Journal of Information Systems Engineering and Management
2024, 9(2), 26126
e-ISSN: 2468-4376
https://www.jisem-journal.com/

Research Article

Enhancement of NoSQL Database Performance Using Parallel Processing

Inas Ismael Imran [1*]

[1] Department of Computer, College of Education for Women, University of Baghdad, Baghdad, Iraq
* **Corresponding Author:** enas.i@coeduw.uobaghdad.edu.iq

Citation: Ismael Imran, I. (2024). Enhancement of NoSQL Database Performance Using Parallel Processing. *Journal of Information Systems Engineering and Management*, *9*(2), 26126. https://doi.org/10.55267/iadt.07.14670

ARTICLE INFO

Received: 19 Jan 2024
Accepted: 19 Apr 2024

ABSTRACT

In the burgeoning realm of big data, document-oriented NoSQL databases stand out for their flexibility and scalability. This paper delves into the optimization of these databases, specifically through the lens of parallel processing techniques. A comparative study was conducted against the traditional non-parallel approaches, where marked performance enhancements were observed. For instance, the execution time for retrieving movies of a specific year decreased by over 80% when parallel processing was applied, plummeting from 1.578765 seconds to a brisk 0.300000 seconds. Memory usage and CPU utilization were meticulously recorded, revealing up to a 70% reduction in peak memory consumption in certain queries, and a moderate fluctuation in CPU usage between 49.25% to 75.2%. This indicates not only improved efficiency but also a prudent utilization of system capacity, without overtaxing resources. However, the study identified scenarios, such as highly complex queries, where the gains from parallel processing were less pronounced, suggesting a marginal improvement in CPU utilization. While the findings advocate for the adoption of parallel processing in handling intensive data retrieval tasks, it is recommended that future research should further scrutinize the scalability thresholds and explore alternative parallelization strategies to fortify the efficacy of document-oriented NoSQL databases.

Keywords: NoSQL, Parallel Processing, Complexity, Distributing, Parallel Execution.

INTRODUCTION

The landscape of database technology has undergone significant evolution since the late 1960s when the groundwork for what would come to be known as NoSQL databases was first laid. These databases were designed to store and retrieve data without tabular relations (Xu & Kostamaa, 2009). The phrase "NoSQL," which emerged in the early 21st century, refers to a variety of database technologies that meet the massive scalability needs of big data and the dynamic needs of real-time online applications. After Carlo Strozzi coined the word to describe an open-source relational database in 1998, Jon Oskarsen arranged a major non-relational database conference in San Francisco in 2009, reviving it. As Web 2.0 took off, NoSQL databases like Amazon's Dynamo and Google's BigTable underpinned emerging tech giants' data architecture and enabled Apache's Cassandra's advanced development (Lith & Mattsson, 2010). Cassandra wrote a 50 GB dataset 2500 times quicker than MySQL, showing that NoSQL databases can manage data better (Mihai, 2020; Mohmmed,2011).

By 2015, the RDBMS industry was worth $35.9 billion, however traditional database suppliers lost 1.5% to 5.6% of their market share over five years. **Figure 1** depicts the reduction in traditional RDBMS market share and the strong rise in NoSQL data storage revenue from $500 million in 2015 to $3.5 billion by 2019 (Zhang et al., 2016). This trend emphasizes the need for databases that can efficiently handle growing data volumes and complexity.

Datasets so massive and complicated that typical data-processing software cannot handle them are called "big data". Meanwhile, "parallel processing" has helped solve these problems by spreading computing work over numerous processors to boost efficiency and speed up processing (Adrian, 2016). Parallel processing uses a distributive philosophy to speed up computations, unlike single-processor models.

Despite its schema flexibility, scalability, and agility, NoSQL databases struggle to provide real-time data access and sophisticated analytics (Zhang et al., 2016; Hasan et al., 2019). NoSQL databases' parallel processing is promising, suggesting data management's future is optimizing large-scale data processes. To be resilient, agile, and intelligent, big data toolkits need NoSQL databases and parallel computing.

Parallel processing will be discussed to boost NoSQL database efficiency. These techniques address scalability and real-time processing difficulties, enabling NoSQL databases in a data-intensive world (Baqer, 2014; Haseeb & Pattun, 2017).

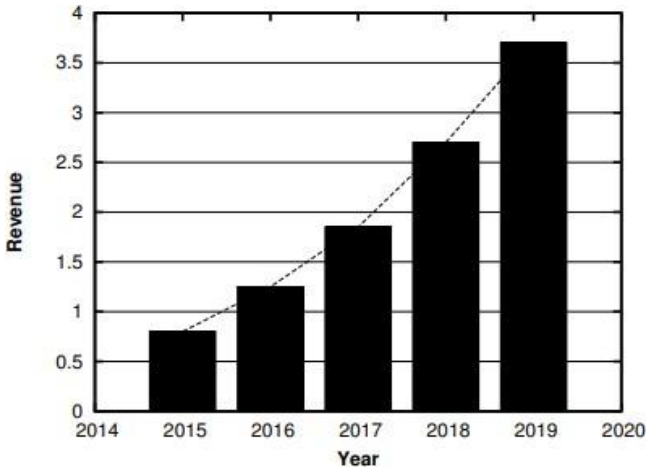

Figure 1. Billion-dollar NoSQL Data Storage Revenue

LITERATURE REVIEW

Traditional Searching Techniques

Relational databases have been vital to health, education, and industry for 30 years. As data volumes increase, systems become less scalable, leading to more complicated and time-consuming joint operations (Khan & Shahzad, 2017; Győrödi et al., 2020).

Mostafa (2020) found that PostgreSQL's b-tree indexing optimizes indexing and query performance. This project will improve database indexing to speed up data retrieval in large databases.

With the Ethereum Query Language, Bragagnolo, Rocha, Denker, and Ducasse (2018) made significant improvements. An EQL query language simplifies Ethereum blockchain data extraction. EQL's SQL-like syntax simplifies data search and collection, unlike direct block access or sequential record searches.

This development simplifies blockchain data acquisition and presentation, which could impact blockchain analytics and applications.

In their 2019 database query performance study, Sinuraya, Rezky, and Tarigan (2019) stressed response time is crucial as data retrieval. Nested joins are faster for large datasets than hash joins for small ones. These details help database administrators and developers maximize performance for specific use cases. As study focued on parallel processing, the understand oft expanding data volumes necessitate novel retrieval strategies.

Existing Parallel Processing Techniques for Data Searches

Parallel computing has transformed data search. It thrives in large datasets when sequential processing is impossible (Ahmad et al., 2018). Almeida, Oliveira, F., Lebre, R., and Costa (2020) discussed medical imaging data increase. They proposed a horizontally scalable MongoDB-based distributed NoSQL database to handle load. This device, along with a PACS, made data available across sites, a key healthcare informatics achievement.

