Implementation of Slack Backend

Brinda Dhawan, Marco Tortolani, Emily Wang, Xinyu Wu, Jason Zhang

Northeastern University, Boston, MA, USA

Project Github Repository: https://github.com/Mtortolani/slack-backend

Abstract

Slack is a popular communications platform that has workspaces, which contain channels and direct messages where users interact with one another. In this project, we use Python to replicate Slack's backend and performance test two different databases — MongoDB and Apache Spark — and their ability to query and retrieve randomized data. Based on our preconceptions, we expected Spark to be faster, but MongoDB ended up being significantly more efficient since Spark relies on Spark SQL Tables and therefore required more time to convert the MongoDB document-style database to SQL table-style data.

Introduction

Slack is a communications platform primarily used by small businesses and organizations. It allows groups to communicate through public and private channels and direct messages. Slack currently hosts 10+ million active users. Some available features include channels, messaging, and voice or video calls. In this project, we attempted to replicate Slack's backend data architecture and basic functionalities as well as optimize the speed of interactions and data processing that Slack currently has. With the massive scale that Slack operates on, processing terabytes of data from its active users requires an advanced and robust system. Therefore, our project focused on creating a clone of their solution which simultaneously supports the necessary features while also ensuring safe and secure data access. Using both Apache Spark and MongoDB to test the speed of queries of Slack data, we initially expected Spark to be faster in running queries. However, this hypothesis was proven incorrect through our performance testing. Ultimately, the work completed in this project is significant in comparing the speed of both SQL and no-SQL databases, particularly in the context of Slack messaging.

Methods

In an attempt to replicate Slack's backend, we first implemented a number of classes with different attributes, which are the most basic 'objects' in Slack. These classes included: Workspace, Channel, User, Settings, and Message. The attributes and types of each class are as follows:

Slack Backend Structure: Classes & Attributes

Workspace	Channel	User	Settings	Message
workspace id: int	channel id: int	user id: int	***Channel Setting***	message id: int
name: str	workspace id: int	username: str	censored words: set	author id: int
members: set of Users	direct channel: bool	profile picture: 'Default Image'	archive: bool (default = False)	channel id: int
roles: dict {owners:[User1]}	private: bool	channels: set of Channels	***User Setting***	content: str
channels: set of Channels	banned: set of Users	direct channels: set of Channels	notifications: bool	
	roles: dict {administrators:[User1]}	settings: Settings class	language: str	
		friends: set of Users	time zone: str	
		blocked: set of Users		

The data utilized in this project was randomly generated as part of the application prototype, so there were no data acquisition processes or pre-processing steps involved. However, we ensured that the essential data was generated with non-trivial types and values. While names, ID numbers, and chat messages were composed of randomized Strings and integers, proper data types such as booleans were generated for the appropriate fields such as Notifications and Archive. The specific random data generation functions and data types are depicted below:

Random Data Generator Functions

Function Name	Output
Generate Random String	Supporting function, returns random string of given size
Generate Random Number	Supporting function, outputs random integer of sig. fig
Generate Random Message	Creates a string of random letters
Generate Random Language	Randomly selects str from list of languages on Slack
Generate Random Timezone	Randomly selects str from list of timezones on Slack
Generate Notifications	Returns boolean, 0 or 1

Generate Random Archive	Returns boolean, 0 or 1
Generate Random User	Uses previous functions to determine random username (string) and settings (language, time zone, notifications
Generate Random Channel	Uses previous functions to create list of random censored words and archive setting
Generate Random Direct Channel	Randomly pulls two existing users to create direct channel and list of random string messages
Generate Random Workspace	Creates random workspace with a pre-defined number of users, channels, and messages in each channel
Random Data Test	Creates a random set of users, workspaces, channels, direct channels, and messages per channel

We chose to use Python as our primary programming language, MongoDB as our fundamental NoSQL database system, and Apache Spark to process faster queries on memory. In order to store the data into a Mongo database, we developed the generator class with functionalities to create random data (using the aforementioned architecture), structure it for a Mongo database, and store it in the database. As listed above, we have functions for creating random messages, users, channels, direct channels, and workspaces. Within these, we randomly define all of the different attributes (names, message content, channel and user settings, censored words, languages, time zones, etc.). Using these random generator functions as part of the Generator class, we then have a separate function (random data test) that iterates through and creates a given number of random users, workspaces, and channels. This is then directly populated into three separate Mongo collections in the following format:

