Universidade da Beira Interior

Department of Computer Science

FACULDADE ENGENHARIA

Departamento de Informática

Performance comparison between Postgres Neo4J and TileDB

Developed by:

João Nogueira

Advisor:

Professor Doutor: Rui Cardoso

July 17, 2023

Acknowledgements

The realization of this project relied on important emotional support and incentives, without which this project would never have materialized. That said, I will forever be grateful for the support network that accompanied me during the realization of this project and my academic career.

To all the teachers I came across during my education at the basic school Fernando Pessoa and the secondary school of Santa Maria da Feira, as I feel they played a crucial role in discovering my own path.

To my course mates, Pedro Diogo, Miguel Baptista, Rodrigo Gomes, Miguel Matos and Luís Santos, who made me see that it is never too late to finish a course and for all the companionship and help they showed me after I resumed my studies.

To my friends Tiago Antunes, Guilherme Henriques, Gil Marques and Daniel Rocha for their friendship, for the morals they conveyed to me and for the discipline they instilled in me.

To the professor and advisor Rui Cardoso, for the willingness, patience and wisdom given.

To my aunt Rosário Azevedo for always being a role model, for the way she motivated me throughout my academic career and for always being around even though she was far away.

To my girlfriend Maria Caseiro for her love and unconditional support in the most difficult moments, for always being available when I needed, without her the realization of this project would not be possible.

Finally, a special thanks to my parents, sister and grandparents. Without them none of this would be possible. Thank you all for your patience, unconditional support, friendship and help in overcoming obstacles that have arisen along my journey. Thank you for helping me to have the courage to be a better person. I dedicate this project to them.

Contents

List of Figures

List of Tables

Acronyms

ACID	Atomicity, Consistency, Isolation, Durability
API	Application Programming Interface
APOC	Awesome Procedures on Cypher
CPU	Central Processing Unit
CSV	Comma-Separated Values
DBMS	Database Management System
EPOS	European Plate Observing System
GIS	Geographic Information System
IoT	Internet of Things
JSON	JavaScript Object Notation
NumPy	Numerical Python
REST	REpresentational State Transfer
SQL	Structured Query Language
SSL	Secure Sockets Layer
TLS	Transport Layer Security

Chapter 1

Introduction

The goal of this project is to develop a set of methods and metrics to evaluate the performance of different database systems, using as a starting point a GNSS (Global Navigation Satellite System) production database currently in use, implemented in PostgreSQL. The proposal is to then develop and implement alternative solutions using Neo4J and TileDB systems. A detailed comparison of the metrics evaluated between the different solutions will be performed in order to determine which one best suits the specific needs of the database.

1.1 Problem Contextualization

The increasing amount of GNSS data generated daily requires efficient and scalable database systems for storing, querying and analyzing this data. The current system, based on PostgreSQL, has been used successfully to date, but the need for performance improvements and scalability capabilities has become evident. This project is part of EPOS-IP WP10, a thematic core service of the European Plate Observing System that aims to promote integration of data, data products, and facilities from distributed research infrastructures for solid Earth science in Europe EPOS-IP. European Plate Observing System (EPOS) is composed of a variety of Earth Science stakeholders that are together working towards integrating a set of diverse European Earth Science National Research Infrastructures into one single interoperable platform. The programme will develop implementation plans and use new e-science opportunities to monitor and understand the dynamic and complex solid Earth system. Therefore, it becomes crucial to explore viable alternatives to enhance the existing database system that is the current production solution due to performance problems when querying from specific critical tables.

1.2 Motivation

In today's data-driven world, efficient and scalable database solutions are key to managing and analyzing large and complex data sets. As the production database has accumulated vast amounts of data, it was observed a significant increase in the time required for query execution on our production database developed with PostgreSQL. To address this challenge and explore possible alternatives, we propose to investigate the performance of two database solutions, Neo4J and TileDB, to determine the most suitable solution for our specific dataset and query workload. While PostgreSQL has been a reliable and robust choice for our current database needs, we believe that exploring alternative database solutions may offer additional benefits. However, there is a lack of comprehensive research comparing these solutions to our specific dataset and query patterns.

1.2.1 Expected Contributions

By carrying out this project, we expect the following contributions:

- A detailed evaluation of the performance and scalability of Neo4J and TileDB compared to PostgreSQL for our specific dataset and query workload.
- Insights into the strengths and weaknesses of each database solution in handling different types of queries and data structures.
- Recommendations for selecting the most suitable database solution based on the specific characteristics of our data and query requirements.

1.2.2 Scope Limitations

It is important to note that this project will focus exclusively on comparing Neo4J and TileDB with PostgreSQL for our specific dataset and query patterns. While the results are valuable for guiding database selection decisions in this specific project, they may not be directly applicable to other datasets or scenarios. Furthermore, due to resource constraints, this study will focus primarily on query workloads with a higher emphasis on reads, even though it is limited by available resources.

1.3 Objectives

The main objective of this project is to evaluate and compare the performance of different database systems, aiming to find a more suitable solution to handle the current GNSS database. The specific objectives are:

- Develop a set of methods and metrics to evaluate the performance of database systems.
- Perform a detailed analysis of the requirements and characteristics of the existing GNSS database.
- Implement an alternative solution using the Neo4J database system.
- Implement an alternative solution using the TileDB database system.
- Perform performance testing and compare the metrics evaluated between the PostgreSQL, Neo4J and TileDB solutions.
- Identify the most suitable solution that best meets the performance, scalability, and query requirements of the GNSS database.

1.4 Proposed methodology

1.4.1 Analysis of the Existing GNSS Database

Understand the structure and schema of the current GNSS database. Identify the key requirements and query patterns of the database.

1.4.2 Neo4J Solution Implementation

Model and implement the GNSS database in Neo4J, taking into account the features and requirements identified in the initial analysis. Perform performance testing and collect specific metrics for this solution.

1.4.3 TileDB Solution Implementation

Model and implement the GNSS database in TileDB, considering the particularities and requirements raised previously. Perform performance tests and collect specific metrics for this solution.

1.4.4 Defining Queries to Test Execution Times

Identify and define a set of relevant queries to evaluate the performance of database systems, registering how they perform in each individual technology.

1.4.5 Evaluation and Comparison of Solutions Query Performance

Analyze and compare the results collected between PostgreSQL, Neo4J and TileDB solutions. Identify the most appropriate solution based on the metrics evaluated and the GNSS database requirements.

1.5 Expected Results

- Established set of metrics and methods for evaluating the performance of database systems.
- Detailed analysis of the requirements and characteristics of the existing GNSS database.
- Implementation of an alternative solution in Neo4J and collection of performance metrics.
- Implementation of an alternative solution in TileDB and collection of performance metrics.
- Comparison and evaluation of the collected metrics between PostgreSQL, Neo4J and TileDB solutions.
- Identification of the most suitable solution for the GNSS database, taking into account the requirements and metrics evaluated.

1.6 Document Organization

Firstly, the LaTeX document preparation system was used to prepare the report, using the OverLeaf online editor. In order to represent the work carried out, this document is structured as follows:

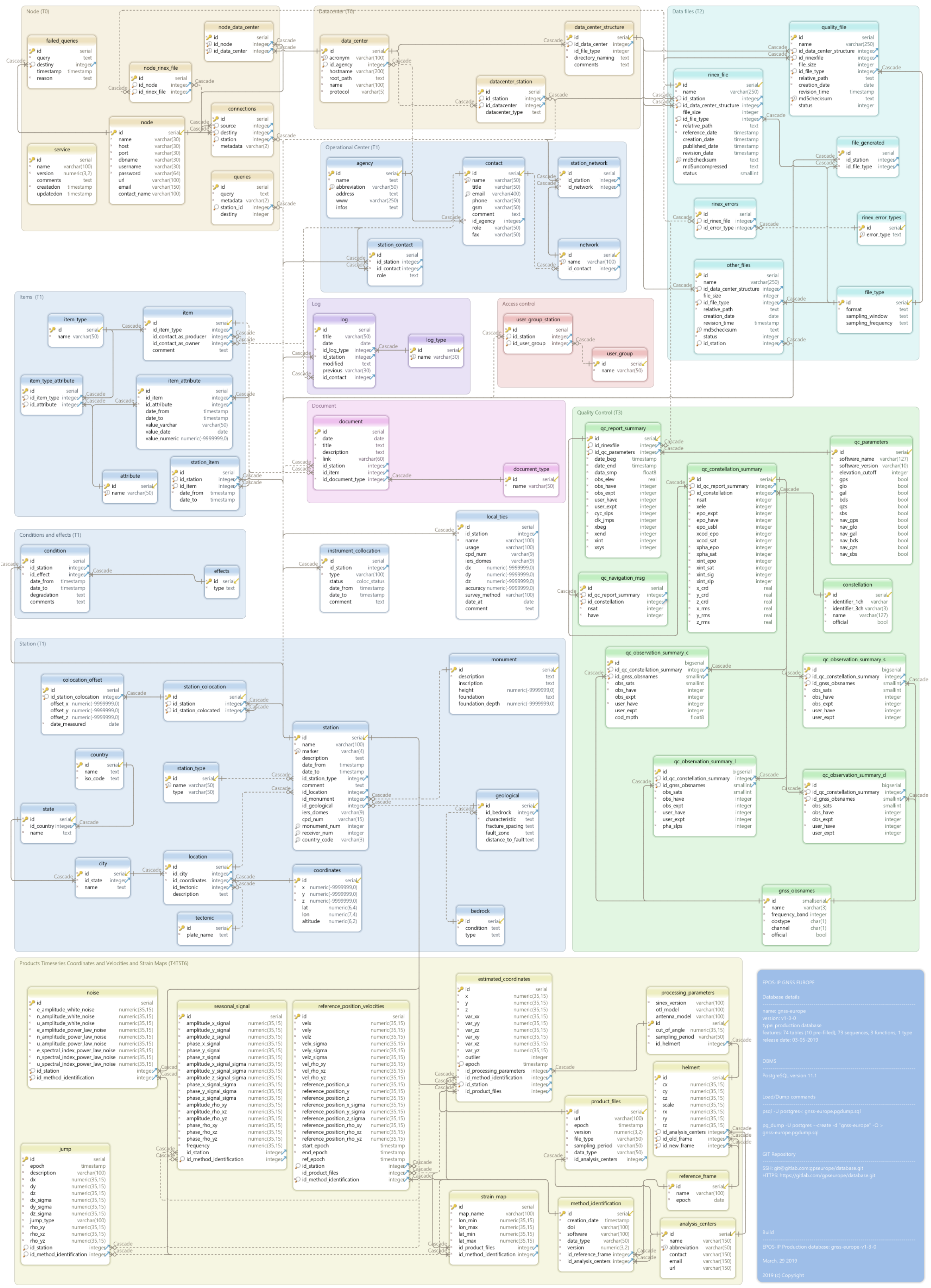
1. First chapter – **Introduction** – Presents the project, giving the framework of the same, the motivation for its choice, its objectives and the respective organization of the document.
2. Second Chapter – **State-of-the-Art** – Describes some similar solutions within the same field of this project, as well as some brief introduction of the technologies utilized to make this project possible and what are the fields that each technology thrives the most.

3. Third Chapter – **Technologies and Tools Used** – This chapter describes in detail the technologies used, as well as the tools associated with each technology, used to develop each of the solutions.

4. Fourth Chapter – **Solution Details** – Exposes the most important steps concerning the implementation of the solutions used in the performance metrics comparisons.

5. Fifth Chapter – **Tests and Results** – This chapter shows all the tests developed based on the critical tables, as well as the analysis of the results obtained and their comparison.

The following figure represents the structure and organization of the database that is currently in production.



