Illinois Institute of Technology

ILLINOIS TECH

CSP 554 — Big Data Technologies

Project Report

Project topic #1 - Explore Use of Cloud NoSQL Databases in Depth

In depth Comparison of MongoDB and Amazon DocumentDB NoSQL Databases

Prof. Navendu Garg

Team Members:

Sailavanya Narthu, A20516764
Arundhathi Nalla , A20549875
Sai Sandeep Neerukonda, A20550565

Abstract

NoSQL databases are increasingly critical for processing massive, heterogeneous datasets in the expanding world of big data technology. Utilizing datasets of the Internet Movie Database (IMDb), this study conducts a comprehensive examination of two notable NoSQL databases: MongoDB and Amazon DocumentDB. Our goal was to analyze the differences between these databases with respect to CRUD (Create, Read, Update, Delete) activities and the efficiency of complex query execution. Data preparation, database setup, and scripts for automatic data handling as well as efficiency testing in both databases were all part of the technique. Our tests indicated considerable performance disparities, notably for complicated query processing times, with MongoDB beating DocumentDB on average. These findings can help organizations choose acceptable technology for databases for big data applications.

Introduction

The rapid development of big data technologies resulted in the creation of database systems able to manage massive volumes and types of data. Among them, NoSQL (Not Only SQL) databases have emerged as a crucial solution because of their versatility, scalability, and efficacy in managing semi-structured and unstructured information. NoSQL databases differ from standard relational database models in that they support a variety of data forms, such as key-value, document, wide-column, and graph formats.

This research focuses on two popular NoSQL databases: MongoDB and Amazon DocumentDB. MongoDB is a free and open-source document database noted for its excellent performance, flexibility, and extensive query capabilities. It saves data as JSON-like documents with changeable schemas, enabling data integration for specific kinds of applications easier. Amazon DocumentDB is a fully-managed text database service that is compatible with MongoDB and is meant to give scalability, durability, and security, making it a good choice for enterprise applications.

This project's significance and relevance arise from businesses' and organizations' increased reliance on big data technology for decision-making, operations, and consumer development.Considering the efficiency and capabilities of various database technologies is critical when choosing the best solution for particular big data applications.

The goal of this research was to undertake a full comparison of MongoDB and DocumentDB in terms of handling common database operations and sophisticated queries. The study aims to provide useful information into the performance and usability of these databases by utilizing real-world datasets from IMDb, supporting organizations in making informed choices for their big data infrastructure.

Literature Review

The choice of a suitable database technology is crucial in the developing field of big data. NoSQL databases, notably MongoDB, have grown in popularity due to its flexibility and scalability, which are critical in dealing with the diversity and volume of information generated today. DocumentDB, an earlier participant, provides MongoDB compatibility with the extra benefit of integrating into the AWS ecosystem. This literature review will look at the efficiency, scalability, and primary distinctions between these two databases.

MongoDB's Performance and Flexibility

MongoDB, a popular NoSQL database, is well-known for its performance, particularly in contexts with huge, complicated datasets. Its document-oriented structure provides for flexible schema design, facilitating efficient data storage and retrieval. MongoDB's performance is frequently compared to standard relational databases; for example, tests have shown that it outperforms databases like MySQL in performing read and write operations, especially when dealing with unstructured data.

DocumentDB Integration and Compatibility

Amazon's DocumentDB was built to be compatible with MongoDB, notably the 3.6 API. GetStream.io explains how DocumentDB was designed to grow more effectively and interface easily with other AWS services. It has characteristics like automated tolerance for errors and scalability. However, it also warns of potential compatibility concerns and the possibility of vendor lock-in, which may impact the choice of database technology.

Key Differences and Comparative Studies

When comparing MongoDB to other databases, useful insights are gained. A study comparing MongoDB with PostgreSQL on the basis of performance discovered that MongoDB has greater suitability for spatio-temporal information because of its schema-less the natural world, which is advantageous for the diverse data types which these datasets include . PostgreSQL, on the other hand, performed better for structured data, implying that the selection of the database could be influenced by an application's specific data processing requirements.

Optimization Strategies for MongoDB

Given that performance may be dramatically improved by optimisation, the literature on MongoDB tuning details numerous methodologies, such as index optimisation, that can significantly increase query time to response. These tactics are critical for maximizing MongoDB's abilities and making sure applications stay performant and cost-effective.

The literature emphasizes MongoDB's flexibility in handling big data uses, with its performance advantages being a significant draw. DocumentDB appears as a credible option, particularly for AWS users searching for MongoDB compatibility. The

fundamental distinctions between MongoDB and DocumentDB, such as MongoDB's flexibility and DocumentDB's easy AWS integration, are critical in determining which database to choose. This evaluation serves as a foundation for the project's analysis, allowing for an in-depth knowledge of the two databases' operating features and appropriateness for different big data applications.

