

Department of Mathematics and Computer Science

Faculty of Data Analysis

UNIME

Database mod NOSQL

Final Report

Comparative Analysis of Database Performance in a Library Management System across Various NoSQL solutions

Students:

Baisalov Daniiar(522597)  
Aibek Abdez(537907)

Academic Year 2023/2024

**Introduction**

Description:

In today's digital age, efficient data management is imperative for the seamless operation of various systems and applications. With the proliferation of diverse database technologies, selecting the optimal database solution for a specific use case has become a significant challenge. In this project, we embark on a comprehensive exploration of different database management systems (DBMS) to evaluate their performance in a real-world scenario – a game review platform.

The aim of this project is to conduct an in-depth comparative analysis of three prominent databases: MongoDB, Redis, and Neo4j. These databases represent a diverse spectrum of technologies, ranging from traditional relational databases to cutting-edge graph databases. By subjecting each DBMS to rigorous testing and evaluation, we seek to identify their strengths and weaknesses in handling the data-intensive demands of a game review platform.

Problem Addressed

The exponential growth of the gaming industry has led to an unprecedented volume of data generated by gamers, game developers, and gaming platforms. Managing this vast amount of data efficiently is crucial for providing users with seamless gaming experiences and insightful game recommendations. A robust database management system is essential for storing, retrieving, and analyzing this data effectively.

Design

The design phase of our project encompasses the creation of a robust data model that captures the essential entities and relationships within a game review ecosystem. We will design three primary entities: games, players, and reviews.

Games: This entity represents individual games available on the platform. It includes attributes such as game ID, title, genre, and release date.

Players: Players are the users who interact with the platform by submitting reviews and ratings for games. This entity includes attributes such as player ID, their in-game nicknames, and email.

Reviews: Reviews capture the opinions and ratings provided by players for specific games. Each review is associated with a game and a player, along with attributes such as review ID, rating, and review date.

The data model will be designed to ensure data integrity and consistency across all database systems. Additionally, we will establish appropriate relationships between entities to facilitate efficient data retrieval and analysis.

Implementation

In the implementation phase of our project, we encountered the challenge of generating large volumes of realistic data for our experiments. While online resources like Mockaroo provide data generation capabilities, they often have limitations on the amount of data that can be generated for free. Consequently, we opted to leverage the Python Faker library to create custom datasets of varying sizes. Using Python Faker, we scripted the generation of datasets ranging from 250,000 to 1,000,000 records, ensuring consistency in data structure and content across all databases. Moreover, Python's flexibility enabled us to customize the datasets according to the characteristics of our game review platform, ensuring that the generated data closely resembled real-world scenarios. We paid attention to details such as data distribution, relationships between entities, and the distribution of review ratings to create realistic datasets that accurately reflected the dynamics of a game review ecosystem. Once the datasets were generated, we proceeded to insert them into each selected database using appropriate programming interfaces or libraries. I wrote scripts to automate the data insertion process, ensuring efficiency and consistency across experiments. Additionally, I’ve implemented code to execute predefined queries on each database, measuring query execution times and capturing relevant performance metrics. These scripts were designed to be modular and extensible, allowing us to easily adapt them to different database systems and experiment scenarios. Overall, the implementation phase of my project involved a combination of data generation, database interaction, and experiment automation using Python programming. By harnessing the power and flexibility of Python and its libraries, we were able to overcome challenges and conduct rigorous experiments to evaluate the performance of various database management systems for our game review platform.

Here I’ve represented my Python code to generate the data:

import csv

import random

import os

from faker import Faker

fake = Faker()

# Set the path to the desktop directory

desktop\_path = os.path.expanduser("~/Desktop")

print('Desktop path: ', desktop\_path)

# Generate unique primary keys

game\_id\_counter = 1

player\_id\_counter = 1

review\_id\_counter = 1

list\_of\_genres = ['Adventure game', 'Action', 'RPG', 'Simulation video game', 'Sports', 'Fighting',

'Platformer', 'Puzzle', 'Survival horror', 'Real-time strategy', 'Sandbox', 'Shooter',

'First-person shooter', 'Multiplayer online battle arena', 'Strategy',

'Massively Multiplayer Online Game (MMO)', 'Racing', 'Survival games', 'Action RPG',

'Battle royale game', 'Stealth game', 'Tactical role-playing game', 'Casual']

release\_dates=[]

# Generate data and save to CSV files

with open(os.path.join(desktop\_path, 'games.csv'), 'w', newline='') as games\_file, \

open(os.path.join(desktop\_path, 'players.csv'), 'w', newline='') as players\_file, \

open(os.path.join(desktop\_path, 'reviews.csv'), 'w', newline='') as reviews\_file:

games\_stat = csv.writer(games\_file)

players\_stat = csv.writer(players\_file)

reviews\_stat = csv.writer(reviews\_file)