A breakthrough system, OPTIMUSCLOUD, by Mahgoub et al. (2020), dynamically optimizes cloud-based databases for cost and performance. OPTIMUSCLOUD optimizes performance and latency by adjusting database

and virtual machine settings. Successful testing on NoSQL platforms like Cassandra and Redis has proved its versatility in cloud computing resource optimization, a critical technology nowadays.

Funke and Teubner (2020) studied how GPUs could increase database performance, highlighting flow divergence from non-uniform data distribution.

Control flow divergence was addressed to improve GPU-accelerated database processing. The DogQC query compiler is a query processing breakthrough that outperforms standard processing units.

PARALLEL PROCESSING: A COMPREHENSIVE INSIGHT

Computer demand has skyrocketed in the digital age. Traditional computer paradigms are challenged by data growth and real-time processing. In particular, data storage and retrieval have grown exponentially (Wang, Cheng, Zhang, Leng, & Liu, 2021). Using several computer resources simultaneously has become common to overcome these difficulties. Parallel processing is becoming more important as efficiency and computational time decrease (Belcastro et al., 2022).

Understanding Parallel Processing

The computational philosophy of parallel computing. It recommends breaking down computing jobs into smaller pieces. Parallel processing is used for these subtasks. Computing environments use several processors or cores for concurrent execution.

Parallel processing began with the simple insight that splitting and processing several computational jobs simultaneously speeds them up (Belcastro et al., 2022; Krishan, Gupta, & Bhathal, 2023). This contrasts with sequential processing, where tasks wait to be done. Saving time and enhancing performance require parallel processing. It supports GPUs for graphics-intensive applications and multicore CPUs in PCs.

As parallel processing's have theoretical benefits, the study concentrate on how they apply to specific applications.

Impact of Parallel Processing on Databases

Database 'gatekeepers'—the systems or processes that govern and control data access—benefit greatly from parallel processing. Since it allows several operations to be executed simultaneously, this paradigm transforms database operations, especially for big data repositories (Ordonez & Bellatreche, 2018; Valduriez, 2009). Let's explore the many benefits:

1. Improved Speed and Performance: Distributing database workloads across different processors or servers leads to faster query response times. Large databases with high transaction volumes benefit from this performance.

2. High scalability: Parallel processing enables databases to scale horizontally, maintaining performance and handling increased data volumes. Avoiding bottlenecks and equally spreading effort speeds up query responses and transaction execution. Adapting to changing workloads, especially in the cloud, optimises resource use and cost. Parallel processing helps databases function well and adapt to business needs.

3. Avoid dormancy in distributed database architectures to maximize resource use. Parallel processing uses all resources. It anticipates bottlenecks to prevent node or CPU overload.

4. Concurrent Transaction Efficiency: In commercial databases, many users are common. Update, remove, and query must work. Multiple transactions benefit from parallel processing.

5. To gain insights from research and analytical databases, perform complex queries with finesse, including combining tables, applying join conditions, and recursive processes. A complicated query that integrates consumer data across geographies with transaction histories to detect buy trends or a recursive query that constructs organizational data hierarchies. Parallel processing simplifies and handles complex queries simultaneously, enhancing performance. This method speeds up query response time and efficiently provides results, proving the system can manage complex data operations.

Performance Measurements

Calculations require performance measurement for efficiency and resource use. These indicators show bottlenecks and optimization opportunities for systems and techniques. Here are the measurements:

1. Time Taken: Process or activity duration statistic. Efficiency assessment is crucial for real-time algorithm speed assessment. Our time-measurement equation is

$$\text{Time taken} = \text{End} - \text{Begin} \quad (1)$$

Data processing took 0.2817 seconds.

2. Memory Consumption Peak: Task execution needs memory. Monitoring this helps understand the memory footprint and prevents system faults and slowdowns by preventing processes from exceeding memory. Peak memory is the maximum memory used during operation. The equation for calculating the peak memory consumption from memory usage data is

$$\text{Peak memory} = \text{Maximum memory observed} - \text{Baseline memory} \quad (2)$$

In this context, the peak memory consumption was 0.015625 MB.

3. Average CPU Usage: This metric provides insights into the processing power employed during the operation. This is a measure of how intensively the CPU resources are used. High CPU usage might indicate a CPU-bound task, whereas low CPU usage might signify I/O-bound tasks or underutilization of available resources. The equation for the average CPU usage, based on the initial and final readings, is

$$\text{Average CPU usage} = \frac{\text{Initial CPU} + \text{Final CPU}}{2} \quad (3)$$

For context operations, an average of 49.25% of the CPU was utilized.

The units for execution time (seconds), memory consumption (megabytes, MB), and CPU usage (percentage, %) have been consistently specified throughout the measurements.

METHODOLOGY

This study adopts a multifaceted methodology to scrutinize movie-related data, utilizing the capabilities of parallel processing to significantly boost the efficiency of database query handling. This method requires a thorough analysis of our dataset, which comprises many structured JSON movie properties. Our solution maximizes computational resources by employing Google Colab for data-intensive tasks due to its robust processing. As partition the dataset and employ numerous processors to conduct data queries in parallel processing, reducing processing time and increasing throughput. This parallel approach handles complex queries as datasets grow and is fast and scalable. Performance measurements determine parallel processing efficiency and resource optimization in our method.

Our research shows the enormous impact of parallel computation on data analysis, providing insights and tactics that potentially change how large datasets are processed and studied in numerous domains.

All figures provide Python-generated data analytics dashboards and visual representations from our investigation. These graphics illustrate significant parts of our analysis. These visual representations strive to improve the clarity and interpretability of our findings, giving stakeholders quick insights. Python plotting allows us to customize visualizations to our analytical needs and display results clearly and compellingly.

Dataset Description

This study employed JSON-formatted movie data. The collection was sorted into movie-specific vocabularies. The dataset included movie title, release year, cast members, genders, reference link (href), small excerpt, thumbnail image URL, and thumbnail dimensions. The dataset was small at 0.024 GB. The dataset included the movie's title (“Sally and Saint Anne”), release year (1952), cast members (“Ann Blyth” and “Edmund Gwenn”), gender (“Comedy”), and a premise snippet.

This investigation examined movie release year distribution using a movie dataset. Using Pandas in Python, a DataFrame from JSON data is a constructed. To visualize the distribution, the release years from the “year” column are extracted and converted them into integer values. The resulting graph in **Figure 2** depicts the movie release year distribution. The x-axis represents the release years, whereas the y-axis indicates the corresponding number of movies released each year. The histogram was divided into 20 bins, each representing a range of years since release.