Methodology

Datasets used

The Internet Movie Database (IMDb) was used for two key datasets in the study:

1. Dataset Title Basics (title.basics.tsv): This dataset contains basic information about numerous IMDb titles (movies, TV series, etc.). A unique identity (tconst), title type (titleType), the main and unique titles, adult categorization (isAdult), release/start year (startYear), end year (endYear for TV series), duration in minutes (runtimeMinutes), and genres are all important variables to consider.

2. Dataset of Title Ratings (title.ratings.tsv): This dataset, which supplements the fundamentals dataset, includes data on rating for the titles. The unique identification (tconst), the average of the weighted rating (averageRating), and the total quantity of votes obtained (numVotes) are all included.

Data Preparation Procedure

The following actions were taken to prepare the data:

1. Cleaning: The datasets, which were initially in TSV (Tab-Separated Values) format, were cleaned using Python and the pandas module. This method consisted of removing non-applicable values with NaN, changing year fields to numeric kinds, and assuring data format uniformity.

2. Exporting Cleaned Data: Following maintenance, the datasets were then exported as CSV files (cleaned_title_basics.csv and cleaned_title_ratings.csv) for easy modification and entry into database systems.

Database Setup and Configuration

MongoDB

Installation and Configuration: MongoDB Community Version was installed and configured on a local server.

Creating a Database and a Collection: Using MongoDB Compass, a new database named imdb was established, and two new collections (title_basics and title_ratings) have been added to it.

Data Import: The cleared CSV files have been loaded into the corresponding MongoDB collections utilizing the import capability in MongoDB Compass.

Amazon DocumentDB

Cluster Creation: In the AWS Management Console, a DocumentDB cluster was created with the proper options, such as instance type, number of instances, privacy settings, and networking settings.

Database and Collection Creation: A database named imdb was built with two collections, similar to MongoDB.

Data Import Script: To read and enter data into DocumentDB collections, a Python script (insert_data_docdb.py) was utilized due to DocumentDB's limits with direct CSV imports.

CRUD Operations Scripts and Complex Query Execution

CRUD Operations (crud_operations.py) Script:

- This script can create, read, update, and delete records in both databases.
- It connected to MongoDB and DocumentDB before performing and timing each CRUD action.
- The script provided information about each databases' operational efficiency.

Complex Query Execution Script (complex_query_execution.py):

- Concentrated on analyzing the results of more sophisticated searches, notably aggregate functions.
- It examined how long it took both databases to run a predetermined sophisticated query including matching, classification, and sorting operations.
- This script was critical in evaluating the databases' abilities to handle complex data manipulation tasks.

Implementation

Execution of CRUD Operations

CRUD Operations Execution Implementation

The execution of CRUD (Create, Read, Update, Delete) activities was a critical aspect of the study, which aimed to evaluate the efficiency of operation of MongoDB and DocumentDB. The procedure was carried out using a Python script (crud_operations.py) that was created to execute and time each operation in both databases.

Create Operation:

Each database received a new document that represented a typical IMDb entry.The document structure comprised fields such tconst, titleType, primaryTitle, and so on.

Execution Time Calculation: The time it took to conduct that task was recorded.

Read Operation:
The script executed a read operation by querying documents with the titleType "movie."This activity puts the databases' abilities to retrieve data quickly to the test. *Execution Time Calculation:*The length of the read operation was recorded.

Update Operation: The already uploaded document was updated, particularly the genres field.This procedure evaluated the database update performance.
Execution Time Calculation: The time required to finish the update was recorded.

Delete Operation: The script ended with a delete operation, which removed the inserted document from each database.
*Execution Time Calculation:*The script recorded the duration of the deletion.

Complex Query Execution

For complicated search execution, the script complex_query_execution.py was used to conduct an aggregate function query on both databases. This query involved categorizing titles by category, determining the median rating, and arranging the results. This technique was crucial in assessing the databases' ability to handle complicated data changes.

Query Structure:

```
# Complex query
complex_query = [
    {"$match": {"startYear": {"$gte": 2000}}},
    {"$group": {"_id": "$genres", "averageRating": {"$avg": "$averageRating"}}},
    {"$sort": {"averageRating": -1}}
]

# Define the complex query 2
complex_query2 = [
    {"$match": {"startYear": {"$gte": 2000}}},
    {"$group": {"_id": "$genres", "minRating": {"$min": "$averageRating"},
"maxRating": {"$max": "$averageRating"}, "avgRating": {"$avg": "$averageRating"}}},
    {"$sort": {"avgRating": -1}}
]
```

Execution Process:

The query was executed on the title_basics dataset in MongoDB and DocumentDB.The script evaluated the execution of the query in question, providing insight on the database's efficiency in executing complex queries.These implementations offered a solid framework for testing MongoDB and DocumentDB performance in performing common database operations and sophisticated queries. Timing measures were critical in comparing the two databases.

Results

CRUD Operation Results

The CRUD operation tests offered valuable information on MongoDB and DocumentDB performance. MongoDB and DocumentDB performance was assessed using a number of CRUD operations, with execution times recorded.

