UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

OPTIMIZING A REAL-LIFE DATABASE BASED ON WORKLOAD EVOLUTION: A COMPARATIVE ANALYSIS OF SQL AND NOSQL APPROACHES

GABRIEL ZARATE CALDERON

Thesis supervisor
ALBERTO ABELLO GAMAZO (Department of Service and Information System Engineering)

Thesis co-supervisor
ENRICO GALLINUCCI

Degree
Master's Degree in Data Science

Master's thesis

Facultat d'Informàtica de Barcelona (FIB)

Universitat Politècnica de Catalunya (UPC) - BarcelonaTech

26/06/2024

Optimizing a Real-Life Database Based on Workload Evolution: A Comparative Analysis of SQL and NoSQL Approaches

Gabriel Helard Zarate Calderon

Facultat de Informàtica de Barcelona
Universitat Politècnica de Catalunya

A thesis submitted for the degree of

Master in Data Science

Barcelona, June 2024

This thesis is submitted to the Facultat de Informàtica de Barcelona, Universitat Politècnica de Catalunya in fulfilment of the requirements for the degree in Master in Data Science.

Gabriel Helard Zarate Calderon, June 2024

Acknowledgements

I extend my deepest gratitude to my director and co-director for their invaluable support throughout the duration of this project. Their readiness to assist at any time, coupled with their consistently positive attitude, has greatly encouraged me to deliver my best work. I am immensely grateful for the opportunity to develop this project under their guidance.

I would also like to thank my parents and my sister for their extensive and continuous support during this significant phase of my life. Their unwavering encouragement and the opportunities they have provided have been invaluable. I am sincerely thankful for their love and belief in my abilities.

Abstract

This study performs a dynamic cost-benefit analysis of migrating the Sloan Digital Sky Survey (SDSS) SkyServer database to optimized SQL and NoSQL schemas based on its workload evolution.

Utilizing data spanning over two decades, the research explores the adaptation of database schemas at three pivotal historical points—2003, 2013, and 2023. Each phase is analyzed and optimized in both SQL and NoSQL frameworks to evaluate and compare the performance benefits. The project highlights the necessity for continual schema adjustments driven by evolving workloads and assesses the comparative effectiveness of SQL versus NoSQL solutions in optimizing database operations.

Keywords

Dynamic Cost-Benefit Analysis, Database Migration, SQL Schemas, NoSQL Schemas, Schema Optimization, Workload Evolution, Sloan Digital Sky Survey (SDSS) SkyServer, Database Performance, Schema Adaptation, Performance Benchmarking, Historical Data Analysis, Indexing, Vertical Partitioning, Embedding, PostgreSQL, Document Type Storage.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The increasing volume and complexity of data in modern databases necessitate effective optimization strategies to improve performance and manageability. This thesis addresses the challenge of optimizing the SDSS SkyServer, a critical astronomical database, by adapting its structure in response to changes in the workload over time. The objective is to analyze and compare the impacts of SQL and NoSQL optimization techniques across different phases of the database's lifecycle.

Given the extensive history and significant size of the SDSS SkyServer, which contains decades of detailed astronomical data, this study examines three distinct historical snapshots of the database's workload. These snapshots correspond to years significant for both data growth and technological advancements: 2003, 2013, and 2023. For each period, the workload characteristics are analyzed, and the database is replicated and optimized using both traditional SQL and modern NoSQL approaches. The methods involve extracting and analyzing query features, simulating optimizations, and evaluating performance outcomes.

This comparative analysis seeks to determine the most effective strategies for each period and to draw broader conclusions about the advantages and limitations of SQL and NoSQL technologies in evolving database ecosystems. The findings aim to provide insights applicable to the SDSS SkyServer and other large-scale databases facing continuous data growth and changing user demands.

1.1 Motivation

The ongoing evolution of database schemas and workloads needs a comprehensive analysis of different database management strategies, particularly the contrasting approaches of SQL and NoSQL systems. This thesis explores how these two distinct database technologies manage the dynamic requirements posed by evolving data structures and increasing data volumes. It aims to compare the advantages and limitations inherent to each approach to provide a clear understanding of their respective efficiencies and drawbacks.

Focusing on the SDSS SkyServer as a case study, this research tracks and analyzes the server's schema evolutions and workload patterns over significant time intervals—specifically 2003, 2013, and 2023. By replicating the database conditions and applying both SQL and NoSQL optimization strategies, the study evaluates their effectiveness over different stages of the database lifecycle. This systematic approach not only uncovers the most effective practices for database optimization but also contributes deeper insights into the adaptability and scalability of SQL and NoSQL technologies [Stonebraker 2010, Vogels 2009].

1.1.1 Workload Evolution

Database workloads evolve over time due to changing user requirements, data growth, and technological advancements. Accurately predicting and adapting to these changes is crucial for maintaining performance and efficiency. Previous research has focused on understanding and predicting workload patterns, particularly in NoSQL databases where query languages and data models can be more complex than traditional SQL [Bonnet et al. 2001, Sakr et al. 2013]. Effective workload prediction involves analyzing current query patterns and forecasting future trends to optimize schemas accordingly [Bonnet et al. 2001]. This thesis examines the workload evolution of the SDSS SkyServer across 2003, 2013, and 2023 by analyzing historical query data and simulating future workloads. The insights from this analysis will inform adaptive optimization strategies that can dynamically respond to workload changes, ensuring sustained performance over time [Stonebraker et al. 2007, Vogels 2009, Lih 2009].

1.1.2 Schema Evolution

Schema evolution is crucial in database management, especially in dynamic environments where data requirements frequently change. Research shows that schemas must evolve to accommodate new information, remove obsolete data, and adapt to changes in data types. This evolution is documented in both relational and NoSQL databases, highlighting the patterns and challenges associated with schema updates [Liu et al. 2011, Thor et al. 2017, Curino et al. 2013]. Relational databases often undergo rapid schema expansion early in an application's lifespan to incorporate new data elements [Curino et al. 2013]. NoSQL databases, with their schemaless nature, offer greater flexibility, making them suitable for agile development environments [Thorat et al. 2018]. Despite this flexibility, managing schema evolution remains complex, often requiring automated tools and methodologies. This thesis explores these challenges within the SDSS SkyServer context, aiming to develop strategies ensuring seamless schema evolution while maintaining optimal performance [Stonebraker et al. 2007, Vogels 2009].

1.2 Research Questions and Hypotheses

This study addresses the following research questions:

1. Does optimizing a database at a specific point based on that point in time's workload guarantee optimal long-term performance?

2. Can a NoSQL optimization approach achieve as good performance as an SQL approach in a query-based workload environment?

3. Considering the evolution of schemas and workloads, which optimization approach—SQL or NoSQL—is more effective over time?

The hypotheses corresponding to these questions are:

1. No, optimizing at a single point does not guarantee long-term performance, as the optimal schema configuration heavily depends on how the workload evolves.

2. Yes, a NoSQL approach can achieve comparable performance to an SQL approach, although it may require additional optimization techniques, given that SQL typically outperforms NoSQL in structured query tasks.

3. SQL optimization approaches are expected to perform better over time due to their structured nature and efficiency in handling complex queries, despite the evolving requirements of database systems.

Chapter 2

State of the Art

2.1 Theoretical Background

2.1.1 SQL Database

SQL (Structured Query Language) databases are relational databases that use structured query language for defining and manipulating data. SQL databases, such as MySQL, PostgreSQL, and Oracle, are known for their ACID (Atomicity, Consistency, Isolation, Durability) properties, which ensure reliable transactions. Relational databases organize data into tables, which can be linked using foreign keys, allowing complex queries and transactions across multiple tables [Elmasri and Navathe 2006].

2.1.2 NoSQL Database

NoSQL databases are designed to handle unstructured data and provide flexible schema designs. Unlike SQL databases, NoSQL databases do not rely on a fixed schema, allowing for more dynamic and scalable data models. There are several types of NoSQL databases, including key-value stores, document stores, column-family stores, and graph databases. Examples include MongoDB, Cassandra, and Amazon DynamoDB. NoSQL databases are often used in big data and real-time web applications due to their scalability and performance [Cattell 2011, Stonebraker 2010].

2.1.3 Optimization Techniques

2.1.3.1 Indexing

Indexing is a fundamental technique used to improve the performance of database queries. By creating indexes on specific columns, databases can quickly locate and retrieve the required data without scanning the entire table. Indexes can be created on single or multiple columns and can significantly reduce query execution time. In SQL databases, common types of indexes include B-tree, hash, and bitmap indexes [Ramakrishnan and Gehrke 2003]. In NoSQL databases, secondary indexes are often used to enhance query performance [Cattell 2011].

2.1.3.2 Table Inheritance (Vertical Partition)

Table inheritance, or vertical partitioning, is an optimization technique where a table is divided into multiple smaller tables, each containing a subset of the columns. This can reduce the amount of data read during a query and improve performance. In PostgreSQL, table inheritance allows child tables to inherit columns from a parent table, enabling more efficient data organization and access patterns. This technique is particularly useful for managing large datasets with varying access patterns [Stonebraker et al. 2007].

2.1.3.3 Data Embedding

Data embedding involves embedding related data within a single document or record, reducing the need for complex joins and enhancing query performance. This technique is commonly used in document-oriented NoSQL databases like MongoDB, where related data is stored together in a single document. Data embedding can simplify data retrieval and improve performance, especially in read-heavy applications [Pokorny 2013].

2.2 Technological Background

2.2.1 JSON Based PostgreSQL

PostgreSQL, a popular open-source SQL database, has robust support for JSON (JavaScript Object Notation) data types. This allows PostgreSQL to handle semi-structured data and provide the flexibility of NoSQL databases while maintaining the relational model. JSON and JSONB (binary JSON) data types enable efficient storage, indexing, and querying of JSON documents within PostgreSQL. This hybrid approach makes PostgreSQL a versatile choice for modern applications that require both relational and document-oriented data management [Oblak et al. 2017].

2.2.1.1 GIN Indexes

Generalized Inverted Indexes (GIN) are a type of index in PostgreSQL designed to handle complex data types like arrays, JSONB, and full-text search. GIN indexes allow efficient indexing and querying of JSONB data, making them ideal for applications that require fast access to semi-structured data. By indexing the individual elements within a JSONB document, GIN indexes enable rapid retrieval and filtering of data, enhancing query performance [Oblak et al. 2017, Bayer and McCreight 1972].

2.3 Related Work

Numerous studies have explored the evolution of database schemas and workloads, highlighting the need for continuous optimization to maintain performance. Curino et al. (2013) discussed automating schema evolution in relational databases, emphasizing the importance of tools and methodologies to manage schema changes effectively [Curino et al. 2013]. Thor et al. (2017) proposed a benchmark for schema evolution, demonstrating the impact of schema changes on database performance [Thor et al. 2017].

In the context of NoSQL databases, Vogels (2009) introduced the concept of eventual consistency, which allows NoSQL databases to provide high availability and partition tolerance at the cost of temporary inconsistencies [Vogels 2009]. This principle underpins many NoSQL optimization strategies, enabling scalable and resilient data management.

Recent research by Liu et al. (2011) and Sakr et al. (2013) has focused on the performance implications of schema and workload evolution in both SQL and NoSQL databases. These studies provide valuable insights into the trade-offs and challenges associated with maintaining optimal database performance in dynamic environments [Liu et al. 2011, Sakr et al. 2013].

Overall, the related work underscores the necessity of adaptive optimization techniques to handle evolving database schemas and workloads effectively. This thesis builds on these foundational studies to develop and evaluate optimization strategies for the SDSS SkyServer, addressing the unique challenges posed by its extensive and evolving dataset.

Chapter 3

Case Study

3.1 Data

This section introduces the real-life database selected for analysis in this project.

3.1.1 Sloan Digital Sky Survey SkyServer

The Sloan Digital Sky Survey (SDSS) SkyServer is one of the most ambitious and influential astronomical databases, designed to map the cosmos in unprecedented detail. The SDSS SkyServer[1] provides extensive astronomical data accumulated over several phases of the survey. Each phase culminates in a Data Release (DR), encapsulating both refined and newly acquired observational data. Over the years, these releases have progressively expanded in scope and depth, integrating novel data from successive survey extensions and incorporating advanced processing techniques. As of the latest releases, SDSS data includes a wide array of astronomical measurements such as spectral data, photometric images, and detailed redshift measurements [SDS b].

The data in the SDSS SkyServer is updated through these periodic Data Releases, which typically occur annually and correspond to different phases of the ongoing survey. Each release not only adds new data but also enhances the quality of existing data, adhering to rigorous calibration standards to ensure accuracy and reliability [SDS b].

The most recent phase, SDSS-V, introduced its first data release, DR18, in January 2023, continuing to showcase the survey's commitment to expanding and enhancing the astronomical data available to the community [Almeida et al. 2023].

[1]Hosted at `https://www.sdss.org/`

Access to SDSS data is facilitated via the SkyServer's web interface, where users can query and download data through structured query tools and interfaces designed for both professional astronomers and educational purposes. Additionally, the server supports programmatic access via SkyQuery, a service that enables users to write and execute complex queries on the SDSS database programmatically, making it a versatile tool for scientific research and discovery [SDS a].

For the upcoming project, the Sloan Digital Sky Survey (SDSS) SkyServer has been selected as the real-time database for analysis and subsequent optimization. This choice is driven by several factors: the SkyServer is publicly accessible, providing open access to its actual operational workload which is crucial for this project. Moreover, the SkyServer's functionality allows users to access and analyze different stages of the database workload and the database itself, by allowing to access the different data releases, simplifying the task of tracking its evolution over time.

3.2 Workload

3.2.1 SDSS SkyServer Workload

The SDSS SkyServer provides an invaluable resource for tracking the server's operational workload through the *SqlLog* database, accessible via its web interface at SkyServer Traffic Log. This specialized tool is designed to capture and display real-time statistics on SQL query traffic, essential for both administrative management and academic research.

The *SqlLog* records details of both successfully and unsuccessfully executed SQL queries. Each query's data is logged by a stored procedure that executes each SkyServer query. The log database includes columns such as *theTime* (datetime of the query), *webserver* (URL of the server), *winname* (Windows name of the server), *clientIP* (client's IP address), and *sql* (the SQL statement executed), among others. It also captures performance metrics like *elapsed* (query execution time), *busy* (CPU time used by the query), and *[rows]* (number of rows returned by the query), providing a comprehensive snapshot of server activity.

For the upcoming project, access to the SDSS SkyServer's SqlLog will be essential. This log will be used to download samples of the workload for detailed processing and analysis. Such an analysis aims to deeply understand the characteristics of the workload, thereby enabling the optimization of the cost-effectiveness of database queries based on actual workload data.