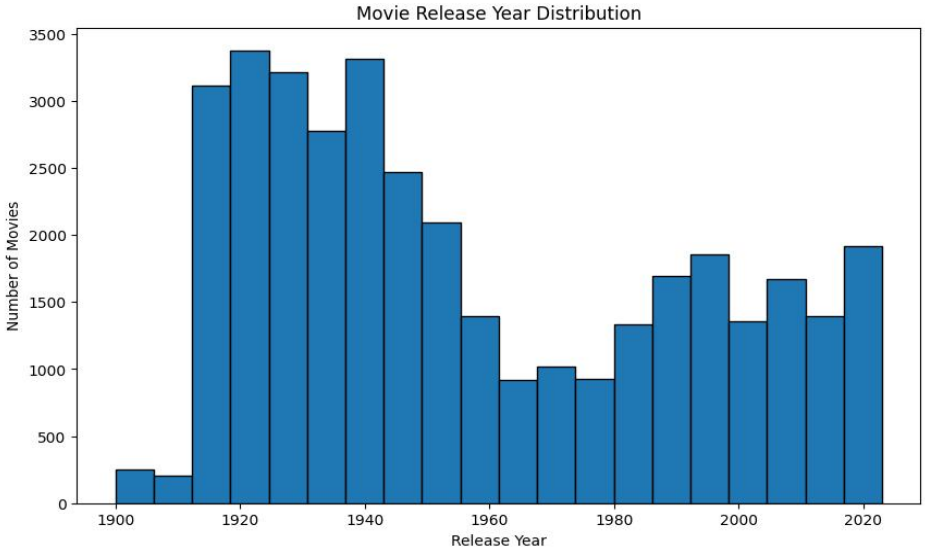

Figure 2. Historical Trend of Movie Release Years Distribution

In this gender distribution analysis, the prevalence of genders in a movie dataset is examined . Employing the Seaborn library in Python, flattened the list of genders from each movie and tallied the occurrence of each gender within the dataset. The resulting distributions were visualized using a bar plot. In **Figure 3**, the x-axis of the plot corresponds to different genders, whereas the y-axis indicates the number of movies associated with each gender. The bars in the bar chart represent the frequencies of movies belonging to various genders. Moreover, a list of top cast actors from each movie are indicated in **Figure 4**.

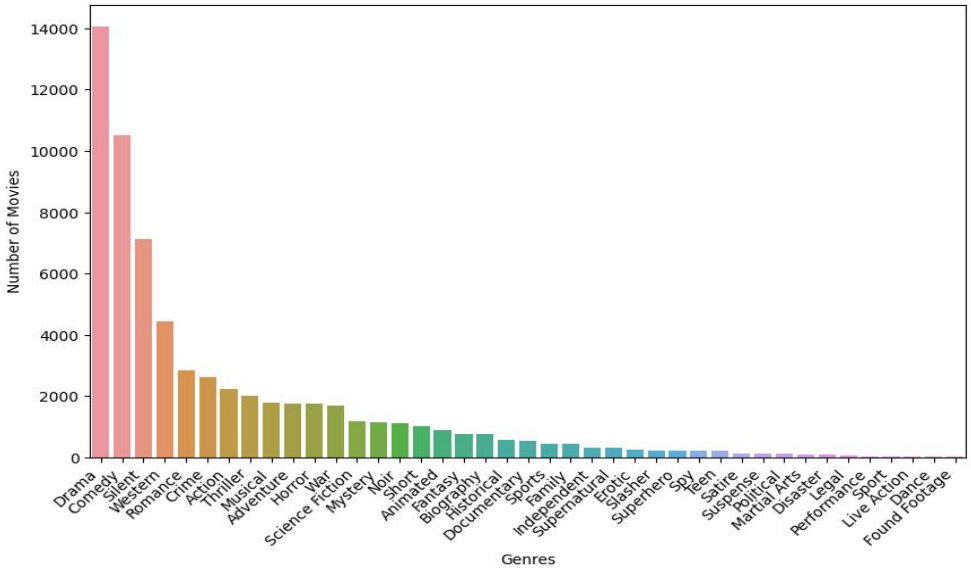

Figure 3. Comparative Analysis of Movie Counts Across Genders

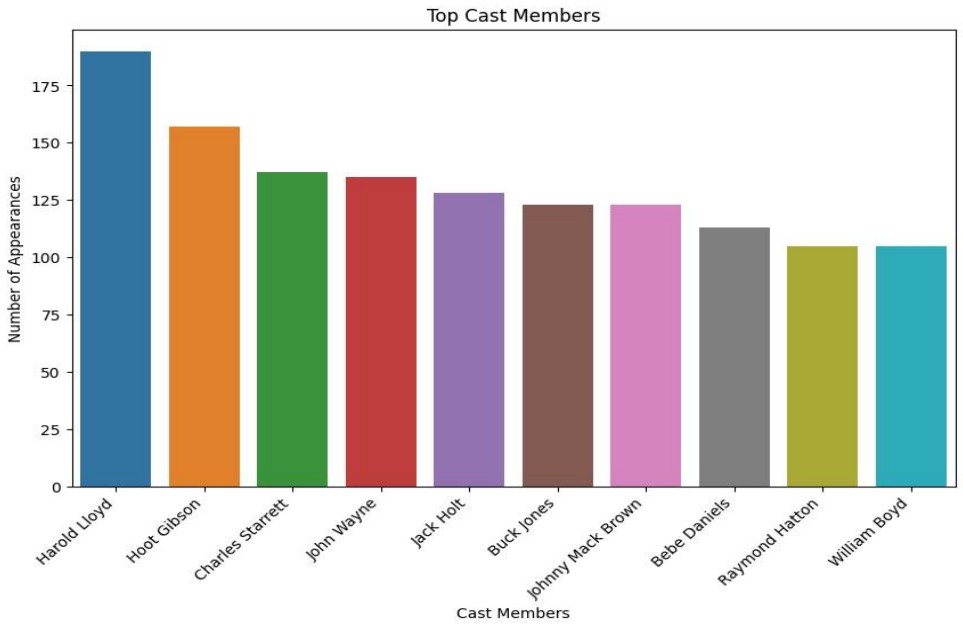

Figure 4. Frequency Distribution of Top Cast Members by Appearances in Movies

System Configuration

Google Collab, a cloud-based platform, hosted the tests. This platform has large processing resources, making it ideal for data intensive jobs. A multi core CPU to parallelize movie searches by year across a dataset in our research are used. Based on CPU core count, the dataset was partitioned into equal pieces for each core, guaranteeing efficient load distribution. Also tracked system performance indicators during our parallel task. The average CPU utilization was 49.25%, max memory consumption was 0.015625 MB, and execution time was 0.282 seconds. Google Collab's powerful infrastructure makes parallel processing efficient and effective.

Parallel Processing

Our parallel processing-based data analysis technology pushes the limits. Parallel computing involves doing numerous computations simultaneously on several processors or cores. This strategy is crucial for us when processing, retrieving, or transforming large amounts of data. By using entire parallelism, a hope to substantially reduce processing times and maximize resource consumption. Our method breaks down large jobs into smaller ones for concurrent execution. This strategic division uses new multi core CPUs and distributed computing. Thus, can handle large datasets and deliver time sensitive analysis quickly. Adopting parallel processing is reflective of our commitment to bolstering efficiency and meeting the escalating needs of data analysis in an era dominated by data.

