Comparative Performance Benchmarking of four Databases using TPC-H

Ali Eslamian*, Fairoz Nower Khan*, Nabuat Zaman Nahim*, Purushottam Panta*, A.B. Siddique
Department of Computer Science
University of Kentucky
Lexington, KY, USA
Email: {ali.eslamian, fairoz.khan, nabuat.nahim, purupanta, ab.siddique}@uky.edu

***Abstract*—This paper conducts a comparative study on the performance of SQLite, PostgreSQL, H2, and MySQL using the TPC-H decision support benchmark, designed to evaluate the performance of ad-hoc Decision Support System (DSS) queries. The TPC-H results typically involve large databases and are conducted on high-end hardware. We explore downsizing the benchmark to run on standard student laptop hardware and provide a guide for non-experts to conduct the test. We then present our benchmark test results, including a comprehensive evaluation of the four widely used database engines revealing insights into their performance and capabilities for handling complex analytical queries. Our results show that SQLite consistently outperforms PostgreSQL, H2, and MySQL in handling a range of queries, attributed to its efficient indexing and query optimization. While MySQL shows strong scalability for larger datasets, PostgreSQL emphasizes extensibility over speed. We also discuss concurrency handling, highlighting SQLite's suitability for smaller datasets and MySQL's robust optimization techniques. These findings offer valuable insights for database selection in diverse application scenarios.**

***Index Terms*—Database, TPC-H Benchmark, SQLite, PostgreSQL, H2, MySQL, OLTP: Online Transaction Processing; OLAP: Online Analytical Processing; TPC: Transaction Processing Performance Council; API: Application Programming Interface; MVCC: Multi-Version Concurrency Control**

I. Introduction

Databases play a critical role in modern software systems as they drive data storage, retrieval, and analysis. The choice of database can significantly impact application performance, scalability, and maintainability across various use cases. In this paper, we aim to conduct a comprehensive comparative analysis of four different databases using the industry-standard TPC-H benchmark. We found it interesting to benchmark these most used databases and see how they are different from each other.

Initially, we had the goal of gaining a deep understanding of the strengths and weaknesses of some selected databases in transactional (OLTP) and analytical (OLAP) workloads. We identified factors influencing database performance, including hardware configurations, database settings, and query complexity, intending to provide actionable insights and recommendations for database selection based on specific use-case requirements.

*These authors contributed equally to this work.

Our initial plan was to evaluate a diverse range of databases; however, resource constraints and limitations in the free versions of certain databases and other issues and considerations explained below prompted us to focus on four widely used databases [1]: SQLite, PostgreSQL, H2, and MySQL. Each of these databases offers unique features and capabilities, making them suitable candidates for benchmarking under the TPC-H benchmark to evaluate their performance in handling decision support systems.

Furthermore, in our updated plans, we refined our focus to exclusively evaluate the performance of these databases under the TPC-H benchmark only. This benchmark comprehensively evaluates the performance of each database and highlights the differences among them. Instead of selecting certain classes of queries from TPC-H and TPC-C, we decided to delve deeply into each database by executing all classes of queries in the TPC-H benchmark.

We selected random subsets of tuples from the standard TPC-H database to conduct our experiments. Additionally, we shifted from databases based on cloud platforms to those that could be locally hosted, aiming to evaluate performance under conditions closely resembling industry settings.

Despite initial plans to include MongoDB, SciDB, and MonetDB, limitations in the free versions and specific considerations led us to exclude them from our study. MongoDB, a document-oriented NoSQL database, offers flexibility and scalability for handling unstructured data. However, our focus on relational database management systems (RDBMS) for the TPC-H benchmark necessitated excluding MongoDB from our study. Similarly, SciDB, a specialized array-oriented database designed for scientific applications, provides powerful analytics capabilities for multidimensional data. Despite its strengths, SciDB's specialized use case and distinct data model were not aligned with the objectives of our study, which centered on evaluating RDBMS performance under the TPC-H benchmark. Challenges encountered with certain databases, such as MonetDB's lack of explicit support for viewing query execution times, necessitated adaptations to our approach. Hence, we opted to focus exclusively on SQLite, PostgreSQL, H2, and MySQL, as they are widely used RDBMS with relevance to our research objectives. Our experiment involved executing 22 queries across these databases.

By analyzing the performance of these databases under the TPC-H benchmark, we aim to provide valuable insights into their strengths and weaknesses, facilitating informed decision-making processes for database selection. Our findings will contribute to the body of knowledge on database performance evaluation and inform best practices in database management for diverse application scenarios.

The rest of the paper is structured as follows: In Section II, we review the related work in the field of database performance benchmarking and provide brief relevant introductions of the databases under consideration. Section III details the experimental setup, including database configurations and benchmarking procedures while also providing insights into TPC-H benchmarking for each database. Section IV presents the results and analysis including key findings for benchmarking and tables of numbers of the performance metrics obtained from our experiments executing the TPC-H benchmark across SQLite, PostgreSQL, H2, and MySQL. In Sections V and VI, we discuss the implications of our findings and conclude with avenues for future research while Section VII clearly explains what each of the authors did in contributing to this project and paper report.

