Course Name: Software and Data Engineering Design Documentation

Topic: A performance comparison of SQL and NoSQL databases

Submitted by:

Akaash Chatterjee
Roll No: M24CSE002

Aman Saini
Roll No: M24CSE003

Bera Swaminath Ansuman Sabita
Roll No: M24CSE007

Department of Computer Science and Engineering
Indian Institute of Technology Jodhpur

2024

Contents

Chapter 1

1.1 Introduction

In recent years, there has been a notable shift in the landscape of data storage solutions. Traditionally, data storage has been dominated by relational database systems, which structure data according to the relational model and are queried using SQL, the Structured Query Language. However, as data needs have evolved, non-relational databases—commonly known as NoSQL databases—have gained popularity. NoSQL databases are built to store simple key-value pairs and leverage flexibility and speed, making them ideal for applications where the rigidity of traditional relational databases may be a limitation.

The rise of Big Data has also driven the adoption of NoSQL databases. With the growth of the Internet, IoT, and the increasing affordability of storage, vast amounts of structured, semi-structured, and unstructured data are now being captured and stored. This "Big Data" requires databases that support high-speed processing, flexible schemas, and scalability across distributed systems. NoSQL databases, which often meet these requirements, have therefore become widely adopted for Big Data applications, with implementations like Google's Bigtable and Apache HBase gaining attention.

In our project, we aim to provide a focused comparison of NoSQL and SQL database implementations specifically for key-value store operations. Although NoSQL databases are typically optimized for such operations, SQL databases are generally not.

Our project is based on the research paper by Y. Li and S. Manoharan, titled "A performance comparison of SQL and NoSQL databases," presented at the 2013 IEEE Pacific Rim Conference on Communications, Computers, and Signal Processing (PACRIM), Victoria, BC, Canada. This study investigates the performance differences between SQL and NoSQL databases, particularly focusing on key-value stores and evaluating read, write, delete, instantiate, and iteration operations. The results indicate that while NoSQL databases are often touted for superior performance, SQL databases sometimes outperform them in specific operations, with performance varying widely across operations and databases.

Chapter 2

2.1 Objectives

The primary objective of this study is to benchmark the performance of various local databases on common key-value operations, including read, write, delete, instantiate, and retrieve-all-keys. By comparing the performance of databases, we aim to provide insights into their suitability for various applications. The key objectives of this study are outlined as follows:

- **Benchmark Local Databases:** We evaluate and compare the speed of key-value operations on a selection of local, lightweight databases. For this study, we chose CouchDB, DuckDB, MongoDB,TinyDB,MySQL,MS SQL Server,SQLite due to their accessibility and their suitability for local use cases.
- **Simplicity:** We use databases that required minimal setup to enable easy testing and quick deployment. Some of the databases originally considered in our study were changed due to factors such as licensing restrictions, deprecation, or placement behind a paywall. The chosen databases in this study allow for a straightforward comparison and setup, enabling us to evaluate a variety of database types.
- **Analyze Performance:** We then generate detailed performance metrics for each database and visualize the results to facilitate comparison. By analyzing these metrics, we aim to identify strengths and limitations of each database in performing key-value store operations, providing insights into their optimal use cases.

Table 2.1 provides a summary of the databases used in this study, along with their respective versions and types (SQL or NoSQL).

Database	Version	Type
CouchDB	3.4.2	NoSQL
DuckDB	1.1.3	SQL
MongoDB	8.0.3	NoSQL
TinyDB	4.8.0	NoSQL
MySQL	8.0.40	SQL
MS SQL Server	2022	SQL
SQLite	3.47.0	SQL

Table 2.1: Databases Used in the Study

2.2 Setup Requirements for Databases

This section details the installation and setup requirements for each database used in the experiment. Each database has been configured to support basic CRUD operations as a key-value store.

2.2.1 CouchDB

Installation:

- **macOS**: Install with Homebrew using `brew install couchdb`.
- **Ubuntu**: Update the package list and install CouchDB with

  ```
  sudo apt update
  sudo apt install couchdb
  ```

- **Windows**: Download and install CouchDB from the official site at `https://couchdb.apache.org/`.