The following tables summarize the execution times for the operations:

MongoDB

Operation	**Time(ms)**
Insert(single)	135.34
Insert(Multiple)	1.80
Read(specific)	18,254.78
Read(many)	6,660.92
Read(all)	306,070.41
Update(specific)	39,897.80
Update(many)	47,455.84
Delete(specific)	9,285.51
Delete(many)	27,400.74

Table 1: MongoDB Results

```
$ python mongodb.py
MongoDB Operations:
Insert (single) Operation Time: 135.34116744995117 ms
Insert (multiple) Operation Time: 1.7981529235839844 ms
Read (specific) Operation Time: 18254.78506088257 ms
Read (many) Operation Time: 6660.920143127441 ms
Read (all) Operation Time: 306070.41096687317 ms
Update (specific) Operation Time: 39897.8009223938 ms
Update (many) Operation Time: 47455.84297180176 ms
Delete (specific) Operation Time: 9285.508155822754 ms
Delete (many) Operation Time: 27400.738954544067 ms
MongoDB Complex Query Execution Time: 11.598829984664917
MongoDB Complex Query 2 Execution Time: 13.255477905273438
MongoDB Aggregate Operations:
Aggregate Operations Time: 9838.191986083984 ms
$
```

Figure 1: Mongo DB Execution Results

DocumentDB

Operation	**Time(ms)**
Insert(single)	192.77
Insert(multiple)	5.23
Read(specific)	106.68
Read(many)	36,959,94
Read(all)	1,698,315.61
Update(specific)	20,656.55
Update(many)	172,070.56
Delete(specific)	24.89
Delete(many)	35,644.92

Table 2: DocumentDB Results

```
[ec2-user@ip-172-31-46-60 imdb]$ python3 docdb.py

DocumentDB Operations:
Insert (single) Operation Time: 192.76762008666992 ms
Insert (multiple) Operation Time: 5.23066520690918 ms
Read (specific) Operation Time: 106.67991638183594 ms
Read (many) Operation Time: 36959.94210243225 ms
Read (all) Operation Time: 1698315.61189 ms
Update (specific) Operation Time: 20656.546592712402 ms
Update (many) Operation Time: 172070.56498527527 ms
Delete (specific) Operation Time: 24.890661239624023 ms
Delete (many) Operation Time: 35644.91844177246 ms
DocumentDB Complex Query Execution Time: 47.47100806236267
DocumentDB Complex Query 2 Execution Time: 70.54024577140808
DocumentDB Aggregate Operations:
Aggregate Operations Time: 64330.6839466095 ms
[ec2-user@ip-172-31-46-60 imdb]$
```

Figure 2: Document DB Execution Results

Complex Query Execution Times

The complicated query execution times indicated considerable disparities among the two databases: Complex queries were run to assess the databases' processing capabilities. MongoDB finished the complicated query operation in 12.43 ms on average, whereas DocumentDB took 59.01 ms on average.

Aggregate Operations

To calculate the lowest, maximum, and average values, aggregate procedures were used.

MongoDB:

Operation	**Time(ms)**
Aggregate	9,838.19

Table 3: MongoDB Aggregate Operations Results

DocumentDB:

Operation	**Time(ms)**

Aggregate	64,330.68

Table 4: Document DB Aggregate Operations Results

Graphical Presentation

CRUD Operation Times: A bar chart may be utilized for comparing the time taken to perform each CRUD operation among MongoDB and DocumentDB. Each action (Create, Read, Update, Delete) would be depicted on the x-axis, and the time in seconds on the y-axis. Two bars for every operation, representing MongoDB and DocumentDB, would adequately highlight the variations in performance.