II. Related Work

In exploring the efficiency of query subsets within the TPC-H benchmark, Vandierendonck and Trancoso (2006) introduce an effective method that reduces the TPC-H benchmark to just six queries while maintaining high representativeness [2]. This approach significantly reduces execution time (down to 40%) and is validated against the full benchmark execution across various architecture case studies. Barata, Bernardino, and Furtado (2015) provide an overview of the key characteristics and functionalities of the TPC-DS, TPC-H, and SSB benchmarks [3]. Their work is essential for evaluating and comparing the performance of different decision support systems. In another study, Chen et al. (2019) analyze the performance of the TPC-H benchmark on Exascale computing architecture, providing insights into the scalability and efficiency of database systems on large-scale distributed environments [4]. Gülcü and Kaya (2016) conducted a comparative performance analysis of PostgreSQL, SQLite, MongoDB, and Cassandra using the TPC-H benchmark, offering valuable insights into the strengths and weaknesses of these database systems in decision support environments [5]. Wang et al. (2017) evaluate the performance of MySQL and PostgreSQL on the TPC-H benchmark, providing a comparative analysis of these two popular relational database management systems in decision support scenarios [6].

A. SQLite

SQLite stands out as a lightweight, file-based database management system, ideal for scenarios where a full-fledged DBMS might be excessive [7]. Unlike server-based systems like MySQL or PostgreSQL, SQLite operates directly from a single file, eliminating the need for separate server installation and configuration. This file-based approach makes SQLite not only user-friendly for developers but also ensures high reliability for applications that require database functionality without the overhead of a complex system. This architecture makes it a perfect fit for embedded devices and small-scale applications, offering robust data management capabilities without compromising speed or data integrity [8].

B. PostgreSQL

PostgreSQL is an open-source relational database management system that stands out for its reliability, extensibility, and adherence to SQL standards. It is renowned for its robustness and stability and offers ACID properties to ensure data integrity in complex transactional scenarios. It scales vertically and horizontally, offering high-availability features like built-in replication and OLTP. Its broad ecosystem, seamless integration with programming languages and frameworks, and extensive documentation make PostgreSQL a popular choice for projects ranging from small-scale applications to large enterprise systems.

C. H2

H2 is a Java-based relational database management system that is open-source and lightweight. It supports standard SQL API and JDBC API, along with the PostgreSQL ODBC driver. It is known for its fast performance and low memory requirement. H2 can be easily integrated into Java applications or used as a database server. Users have the option to configure H2 databases either in-memory or disk-based setups. Additionally, H2 has strong security features. H2 is designed to be platform-independent, meaning it can run on various operating systems including Windows, Linux, and macOS. It is commonly used in Java applications and can be easily used on Windows systems. H2 is lightweight but supports both embedded and server modes, offering transactional support and a multi-version concurrency model, which is useful for testing in environments like SQLite with the capability to scale up slightly.

D. MySQL

A widely used open-source RDBMS which claims to have better performance than most of the databases. It is widely used in web applications and is better for scalability. MySQL, like PostgreSQL, is a popular choice for web-based applications and offers strong performance for OLTP (Online Transaction Processing) systems. It combines simplicity in setup and management with a powerful query engine, making it a versatile choice for a wide range of applications. MySQL has a large and active community of developers, DBAs, and enthusiasts who contribute not only to its development but also provide support, and share knowledge through forums, mailing lists, and conferences.

III. EXPERIMENT SETUP

The benchmarking experiment was set up in a laptop with the most common configurations - 16GB of memory with an Intel Core i7 processor and Windows 11.

Python scripts [9], [10] were written to import the TPC-H schema structures, metadata and tuples into the database engines to measure the relative performances for our experimental purposes. TPC-H standard databases of 3k, 12k, 60k and 120k tuples were imported into the databases.

A. Data selection methodology

TPC-H-SF1 refers to the TPC-H benchmark executed at a scale factor of 1, which corresponds to a database size of approximately 1GB. This scale factor is commonly used to evaluate the performance of database and data warehousing solutions when handling relatively small data volumes.

The dataset was split into four different-sized pieces. For each piece, tuples were randomly selected until the desired dataset size was achieved. The main file consists of 8 tables with a total of 150,000 tuples. All 8 tables were included in the dataset but with varying numbers of tuples: 3,000, 12,000, 60,000, and 120,000. Despite being smaller than the full-scale dataset, the last two datasets (60,000 and 120,000 tuples) were still large enough to potentially cause process crashes in certain cases.

B. TPC-H Benchmarking

The TPC-H benchmark serves as a valuable tool for evaluating database systems like SQLite, PostgreSQL, H2, and MySQL. It is particularly relevant in the context of decision support systems, where complex queries and large volumes of data are common. By simulating real-world scenarios, TPC-H assesses each database's analytical capabilities, measuring factors such as query execution efficiency, throughput under load, and scalability. This standardized benchmark enables meaningful and directly comparable performance assessments across these databases.

C. SQLite

We used a bash script to run the code in order[1]. It organizes its operations around three main directories: one for databases (Databases), one for SQL query files (Queries), and one for output results (Results). Initially, the script ensures that the output directory exists and then gathers all SQL files from the specified directory.