EPOS-IP GNSS EUROPE

Database details

name: gnss-europe
version: v1-3-0
type: production database
features: 74 tables (10 pre-filled), 73 sequences, 3 functions, 1 type
release date: 03-05-2019

DBMS

PostgreSQL version 11.1

Load/Dump commands

```
psql -U postgres< gnss-europe.pgdump.sql

pg_dump -U postgres --create -d "gnss-europe" -O >
gnss-europe.pgdump.sql
```

GIT Repository

SSH: git@gitlab.com:gpseurope/database.git
HTTPS: https://gitlab.com/gpseurope/database.git

Build

EPOS-IP Production database: gnss-europe-v1-3-0

March, 29 2019

2019 (c) Copyright

Figure 1.1: Production database schema

Chapter

2

Estado da Arte

2.1 Introduction

The chapter of the state of the art of this report aims to explore the current state of the technologies used in this project, those being PostgreSQL[1], Neo4J[2] and TileDB[3] and make a brief introduction of each technology, analyzing their practical applications, their research and how they stack against each other, specially in the theme of this project which analyzes query performance metrics in storage and access of large data sets and a simple example of how each language works in the different technologies (for demonstration purposes, it'll be shown how to create a simple table in each language). For example, Google chooses not to use relational databases exclusively because it needs more scalable, flexible and efficient solutions to handle the huge amount of data generated by its services and applications. It takes a diversified approach to data management, combining various technologies and solutions according to the specific needs of each application. With this project, we hope to determine which solution fits the data better or even if a hybrid solution approach would have some advantages compared to the original implementation.

2.2 PostgresSQL

PostgreSQL was initially released in 1989 and became one of the most popular database management systems (DBMS) still to date. PostgreSQL is open-source, powerful, has high performance and is highly reliable when it comes to deploying and developing scalable applications, due to having advanced features such as ACID compliance which is later discussed in the next chapter. This technology is also designed to run on all major operating systems, those being Windows, Linux and macOS. PostgreSQL also has a solid and robust community support which contributes to the development of various extensions and third-party tools.

Some of the most common applications of PostgreSQL are as follow:

Web Applications: Used as the backend database for web applications due to the capacity of handling high volume of concurrent requests as well as great support for popular frameworks like Django, Node.js and Ruby on Rails.[4]

GIS Applications: Features built-in support for geographic data, therefore it's a fantastic choice for Geographic Information System (GIS) related applications.[5]

Financial Applications: Due to postgreSQL high reliability, security and data integrity it makes an attractive choice for financial applications.[6]

Data warehousing: PostgreSQL is a suitable option for data warehousing for large volumes of data due to its capacity of efficiently performing complex queries and transactions. [7]

Here are some of the companies that reportedly use PostgreSQL in their tech stack:

- Instagram
- Imdb
- Uber
- Spotify
- Reddit
- Netflix

Being a relational database management system, PostgresSQL supports the Structured Query Language (SQL) language which is the standard language used to interact with Database Management System (DBMS). Here's an example of the SQL language:

```
CREATE TABLE students (
    id INT PRIMARY KEY,
    name VARCHAR(255),
    email VARCHAR(255),
    phone VARCHAR(20)
);

CREATE TABLE courses (
```

```sql
    id INT PRIMARY KEY,
    name VARCHAR(255),
    instructor VARCHAR(255)
);

CREATE TABLE enrollment (
    id INT PRIMARY KEY,
    student_id INT,
    course_id INT,
    grade VARCHAR(2),
    FOREIGN KEY (student_id) REFERENCES students(id),
    FOREIGN KEY (course_id) REFERENCES courses(id)
);

INSERT INTO students (id, name, email, phone)
VALUES (1, 'John Doe', 'johndoe@example.com', '555-1234');

INSERT INTO courses (id, name, instructor)
VALUES (1, 'Introduction to SQL', 'Jane Smith');

INSERT INTO enrollment (id, student_id, course_id, grade)
VALUES (1, 1, 1, 'A');
```

Code Snippet 2.1: Example of SQL language. Table creation and data insertion.

This SQL example creates three tables: one for students, one for courses, and one for enrollment. The enrollment table has foreign keys referencing the student and course tables to enforce referential integrity. Data is then inserted into the tables using the INSERT INTO statement.

2.3 Neo4J

In a different fashion Neo4J is not a relational database management system, but a graph database management system that is designed to store, access, manage and query data in the form of relationships, properties and nodes. It allows users to represent complicated, mutual dependent data in a seamless and intuitive way. To query and manipulate the data stored in the graph, Neo4J uses the query language called Cypher. It also provides numerous Application Programming Interface (API) and integrations for developers to facilitate the interaction with the database.

Some of the most common applications of Neo4J are as follow:

Social Networking: Used for building social networking applications that involve modeling relationships between users, activities, and groups.[8]

Bioinformatics: One of the primary choices to store and analyze complex biological data, such as disease associations, protein-protein interactions and gene pathways.[9]

Knowledge Graphs: Used for building knowledge graphs that allows companies/organizations to manage and model complex data regarding relationships between data points. Ex: people, concepts, products.[10]

Fraud Detection: Neo4J is used to detect fraudulent activities by modeling relationships between entities, allowing the detection of behavioral patterns that may indicate fraudulent activity.[11]

Recommendation Engines: Neo4J's ability to model complex relationships between data points makes it an ideal choice for building recommendation engines that can provide personalized recommendations to users. [12]

Here are some of the companies that reportedly use Neo4J in their tech stack:

- Airbus
- Levi's
- Ebay
- ADEO
- die Bayerische
- Comcast

Neo4J supports the Cypher language, which is used to execute queries, managing and accessing data. Here's an example of the Cypher language:

```
CREATE (student:Student {id: 1, name: 'John Doe', email: 'johndoe@example.com',
    phone: '555-1234'});
CREATE (course:Course {id: 1, name: 'Introduction to SQL', instructor: 'Jane
    Smith'});
CREATE (student)-[:ENROLLED_IN {grade: 'A'}]->(course);
```

Code Snippet 2.2: Neo4J node creation

This Cypher example creates three nodes: one for the student, one for the course, and one for the enrollment relationship between the two. The ENROLLED_IN relationship has a grade property associated with it.

2.4 TileDB

TileDB is a potent open-source data management system made to manipulate and store very large and complex data sets efficiently. It can handle a wide variety of data types due to having a unified compute layer and storage.

TileDB architecture consists of a multi-dimensional array format that's capable of efficient data manipulation and access with the aid of an intuitive API. In addition, TileDB is built to support parallelism and scalability, making it a great choice for both single-node and distributed computing environments.

TileDB has been increasingly more popular in the scientific community since it has the ability to process very large and heterogeneous data sets, making a great choice for data-intensive applications in areas such as machine learning, geospatial analysis and genomics.

TileDB has a wide range of applications in different industries. Here are some of the most common application areas of TileDB:

Data Science and Machine Learning: TileDB is utilized as a storage solution for large-scale data analytics and machine learning workflows. It provides efficient storage and access to structured and unstructured data, enabling rapid data ingestion, processing and model training. [13]

Genomics and Bioinformatics: TileDB is widely used in genomics and bioinformatics to store and analyze large-scale genomics data sets. It offers efficient storage and retrieval of genomics data, allowing researchers to perform complex queries and analyses on genetic variants, DNA sequences and other biological data.[14]

Geospatial Data Management: TileDB is used in geospatial applications to manage and analyze large-scale geospatial data sets. It enables efficient storage, indexing and querying of spatial data such as satellite imagery, GIS data and location-based information.[15]

Internet of Things (IoT): TileDB is used in Internet of Things (IoT) applications to store and analyze time-series sensor data generated by connected devices. It offers efficient storage, compression and indexing of time-series data, enabling real-time analyses and anomaly detection.[16]

Image and Video Processing: TileDB is used in image and video processing applications to store and analyze large-scale image and video data sets. It enables operations such as image segmentation, object detection and video analysis, while providing fast access to image and video data.[17]

Here are some of the companies that reportedly use TileDB in their stack:

- Plantir Technologies
- Regeneron Pharmaceuticals
- Climate Corporation
- Astra Zeneca
- Space Telescope Science Institute (STScI)

There are numerous APIs that can be used to work with TileDB arrays, here are a few examples. The TileDB Python API allows developers to interact with TileDB using the Python programming language. It provides a high-level interface to create, read, write, and query data in TileDB arrays. The Python API is widely used for integration with data analysis libraries such as pandas and Numerical Python (NumPy). The TileDB C/C++ API is the native, low-level API for interacting with TileDB. It offers high performance and direct access to TileDB resources. This API is suitable for developers who want to build C/C++ applications and want maximum control over TileDB. The TileDB Java API allows developers to interact with TileDB using the Java programming language. It provides similar functionality to the Python API and is useful for developers who prefer to work with Java in their applications. The TileDB Go API allows developers to interact with TileDB using the Go programming language. It offers similar functionalities to the Python and Java APIs, allowing developers to work with TileDB in their applications written in Go.

```
// Create the TileDB context and array schema
TileDB::Context ctx;
TileDB::ArraySchema schema(ctx, TileDB_DENSE);
schema.set_domain(TileDB::Domain(ctx, {{1, 1}}, {{1, 1}}));
schema.add_attribute(TileDB::Attribute(ctx, "student_id", TileDB_INT32));
schema.add_attribute(TileDB::Attribute(ctx, "course_id", TileDB_INT32));
schema.add_attribute(TileDB::Attribute(ctx, "grade", TileDB_CHAR));

// Create the TileDB array and write data to it
TileDB::Array array(ctx, "enrollment", TileDB_WRITE);
TileDB::Query query(ctx, array, TileDB_WRITE);
std::vector<int> student_ids = {1};
std::vector<int> course_ids = {1};
std::vector<std::string> grades = {"A"};
query.set_buffer("student_id", student_ids);
query.set_buffer("course_id", course_ids);
query.set_buffer("grade", grades);
query.submit();
```

Code Snippet 2.3: Example of TileDB array creation and data insertion

This TileDB array example creates an array schema with three attributes: student_id, course_id, and grade. Data is written to the array using the TileDB Query object, which allows us to set buffers for each attribute and submit the data for writing.

The table 2.1 describes the advantages and disadvantages of each technology, as well as their different characteristics:

Table 2.1: Comparison of PostgreSQL, Neo4J, and TileDB

Aspect	**PostgreSQL**	**Neo4J**	**TileDB**
Data Model	Relational (SQL)	Graph	Multi-dimensional Array
Use Cases	Various applications	Highly connected data	Scientific analysis
Query Language	SQL	Cypher	APIs (Python, C++, R)
Scalability	Good	Moderate	Good
Performance	Optimized for relational queries	Optimized for graph traversals	Optimized for multi-dimensional data
Ecosystem	Mature with extensive tools	Growing with graph-specific tools	Growing with scientific analysis tools
Community	Large and active	Active	Growing

2.5 Conclusion

In this chapter the tools and technologies used were described as well as the areas in which each of them is most used. Since there is some similarity in the fields in which they are applied, a whole analysis will be developed, especially regarding the performance of different queries through the different technologies.

Chapter 3

Technologies and Tools Used

3.1 Introduction

This chapter of the report aims to explore the current state of the technologies used in this project, those being PostgreSQL, Neo4J, and TileDB[3]. A review and analysis of their current architectures, features, and trends will be provided.

3.2 PostgresSQL

3.2.1 Architecture:

PostgreSQL is a client-server database management system that uses a multi-process architecture. The server process manages the databases and listens for client requests. When a request from a client is received, the server forces a new process to handle it. The server process has multiple sub-processes to handle tasks such as logging, background tasks, and query execution.

PostgreSQL uses a shared-nothing architecture where each process has its own memory and disk storage. This architecture allows PostgreSQL to scale horizontally by adding more nodes to the cluster. PostgreSQL also supports a master-slave replication architecture, which allows read scaling.

3.2.2 Characteristics:

ACID compliance: PostgreSQL follows the Atomicity, Consistency, Isolation, Durability (ACID) model which consists in having these following characteristics: Atomicity, Consistency, Isolation, and Durability. Consistency and integrity of the data are ensured by the time that transitions are executed.

Extensibility: PostgreSQL allows users to define data types, operators, and custom functions. This feature allows programmers to create complex data structures and algorithms.

Full-text search: PostgreSQL supports full-text searching using its built-in text search engine. This engine supports features such as slicing, sorting, and highlighting.

JSON support: PostgreSQL supports the native JavaScript Object Notation (JSON) data type, which allows programmers to store and query JSON data efficiently.

PostGIS: PostgreSQL has a spatial extension called PostGIS, which adds support for geographic data types and spatial queries.

Parallel Query: PostgreSQL 11 introduced support for parallel query execution, which allows queries to be executed across multiple cores in parallel.

Security: PostgreSQL has a robust security model that includes role-based access control, encryption, and Secure Sockets Layer (SSL)/Transport Layer Security (TLS) support.