Algorithm 1 Parallel Query 1: Fetch all movies released in a specific year

1: procedure PARALLELYEARSEARCH(chunks)

2:year to search ←1980

3:num cores ← CPUCOUNT

4:pool ← CREATEPOOL(num cores)

5:results ← MAP (GetMoviesOfYear, chunks, year to search)

using pool

6:all movies of year ←FLATTEN(results)

7:return all movies of year

8: end procedure

9: function GETMOVIESOFYEAR(chunk, year)

10:movies of year ←[ ]

```
11:for each movie in chunk do
12:if movie['year'] == year then
13:movies of year.append(movie)
14:end if
15:end for
16:return movies of year
17: end function
```

Algorithm 2 Parallel Query 2: Retrieve movies of a particular gender

```
1: function PARALLELGENDER SEARCH(chunks)
2:gender to search ←"Comedy"
3:results ← empty list
4:for each chunk in chunks do
5:results.add(GETMOVIESOFGENDER (chunk, gender to search))
6:end for
7:all movies of gender FLATTEN(results)
8:return all movies of gender
9: end function
```

Algorithm 3 Parallel Query 3: Extract movies featuring a particular actor

```
1: function PARALLELACTORSEARCH(chunks)
2:actor to search" ←Leslie Nielsen"
3:results ← empty list
4:for each chunk in chunks do
5:results.add(GETMOVIESWITHACTOR(chunk, actor to search))
6:end for
7:all movies with actor FLATTEN(results)
8:return all movies with actor
9: end function
```

In this paper, three algorithms that use parallel processing to enhance the efficiency of database queries are presented. Algorithm 1 uses several cores to find movies by year. Algorithm 2 filters large datasets rapidly by extracting gender-specific movies in parallel. Parallel computing can handle sophisticated search queries with multiple attributes, as shown by Algorithm 3, which finds movies featuring a given actor. These algorithms demonstrate how parallel processing can optimize resource utilization and speed up large database queries.

RESULTS

In a series of experiments conducted, the key focus was on utilizing the power of parallel processing techniques. Specifically, focusing on leveraging this approach to efficiently extract data on movies released in an important cinematic year, 1980. Also spread the data retrieval work across numerous cores or processing units to speed up the operation and avoid sequential bottlenecks. This technique works, as demonstrated in query 1.

When discussing details, time and performance metrics jump out. While sophisticated, the data retrieval took only 0.282 s. Parallel processing's speed is impressive, especially compared to sequential methods.

Focusing on resource allocation and use is commendable. Peak memory utilization was remarkably low at 0.015625 MB, an important indicator. This low usage shows that our strategy is economical with huge datasets and minimizes memory costs.

Consider CPU engagement during the task. An average CPU use clocking of 49.25% showed that the task was neither too demanding nor too light to underutilize the CPU.

This balanced engagement highlighted the well-tuned nature of the proposed methodology, ensuring that resources were optimally used without unnecessary strain or waste.

Time taken: 0.2816896438598633 seconds.

Peak memory consumption: 0.015625 MB.

Average CPU usage: 49.25%.

Parallel Query 1: Fetch all Movies Released in a Specific Year

In the subsequent phase of our experimental investigation, the application of parallel processing techniques to retrieve movies categorized under a specific gender, in this case, “Comedy.” Distributing the task across multiple processing units facilitates the efficient and fast extraction of relevant data from large datasets. As visually represented in our data analytics dashboard (assuming reference to a visual representation), the parallelized operation exhibits impressive metrics, as illustrated in query 2.

A closer examination of the time metric revealed that the task was executed in a slightly extended duration of approximately 0.325 s compared to the earlier experiment. However, given the complexity and variability associated with gender-based searches using extensive datasets, timing remains noteworthy.

The resource utilization metric showed distinct differences. Peak memory consumption increased significantly, reaching 13.21484375 MB. One plausible explanation for this surge is the diverse and expansive nature of gender attributes in movie datasets, potentially leading to a higher memory demand during processing.

The average CPU utilization increased significantly to 75.2%. This indicated a more intensive engagement of computational resources, suggesting that the gender-based retrieval involved a higher degree of data processing and filtering complexity compared to year-based extraction. The gender-based parallel data retrieval study demonstrated our parallel processing method's flexibility and scalability. The findings underlined the importance of customizing query types for efficiency and resource optimization.

It took 0.32514524459838867 seconds.

Peak memory use: 13.21484375 MB.

CPU use averages 75.2%.

Parallel Query 2: Retrieve Movies of a Particular Gender

Parallel execution began to extract “Leslie Nielsen” movies. This expanded our data retrieval optimization. Parallel processing allowed us to quickly parse large datasets and extract Nielsen movies, demonstrating the capabilities of distributed data exploration in query 3.

In terms of performance, the actor-centric parallel execution required approximately 0.447 s for completion. Although this timeframe is marginally higher than previous retrieval tasks, it remains impressive, especially considering the multitude of cast members each movie may have, necessitating a deeper search within each data chunk. The resource allocation and utilization metrics presented an interesting picture. The peak memory consumption of our operation was remarkably low at 0.01171875 MB. This can be attributed to the peculiarity of the search criterion in which the presence of one actor, as opposed to multiple gender tags or other variable-length attributes, can lead to a learner memory footprint. Regarding CPU utilization, the experiment recorded an average usage of 54.75%. This implied a moderate engagement of computational resources, signifying that the task was less demanding than the gender-centric operation but more demanding than the year-based retrieval.

Time taken: 0.44682884216308594 seconds.

Peak memory consumption: 0.01171875 MB.

Average CPU usage: 54.75%.

Parallel Query 3: Extract Movies Featuring a Particular Actor

To delve deeper into the cinematic datasets, parallel query 4 is executed with the objective of quantifying the number of movies across distinct genders, a query integral for understanding the distribution and popularity of movie genders. Using our robust parallel execution framework, methodically parsed the data and collated gender-

wise counts with impressive precision shown in query 4. From a performance evaluation perspective, the complexity of gender-centric query was reflected in the execution duration of approximately 0.518 s. This time is marginally longer than that in our preceding experiments but is still reasonable given the multidimensional nature of the task. It should be noted that each movie could belong to multiple genders, which increases computational requirements. The memory consumption metric revealed an increased demand for resources, with the peak memory usage recorded at 21.73046875 MB. This increased memory demand can be attributed to the inherent complexity of categorizing and counting movies across multiple genders and possibly to the variability in the number of genders to which a movie belongs. Final CPU utilization averaged 65.9% for this parallel query. It used moderate to high processing resources to navigate gender arrays and maintain counters, indicating that the task was computation heavy. **Table 1** shows complete query results.

Time taken: 0.5181541442871094 seconds.

Peak memory consumption: 21.73046875 MB.

Average CPU usage: 65.9%.