For each database file in the Databases directory, the script creates a new results file intended to log performance metrics for the queries executed against that database. It then iterates over each SQL file, executing the query within, on the current database using sqlite3. For each execution, it measures the time taken, computes the throughput (queries per second), and uses the *perf* command to measure CPU cycles per second. These metrics-execution time, throughput, and CPU cycles are logged into the database-specific results file. Finally, the script prints out all results from this file to the console, providing a clear performance summary for each database with respect to the set of queries tested.

[1]The code is available at
https://github.com/aseslamian/SQLite-TPCH-Benchmark

D. PostgreSQL

We started by installing and setting up PostgreSQL on the system and created each database using the graphical interface pgAdmin. Once connected, we used the SQL commands[2] to upload the databases from the .sql files. Finally, we used the SQL queries for the TPC-H benchmark for each database and used the statistics time query to get the CPU time for each query. The results of each query with CPU time are visible at the pgAdmin console which gives an understanding of the PostgreSQL performance.

E. H2

We downloaded the Windows Installer from [11] and then ran the installer following the steps in the setup window to install the H2 database in the computer specified. We started the H2 database server by running the 'h2.bat' script that is included in the H2 database installation. After connecting to the database and when the server was running, we connected to the H2 database using a client application, which was accessed by opening a web browser (Mozilla Firefox used as recommended in the documentation) [12]. After connecting, we uploaded the databases from the .sql files[3] using the SQL commands and then executed the SQL queries for the TPC-H benchmark for each database in the H2 console and recorded the execution time from the interface which gave us an idea of the performance of H2 compared to our other selected databases.

F. MySQL

Python script was written to import the TPC-H schema structures, metadata and tuples into the database engine. TPC-H standard databases of 3k, 12k, 60k, and 120k tuples were imported into MySQL and considered for the experiment. The original TPC-H data needed to be converted to .sql format with the help of a Python script that we wrote and were successfully able to create a performance test bench environment at MySQL end. Python scripts[4] and MySQL workbench was set up to measure and verify the execution time and throughput of the query execution. The Python script has a basic configuration setting such as with the database (3k, 12k, 60k, or 120k) we want to run our TPC-H query on, and we can specify if we want to run all or certain queries on the specified database. It finally generates a ".csv" file with the performance parameters such as execution time and throughputs for each specified query on each specified dataset.

IV. RESULT AND ANALYSIS

In this section, we first present the implementation times in the form of a table of numbers which are then appropriately complemented with relevant explanations and commentary highlighting key findings for benchmarking.

[2]The code is available at
https://github.com/FairozNowerKhan/DatabaseProject

[3]The code is available at
https://github.com/Nabuat/CS-505-DB-Project

[4]The code is available at
https://github.com/purupanta/Courses/tree/main/uky_cs505/project.

We picked 3k, 12k, 60k, and 120K tuples from the standard set of TPC-H. The following tables show the details of our experimental results when executing the standard TPC-H queries (query 1 through query 22) in the four database engines.

TABLE I
TPC-H 3K PERFORMANCE RESULTS: EXECUTION TIMES IN SECONDS

	SQLite	PgSQL	H2	MySQL
Query 1	0.050	0.089	0.089	0.006
Query 2	0.009	1.952	1.952	0.077
Query 3	0.008	0.897	0.897	0.001
Query 4	0.017	0.111	0.111	0.003
Query 5	0.014	0.984	0.984	0.008
Query 6	0.002	0.014	0.014	0.001
Query 7	0.007	0.1077	0.1077	0.001
Query 8	0.005	0.982	0.982	1.900
Query 9	0.009	1.479	1.479	0.008
Query 10	0.007	0.08	0.08	0.048
Query 11	0.010	1.23	1.23	0.001
Query 12	0.004	0.085	0.085	0.017
Query 13	0.007	1.603	1.603	0.010
Query 14	0.006	0.089	0.089	0.004
Query 15	0.010	0.482	0.482	0.007
Query 16	0.014	0.776	0.776	0.019
Query 17	0.017	0.006	0.006	0.001
Query 18	0.009	0.02	0.02	0.001
Query 19	0.469	15.888	15.888	0.016
Query 20	0.006	4.297	4.297	0.067
Query 21	0.133	5476	5476	0.01
Query 22	0.007	1.94	1.94	0.017

TABLE II
TPC-H 12K PERFORMANCE RESULTS: EXECUTION TIMES IN SECONDS

	SQLite	PgSQL	H2	MySQL
Query 1	0.013	0.389	0.043	0.0470
Query 2	0.031	0.403	24.788	54.232
Query 3	0.020	0.245	13.582	0.070
Query 4	0.250	0.418	1.281	0.020
Query 5	0.232	4.402	117.609	1.472
Query 6	0.004	0.199	0.01	0.020
Query 7	0.020	0.203	17.726	0.031
Query 8	0.026	0.93	15.443	3.484
Query 9	0.029	0.213	23.542	1.406
Query 10	0.019	0.122	1.359	0.380
Query 11	0.028	0.224	5.46	0.040
Query 12	0.009	0.214	0.542	0.030
Query 13	0.020	0.13	25.638	0.100
Query 14	0.009	0.11	1.092	0.020
Query 15	0.013	0.157	2.78	0.030
Query 16	0.014	0.225	10.649	0.065
Query 17	0.020	0.109	0.248	0.030
Query 18	0.011	0.99	0.294	0.020
Query 19	7.123	0.124	98.126	0.801
Query 20	0.103	20.025	0.906	1.903
Query 21	2.014	0.255	25.582	0.050
Query 22	0.020	0.13	23.588	0.040