3.2.3 Current Trends:

Cloud: PostgreSQL has been increasingly adopted in *cloud* environments, with major *cloud* vendors offering PostgreSQL as a managed service. This trend is driven by the scalability, flexibility, and cost-effectiveness benefits that *cloud* environments offer.

Big Data and Analytics: PostgreSQL is increasingly being used for big data and analytics applications. This trend is driven by PostgreSQL's ability to handle large volumes of data, support complex queries, and integrate with other big data tools.

Machine Learning: PostgreSQL is increasingly being used as a data warehouse for Machine learning applications. This trend is driven by PostgreSQL's ability to support advanced data analysis and processing.

Community Development: The PostgreSQL community continues to grow and contribute to database development. This trend is driven by the open source nature of PostgreSQL, which allows anyone to contribute to its development.

Compatibility with other Databases: PostgreSQL has increasingly been developed to support compatibility with other databases, such as Oracle, MySQL, and Microsoft SQL Server. This trend is driven by the need for organizations to integrate data from multiple sources.

3.2.4 SQL:

SQL (Structured Query Language) is a programming language used to manage and manipulate relational databases. It provides a standardized way to perform operations such as creating and modifying tables, inserting, updating and deleting data, and querying specific information from a database. With SQL's simple and readable syntax, users can write commands to perform a wide variety of tasks, from simple queries to complex join and aggregate data operations. SQL is widely used in many relational database management systems, making it an essential skill for developers and database administrators.

3.2.5 PgAdmin:

PgAdmin is an open source administration and management tool for PostgreSQL databases. It provides a user-friendly graphical interface that allows users to interact with their PostgreSQL databases and perform various administrative tasks. PgAdmin offers a comprehensive set of features, making it a powerful tool for database developers, administrators and other professionals working with PostgreSQL.

With PgAdmin, users can connect to multiple PostgreSQL servers, create and manage databases, tables and views, execute SQL queries and perform data manipulation tasks. It provides a visual query builder that simplifies the process of creating complex queries without requiring in-depth knowledge of SQL syntax. In addition, PgAdmin provides tools for database backup and restoration, user management, database performance monitoring and analysis, and server parameter management.

The PgAdmin interface is organized in a hierarchical structure, allowing users to navigate through database objects and their properties with ease. It supports syntax highlighting and automatic code completion for SQL queries, making it easy to write and edit complex database scripts. PgAdmin also provides a browser-based interface that can be accessed from a web browser, offering flexibility in managing PostgreSQL databases remotely.

As an open source tool, PgAdmin benefits from an active developer community, which ensures regular updates and improvements in functionality and performance. It is available for various operating systems, including Windows, macOS and Linux, making it accessible to a wide range of users.

In summary, PgAdmin is a valuable tool for simplifying the management and administration of PostgreSQL databases, offering an intuitive interface and a comprehensive feature set for efficient database development and maintenance.

3.3 Neo4J

3.3.1 Architecture:

Neo4J is a native graph database management system that stores data in nodes and edges, which are connected through relationships. Each node represents an entity in the graph, and each relationship represents a link between two entities.

Neo4J uses a native graph storage format that allows you to store and retrieve graph data efficiently. The storage format is optimized for graph traversals, making it ideal for complex queries and data analysis.

3.3.2 Characteristics:

Native graphics processing: Neo4J is optimized for graph processing, which allows it to store and retrieve graph data efficiently. This feature allows programmers to create complex graph-based applications and algorithms.

Cypher Query Language: Neo4J uses a declarative query language called Cypher, which is designed specifically for graph databases. Cypher allows programmers to write expressive and intuitive consultations that transverse the graph and gather data in a efficient manner.

Built-in Indexing and Fulltext Search: Neo4J has built-in indexing and full-text search capabilities that allow programmers to efficiently search and retrieve data. This functionality is useful for applications that require complex search queries.

ACID Compliance: Neo4J is ACID compliant, which means that transactions are executed in a way that ensures data consistency and integrity.

High Availability: Neo4J supports high availability through master-slave replication and clustering. This feature enables read scaling and fault tolerance.

Graphic Algorithm Library: Neo4J has a built-in library of graph algorithms that can be used to efficiently analyze and process graph data. This feature is useful for applications that require complex data analysis and processing.

3.3.3 Current Trends:

Chart Analysis: Neo4J is increasingly being used for graph analytics applications where data is highly interconnected and complex. This trend is driven by Neo4J's ability to store and retrieve graph data efficiently, and the availability of graph algorithms for complex data analysis.

Machine Learning: Neo4J is increasingly being used as a data store for machine learning applications. This trend is driven by Neo4J's ability to store and retrieve graph data efficiently, and the availability of graph algorithms for data analysis and processing.

Cloud: Neo4J has been increasingly adopted in cloud environments, with leading cloud providers offering Neo4J as a managed service. This trend is driven by the benefits of scalability, flexibility, and cost-effectiveness that cloud environments offer.

Knowledge Graphs: Neo4J is increasingly being used for knowledge graph applications, where data is organized into a graphical structure to represent knowledge and relationships. This trend is driven by Neo4J's ability to store and retrieve graphical data efficiently, and the availability of graph algorithms for data analysis and processing.

3.3.4 Neo4J Browser:

Neo4J Browser is a powerful tool for visualizing and interacting with graph-oriented databases in Neo4J. It provides an intuitive graphical interface that allows users to efficiently explore and query data using the Cypher query language.

With Neo4J Browser, users can view and navigate through the nodes, relationships and properties in a graph database. It offers interactive visualization features, such as the display of nodes and relationships in graph form, tables and detailed property panels. This makes it easier to understand the structure and connections of the data stored in the database.

In addition, Neo4J Browser allows users to execute queries in real time using the Cypher language. Through a query editing interface, users can write and execute queries to retrieve specific information from the graph data. Neo4J Browser provides syntax highlighting and autocompletion features for the Cypher language, making it easy to write accurate and efficient queries. Neo4J Browser also offers advanced features, such as the ability to save favorite queries, share data visualizations via URLs and export query results in different formats such as JSON and CSV.

Overall, Neo4J Browser is an essential tool for exploring, querying and visualizing data stored in graph databases in Neo4J. With its intuitive graphical interface and advanced features, it helps users understand data structure, write efficient queries and extract valuable information from their graph databases.

3.3.5 APOC Library:

The Awesome Procedures on Cypher (APOC) Library is an open source extension to Neo4J that provides a diverse set of additional procedures and functions to expand the capabilities of the Cypher Query Language.

Developed by the Neo4J community, APOC Library provides a wide range of additional features for querying and manipulating data in graph databases. These features include advanced data import/export operations, string and date manipulation, graph transformations, custom index creation, full-text processing, calls to external APIs, and more.

The APOC Library is designed to facilitate complex tasks in Neo4J, allowing users to extend their Cypher queries with ease and perform advanced operations efficiently. The library's procedures and functions are called directly in Cypher queries, expanding the capabilities of the language and providing a more flexible and powerful way to work with graph data.

Due to its open source nature, the APOC Library is continuously updated and improved by the Neo4J community, offering new features and additional functionality over time. With its wide range of procedures and functions, the APOC Library is a valuable tool for Neo4J developers and users who wish to explore and manipulate graph data in more advanced and customized ways.

3.3.6 Cypher Query Language:

The Cypher Query Language is a declarative query language developed specifically for graph databases and is widely used in Neo4J. It allows users to retrieve and manipulate data in a graph format in an intuitive and efficient manner.

Cypher was designed to express queries in a manner similar to the visual structure of a graph, using a readable and expressive syntax. The key elements of the language are nodes, relationships and properties. Based on these elements, you can build queries that describe patterns and relationships between data.

Cypher queries are composed of clauses that define the actions to be performed on the graph. Some of the most common clauses are:

- MATCH: specifies patterns of nodes and relationships to be found in the graph.
- WHERE: adds conditions to filter results based on specific properties.
- RETURN: defines what data should be returned as the result of the query.
- CREATE: creates new nodes and relationships in the graph.
- DELETE: removes nodes and relationships between them.

In addition, Cypher offers a wide range of functions and operators to perform calculations, transformations and aggregations on data.

One of Cypher's most powerful features is its ability to navigate the graph using matching patterns. These patterns allow users to find specific paths in the graph, discover complex relationships between data, and perform advanced queries with ease.

Overall, the Cypher Query Language is a fundamental tool for interacting with graph databases, such as Neo4J. With its intuitive syntax and advanced features, it allows users to perform complex queries, gain valuable insights and efficiently explore the relationships between data in the context of a graph.

3.4 TileDB

3.4.1 Architecture:

TileDB is based on a new storage format called TileDB Array, which provides a highly optimized and efficient way to store and access multi-dimensional data. TileDB Array is based on a tile approach, where data is divided into small tiles and stored in a way that allows efficient query and access.

TileDB also provides a range of interfaces and APIs for data access and manipulation, including a C++ API, a Python API, a SQL interface, and a REpresentational State Transfer (REST) API. These interfaces provide a way to work with data, regardless of the type or structure of the data.

3.4.2 Characteristics:

Managing Multi-Dimensional Data: TileDB is designed to store and manage multidimensional data, such as arrays, matrices, and tensors. This feature allows programmers to work with complex data sets that require high-dimensional representations.

Cloud: TileDB is designed to work seamlessly with *Cloud* environments, with native support for cloud storage providers such as Amazon S3 and Microsoft Azure. This feature allows developers to easily deploy and manage TileDB in the cloud.

Unified Data Management: TileDB provides a unified framework for storing and managing various types of data, including structured, semi-structured and unstructured data. This functionality allows developers to work with data in a consistent and scalable way.

High Performance Data Access: TileDB provides highly optimized and efficient ways to access and manipulate data, including parallel and vectorized operations. This feature allows programmers to work with large, complex data sets in a fast and efficient manner.

Data Compression: TileDB supports data compression at multiple levels, including tile-level and global-level compression. This feature allows developers to reduce storage requirements and optimize data transfer.

Data versioning: TileDB provides data versioning support, allowing programmers to manage and track changes to data over time. This feature is useful for applications that require version control and auditing.

3.4.3 Current Trends:

Big Data Analytics: TileDB is increasingly being used for big data analytics applications, where large and complex data sets need to be processed and analyzed. This trend is driven by TileDB's ability to store and manage multidimensional data efficiently, and the availability of optimized operations for data access and manipulation.

Genomic Data Management TileDB has been increasingly used for genomic data management, where large-scale genomic datasets need to be stored, managed and analyzed. This trend is driven by TileDB's ability to efficiently store and access genomic data, as well as the availability of optimized operations for genomic data processing.

Cloud: TileDB has been increasingly adopted in cloud environments, with leading cloud providers offering TileDB as a managed service. This trend is driven by the benefits of scalability, flexibility, and cost-effectiveness that cloud environments offer.

Machine Learning: TileDB has been increasingly used in data storage for machine-learning applications. This trend is driven by TileDB's capacity of storing and managing many different types of data, as well as a wide array of optimized access and modifying data operations.

3.4.4 Python:

Python is a high-level, interpreted, general-purpose programming language. It is known for its simplicity and readability, making it a popular choice for both beginners and experienced developers.

With a clean and concise syntax, Python allows developers to write code more clearly and efficiently. It is an object-oriented language, which means that it supports concepts such as inheritance, encapsulation and polymorphism. In addition, Python also offers functional programming features and supports the creation of lambda functions and the application of techniques such as map, filter and reduce.

Python has an extensive standard library that provides a variety of modules and tools for various tasks, from string manipulation, file processing, database access, to web development and graphical interface creation. This vast library, along with a large developer community, contributes to the availability of many third-party packages and frameworks that facilitate development in Python.

One of Python's main features is its portability, as it runs on a variety of operating systems, including Windows, macOS and Linux. In addition, its growing popularity has driven the creation of extensions and integrations with other languages, allowing the development of cross-platform applications and the incorporation of code from other languages into Python projects.

Due to its versatility, Python is widely used in different domains such as web development, data analytics, machine learning, task automation, data science and many others. Its combination of simplicity, power and flexibility makes Python a popular choice for a wide range of programming projects.

3.4.5 NumPy:

The NumPy library is a fundamental library for scientific computing in Python. It provides efficient data structures and tools for working with multidimensional arrays, which is especially useful for manipulating and processing numerical data.

In the context of TileDB, a library that allows the efficient storage and management of large datasets in different formats, NumPy plays an important role. TileDB provides support for storing multidimensional arrays as TileDB data on disk or in the cloud, and the NumPy library allows users to interact with these arrays conveniently and efficiently.

