EXAMENSARBETE INOM TEKNIK,
GRUNDNIVÅ, 15 HP

STOCKHOLM, *SVERIGE 2021*

Comparing database optimisation techniques in PostgreSQL

Indexes, query writing and the query optimiser

ELIZABETH INERSJÖ

KTH
SKOLAN FÖR ELEKTROTEKNIK OCH DATAVETENSKAP

Abstract

Databases are all around us, and ensuring their efficiency is of great importance. Database optimisation has many parts and many methods, two of these parts are database tuning and database optimisation. These can then further be split into methods such as indexing. These indexing techniques have been studied and compared between Database Management Systems (DBMSs) to see how much they can improve the execution time for queries. And many guides have been written on how to implement query optimisation and indexes. In this thesis, the question "How does indexing and query optimisation affect response time in PostgreSQL?" is posed, and was answered by investigating these previous studies and theory to find different optimisation techniques and compare them to each other. The purpose of this research was to provide more information about how optimisation techniques can be implemented and map out when what method should be used. This was partly done to provide learning material for students, but also people who are starting to learn PostgreSQL. This was done through a literature study, and an experiment performed on a database with different table sizes to see how the optimisation scales to larger systems.

What was found was that there are many use cases to optimisation that mainly depend on the query performed and the type of data. From both the literature study and the experiment, the main take-away points are that indexes can vastly improve performance, but if used incorrectly can also slow it. The main use cases for indexes are for short queries and also for queries using spatio-temporal data - although spatio-temporal data should be researched more. Using the DBMS optimiser did not show any difference in execution time for queries, while correctly implemented query tuning techniques also vastly improved execution time. The main use cases for query tuning are for long queries and nested queries. Although, most systems benefit from some sort of query tuning, as it does not have to cost much in terms of memory or CPU cycles, in comparison to how indexes add additional overhead and need some memory. Implementing proper optimisation techniques could improve both costs, and help with environmental sustainability by more effectively utilising resources.

Keywords

PostgreSQL, Query optimisation, Query tuning, Database indexing, Database tuning, DBMS

Sammanfattning

Databaser finns överallt omkring oss, och att ha effektiva databaser är mycket viktigt. Databasoptimering har många olika delar, varav två av dem är databasjustering och SQL optimering. Dessa två delar kan även delas upp i flera metoder, så som indexering. Indexeringsmetoder har studerats tidigare, och även jämförts mellan DBMS (Database Management System), för att se hur mycket ett index kan förbättra prestanda. Det har även skrivits många böcker om hur man kan implementera index och SQL optimering. I denna kandidatuppsats ställs frågan "Hur påverkar indexering och SQL optimering prestanda i PostgreSQL?". Detta besvaras genom att undersöka tidigare experiment och böcker, för att hitta olika optimeringstekniker och jämföra dem med varandra. Syftet med detta arbete var att implementera och kartlägga var och när dessa metoder kan användas, för att hjälpa studenter och folk som vill lära sig om PostgreSQL. Detta gjordes genom att utföra en litteraturstudie och ett experiment på en databas med olika tabell storlekar, för att kunna se hur dessa metoder skalas till större system.

Resultatet visar att det finns många olika användingsområden för optimering, som beror på SQL-frågor och datatypen i databasen. Från både litteraturstudien och experimentet visade resultatet att indexering kan förbättra prestanda till olika grader, i vissa fall väldigt mycket. Men om de implementeras fel kan prestandan bli värre. De huvudsakliga användingsområdena för indexering är för korta SQL-frågor och för databaser som använder tid- och rum-data - dock bör tid- och rum-data undersökas mer. Att använda databassystemets optimerare visade ingen förbättring eller försämring, medan en korrekt omskrivning av en SQL fråga kunde förbättra prestandan mycket. The huvudsakliga användi-ngsområdet för omskriving av SQL-frågor är för långa SQL-frågor och för nestlade SQL-frågor. Dock så kan många system ha nytta av att skriva om SQL-frågor för prestanda, eftersom att det kan kosta väldigt lite när det kommer till minne och CPU. Till skillnad från indexering som behöver mer minne och skapar så-kallad överhead". Att implementera optimeringstekniker kan förbättra både driftkostnad och hjälpa med hållbarhetsutveckling, genom att mer effektivt använda resuser.

Nyckelord

PostgreSQL, SQL optimering, DBMS, SQL justering, Databasoptimering, Indexering

Acknowledgements

I would like to thank Leif Lindbäck, the supervisor for this thesis, for making this thesis possible. You helped me a lot with the planning and narrowing down of the ideas, as well as provided me with an examiner.