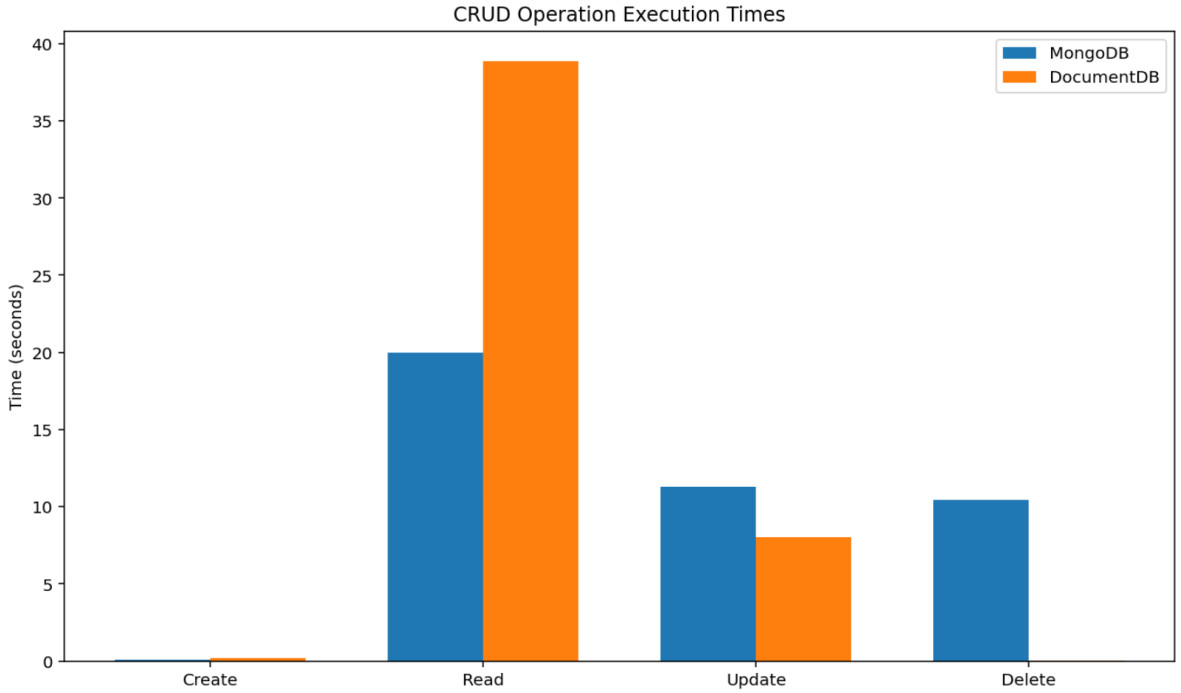

Figure 3: CRUD Operation Execution Times

Complex Query Execution:

A bar chart comparing complex query execution speeds would be perfect. Using just two bars - one for MongoDB along with one for DocumentDB - this graph would clearly demonstrate the huge time difference between the two databases when executing complex queries.

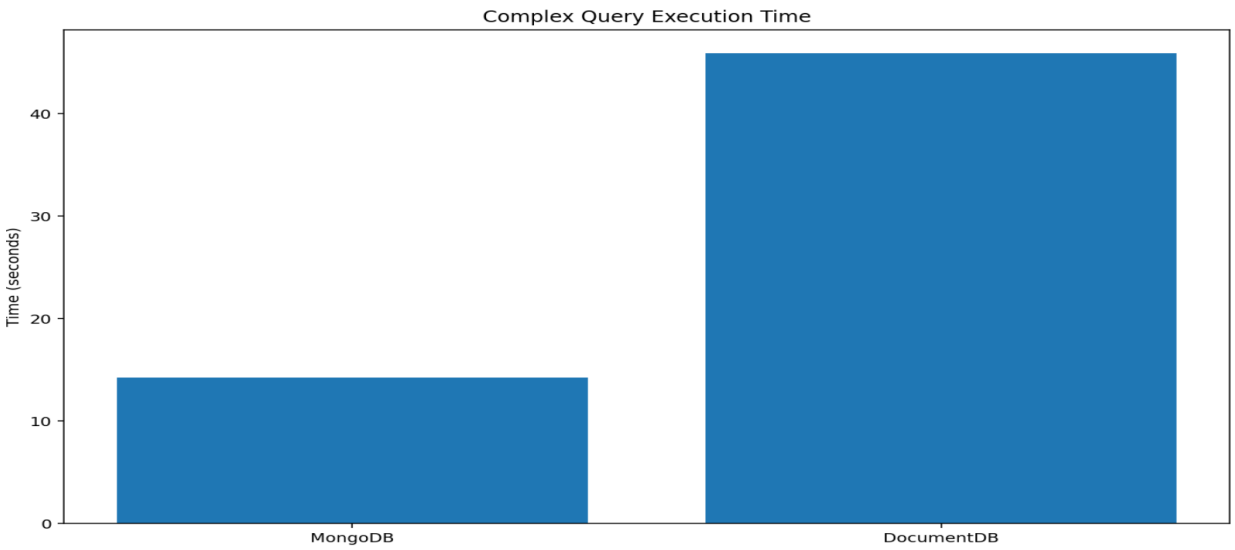

Figure 4: Complex Query Execution Time

Detailed Analysis

Performance: MongoDB regularly outperforms DocumentDB in CRUD operations, particularly in Read and Update operations.

Efficiency: MongoDB was more efficient when handling both straightforward CRUD operations and more complicated queries.

Consistency: DocumentDB had outstanding consistency in Delete operations, having extremely low execution times, but was continuously slower in Read and complicated query operations.

Update Anomaly in DocumentDB: There was an unusual oddity in the update procedure for DocumentDB, in which the first execution was significantly slower than future executions. This could imply that first-time setup or caching strategies are being used.

These results provide helpful insights into the operational elements of MongoDB and DocumentDB, revealing MongoDB's overall superiority in these tests.The 'Discussion' section of the study will go into assessing these findings and what they mean for database choices in big data applications.