# Adding headers to csv files

games\_stat.writerow(['game\_id', 'title', 'genre', 'release\_date'])

players\_stat.writerow(['player\_id', 'nickname', 'email'])

reviews\_stat.writerow(['review\_id', 'game\_id', 'player\_id', 'mark', 'review\_date'])

print('Headers written to CSV files')

# Creating game's data

for i in range(1000000):

title = fake.catch\_phrase()

genre = random.choice(list\_of\_genres)

release\_date = fake.date\_between(start\_date='-30y', end\_date='today')

games\_stat.writerow([game\_id\_counter, title, genre, release\_date])

release\_dates.append(release\_date)

game\_id\_counter += 1

if (i + 1) % 100000 == 0:

print(f"{i + 1} samples has been created")

print('Game data has been successfully created')

# Creating player's data

for i in range(1000000):

nickname = fake.name()

email = fake.email()

players\_stat.writerow([player\_id\_counter, nickname, email])

player\_id\_counter += 1

if (i + 1) % 100000 == 0:

print(f"{i + 1} samples has been created")

print('Player data has been successfully created')

# Creating review data

for i in range(1000000):

game\_id = random.randint(1, 1000000)

player\_id = random.randint(1, 1000000)

mark = random.randint(1, 5)

release\_date = release\_dates[review\_id\_counter - 1]

review\_date = fake.date\_between(start\_date=release\_date, end\_date='today')

reviews\_stat.writerow([review\_id\_counter, game\_id, player\_id, mark, review\_date])

review\_id\_counter += 1

if (i + 1) % 100000 == 0:

print(f"{i + 1} samples has been created")

print('Review data has been successfully created')

Experiments

The experiments will focus on comparing the performance of the selected databases across four predefined queries with increasing levels of complexity. We will measure query execution times, resource utilization, and scalability across different dataset sizes. The results will be recorded in electronic spreadsheets and visualized using histograms to facilitate easy interpretation.

MongoDB

In this section, we delve into the experiments conducted with MongoDB, one of the leading NoSQL database management systems. MongoDB offers a flexible document-oriented data model, making it well-suited for applications with dynamic schemas like our game review platform. Leveraging the scalability and performance of MongoDB, we aimed to assess its suitability for handling the data-intensive demands of our project. Furthermore, to evaluate MongoDB's performance under different query scenarios, we designed a series of queries with increasing levels of complexity. These queries spanned various aspects of our game review platform, including game retrieval, player analysis, and review aggregation. By automating query execution using Python loops, we conducted each query 30 times to obtain robust performance metrics.

Here follows a Python code snippet demonstrating how to connect to the MongoDB database and execute a query 30 times:

import pymongo

import time

import statistics

import math

# Creating MongoDB connection

client = pymongo.MongoClient("mongodb://localhost:27017/")

db = client["Reviews"]

collection = db["games"]

# List for every runtime

execution\_times = []

# Function for cheking the runtime and query

def execute\_query():

start\_time = time.time()

# The query

result = collection.find({})

# Query checking

for doc in result:

pass

end\_time = time.time()

execution\_time = end\_time - start\_time

return execution\_time

# Function usage

for \_ in range(30):

execution\_time = execute\_query()

execution\_times.append(execution\_time)

# Mean

mean\_execution\_time = statistics.mean(execution\_times)

# Confidence interval

confidence\_interval = 1.96 \* statistics.stdev(execution\_times) / math.sqrt(len(execution\_times))

lower\_bound = mean\_execution\_time - confidence\_interval

upper\_bound = mean\_execution\_time + confidence\_interval

print(f"Average runtime: {mean\_execution\_time:.4f} secs")

print(f"95% confidence interval: [{lower\_bound:.4f}, {upper\_bound:.4f}] secs")

Here I also left a list of my queries:

Complexity level 1:  
 db.games.find({})

Complexity level 2:

pipeline = [

{

"$match": {

"mark": {

"$eq": 1

}

}

},

{

"$project": {

"\_id": 0,

"review\_id": 1,

"review\_date": 1

}

}

]

Complexity level 3:

pipeline = [

{

"$lookup": {

"from": "games",

"localField": "game\_id",

"foreignField": "game\_id",

"as": "game"

}

},

{

"$unwind": "$game"

},

"$match": {

"$expr": {

"$and": [

{ "$gte": [{ "$year": "$review\_date" }, 2021] },

{ "$lt": [{ "$year": "$review\_date" }, 2022] }

]

}

}

},

{

"$lookup": {

"from": "players",

"localField": "player\_id",

"foreignField": "player\_id",

"as": "player"