I also would like to thank Thomas Sjöland for agreeing to be my examiner.

Lastly, I would like to thank my friend for helping me by answering questions about report structure, and proofreading.

Thank you.

Contents

List of Figures

List of acronyms and abbreviations

BRIN Block Range Index

CD Compact Disk

CPU Central Processing Unit

DAG Directed A-cyclical Graph

DBMS Database Management System

DDL Data Definition Language

DML Data Management Language

GIN Generalised Inverted Index

GiST Generalised Search Tree

HDD Hard Disk Drive

I/O Input/Output

ID Identity Document

MCV Most Common Value

MVCC Multi-Version Concurrency Control

RAM Random Access Memory

SP-GiST Space Partitioned Generalised Search Tree

SQL Structured Query Language

SSD Solid State Drive

Chapter 1

Introduction

Traditionally, a database is a collection of related data that has inherent meaning. What does this mean? For example, in a university, the database keeps track of all the students registered to the university, their courses, and other things related to the students and the university. This data can be stored in different ways, like in a file or an excel sheet. Therefore, the database is the information in it, and what the data's value is in the real world [1, pg.3]. The database needs to represent aspects of the real world. These aspects that build up the database are called a miniworld. Changes that happen in the miniworld need to be reflected in the database. The database also has other defining traits such as the data it contains need to have logical coherence and inherent meaning. As well as a purpose. A database cannot exist without being used, as its purpose is to store data that can be retrieved, and for the database to have meaning it needs to reflect changes that happen to its miniworld [1, pg.4-5].

Databases have had and continue to have an important role in many areas that involve computers. It can even be said that databases have had a major impact on the growth of computer usage [1, pg.3-4]. They are used in many areas, such as business, social media, and medicine as a notable few. Even normal everyday actions like bank transactions or shopping most likely have a database backing them. For example by subtracting from shelf-inventory in the store at check-out or accessing your bank account to see how much money you have on your card. Another example of how prevalent databases are in our everyday life is that most websites have a database backing them. This can be explained by the three-tier model, the client tier, which contains the internet, applications, and the users. The middle tier, which contains webservers, scripts, and a scripting engine. And the database tier, which contains the database and the DBMSs (Database Management System) which

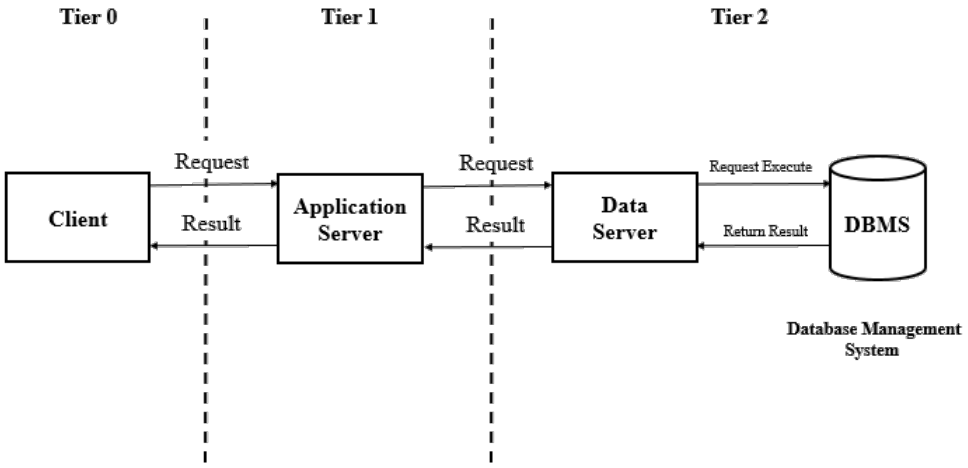

Figure 1.1: The three tier database design.
[3]

is used to handle the database [2]. The three-tier architecture can be seen in Figure 1.1.

What does this look like in practice? Whenever a user requests a website, the request gets sent to the webserver that requests the database to retrieve or operate on necessary data, and then finally display the results to the user [2]. Applying this logic to a social media application, logging into an account requires access to a database, loading the post history, or even message history also needs access to a database. The database is used to store the related data and efficiently retrieve it [1, pg.4].

1.1 Background

Now that it has been concluded that databases are all around us and used in variety of situations, it would be very noticeable if they were slow. This is due to how it takes just three seconds for users to drop a website if it's still loading, according to Sitechecker [4], which is a company that offers resources to analyse statistics on web pages. Their target audience is other companies that have some type of web traffic to monitor, and offer customer stories and ratings to prove that the product they are selling is reliable.