MongoDB Collections Format

User — {user id, name, settings : {notifications, language, timezone}}
Direct Message — {member ids, messages, settings : {censored words, archive}}
Workspace — {name, members, channels : {name, messages}}

To test the efficiency and speed of our database and architectural implementation, we developed a number of queries for both Mongo and Spark. This includes basic functionalities such as searching for a random user or random workspace. Other Mongo queries include: returning all workspaces that a user belongs to, finding and returning all messages in a direct channel between two users, finding the names of all users in a workspace by using their workspace ID, finding all available channels in a random workspace, and finding the names of all available channels in a workspace. In order to implement these same queries using Spark, we had to alter the syntax of each query to be similar to SQL queries, as this is what Spark uses rather than using the native Mongo database structure.

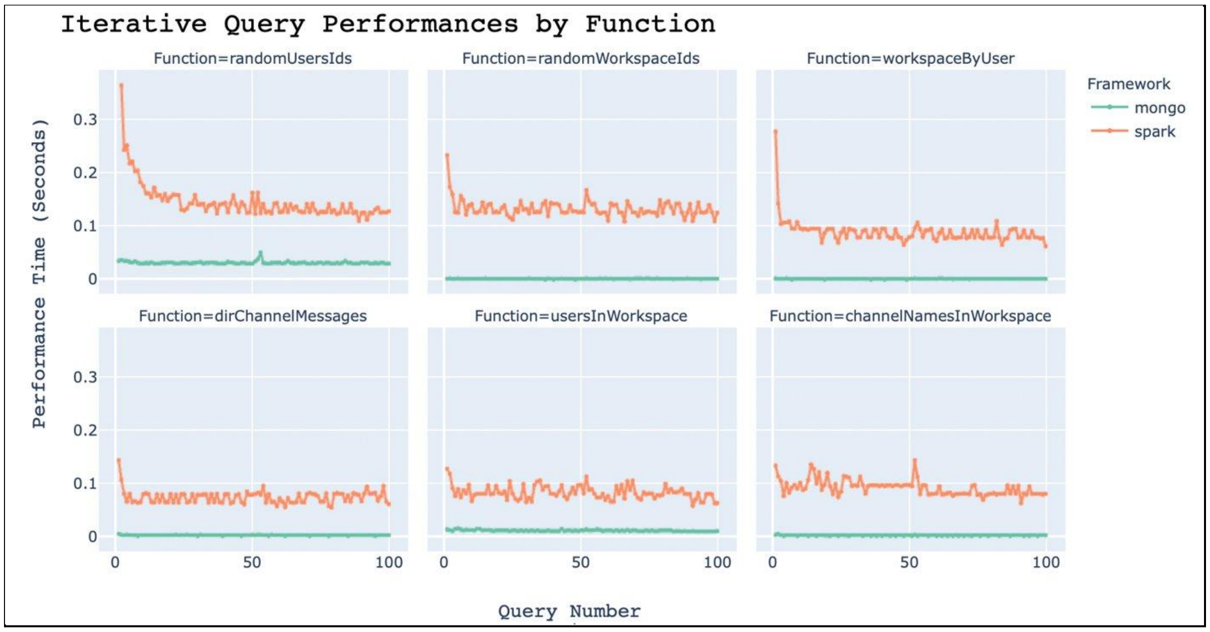
To implement performance testing, for each function we created, such as retrieving all the users in a workspace or retrieving user IDs (named "usersInWorkspace" and "randomUsersIds" respectively as depicted in *Figure 1*), we created a profiler method to repeatedly run each query a specified number of times (which in our case, was one hundred queries), time the queries to find the duration of each query in seconds, and store the resulting performance times within a JSON file. By running the profiler on Mongo queries as well as Spark SQL queries, we were able to compare the performance of both frameworks on the same query functions for an accurate comparison.

Analysis

Spark works best with SQL and is designed for such, our performance suffers because we need to convert our Mongo database to Spark SQL Tables (Spark Dataframe) before running queries, which creates a lot of overhead and negates the performance benefits of Spark's distributed processing. In practice, when Spark converts our Mongo database, which is essentially nested dictionaries, into standard SQL tables, each row ends up storing multiple arrays/structs, which are very impractical. And with a scaled up Mongo database, this leads to a series of nested arrays, which is a highly inefficient data structure.