The study will focus on analyzing the evolution of the database over a span of 10 years, starting with its state in December 2003, which corresponds to Data Release 1 (DR1). This starting point is chosen because the workload logs prior to 2003 show minimal activity. Subsequent analyses will cover December 2013 (corresponding to DR8), and December 2023, aligning with the latest data release. This approach allows us to examine how the database and its workload have evolved over the years and across different data releases.

Chapter 4

Methods

4.1 Tools and Technologies Used

This section provides a brief overview of the tools and technologies employed throughout this project.

Python: Python is the primary programming language utilized in this thesis. It is used extensively for processing and analyzing workloads, executing simulations, and measuring and analyzing results. The key libraries employed include:

- `pandas`: for data manipulation and analysis.
- `sqlparse`: for parsing SQL statements more easily.
- `re`: for regex operations, crucial in finding query patterns.
- `sqlalchemy`: for connecting to the PostgreSQL database.

DuckDB: DuckDB is an open-source, column-oriented relational database management system (RDBMS) that excels in online analytical processing (OLAP) tasks [Duc]. Designed for high performance across complex queries and large datasets, it is especially suitable for embedded configurations. One of its standout features is the ability to store data directly in files, which was particularly beneficial for integrating with JSON-based feature extraction files in this project. As an embeddable database, DuckDB operates without a server and does not require external dependencies, greatly simplifying deployment and minimizing overhead. These characteristics made it an invaluable tool for facilitating the query analysis process.

4.2 Project Workflow

This section describes the workflow of the project, as illustrated in Figure 4.1. It details the main tasks and their corresponding subtasks undertaken to achieve the objectives of the study.

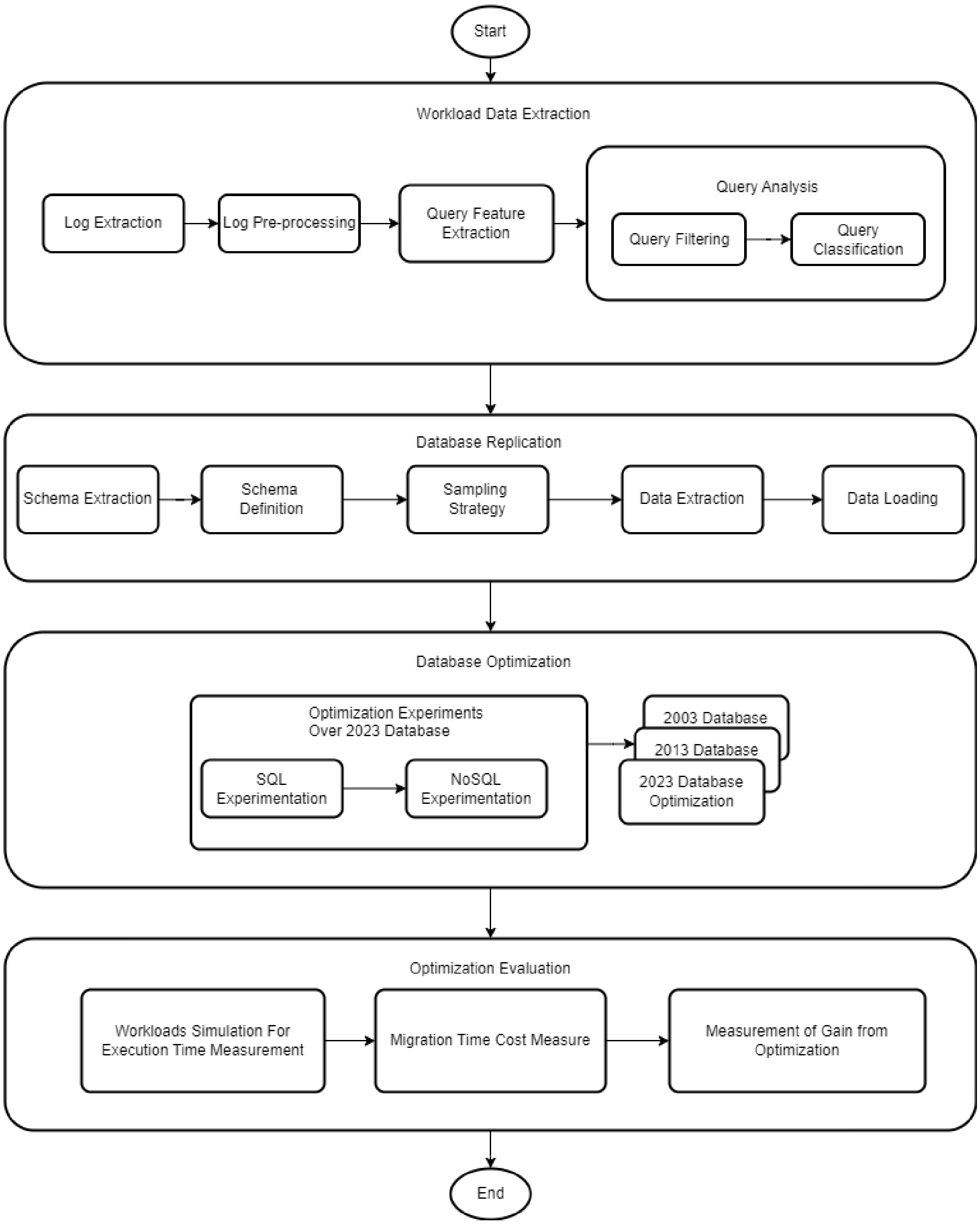


Figure 4.1: Project Workflow

4.3 Acronyms

This section provides a detailed explanation of some nomenclature that will be used throughout this report to facilitate understanding.

Schemas: These acronyms are primarily used to refer to the database schemas that have been optimized based on the analysis of previous years' workloads.

Schema Name	Definition
2003	2003 schema optimized based on its workload
2013	2013 schema optimized based on its workload
2013_03	2013 schema optimized based on the 2003 workload
2023	2023 schema optimized based on its workload
2023_03	2023 schema optimized based on the 2003 workload
2023_13	2023 schema optimized based on the 2013 workload

Table 4.1: Acronyms for the different defined schemas

4.4 Workload Data Extraction

4.4.1 Log Extraction and Pre-processing

Using the SkyServer Traffic Log tool, as previously described, the desired workloads were extracted from the SqlLog table by filtering according to the defined date intervals. The queries were specifically filtered to include only those that were successful, as there was no value in analyzing queries that failed to execute properly.

The workloads were downloaded as CSV files, which contained the executed SQL queries along with their respective timestamps. The data volumes from different years show a significant growth: the 2003 workload included 50,333 queries, the 2013 workload expanded to 1,489,126 queries, and the 2023 workload further increased to 3,623,390 queries. These figures clearly indicate a steady increase in the database's usage over time.

The subsequent step involved processing the extracted CSV files to facilitate further analysis. A Python script was developed to correctly format these files and filter only the relevant queries. The primary formatting challenge addressed was the incorrect quoting of some queries, which caused issues due to commas being interpreted as data separators. The script was thus designed to appropriately add quotes where necessary.

For the filtering step, understanding the nature of the database is crucial. As previously discussed, database access is provided through platforms such as SkyQuery and CasJobs. These platforms allow data access but restrict modifications, deletions, or additions, which are exclusively managed by the database administrators through Data Releases. CasJobs, particularly, serves as an online workbench for large scientific catalogs, offering a web environment that emulates and enhances local free-form query access. It provides a server-side, personalized user database where users can create and modify tables, functions, and other database objects. However, these modifications do not directly alter the main database.

Given the project's scope, which focuses solely on queries made directly to the SDSS SkyServer database. The script was specifically designed to exclude queries that operate on the server-side personalized user database, such as creating or manipulating personal tables and functions. This is because these queries do not interact directly with the main database. This filtering resulted in the retention of all 50,333 queries from 2003, as no queries needed exclusion. In 2013, 23,688 queries were filtered out, leaving 1,465,438 queries. For 2023, 11,993 queries were filtered, resulting in a final count of 3,611,397 queries.

4.4.2 Query Feature Extraction

Once the workloads were correctly processed, a Python script was developed to facilitate the extraction of crucial information from the queries for analysis and classification based on common patterns. The script effectively deconstructed the SQL statements. Given that the query syntax adhered to Microsoft SQL Server, the following components were extracted and subsequently exported in JSON format:

- **query**: Contains the full query.
- **select_columns**: A list of the columns being selected. Example: `['p.run', 'p.rerun', 'p.camcol']`.
- **tables**: A list of the tables queried, including those used in joins. Example: `['PhotoPrimary p']`.
- **join_details**: A list detailing the types of joins used in the queries. Example: `['LEFT JOIN']`.

- **join_columns**: A list of the join conditions. Example: `['table1.id = table2.id']`.
- **where_clause**: A string containing the full where clause. Example: `"WHERE column1 = 'value'"`.
- **order_by_clause**: A string containing the full order by clause. Example: `"column1, column2"`.
- **is_selection**: Flag that indicates if the query is a selection.
- **is_projection**: Flag that indicates if the query is a projection.
- **is_select_all**: Flag that indicates if the query uses a select "*" clause.
- **has_nested_queries**: Flag that indicates if the query includes nested queries.
- **aggregation**: Flag that indicates if the query includes an aggregation function such as `COUNT`, `AVG`, etc.
- **top_value**: Includes the number used in the TOP clause of the query if being used; otherwise, it is null. Example: `1, 100`.
- **distinct**: Flag that indicates if the query includes the DISTINCT clause.

4.4.3 Query Analysis

The primary objective of the Query Analysis process in this project is to categorize queries based on shared patterns. As detailed in Subsection 4.4.1, the analysis targets workloads that remain after user-based queries have been filtered out during Workload processing. Additionally, due to database restrictions that prevent regular users from modifying data—a responsibility assigned exclusively to professional database administrators—the workloads are composed solely of selection-type queries. Therefore, the classification of these queries is systematically organized: it starts with the tables they access, then the columns involved in the projections, and finally, it concentrates on the `WHERE` clause patterns, which represent the most variable component. The aim is to identify and generalize common patterns within these clauses to facilitate better query management and optimization.

The approach adopted for this task begins with an analysis of the table combinations used by the queries. This simpler initial step facilitates the initial clustering of queries into more manageable groups. The process is divided into two stages: a filtering stage, where queries not corresponding to the main table combinations are excluded, and a refinement stage, where focus shifts to the columns used and the `WHERE` clauses. This second stage aims to split the initial clusters into more precise categories based on their distinct patterns.

4.4.3.1 Query Filtering

As previously described, the query filtering process required an initial analysis to determine the most commonly accessed table combinations within the queries. A Python script was developed to parse the table names from the JSON files, count the various table combinations present in the queries, and analyze these combinations. This process involved normalizing the table names by converting them to lowercase and sorting them within their respective lists to ensure accurate grouping of combinations.

Consequently, for each workload, a dataset was generated that listed the table combinations, along with the counts and their corresponding percentages.

After analyzing these results, a minimum threshold of **1%** was established for the frequency of table combination occurrences. Only those combinations meeting or exceeding this threshold were retained for further analysis. This criterion was set to ensure that only the most representative queries of each workload were considered, thereby focusing on the most significant interactions within the database.

Ultimately, after filtering out table combinations below the 1% occurrence threshold, the remaining queries numbered 48,337 for 2003, 1,352,498 for 2013, and 3,485,018 for 2023. These totals represent 96.03%, 92.23%, and 96.5% of the original workloads for each respective year, involving only 3, 19, and 13 table combinations. This substantial reduction in the number of table combinations highlights the efficiency of the filtering process. As a result, the number of initial clusters necessary for query classification was significantly decreased, thereby streamlining the subsequent task of detailed classification within these clusters.

4.4.3.2 Query Classification

The classification of queries was performed using DuckDB, as previously described. This process involved two main outcomes. Firstly, each query was categorized into a distinct group based on the structure of the query construction. Secondly, the frequency of these categories was analyzed to provide a realistic depiction of how the workload was executed on the actual database. This analysis aimed to facilitate the simulation of real-life query executions on the database, using the frequency of each category as weights to guide the simulation process.

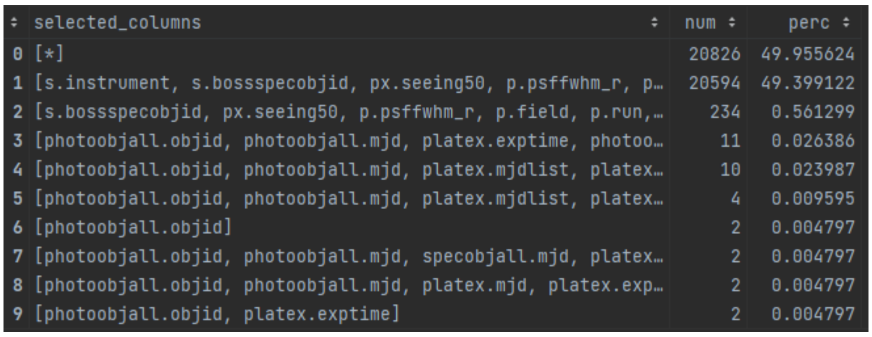
So for the analysis done over the workloads, the following process was undertaken for each workload to analyze the query patterns and classify them into different categories:

Using the table combinations as a starting point, the analysis proceeded for each table combination as follows:

1. **Analysis of the Selected Columns:** The analysis began by examining the patterns in the selected columns. This was accomplished by executing a query that counted the number of appearances of each list of selected columns and calculated the percentage of these occurrences. The corresponding SQL query is shown below:

```
SELECT selected_columns,
       COUNT(*) AS num,
       (num / SUM(num) OVER ()) * 100 AS perc
FROM result_{workload_year}_12
WHERE cleaned_tables = '{table_list}'
GROUP BY selected_columns
ORDER BY num DESC
```

   In this query, *cleaned_tables* refers to the list of table combinations used by the queries, formatted by ordering the tables alphabetically. It is important to note that in the `WHERE` clause, the operator '=' is used instead of 'IN'. This usage is specific to DuckDB, which interprets the list in *cleaned_tables* as a single string. Figure 4.2 illustrates a sample result generated by this query. Here, *selected_columns* displays the list of columns involved in the projection, *num* indicates the frequency of these selected columns, and *perc* represents the corresponding percentage of this frequency.



```
  selected_columns                                              num       perc
0 [*]                                                         20826  49.955624
1 [s.instrument, s.bossspecobjid, px.seeing50, p.psffwhm_r, p…  20594  49.399122
2 [s.bossspecobjid, px.seeing50, p.psffwhm_r, p.field, p.run,…    234   0.561299
3 [photoobjall.objid, photoobjall.mjd, platex.exptime, photoo…     11   0.026386
4 [photoobjall.objid, photoobjall.mjd, platex.mjdlist, platex…     10   0.023987
5 [photoobjall.objid, photoobjall.mjd, platex.mjdlist, platex…      4   0.009595
6 [photoobjall.objid]                                                2   0.004797
7 [photoobjall.objid, photoobjall.mjd, specobjall.mjd, platex…      2   0.004797
8 [photoobjall.objid, photoobjall.mjd, platex.mjd, platex.exp…      2   0.004797
9 [photoobjall.objid, platex.exptime]                               2   0.004797
```

Figure 4.2: Selected columns analysis per table combination result example

2. **Analysis of Where Clauses:** The next step involved repeating the analysis for the *where clauses*. This phase often required more manual effort using regular expressions, due to the slight variations in patterns within the clauses, such as filters by specific primary keys or range values. This approach allowed for the refinement of previously grouped clusters into more accurate subgroups that share the same patterns in the `SELECT` and `FROM` parts, with only slight changes in the *WHERE* clause.