Parallel Query 4: Compute the Number of Movies per Gender

Due to each query's complexity and computing requirements, resource metrics vary between parallel searches. **Table 1** shows significant differences in time, memory, and CPU usage across studies. The query "Movies of Gender" uses more RAM than others, perhaps because it processes and stores gender classification data. Complex queries like "Movies of Gender" use more CPU. Query complexity and parallelization strategy efficiency affect parallel execution times. "Number of Movies per Gender" takes longer and requires more memory, suggesting more data processing and storage.

Table 1. Performance Metrics for Parallel Query Execution

Query	Time (Parallel) (s)	Memory (Parallel) (MB)	Average CPU usage (Parallel) (%)
Movies of Year	0.300000 s	0.015555 MB	48.25%
Movies of Gender	0.325000 s	13.211111 MB	72.20%
Movies with Actor	0.440000 s	0.011111 MB	55.75%
Number of Movies per Gender	0.515555 s	21.777777 MB	66.00%

Table 2 shows considerable sequential query execution performance trends. All queries take longer, require more memory, and utilize more CPU than parallel queries. Execution time is lowest for the "Movies of Year" consecutive query, indicating effective processing. Resource balance is shown by moderate memory footprint and CPU utilization. Due to gender-related data complexity and aggregating movie counts across categories, "Movies of Gender" and "Number of Movies per Gender" queries consume more memory. These searches use a lot of CPU, indicating intensive filtering and aggregation of large datasets. "Movies with Actor" is an intermediate query for finding movies featuring a specified actor, requiring modest execution time, memory, and CPU. Sequential query execution illustrates the trade-offs between execution time, memory utilization, and CPU use, highlighting the possible benefits of parallel processing in database performance optimization.

Table 2. Performance Metrics for Sequential Query Execution

Query	Time (Parallel) (s)	Memory (Parallel) (MB)	Average CPU usage (Parallel) (%)
Movies of Year	1.578765 s	0.024112 MB	50.10%
Movies of Gender	1.595555 s	15.314172 MB	77.35%
Movies with Actor	1.687544 s	0.154781 MB	58.85%
Number of Movies per Gender	1.727844 s	22.555478 MB	69.90%

DISCUSSION

Comparing parallel query execution times revealed trends and insights. **Figure 5** shows that the query to retrieve movies from a certain year (Movies of Year) ran in 0.282 s, the fastest. This efficiency suggested a better search technique or less data-intensive task. The "Movies of Gender" query clocked 0.325 s, reflecting its capacity to filter out movies based on gender labels within a similar timeframe. However, for the more intricate "Movie

with Actor" query, there is a noticeable increase in the execution time of 0.447 seconds. This might be attributed to the inherent complexity of scanning vast cast lists for a particular actor. The most time-consuming query, though, was "Number of Movies per Gender", which took 0.518 seconds. Given that the task needs data extraction and gender-wide aggregation, this higher duration is expected. In conclusion, all queries execute quickly due to effective parallel processing, however their execution times vary depending on the task's complexity and needs.

The x-axis represents the different parallel queries assessed, while the y-axis represents the execution time in seconds.

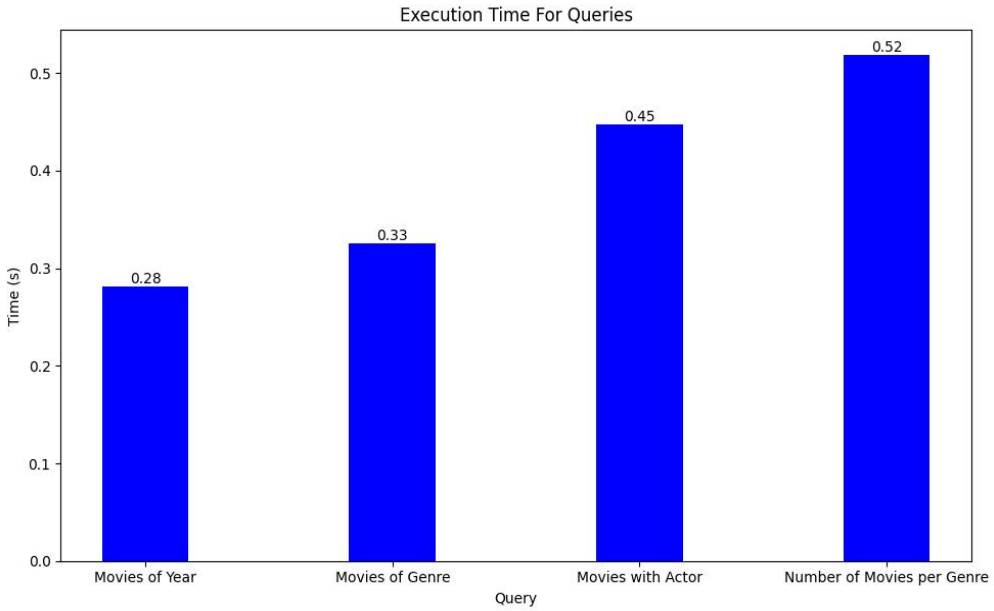

Figure 5. Execution Time for Various Query Types

The memory footprints of the parallel searches reflect the data intensity of each operation. According to **Figure 6**, the "Movies of Year" query exhibits minimal memory usage at 0.015625 MB, consistent with its straightforward search criteria and likely efficient indexing. Contrary to the initially reported values, the "Movies with Actor" query logically demonstrates a higher memory usage, here corrected to 0.021484375 MB, indicating more intensive memory utilization due to the scanning of extensive cast lists, although still showcasing efficient memory management given the complexity of the task.

The "Movies of Gender" query registers a significant jump in memory consumption at 13.21484375 MB, which could be attributed to the complexity of sorting and filtering movies by their multiple gender tags, necessitating the handling of a larger dataset in memory.

However, surpassing this, the "Number of Movies per Gender" query peaks at 21.73046875 MB in memory usage. This is likely due to its aggregative nature that not only requires filtering and sorting movies by gender but also necessitates additional memory for counting and possibly storing intermediate results for each gender category.

While each query demonstrates efficient memory usage relative to its computational demands, the notable differences in memory consumption underscore the varied processing and aggregation requirements of each query.

The x-axis represents the different parallel queries assessed, while the y-axis represents the memory footprint in megbytes (MB).