With NumPy and TileDB, users can perform sophisticated operations on large volumes of data, such as filtering, aggregation, statistical calculations, and parallel processing on multidimensional arrays. The integration between NumPy and TileDB allows users to leverage the advanced functionality of NumPy and the scalability and storage efficiency of TileDB.

In addition, NumPy is also used to load and save data in TileDB format, allowing users to easily read and write multidimensional arrays in TileDB format. This facilitates integration between the NumPy-based ecosystem of tools and TileDB, providing a more consistent and efficient development experience.

Overall, using the NumPy library in conjunction with TileDB allows users to work with multidimensional arrays efficiently, perform advanced operations on large volumes of data, and take advantage of the efficient storage and management capabilities offered by TileDB. This combination is particularly useful for scientific and data analysis applications that require the processing of large data sets.

3.4.6 Pandas:

The Pandas library is a powerful Python tool for analyzing and manipulating tabular data. It offers flexible and efficient data structures, such as the DataFrame, that allow working with structured data in an intuitive way.

In the context of TileDB, a library that enables the efficient storage and management of large datasets in different formats, Pandas plays an important role by providing convenient integration with TileDB.

With Pandas and TileDB, users can load and save data in TileDB format directly into a DataFrame. This makes it easy to read and write tabular data stored in the TileDB format, enabling users to take advantage of Pandas' advanced functionality for data analysis and table manipulation.

By using Pandas with TileDB, users can perform powerful DataFrame operations such as filtering, aggregation, column selection and applying bulk functions. These operations can be applied to large-volume tabular data while maintaining the storage efficiency and scalability provided by TileDB.

In addition, Pandas offers a wide range of statistical, visualization and data manipulation features that can be applied directly to data stored in TileDB. This enables users to extract valuable insights, perform complex analysis and prepare data for modelling and visualisation.

In summary, using the Pandas library in conjunction with TileDB provides a powerful combination for analyzing and manipulating tabular data. The integration between the two libraries allows users to take advantage of the advanced functionality of Pandas and the storage and management efficiency of TileDB when working with large tabular datasets.

Chapter

4

Solution details

4.1 Introduction

This chapter covers the implementation details of three different solutions for this project, each based on a specific technology: PostgreSQL, Neo4J and TileDB. In this introduction, we will provide an overview of these technologies and discuss the steps required to install and use each of them in a Linux environment.

4.1.1 PostgreSQL:

PostgreSQL is a widely used open source relational database management system. It offers advanced features such as ACID (Atomicity, Consistency, Isolation, and Durability) support, triggers, stored procedures, and extensibility. To install PostgreSQL in a Linux environment, you can follow these steps:

Open a terminal and run the following command:

```
sudo apt-get install postgresql
```

Code Snippet 4.1: Command to install Postgres on linux.

Wait until the installation process completes. Once installed, you can begin using PostgreSQL by running the command:

```
psql
```

Code Snippet 4.2: Command to start Postgres on linux.

4.1.2 Neo4J:

Neo4J is a high-performance graph-oriented database designed to store and query highly connected data. It provides a flexible framework for modeling complex data, allowing efficient queries on large sets of related data. To install Neo4J in a Linux environment, follow these steps: Go to the official Neo4J website (https://Neo4J.com) and download the appropriate version for your operating system. In the terminal, navigate to the folder where the file with the .appImage extension is stored and run the following commands:

```
chmod a+x Neo4J-desktop-1.0.3-x86_64.AppImage

./Neo4J-desktop-1.0.3-x86_64.AppImage
```

Code Snippet 4.3: Command to install and execute Neo4J's Browser.

Wait until the Neo4J server starts. To interact with the Neo4J database, you can use the Neo4J Browser by going to the following address in your web Browser: http://localhost:7474..

4.1.3 TileDB:

TileDB is an efficient and versatile multidimensional data storage library. It enables optimized storage and retrieval of large volumes of multidimensional data, making it suitable for many applications such as data science, machine learning, and geospatial analysis. To install TileDB in a Linux environment, I took the approach of installing a pre-built package in the terminal, we can do it with one of two following ways:

```
# Install from PyPI:
$ pip install TileDB

# Or Conda:
$ conda install -c conda-forge TileDB-py
```

Code Snippet 4.4: Command to install TileDB.

4.2 Solution Details

The two solutions developed in this project were made by using a database dump from the database currently in production, which was developed with POSTGRES. To use this database dump, firstly we need to install pgadmin so we can import the database to pgadmin to later use the data and export it to csv files. Once pgadmin is installed, we use the psql tool and with the command /i "file path of the dump" we can import the database. To export all the data, we just go to each individual table, we click with mouse button 2, and we select the export option, which will generate a Comma-Separated Values (CSV) file with the headers and data from the corresponding table. These csv files will be later used to develop the solutions in Neo4J and Tildb, since all the tests will be made in two local databases.

4.2.1 Neo4J:

4.2.1.1 Data Ingestion:

In this section, we will present the details of implementing a database using CSV files in Neo4J Browser, on a Linux operating system. The Neo4J Browser was used to facilitate the creation and visualization of the nodes created themselves as well as the relationships created between them. In this way it was extremely easy to verify that the data was well imported into the database in a consistent manner, as it is not only possible to verify that the nodes were created correctly as well as verify that the properties and their types match the types in the database in production. To perform the ingestion of the CSV data, the LOAD CSV WITH HEADERS FROM "filepath" and CREATE commands were used. The CREATE command has two uses, it is used to create the nodes themselves and also to create indexes for the nodes with the label being created. Here is an example:

```
LOAD CSV WITH HEADERS FROM 'file:///analysis_centers.csv' AS rowACenter
WITH rowACenter WHERE NOT rowACenter.id IS NULL
CREATE(ac: analysis_centers
  {
    id: toInteger(rowACenter.id),
    name: rowACenter.name,
    abbreviation: rowACenter.abbreviation,
    contact: rowACenter.contact,
    email: rowACenter.email,
    url: rowACenter.url

});

CREATE INDEX analisys_centers_id FOR (ac:analysis_centers) ON (ac.id);
```

Code Snippet 4.5: Data ingestion of a CSV in Neo4J

However, this approach does not work for tables with large volumes of data, some of them with millions of entries. In this case with this first approach at the time of creating nodes for these specific tables, exceptions of lack of memory were raised, since these implementations were made locally on a personal computer. For this, an alternative approach had to be developed, using the CALL... ÎN TRANSACTIONS commands. These two commands together avoid any kind of memory errors when creating nodes. Here's an example of a table that uses this approach:

```
    CREATE INDEX processing_parameters_id FOR (param:processing_parameters) ON
        (param.id);


:auto LOAD CSV WITH HEADERS FROM 'file:///estimated_coordinates.csv' AS rowEst
CALL{
    WITH rowEst
    CREATE(rowEstim: Estimated_coordinates
      {
            id: toInteger(rowEst.id),
            x: toFloat(rowEst.x),
            y: toFloat(rowEst.y),
            z: toFloat(rowEst.z),
            var_xx: toFloat(rowEst.var_xx),
            var_yy: toFloat(rowEst.var_yy),
            var_zz: toFloat(rowEst.var_zz),
            var_xy: toFloat(rowEst.var_xy),
            var_xz: toFloat(rowEst.var_xz),
            var_yz: toFloat(rowEst.var_yz) ,
            outlier: toInteger(rowEst.outlier),
            epoch: rowEst.epoch,
            id_processing_parameters: toInteger(rowEst.id_processing_parameters
                ),
            id_method_identification: toInteger(rowEst.id_method_identification
                ),
            id_station: toInteger(rowEst.id_station),
            id_product_files: toInteger(rowEst.id_product_files)

      })
    }IN TRANSACTIONS OF 100000 ROWS;

CREATE INDEX Estimated_coordinates FOR (rowEstim:Estimated_coordinates) ON (
    rowEstim.id);
```

Code Snippet 4.6: Data ingestion of a large CSV in Neo4J

4.2.1.2 Relationships

In addition, relationships were created between the nodes using the primary and foreign keys of each table, using the MATCH and WHERE commands to check the equality of the properties and MERGE to create the relationships. The command MATCH, makes match to the nodes that possess the labels we want to use to create the relationships, and the command WHERE allows using properties of the nodes where we can check if the properties between these nodes match, thus simulating the utility between Primary keys and Foreign keys of SQL. The MERGE command is used to create relationships just to ensure that we do not duplicate any relation since the data volume is very high (close to 30 million relationships). Here is an example:

```
MATCH (st:Station), (sc:Station_colocation)
WHERE st.id = sc.id_station
MERGE (st)-[:IS_COLLOCATED]->(sc);
```

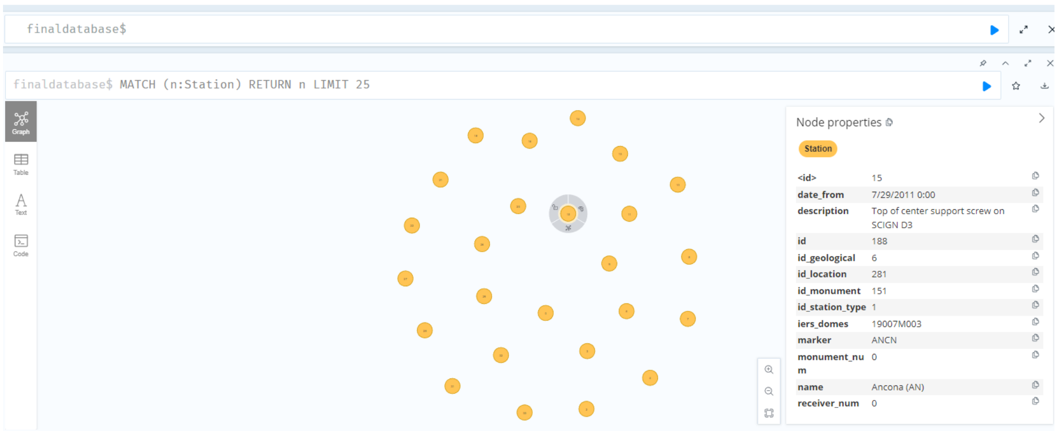
Code Snippet 4.7: Relationship creation

In the same way as the creation of nodes, the creation of relations for tables with a large volume of data has to take a data approach. To enable this task to be performed, two commands in particular had to be used, the SKIP command and the LIMIT command. The SKIP command is responsible for ignoring a number of rows starting from the top of the csv file, and the LIMIT command has the functionality of limiting the number of rows used for the creation of nodes, thus providing the possibility of batch creation. This way we avoid any kind of memory error. Here is an example:

```
MATCH (st:Station),(e:Estimated_coordinates)
WHERE st.id = e.id_station
WITH st, e
SKIP 0
LIMIT 1500000
MERGE (st)-[:HAS_ESTIMATED_COORDINATES_ST]->(e)
```

Code Snippet 4.8: Relationship creation for label with large number of nodes

To finish this sub-section we now present two print-screens, the first one 4.1 about ingesting and creation of data from a label (table) and visualizing it in the Neo4J Browser, and the second one 4.2 about creating and visualizing a relationship.



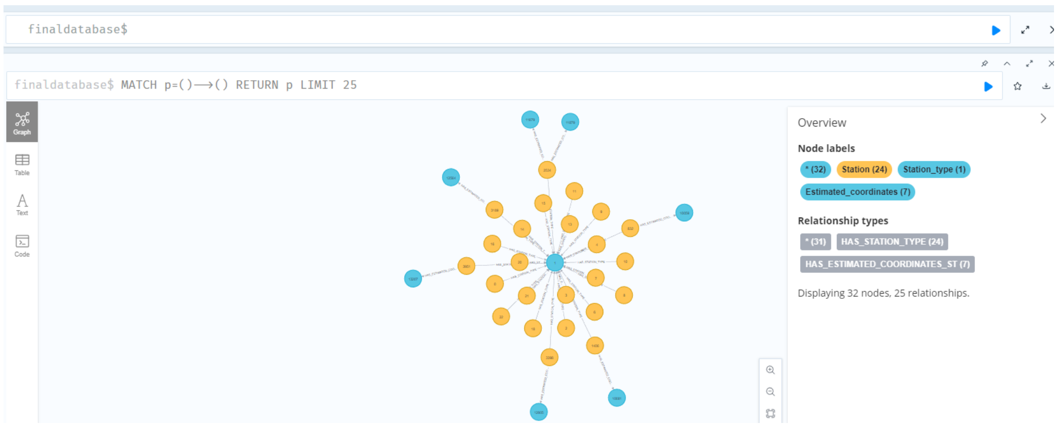
```
finaldatabase$
finaldatabase$ MATCH (n:Station) RETURN n LIMIT 25
```

Node properties

Station

<id>	15
date_from	7/29/2011 0:00
description	Top of center support screw on SCIGN D3
id	188
id_geological	6
id_location	281
id_monument	151
id_station_type	1
iers_domes	19007M003
marker	ANCN
monument_num	0
name	Ancona (AN)
receiver_num	0

Figure 4.1: Created nodes for the label station after data ingestion



```
finaldatabase$
finaldatabase$ MATCH p=()⟶() RETURN p LIMIT 25
```

Overview

Node labels

* (32) Station (24) Station_type (1) Estimated_coordinates (7)

Relationship types

* (31) HAS_STATION_TYPE (24) HAS_ESTIMATED_COORDINATES_ST (7)

Displaying 32 nodes, 25 relationships.