3. **Analysis of Top_Value Column:** The analysis was then extended to the *top_value* column to maintain consistency in examining all query aspects. This step aimed to enhance the accuracy of categorization by considering details such as whether a query is limited to a single value (e.g., `TOP 1`) or not. Such distinctions are crucial as they impact query performance by altering the query plan generated by the DBMS. Therefore, this was an important detail to consider when refining the query categories.

4. **Global Pattern Recognition:** After identifying distinct patterns across the workload, a manual process was undertaken to associate common patterns among selected columns, WHERE clauses, and TOP values (if used). The most significant patterns were identified and established as the initial version of the possible categories. In some cases, it was necessary to disregard less significant patterns. This refined the categorization of query types as accurately as possible.

5. **Final Threshold Filtering:** With an initial classification of query patterns and their respective frequencies established, further filtering was applied to retain only the most relevant categories. A minimum threshold of 0.5% appearance rate within the workload was set for a category to be considered significant. This threshold was set at 0.5% because, as described in Section 4.4.3.1, the initial grouping by table combinations used a 1% threshold. However, after dividing these larger clusters into smaller ones, some of the new subclusters had appearance rates smaller than 1%. Maintaining the 1% threshold would have caused the disappearance of these significant clusters. Therefore, the threshold was reduced to 0.5% to preserve a representative portion of the initially defined clusters.

The result of the query classification process identified 5 query categories for 2003, 22 for 2013, and 21 for 2023.

4.5 Database Replication

To accurately measure the execution time costs of the workloads, it was necessary to replicate the real database for empirical analysis. This need arose because the platforms available for executing queries on the real server, such as SkyQuery and CasJobs, were not suitable for measuring these costs.

The SDSS SkyServer database, which contains extensive astronomical data collected over several phases of the survey, is exceedingly large, exceeding 10 TB per data release (DR). Consequently, it was impractical to download a complete replica of the database. Therefore, a tailored approach to replication was required.

4.5.1 Schema Extraction and Definition

Given that each workload corresponds to a different data release, it was essential to create a separate schema for each one. These schemas were designed to include only the necessary tables for each workload, thereby allowing for the download of only relevant data.

It is important to note that the original database catalog indicates that the database consists of both tables and materialized views. This is understandable, as views are commonly used in large databases to make querying tasks more efficient and simpler. For the purposes of this project, materialized views will be treated as tables both in the database replication and in the subsequent design of the optimized schemas.

This approach entailed a thorough, independent analysis of each workload, culminating in the development of three distinct schemas. These schemas are visually represented in Figures 4.3, 4.4, and 4.5 using Database Markup Language (DBML). For a clearer understanding, each DBML schema visualization is divided into two main parts. The first part displays the table distribution, illustrating the relationships between the tables and the overall schema logic. The second part shows the materialized views, which are depicted as independent tables located at the bottom of each image, in the last row of tables.

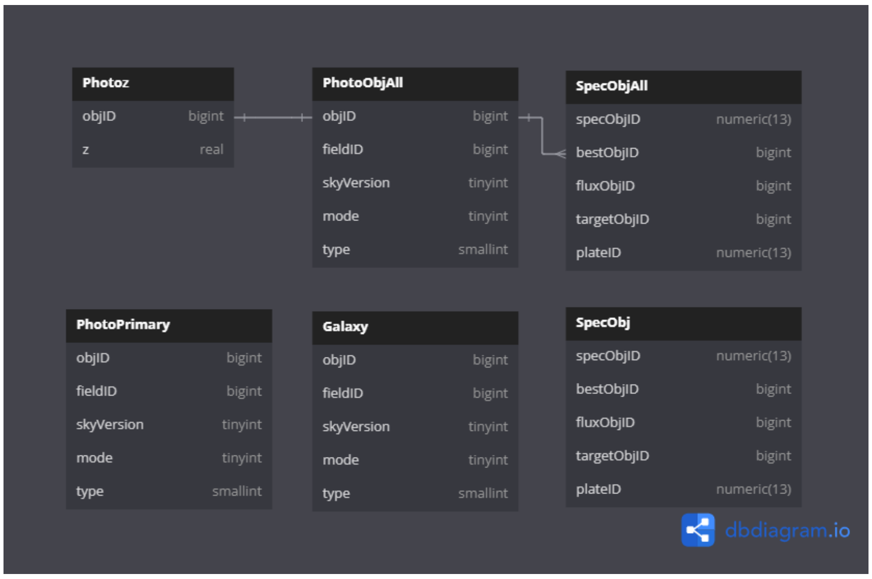


Figure 4.3: 2003 Schema

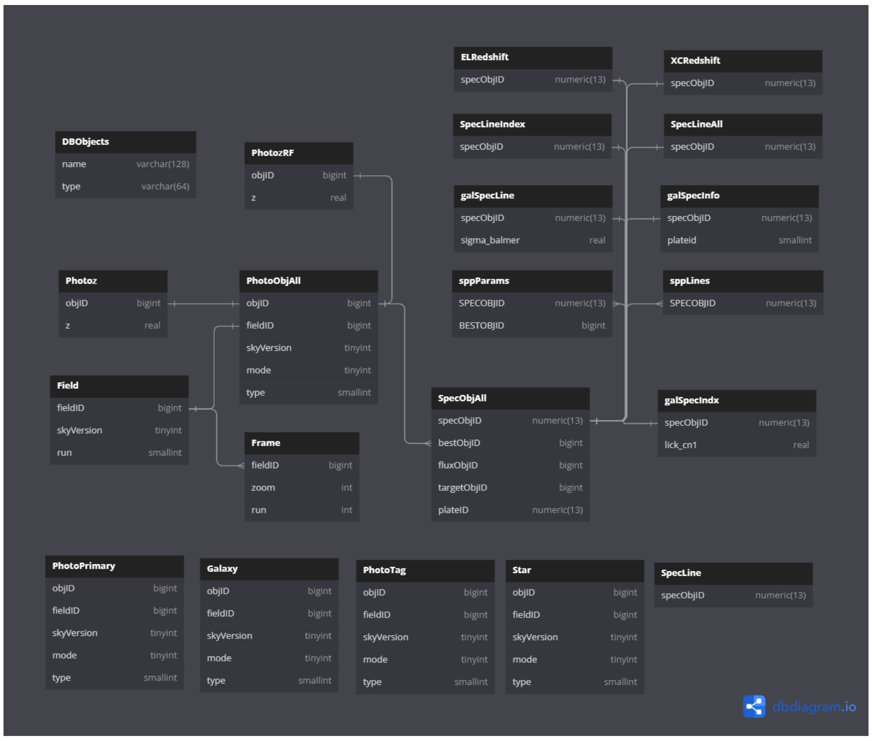


Figure 4.4: 2013 Schema

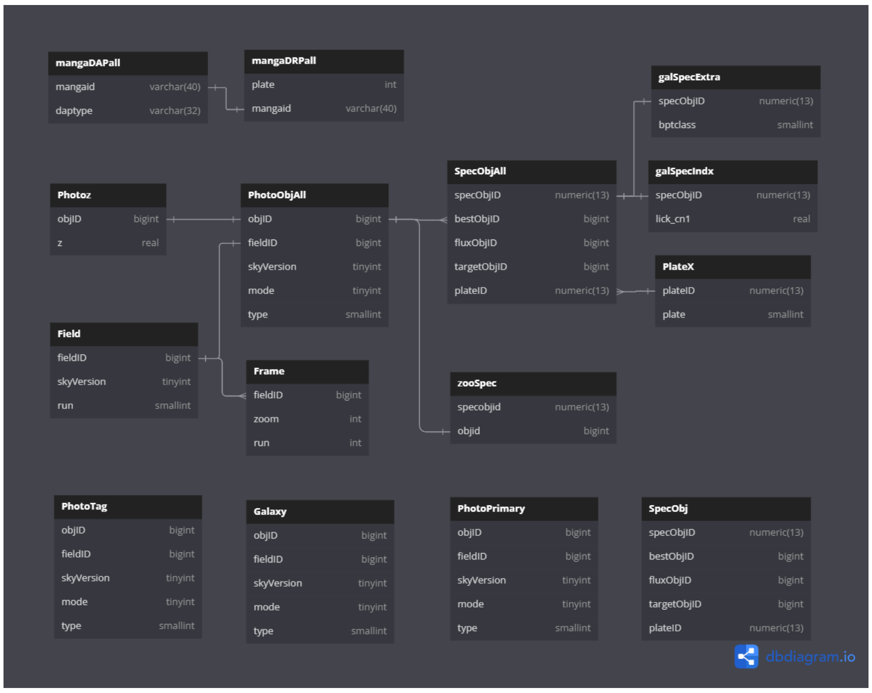


Figure 4.5: 2023 Schema

4.5.2 Sampling Strategy

In the process of schema replication, even after reducing the number of tables, certain tables remained that were exceptionally large, occupying approximately 5.08 TB and containing about 1,231,051,050 rows. Consequently, a sampling strategy was required to manage these vast datasets effectively.

To achieve a balanced representation across tables of varying sizes, a mathematical formula was applied to determine the number of sample rows required from each table. Let $|T|$ represent the cardinality of a table. The formula to calculate the number of sample rows required, denoted by S, is given by:

$$S = \lceil |T|^{0.6} \rceil$$

where $\lceil \cdot \rceil$ denotes the ceiling function, which rounds a number up to the nearest integer.

This formula strategically uses an exponent of 0.6, a fractional power less than one, to moderate the impact of extremely large row counts. By doing so, it disproportionately reduces the sample size required from larger tables, which helps in mitigating processing load while still preserving a representative sample from smaller tables. Consequently, this approach not only maintains feasibility with the available computational resources but also enables the practical execution of simulations, ensuring both efficiency and data representativeness.

4.5.3 Data Extraction and Loading

The next phase involved downloading the defined sample data and uploading it to a dedicated server. Originally, the database was hosted on Microsoft SQL Server. However, for replication, PostgreSQL was chosen to allow testing both SQL and NoSQL approaches in the same environment, as PostgreSQL offers the appropriate tools for this purpose.

The primary tool used for creating the database, tables, constraints, and downloading the data was CasJobs.

Initially, the Schema Browser feature of CasJobs was utilized to inspect the definitions of all tables and their respective constraints for a specific Data Release (DR). The 2003 workload corresponded to DR1, the 2013 workload to DR7 and DR8, and the 2023 workload to DR18, acknowledging that access to previous data releases remains despite updates.

Subsequently, the scripts to replicate the sample schemas were manually created, maintaining a similar structure as the original database with three distinct schemas for each replication database: `db_2003`, `db_2013`, and `db_2023`.

The next step involved downloading the sample data, aiming to preserve a realistic distribution reflective of the actual database. This included maintaining relationships and distributions pivotal for generating materialized views. Queries designed for downloading this data took into account the necessary table relationships and data distribution. CasJobs facilitated the download of data as CSV files. Challenges arose with queries that had lengthy execution times, needing the creation of temporary tables in CasJobs' personal database, which is limited to 500 KB. This limitation required splitting the tables into smaller segments and repeatedly processing this step to download all the data.

Finally, a Python script was developed to automate loading the data into the PostgreSQL database on the dedicated server. This utilized `pandas` and `sqlalchemy`, the latter facilitating database connections via Python. It is crucial to note that database replication was conducted on a dedicated server to ensure more reliable simulations later, minimizing measurement noise unlike scenarios where a database on a personal computer might be affected by background tasks, thus impacting performance.

4.6 Database Optimization

The project's subsequent phase involved optimizing each database in accordance with its specific workload. Two new schemas per database, one SQL and one NoSQL, were created to evaluate how these differing approaches could perform under the same conditions and determine the most suitable approach for the designated use case.

To facilitate this, the initial focus was on the 2023 database to experiment with the data using both SQL and NoSQL methodologies. Upon successfully establishing two optimized schemas for 2023, similar strategies were adapted for the 2003 and 2013 schemas, tailored to their respective workloads. It is important to note that optimization patterns varied due to the evolution of the workload over the years.

Subsequently, after establishing these optimized schemas, the next goal was to evaluate how earlier optimizations performed against later ones. For instance, for the 2023 database, four different schemas were assessed—two from 2003 and two from 2013, spanning both SQL and NoSQL approaches. This analysis was crucial to demonstrate that despite previous optimizations, the evolving nature of workloads needs periodic reevaluation and possible redesign of schemas to maintain optimal performance.

4.6.1 Optimization Strategies

Before delving into the specific optimization strategies employed in this project, it is crucial to outline the implementation of the NoSQL approach. In this project, the NoSQL paradigm is adopted using key-value data modeling to streamline the database design and maximize the advantages of key-value systems. This approach is implemented in PostgreSQL through a schema comprising two columns: a primary key column, which retains its original datatype, and a JSONB column that encapsulates all other attributes as a JSON object. Here, each attribute name serves as a key, and the attribute value as the corresponding value. This structural arrangement is depicted in Figure 4.6.

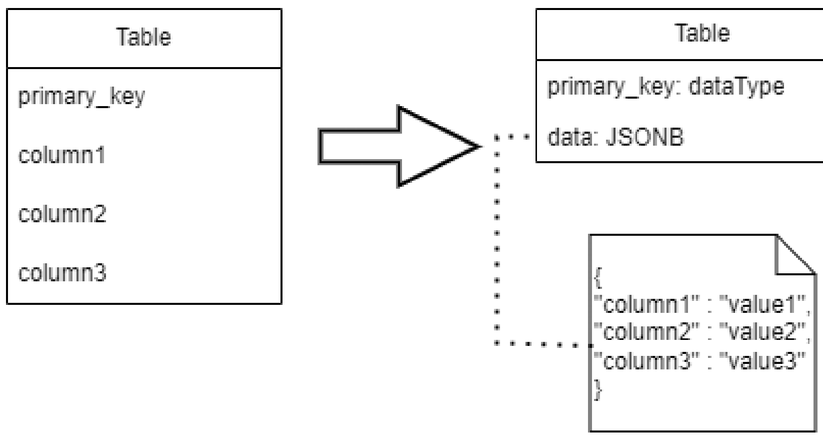


Figure 4.6: NoSQL Table Structure

This project employs straightforward but effective optimization strategies for both SQL and NoSQL approaches. These strategies include indexing and table inheritance for both approaches, with the addition of data embedding for the NoSQL approach. The optimization strategies are detailed below:

- **Indexing**: Indexes will be created based on the workload to optimize as many query types as possible, improving query performance.
- **Inheritance**: For both SQL and NoSQL, inheritance involves splitting a table into two parts. The parent table (‘TableNameMain‘) contains the most frequently used columns, while the child table includes all columns from the parent table plus additional, less frequently used columns. This structure reduces data read during queries, enhancing performance. This can be seen on Figure 4.7.