TABLE III
TPC-H 60K PERFORMANCE RESULTS: EXECUTION TIMES IN SECONDS

	SQLite	PgSQL	H2	MySQL
Query 1	0.061	0.605	0.641	0.215
Query 2	0.097	3.49	518.467	35.406
Query 3	0.059	0.273	314.16	0.332
Query 4	0.060	0.571	521.227	0.127
Query 5	3.943	0.495	34205	9.627
Query 6	0.011	0.182	15.759	0.021
Query 7	0.073	0.216	327.944	0.188
Query 8	0.098	0.147	373.525	2.406
Query 9	0.067	0.273	541.438	2.968
Query 10	0.071	0.148	304.36	3.596
Query 11	0.089	0.235	397.45	0.125
Query 12	0.033	0.248	6.533	0.133
Query 13	0.140	0.194	513.669	0.986
Query 14	0.035	0.115	9.939	0.105
Query 15	0.025	0.533	107.63	0.085
Query 16	0.045	0.305	138.24	1.738
Query 17	0.093	0.169	1.011	1.799
Query 18	0.072	0.146	0.381	0.447
Query 19	3.021	0.185	1465.859	0.400
Query 20	0.062	850	9.406	53.170
Query 21	62.285	0.824	433.182	1.041
Query 22	0.089	0.165	410.788	0.246

TABLE IV
TPC-H 120K PERFORMANCE RESULTS: EXECUTION TIMES IN SECONDS

TPCH120K	SQLite	PgSQL	H2	MySQL
Query 1	0.114	0.71	0.635	0.352
Query 2	0.208	436.103	2209.092	126.625
Query 3	0.228	0.912	1842.17	1.781
Query 4	30.900	0.658	74.528	0.230
Query 5	26.847	0.778	-	3.712
Query 6	0.021	0.385	0.06	0.141
Query 7	0.166	0.406	-	1.516
Query 8	0.289	0.315	-	6.812
Query 9	0.346	0.571	-	4.094
Query 10	0.252	0.288	-	5.109
Query 11	0.202	0.47	-	0.358
Query 12	0.071	0.394	-	0.373
Query 13	0.354	0.665	-	1.976
Query 14	0.082	0.253	-	0.252
Query 15	0.098	0.837	-	0.141
Query 16	0.13	0.617	-	2.498
Query 17	0.223	5.452	-	11.991
Query 18	0.158	0.497	-	0.804
Query 19	930.034	0.268	-	78.697
Query 20	0.138	2905	-	218.399
Query 21	266.479	0.419	-	2.315
Query 22	0.216	0.658	-	0.906

Execution times for queries marked with "-" were not obtainable due to the respective database and system constraints resulting in prolonged execution durations making them not useful for our study.

The TPC-H benchmark consists of a set of complex SQL queries designed to simulate decision support systems for businesses like retailers, suppliers, and manufacturers. These queries involve joining several tables representing various aspects of a fictional company's operations, such as orders, line items, suppliers, parts, and customers. Each query typically involves selecting certain fields from multiple tables, applying various filters and aggregations, and sometimes sorting the results. These operations almost always require joins to connect the related data across different tables [13].

We tested the benchmark on SQLite, PostgreSQL, H2, and MySQL databases which can serve as OLTP (Online Transaction Processing) databases. They are widely used in various industries and applications to support high-volume

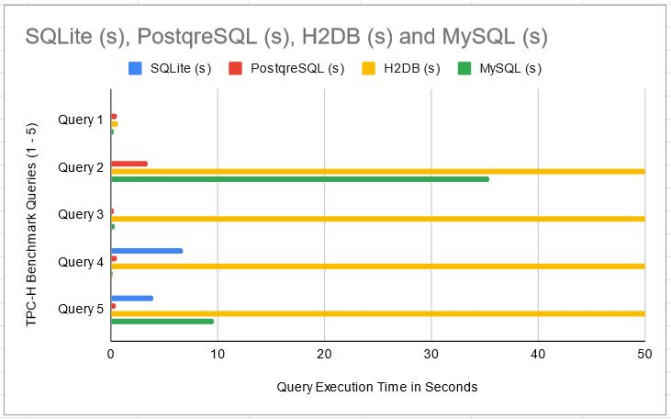

Fig. 1. Execution time of the first 5 queries on SQLite, PostgreSQL, H2 and MySQL

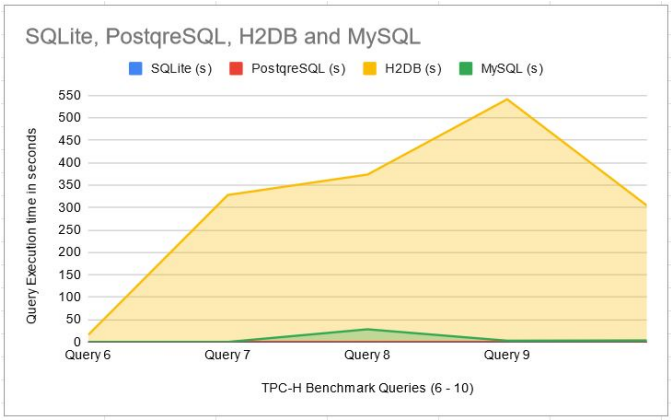

Fig. 2. Execution time of the queries (6-10) on SQLite, PostgreSQL, H2 and MySQL

transactional workloads with real-time data processing requirements.