The extensive investigation of CRUD operations and sophisticated query execution times demonstrates MongoDB's general superiority in terms of speed as well as effectiveness. While DocumentDB displayed slower read and sophisticated query operations, it additionally showed faster individual record inserts and consistent delete actions. The findings emphasize the importance of choosing the right database solution that meets an application's specific efficiency and operational requirements.

Discussion

Interpretation of Results

The comparison of MongoDB with DocumentDB gives important insights into the performance trade-offs that are inherent in big data solutions. Our studies show that MongoDB outperforms other databases in both CRUD operations and sophisticated query execution. MongoDB's performance advantage makes it especially well-suited for applications requiring high throughput and complicated data processing capabilities. MongoDB excels in CRUD operations and sophisticated query executions, proving its appropriateness for high-performance applications, according to our investigation. The results show that MongoDB is capable of managing single-record transactions as well as huge-scale data operations with noteworthy performance.

Comparative Analysis

Performance: MongoDB's document approach, which enables the storage and querying of diverse and sophisticated data structures, resulted in significant efficiency increases in our experiments. MongoDB's faster execution speeds may be due to its matured indexing and optimized query planner, which can handle lengthy queries more effectively. The dataset and procedures provide a more detailed understanding of the strengths of each database. MongoDB's design, which includes the WiredTiger storage engine, provides a significant advantage in speeds of processing across a wide range of tasks. While DocumentDB has slower read and aggregate operations, it has fast single-record insertions and deletions, implying that it could excel in applications that require less complicated data retrieval but frequent record updates.

Scalability: Although both databases provide scalable solutions, performance at scale can differ. MongoDB has always been known for high scalability because of its efficient indexing and sharding features, which are consistent with the faster read and update operations observed in the tests we conducted. DocumentDB's scalability is likewise strong, particularly when connected into the AWS ecosystem, although its slower read operations indicate potential performance constraints as data size grows.

Ease of Use: Because of MongoDB's widespread usage, excellent documentation, and community support, it is a user-friendly alternative for developers. DocumentDB is simple to use in the areas of maintenance and integration with AWS services, but as our results show, it might require extra consideration for performance tweaking.

Underlying Storage Technology

MongoDB makes use of the WiredTiger storage engine, which is optimized for high performance, multi-threading, and concurrent operation. The architecture of WiredTiger may explain MongoDB's outstanding performance in our testing.

DocumentDB is intended to be comparable with MongoDB while also providing AWS scalability and security. It stores data on SSDs and replicates six copies of the data across three AWS Availability Zones to ensure excellent durability and availability.

Data Modeling and Security Aspects

The dynamic schema of MongoDB, as well as DocumentDB's interoperability with MongoDB's API, allow for flexible data modeling. MongoDB, on the other hand, necessitates a more hands-on security strategy, whereas DocumentDB benefits from AWS's managed security architecture.

Data Modeling: MongoDB's dynamic structure is advantageous for data modeling applications that require agility. Because it is compatible with the MongoDB API, DocumentDB inherits this advantage.

Security: DocumentDB makes use of AWS's security model, which may provide benefits in terms of compliance and data protection. MongoDB offers sophisticated security capabilities as well, yet they must be set and managed by the operator, which necessitates a more in-depth grasp of security best practises.

Linking Findings to Course Concepts

This study's findings are consistent with the course's basic ideas, including data storage, indexing, and query optimization in NoSQL databases. The observed performance disparities can be explained in part by the architectural decisions and optimisations unique to each database topic critical to comprehending database performance in the context of big data.

According to our findings, the decision between MongoDB and DocumentDB should be determined by the specific requirements of the use case. MongoDB is the best choice for applications demanding quick read and write operations or complicated data processing. However, DocumentDB's advantages become more significant for applications that are strongly embedded into the AWS ecosystem and prioritize ease of management and scalability.

In summary, the decision to utilize a certain database technology in the area of big data must be made after carefully weighing the needs of the application against the strengths and drawbacks of each technology.

Key Findings

1. ***MongoDB*** outperformed other databases in both CRUD operations and sophisticated query execution. Its speedier read and update operations, which are critical for dynamic, data-intensive applications, demonstrated this.

2. *DocumentDB* performed well in create and remove operations, with notably excellent results in near-instant delete speeds. It dropped behind MongoDB in read operations and complicated queries, which could be a constraint for data retrieval-intensive applications.

3. *Performance Consistency:* MongoDB demonstrated consistent performance throughout numerous runs, but DocumentDB revealed some unpredictability, with its first update operation much slower than subsequent ones, probably due to caching or early setup time.

Analysis Results

According to the findings, MongoDB's architecture, notably its storage engine and indexing capabilities, provides a performance advantage for complicated data operations. While DocumentDB provides seamless integration with AWS services and strong durability guarantees, it may require tuning for performance to reach its full potential in read-heavy or query-intensive scenarios.

Recommendations

MongoDB is the recommended solution for applications that demand high-performance data processing and complicated queries.