Configuration:

- Access CouchDB's admin console at `http://127.0.0.1:5984/_utils/` and create a database named `KeyValueDB`.

2.2.2 DuckDB

Installation:

- DuckDB is an in-process SQL OLAP database, which can be installed directly via `pip`:

  ```
  pip install duckdb
  ```

Configuration:

- DuckDB requires no separate configuration or server setup. The database operates on local files created at runtime.

2.2.3 MongoDB

Installation:

- **macOS**: Install via Homebrew using `brew install mongodb-community`.
- **Ubuntu**: Install with

  ```
  sudo apt update
  sudo apt install -y mongodb
  ```

- **Windows**: Download MongoDB from `https://www.mongodb.com/`.

Configuration:

- Start MongoDB with `mongod --dbpath <your-db-path>`.
- MongoDB should be running on `localhost` at port 27017.

2.2.4 Microsoft SQL Server Express

Installation:

- Download and install SQL Server Express from the Microsoft SQL Server website at `https://www.microsoft.com/sql-server/sql-server-downloads`.

Configuration:

- Use SQL Server Management Studio (SSMS) to create a new database named `KeyValueDB`.
- Ensure SQL Server is configured to accept local connections.

2.2.5 MySQL

Installation:

- **macOS**: Install via Homebrew with `brew install mysql`.
- **Ubuntu**: Install using

```
sudo apt update
sudo apt install mysql-server
```

- **Windows**: Download from the MySQL website at `https://dev.mysql.com/downloads/installer/`.

Configuration:

- Start MySQL server:

```
mysqld;
net start MySQL[80];
```

- MySQL should be running on `localhost` at port `3306`.

2.2.6 SQLite

Installation:

- SQLite is included with most Python installations and does not require additional setup.

Configuration:

- SQLite databases operate directly on local `.db` files created during runtime, so no separate configuration is needed.

2.2.7 TinyDB

Installation:

- TinyDB can be installed via `pip`:

```
pip install tinydb
```

Configuration:

- TinyDB operates directly on JSON files and requires no server configuration.

—

This setup prepares each database for the experiment, ensuring a consistent configuration for accurate benchmarking of key-value store operations.

2.3 System Architecture

The system is a Python application designed to test and analyze various local databases in an isolated environment. Each database's operations are encapsulated in a class, which provides a unified interface for testing and benchmarking.

2.4 Modules and Components

- Each database test has the following parts:

2.4.1 Random data generator

We generate random set of 1000 key-value pairs data, Each database uses a simple key-value structure:

- **Key**: The unique identifier for each entry. Example format: `k0`, `k1`, . . .
- **Value**: An alphanumeric string representing the stored data. This string is randomly generated with a length of 20 characters for testing purposes.

2.4.2 Database Adapters

Each database's operations are encapsulated in a Python class that exposes a consistent API for each database. Each adapter class implements the following methods:

- **instantiate**: Clears the dataset for a fresh start.
- **read**: Retrieves a value by key.
- **write**: Inserts or updates a key-value pair.
- **delete**: Deletes a key-value pair by key.
- **get_all_keys**: Retrieves all keys in the dataset.

2.4.3 Benchmarking Module

A central module that:

- Repeats each operation multiple times (5 times in this design) to obtain an average execution time.
- Logs the time taken for each operation.
- Calculates average times across multiple operations for comparison.

2.4.4 Visualization Module

A matplotlib based program that plots the metrices observed. We pass on the metrices obtained from benchmarking operation to this module for visualization.