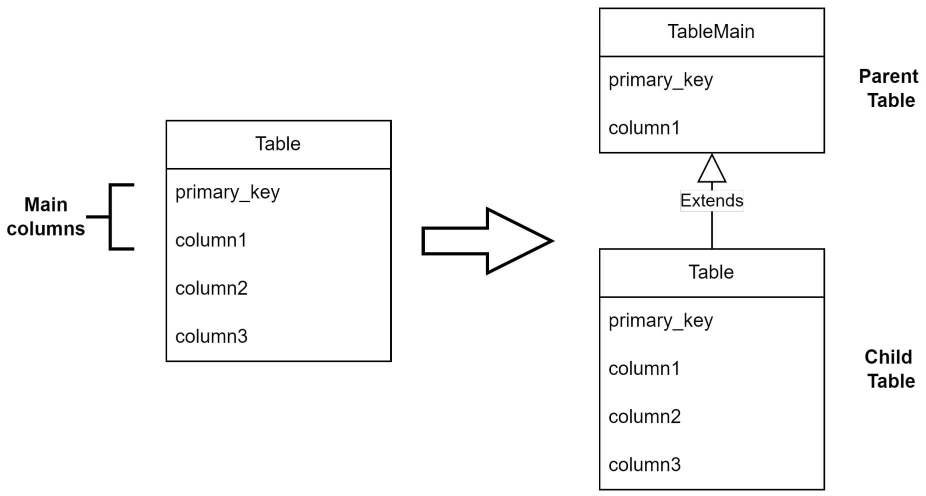


Figure 4.7: Table Inheritance Structure

In the NoSQL context, due to the key-value modeling, the JSONB column is split into two: one for the most important data and another named 'dataExtra' for the remaining data. This creates a key-data-dataExtra structure, as illustrated in Figure 4.8.

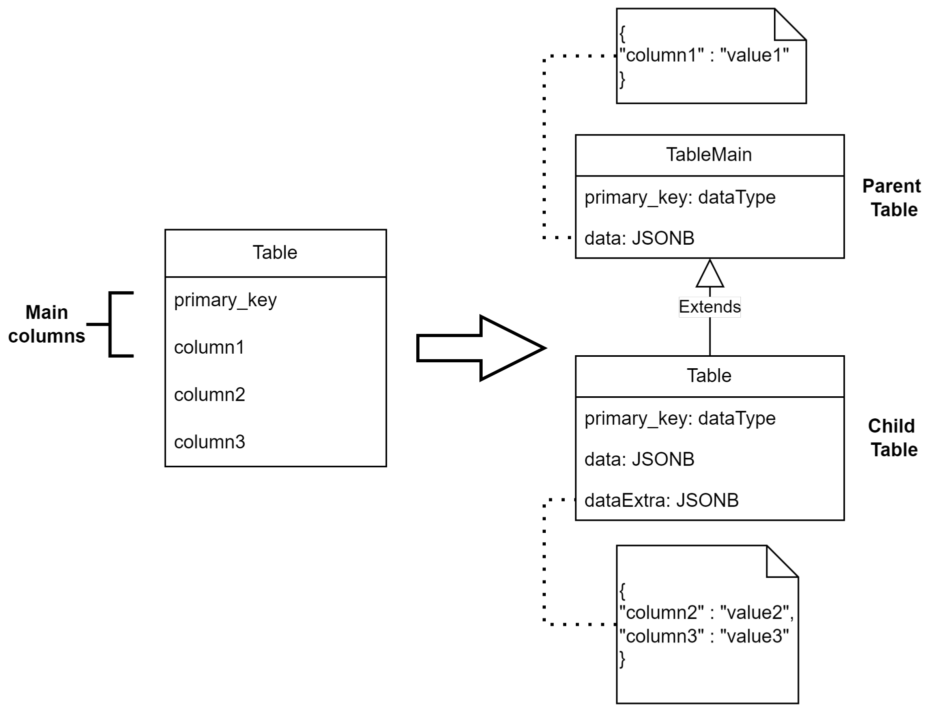


Figure 4.8: NoSQL Table Inheritance Structure

- **Embedding**: For the NoSQL approach, embedding is used to reduce the need for complex joins, which SQL traditionally excels at. Embedding related data within a single document leverages NoSQL's flexibility and improves query performance. This technique balances the performance differences between SQL and NoSQL without requiring a complete table redesign in the NoSQL context.

4.6.2 Experimentation over the 2023 Database

Before optimizing, a Python script was developed to simulate realistic workload executions. This script, leveraging pre-calculated weights for each query type, randomly generated queries from a set of templates by filling in parameters specific to each query class. Parameters varied widely—from IDs and lists of table IDs to floats generated using the mean and standard deviation of respective columns. The PostgreSQL `EXPLAIN ANALYZE` statement was utilized to measure execution times and provide detailed execution plans, aiding in identifying potential optimization points [PostgreSQL Documentation 2023]. The detailed pseudocode implemented is attached on the Appendix B.

The output from the simulation script contained details of each executed query, such as the query type, parameters used, execution plan, planning time, execution time, triggers, and the number of returned rows.

A Jupyter notebook was then used to analyze these results. Each simulation executed 11,000 randomly chosen queries, with the initial 1,000 discarded to mitigate cold cache effects, ensuring that the subsequent 10,000 queries provided comparable execution times. The analysis involved assessing the execution frequency and total execution time per query type, as well as computing the mean and standard deviation of the execution times to identify the most problematic queries and determine which tables required optimization.

As previously mentioned, the two optimization approaches were applied alternatively to achieve the two optimized schemas for the 2023 database (i.e, initially, experiments were conducted using a SQL approach, followed by a NoSQL approach).

4.6.2.1 SQL Optimization Experiments for 2023 Database

For the SQL schema, initial tests focused on creating new indexes to improve performance for the most time-consuming queries, this was done by using B-tree indexes. Further experiments involved doing vertical partitions to some tables. In

PostgreSQL, this technique is managed through table inheritance. This was specifically applied to large tables, dividing them column-wise to ensure that only the most accessed columns were included in the parent table, thus enhancing access efficiency.

4.6.2.2 NoSQL Optimization Experiments for 2023 Database

First, it is important to recall that the NoSQL approach used in this project utilizes the JSON data type functionality provided by PostgreSQL.

JSON Sizes Experiments: The initial experiments focused on the `PhotoObjAll` table for two primary reasons: it is the largest table, and the query types 2 and 3 from this workload are useful for assessing the performance of different querying approaches. Query type 2 involves filtering by a specific value, while query type 3 is a range-based query. The patterns for these query types are detailed below:

- **Query Type 2 of the 2023 Workload:**

```
SELECT u, g, r, i, z, objID, type
FROM db_2023.photoobjall
WHERE objid = {objid}
```

- **Query Type 3 of the 2023 Workload:**

```
SELECT ra, dec, clean, u, g, r,
    i, z
FROM db_2023.photoobjall
WHERE ((ra BETWEEN {ra1} AND {ra2})
AND (dec BETWEEN {dec1} AND {dec2}))
```

Four scenarios were defined for the `PhotoObjAll` table, all utilizing the same B-tree indexes identified in the SQL experiments. These scenarios enabled an assessment of the impact of splitting data into smaller JSONs, and whether accessing the parent or child table when using inheritance differs in performance terms. Furthermore, it was evaluated whether it is more efficient to handle different tables with inheritance or just one table with multiple columns. The detailed scenarios are:

- **S1**: key-data, where all columns except the key are stored in a JSON in the `data` column.

- **S2**: Using inheritance, the main table has key-data, and the child table adds the dataExtra column.
  - **S21**: Queries are executed on the main table.
  - **S22**: Queries are executed on the child table.
- **S3**: The same key-data-dataExtra concept but in the same table, with `data` and `dataExtra` as separate columns to assess if using inheritance affects performance.

Index Experiments: Subsequently, it was necessary to determine which type of index performed best for the most common query patterns over the workloads between B-tree and GIN indexes (which where previously explained on Subsection 2.2.1.1). Although GIN indexes are commonly recommended for JSON data in PostgreSQL, it was important to verify if this was the optimal choice for the specific cases studied or if continuing with B-tree indexes would be more advantageous.

This experiment involved the previously detailed Query Type 3, executed over two copies of the same table. Both tables followed the NoSQL definition but utilized different index types. To ensure a fair comparison of the indices' performances, the experiment included a cache-warming phase where a significant number of queries were executed and then excluded from the analysis. Specifically, 3,000 queries were executed, with the first 1,000 being excluded to mitigate the cache's impact on performance.

The definitions of the two indices used in the experiment are as follows:

- **B-tree Index Definition:**

```sql
CREATE INDEX idx_PhotoObjAll1_ra_dec
ON db_2023_ns.PhotoObjAll1(
    CAST(data->'ra' AS FLOAT),
    CAST(data->'dec' AS FLOAT)
);
```

- **GIN Index Definition:**

```sql
CREATE INDEX idx_PhotoObjAll2_gin
ON db_2023_ns.PhotoObjAll2 USING gin (
    (data->'ra'),
    (data->'dec')
);
```

Limit Clause over Inheritances Experiments: An investigation was also conducted into the impact of using the limit clause on inheritance structures. It was observed that queries including a limit clause performed worse when accessing the parent table compared to the child table. Reasons for this discrepancy were explored to inform further optimizations.

This experiment utilized Query Type 6 from the workload, specifically chosen because it employs the `TOP 1` clause, which significantly impacts performance analysis. Due to space constraints, only the relevant part of the SQL query is shown below:

- **Query Type 5 of the 2023 Workload:**

  ```
  SELECT ...
  FROM db_2023.PhotoPrimary
  WHERE p.objid IN ({objidlist})
  LIMIT 1;
  ```

The experiment was designed to compare the performance of this query under two different scenarios:

- **S1**: The query is executed using the parent table.
- **S2**: The query is executed using the child table.

These scenarios were set up to determine the efficiency of query execution when accessing data from the main table compared to a partitioned child table, providing insights into the optimal database schema configuration for similar query types.

Embeddings: Finally, to leverage several NoSQL capabilities such as Document-Centric Design and Application-Centric Data Modeling, the embeddings optimization technique was applied to some NoSQL schemas. This strategy was implemented with the goal of achieving querying performance comparable to SQL. For example:

- **Galaxy**: Values used in queries from joined tables were embedded into this table to avoid joins.
- **GalSpecIndx and GalSpecExtra**: Given the one-to-one relationship of them, with `GalSpecExtra` containing additional information about `GalSpecIndx`, it was decided to treat `GalSpecExtra` as a child table to reduce the number of joins.

4.6.3 2003 and 2013 Databases Optimization

As previously discussed, the enhancements implemented for the 2023 database were replicated for the 2003 and 2013 schemas. For the SQL schemas, this included the addition of indexes and the application of the inheritance approach to certain tables. Similarly, for the NoSQL schemas, embeddings were incorporated alongside the other improvements.

The final detailed resultant optimized schemas can be checked on Appendix C.

4.7 Optimization Evaluation

This section outlines the evaluation process for the optimizations implemented. It details the methods used to measure the execution times of the workloads, the associated migration costs, and the calculations performed to determine the overall gains from these optimizations. The aim is to provide a comprehensive analysis of the efficiency and cost-effectiveness of the applied optimization strategies.

4.7.1 Workloads Simulation Execution Time Measurement

Following the optimization of all databases, the execution times for each database and its corresponding workloads were measured in milliseconds. This measurement was performed following the same simulation process done for the experimentation done over the 2023 database detailed on the Subsection **??**. These execution times serve as a metric for the cost of workload execution and will later be used to assess the value of the optimizations.

4.7.1.1 Simulations Assessing the Impact of Previous Optimizations

The final step involved evaluating how earlier optimizations impacted later ones. For the 2023 database, four different schemas were assessed—two each from the 2003 and 2013 schemas, covering both SQL and NoSQL approaches. Additionally, the two 2003 optimizations were tested on the 2013 database.

This series of experiments was designed to determine whether the evolving nature of workloads requires periodic reevaluation and potential redesign of schemas to maintain optimal performance. Furthermore, these experiments provided insights into the long-term effects of choosing SQL or NoSQL optimization strategies based solely on the specific workload characteristics of a given timespan.

4.7.2 Migration Cost Measurement

After optimizing the schema for each workload, it became necessary to measure the migration costs from the original schemas to the optimized ones. These costs varied significantly between the SQL and NoSQL approaches. Since the original schemas were designed for a relational database, the SQL approach primarily required simple optimizations such as creating indexes and adding new tables using the inheritance approach. In contrast, the NoSQL approach required migrating all relational tables into JSON-based structures.

Despite these differences, it was essential to quantify the costs of each migration to determine whether the effort was justified.

To facilitate these measurements, a Python script was developed to automate the evaluation process. The script utilizes SQL migration scripts to capture and measure the execution times of each statement. Initially, the `EXPLAIN ANALYZE` clause was employed; however, it proved effective only with `CREATE TABLE AS` and `INSERT INTO` statements. For other SQL commands such as `CREATE INDEX`, `CREATE SCHEMA`, `CREATE TABLE`, and `ALTER TABLE`, this clause did not yield the necessary data. To address this limitation, a custom SQL function was created to accurately measure execution times for all types of statements. The accuracy of this function was validated by comparing its results with those obtained from the `EXPLAIN ANALYZE` clause, revealing only negligible differences. Consequently, the Python script was configured to selectively invoke either measurement method based on the type of SQL statement being executed, and subsequently exported the results for analysis.

To facilitate these measurements, a Python script was developed to automate the task. The script utilized SQL migration scripts to read each statement and measure its execution time. Initially, there was an attempt to use the `EXPLAIN ANALYZE` clause; however, this clause only worked with `CREATE TABLE AS` and `INSERT INTO` statements. For other statements such as `CREATE INDEX`, `CREATE SCHEMA`, `CREATE TABLE`, and `ALTER TABLE`, the clause was ineffective. To overcome this limitation, an SQL function was implemented to measure execution times. The accuracy of this function was validated by comparing its results with those obtained from the `EXPLAIN ANALYZE` clause, revealing only negligible differences. Consequently, the Python script was configured to selectively invoke either measurement method based on the type of SQL statement being executed, and subsequently exported the results for analysis.