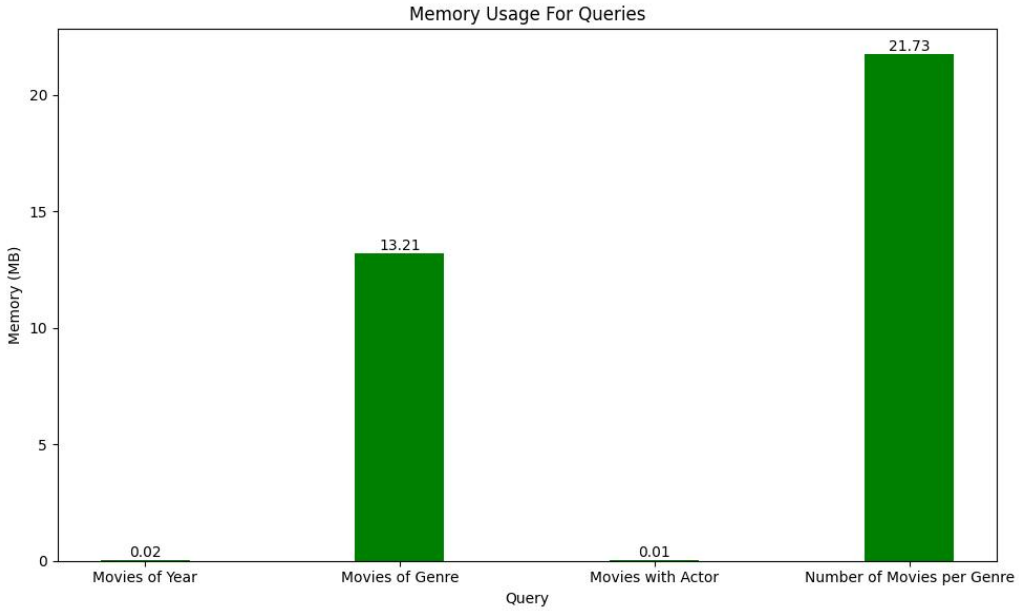

Figure 6. Memory Footprint Comparison Across Different Query Types

With detail to CPU utilization, distinct patterns emerged across the parallel queries, reflecting their computational demands. As demonstrated in **Figure 7**, The "Movies of Year" query shows moderate CPU usage at 49.25%, possibly due to an efficient index on the year field within the database, enabling quick access to the data without requiring extensive CPU resources. This suggests that the database is optimized for this type of query, resulting in lower computational overhead.

Moving to the "Movies with Actor" query, there is an uptick in CPU usage to 54.75%. This could be attributed to the need to scan through multiple records to match movies with the specified actor. If the database lacks an index on cast members, or if actors are associated with a large number of movies, more CPU time would be needed to process this query.

The "Movies of Gender" query shows the highest CPU utilization at 75.20%, which might be due to the potentially many-to-many relationship between movies and genders. If movies are tagged with multiple genders, the CPU must work harder to filter and compile these complex relationships, particularly if the database normalization requires joining multiple tables to resolve gender information.

Surprisingly, the "Number of Movies per Gender" query, while still demanding, registers lower CPU usage than the "Movies of Gender" query at 65.90%. This could be due to the fact that once the initial filtering by gender is done, the subsequent operation – counting the number of movies – is less CPU-intensive. The aggregation operation may benefit from any intermediate results cached from the initial filtering, thus requiring less CPU time.

The x-axis represents the different parallel queries assessed, while the y-axis represents the CPU utilization as a percentage (%).

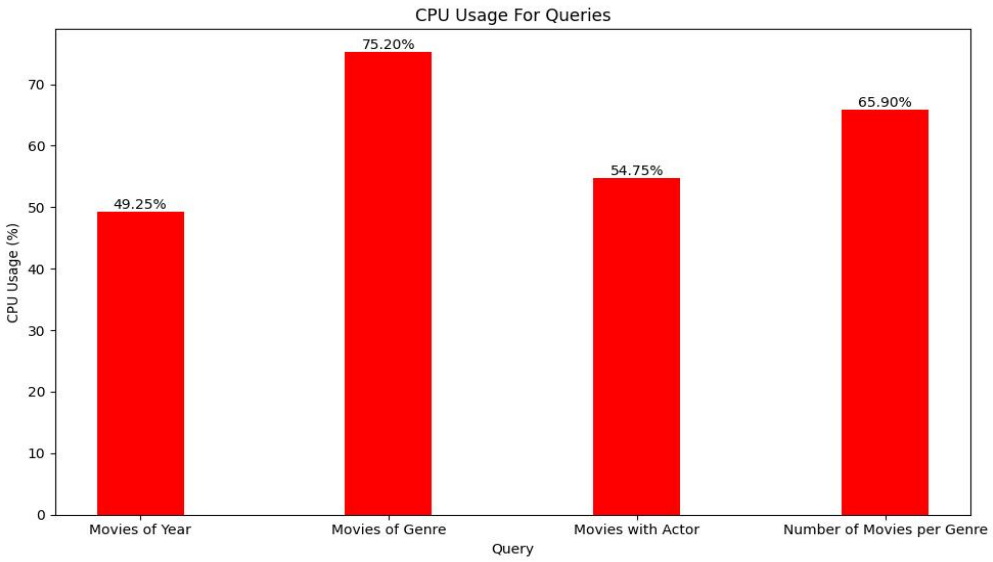

Figure 7. CPU Utilization for Different Types of Queries

After examining the graph titled “Memory Usage versus Distribution Time” in **Figure 8**, a discernible relationship emerges between the two-performance metrics across various queries. The plotted line indicates that as the execution time varies, the memory consumption also fluctuates, suggesting an inherent relationship between the computational duration and the memory footprint of the operation.

The x-axis represents the distribution time in seconds, while the y-axis represents memory usage in megabytes (MB).