}

},

{

"$unwind": "$player"

},

{

"$project": {

"\_id": 0,

"game\_title": "$game.title",

"player\_nickname": "$player.nickname",

"mark": 1,

"review\_date": 1

}

}

]

{

"$match": {

"$expr": {

"$and": [

{"$gte": [{"$year": "$review\_date"}, 2021]},

{"$lt": [{"$year": "$review\_date"}, 2022]}

]

}

}

},

{

"$lookup": {

"from": "players",

"localField": "player\_id",

"foreignField": "player\_id",

"as": "player"

}

},

{

"$unwind": "$player"

}, {

"$project": {

"\_id": 0,

"game\_title": "$game.title",

"player\_nickname": "$player.nickname",

"mark": 1,

"review\_date": 1

}

}

]

Complexity level 4:

pipeline = [

{

"$lookup": {

"from": "games",

"localField": "game\_id",

"foreignField": "game\_id",

"as": "game\_info"

}

},

{

"$unwind": "$game\_info"

},

{

"$match": {

"game\_info.genre": "Shooter",

"mark": 5,

"review\_date": {

"$regex": "^2020"

}

}

},

{

"$sort": {

"review\_date": -1

}

}

]

And now here comes the result:

As we can see from histograms, the first query usually takes much more time to do a query, than doing it few times.

Neo4j Database

In this section, we delve into the experiments conducted with Neo4j, a leading graph database management system renowned for its ability to store and query interconnected data efficiently. Neo4j's graph-based data model aligns closely with the relational structure of our game review platform, making it an intriguing candidate for our evaluation. Following a similar methodology as with MongoDB, we successfully imported custom datasets into Neo4j using Python scripts. Through these experiments, we aim to gain insights into Neo4j's capabilities and performance characteristics, particularly in handling the complex relationships inherent in our game review platform. The findings will contribute to our overall assessment of Neo4j's suitability for similar graph-based applications and inform database selection decisions accordingly.

Here is all queries:

Query1:  
MATCH (g:Game)

RETURN g;

Query2:

MATCH (g:Game)

WHERE g.release\_date STARTS WITH '2019'

RETURN g.title, g.genre;  
  
Query3:  
MATCH (r:Reviews),(g:Game)

WHERE r.game\_id=g.game\_id

AND r.mark='5'

AND g.release\_date STARTS WITH '2014'

RETURN g.title, g.genre;

Query4:

MATCH (g:Game)

WHERE substring(g.release\_date, 0, 4) = '2022' AND g.genre = 'Shooter'

MATCH (r:Reviews)-[:REVIEWS]->(g)

MATCH (p:Players)-[:WROTE]->(r)

RETURN g.title AS game\_title,

g.genre AS game\_genre,

p.nickname AS player\_nickname,

r.mark AS review\_mark,

r.review\_date AS review\_date

The results are listed here:

As we can see Neo4j takes a lot of time to run a query, but due to caching data, next attempts work a little bit faster even with increasing amount of data.

Redis

In this segment, we embark on the exploration of Redis, a high-performance in-memory data store renowned for its exceptional speed and versatility. Redis's ability to store and retrieve data rapidly makes it an intriguing candidate for our game review platform, where responsiveness and scalability are paramount.

Similar to our approach with MongoDB and Neo4j, we meticulously imported custom datasets into Redis using Python scripts. Leveraging the capabilities of Python and the Redis-py library, we seamlessly integrated our generated datasets into Redis, ensuring consistency and reliability across experiments. This process allowed us to create datasets of varying sizes, mirroring the growth trajectory of our game review platform.  
  
Here I put the list of queries:

Query1:  
MATCH (g:Game)

RETURN g;

Query2:

game\_keys = r.scan\_iter(match='games:\*')

for key in game\_keys:

game\_data = r.hgetall(key)

if game\_data:

release\_date = game\_data.get(b'release\_date', b'').decode()

if release\_date.startswith('2021'):

game\_id = key.decode().split(":")[-1]

title = game\_data.get(b'title', b'').decode()

genre = game\_data.get(b'genre', b'').decode()

print("Game ID:", game\_id)

print("Title:", title)

print("Genre:", genre)  
  
Query3:

review\_keys = r.keys("reviews:\*")

for key in review\_keys:

review = r.hgetall(key)

review\_date = review[b'review\_date'].decode('utf-8')

if review\_date.startswith('2023'):

review\_id = key.decode('utf-8').split(':')[1]

game\_key = "game:" + game\_id

player\_key = "player:" + player\_id

game = r.hgetall(game\_key)

reviewer = r.hgetall(player\_key)

Query4:

Advanced:

# Step 1: Fetch all book IDs

book\_ids = []

books = r.keys("book:\*")

for book\_key in books:

book = r.hgetall(book\_key)

if b'book\_id' in book:

book\_id = book[b'book\_id'].decode('utf-8')

book\_ids.append(book\_id)

# Step 2: Filter books by author and borrowing year

for book\_id in book\_ids:

book\_key = f"book:{book\_id}"

book = r.hgetall(book\_key)

if book.get(b'author', b'').decode('utf-8') == 'Rebecca Patterson':

borrowing\_key = f"borrowing\_history:{book\_id}"

borrowings = r.smembers(borrowing\_key)

for borrowing in borrowings:

borrow\_data = r.hgetall(borrowing)

borrow\_date = borrow\_data.get(b'borrow\_date', b'').decode('utf-8')

if borrow\_date.startswith('2022'):

result = {

'title': book.get(b'title', b'').decode('utf-8'),

'author': book.get(b'author', b'').decode('utf-8'),

'borrow\_date': borrow\_date

}

print(result)

# Step 3: Compute average borrows per book

borrowing\_key = f"borrowing\_history:{book\_id}"

borrowings = r.smembers(borrowing\_key)

borrow\_count = len(borrowings)

avg\_borrows\_per\_book = sum(len(r.smembers(f"borrowing\_history:{other\_book\_id}")) for other\_book\_id in book\_ids) /

len(book\_ids)

# Step 4: Filter books with borrow count greater than average

if borrow\_count > avg\_borrows\_per\_book:

result = {

'title': book.get(b'title', b'').decode('utf-8'),

'author': book.get(b'author', b'').decode('utf-8'),

'borrow\_date': borrow\_date

}

print(result)

In summary, while Redis offers impressive speed and versatility, its performance characteristics under specific conditions warrant careful consideration. By understanding its strengths and limitations, developers can make informed decisions about incorporating Redis into their projects, ensuring optimal performance and scalability for their applications.

MariaDB

In this section, we delve into the experiments conducted with MariaDB, a popular open-source relational database management system known for its robustness, reliability, and compatibility with MySQL. MariaDB offers advanced features, high performance, and scalability, making it a compelling choice for data-intensive applications like our game review platform.

Here is a Python code snippet demonstrating how to connect to the MariaDB database and execute a query 30 times:

Query1:

SELECT \* FROM games;

Query2:

SELECT g.title, r.mark FROM reviews r

JOIN games g ON r.game\_id = g.game\_id

WHERE g.genre = 'Shooter' AND mark = 4;

Query3:  
  
SELECT g.title, r.mark

FROM reviews r

JOIN games g ON r.game\_id = g.game\_id

WHERE g.genre = 'Shooter'

AND r.mark = 4

AND r.mark > (SELECT AVG(r2.mark)

FROM reviews r2

JOIN games g2 ON r2.game\_id = g2.game\_id

WHERE g2.genre = 'Shooter');

Query4:

SELECT g.title, g.genre, r.review\_date

FROM games g

JOIN reviews r ON g.game\_id = r.game\_id

JOIN players p ON r.player\_id = p.player\_id

WHERE YEAR(r.review\_date) = 2022

AND p.nickname = 'April Moore'

GROUP BY g.title, g.genre, r.review\_date

HAVING COUNT(\*) > (

SELECT AVG(reviews\_per\_game)

FROM (

SELECT COUNT(\*) AS reviews\_per\_game

FROM reviews

WHERE YEAR(review\_date) = 2022

GROUP BY game\_id

) AS subquery

)

ORDER BY r.review\_date DESC;

AND p.nickname = 'April Moore'

GROUP BY g.title, g.genre, r.review\_date

HAVING COUNT(\*) > (

SELECT AVG(reviews\_per\_game)

FROM (

SELECT COUNT(\*) AS reviews\_per\_game

FROM reviews

WHERE YEAR(review\_date) = 2022

GROUP BY game\_id

) AS subquery

)

ORDER BY r.review\_date DESC;

Based on the performance analysis of MariaDB queries, it is evident that query execution times are influenced less by the volume of data and more by the complexity of the queries themselves. Unlike some systems where query times linearly scale with data size, MariaDB's performance is more intricately tied to how efficiently queries are structured and optimized.

Conclusions

The findings of this project will provide valuable insights into the performance and suitability of MongoDB, Redis, MariaDB, and Neo4j for a game review platform. By systematically evaluating each database across various parameters, we aim to empower developers, system architects, and database administrators to make informed decisions when selecting a DBMS for similar applications.

In conclusion, this project represents a comprehensive effort to address the challenges of database selection and performance evaluation in the context of a game review platform. Through meticulous experimentation and analysis, we endeavor to contribute valuable knowledge to the field of database management and inform best practices for future projects in the gaming industry.