2.5 Testing Strategy

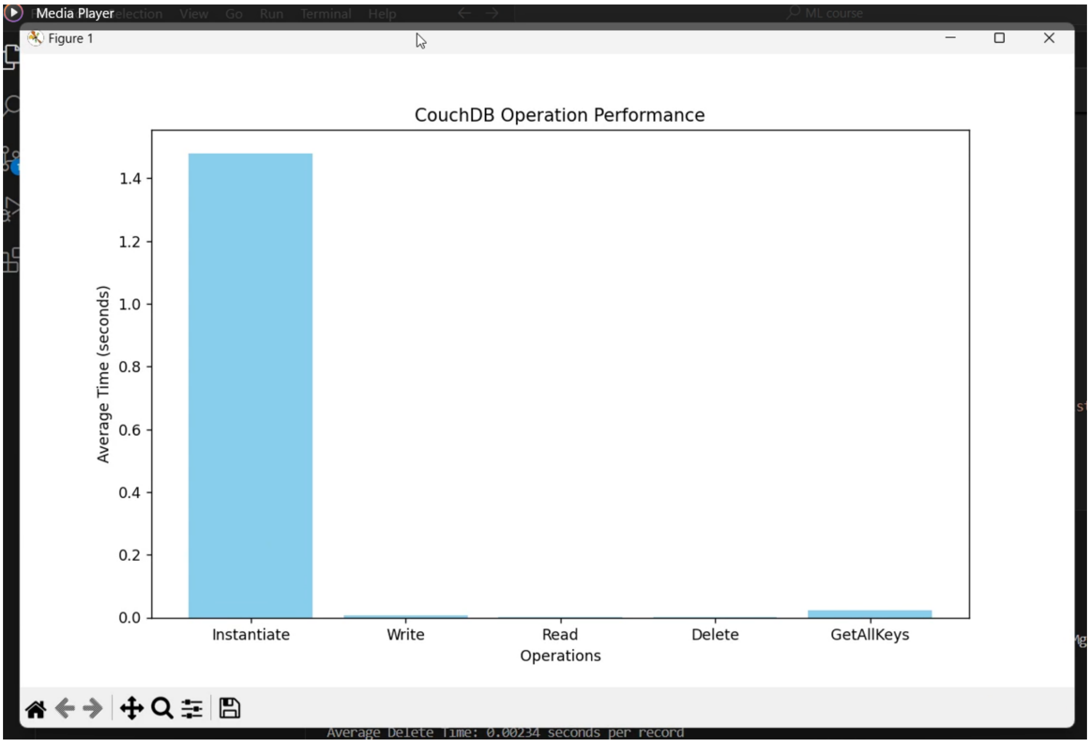
Finally, adapter program initializes each database adapter, performs the benchmark tests on all operations, logs the performance metrics, and generates a summary graph of results.
Input: A dataset of 1000 key-value pairs.
Operations: For each database adapter, perform the following:

- Initialize / Clear all records (*instantiate*).
- Insert key-value pairs (*write*).
- Retrieve values by key (*read*).
- Delete key-value pairs (*delete*).
- Fetch all keys (*get_all_keys*).

