). *The Power of B-trees.* URL: https://guide.couchdb.org/draft/btree.html. (accessed: 28.11.2024).

[IBM] IBM (n.d.). *What is a NoSQL database?* URL: https://www.ibm.com/topics/nosql-databases. (accessed: 15.11.2024).

[Inca] Inc., Amazon (n.d.[a]). *What Is a Document Database?* URL: https://aws.amazon.com/nosql/document/. (accessed: 15.11.2024).

[Incb] Inc., CouchBase (n.d.[b]). *Buckets.* URL: https://docs.couchbase.com/server/current/learn/buckets-memory-and-storage/buckets.html. (accessed: 12.11.2024).

[Incc] — (n.d.[c]). *Manage Nodes and Clusters.* URL: https://docs.couchbase.com/server/current/manage/manage-nodes/node-management-overview.html. (accessed: 12.11.2024).

[Incd] — (n.d.[d]). *Querying With N1QL.* URL: https://docs.couchbase.com/files/Couchbase-N1QL-CheatSheet.pdf. (accessed: 19.11.2024).

[Ince] — (n.d.[e]). *Run Your First SQL++ Query.* URL: https://docs.couchbase.com/server/current/getting-started/try-a-query.html. (accessed: 19.11.2024).

[Incf] — (n.d.[f]). *Scopes and Collections.* URL: https://docs.couchbase.com/server/current/learn/data/scopes-and-collections.html. (accessed: 12.11.2024).

[Incg] Inc., CouchBase (n.d.[g]). *Take Database Offline/Online.* URL: `https://docs.couchbase.com/sync-gateway/current/database-offline.html`. (accessed: 30.11.2024).

[Inch] — (n.d.[h]). *Types of Primary and Secondary Index.* URL: `https://docs.couchbase.com/server/current/learn/services-and-indexes/indexes/indexing-and-query-perf.html`. (accessed: 24.11.2024).

[Mur] Murthy, Keshav (n.d.). *Using YCSB to Benchmark JSON Databases.* URL: `https://www.couchbase.com/blog/ycsb-json-benchmarking-json-databases-by-extending-ycsb/`. (accessed: 02.12.2024).

[Sli] Slingerland, Cody (n.d.). *Horizontal Vs. Vertical Scaling: Which Should You Choose?* URL: `https://www.cloudzero.com/blog/horizontal-vs-vertical-scaling/`. (accessed: 15.11.2024).

[Uni] University, Stanford (n.d.). *Balanced Tress: Part One.* URL: `https://web.stanford.edu/class/archive/cs/cs166/cs166.1146/lectures/02/Small02.pdf`. (accessed: 16.11.2024).

[Unk] Unknown (n.d.). *Insert Operation in B-Tree.* URL: `https://www.geeksforgeeks.org/insert-operation-in-b-tree/`. (accessed: 30.11.2024).