Our initial plan was to utilize Redis as our NoSQL database due to its unmatched speed and expandability. However, we quickly found that running a key-value store would make searches difficult or impossible as we delved into more nested objects (such as workspaces having channels with settings). Additionally, Redis does not have a premade connector to Spark, making Spark integration difficult. We chose MongoDB as it addresses both of these problems: document-store allows easier searches through nested fields, and the mongo-spark-connector would allow us to connect to our already existing MongoDB database with minimal overhead.

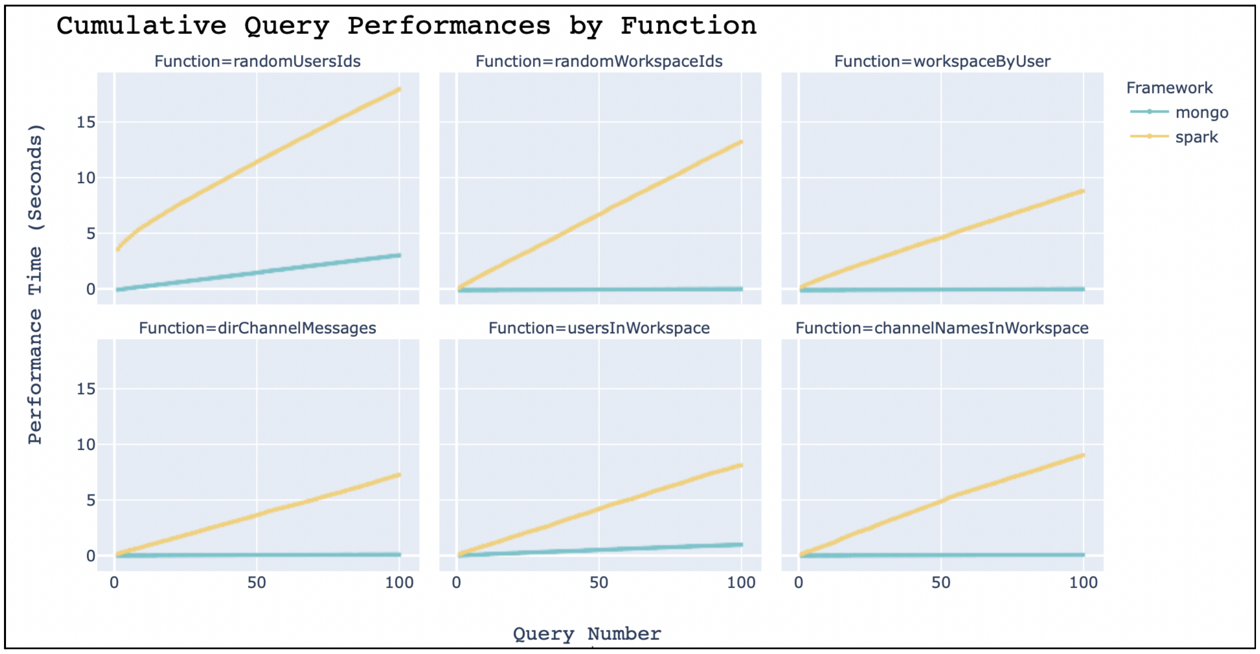
Figure 1 - Performance Times of Iterative Queries by Function



The visualization above demonstrates the performance of our iterative queries for each of the six functions we created. Each subplot/query demonstrates a similar trend with Spark (as depicted by the orange line) consistently exhibiting slower performance times than Mongo (as shown by the green line).

We expected that Spark would be faster in retrieving query results as Spark is known for being a fast, in-memory data processing engine. However, as shown in Figure 1 above, we found that our NoSQL database had faster retrieval times. In the graph, the x-axis indicates each of the one hundred individual query repetitions for each query function and the y-axis indicates the performance time in seconds. The subplots demonstrate that the Mongo queries were consistently faster than the Spark queries, with most Mongo queries falling between 0 and 0.04 seconds, while the majority of Spark queries lasted longer than 0.06 seconds. Furthermore, we can compare the performances of the query repetitions between functions. For both Mongo and Spark queries, the function that retrieved random user IDs took the most amount of time, while the function that retrieved messages in a direct channel lasted for the shortest amount of time.

Figure 2 - Performance Times of Cumulative Queries by Function



The visualization above demonstrates the performance times of cumulative queries for each of the six functions we created. Each subplot/query demonstrates follows the same trend with Spark consistently exhibiting a slower performance time with a sharper increase in time for each additional query than Mongo.