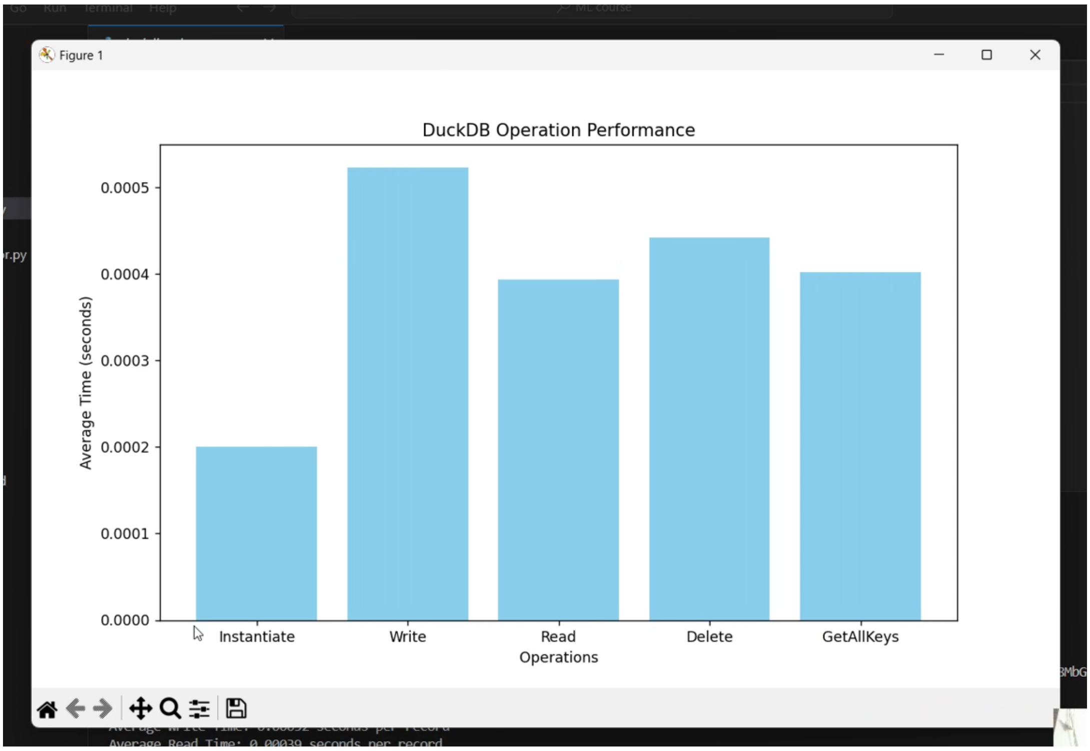
Output: Log the average time for each operation per database adapter.

Chapter 3

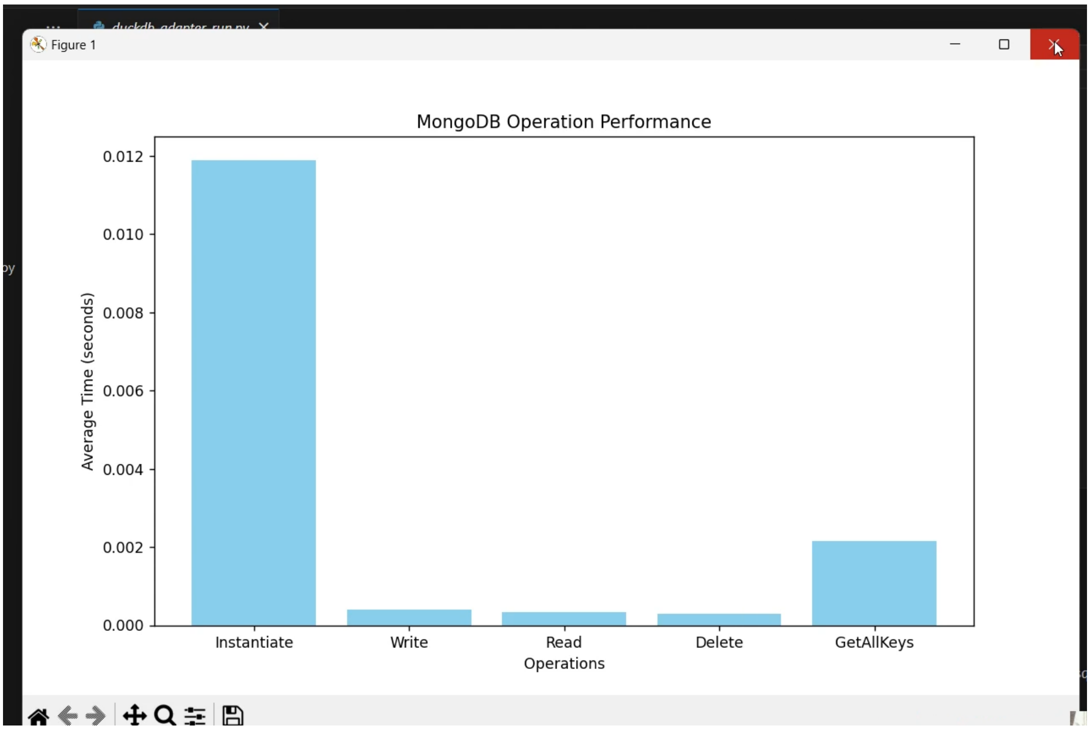
3.1 Results



```
Average Delete Time: 0.00234 seconds per record
```



```
Average Write Time: 0.00052 seconds per record
Average Read Time: 0.00039 seconds per record
```