4.7.2.1 Migration Cost Time Measure from Previous Optimizations

In the evaluation of optimized execution times, there was an interest in maintaining previous optimizations and testing their performance across corresponding workloads. Specifically, this analysis aimed to determine how the migration costs, measured in milliseconds, were affected when existing optimizations were preserved and subsequently required further optimization. For this analysis, it is assumed that previous optimizations have been implemented, necessitating migration from these schemas to the newly optimized versions.

The measurements focus on specific years, namely 2013 and 2023, examining the impact of each prior optimization. Migrations are conducted using consistent methodologies; for instance, migrating from the SQL schema optimized in 2013, which was based on enhancements from 2003, to the 2013 SQL optimizations. A similar approach is applied for NoSQL schemas, ensuring that migrations are not cross-referenced between SQL and NoSQL frameworks.

These additional measurements were conducted for hypothetical migrations planned between the 2013 and 2023 databases, which were purportedly optimized based on earlier data, to their corresponding optimized schemas. For instance, migrations were simulated from the 2013_03 schema to the optimized 2013 schema. This process was exclusively applied to the 2013 and 2023 databases. Additionally, the migrations were restricted to the same paradigms, ensuring that migrations occurred only from SQL to SQL and NoSQL to NoSQL, thereby avoiding cross-paradigm transfers.

4.7.3 Measurement of Gain from Optimization

Utilizing the results from workload simulations and migration cost execution time measures, this section evaluates the feasibility and benefits of migrating to an optimized database. The analysis focuses on comparing execution times of queries before and after optimization, aiming to determine the break-even point in days, at which the benefits of migration justify the costs. Below, the calculated metrics used in this evaluation are detailed:

1. **Total Execution Time Baseline (T_{base}):** The total time required to execute 10,000 queries on the current database system.

2. **Total Execution Time Optimized (T_{opt}):** The total time required to execute 10,000 queries on the optimized database system.

3. **Average Gain Per Query (G_{query}):** The reduction in execution time per query, calculated as:
$$G_{query} = \frac{T_{base} - T_{opt}}{10000}$$

4. **Average Number of Queries per Day (Q_{day}):** Represents the average number of queries processed daily by the system of the resultant workloads after all the filtering.

5. **Average Daily Time Improvement (I_{day}):** The daily time savings achieved through optimization, expressed as:
$$I_{day} = G_{query} \times Q_{day}$$

6. **Migration Cost ($C_{migration}$):** The overall cost of the migration, quantified in milliseconds of execution time.

7. **Break-even Period ($D_{break-even}$):** The duration required for the optimization's time savings to recoup the migration costs, calculated by:
$$D_{break-even} = \frac{C_{migration}}{I_{day}}$$

These calculations facilitate the assessment of the economic viability and efficiency gains from the database migration. The break-even analysis provides a quantifiable basis for determining whether the investment in migration is warranted, based on expected improvements in daily operations.

Chapter 5

Results

5.1 Workload Analysis

This section presents the results obtained from each subtask comprising the workload analysis.

5.1.1 Workload Summaries

After completing the feature extraction for each of the workloads and doing the respective filtering, it was important to analyze the percentages of the different characteristics that the queries exhibited in each workload. The results are shown in Tables 5.1, 5.2, and 5.3.

Metric	Count	Percentage (%)
Aggregation	83	0.17
Top Value	28,345	58.64
Distinct	18	0.04
Is Selection	48,228	99.77
Is Projection	47,470	98.21
Has Nested Queries	119	0.25

Table 5.1: Summary statistics of the December 2003 workload

Metric	Count	Percentage (%)
Aggregation	2,144	0.16
Top Value	139,602	10.32
Distinct	2,378	0.18
Is Selection	1,261,045	93.24
Is Projection	484,682	35.84
Has Nested Queries	408	0.03

Table 5.2: Summary statistics of the December 2013 workload

Metric	Count	Percentage (%)
Aggregation	117,559	3.37
Top Value	519,605	14.91
Distinct	250,633	7.19
Is Selection	3,396,148	97.45
Is Projection	2,596,268	74.50
Has Nested Queries	63,038	1.81

Table 5.3: Summary statistics of the December 2023 workload

The data presented in Tables 5.1, 5.2, and 5.3 offer insights into the evolution of workloads over the years.

The results reveal consistent patterns across the three time periods: a small percentage of the queries are of the aggregation type, consistently below 3.5%, and an even smaller percentage are nested queries, never exceeding 2%. Furthermore, at least 93% of the queries across all periods are selection queries, which predominantly include WHERE clauses. This consistency underscores common usage trends in query types over time.

Despite these enduring patterns, notable changes are observed over time. For instance, the percentage of nested queries, although small, has increased over the years. Similarly, the use of the DISTINCT clause has grown. In contrast, the application of the TOP clause and the proportion of queries that are projections have shown fluctuating trends—initially decreasing, then increasing again.

These results underscore that while some patterns remain constant in different workloads, significant variations do occur over time, reflecting the evolving nature of workloads. This evolution implies that the load on databases varies over time, needing periodic adjustments to optimize performance and maintain efficiency.

Additionally, the average number of queries per day for each workload was calculated. It is important to note that these statistics were derived from the final filtered workloads and will be used to measure the overall gain from the optimizations. These figures are presented in Table 5.4.

Year	Num. Queries
2003	1,560
2013	43,629
2023	268,079

Table 5.4: Average Number of Queries per Day by Workload

5.1.2 Query Classification

The classification results of the queries are presented in three tables: Table 5.5 details the 2003 workload, Table 5.6 the 2013 workload, and Table 5.7 the 2023 workload. Each table organizes the query types according to the table combinations utilized, illustrating the percentage each table combination contributes to the overall workload. Additionally, these tables provide the specific percentages that each query type represents within their respective combinations.

For further details on the query types, such as the patterns each query type follows, please refer to Appendix A.

Table Comb.	**Table Comb. %**	**Query Type**	**Query Type %**
photoprimary	82.76	1	60.30
		2	22.46
galaxy, photoz, specobjall	16.15	3	04.75
		4	11.40
specobj	1.09	5	1.09

Table 5.5: 2003 Query Classification Results

Table Comb.	Table Comb. %	Query Type	Query Type %
phototag	15.04	1	15.04
galaxy, photoobjall, photoz, photozrf	14.13	2	14.13
photoobjall	10.78	3	10.78
photoprimary	11.99	4	10.66
		5	1.33
dbobjects	4.93	6	1.67
		7	1.61
		8	0.65
		9	1.00
specobjall	5.47	10	5.47
star	4.64	11	4.64
photoz	4.68	12	4.68
frame	3.49	13	3.49
speclineindex	3.07	14	3.07
field	2.63	15	2.63
galspecindx	2.73	16	2.73
galspecinfo	2.72	17	2.72
galspecline	2.68	18	2.68
elredshift	2.66	19	2.66
xcredshift	2.52	20	2.52
sppparams	2.48	21	2.48
specline	1.75	22	1.75
spplines	1.61	23	1.61

Table 5.6: 2013 Query Classification Results

Table Comb.	**Table Comb. %**	**Query Type**	**Query Type %**
photoobjall	42.54	1	21.85
		2	16.16
		3	4.53
photoprimary	14.09	4	7.93
		5	6.16
galaxy, photoz, specobj	9.03	6	9.03
photoobj	8.20	7	7.60
		8	0.60
specobj	7.67	9	7.67
galspecextra, galspecindx, specobjall	7.69	10	7.69
photoobjall, specobjall	2.13	11	0.61
		12	0.55
		13	0.97
photoobjall, zoospec	1.87	14	1.87
fgetnearbyframeeq, field, frame	1.62	15	1.62
photoz	1.42	16	1.42
phototag	1.27	17	1.27
photoobjall, platex, specobjall	1.23	18	0.62
		19	0.61
mangadapall, mangadrpall	1.24	20	0.62
		21	0.62

Table 5.7: 2023 Query Classification Results

The data in Tables 5.5, 5.6, and 5.7 illustrate the evolution of database schemas and workloads over the years. A notable observation is the change in the number of tables used in different workloads. For example, the 2003 results show usage of only 5 tables, which increases in later years. Specifically, the 2013 and 2023 schemas demonstrate an expansion in table usage, and between 2013 and 2023, some tables appear to become deprecated, as they are no longer used significantly by 2023.

Regarding query classifications, the diversity of query patterns also changes over time: there are 5 distinct patterns in 2003, 23 in 2013, and 21 in 2023. The only query type consistently observed across all three workloads with a significant frequency is associated with the table `PhotoPrimary`, which maintains a representative appearance frequency of at least 11.99% in each period. The absence of similar patterns across other query types further underscores the dynamic evolution of workloads, reflecting changes in database interaction and schema design over the years.

5.2 Database Optimizations

5.2.1 Experimental Results for the 2023 Database

This subsection presents the results of the experiments conducted on the 2023 database to obtain the two desired optimized schemas: the SQL and NoSQL versions.

5.2.1.1 SQL Optimization Experiments for 2023 Database

The outcomes of the SQL approach for the 2023 schema optimization are detailed below, emphasizing both total and mean execution times for enhanced clarity. The analysis specifically targets the top three costliest queries in terms of both total and mean execution times, illustrating their performance before and after optimization.

Table 5.8 showcases the significant reductions in total execution time for the three most time-consuming queries, as well as for the entire workload. This highlights the effectiveness of the schema optimizations in reducing execution overhead.

	Original Schema		Index Improvement		Index + Inheritance Imp.	
Query Type	Total Ex. (ms)	%	Total Ex. (ms)	% Imp.	Total Ex. (ms)	% Imp.
Q3	53,693.472	100.00%	45,894.266	85.47%	22,125.605	41.21%
Q6	16,620.061	100.00%	16,561.127	99.65%	15,756.741	94.81%
Q4	14,332.219	100.00%	13,353.945	93.17%	9,964.759	81.72%
Total	92,908.09	100.00%	85,456.556	91.98%	60,325.25	64.93%

Table 5.8: Total Execution Time Comparisons of the Most Expensive Queries With Different SQL Schema Improvements for 2023 Database

Meanwhile, Table 5.9 details the improvements in mean execution time, focusing specifically on the top three queries with the highest average execution time. These enhancements demonstrate the targeted efficiency gains achieved through optimization strategies applied to these particular queries.

	Original Schema		Index Improvement		Index + Inheritance Imp.	
Query Type	Mean Ex. (ms)	%	Mean Ex. (ms)	% Imp.	Mean Ex. (ms)	% Imp.
Q3	155.633	100.00%	145.696	93.62%	60.018	38.56%
Q11	37.918	100.00%	35.796	94.40%	29.997	79.11%
Q12	35.399	100.00%	35.213	99.47%	29.448	90.51%

Table 5.9: Mean Execution Time Comparisons of the Most Expensive Queries With Different SQL Schema Improvements for 2023 Database

The effectiveness of the SQL optimization approach for the 2023 schema is obvious from the data presented in Tables 5.8 and 5.9. Adding indexes resulted in a modest improvement in overall database workload performance, reducing costs by approximately 9%. However, the more significant impact comes from applying inheritance strategies to the most problematic tables. This approach reduced the execution time costs to 64.93% of the original, pre-optimized schema values.

A marked improvement is particularly noticeable in the most expensive query type, Q3. For this query, the mean execution time was significantly reduced to just 38.56% of the pre-optimization value, and the total execution time decreased to 41.21% of its initial value. These substantial reductions highlight the dramatic enhancements achieved through focused optimization efforts, significantly contributing to the overall efficiency of the database.

5.2.1.2 NoSQL Optimization Experiments for 2023 Database

This section presents the results of the experiments conducted using the NoSQL approach for optimizing the 2023 database. Given the specific nature of these experiments, the results are displayed in separate tables for clarity.

JSON Sizes Experiments: As previously described, the experiments aimed to evaluate the impact of handling access to columns with different JSON sizes. The results from simulating the execution of Query Types 2 and 3 for the 2023 workload, across each defined scenario, are shown in Table 5.10.

Scenario	Query Type 2 (ms)	Query Type 3 (ms)
S1	0.777324	6427.875324
S21	0.123202	74.203031
S22	0.104672	76.419371
S3	0.110893	75.536992

Table 5.10: Mean Execution Time for of the JSON Sizes Experiments over 2023 Database

The data from Table 5.10 demonstrates that the execution times for accessing specific fields in JSON-based columns are significantly affected by the size of the JSON. As the JSON size increases, the execution time also increases, sometimes substantially depending on the query type. For instance, for query type 2, when the JSON is not split by the use of inheritance, the execution time is seven times greater than when it is split. For query type 3, the cost escalates to approximately 86 times higher. These findings underscore the importance of appropriately sizing and structuring JSON documents in a NoSQL environment to optimize performance.

Further analysis revealed that for general queries accessing a table that employs inheritance, there is negligible difference whether access is through the parent or the child table. Similarly, no significant performance difference was observed between handling multiple tables with inheritance versus a single table with multiple columns. Consequently, the decision was made to maintain the inheritance approach with two distinct tables for better data consistency.

Index Experiments: The outcomes of the index experiments, which assessed the performance of different types of indices on Query Type 3, are presented in Table 5.11. This experiment tested both GIN and B-tree indices to identify the most effective index type for handling JSON data within PostgreSQL. The findings highlight the relative efficiencies and potential advantages of each indexing strategy in the tested scenarios. The measuring unit was the mean execution time of the query with each index type, because the same query pattern (Q3) was executed multiple times only changing the range parameters of used on the WHERE clause, so it was needed to test the mean performance of them.

The outcomes of the index experiments, which evaluated the performance of different types of indices on Query Type 3, are presented in Table 5.11. This experiment compared GIN and B-tree indices to determine the most effective type for managing JSON data within PostgreSQL. The findings elucidate the relative efficiencies and advantages of each indexing strategy under the conditions tested. The primary metric was the mean execution time of the query, using each index type. The same query pattern (Q3) was executed multiple times, varying only the range parameters in the WHERE clause, to assess the average performance of each index type.

Index Type	**Mean Execution Time (ms)**	**% Imp.**
Q3 Baseline	155.633	100.00%
Q3 w/ B-tree	82.795	53.20%
Q3 w/ GIN	124.341	79.89%

Table 5.11: Mean Execution Time by Index Type Experiments over 2023 Database

Table 5.11 demonstrated that for the query patterns observed in this database, regular B-tree indexes outperformed GIN indexes. This superiority was quantitatively supported with B-tree indexes achieving a mean execution time of only 53.20% of the baseline, compared to GIN indexes, which reduced execution time to 79.89% of the baseline. This outcome can be attributed to the nature of the queries involved. Although GIN indexes are effective for JSON-based columns, they are less efficient for typical queries, such as those involving range operations, which do not exclusively manipulate JSON data. Therefore, B-tree indexes were chosen for all optimizations in this project, as the predominant query types did not involve JSON-based operations that would benefit from the specific advantages offered by GIN indexing.