Similarly, the cumulative performance times, in which the durations of each of the query repetitions are added together, were significantly slower for Spark than Mongo. As shown in Figure 2 above, the yellow lines representing Spark's performance time consistently have sharper positive slopes than the blue lines representing Mongo's performance time, meaning that each additional Spark query takes significantly longer than each Mongo query. As a result, by the end of the 100 queries, Spark's performance time ranges between 7 seconds (from retrieving messages in a direct channel in "dirChannelMessages") to 18 seconds (from retrieving random user IDs in "randomUsersIds"). On the contrary, Mongo's performance time ranges between approximately 0.1 to 3 seconds for each of the functions' 100 queries, which is significantly less time than Spark.

Because Spark pulled data from our Mongo database and converted it to an SQL table before processing the queries, Spark ended up overcomplicating the process, and therefore increasing the retrieval time. On the other hand, Mongo as a non-relational, document database had faster query retrievals since it stored data as hierarchical objects and had rich query support for searching through nested fields.

SQL remains an industry standard as a foundation for supporting data integration efforts and relational databases are generally known to have a higher quality of database consistency. Therefore, being able to continue to work on getting faster retrieval times with Spark would be a goal worth working towards as it is better suited to complement SQL databases. Another way we could have approached retrieving our queries could have been to populate a database in MySQL and then connect Spark to MySQL.

Conclusions

As depicted by the visualizations above and discussed in the Analysis portion, the iterative Spark queries were significantly slower (0.1 - 0.2 seconds) than the Mongo queries (0.0 - 0.1 seconds). This was consistent across all queries that were tested. In many of our bar graphs (as shown below in the appendix), it is even difficult to see the difference between the two run times because it is so drastic that the Mongo query runtimes seem negligible compared to the Spark query run times.

Throughout this project, we were able to accomplish a lot in terms of developing a replication of Slack's backend and also performance testing it on two different types of databases. Originally, we had tried other databases such as Redis and different approaches to the Slack object classes, but the structure of Redis did not end up making sense for this use case. In terms of scaling our database for performance testing, we were able to write a total of 6 unique queries and run it on 100 randomly generated workspaces, channels, and users. Overall, we were able to implement everything that we had set out to do in our original proposal and the findings from this project are meaningful.

Although we were able to recreate Slack's backend in terms of data architecture and functionality, we discovered that using Spark to process and retrieve queries was not efficient, as Spark is better suited for significantly larger databases. Thus, one major limitation of our project approach is that our database can only be queried efficiently using Mongo queries. Therefore, next steps for this project would be to test out other types of databases and combine them with Spark queries, such as PostGreSQL. We would expect SQL-based data storage approaches to be much more compatible and efficient. Overall, the main issue we found was connecting Spark to Mongo was inefficient, so we hope to further research other SQL and NoSQL alternatives that would pair better with Spark.

Author Contributions

For this group project, we split up the tasks as fairly as possible. Jason took the lead in creating the Mongo database and populating it with randomly generated channel, message, user, and workspace data from the generator class, while Marco worked on the implementation of connecting Mongo to Spark to process queries and compare retrieval times as well as writing the queries for Spark performance testing. Both Jason and Marco primarily worked on the programming side of the project. Xinyu, Emily, and Brinda created the settings class, as well as the data generator functions specific to the settings class. Additionally, Brinda wrote interesting Mongo queries for performance testing. Emily worked on creating unique, interactive visualizations for ease of analysis and to demonstrate our findings using the Python Plotly library. Xinyu initiated writing the report and presentation, with a focus on describing our methodology and motivation for the project, as well as making the presentation easy to interpret. Emily and Brinda shared responsibility for the analysis and conclusion sections of both the presentation and writeup. Overall, we did our best as a group to communicate as frequently as possible through group text chats and weekly zoom meetings in order to make meaningful contributions to the project.