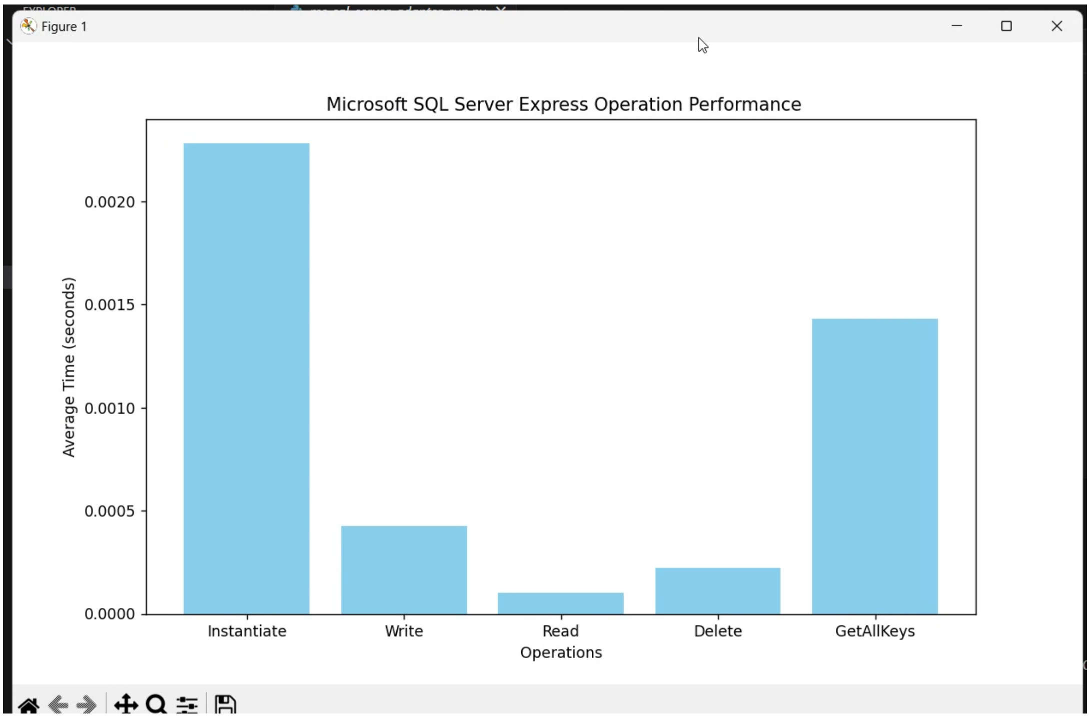
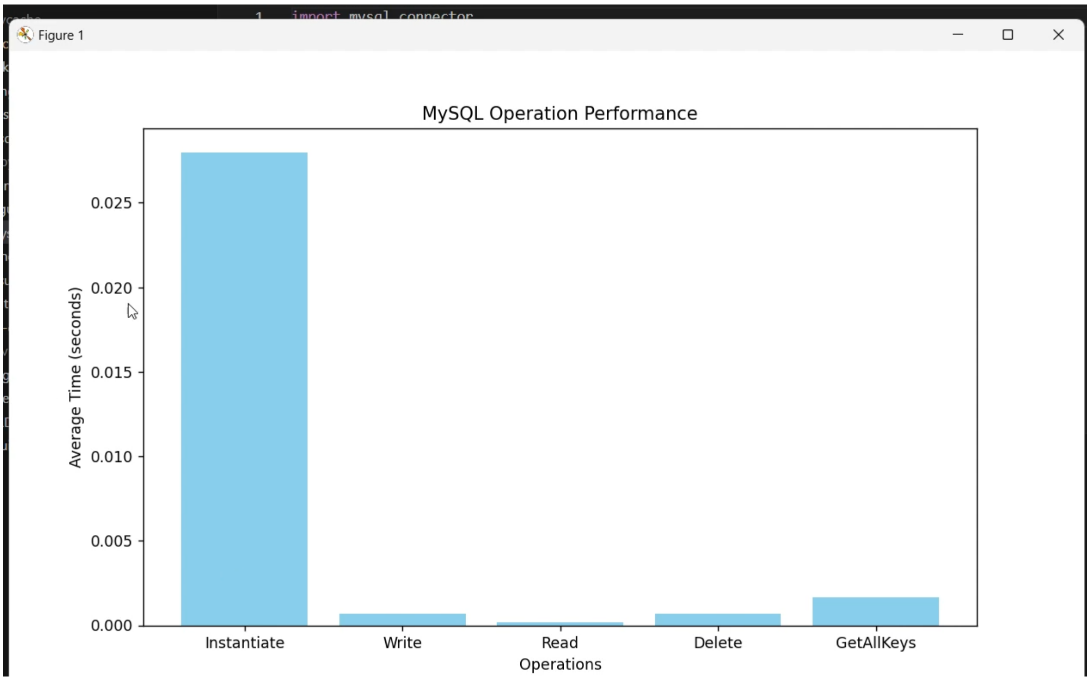
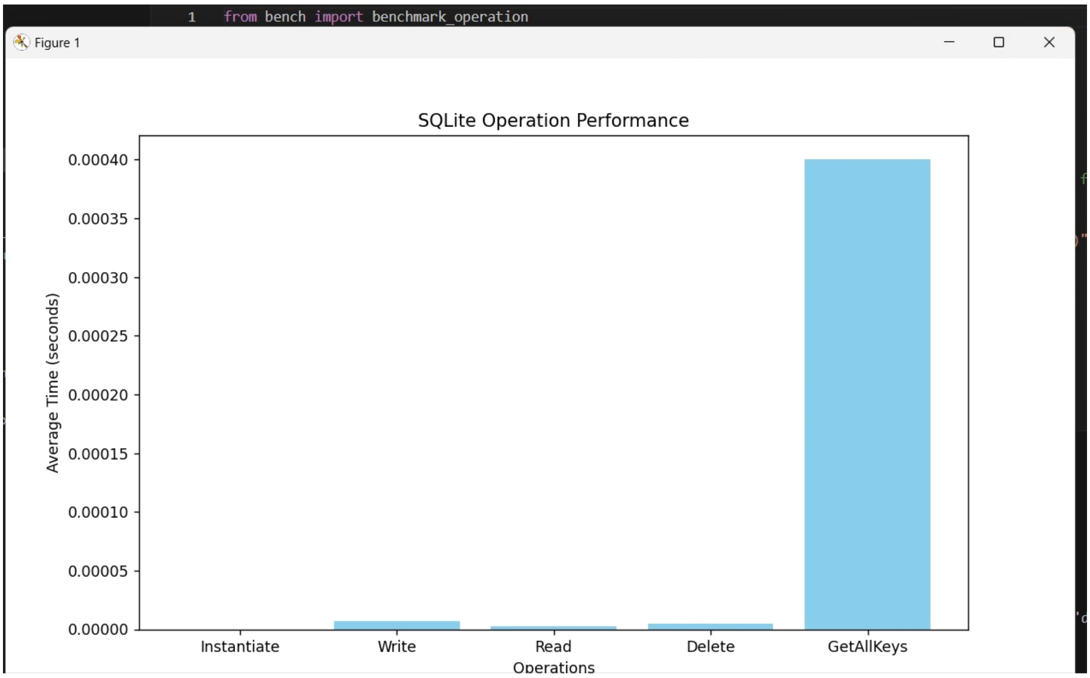