Limit Clause over Inheritances Experiments: The results of the experimentation done to evaluate how query execution efficiency varies between accessing data from the main table versus a derived or partitioned child table when using a NoSQL approach are shown in Table 5.12.

Scenario	**Mean Execution Time (ms)**	**%**
S1	9.360034	100.00%
S2	1.637353	17.49%

Table 5.12: Mean Execution Time for Query Type 5 for Limit Clause over Inheritances Experiments over 2023 Database

The experimental results, as displayed in Table 5.12, reveal a significant disparity in performance when the limit clause is employed in queries that access inheritance structures. Specifically, for the case of Query Type 5 (Q5), accessing the parent table resulted in a mean execution time approximately nine times longer than that observed when accessing the child table. These findings suggest that for queries incorporating limit clauses—especially those that retrieve a minimal number of records—database schemas can be substantially optimized by favoring child tables. Based on these insights, this approach will be applied to the original workload in the transition to the optimized schemas, aiming to enhance overall query performance significantly.

Embeddings: Similar to the SQL approach, this section presents the results of the final optimization phase, which included embedding techniques applied to specific tables—namely, Galaxy and GalSpecIndex+GalSpecExtra. These enhancements are aimed at improving both the total and mean execution times, thereby increasing the overall efficiency of the database schema.

The impact of embeddings on the execution times for specific queries is documented comprehensively. Table 5.13 displays the changes in total execution time for these queries, as well as for the entire workload, illustrating the effectiveness of the schema optimizations.

	Original Schema		Index + Inh. + Emb.	
Query Type	Total Ex. Time (ms)	%	Total Ex. Time (ms)	% Imp.
Q6	18,486.692	100.00%	1,858.243	10.05%
Q10	146.336	100.00%	92.676	63.33%
Total Execution	92,908.09	100.00%	58,619.93	63.09%

Table 5.13: Total Execution Time Comparisons for Queries affected by the NoSQL Embedding over 2023 Database

Furthermore, Table 5.14 provides detailed insights into the improvements in mean execution times, concentrating on the queries that were directly affected by the embedding optimizations.

	Original Schema		Index + Inh. + Emb.	
Query Type	Mean Ex. (ms)	%	Mean Ex.(ms)	% Imp.
Q6	27.592078	100.00%	2.944918	10.67%
Q10	0.243488	100.00%	0.163739	67.25%

Table 5.14: Mean Execution Time Comparisons for Queries affected by the NoSQL Embedding over 2023 Database

Tables 5.14 and 5.13 provide compelling evidence of the benefits derived from implementing embedding in NoSQL databases. Specifically, the mean execution times for queries Q6 and Q10 are reduced to just 10.67% and 67.25% of their original times, respectively, demonstrating the substantial impact of this optimization. These improvements are not only reflected in the mean execution times but are also corroborated by significant gains in total execution time for these queries and the overall workload, as detailed in Table 5.13. This data reinforces the value of applying embedding techniques and leveraging NoSQL characteristics to enhance database performance, potentially counterbalancing the superior querying capabilities typically associated with SQL databases.

5.3 Optimization Evaluation

5.3.1 Workloads Simulation Execution Time Measurement

Finally, after applying the same optimization techniques used for the 2023 database to the 2003 and 2013 databases, tailored to their specific workloads, the results are presented. Table 5.15 compares the final execution times from complete simulations of the entire workload across the original, SQL optimized, and NoSQL optimized schemas. This table displays the total execution times and illustrates the percentage reductions in time compared to the original schema, effectively demonstrating the impact of the optimizations.

Schema	2003 Database		2013 Database		2023 Database	
	Total Ex. Time (ms)	%	Total Ex. Time (ms)	%	Total Ex. Time (ms)	%
Original	107186.22	100.00%	57809.488	100.00%	92908.09	100.00%
SQL Optimized (Idx.+ Inh.)	91877.06	85.72%	38004.742	65.74%	60325.25	64.93%
NoSQL Optim. (Idx+Inh.+Emb.)	84051.133	78.42%	36847.382	63.74%	58619.93	63.09%

Table 5.15: Comparative Analysis of Total Simulation Times Across Database Optimizations

The simulations of executing corresponding workloads over the original and optimized schemas, both SQL and NoSQL, as depicted in Table 5.15, successfully achieved the initial goal of optimizing each year's schema based on specific workload demands. This is demonstrated by substantial reductions in total execution times across all cases. The same optimization principles were consistently applied, leading to comparable execution time reductions for both the 2013 and 2023 scenarios. Specifically, in the SQL approach, the total execution time was reduced to approximately 65% of the original for both years, while in the NoSQL approach, it was reduced to about 63.5%. However, the 2003 schema, having significantly fewer tables, displayed improvements that, though positive, were less pronounced than in later years.

Moreover, on a global scale, the NoSQL approach, especially with the incorporation of embedding techniques, managed to outperform the SQL approach. However, the advantage in 2013 and 2023 was not significantly large, demonstrating that SQL, even with fewer optimizations, can nearly match NoSQL in querying performance.

5.3.1.1 Simulations Assessing the Impact of Previous Optimizations

The final simulation results, which evaluate how earlier optimizations impacted later ones, are presented in this section.

Firstly, the 2013 database was tested using the optimization approaches originally developed for the 2003 database. These optimizations were then adapted to the 2013 schema to assess how the corresponding 2013 workload would perform. Table 5.16 shows the comparison of the executions of this adaptation versus the optimization specifically oriented toward 2013. Additionally, Figures 5.1 and 5.2

provide a visual representation of the database performance across the different explained scenarios for the SQL and NoSQL approaches, respectively.

Schema	**Total Execution Time (ms)**	**Percentage**
Original	57,809.488	100.00%
SQL Opt. for 2003	53,056.482	91.78%
SQL Opt. for 2013	38,004.742	65.74%
NoSQL Opt. for 2003	2,176,775.454	3765.43%
NoSQL Opt. for 2013	36,847.382	63.74%

Table 5.16: 2013 Workload Execution Times including previous optimizations

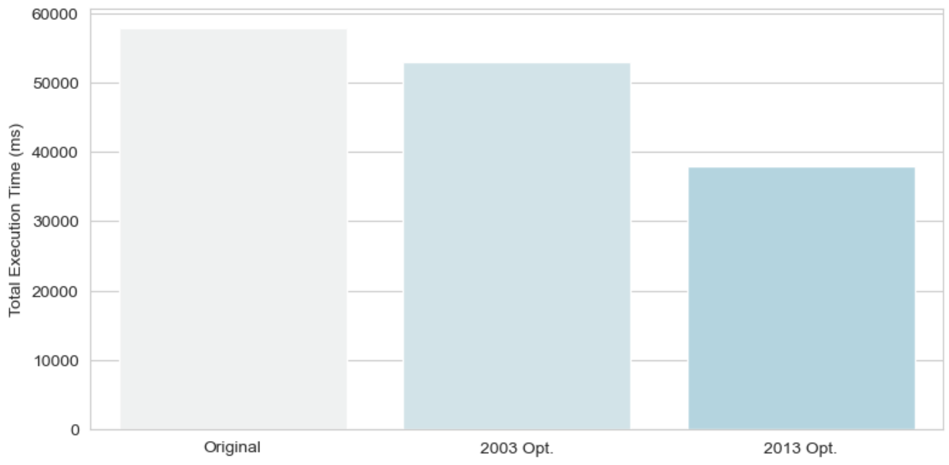


Figure 5.1: SQL Optimization Execution Times for 2013 Workload over different optimized schemas

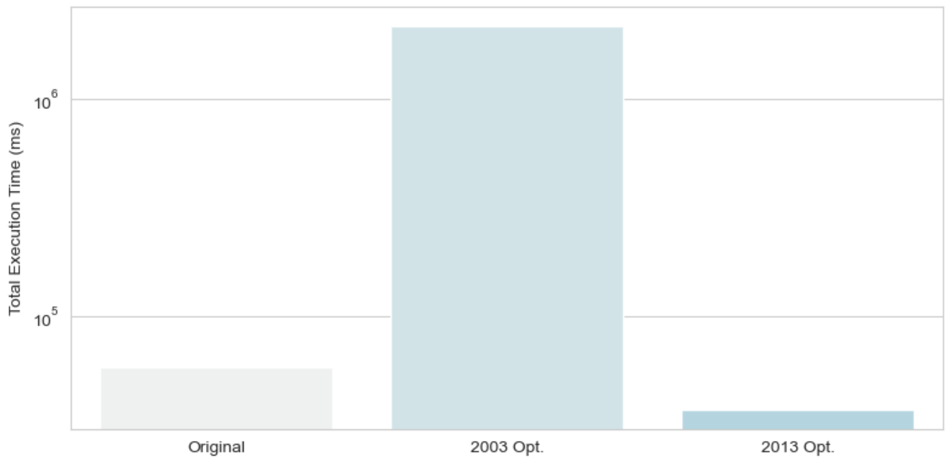


Figure 5.2: NoSQL Optimization Execution Times for 2013 Workload over different optimized schemas (Log Scale)

Finally, for the 2023 database—the latest one—it was possible to test both the 2003 and 2013 optimizations to see how the 2023 workload performs with these adaptations. Table 5.17 displays the comparison of the executions of these adaptations versus the optimizations specifically oriented toward 2023. Likewise, Figures 5.3 and 5.4 offer a visual representation of the database performance across the various explained scenarios for the SQL and NoSQL approaches, respectively.

Schema	**Total Execution Time (ms)**	**Percentage**
Original	92,908.09	100.00%
SQL Opt. for 2003	90,345.097	97.24%
SQL Opt. for 2013	97,143.995	104.56%
SQL Opt. for 2023	60,325.25	64.93%
NoSQL Opt. for 2003	3,619,897.419	3896.21%
NoSQL Opt. for 2013	3,625,606.833	3902.36%
NoSQL Opt. for 2023	58,619.93	63.09%

Table 5.17: 2023 Workload Execution Times including previous optimizations

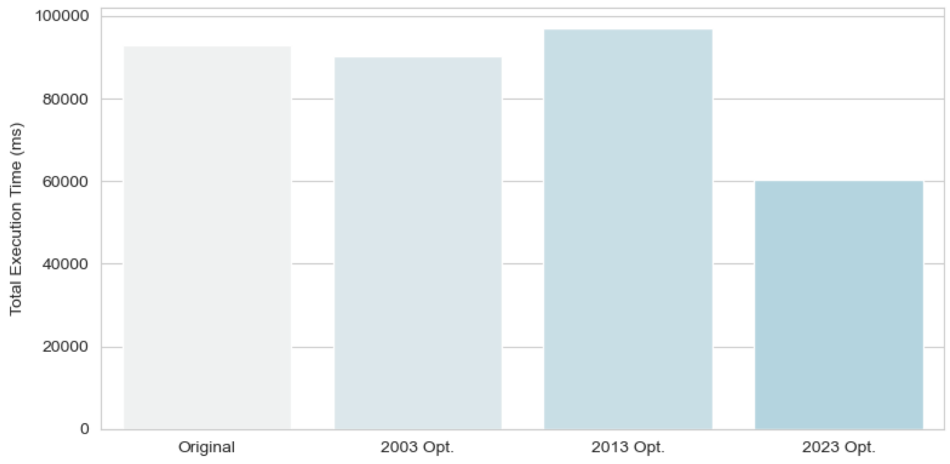


Figure 5.3: SQL Optimization Execution Times for 2023 Workload over different optimized schemas

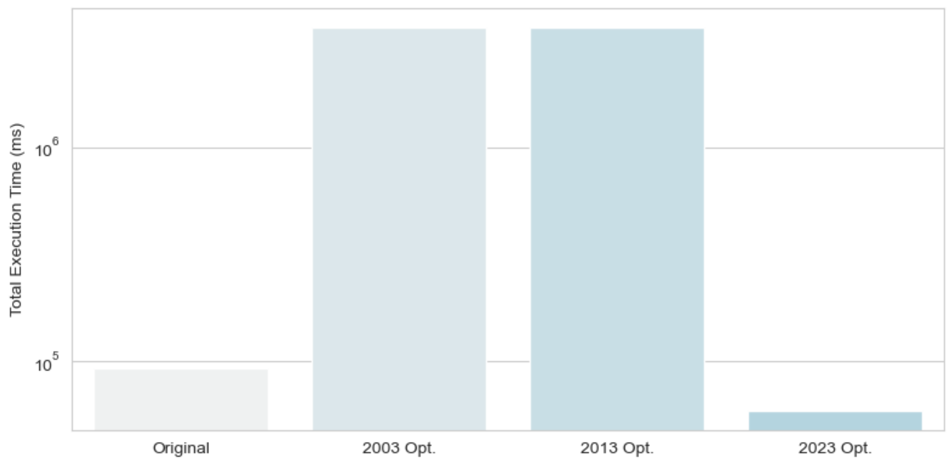


Figure 5.4: NoSQL Optimization Execution Times for 2023 Workload over different optimized schemas (Log Scale)

Experiments designed to evaluate the impact of earlier optimizations on later ones were crucial for determining if the evolving nature of workloads necessitates periodic reevaluation and potential redesign of schemas to sustain optimal performance.

From the results shown in Tables 5.16 and 5.17 the following insights can be drawn:

- SQL optimizations proved robust over the years, supporting the evolution of workloads effectively. For instance, the 2013 database optimized using the 2003 approach and the 2023 database using both 2003 and 2013 approaches demonstrated at least slight improvements or maintained performance equivalent to the original. However, these results suggest that maintaining past improvements alone is insufficient; conducting periodic optimizations based on current workloads is always beneficial. This is further illustrated by the bar plots in Figures 5.1 and 5.3, which show only minor variations in total execution times.

- Contrarily, NoSQL optimizations performed poorly considering workload evolutions, with execution times increasing by almost 4000% from the original, indicating a strong dependency on specific workloads. This suggests that a NoSQL approach might require more frequent reevaluations and adjustments compared to the SQL approach, which showed better performance stability over the years. The significant increase in total execution times when optimizations are not adequately aligned with the corresponding workload is clearly depicted in the bar plots of Figures 5.2 and 5.4.

5.3.2 Migration Cost Measurement

The subsequent step involved measuring the migration times for transitioning from the original schemas to their optimized SQL and NoSQL versions for each workload. These results are detailed in Table 5.18. The table itemizes the migration times by workload and summarizes the durations of various tasks involved in the migration process, including schema creation, table creation, data transferring, and index creation.