References

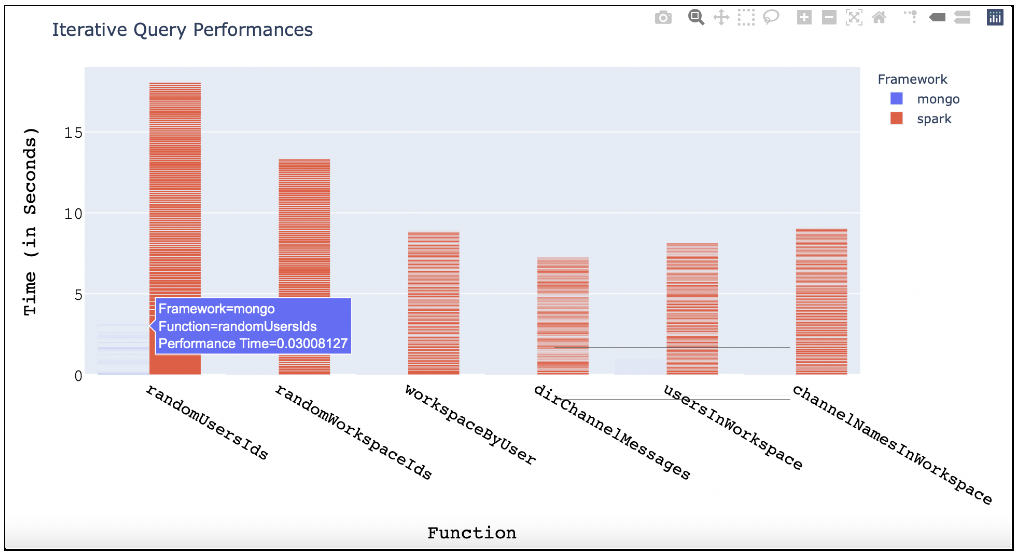
1. *Apache spark™ - what is spark*. Databricks. (2022, March 18). Retrieved April 27, 2022, from https://databricks.com/spark/about

2. *The Application Data Platform*. MongoDB. (n.d.). Retrieved April 27, 2022, from https://www.mongodb.com/

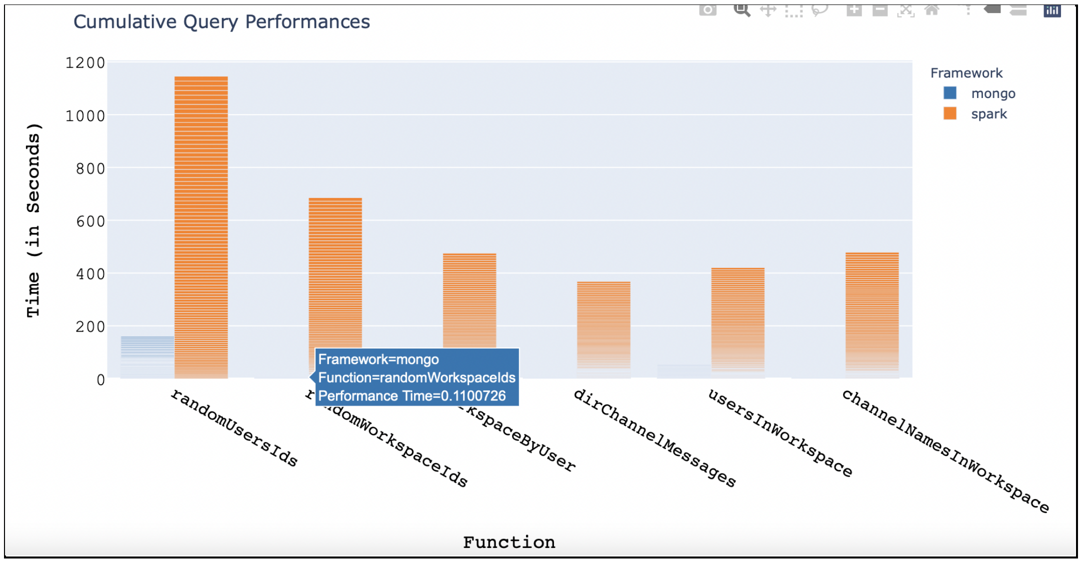
3. Slack. (n.d.). *Slack is your digital HQ*. Slack. Retrieved April 27, 2022, from https://slack.com/