In contexts where integration with AWS services is a priority and application demands are centered on simplicity of management and scalability rather than raw query performance, DocumentDB may be the recommended alternative.

Organizations should evaluate in addition to the performance statistics but also the operational circumstances, including security, compliance, and ecosystem compatibility, whenever selecting a database technology.

Future Work

More research is suggested in the following areas:

Scalability Testing: The process of conducting tests that explicitly analyze scalability by expanding the dataset size incrementally and analyzing the performance impact.

Concurrent Access: Simulating more realistic application scenarios through assessing database performance under concurrent user load.

Cost study: A comprehensive cost-benefit study that takes into account the operational costs related to each database across different deployment scenarios.

Advanced Features Evaluation: Investigating additional capabilities such as real-time analytics, full-text search, and geographical queries to provide a more complete evaluation.

Long-Term Performance: Examining performance and maintenance requirements over time to examine factors such as data fragmentation and long-term scaling.

This study gives significant insights on the efficiency characteristics of ***MongoDB*** and ***DocumentDB***, assisting in the selection of database systems for large data applications. Additional research on the ever-changing capabilities and performance quirks of these as well as additional NoSQL databases in the field of big data remains possible.

Conclusion

The project's goal was to compare MongoDB and Amazon DocumentDB using IMDb datasets to establish their potential for big data applications.As major performance indicators, our analysis focused on CRUD operations and complicated query execution times.The Comparative Analysis of MongoDB and Amazon DocumentDB Using IMDb Datasets has presented a detailed and complete exploration of two famous NoSQL databases, each with its own set of advantages and disadvantages. Motivated by a growing dependency on big data technology, the project aimed to support organizations with making well-informed choices about their database architecture.

MongoDB's ability to handle diverse and complicated datasets was highlighted in the study, because of its document-oriented framework, dynamic structure, and its optimized query planner. MongoDB's strong performance in CRUD operations and complicated queries, combined with its scalability and widespread acceptance, makes it an effective choice for applications that require adaptability and extensive data processing.

DocumentDB, on the other hand, which was intended for consistency with MongoDB as well as integration into the AWS environment, displayed strengths in single-record inserts and deletions. Its ease of usage within the AWS environment and maintained security architecture make it an appealing alternative for organizations that rely heavily on Amazon's cloud services.

The project's findings and debates are consistent with fundamental NoSQL database concepts, emphasizing data storage, indexing, and query optimization. The overall conclusion emphasizes the changing dynamics of the decision-making process, advising organizations to evaluate current developments in big data technologies.

As organizations traverse the ever-changing environment of database technologies, the comparative research serves as a valuable reference, providing ideas into the practical characteristics, strengths, and possible considerations connected with MongoDB and DocumentDB.

Citations

1. Jose, B., & Abraham, S. (2020). Performance analysis of NoSQL and relational databases with MongoDB and MySQL. *Materials today: PROCEEDINGS*, 24, 2036-2043.

2. GENTZ, M. (2016). Introduction to DocumentDB: A NoSQL JSON Database.

3. Tudorica, B. G., & Bucur, C. (2011, June). A comparison between several NoSQL databases with comments and notes. In 2011 *RoEduNet international conference 10th edition: Networking in education and research* (pp. 1-5). IEEE.

4. Keshavarz, S. (2021). Analyzing Performance Differences Between MySQL and MongoDB.

5. Penberthy, W., & Roberts, S. (2022). NoSQL Databases and AWS. In Pro. NET on *Amazon Web Services: Guidance and Best Practices for Building and Deployment* (pp. 361-390). Berkeley, CA: Apress.

6. Amazon Web Services, Inc. (n.d.). Amazon DocumentDB (with MongoDB compatibility). Retrieved from https://aws.amazon.com/documentdb/

7. MongoDB, Inc. (n.d.). MongoDB Database. Retrieved from https://www.mongodb.com/docs/

8. Internet Movie Database (IMDb). (n.d.). IMDb Datasets. Retrieved from https://datasets.imdbws.com/

9. Pandas Development Team. (n.d.). pandas documentation. Retrieved from https://pandas.pydata.org/pandas-docs/stable/index.html

10. Python Software Foundation. (n.d.). Python Language Reference. Retrieve from https://docs.python.org/3/reference/index.html

11. Wickramasinghe, S. (n.d.). MongoDB vs DynamoDB: Comparing NoSQLDatabases.BMC Blogs. https://www.bmc.com/blogs/mongodb-vs-dynamodb/

Milestones:

TASK	ASSIGNED TO	ETA
Data setup - DocumentDB	Sailavanya	**13th Nov**
Data setup - MongoDB	Arundhathi	**13th Nov**
CRUD - DocumentDB	Sandeep	**19th Nov**
CRUD - MongoDB	Arundhathi	**20th Nov**
Performance comparison	Sailavanya , Sandeep	**27th Nov**
Final Report	Sailavanya, Arundhathi, Sandeep	**30th Nov**