Figure 4.2: Created relationships between different labels

4.2.2 TileDB:

4.2.2.1 Data Ingestion:

To make the data ingestion in the TileDB solution it was used the Pandas library command read_csv(). The functionality of this command consists of the following:

1. It takes the path or URL of the CSV file as its first argument.

2. It parses the contents of the CSV file and creates a DataFrame object, which is a two-dimensional labeled data structure with columns of potentially different types.

3. The function automatically detects the structure of the CSV file, including the column headers and the delimiter used (typically a comma , but it can be customized).

4. It infers the data types of the columns based on the values present in the file.

5. It can handle various configurations and options, such as handling missing values, specifying column data types, specifying custom delimiters, skipping rows, and more.

All the attributes of the tables have been hard-coded previously so that after ingestion there is no incompatibility of types, in this way the full functioning of the arrays is ensured. If the tables have an attribute or attributes that represent a date type, an array is created with the name of the attributes to be passed as a parameter to the read_csv() function to be parsed later. And finally with the help of the command from_Pandas, using the data frame created previously the arrays can now be created since from_Pandas converts the Pandas DataFrame into a TileDB array and stores it in the specified location, the function automatically detects the schema and data types from the Pandas DataFrame and creates the appropriate TileDB array schema.

Here are two examples of data ingestion, one has no attributes representing a data type and the other one has.

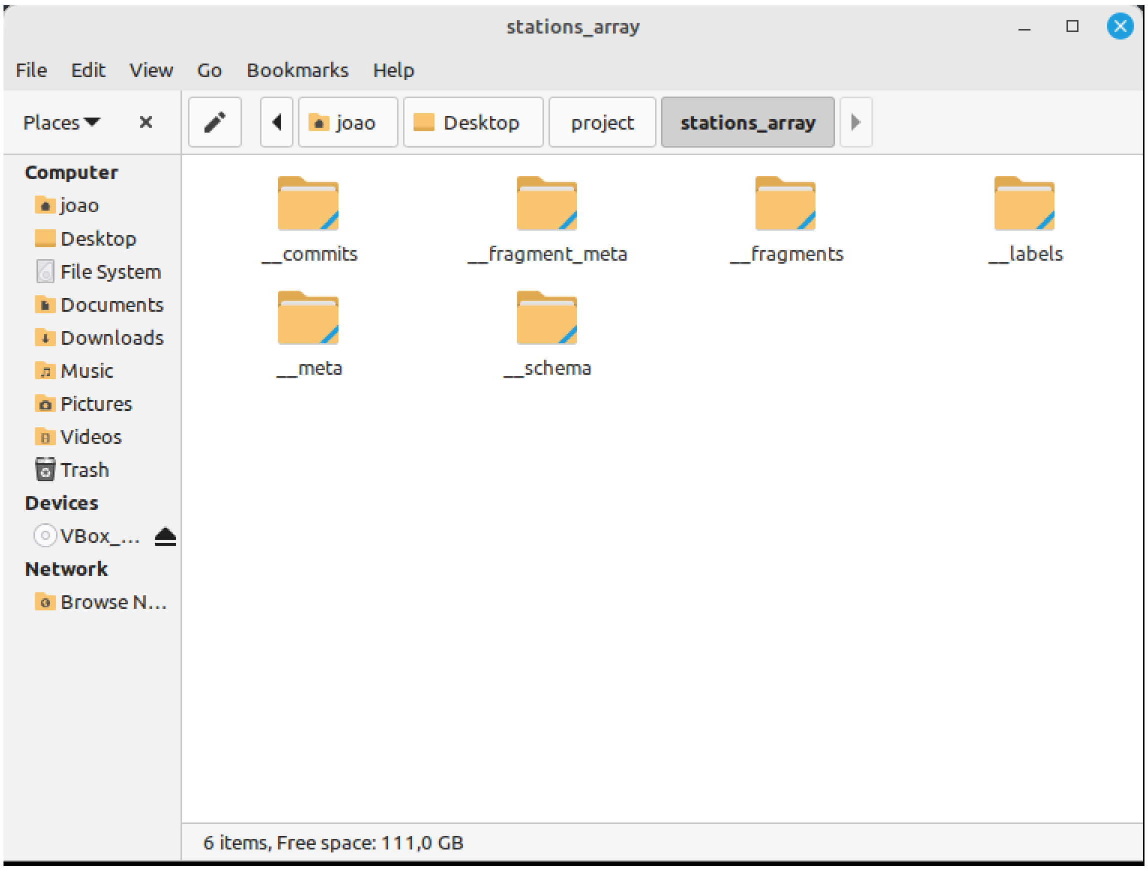
```
csv_file = "/home/joao/Desktop/project/csvs/user_group_station.csv"

dtype = {
    "id": "int32",
    "id_station": "int32",
    "id_user_group": "int32"

}



df = pd.read_csv(csv_file, dtype=dtype)

TileDB.from_Pandas("user_group_station_array", df)
```

Code Snippet 4.9: Data ingestion of csv file with no data attributes.

```
csv_file = "/home/joao/Desktop/project/csvs/estimated_coordinates.csv"

dtype = {
    "id": "int32",
    "x": "float32",
    "y": "float32",
    "z": "float32",
    "var_xx": "float32",
    "var_yy": "float32",
    "var_zz": "float32",
    "var_xy": "float32",
    "var_xz": "float32",
    "var_yz": "float32",
    "outlier": "int32",
    "epoch": "str",
    "id_processing_parameters": "int32",
    "id_method_identification": "int32",
    "id_station": "int32",
    "id_product_files": "int32"
}

parse_dates = ["epoch"]

df = pd.read_csv(csv_file, dtype=dtype, parse_dates=parse_dates)


TileDB.from_Pandas("estimated_coordinates_array", df)
```

Code Snippet 4.10: Data ingestion of csv file with data attributes.

The result of the data ingestion is a folder with the data and information about the created array, it contains its meta-data, data fragments, the array schema and the data used in the previously created data frame to be used in the from_Pandas function. The image 4.3 represents the result of the array creation.



__commits
__fragment_meta
__fragments
__labels
__meta
__schema

6 items, Free space: 111,0 GB

Figure 4.3: Result of array creation

TileDB follows a multidimensional array data model, which is different from the relational model used in traditional databases. In the array data model, data is organized and accessed based on multi-dimensional indices rather than through primary and foreign keys. The arrays just need to be created, because all the data manipulation will be done later when the queries are made and the data is retrieved.

4.3 Conclusion

This chapter gives an overview regarding the implementation of two different solutions based on the database currently in production developed using PostgreSQL. In summary, Neo4J uses a graph-based data model where nodes represent entities (similar to tables in relational databases), node properties represent attributes (similar to columns in relational databases), and relationships between nodes represent associations (analogous to foreign keys in relational databases).
In TileDB, the main data structure is the multidimensional array, which can have multiple dimensions and attributes associated with it. Arrays can be sparse or dense and store data efficiently, allowing to easily slice and dice the data along specified dimensions. Dimensions are used for indexing and attributes store the actual data associated with the array. This data model enables efficient storage and retrieval of large-scale multidimensional data.

Chapter

5

Query Development and Performance Testing

5.1 Introduction

This next chapter will analyze the execution times of materialized views already implemented in PostgreSQL, as well as several queries developed for performance testing purposes in different technologies.

5.2 Methodology

As mentioned in the introduction we will measure the execution time of the developed queries and analyze their performance. For that, these tests will follow a set of rules that are as follows:

- **Exclude Planning Time:** To accurately measure query execution time, it is common practice to run the query once to warm up the database's query planner. This means that you execute the query once, discard the result, and then measure the execution time for subsequent runs. This way, you only measure the time taken to execute the already planned query and exclude the planning overhead.

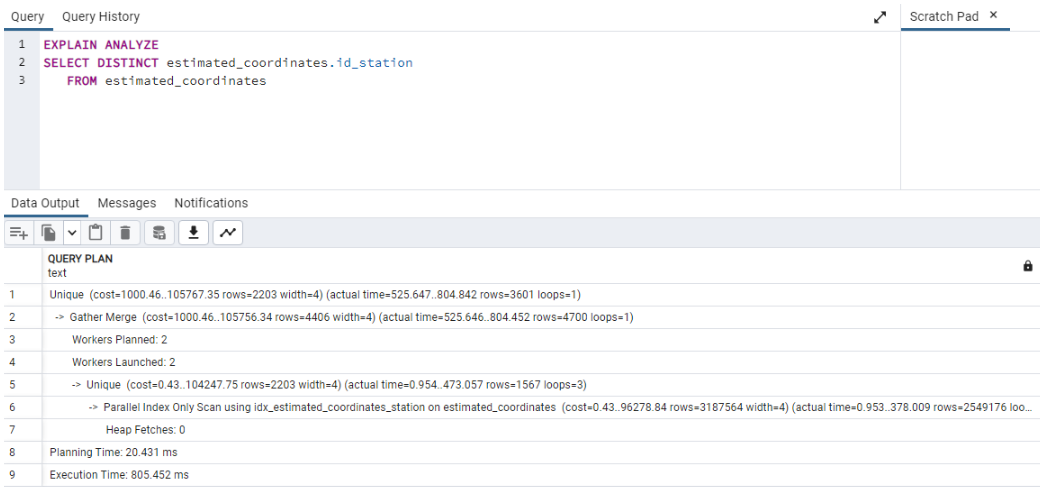
- **Isolate the Query:** Run the query in isolation to minimize interference from other concurrent queries or processes that might affect the execution time.

- **Average Multiple Runs:** Execute the query multiple times (at least 5-10 times) and take the average of the execution times. This helps to reduce the impact of fluctuations due to external factors like system load, caching, or network latency.

- **Consistency:** Ensuring that the data used for testing is consistent between different runs to get reliable and comparable results.
- **Resource Monitoring:** Monitor system resource utilization during query execution. High Central Processing Unit (CPU), memory, or disk usage can indicate potential performance issues.