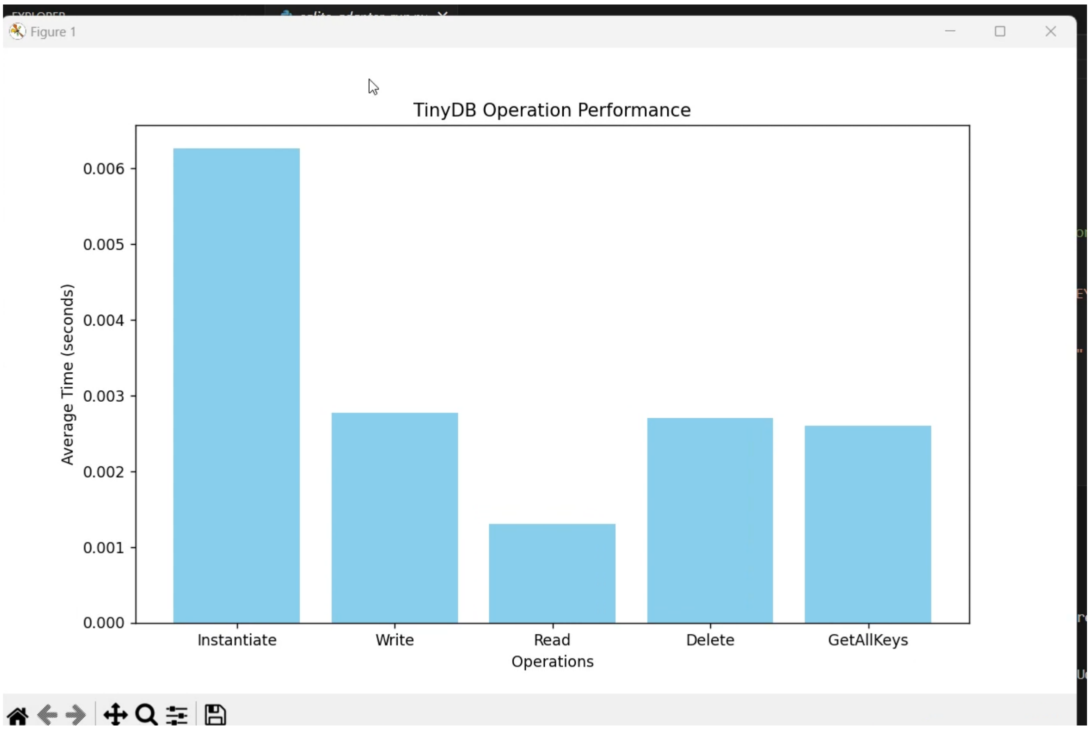
Figure 1
Microsoft SQL Server Express Operation Performance
Average Time (seconds)
0.0020
0.0015
0.0010
0.0005
0.0000
Instantiate
Write
Read
Delete
GetAllKeys
Operations