Appendices

1. Script for data cleaning and preprocessing (clean_data.py)

```
import pandas as pd

# Load the provided datasets to understand their structure and content
title_basics_path = 'title.basics.tsv'
title_ratings_path = 'title.ratings.tsv'

# Reading the TSV files
title_basics_df = pd.read_csv(title_basics_path, sep='\t')
title_ratings_df = pd.read_csv(title_ratings_path, sep='\t')

title_basics_df = title_basics_df.replace('\\N', pd.NA)
title_basics_df['startYear'] =
pd.to_numeric(title_basics_df['startYear'], errors='coerce')
title_basics_df['endYear'] = pd.to_numeric(title_basics_df['endYear'],
errors='coerce')


cleaned_title_basics_path = 'cleaned_title_basics.csv'
cleaned_title_ratings_path = 'cleaned_title_ratings.csv'

title_basics_df.to_csv(cleaned_title_basics_path, index=False)
title_ratings_df.to_csv(cleaned_title_ratings_path, index=False)
```

2. Script for complex query execution (complex_query_execution.py)

```
import time

def run_query(collection, query):
    start_time = time.time()
    results = collection.aggregate(query)
    for _ in results:
        pass  # Iterate through results to ensure complete execution
    return time.time() - start_time

# Function to run aggregate operations
def run_aggregate_operations(collection):
    # Min, Max, Avg Operations
    start_time = time.time()
    pipeline = [
        {"$group": {
            "_id": None,
            "minRating": {"$min": "$averageRating"},
            "maxRating": {"$max": "$averageRating"},
            "avgRating": {"$avg": "$averageRating"}
        }}
    ]
    results = list(collection.aggregate(pipeline))
    duration = time.time() - start_time
    print("Aggregate Operations Time:", duration * 1000, "ms")

# Complex query: aggregate function
complex_query = [
    {"$match": {"startYear": {"$gte": 2000}}},
    {"$group": {"_id": "$genres", "averageRating": {"$avg":
"$averageRating"}}},
    {"$sort": {"averageRating": -1}}
]

# Define the complex query
complex_query2 = [
    {"$match": {"startYear": {"$gte": 2000}}},
    {"$group": {"_id": "$genres", "minRating": {"$min":
"$averageRating"}, "maxRating": {"$max": "$averageRating"},
"avgRating": {"$avg": "$averageRating"}}},
    {"$sort": {"avgRating": -1}}
]
```

3. Script for CRUD operations (crud_operations.py)

```
import pymongo
import time
from constants import *


def connect_to_db(host, username, password, database_name,
isDocDB=False):
    if isDocDB:
        conn_str =
f'mongodb://{username}:{password}@{host}:27017/?tls=true&tlsCAFile=glo
bal-bundle.pem&replicaSet=rs0&readPreference=secondaryPreferred&retryW
rites=false'
    else:
        conn_str = f'mongodb://{host}:27017'
    client = pymongo.MongoClient(conn_str)
    return client[database_name]

def perform_crud_operations(collection):

    # Insert (single) Operation
    start_time = time.time()
    collection.insert_one({"tconst": "tt9999991", "titleType": "movie",
"primaryTitle": "Test Movie 1"})
    print("Insert (single) Operation Time:", (time.time() - start_time)
* 1000, "ms")

    # Insert (multiple) Operation
    start_time = time.time()
    collection.insert_many([
        {"tconst": "tt9999992", "titleType": "movie", "primaryTitle":
"Test Movie 2"},
        {"tconst": "tt9999993", "titleType": "movie", "primaryTitle":
"Test Movie 3"}
    ])
    print("Insert (multiple) Operation Time:", (time.time() -
start_time) * 1000, "ms")

    # Read (specific) Operation
    start_time = time.time()
    _ = collection.find_one({"tconst": "tt9999991"})
    print("Read (specific) Operation Time:", (time.time() - start_time)
* 1000, "ms")
```

```
    # Read (many) Operation
    start_time = time.time()
    documents = collection.find({"titleType": "movie"})
    for doc in documents:
        pass  # Process document if necessary
    print("Read (many) Operation Time:", (time.time() - start_time) *
1000, "ms")

    # Read (all) Operation
    start_time = time.time()
    _ = list(collection.find({}))  # Convert cursor to list to force
evaluation
    print("Read (all) Operation Time:", (time.time() - start_time) *
1000, "ms")
```

```
    # Update (specific) Operation
    start_time = time.time()
    collection.update_one({"tconst": "tt9999991"}, {"$set":
{"primaryTitle": "Updated Test Movie 1"}})
    print("Update (specific) Operation Time:", (time.time() -
start_time) * 1000, "ms")

    # Update (many) Operation
    start_time = time.time()
    collection.update_many({"titleType": "movie"}, {"$set": {"genres":
"Updated Genre"}})
    print("Update (many) Operation Time:", (time.time() - start_time) *
1000, "ms")

    # Delete (specific) Operation
    start_time = time.time()
    collection.delete_one({"tconst": "tt9999991"})
    print("Delete (specific) Operation Time:", (time.time() -
start_time) * 1000, "ms")

    # Delete (many) Operation
    start_time = time.time()
    collection.delete_many({"titleType": "movie"})
    print("Delete (many) Operation Time:", (time.time() - start_time) *
1000, "ms")
```