	2003 Database		2013 Database		2023 Database	
Query Type	SQL	NoSQL	SQL	NoSQL	SQL	NoSQL
Schema Cre.	0.68	0.32	0.28	0.37	0.42	0.31
Table Creat.	42.76	339.05	217.07	359.25	728.00	776.10
Data Transfer	1,686.84	116,916.42	27,195.67	953,404.74	36,966.82	1,049,585.13
Index Creat.	225.34	1,486.43	899.46	5,781.04	1,955.35	3,201.56
Total	1,955.63	118,742.21	28,312.48	959,545.40	39,650.59	1,053,563.09

Table 5.18: Migration times in milliseconds from the original schemas into their respective optimized versions

The analysis of migration costs as depicted in Table 5.18 reveals several critical insights. Firstly, there is a noticeable increase in migration times across the years, which can be primarily attributed to the growth in data size. Secondly, migration into SQL databases is substantially less costly compared to NoSQL. This difference largely stems from the complexities involved in converting data into JSON-based documents and the requirement for a complete database migration into NoSQL schemas, as opposed to selective table migrations in the SQL approach. Notably, the most time-consuming and therefore the most expensive phase is data transfer, which aligns with expectations.

5.3.2.1 Migration Cost Time Measure from Previous Optimizations

The results of analyzing the migration costs over different years while maintaining previously implemented optimizations are presented here. The measurements evaluate the time efficiencies gained or lost when prior optimizations from the years 2003 and 2013 are preserved and utilized in subsequent schema migrations for the years 2013 and 2023. The focus is particularly on how the preservation of past optimization efforts impacts the execution times of various database operations such as schema creation, table creation, data transfer, and index creation. Table 5.19 delineates the migration times in milliseconds for these operations across SQL and NoSQL databases for the specified years, illustrating the differential impact of reusing earlier optimizations in newer database environments. This data provides valuable insights into the practical implications of iterative optimization on migration costs and highlights the comparative performance across different database frameworks.

	2013 with 2003 Opt.		2023 with 2003 Opt.		2023 with 2013 Opt.	
Query Type	SQL	NoSQL	SQL	NoSQL	SQL	NoSQL
Schema Cre.	1.15	0.32	0.24	1.23	0.31	0.96
Table Creat.	84.86	297.32	372.91	2,595.52	344.62	582.74
Data Transfer	7,730.23	457,606.90	14,346.80	350,379.70	19,664.37	336,512.70
Index Creat.	551.17	741.87	12,991.22	6,952.68	14,572.38	2,432.89
Total	8,367.41	458,646.41	27,711.17	359,929.13	34,581.68	339,529.30

Table 5.19: Migration times in milliseconds from schemas with previous optimizations into their respective optimized versions

This section explores scenarios where optimizations were implemented in a given year and then reoptimization was needed at a later time. Analyzing such scenarios is crucial for understanding whether initial optimizations affect subsequent migration efforts. Specifically, the study examines if the migrations from a previously optimized NoSQL to a new version incur lower costs due to the non-requirement to migrate all tables. The findings from Table 5.19 indicate that, although the total migration times from a NoSQL schema are significantly reduced, they are still at least ten times higher than those for SQL migrations. This suggests that the number of tables migrated does not solely account for the differences in migration times. However, an encouraging observation is that migrations from previously optimized schemas are less costly than those from non-optimized ones, reinforcing the benefit of periodic schema optimizations.

5.3.3 Measurement of Gain from Optimization

This section highlights the principal outcomes of the project, focusing on the benefits derived from the two optimization approaches, SQL and NoSQL, tailored to the specific workload of each schema. As detailed earlier, the effectiveness of these optimizations is quantified through the break-even point—measured in days—which determines when the benefits of migration offset the associated costs. This provides a pragmatic approach to assess whether a migration is justifiable.

Table 5.20 presents an essential preliminary step in this analysis by illustrating the average daily improvements in execution times.

Subsequently, the complete financial analysis, which includes the break-even calculations, is detailed in Table 5.21. This two-step presentation not only clarifies the incremental benefits of optimizations but also underscores their financial viability.

Schema	Opt. Type	Total Ex. Time Base Case (ms)	Total Ex. Time Optimized (ms)	Avg. Gain Per Query (ms)	Avg. N. of Queries per Day	Avg. Daily Time Imp. (ms)
2003	SQL	107,186.22	91,877.06	1.53	1,560.00	2,388.23
2003	NoSQL	107,186.22	84,051.13	2.31	1,560.00	3,609.07
2013	SQL	57,809.49	38,004.74	1.98	43,629.00	86,406.13
2013	NoSQL	57,809.49	36,847.38	2.10	43,629.00	91,455.57
2023	SQL	92,908.09	60,325.25	3.26	268,079.00	873,477.52
2023	NoSQL	92,908.09	58,619.93	3.43	268,079.00	919,193.56

Table 5.20: Detailed Execution Time and Daily Time Improvement Post-Optimization

Schema	Opt. Type	Avg. Daily Time Imp. (ms)	Migration Cost (ms)	Break-even Period (Days)
2003	SQL	2,388.23	1,955.63	0.82
2003	NoSQL	3,609.07	118,742.21	32.90
2013	SQL	86,406.13	28,312.48	0.33
2013	NoSQL	91,455.57	959,545.40	10.49
2023	SQL	873,477.52	39,650.59	0.05
2023	NoSQL	919,193.56	1,053,563.09	1.15

Table 5.21: Break-even Analysis for Database Schema Optimizations

Additionally, the analytical approach previously described was applied to scenarios discussed in Subsection 4.7.2.1. This subsection elaborates on the need to evaluate scenarios in which existing optimizations were maintained and subsequently subjected to further optimization. As outlined earlier, the intermediate calculations for these scenarios are presented in Table 5.22. The `'Schema'` column in this table describes the schemas being optimized, and the workload basis for each optimization is noted in parentheses; for instance, `'2013 (2003 opt.)'` refers to the 2013 schema optimized based on the 2003 workload.

The final assessments of their financial viability are summarized in Table 5.23. Based on the results from Table 5.22, only schemas that demonstrated improvement were retained for further analysis. In Table 5.23, the schema column follows the same notation as earlier. Migrations are measured from these schemas to their optimized counterparts based on the respective workloads. For example, a `'2013 (2003 opt.)'` migration would move from that schema to one optimized in 2013 based on the 2013 workload.

Schema	Opt. Type	Total Ex. Time Base Case (ms)	Total Ex. Time Optimized (ms)	Avg. Gain Per Query (ms)	Avg. N. of Queries per Day	Avg. Daily Time Imp. (ms)
2013 (2003 opt.)	SQL	57,809.49	53,056.48	0.48	43,629.00	20,736.89
2013 (2003 opt.)	NoSQL	57,809.49	2,176,775.45	-211.90	43,629.00	-9,244,836.61
2023 (2003 opt.)	SQL	92,908.09	90,345.10	0.26	268,079.00	68,708.46
2023 (2003 opt.)	NoSQL	92,908.09	3,619,897.42	-352.70	268,079.00	-94,551,177.23
2023 (2013 opt.)	SQL	92,908.09	97,144.00	-0.42	268,079.00	-113,555.72
2023 (2013 opt.)	NoSQL	92,908.09	3,625,606.83	-353.27	268,079.00	-94,704,234.63

Table 5.22: Detailed Execution Time and Daily Time Improvement Post-Optimization Considering Previous Optimizations

Schema	Opt. Type	Avg. Daily Time Imp. (ms)	Migration Cost (ms)	Break-even Period (Days)
2013 (2003 opt.)	SQL	20,736.89	8,367.41	0.404
2023 (2003 opt.)	SQL	68,708.46	27,711.17	0.403

Table 5.23: Break-even Analysis for Database Schema Optimizations Considering Previous Optimizations

Analyzing the results from Table 5.21, it is evident that SQL consistently outperforms NoSQL, with a break-even period of less than one day in almost all instances, indicating a rapid return on investment. Although NoSQL also shows promising results, with recovery times never exceeding one month and even as short as a little over a day in the most recent 2023 database scenario, it still does not match the efficiency of SQL migrations.

Further evaluations were conducted on scenarios where previous optimizations were preserved and then subjected to additional optimization to assess their continued efficacy. The findings from Tables 5.22 and 5.23 reveal that, in most instances, these subsequent optimizations do not yield significant gains, and in some cases, result in losses. However, the SQL optimizations based on the 2003 workload for the years 2013 and 2023 show notable improvements, achieving break-even in less than a day, consistent with the patterns observed in Table 5.21.

The lack of gains from the 2013-based optimizations of the 2023 database may be attributed to the 2013 optimizations being highly tailored to specific workloads. Furthermore, as previously noted, NoSQL optimizations perform poorly over time with evolving workloads, underscoring the importance of timely reevaluations to prevent potential losses.

Chapter 6

Discussion

6.1 Workload Analysis

The analysis of workload data from 2003, 2013, and 2023 reveals important insights into the evolving nature of database queries and schema utilization over the years. The consistent trends observed across these periods include a low percentage of aggregation and nested queries, with at least 93% of all queries being selection types, primarily including WHERE clauses. This indicates a stable pattern in the fundamental use of databases.

Despite these consistent trends, there have been notable changes. The frequency of nested queries and the use of the DISTINCT clause have increased over time, while the application of the TOP clause and the proportion of projection queries have fluctuated. These variations suggest that while some query patterns remain steady, other aspects of database interactions evolve, necessitating periodic optimizations to address changing workloads and maintain efficiency.

Further, the query classification results highlight a dynamic evolution in schema usage, with an increasing number of tables over the years and changing query patterns. For instance, the number of distinct query patterns increased significantly between 2003 and 2013 but remained relatively stable into 2023. The consistent usage of the table `PhotoPrimary` across all periods reflects ongoing dependencies on specific database components, despite the broader changes in schema design and query diversity. These findings underscore the complexity of database management and the need for ongoing evaluation of database schemas to adapt to evolving requirements.

6.2 Database Optimizations

The experiments conducted on the 2023 database demonstrate significant improvements in database performance through both SQL and NoSQL optimization strategies. The SQL optimizations, particularly the application of indexing and inheritance strategies, substantially reduced execution times and costs. Notably, the execution time for the most costly query type, Q3, was reduced to just 38.56% of its original duration, illustrating the effectiveness of targeted optimization efforts.

In the realm of NoSQL, experiments underscored the critical role of JSON size and structure in performance outcomes. Smaller, well-structured JSON documents consistently performed better, emphasizing the need for efficient JSON management. The performance of B-tree indexes was found to surpass that of GIN indexes, leading to their selection for the majority of query types in the project due to their superior handling of non-JSON data operations.

Moreover, the experiments involving inheritance structures revealed that accessing data through child tables, especially in queries with limit clauses, significantly optimizes performance. This finding has informed strategies favoring child tables for future schema optimizations.

The embedding optimizations in NoSQL databases also yielded profound improvements, with mean execution times for certain queries reduced to as little as 10.67% of their original times. This demonstrates the potential of NoSQL features to enhance performance to levels that are competitive with traditional SQL approaches.

These results collectively affirm the benefits of strategic database optimizations in adapting to evolving workload demands and underscore the importance of continual assessment and application of both SQL and NoSQL technologies to maximize database efficiency and performance.

6.3 Optimization Evaluation

This section presents a summarized discussion of each subtask undertaken to evaluate the optimizations performed.

6.3.1 Workloads Simulation Execution Time Measurement

The findings underscore the necessity of continuous schema optimization to adapt to evolving database workloads effectively. While SQL optimizations have demonstrated sustained improvements with fewer adjustments, NoSQL systems, despite their flexibility, demand more frequent updates to maintain efficiency. These insights are pivotal for database administrators and developers in planning and implementing database strategies that are not only reactive to current needs but proactive in anticipating future requirements. The optimization practices should be viewed as an ongoing process rather than a one-time task, especially in environments experiencing rapid data growth and change.

6.3.2 Migration Cost Measurement

The analysis underscores significant differences in migration costs and times between SQL and NoSQL databases, with SQL migrations proving more cost-effective due to less comprehensive data transformation requirements. The increase in migration times over the years reflects the growing data volumes. Furthermore, while prior optimizations reduce the cost of subsequent migrations, the inherent advantages of SQL in terms of simpler data handling continue to make it a more economical choice for database migration projects. Repeated optimizations, while beneficial in reducing costs, do not bridge the fundamental cost disparity between SQL and NoSQL migrations.

6.3.3 Measurement of Gain from Optimization

This section addresses the crucial aspect of evaluating the economic viability and efficiency gains from database migrations. The break-even analysis offers a quantifiable basis to determine whether the investment in migration is justified based on anticipated improvements in daily operations.

The analysis substantiates significant benefits from SQL database optimizations with consistently short break-even periods, affirming their economic advantage. In contrast, NoSQL optimizations, while viable in certain contexts, generally require longer periods to justify the investments due to their dependency on specific workload characteristics. Reoptimizations, especially in the SQL domain, tend to maintain or enhance the economic benefits, validating the strategic value of periodic reviews and updates to database schemas. However, the efficacy of such reoptimizations is closely linked to the specific nature of initial optimizations and may not universally apply across different temporal and operational contexts. This highlights the necessity for strategic planning in database management, particularly in selecting between SQL and NoSQL technologies based on the specific operational needs and expected evolution of data workloads.

Chapter 7

Conclusions

7.1 Synthesis of Key Findings

This thesis offered an in-depth examination of the evolving nature of database workloads and their impact on the effectiveness of SQL and NoSQL database optimizations, as well as the economic viability of these optimizations through break-even analyses. The study revealed profound insights into strategic database management, emphasizing the adaptability and superior long-term efficiency of SQL systems compared to NoSQL systems.

7.2 Detailed Findings

This section provides an in-depth exploration of the key findings from the project, offering detailed insights into each aspect of the research.

7.2.1 Workload Evolution and Query Performance

The analysis of workload summaries and query classifications over different periods highlighted consistent trends and notable changes. The persistence of a high percentage of selection queries across all periods underscores the stable demand for data retrieval operations that include WHERE clauses. Conversely, the increasing use of DISTINCT and nested queries over time suggests evolving complexities in data handling and retrieval needs.

This evolution stresses the necessity for databases to adapt dynamically to changing query patterns and workload characteristics, highlighting the importance of continuous optimization to maintain operational efficiency and performance stability.

7.2.2 Optimization Impact Analysis

Empirical assessments reveal that SQL optimizations, notably through indexing and inheritance strategies, significantly enhance system efficiency by consistently reducing execution times. These straightforward modifications bolster SQL's suitability for environments with dynamic data schemas by improving performance predictably and stabilizing operational costs.

Conversely, NoSQL optimizations, while potentially beneficial, produce variable outcomes and often require more complex adjustments such as extensive JSON document restructuring and intricate indexing. This inconsistency underlines SQL's advantages in providing a more stable and cost-effective optimization path.