5.2.1 PostgreSQL

For the implementation in PostgreSQL the queries that were developed were executed in Pgadmin using the query tool and the EXPLAIN ANALYZE command at the beginning of each script. In SQL, the EXPLAIN ANALYZE command is a useful tool to understand the execution plan of a query and obtain detailed information about how the database intends to execute the query. From this information we only extract the execution   time.
An example of an output of this command is shown in the following figure 5.1:



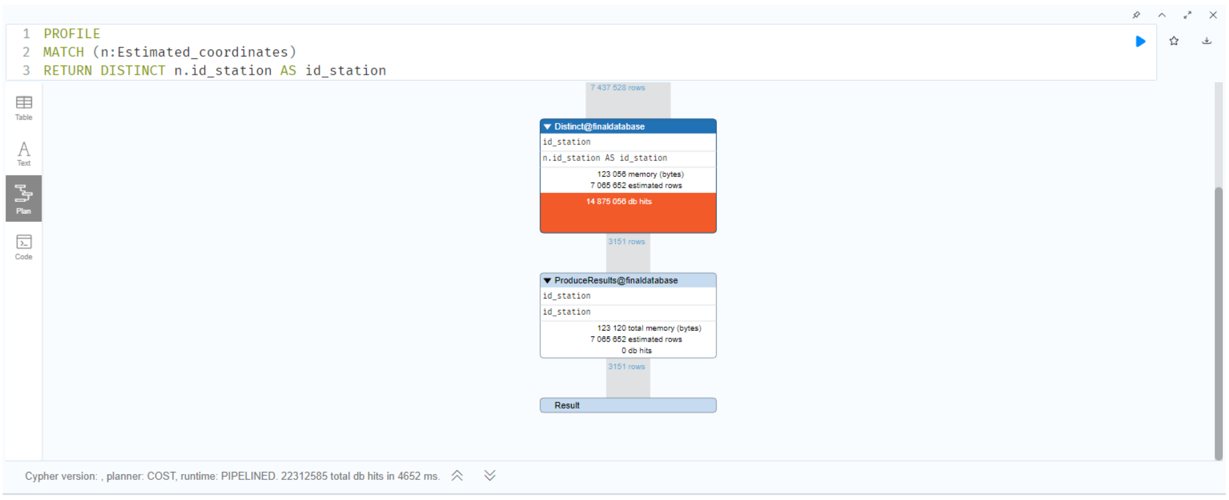
```sql
EXPLAIN ANALYZE
SELECT DISTINCT estimated_coordinates.id_station
   FROM estimated_coordinates
```

```
QUERY PLAN
text
Unique  (cost=1000.46..105767.35 rows=2203 width=4) (actual time=525.647..804.842 rows=3601 loops=1)
  ->  Gather Merge  (cost=1000.46..105756.34 rows=4406 width=4) (actual time=525.646..804.452 rows=4700 loops=1)
        Workers Planned: 2
        Workers Launched: 2
        ->  Unique  (cost=0.43..104247.75 rows=2203 width=4) (actual time=0.954..473.057 rows=1567 loops=3)
              ->  Parallel Index Only Scan using idx_estimated_coordinates_station on estimated_coordinates  (cost=0.43..96278.84 rows=3187564 width=4) (actual time=0.953..378.009 rows=2549176 loo...
                    Heap Fetches: 0
Planning Time: 20.431 ms
Execution Time: 805.452 ms
```

Figure 5.1: Example of the output of the EXPLAIN ANALYZE command

5.2.2 Neo4J

In the case of Neo4J the query execution times were measured in the Neo4J browser using the PROFILE functionality. The PROFILE 5.2 command in the Neo4J Browser indeed provides both the query execution plan and the execution time. This makes it a valuable tool for query optimization and performance tuning in the Neo4J environment.



```
PROFILE
MATCH (n:Estimated_coordinates)
RETURN DISTINCT n.id_station AS id_station
```

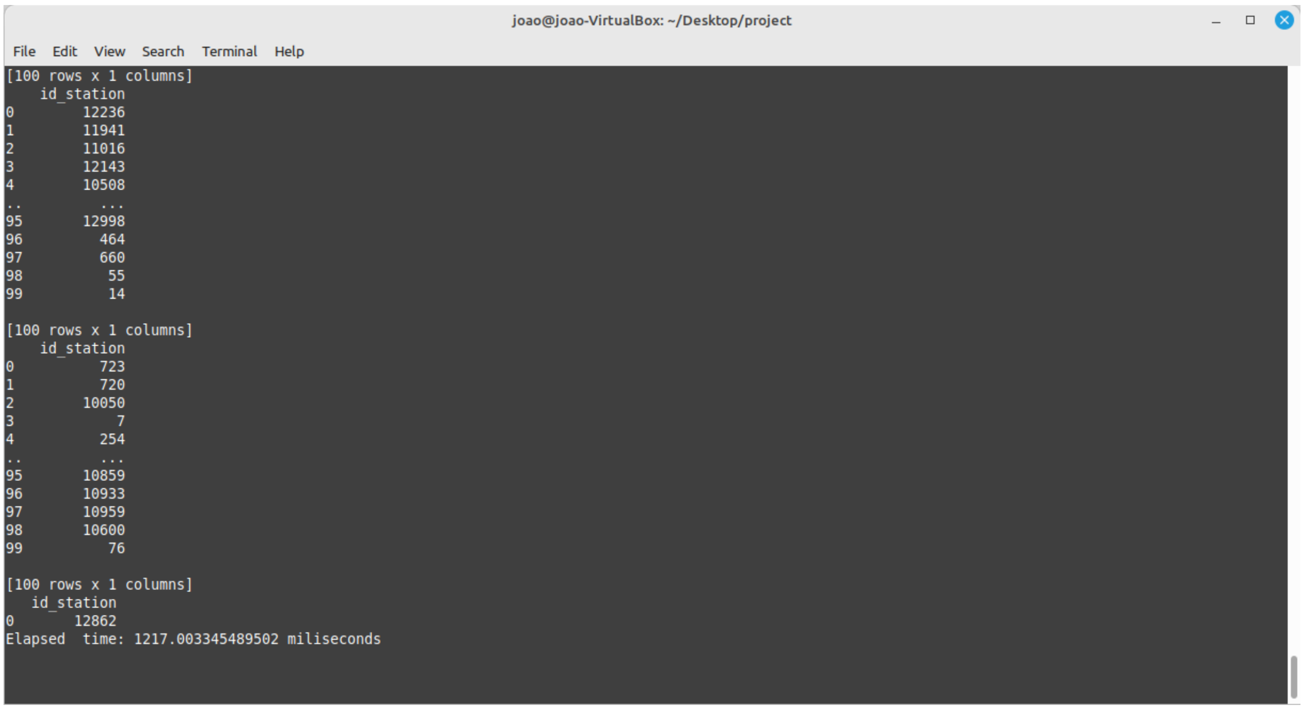
```
7 437 528 rows
Distinct@finaldatabase
id_station
n.id_station AS id_station
123 056 memory (bytes)
7 065 652 estimated rows
14 875 056 db hits
3151 rows
ProduceResults@finaldatabase
id_station
id_station
123 120 total memory (bytes)
7 065 652 estimated rows
0 db hits
3151 rows
Result
```

```
Cypher version: , planner: COST, runtime: PIPELINED. 22312585 total db hits in 4652 ms.
```

Figure 5.2: Example of the output of the PROFILE command

5.2.3 TileDB

The approach taken to measure query execution times in TileDB is actually quite simple. Since the python API was chosen to implement this solution, we can use the library time to track the query execution time. Two variables were used to store two different objects, one at the start of the query and another at the end. Following this approach we can get the elapsed time of the query while running the respective python script.


joao@joao-VirtualBox: ~/Desktop/project
File Edit View Search Terminal Help

```
[100 rows x 1 columns]
    id_station
0        12236
1        11941
2        11016
3        12143
4        10508
..         ...
95       12998
96         464
97         660
98          55
99          14

[100 rows x 1 columns]
    id_station
0          723
1          720
2        10050
3            7
4          254
..         ...
95       10859
96       10933
97       10959
98       10600
99          76

[100 rows x 1 columns]
   id_station
0       12862
Elapsed  time: 1217.003345489502 miliseconds
```

Figure 5.3: Example of the output of a TileDB array script

5.3 Results

This section of the report will show the results obtained during the execution of the queries. The execution times were all measured in milliseconds and will be presented through bar charts(5.4, 5.5, 5.6,5.7, 5.8, 5.9, 5.10) for a better visualization and comparison of the results.