import mysql.connector
Figure 1
MySQL Operation Performance
Average Time (seconds)
0.025
0.020
0.015
0.010
0.005
0.000
Instantiate
Write
Read
Delete
GetAllKeys
Operations



```
from bench import benchmark_operation
```



3.2 Analysis

The following tables present a comparison of performance metrics for various databases across different operations. Each table shows the time taken per database for a specific operation, and a brief inference is provided below each table.

Table 1: Instantiation Time

Database	Instantiation Time (seconds)
CouchDB	1.48044
DuckDB	0.00020
MongoDB	0.01190
MS SQL Server	0.00228
MySQL	0.02800
SQLite	0.00000
TinyDB	0.00625

Table 3.1: Average time taken to instantiate (clear all records).

Inference: SQLite has the fastest instantiation time, showing near-zero delay. CouchDB, on the other hand, has a significantly higher instantiation time, which may indicate slower initialization routines.

Table 2: Average Write Time

Database	Average Write Time (seconds per record)
CouchDB	0.00694
DuckDB	0.00052
MongoDB	0.00040
MS SQL Server	0.00043
MySQL	0.00068
SQLite	0.00001
TinyDB	0.00277

Table 3.2: Average time taken to insert key-value pairs.

Inference: SQLite and MongoDB exhibit very fast write times, while CouchDB has the highest average write time among the databases tested, potentially due to its overhead with larger transactions.