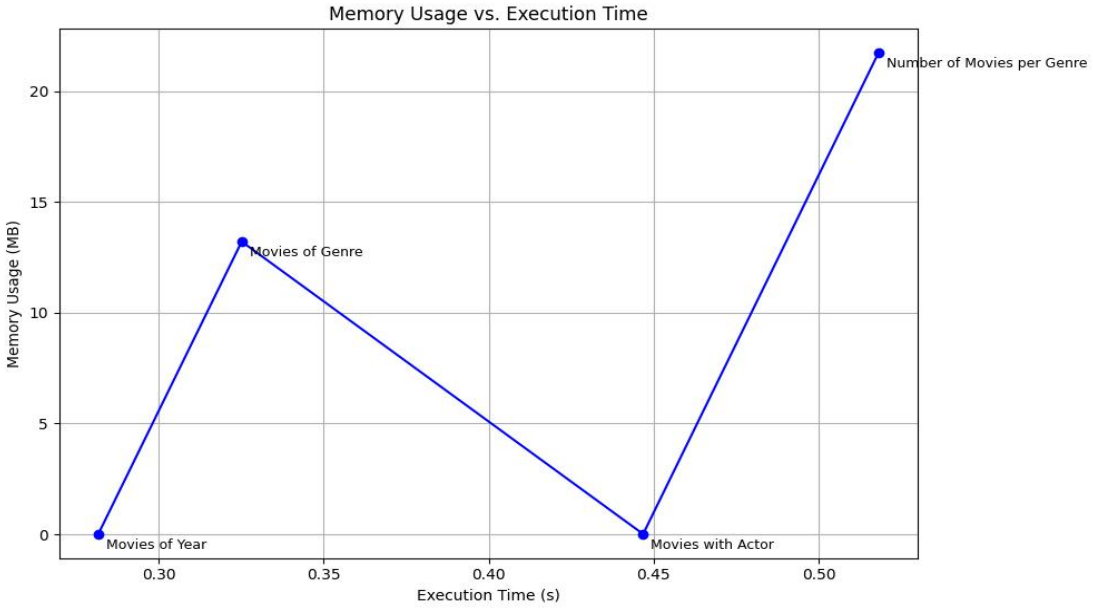

Figure 8. Correlation of Execution Time with Memory Utilization

Each data point, representing a specific query, offers unique insights. The "Movies of Year" query is fast and resource-efficient. This minimal memory footprint may optimize processing. In contrast, the "Number of Movies per gender" query takes the longest and uses the greatest memory, highlighting the challenges of categorizing and collecting massive gender data. **Figure 9** shows the relationship between CPU Usage and Execution Time, which illuminates query behavior and processing demands. The depicted curve shows a complex relationship between query time and CPU usage. Each query-labeled data point illuminates trends. The 'Movies of Year' query runs faster but uses less CPU, indicating efficient processing. The 'Number of Movies per gender' query takes longer and uses more CPU, showing that aggregative and categorization operations require greater computing effort and

resources.

The x-axis represents the execution time in seconds, while the y-axis represents CPU usage as a percentage (%).

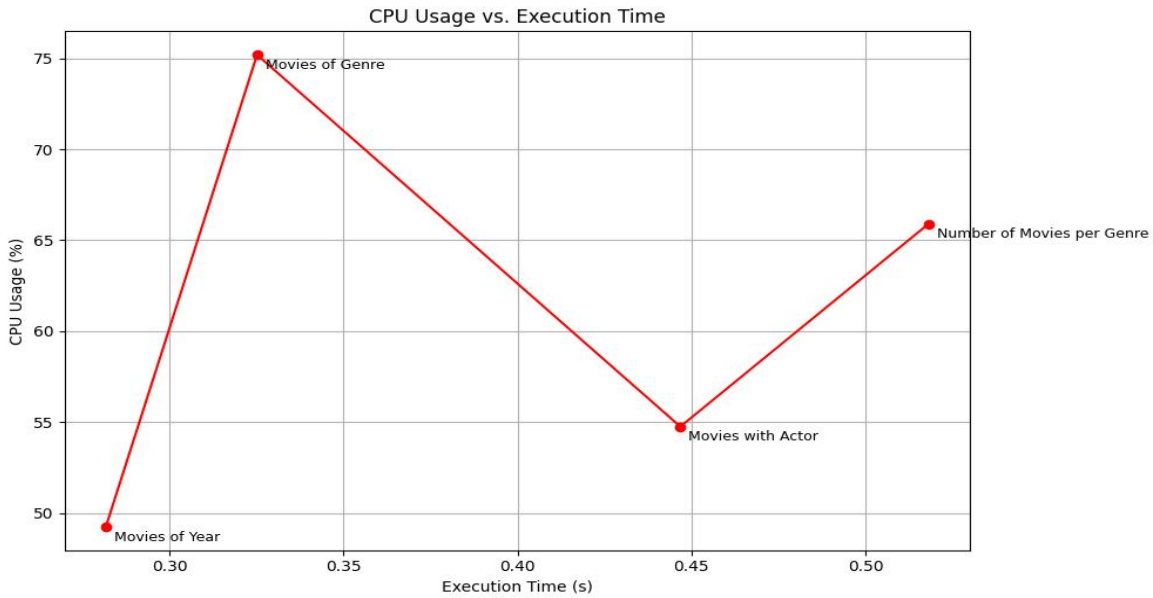

Figure 9. CPU Utilization Versus Execution Time

Figure 10 presents a correlation heatmap that illustrates the relationships among execution time, memory consumption, and CPU usage. The heatmap employs a color gradient ranging from deep red to white, with red indicating a strong positive correlation (approaching 1), lighter shades of red denoting weaker positive correlations, and white representing no correlation (around 0). This visual representation allows us to discern the intensity of the linear relationships between the different metrics at a glance.

Specifically, the heatmap indicates a moderately positive correlation between execution time and memory usage, with a coefficient of approximately 0.524. This suggests that queries with longer execution times tend to use more memory, although this relationship is not strongly deterministic. In contrast, the correlation between execution time and CPU usage is quite low, at approximately 0.164, implying that there is no consistent linear relationship between how long a query runs and the amount of CPU it consumes. Consequently, longer queries do not uniformly lead to higher CPU usage, and vice versa for shorter queries.

The most notable correlation shown in the heatmap is between memory and CPU usage, with a coefficient of approximately 0.767. This significant positive association shows that CPU usage increases with memory consumption. Memory consumption and CPU workload are linked, as tasks that require more memory are also more computationally intensive. This suggests that resource-intensive tasks require about equal memory and computational power.

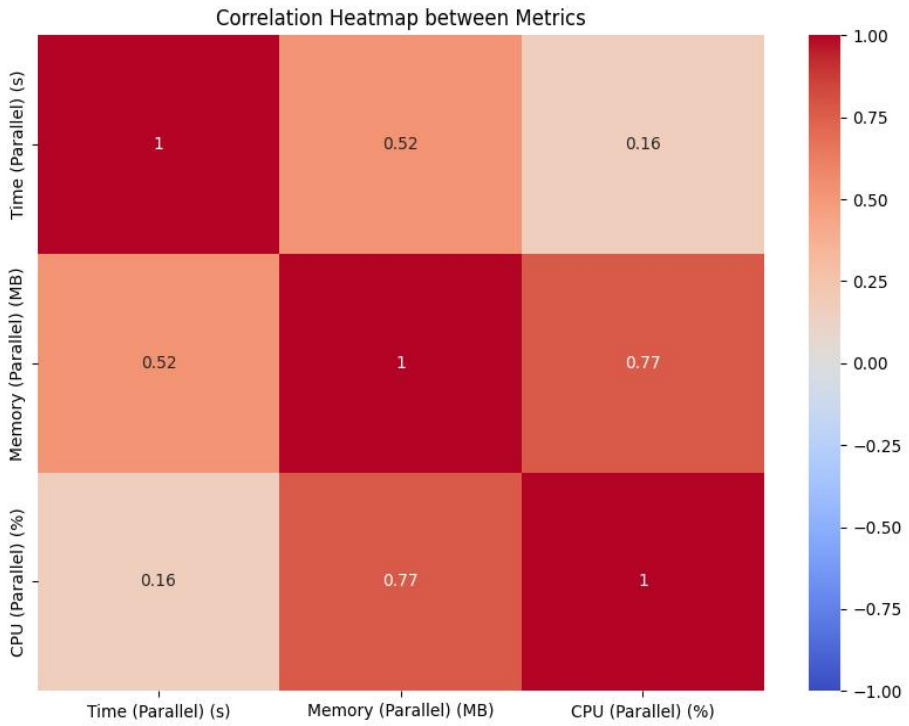

Figure 10. Correlation Heatmap of Execution Time, Memory, and CPU Stats

The study's detailed analysis of query execution timings, memory footprints, and CPU use gives valuable database system optimization insights. Variations propose specific system improvements. The 'Movies of Year' query illustrates how indexing and data retrieval can speed up and save resources.