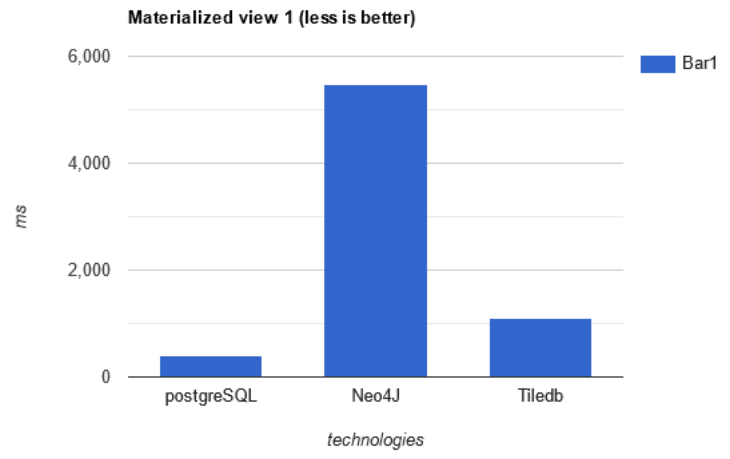


Figure 5.4: Execution times for materialized view 1 query

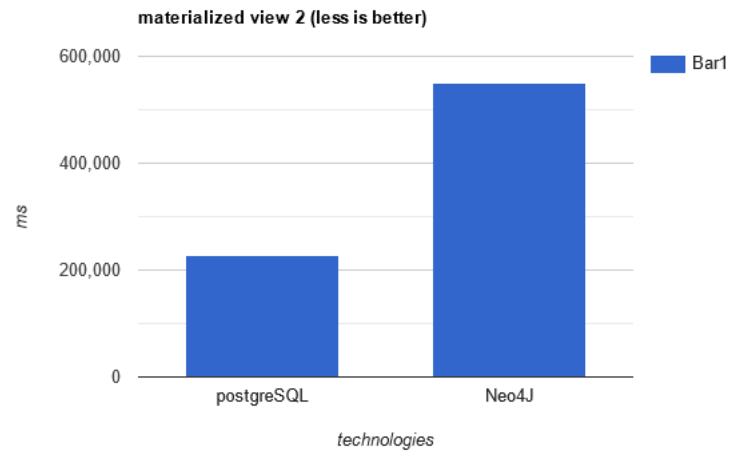


Figure 5.5: Execution times for materialized view 2 query

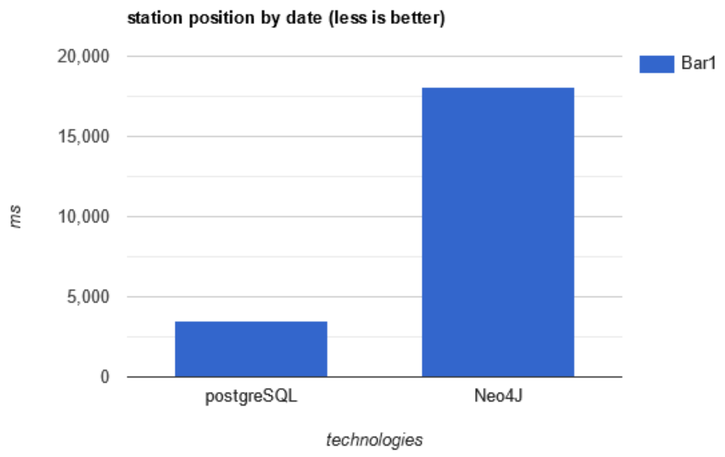


Figure 5.6: Execution times for station position by date query

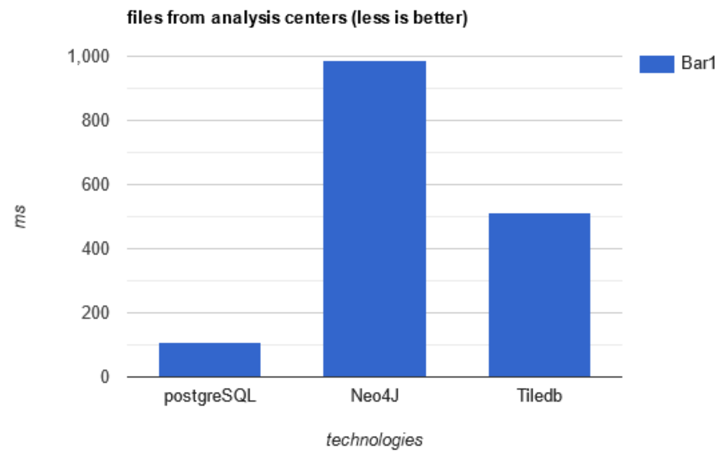


Figure 5.7: Execution times for product files from analysis centers query

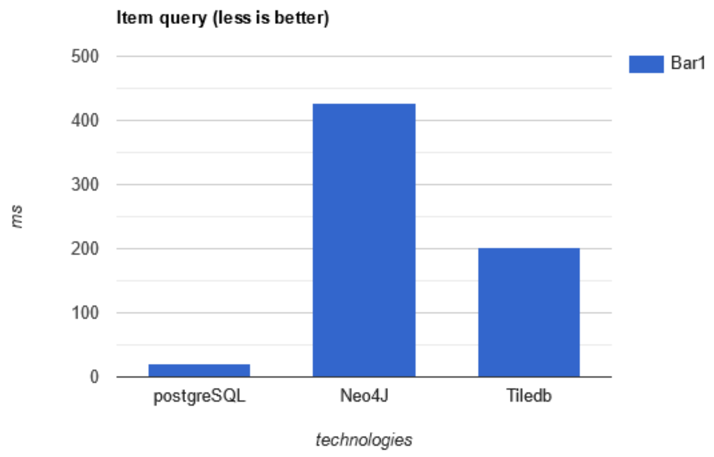


Figure 5.8: Execution times for item type per item id query

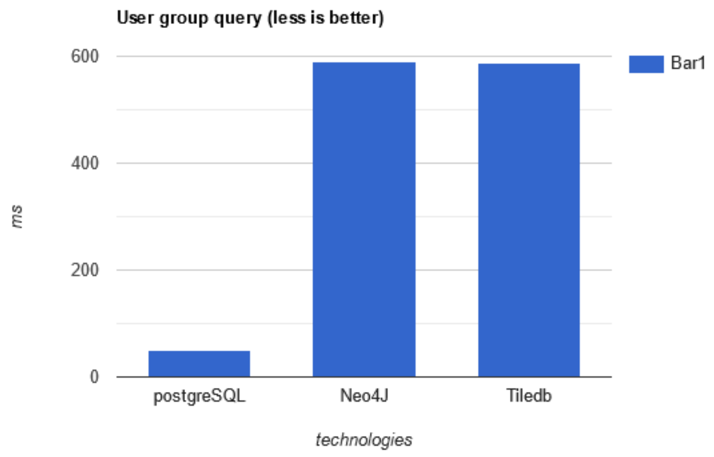


Figure 5.9: Example of the output of the PROFILE command

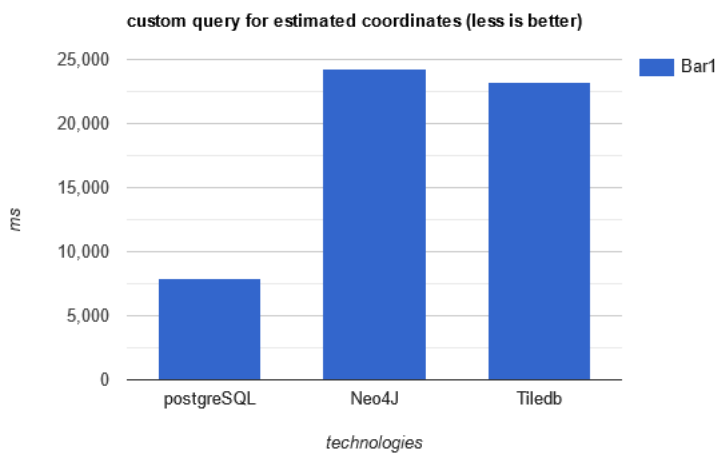


Figure 5.10: Example of the output of the PROFILE command

Table 5.1: Comparison of PostgreSQL, Neo4J, and TileDB query performance (values presented in ms)

Aspect	**PostgreSQL**	**Neo4J**	**TileDB**
Materialized View 1	393.505	5485.049	1094.330
Materialized View 2	55481.374	227073.206	N/A
Station position by date	3518.232	18062.915	N/A
A. Centers P. Files	107.564	985.735	512.084
User-group	51.175	589.884	588.409
Items by item type	21.546	426.239	201.621
Custom query	7941.301	24309.135	23231.364

5.4 Conclusion

With the development and execution of the queries and the subsequent analysis of the results obtained, it was possible to obtain conclusive data that will be further discussed in the next chapter. Despite the difficulties regarding the queries developed on the table "Estimated_coordinates" - due to its large volume of data, it was difficult to develop them locally, given the hardware limitations - it was still possible to get sufficient data to obtain a solid conclusion.

Chapter 6

Conclusions and Future Work

6.1 Key Conclusions

It was the goal of this project to develop two alternative solutions for the production database that is implemented in PostgreSQL and compare the performance of the three solutions. With the tests carried out for the purpose of measuring the execution times of the different queries it can be concluded PostgreSQL is faster in this specific scenario where the queries are predominantly based on common tabular operations and do not require specific features of TileDB, such as support for multidimensional arrays and dense data or do not require the analysis of complex graph structures as in Neo4j. Therefore, the work developed successfully achieves the objectives delimited at the beginning of the project. In personal terms, it is the author's view that through the elaboration of this project, there was not only the solidification of acquired knowledge but also the achievement of new paradigms that will prove useful in the future.

6.2 Future Work

Although the project was successfully completed two of the test queries that focused on the Estimated Coordinates table did not return data, possibly due to lack of optimization of the queries and hardware limitations given the massive size of the table. However, after analyzing the results obtained, it would be interesting to test different DMBSs that better adapt to the type of data present in the database in production, or test some hybrid approach with the technologies used in the project. It would also be interesting to analyze the performance of the technologies in a larger scope, where hardware limitations are minimized.

In summary we have:

1. Further optimization work regarding the queries, specially the TileDB ones that didn't return data.
2. Study and implement solutions with others DBMS that better adapt to a relational model.
3. Investigate the possibility of a hybrid solution.
4. Test developed solutions within a larger scope, to see how they behave with less hardware limitations.

Chapter

7

Appendix A

7.1 Relevant Code Snippets

7.1.1 Neo4J Snippets

```
LOAD CSV WITH HEADERS FROM 'file:///station.csv' AS rowSt

WITH rowSt WHERE NOT rowSt.id IS NULL
CREATE (st:Station
  {
    id: toInteger(rowSt.id),
    name: rowSt.name,
    marker: rowSt.marker,
    description: rowSt.description,
    date_from: rowSt.date_from,
    date_to: rowSt.date_to,
    id_station_type: toInteger(rowSt.id_station_type),
    comment: rowSt.comment,
    id_location: toInteger(rowSt.id_location),
    id_geological: toInteger(rowSt.id_geological),
    id_monument: toInteger(rowSt.id_monument),
    iers_domes: rowSt.iers_domes,
    cpd_num: toInteger(rowSt.cpd_num),
    monument_num: toInteger(rowSt.monument_num),
    receiver_num: toInteger(rowSt.receiver_num),
    country_code: rowSt.countrycode
  });

CREATE INDEX station_id IF NOT EXISTS FOR (st:Station) ON (st.id);
```

```
LOAD CSV WITH HEADERS FROM 'file:///product_files.csv' AS rowPfiles
WITH rowPfiles WHERE NOT rowPfiles.id IS NULL
CREATE(pfil: product_files
  {
    id: toInteger(rowPfiles.id),
    url : rowPfiles.url,
    epoch: rowPfiles.epoch,
    version: toInteger(rowPfiles.version),
    file_type: rowPfiles.file_type,
    sampling_period: rowPfiles.sampling_period,
    data_type: rowPfiles.data_type,
    id_analysis_centers: rowPfiles.id_analysis_centers
  });
CREATE INDEX product_files_id FOR (pfil:product_files) ON (pfil.id);


LOAD CSV WITH HEADERS FROM 'file:///method_identification.csv' AS rowMeth

WITH rowMeth WHERE NOT rowMeth.id IS NULL
MERGE(mt: Method_identification
  {
    id: toInteger(rowMeth.id),
    creation_date: rowMeth.creation_date,
    doi: rowMeth.doi,
    software: rowMeth.software,
    data_type: rowMeth.data_type,
    version: rowMeth.version,
    id_reference_frame: toInteger(rowMeth.id_reference_frame),
    id_analysis_centers: toInteger(rowMeth.id_analysis_centers)

    });

CREATE INDEX method_identification_id FOR (mt:Method_identification) ON (mt.id)
    ;


LOAD CSV WITH HEADERS FROM 'file:///analysis_centers.csv' AS rowACenter
WITH rowACenter WHERE NOT rowACenter.id IS NULL
CREATE(ac: analysis_centers
  {
    id: toInteger(rowACenter.id),
    name: rowACenter.name,
    abbreviation: rowACenter.abbreviation,
    contact: rowACenter.contact,
    email: rowACenter.email,
    url: rowACenter.url

});
```

```
CREATE INDEX analisys_centers_id FOR (ac:analysis_centers) ON (ac.id);
```

Code Snippet 7.1: Code examples to ingest data from csv files.

```

PROFILE
MATCH (n:Estimated_coordinates)
RETURN DISTINCT n.id_station AS id_station
```

Code Snippet 7.2: Materialized view number 1 - query in Cypher query language

```

PROFILE
MATCH (ec:Estimated_coordinates), (mi:Method_identification), (ac:
    analysis_centers), (st:Station), (pf:product_files)
WHERE ec.id_method_identification = mi.id AND mi.id_analysis_centers = ac.id
    AND ec.id_station = st.id AND ec.id_product_files = pf.id
WITH ec, mi, ac, st, pf
SKIP 0
LIMIT 5000000
RETURN DISTINCT ec.id_station, ec.id_method_identification, mi.id, mi.
    creation_date, mi.doi, mi.data_type, mi.version, ac.id, ac.abbreviation, st
    .id, st.marker, pf.sampling_period
ORDER BY s.marker, pf.sampling_period, min(ec.epoch), max(ec.epoch)
```

Code Snippet 7.3: Materialized view number 2 - query in Cypher query language

```
PROFILE
MATCH (ug:User_group_station), (st:Station), (u:User_group)
WHERE ug.id_station = st.id AND ug.id_user_group = u.id
WITH ug, st,  u
RETURN st, ug, u
ORDER BY st.id
```

7.1.2 TileDB

```

 csv_file = "/home/joao/Desktop/project/csvs/method_identification.csv"

 attribute
dtype = {
    "id": "int32",
    "creation_date": "object",
    "doi": "str",
    "software": "str",
    "id_analysis_centers": "int32",
    "id_reference_frame": "int32",
    "version": "float32",
    "data_type": "str"
}


parse_dates = ["creation_date"]

df = pd.read_csv(csv_file, dtype=dtype, parse_dates=parse_dates)


TileDB.from_pandas("method_identification_array", df)

csv_file = "/home/joao/Desktop/project/csvs/estimated_coordinates.csv"

# Define the desired data types for each attribute
dtype = {
    "id": "int32",
    "x": "float32",
    "y": "float32",
    "z": "float32",
    "var_xx": "float32",
    "var_yy": "float32",
    "var_zz": "float32",
    "var_xy": "float32",
    "var_xz": "float32",
    "var_yz": "float32",
    "outlier": "int32",
    "epoch": "str",
    "id_processing_parameters": "int32",
    "id_method_identification": "int32",
    "id_station": "int32",
    "id_product_files": "int32"
}


parse_dates = ["epoch"]

df = pd.read_csv(csv_file, dtype=dtype, parse_dates=parse_dates)


TileDB.from_pandas("estimated_coordinates_array_2", df)


csv_file = "/home/joao/Desktop/project/csvs/reference_frame.csv"

# Define the desired data types for each attribute
dtype = {
    "id": "int32",
    "name": "str",
    "epoch": "str",

}

parse_dates = ["epoch"]


df = pd.read_csv(csv_file, dtype=dtype, parse_dates=parse_dates)

TileDB.from_pandas("reference_frame_array", df)


csv_file = "/home/joao/Desktop/project/csvs/analysis_centers.csv"

# Define the desired data types for each attribute
dtype = {
    "id": "int32",
    "name": "str",
    "abbreviation": "str",
    "contact": "str",
    "email": "str",
    "url": "str"

}

parse_dates = ["epoch"]

df = pd.read_csv(csv_file, dtype=dtype)


TileDB.from_pandas("analysis_cente_array", df)
```

Code Snippet 7.4: Data ingestion from csv files in tileb using Pandas and TileDB library in Python

```
import TileDB
import numpy as np
import time

start_time = time.time()
array_uri = "/home/joao/Desktop/project/estimated_coordinates_array"
array = TileDB.open(array_uri)
print(array.schema)

result = array.query(attrs=["id_station"])


station_ids = np.unique(result[:]["id_station"])
elapsed_time = time.time() - start_time
```

```


for station_id in station_ids:
    print(station_id)


print(elapsed_time)
```