The first 10 queries of the TPC-H benchmark measure various aspects of a database's performance and capabilities, focusing primarily on data retrieval, aggregation, and filtering. These queries simulate real-world scenarios involving sales, order processing, and supplier management, providing a comprehensive evaluation of a database system's performance and efficiency in handling complex analytical queries.

Databases like PostgreSQL, MySQL, and SQLite are designed with relational data in mind and are optimized for handling joins efficiently. They use algorithms like nested loop joins, hash joins, and merge joins depending on the query and data distribution.

TPC-H Query 1 is the pricing summary report query that includes all line items shipped before a specified date. The query's execution time varies across databases: it's fastest in SQLite (0.061 s), followed by MySQL (0.215 s), indicating SQLite's efficient indexing; PostgreSQL (0.605 s) and H2 (0.641 s) show longer times due to differing optimization approaches.

TPC-H Query 2 is the Minimum Cost Supplier Query which finds, in a given region, for each part of a certain type and size, the supplier who can supply it at minimum cost. The query shows significant performance disparities across databases, with SQLite exhibiting the fastest execution time (0.097 s) due to its lightweight nature, while H2 experiences a substantial slowdown (518.467 s) due to optimization challenges; MySQL (35.406 s) and PostgreSQL (3.49 s) demonstrate intermediate performance, indicating variations in query processing efficiencies, data pruning, and optimization strategies.

TPC-H Query 3, which is the Shipping Priority Query retrieves the 10 unshipped orders with the highest value. SQLite again excels due to its simplicity and optimized handling of lightweight queries, while PostgreSQL and MySQL offer a balance between performance and robust features, with H2 demonstrating performance issues stemming from suboptimal resource management and resource allocation.

TPC-H Query 4 is the Order Priority Checking Query that counts the number of orders in a given quarter of a given year in which at least one line item was received by the customer later than its committed date. The query lists the count of such orders for each order priority sorted in ascending priority order. SQLite's limited optimization capabilities result in slower execution for this specific query, while PostgreSQL and MySQL demonstrate efficient query optimization and indexing, leading to faster performance. H2's performance issues are due to sorting and aggregation-heavy queries, suboptimal query planning, and resource management strategies.

TPC-H Query 5 is the Local Supplier Volume Query which lists the revenue volume done through local suppliers. SQLite's limitations in optimization and parallelism result in slower performance again here, while PostgreSQL benefits from its advanced query optimization techniques. H2's performance issues are again from possible resource constraints, while MySQL demonstrates comparatively efficient processing but lags PostgreSQL due to differences in query optimization approaches.

TPC-H Query 6 is the Forecasting Revenue Change Query that quantifies the amount of revenue increase that would have resulted from eliminating certain companywide discounts in a given percentage range each year. Asking this type of "what if" query can be used to look for ways to increase revenues.

TPC-H Query 7 is the Volume Shipping Query which determines the value of goods shipped between certain nations to help in the re-negotiation of shipping contracts.

TPC-H Query 8 is the National Market Share Query that determines how the market share of a given nation within a given region has changed over two years for a given part type.

TPC-H Query 9 is Product Type Profit Measure Query which determines how much profit is made on a given line of parts, broken out by supplier nation and year.

TPC-H Query 10 is the Returned Item Reporting Query that identifies customers who might be having problems with the parts that are shipped to them. The query aggregates revenue data for customers who returned items within a three-month