Appendix

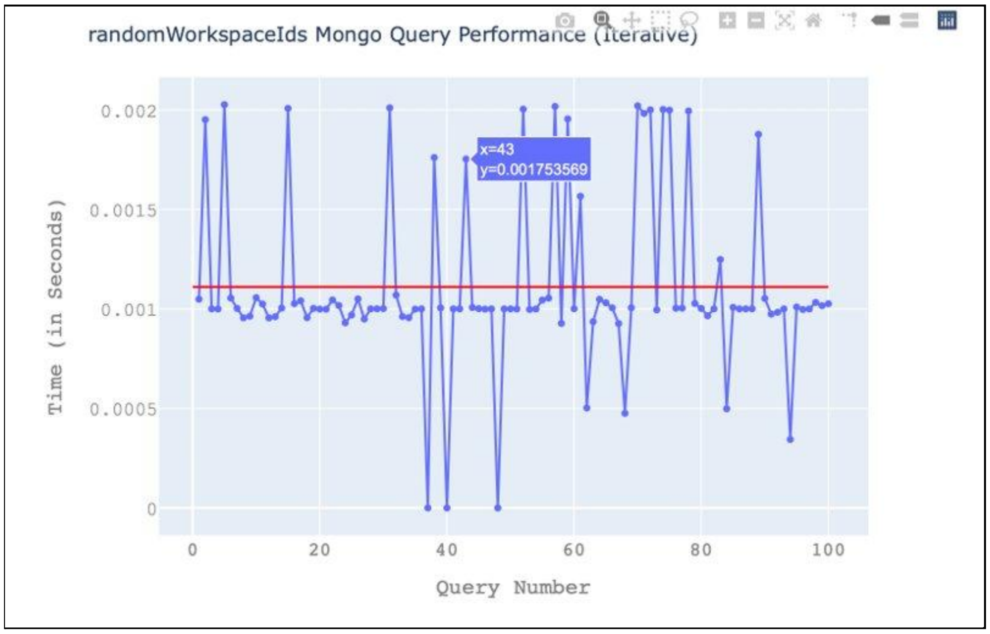
Iterative Query Performances



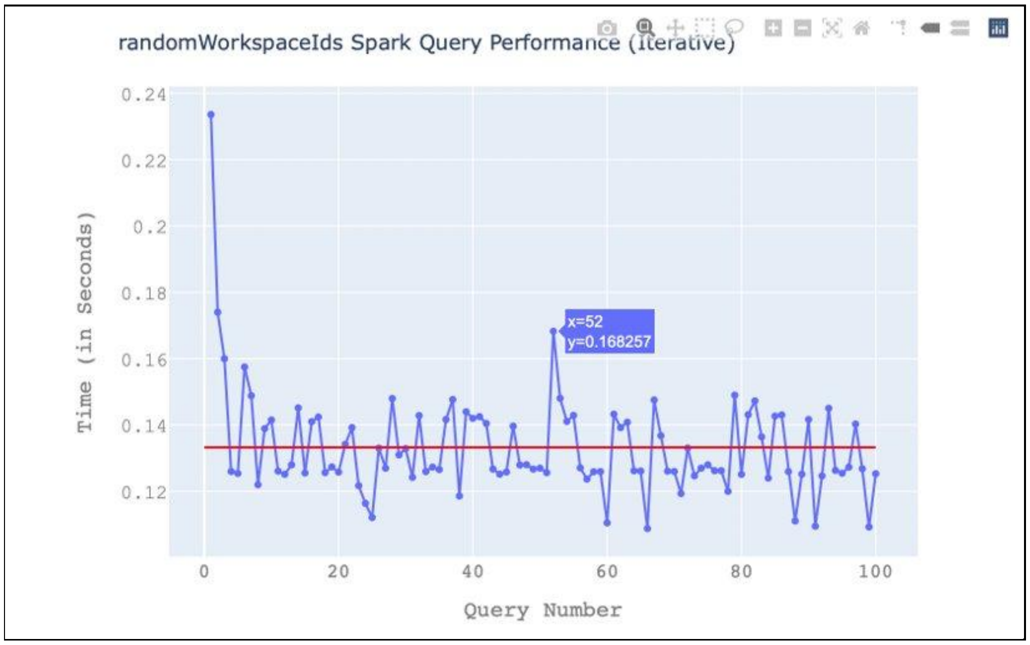
Cumulative Query Performances



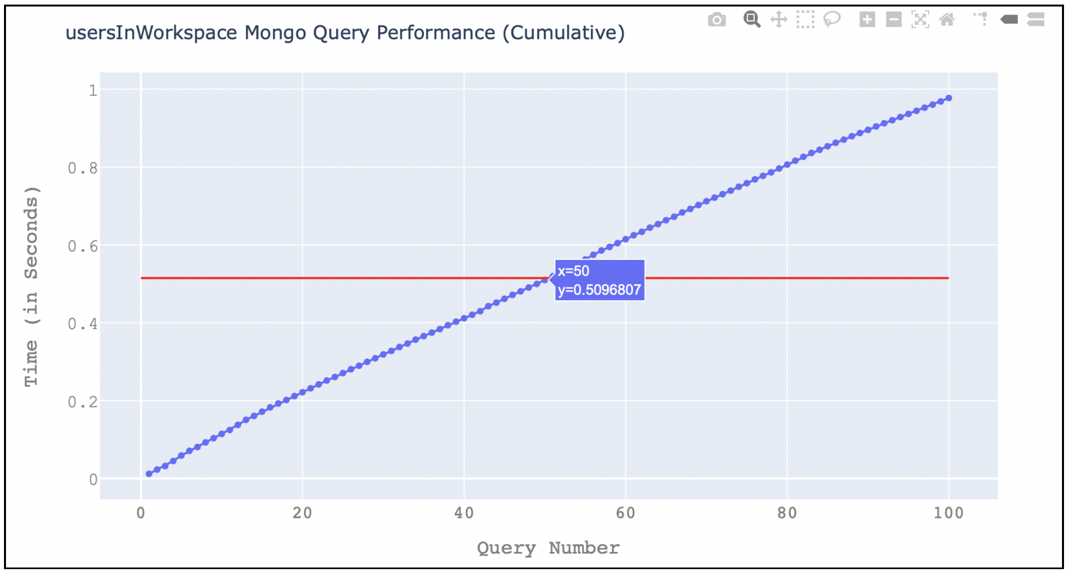
Generating Random Workspace IDs Iterative Mongo Query Performance (Single Function Example)



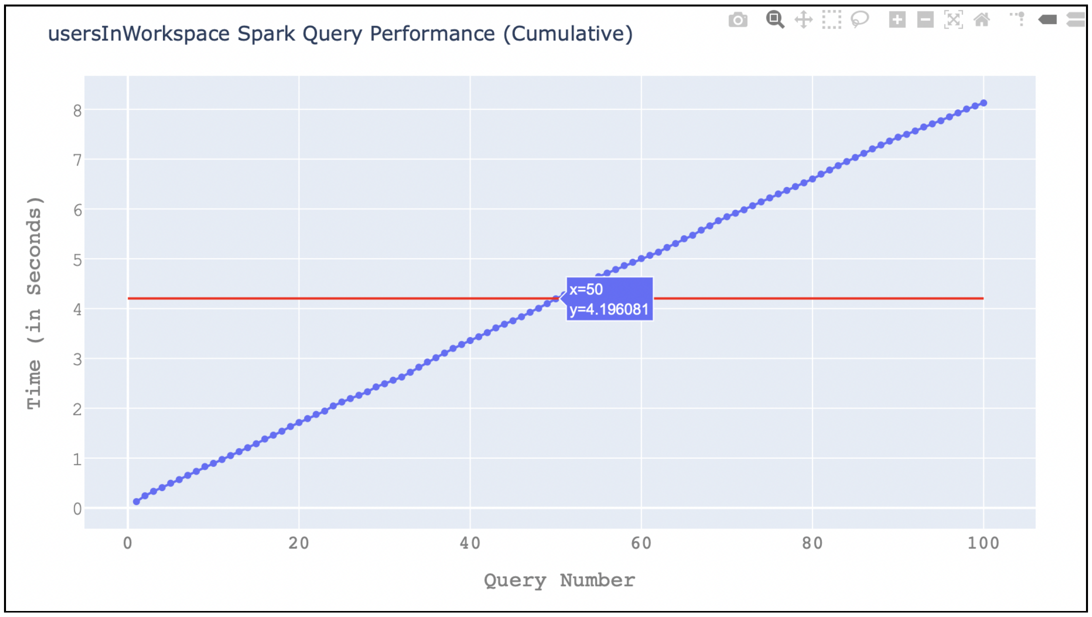
Generating Random Workspace IDs Iterative Spark Query Performance (Single Function Example)



Generating All Users in Workspace Cumulative Mongo Query Performance (Single Function Example)



Generating All Users in Workspace Cumulative Spark Query Performance (Single Function Example)


usersInWorkspace Spark Query Performance (Cumulative)
Time (in Seconds)
Query Number
x=50
y=4.196081