Table 3: Average Read Time

Database	Average Read Time (seconds per record)
CouchDB	0.00133
DuckDB	0.00039
MongoDB	0.00033
MS SQL Server	0.00010
MySQL	0.00018
SQLite	0.00000
TinyDB	0.00130

Table 3.3: Average time taken to retrieve values by key.

Inference: MS SQL Server and SQLite have the quickest read times, with SQLite exhibiting near-zero read time. CouchDB and TinyDB display higher read times in comparison, which may affect retrieval speed in large datasets.

Table 4: Average Delete Time

Database	Average Delete Time (seconds per record)
CouchDB	0.00234
DuckDB	0.00044
MongoDB	0.00030
MS SQL Server	0.00022
MySQL	0.00068
SQLite	0.00000
TinyDB	0.00270

Table 3.4: Average time taken to delete key-value pairs.

Inference: SQLite and MS SQL Server again lead in efficiency, with the lowest delete times. CouchDB and TinyDB show the highest delete times, which could impact performance in applications with frequent deletions.

Table 5: GetAllKeys Time

Database	GetAllKeys Time (seconds)
CouchDB	0.02402
DuckDB	0.00040
MongoDB	0.00216
MS SQL Server	0.00143
MySQL	0.00167
SQLite	0.00040
TinyDB	0.00260

Table 3.5: Average time taken to fetch all keys.

Inference: DuckDB and SQLite demonstrate superior performance for fetching all keys. CouchDB's longer retrieval time may indicate limitations with key-fetching efficiency, especially when handling larger datasets.

3.3 Conclusion

- **SQL Databases (SQLite, MySQL, MS SQL Server)** generally exhibited faster performance across most operations, particularly in *Instantiation*, *Write*, *Read*, and *Delete* times. SQLite notably led in instantiation, write, read, and delete operations, often showing near-zero operation time per record. MS SQL Server also performed well, especially in read and delete times.
- **NoSQL Databases (CouchDB, MongoDB, TinyDB)** showed a range of performance, with MongoDB excelling in *Write* and *Read* operations, performing close to the top SQL databases. CouchDB, however, had slower times for instantiation and delete operations, which may indicate overhead associated with managing larger transactions and data replication processes common to some NoSQL setups.
- **Key Retrieval (GetAllKeys)** was faster in SQL databases (DuckDB, SQLite) than in most NoSQL counterparts, with CouchDB having the slowest performance in this area. This suggests that SQL databases can offer an advantage in applications requiring frequent retrieval of large key sets.

Overall Conclusion: SQL databases, particularly SQLite and MS SQL Server, outperformed NoSQL databases in most metrics. NoSQL databases like MongoDB, however, offer competitive write and read speeds, making them suitable for applications with high write throughput or flexible schema requirements. For tasks involving extensive data retrieval or transaction-heavy processes, SQL databases may provide superior overall performance.

Chapter 4

4.1 References

Y. Li and S. Manoharan, "A performance comparison of SQL and NoSQL databases," 2013 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM), Victoria, BC, Canada, 2013, pp. 15-19, doi: 10.1109/PACRIM.2013.6625441.

Abstract: With the current emphasis on “Big Data”, NoSQL databases have surged in popularity. These databases are claimed to perform better than SQL databases. In this paper we aim to independently investigate the performance of some NoSQL and SQL databases in the light of key-value stores. We compare read, write, delete, and instantiate operations on key-value stores implemented by NoSQL and SQL databases. Besides, we also investigate an additional operation: iterating through all keys. An abstract key-value pair framework supporting these basic operations is designed and implemented using all the databases tested. Experimental results measure the timing of these operations and we summarize our findings of how the databases stack up against each other. Our results show that not all NoSQL databases perform better than SQL databases. Some are much worse. And for each database, the performance varies with each operation. Some are slow to instantiate, but fast to read, write, and delete. Others are fast to instantiate but slow on the other operations. And there is little correlation between performance and the data model each database uses.