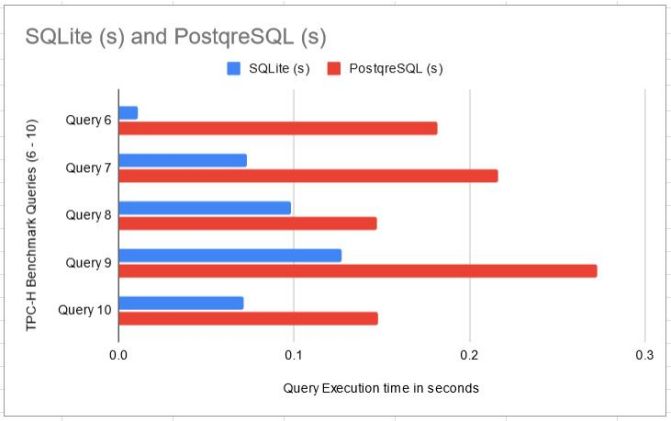

Fig. 3. Execution time of the queries (6-10) on SQLite and PostgreSQL only

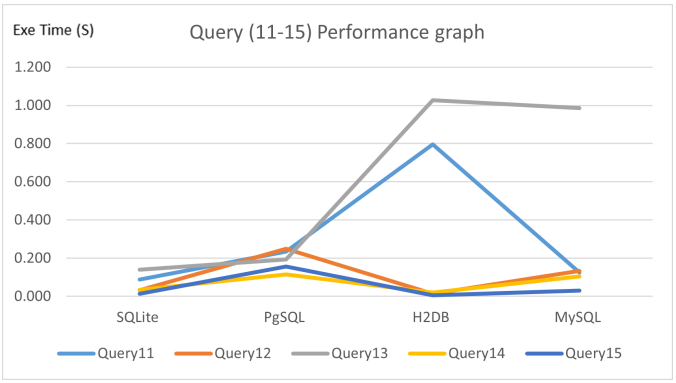

Fig. 4. Execution graph of the queries (11-15) on four of our selected database engines. (Query 11-15: H2 DB performance was downsized by 500-fold to fit in the graph frame.)

period from a specified date.

The significant performance disparity across databases for all these queries is due to optimization differences, with SQLite demonstrating the fastest execution due to its efficient handling of the query's structure, while H2 exhibits the slowest performance, due to suboptimal query planning and resource constraints. PostgreSQL and MySQL show intermediate performance, leveraging their respective optimization mechanisms.

In particular, as illustrated in Figure 3, PostgreSQL does not perform better than SQLite due to various reasons. Here, speed is imperative. At the expense of speed, PostgreSQL was designed with extensibility and compatibility in mind. If a project requires the fastest read operations possible, PostgreSQL may not be the best choice of DBMS. Because of its large feature set and strong adherence to standard SQL, PostgreSQL can be overkill for simple database setups. For read-heavy operations where speed is required, MySQL is typically a more practical choice. Also in terms of complex replication, although PostgreSQL does provide dedicated support for replication, it's still a new feature. Some configurations, like a primary-primary architecture, are only possible with extensions. Replication is a more mature feature of MySQL and many users see MySQL's replication to be easier to implement, particularly for those who lack the requisite database and system administration experience.

TPC-H Query 11 is designed to simulate a complex business analysis query typically performed in data warehousing environments. It involves complex join operations on several tables to simulate real-world analytical queries.

TPC-H Query 12 is designed to simulate a business analysis query related to order shipping modes and order priorities. The query evaluates the efficiency of the database system in handling complex analytical queries with multiple join conditions and aggregations.

TPC-H Query 13 is designed to analyze customer distribution based on the number of orders they have placed, while also filtering out orders with specific characteristics. This type of query basically measures the performance of a database system in efficiently processing complex analytical queries involving multiple join operations and aggregations. Handling multiple join operations, aggregate data, filtering and scalability (operating significant volume of the data) tests are the primary parts of query 13. SQLite is the winner with 0.140 followed by PostgreSQL with 0.194. H2 performed significantly slow especially when working with a large volume of data operations.

TPC-H Query 14 is designed to analyze the revenue generated from promotional and non-promotional items within a specified time period. The query 14 basically serves as a benchmark to assess the overall performance of a database system in executing complex analytical queries typical in decision support and data warehousing scenarios, particularly those involving join operations, aggregations, and filtering based on specific criteria. SQLite is performing much better with 0.035 followed by MySQL with 0.105 as can be seen from our experimental results in Table V. H2 is performing worst, taking more than 9 seconds for this type of query.

TPC-H Query 15 is designed to identify the supplier with the highest total revenue generated from sales during a specific three-month period. While running query 15, the database system performs various operations like filtering, aggregation (finding sum), and joining tables. The clear winner is again SQLite with just 0.013 seconds to execute the query, followed by MySQL 0.030 seconds. H2 is performing the worst in this type of query as well taking more than 2 seconds in execution.

TABLE V
QUERY EXECUTION TIMES (IN SECONDS)

	SQLite	PgSQL	H2	MySQL
Query 11	**0.089**	0.235	397.45	0.125
Query 12	**0.033**	0.248	6.533	0.133
Query 13	**0.140**	0.194	513.669	0.986
Query 14	**0.035**	0.115	9.939	0.105
Query 15	**0.013**	0.157	2.78	0.030

The resulting query execution time of all databases for queries 11 through 15, on dataset of size 60k.

In the experiment on queries 11 to 15, we can see that SQLite is mostly outperforming other database engines.

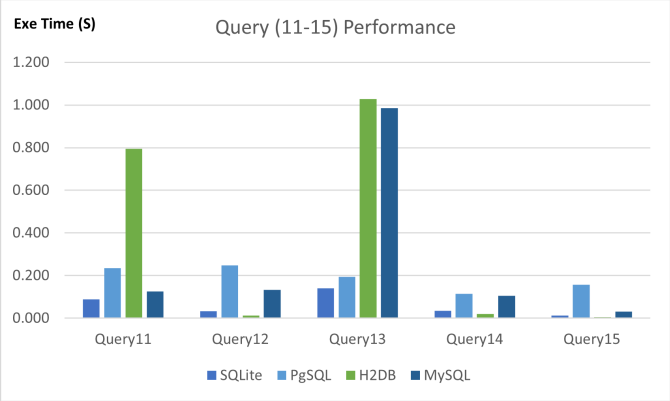

Fig. 5. Execution time of the queries (11-15) on four of our selected database engines. (Query 11-15: H2 DB performance was downsized by 500-fold to fit in the graph frame.)