Code Snippet 7.5: Query for materialized view 1 using Python language

7.2 Query Scripts

7.2.1 PostgreSQL

```

Materialized view 1
SELECT DISTINCT estimated_coordinates.id_station
   FROM estimated_coordinates

Materialized view 2
SELECT DISTINCT ec.id_station,
    s.marker,
    ec.id_method_identification AS id_mi,
    ac.id AS id_ac,
    ac.abbreviation,
    mi.data_type,
    mi.version,
    mi.creation_date,
    mi.doi,
    min(ec.epoch) AS min_epoch,
    max(ec.epoch) AS max_epoch,
    pf.sampling_period
   FROM estimated_coordinates ec,
    method_identification mi,
    analysis_centers ac,
    station s,
    product_files pf
  WHERE ec.id_method_identification = mi.id AND mi.id_analysis_centers = ac.id
     AND ec.id_station = s.id AND ec.id_product_files = pf.id
  GROUP BY ec.id_station, ec.id_method_identification, mi.id, mi.creation_date,
       mi.doi, mi.data_type, mi.version, ac.id, ac.abbreviation, s.id, s.marker
      , pf.sampling_period
  ORDER BY s.marker, (min(ec.epoch)), (max(ec.epoch)), pf.sampling_period


Custom estimated_coordinates
SELECT * FROM estimated_coordinates LIMIT 4000000
```

```
Analysis_centers product files
SELECT
    pf.id_analysis_centers,
    pf.url,
    pf.epoch,
    pf.version,
    pf.file_type,
    ac.id,
    ac.name
    FROM product_files pf,
    analysis_centers ac
    WHERE pf.id_analysis_centers = ac.id
    GROUP BY pf.id, pf.url, pf.epoch, pf.version, pf.file_type, ac.name, ac.id
    ORDER BY pf.epoch
    LIMIT 1000000


Item - Item type
SELECT it.id,
    i.id_item_type,
    i.comment,
    it.name
    FROM item as i, item_type as it
    WHERE it.id = i.id_item_type
    GROUP BY it.id, i.id_item_type, i.comment, it.name


Station position
SELECT s.id,
  s.name,
  s.id_station_type,
    ec.x,
  ec.y,
  ec.z,
  ec.epoch,
  ec.id_station
  FROM station s, estimated_coordinates ec
  WHERE s.id = ec.id_station
  GROUP BY s.id, s.name, s.id_station_type, ec.x, ec.y, ec.z, ec.epoch, ec.
      id_station
  ORDER BY ec.epoch
  LIMIT 1000000

User Group query
SELECT DISTINCT s.id,
    s.name,
    s.id_station_type,
    u.id,
    u.name,
    ug.id_user_group
    FROM station s, user_group u, user_group_station ug
    WHERE s.id = ug.id_station and ug.id_user_group = u.id
    GROUP BY s.id, s.name, s.id_station_type, u.id, u.name, ug.id_user_group
```

Code Snippet 7.6: PostgreSQL queries

7.2.2 Neo4J

```
Materialized view 1

MATCH (n:Estimated_coordinates)
RETURN DISTINCT n.id_station AS id_station

Materialized view 2

MATCH (ec:Estimated_coordinates), (mi:Method_identification), (ac:
    analysis_centers), (st:Station), (pf:product_files)
WHERE ec.id_method_identification = mi.id AND mi.id_analysis_centers = ac.id
   AND ec.id_station = st.id AND ec.id_product_files = pf.id
WITH ec, mi, ac, st, pf
SKIP 0
LIMIT 5000000
RETURN DISTINCT ec.id_station, ec.id_method_identification, mi.id, mi.
    creation_date, mi.doi, mi.data_type, mi.version, ac.id, ac.abbreviation, st
    .id, st.marker, pf.sampling_period
ORDER BY s.marker, MIN(epoch), MAX(epoch), pf.sampling_period

Custom estimated_coordinates

MATCH (n:Estimated_coordinates)
WITH n
SKIP 0
LIMIT 500000
RETURN n


Item - Item type

MATCH (i:Item), (it:Item_type)
WHERE i.id_item_type = it.id
WITH i, it
SKIP 0
LIMIT 1000000
RETURN i.id as id, i.comment as comment, it.name as item_name
```

```
Station position

MATCH (s:Station), (ec:Estimated_coordinates)
WHERE s.id = ec.id_station
WITH s, ec
SKIP 0
LIMIT 1000000
RETURN s.id, s.name, s.id_station_type, ec.x, ec.y, ec.z, ec.epoch
ORDER BY ec.epoch

User Group query

MATCH (s:Station), (u:User_group), (us:User_group_station)
WHERE s.id = us.id_station AND us.id_user_group = u.id
WITH s, u, us
RETURN DISTINCT s.id, s.name, s.id_station_type, u.id, u.name
```

Code Snippet 7.7: Cypher Queries

7.2.3 TileDB

```
Materialized view 1

start_time = time.time()


db = TileDB.sql.connect(init_command="SET GLOBAL time_zone='+00:00'")
for chunks in pd.read_sql(sql="select distinct id_station from `
    estimated_coordinates_array`", con=db, chunksize=100):
        print(chunks)


elapsed_time = time.time() - start_time

print("Elapsed  time: "+str(elapsed_time*1000)+" miliseconds")

Materialized view 2

db = TileDB.sql.connect(init_command="SET GLOBAL time_zone='+00:00'")
for chunk in pd.read_sql(sql='SELECT DISTINCT ec.id_station, s.marker,'
'ec.id_method_identification AS id_mi,'
'ac.id AS id_ac, ac.abbreviation,'
'mi.data_type,'
'mi.version,'
'mi.creation_date,'
'mi.doi,min(ec.epoch) AS min_epoch,'
'max(ec.epoch) AS max_epoch,'
'pf.sampling_period '
'FROM `estimated_coordinates_array` ec, `method_identification_array` mi, `
    analysis_centers_array` ac, `stations_array` s, product_files` pf '
'WHERE ec.id_method_identification = mi.id AND mi.id_analysis_centers = ac.id
    AND ec.id_station = s.id AND c.id_product_files = pf.id '
'GROUP BY ec.id_station, ec.id_method_identification, mi.id, mi.creation_date,
    mi.doi, mi.data_type, mi.version, ac.id, c.abbreviation, s.id, s.marker, pf
    .sampling_period '
'ORDER BY s.marker, (min(ec.epoch)), (max(ec.epoch)), pf.sampling_period LIMIT
    1000', con=db, chunksize=10000):
    print(chunk)

elapsed_time = time.time() - start_time

print(elapsed_time*1000)


Custom estimated_coordinates

start_time = time.time()

db = TileDB.sql.connect(init_command="SET GLOBAL time_zone='+00:00'")

for chunk in pd.read_sql(sql='SELECT * FROM
`estimated_coordinates_array`
LIMIT 4000000', con=db, chunksize=10000):
    print(chunk)

elapsed_time = time.time() - start_time

Analysis_centers product files

start_time = time.time()

db = TileDB.sql.connect(init_command="SET GLOBAL time_zone='+00:00'")


for chunk in pd.read_sql(sql='SELECT pf.id_analysis_centers,'
'pf.url,'
'pf.epoch,'
'pf.version,'
'pf.file_type,'
'ac.id,'
'ac.name '
'FROM `product_files` pf, `analysis_centers_array` ac '
'WHERE pf.id_analysis_centers = ac.id '
'GROUP BY pf.id, pf.url, pf.epoch, pf.version, pf.file_type, ac.name, ac.id '
'ORDER BY pf.epoch '
'LIMIT 1000000',
con=db, chunksize=10000):
    print(chunk)


elapsed_time = time.time() - start_time

print(elapsed_time*1000)


Item - Item type


start_time = time.time()

db = TileDB.sql.connect(init_command="SET GLOBAL time_zone='+00:00'")


for chunk in pd.read_sql(sql='SELECT it.id,'
'i.id_item_type,'
'i.comment,'
'it.name '
'FROM `item_array` as i, `item_type_array` as it '
'WHERE it.id = i.id_item_type '
'GROUP BY it.id, i.id_item_type, i.comment, it.name',
con=db, chunksize=100):
    print(chunk)


elapsed_time = time.time() - start_time

print(elapsed_time*1000)


Station position


start_time = time.time()

db = TileDB.sql.connect(init_command="SET GLOBAL time_zone='+00:00'")


for chunk in pd.read_sql(sql='SELECT SELECT s.id,'
  's.name,'
  's.id_station_type,'
    'ec.x,'
  'ec.y,'
  'ec.z,'
  'ec.epoch,'
  'ec.id_station'
  'FROM station s, estimated_coordinates ec'
```

```
    'WHERE s.id = ec.id_station'
    'GROUP BY s.id, s.name, s.id_station_type, ec.x, ec.y, ec.z, ec.epoch, ec.
        id_station'
    'ORDER BY ec.epoch'
    'LIMIT 1000000'
con=db, chunksize=100):
    print(chunk)


elapsed_time = time.time() - start_time

print(elapsed_time*1000)


User Group query
```

Bibliography

[1] Bruce Momjian. *PostgreSQL: introduction and concepts*, volume 192. Addison-Wesley New York, 2001.

[2] Jim Webber. A programmatic introduction to neo4j. In *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*, pages 217–218, 2012.

[3] Stavros Papadopoulos, Kushal Datta, Samuel Madden, and Timothy Mattson. The tiledb array data storage manager. *Proceedings of the VLDB Endowment*, 10(4):349–360, 2016.

[4] Ewald Geschwinde and Hans-Jürgen Schönig. *PHP and PostgreSQL: Advanced Web Programming*. Sams Publishing, 2002.

[5] Puyam S Singh, Dibyajyoti Chutia, and Singuluri Sudhakar. Development of a web based gis application for spatial natural resources information system using effective open source software and standards. 2012.

[6] Rakesh Chandra and Arie Segev. *Managing temporal financial data in extensible databases*. PhD thesis, University of California, Berkeley, 1994.

[7] Xiufeng Liu and Xiaofeng Luo. A data warehouse solution for e-government. *International Journal of Research and Reviews in Applied Sciences*, 4(1):101–105, 2010.

[8] Łukasz Warchał. Using neo4j graph database in social network analysis. *Studia Informatica*, 33(2A):271–279, 2012.

[9] Christian Theil Have and Lars Juhl Jensen. Are graph databases ready for bioinformatics? *Bioinformatics*, 29(24):3107, 2013.

[10] Pengcheng Liu, Yinliang Huang, Ping Wang, Qifan Zhao, Juan Nie, Yuyang Tang, Lei Sun, Hailei Wang, Xuelian Wu, and Wenbo Li. Construction of typhoon disaster knowledge graph based on graph database neo4j. In *2020 Chinese Control And Decision Conference (CCDC)*, pages 3612–3616. IEEE, 2020.

[11] Emsaieb Geepalla and Salwa Asharif. Analysis of physical access control system for understanding users behavior and anomaly detection using neo4j. In *Proceedings of the 6th International Conference on Engineering & MIS 2020*, pages 1–6, 2020.

[12] Aleksa Vukotic, Nicki Watt, Tareq Abedrabbo, Dominic Fox, and Jonas Partner. *Neo4j in action*, volume 22. Manning Shelter Island, 2015.

[13] Arun Kumar, Matthias Boehm, and Jun Yang. Data management in machine learning: Challenges, techniques, and systems. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1717–1722, 2017.

[14] Xiuwen Zheng, Stephanie M Gogarten, Michael Lawrence, Adrienne Stilp, Matthew P Conomos, Bruce S Weir, Cathy Laurie, and David Levine. Seqarray—a storage-efficient high-performance data format for wgs variant calls. *Bioinformatics*, 33(15):2251–2257, 2017.

[15] Ramon Antonio Rodriges Zalipynis. Chronosdb: distributed, file based, geospatial array dbms. *Proceedings of the VLDB Endowment*, 11(10):1247–1261, 2018.

[16] Ramon Antonio Rodriges Zalipynis. Generic distributed in situ aggregation for earth remote sensing imagery. In *International Conference on Analysis of Images, Social Networks and Texts*, pages 331–342. Springer, 2018.

[17] Hirokatsu Kataoka, Asato Matsumoto, Ryosuke Yamada, Yutaka Satoh, Eisuke Yamagata, and Nakamasa Inoue. Formula-driven supervised learning with recursive tiling patterns. In *Proceedings of the IEEE/CVF International Conference on Computer Vision*, pages 4098–4105, 2021.