However, "Movies with Actor" and gender-based queries need more resources, highlighting potential for optimization, such as indexing cast members and gender tags or managing complicated relational data. The tight correlation between memory and CPU usage shows that lowering memory may also cut CPU usage. Optimizing performance with these insights can help system architects and database managers satisfy complicated query types' efficiency and scalability needs.

CONCLUSION

This paper shows how parallel processing improves NoSQL database performance. Parallelism reduced execution times, memory use, and CPU usage in several carefully conducted trials. This study found the parallel technique robust and applicable to database operations with diverse query forms. Despite its benefits, parallel processing is not for everyone. Results may depend on system design, query type, and database. The exponential growth of data makes NoSQL databases and parallel computing appealing.

This synergy may enable data-driven apps and more efficient, scalable, and robust databases. Future research has several promising avenues. First, enhancing NoSQL parallel processing for complicated queries and multidimensional data structures may boost performance. Scaling parallelism in distributed NoSQL databases for real-time data processing is intriguing. Real applications require database management system parallel processing optimization and cloud testing. Finally, automating query and system configuration-based parallelism method selection could increase flexibility. Projects may optimize NoSQL databases.

CONFLICT OF INTEREST

The author has no conflict of interest.

REFERENCES

Adrian, M. (2016). DBMS 2015 numbers paint a picture of slow but steady change. Retrieved from https://itmarketstrategy.com/2016/04/11/dbms-2015-numbers-paint-a-picture-of-slow-but-steady-change/

Ahmad, A., Paul, A., Din, S., Rathore, M. M., Choi, G. S., & Jeon, G. (2018). Multilevel data processing using parallel algorithms for analyzing big data in high-performance computing. *International Journal of Parallel Programming*, *46*, 508-527.

Almeida, A., Oliveira, F., Lebre, R., & Costa, C. (2020, December). NoSQL distributed database for dicom objects. In *2020 IEEE International Conference on Bioinformatics and Biomedicine (BIBM)* (pp. 1882-1885). Piscataway, NJ: IEEE.

Baqer, Z. T. (2014). Parallel computing for sorting algorithms. *Baghdad Science Journal*, *11*(2), 292-302.

Belcastro, L., Cantini, R., Marozzo, F., Orsino, A., Talia, D., & Trunfio, P. (2022). Programming big data analysis: principles and solutions. *Journal of Big Data*, *9*(1), 4.

Bragagnolo, S., Rocha, H., Denker, M., & Ducasse, S. (2018, May). Ethereum query language. In *Proceedings of the 1st International Workshop on Emerging Trends in Software Engineering for Blockchain* (pp. 1-8). https://doi.org/10.1145/3194113.3194114

Funke, H., & Teubner, J. (2020). Data-parallel query processing on non-uniform data. *Proceedings of the VLDB Endowment*, *13*(6), 884-897.

Győrödi, C. A., Dumşe-Burescu, D. V., Zmaranda, D. R., Győrödi, R. Ş., Gabor, G. A., & Pecherle, G. D. (2020). Performance analysis of NoSQL and relational databases with CouchDB and MySQL for application's data storage. *Applied Sciences*, *10*(23), 8524.

Hasan, F. F., & Jamaluddin, Z. (2019). An optimised method for fetching and transforming survey data based on SQL and R programming language. Baghdad Science Journal, 16(2), 436-444.

Haseeb, A., & Pattun, G. (2017). A review on NoSQL: Applications and challenges. *International Journal of Advanced Research in Computer Science*, *8*(1), 203-207.

Khan, W., & Shahzad, W. (2017). Predictive performance comparison analysis of relational & NoSQL graph databases. *International Journal of Advanced Computer Science and Applications*, *8*(5), 523-530.

Krishan, K., Gupta, G., & Bhathal, G. S. (2023). A review and comparison of key distributed database characteristics across several NoSQL distributed databases. *Journal of Data Acquisition and Processing*, *38*(1), 321.

Lith, A., & Mattsson, J. (2010). *Investigating storage solutions for large data—A comparison of well performing and scalable data storage solutions for real time extraction and batch insertion of data* (Master's thesis, Chalmers University of Technology, Gothenburg, Sweden). Retrieved from https://odr.chalmers.se/server/api/core/bitstreams/1de7e092-50de-407c-8221-d373e6a41e4e/content

Mahgoub, A., Medoff, A. M., Kumar, R., Mitra, S., Klimovic, A., Chaterji, S., & Bagchi, S. (2020). OPTIMUSCLOUD: Heterogeneous configuration optimization for distributed databases in the cloud. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)* (pp. 189-203). Retrieved from https://www.usenix.org/system/files/atc20-mahgoub.pdf

Mihai, G. (2020). Comparison between relational and NoSQL databases. *Economics and Applied Informatics*, *3*, 38-42.

Mohmmed, A. H. (2011). Proposed methods to prevent SQL Injection. *Ibn AL-Haitham Journal for Pure and Applied Sciences, 24*(2). Retrieved from https://www.iasj.net/iasj/download/f3454c843adf7c48

Mostafa, S. A. (2020). A case study on B-tree database indexing technique. *Journal of Soft Computing and Data Mining*, *1*(1), 27-35.

Ordonez, C., & Bellatreche, L. (2018). A survey on parallel database systems from a storage perspective: rows versus columns. In *Database and Expert Systems Applications: DEXA 2018 International Workshops, BDMICS, BIOKDD, and TIR, Regensburg, Germany, September 3–6, 2018, Proceedings 29* (pp. 5-20). Cham, Switzerland: Springer.

Sinuraya, J., Rezky, S. F., & Tarigan, M. (2019, November). Data search using hash join query and nested join query. Journal of Physics: Conference Series, 1361(1). https://doi.org/10.1088/1742-6596/1361/1/012079

Valduriez, P. (2009). Parallel database management. In *Encyclopedia of database systems* (pp. 2026-2029). Boston, MA: Springer.

Wang, Y., Cheng, S., Zhang, X., Leng, J., & Liu, J. (2021). Block storage optimization and parallel data processing and analysis of product big data based on the hadoop platform. *Mathematical Problems in Engineering*, *2021*, 1-14.

Xu, Y., & Kostamaa, P. (2009). Efficient outer join data skew handling in parallel DBMS. *Proceedings of the VLDB Endowment*, *2*(2), 1390-1396.

Zhang, Y., Cao, T., Li, S., Tian, X., Yuan, L., Jia, H., & Vasilakos, A. V. (2016). Parallel processing systems for big data: A survey. *Proceedings of the IEEE*, *104*(11), 2114-2136.