Fig. 6. Comparison of Execution time for Class Query 21

SQLite performs great for smaller data sizes and our dataset in the experiment is not too large, so it's more advantageous for SQLite to perform better than others. However, if we pick the complete TPC-H database or any other dataset having millions of tuples, it most likely won't be able to outperform others. PostgreSQL and MySQL will most likely be the winner in larger datasets while H2 is mainly designed for moderate workloads and datasets.

Queries 16-22 cover various aspects of database operations. Query 16 focuses on the system's ability to handle complex multi-way joins, aggregations, and anti-joins, particularly with large lists and selection criteria. Query 17 tests nested queries and aggregations based on derived calculations, emphasizing efficiency in correlated subqueries. Query 18 evaluates sorting, grouping, and joining capabilities, particularly with large lineated tables, assessing the database's ability to manage and efficiently produce ordered outputs from large datasets. Query 19 examines the execution of multiple predicate selections including range, equality, and compound predicates, focusing on predicate evaluation under complex conditions. Query 20 looks at the database's proficiency in executing complex joins, subqueries, and aggregations, checking the effectiveness of its query optimizer in managing nested and dependent queries, thereby providing a comprehensive view of the database's operational capabilities under various complex data-processing scenarios.

TABLE VI
QUERY EXECUTION TIMES (IN SECONDS)

	SQLite	PostgreSQL	H2	MySQL
Query 16	**0.02**	0.44	49.89	0.61
Query 17	**0.04**	0.09	0.42	0.61
Query 18	**0.03**	0.39	0.23	0.16
Query 19	128.61	**5.40**	526.62	7.07
Query 20	**0.03**	291.44	4.87	18.38
Query 21	21.48	1825.69	1978.25	**0.37**
Query 22	**0.04**	0.75	145.44	0.10

SQLite outperforms other databases in most scenarios due to its lightweight architecture.

Table VI indicates the results for the average of three scales 3K, 12K, and 60K datasets. SQLite clearly performs best with consistent superiority across all data sizes and queries, making it ideal for enterprise-level applications demanding intensive data processing. MySQL also boasts strong performance, particularly with larger datasets, demonstrating good scalability and efficient handling of complex queries, ideal for large-scale applications. In terms of implementation time, H2 has difficulties due to involving sorting and aggregation-heavy queries. It may require further optimization for large-scale use or be better suited for prioritizing smaller datasets.

Figure 6 indicates the execution time for Query 21 to compare these databases in terms of complex joins and subquery executions. PostgreSQL and H2 are less optimized, potentially due to configuration issues and a lack of specific optimizations. In contrast, MySQL's exceptional speed suggests it excels at handling such query patterns, due to efficient index utilization and superior query planning strategies. SQLite strikes a middle ground with a moderate execution time. Its simpler query execution approach might scale better with increasing data complexity when compared to the other databases.

Our comprehensive evaluation of SQLite, PostgreSQL, H2, and MySQL under the TPC-H benchmark revealed insightful findings regarding their performance and capabilities in handling complex analytical queries. Across the range of queries tested, SQLite consistently demonstrated superior performance, owing to its efficient indexing and optimized handling of lightweight queries. MySQL also emerged as a strong contender, particularly with larger datasets, showcasing robust scalability and efficient query processing. PostgreSQL, while exhibiting competitive performance, did not surpass SQLite due to its emphasis on extensibility and compatibility over speed. H2 faced challenges, particularly with sorting and aggregation-heavy queries, highlighting potential optimization requirements for large-scale deployments. Overall, SQLite's consistent superiority underscores its suitability for enterprise-level applications requiring intensive data processing, while MySQL's scalability makes it an ideal choice for large-scale deployments. These findings provide valuable insights into

the performance characteristics of each database, facilitating informed decision-making processes for database selection in diverse application scenarios.

V. Conclusion

There are variations in the performances of these four database engines mainly because of their designs. SQLite is not designed for high scalability or concurrent access. It performs well for small to medium-sized datasets and so is often suitable for use on embedded applications for local storage. Since our evaluated experimental dataset is not big, SQLite is showing better performance in most cases. On the other hand, PostgreSQL and MySQL are quite capable of performing better. H2 scales well for moderate workloads but may face limitations in highly concurrent or large-scale deployments compared to our other databases like PostgreSQL or MySQL.

In terms of Concurrency Handling, SQLite is just providing basic locking and so may lead to contention in highly concurrent scenarios, so it is recommended for single-user or low-concurrency scenarios. On the other hand, PostgreSQL and MySQL support multi-version concurrency control (MVCC) in combination with locking to provide more strict concurrency. H2, on the other hand, supports multi-threaded access and uses a combination of locking and MVCC for concurrency control. It allows for concurrent read operations and serializes write operations to ensure data consistency. As our test is more focused on query performance than testing the concurrency, SQLite showed better performance in most of the queries, which could have been another way if the tests were performed by multiple users executing queries concurrently.