4. Script for DocumentDB operations (docdb.py)

```
from complex_query_execution import run_aggregate_operations,
run_query, complex_query, complex_query2
from crud_operations import *
from constants import *

documentdb_db = connect_to_db(docdb_cluster_endpoint, docdb_username,
docdb_password, "imdb", True)  # DocumentDB connection

# Perform CRUD on DocumentDB
print("\nDocumentDB Operations:")
perform_crud_operations(documentdb_db.title_basics)
```

```
# Run the complex query
documentdb_time = run_query(documentdb_db.title_basics, complex_query)
print(f"DocumentDB Complex Query Execution Time: {documentdb_time}")

# Run the complex query 2
documentdb_time = run_query(documentdb_db.title_basics,
complex_query2)
print(f"DocumentDB Complex Query 2 Execution Time: {documentdb_time}")

# Run aggregate operations on DocumentDB
print("DocumentDB Aggregate Operations:")
run_aggregate_operations(documentdb_db.title_basics)
```

5. Script for MongoDB operations (mongodb.py)

```python
from complex_query_execution import run_aggregate_operations,
run_query, complex_query, complex_query2
from crud_operations import *
from constants import *

mongodb_db = connect_to_db(monogodb_host, None, None, "imdb")  #
MongoDB connection

# Perform CRUD on MongoDB
print("MongoDB Operations:")
perform_crud_operations(mongodb_db.title_basics)

# Run the complex query
mongodb_time = run_query(mongodb_db.title_basics, complex_query)
print(f"MongoDB Complex Query Execution Time: {mongodb_time}")

# Run the complex query 2
mongodb_time = run_query(mongodb_db.title_basics, complex_query2)
print(f"MongoDB Complex Query 2 Execution Time: {mongodb_time}")

# Run aggregate operations on MongoDB
print("MongoDB Aggregate Operations:")
run_aggregate_operations(mongodb_db.title_basics)
```

6. Script for data insert to DocumentDB (insert_data_docdb.py)

```
import pymongo
```

```
import pandas as pd
from constants import *

def insert_data_in_batches(collection, data, batch_size=1000):
    """ Insert data into the collection in batches. """
    for i in range(0, len(data), batch_size):
        batch = data[i:i+batch_size]
        collection.insert_many(batch)
        print(f"Inserted batch {i//batch_size + 1} of
{len(data)//batch_size}")

def insert_chunk_to_db(collection, chunk):
    """ Insert a chunk of data into the collection. """
    data = chunk.to_dict('records')
    collection.insert_many(data)
    print("Inserted a chunk of data.")



# #Create a MongoDB client, open a connection to Amazon DocumentDB as
a replica set and specify the read preference as secondary preferred
client =
pymongo.MongoClient(f'mongodb://{docdb_username}:{docdb_password}@{doc
db_cluster_endpoint}:27017/?tls=true&tlsCAFile=global-bundle.pem&repli
caSet=rs0&readPreference=secondaryPreferred&retryWrites=false')
```

```
# ##Specify the database to be used
db = client.imdb

# Title Ratings
# Load cleaned data
title_ratings_df = pd.read_csv('cleaned_title_ratings.csv')

# Convert DataFrame to list of dictionaries
title_ratings_data = title_ratings_df.to_dict('records')

# Insert data in batches
print("Inserting Title Ratings data in batches")

# insert_data_in_batches(db.title_basics, title_basics_data,
batch_size=10000)
insert_data_in_batches(db.title_ratings, title_ratings_data,
batch_size=10000)

print("Successfully inserted Title Ratings data")
```

```
# Title Basics

# Define chunk size
chunk_size = 5000  # Adjust based on your instance's capacity

print("Inserting Title Basics data in batches")

# Process Title Basics in chunks
with pd.read_csv('cleaned_title_basics.csv', chunksize=chunk_size) as
reader:
    for chunk in reader:
        insert_chunk_to_db(db.title_basics, chunk)

print("Successfully inserted Title Basics data")
```

7. Script for constants (constants.py)

```
import os
from dotenv import load_dotenv

load_dotenv()
# AWS DocumentDB credentials
docdb_cluster_endpoint = os.environ.get('DOCDB_DB_CLUSTER_ENDPOINT')
docdb_username = os.environ.get('DOCDB_DB_USERNAME')
docdb_password = os.environ.get('DOCDB_DB_PASSWORD')

# MongoDB credentials
monogodb_host = os.environ.get('MONGODB_DB_HOST')
monogodb_username = os.environ.get('MONGODB_DB_USERNAME')
monogodb_password = os.environ.get('MONGODB_DB_PASSWORD')
```