7.2.3 Gains from SQL and NoSQL Optimizations

SQL optimizations are shown to consistently deliver significant gains, characterized by notably short break-even periods that underscore their strategic benefits. This quick delivery of value makes SQL a particularly advantageous option when seeking rapid performance improvements and cost efficiency.

In contrast, gains from NoSQL optimizations generally manifest over longer periods and depend heavily on specific workload characteristics, posing challenges to organizations that prioritize immediate returns. Although NoSQL offers substantial benefits under certain conditions, its not as effective as SQL.

Reoptimizations in SQL not only preserve but can enhance the benefits from earlier optimizations, highlighting the importance of regular updates to adapt to changing operational demands. However, the success of such efforts varies and relies significantly on the initial optimization context. This reinforces the need for strategic planning in database management, advocating for informed decisions between SQL and NoSQL technologies based on detailed assessments of operational needs and expected workload evolutions.

7.3 Resolution of Research Questions

1. **Sustainability of Optimizations:** The results confirm that optimizations based solely on a specific time-point's workload do not guarantee sustained performance due to the inherently dynamic nature of database workloads. Periodic evaluations and adjustments are essential to align with evolving data

demands, as demonstrated by the fluctuating success rates of optimizations across various periods.

2. **Comparative Performance of SQL versus NoSQL:** While NoSQL systems are capable of achieving competitive performance levels, their need for more frequent updates makes SQL a more stable and cost-effective option. SQL's quick recovery times in break-even analyses underscore its superior return on investment and stability across diverse operational scenarios.

3. **Long-term Effectiveness of Optimization Approaches:** The empirical evidence strongly supports the superior long-term effectiveness of SQL optimizations. These optimizations offer a structured approach that caters efficiently to complex queries and adapts seamlessly to changing workload conditions without necessitating extensive modifications.

7.4 Evaluation of Hypotheses

1. The hypothesis that single-point optimizations do not ensure long-term effectiveness is corroborated by the findings, which highlight the need for continual optimization strategies to accommodate evolving workloads.

2. The hypothesis proposing comparable performance between NoSQL and SQL is partially supported. Although NoSQL can be effectively optimized, it generally does not match SQL in terms of ease of maintenance and efficiency.

3. The anticipation of SQL's superior performance over time is conclusively validated by the data, which illustrate SQL's robustness and efficiency in managing advanced query operations and adapting to dynamic data environments.

7.5 Concluding Remarks

This thesis underscores the critical importance of continuous optimization and strategic foresight in database management. For organizations heavily reliant on data analytics, opting for SQL could provide a more robust framework, offering economic viability and enhanced operational efficiency, particularly when considering overall lifecycle costs.

The inherent adaptability of SQL to fluctuating workloads without the need for significant overhauls presents a strong argument for its preference in environments with variable data demands. This adaptability ensures that SQL-based systems remain effective and economical over extended periods.

7.6 Future Work

Future research should focus on predictive models for workload evolution and explore the integration of artificial intelligence to automate database optimizations. Investigating the scalability of NoSQL in diverse environments and the potential of hybrid database systems could provide further insights into optimizing database operations in mixed data environments.

Additional studies on adaptive database systems capable of self-optimization could revolutionize database management, aligning technology more closely with future advancements in the field. Such explorations are essential as data management needs become increasingly complex and integral to business and technological advancements.

Appendix A

Query Classification Full Results

The comprehensive details of the query classification are available in the following spreadsheet [1]. It includes the detailed patterns of each query type defined per each workload, which are not included in this report due to the extensive length of some patterns.

[1]https://docs.google.com/spreadsheets/d/1t15v3QaXx_324JeXGOYL_NwA5jfvTqVqSOeoMmmoArw/edit?usp=sharing

Appendix B

Workload Execution Simulation Pseudocode

Algorithm 1: Simulate Workload

```
  Data: schema, num_sims, type_exp
  Result: log_df
1 parameters ← LoadVariablesSim(schema)
2 queries, weights, params_query ← ReadQueryTemplates(schema,
    type_exp)
3 log_df ← Simulate(num_sims, parameters, queries, weights,
    params_query)
4 return log_df
```

Algorithm 2: Load Variables for Simulation

```
  Data: schema
  Result: params
1 Adjust schema for specific cases
2 Initialize params as an empty dictionary
3 if schema matches specific cases then
4     Query the database to populate params with necessary lists and
        statistics
5 end
6 return params
```

Algorithm 3: Read Query Templates

```
   Data: schema, type_exp
   Result: queries, weights, params_query_temp
 1 Initialize empty lists: queries, weights, params_query_temp
 2 Open the schema queries CSV file
 3 for each row in CSV file do
 4     Append weight to weights
 5     Append query to queries
 6     Append parameters to params_query_temp
 7 end
 8 return queries, weights, params_query_temp
```

Algorithm 4: Simulate Queries

```
   Data: num_sims, params, queries, weights, params_query_temp
   Result: log_df
 1 Initialize log_entries as an empty list
 2 n_queries $\leftarrow$ length(queries)
 3 for num_exp from 0 to num_sims-1 do
 4     chosen_index $\leftarrow$ random choice from range(n_queries) with weights
 5     Initialize query_params as an empty dictionary
 6     for param in params_query_temp[chosen_index] do
 7         if param matches specific cases then
 8             Populate query_params with random or sampled values from
                 params
 9         end
10     end
11     pre_query $\leftarrow$ format(queries[chosen_index], query_params)
12     final_query $\leftarrow$ "EXPLAIN (ANALYZE, FORMAT JSON) " + pre_query
13     Execute final_query and fetch result
14     Process result and add necessary details to log_entry
15     Append log_entry to log_entries
16 end
17 log_df $\leftarrow$ convert log_entries to DataFrame
18 return log_df
```

Appendix C

Resultant Optimized Schemas

This appendix outlines the optimizations applied to each schema within both SQL and NoSQL frameworks. It will focus primarily on selected optimization techniques, providing detailed information and, where applicable, examples to illustrate these techniques effectively. Although the NoSQL optimizations mirror those of the SQL in terms of indexes and inheritance, they are adapted to fit the unique characteristics of the NoSQL approach.

C.1 2003 Optimization

Common to SQL and NoSQL:

- **Indexing:**
  - **Galaxy:** B-tree index on `i`.
  - **SpecObjAll:** B-tree index on `BestObjID`.
  - **Photoz:** B-tree index on `z`.
- **Inheritance:**
  - **PhotoPrimary:** Becomes a child table of `PhotoPrimaryMain` with main query columns. Maintains `objid` as PK.

NoSQL-Specific:

- Adapted tables to a key-value document store structure, maintaining SQL-like indexing and inheritance.
- **Embedding + Inheritance:**

- **Galaxy:** Embedded data from `Photoz` and `SpecObj`. The **inheritance** approach was also implemented for this table, organizing critical columns into the primary `data` column and including all additional embedded fields. Non-essential data are stored in `dataExtra` in the child table.. And it was done by using the query below:

```
INSERT INTO db_2003_ns.galaxy
(objid, data, dataExtra)
SELECT p.objid,
    jsonb_build_object('ra', p.ra, 'dec',
    p.dec, ...) AS data,
    (to_jsonb(p.*) - 'objid' - 'ra' -
    'dec' - ...) AS dataExtra
FROM db_2003.galaxy AS p
LEFT OUTER JOIN db_2003.specobj AS s
ON s.bestobjid = p.objid
LEFT OUTER JOIN db_2003.photoz AS pz
ON pz.objid = p.objid;
```

C.2 2013 Optimization

Common to SQL and NoSQL:

- **Inheritance:**
  - **PhotoPrimary:** Changed to a child table of `PhotoPrimaryMain`, maintaining `objid` as PK.
- **Indexing and Inheritance:**
  - **Star:** Transitioned to a child table of `StarMain`, which was tailored to include the primary columns utilized for querying.
    - `objid` was maintained as the primary key (PK) for both the parent and child tables.
    - The following B-tree indexes were implemented:
      - Indexes were established over `ra`, `dec`, `clean`, and `mode` to optimize common queries.
      - A comprehensive index was also created covering `ra` and `dec`, specifically tailored for a complex WHERE clause used extensively across a suite of queries. This clause includes conditions such as `(flags & 8) = 0 AND clean = 1 AND mode = 1` along with multiple bitwise operations on `flags_r`.

NoSQL-Specific:

- Tables adapted to key-value document store with consistent indexing.
- **Embedding + Inheritance:**
  - **Galaxy:** Incorporates selected data from the `Photoz`, `PhotozRF`, and `PhotoObjAll` tables through embedding. The inheritance model was also applied as in the previous schema. An example query illustrating this structure is provided below:

```
INSERT INTO db_2013_ns.Galaxy
(objid, data, dataExtra)
SELECT
g.objid,
CASE
    WHEN pz.z IS NULL OR pz.zerr IS NULL OR
         pzr.z IS NULL OR pzr.zerr IS NULL OR
         poa.u IS NULL OR poa.err_u IS NULL OR
         poa.g IS NULL OR poa.err_g IS NULL OR
         poa.r IS NULL OR poa.err_r IS NULL OR
         poa.i IS NULL OR poa.err_i IS NULL OR
         poa.z IS NULL OR poa.err_z IS NULL
    THEN NULL
    ELSE jsonb_build_object(
        'flags', g.flags,
        'pz_z', pz.z, 'pz_zerr', pz.zerr,
        'pzr_z', pzr.z, 'pzr_zerr', pzr.zerr,
        'poa_u', poa.u, 'poa_err_u', poa.err_u,
        'poa_g', poa.g, 'poa_err_g', poa.err_g,
        'poa_r', poa.r, 'poa_err_r', poa.err_r,
        'poa_i', poa.i, 'poa_err_i', poa.err_i,
        'poa_z', poa.z, 'poa_err_z', poa.err_z
    )
END AS data,
    (to_jsonb(g.*) - 'flags') AS dataExtra
FROM db_2013.galaxy AS g
LEFT JOIN db_2013.photoz AS pz
ON g.objid = pz.objid
LEFT JOIN db_2013.photozrf AS pzr
ON g.objid = pzr.objid
LEFT JOIN db_2013.photoobjall AS poa
ON g.objid = poa.objid;
```

C.3 2023 Optimization

Common to SQL and NoSQL:

- **Indexing:**

  - **Galaxy:** Indexes on `dered_r` and `z`.

- **Inheritance:**
  - **PhotoPrimary:** Child table of `PhotoPrimaryMain` with maintained `objid` as PK.

- **Inheritance + Indexing:**
  - **PhotoObjAll:** Transitioned to a child table of `PhotoObjRD`, specifically designed with main query columns.
    * `objid` is maintained as the primary key (PK) in both parent and child tables.
    * Coverage B-tree indexes were retained from the original table and new indexes were specifically created for `ra` and `dec`.
  - **SpecObj:** Converted into a child table of `SpecObjRD`, containing essential columns for querying.
    * `specObjID` continues as the PK for both tables.
    * B-tree indexes applied over `ra` and `dec`.
  - **SpecObjAll:** Reorganized as a child table of `SpecObjMain`, which includes main columns tailored for querying.
    * `specObjID` is preserved as the PK in both tables.
    * B-tree indexes were developed over `BestObjID` and over `ra` and `dec`.

NoSQL-Specific:

- Adapted tables to document store structure, consistent with SQL indexing.

- **Embedding:**
  - **GalSpecIndx + GalSpecExtra:** As previously described, `GalSpecExtra` is designated as a child table of `GalSpecIndx`. This hierarchical relationship enhances data organization and query efficiency.
    * `specobjid` is retained as the primary key (PK) in both tables to ensure consistent data linkage and integrity.

- **Embedding + Inheritance + Indexing:**

- **Galaxy:** The table incorporates selected data from `Photoz` and `SpecObj` through embedding. In parallel, the inheritance model as in previous schemas.
  - `objid` is maintained as the primary key (PK) for both the parent and child tables, ensuring relational integrity.
  - Indexes were established on `derder_r` and `pzz1` (which corresponds to `z` from `Photoz`).

Bibliography

Duckdb: An embeddable sql olap database management system. `https://duckdb.org/`. Accessed: 2024-05-11.

Accessing data through the sdss skyserver. `https://www.sdss.org/dr18/data_access/`, a.

About sloan digital sky survey. `https://www.sdss.org/`, b.

A. Almeida et al. Data release 18 of the sloan digital sky survey: First release from sdss-v. *Astrophysical Journal Supplement Series*, 259:35, 2023. doi: 10.3847/1538-4365/ac4414. URL `https://ui.adsabs.harvard.edu/abs/2023ApJS..267...44A/abstract`.

R. Bayer and E. M. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1(3):173–189, 1972.

P. Bonnet, J. Gehrke, and P. Seshadri. Predicting the performance of queries. In *Proceedings of the 2001 ACM SIGMOD international conference on Management of data*, pages 271–282. ACM, 2001.

R. Cattell. Scalable sql and nosql data stores. *ACM SIGMOD Record*, 39(4):12–27, 2011.

C. Curino et al. Automating the database schema evolution process. *SpringerLink*, 2013.

R. Elmasri and S. B. Navathe. *Fundamentals of Database Systems*. Pearson/Addison Wesley, 2006.

A. Lih. Wikipedia as a database: Querying wikipedia with ssql. *Communications of the ACM*, 52(3):78–82, 2009.

Z. Liu, B. He, H.-I. Hsiao, and Y. Chen. Efficient and scalable data evolution with column oriented databases. *EDBT*, 2011.

R. Oblak et al. *PostgreSQL: Up and Running: A Practical Guide to the Advanced Open Source Database*. O'Reilly Media, Inc., 2017.

J. Pokorny. Nosql databases: a step to database scalability in web environment. *International Journal of Web Information Systems*, 9(1):69–82, 2013.

PostgreSQL Documentation. Using explain. `https://www.postgresql.org/docs/current/sql-explain.html`, 2023. Accessed: 2024-05-11.

R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill, 2003.

S. Sakr, A. Liu, et al. A survey of large scale data management approaches in cloud environments. *IEEE Communications Surveys & Tutorials*, 13(3):311–336, 2013.

M. Stonebraker. Sql databases v. nosql databases. *Communications of the ACM*, 53 (4):10–11, 2010.

M. Stonebraker et al. The end of an architectural era: (it's time for a complete rewrite). In *Proceedings of the 33rd international conference on Very large data bases*, pages 1150–1160. VLDB Endowment, 2007.

A. Thor et al. Why it is time for yet another schema evolution benchmark. *SpringerLink*, 2017.

M. Thorat et al. Language extensions for the automation of database schema evolution. *SpringerLink*, 2018.

W. Vogels. Eventually consistent. *Communications of the ACM*, 52(1):40–44, 2009.