In terms of query optimization, SQLite's query optimizer is relatively basic compared to server-based databases like PostgreSQL or MySQL. However, it can efficiently handle simple queries on smaller datasets. PostgreSQL has a sophisticated query optimizer that generates efficient execution plans for complex queries. It supports various indexing options, query hints, and advanced optimization techniques. H2 also provides a fairly robust query optimizer and supports various optimization techniques such as indexing, query caching, and query hints. It can efficiently handle complex queries on medium-sized datasets. On the other hand, MySQL also has a highly efficient query optimizer that generates efficient execution plans for complex queries. It also supports various indexing options, query hints, and optimization techniques for improving query performance. However, due to the small dataset size and since the experiment was performed in a one-time-run testing, most of the advanced optimization techniques (such as query hints) from PostgreSQL and MySQL were not very beneficial to our experiments and the performance seems to be lower than expected in MySQL or PostgreSQL. However, in multiple-time execution testing, better performance can be seen for these databases.

Our experiment still needs to consider more scenarios such as a performance test while executing TPC-H queries through multiple users concurrently [14]. We consider such a test also to be our future extension in this benchmarking analysis. On the other hand, we can include more tuple-dataset in our experiment to more fairly judge the performance. It is because some database engines such as SQLite execute great on small datasets but others like MySQL or PostgreSQL are great for large datasets, and picking small-size datasets may be considered as biased in favor of SQLite.

VI. Future Work

The main focus of this paper is the performance evaluation of four widely used database engines, SQLite, PostgreSQL, H2, and MySQL. We can extend our study to include more database engines and some also with vector-database, graph-database, and so on.

We have taken a widely used TPC-H benchmarking for the performance evaluation. We can also take more benchmarking systems such as TPC-DS, AuctionMark [15], H-Store [16] benchmark in our evaluation process, which potentially gives the database engine's performances on more query scenarios.

VII. Contributions

Authors Ali Eslamian, Fairoz Nower Khan, Nabuat Zaman Nahim, and Purushottam Panta worked on the databases SQLite, PostgreSQL, H2, and MySQL respectively. The full details of exactly how each of us implemented the benchmarking problem in the respective databases are explained in the Experimental Setup Section. All of us worked equally in discussing and analyzing the results for the benchmarking problem and in writing this paper report.

References

[1] Db engines ranking. https://db-engines.com/en/ranking. Accessed: April 3, 2024.

[2] Hans Vandierendonck and Pedro Trancoso. Building and validating a reduced tpc-h benchmark. In *14th IEEE International Symposium on Modeling, Analysis, and Simulation*, pages 383–392. IEEE, 2006.

[3] Melyssa Barata, Jorge Bernardino, and Pedro Furtado. An overview of decision support benchmarks: Tpc-ds, tpc-h and ssb. *New Contributions in Information Systems and Technologies: Volume 1*, pages 619–628, 2015.

[4] Jianmin Chen et al. Performance characterization of the tpc-h benchmark on the exascale computing architecture. *IEEE Transactions on Parallel and Distributed Systems*, 30(5):1062–1077, 2019.

[5] Adnan Gülcü and Hüseyin Avni Kaya. Performance comparison of postgresql, sqlite, mongodb, and cassandra on tpc-h benchmark. In *International Conference on Information Technology: New Generations*. IEEE, 2016.

[6] Lei Wang et al. Performance evaluation of mysql and postgresql on tpc-h benchmark. In *2017 International Conference on Information System and Artificial Intelligence*. IEEE, 2017.

[7] Kevin P Gaffney, Martin Prammer, Larry Brasfield, D Richard Hipp, Dan Kennedy, and Jignesh M Patel. Sqlite: past, present, and future. *Proceedings of the VLDB Endowment*, 15(12), 2022.

[8] Mike Owens and Grant Allen. *SQLite*. Apress LP New York, 2010.

[9] purupanta. Python script for data conversion. https://github.com/purupanta/Courses/blob/main/uky_cs505/project/SQLiteToMySQL.ipynb, 2024. Accessed: April 23, 2024.

[10] Aseslamian. Python for tpc-benchmark performance measure. https://github.com/aseslamian/SQLite-TPCH-Benchmark, 2024. Accessed: April 23, 2024.

[11] H2 Database. Downloads. http://www.h2database.com/html/download.html. Accessed: April 7, 2024.

[12] H2 Database. Home. http://www.h2database.com/html/main.html. Accessed: April 7, 2024.

[13] DigitalOcean. Sqlite vs. mysql vs. postgresql: A comparison of relational database management systems. https://www.digitalocean.com/community/tutorials/sqlite-vs-mysql-vs-postgresql-a-comparison-of-relational-database-management-systems. Accessed: April 17, 2024.

[14] AuctionMark. Oltp benchmark. https://cs.brown.edu/media/filer_public/02/c8/02c8db15-c486-4194-abba-f5a6a1463e2b/visawee.pdf. Accessed: Date Accessed.

[15] Yan Zhang, Zhi-Feng Chen, and Yuan-Yuan Zhou. Efficient execution of multiple queries on deep memory hierarchy. *Journal of Computer Science and Technology*, 22:273–279, 2007.

[16] H-Store. H-store. https://hstore.cs.brown.edu/. Accessed: [specific date you accessed the site].