Databases are often connected to applications - which are called database applications [1, pg.9] - such as for social media. As development has brought us faster and faster internet, internet speed can no longer be blamed for slow

access to information [5]. Therefore, it is important to maintain efficient software, to have speedy responses for a good user experience. But how do we optimise database systems for efficiency? And what is a database system?

A database system is the combination of a database and a DBMS. The DBMS is a database software program that is often used to control the database [6]. It generally serves as an interface between the database and its users by performing the needed operations on the database and then presenting the result. It is in the DBMS that performance monitoring and tuning takes place to optimise the database. The DBMS uses Structured Query Language (SQL) queries to communicate with the database from the user interface [6]. The two main categories of database system optimisation are database tuning, which deals with the database hardware and design. As well as query optimisation, which mostly deals with ensuring how queries are performed in the database, which is why knowledge over SQL is important [1, pg.541, 655].

1.2 Problem

There are several methods to optimising a database system, as stated in the introduction, ensuring efficiency and speed is important for many different reasons. But as there are many methods of optimisation, which ones should be used? That is a question that this thesis aims to provide a starting point for. Having a compiled document with methods, their use cases, and how efficient they are in practice could simplify the process of choosing methods. PostgreSQL specifically is a popular open-source DBMS and providing more information to the community could be valuable.

The research question is as follows:

- How do indexing and query optimisation affect response time for a PostgreSQL database?

1.3 Purpose

The purpose of this report is to describe and compare different methods for optimising database systems. The purpose of the project is to develop an understanding of how database tuning and query optimisation operate. It is also to create material that can be used for teaching purposes in database courses. This report should be able to lie as a starting point for further experimentation and research.

1.4 Sustainability and ethics

It can be argued that optimising a database system has an environmental effect as it reduces the resources a database uses. Shorter response time and efficient use of hardware lead to lessening the total computing time and could reduce the wear on hardware as well as a reduction in energy usage.

An ethical problem that is related to database efficiency, is the potential that people more easily can manage to compile data from different data sets. This can then be presented or used to discern information that causes privacy issues.

1.5 Research Methodology

Firstly, a literature study is performed to identify methods for database tuning and query optimisation, and their different use cases. As well as to find research that also does these comparisons, to have as a basis for the experiment and conclusions. The study is of qualitative nature, as information that is chosen to be presented is based on what can be found, some areas might have more information and some less. Every source was carefully examined for relevance and trustworthiness.

After that, the experiment is planned, in part using the information found in the literature study so that a meaningful comparison can be made. The use cases for the methods are analysed to see if there is an overlap. Lastly, data is obtained for evaluating the methods by performing an experiment. The result is compared to the results from the literature study and is compiled in a way that answers the research question.

1.6 Delimitations

Only a couple of optimisation methods are chosen to study in detail, these methods are chosen based on the availability of information and the delimitations of the performed experiment. The chosen areas are database indexing - where indexes are chosen based on the available data - using the PostgreSQL optimiser, as well as query tuning.

The delimitations of the experiment are to use PostgreSQL for the database system and as a query language, the methods evaluated are limited to software improvement. The database has a simple design but contains much data, and the number of queries, indexes, and query improvements are based on the

information found, and limited to a couple of methods. The chosen methods are based on found information and best suited for the data types used in the database. These delimitations are chosen to get precise data and to ensure that the project will be finished in the amount of time specified for it.

1.7 Structure of the thesis

Chapter two presents the relevant theoretical background to understand the rest of the report. As well as introduces the findings from related studies.

Chapter three describes the research methods used.

Chapter four describes the experiment parameters and how it was performed.

Chapter five compiles the results for the experiment and the literature study.

Chapter six discusses the result and the evaluation of the result and methods.

Chapter seven contains the conclusion, answers to the research question posed, and reflections about the work.

Chapter 2

Background

This chapter provides the basic information needed to understand the rest of the report, as well as some related works for the literature study. It starts with briefly going over some basics for SQL and database systems and then moves on to describing memory aspects of database and indexes to provide a background for tuning. As well as explaining what query optimisation is, before moving on to the related works.

2.1 Database systems

The introductory chapter briefly describes a database system as the combination of a DBMS and the database. The more detailed description of its parts is as follows.

2.1.1 Relational databases

A relational database stores and organises data in tables that are linked based on related data. The purpose of this is to ensure the ability to create a new table from data in multiple tables with a single query. It can also help with understanding how data is related, which could lead to improving decision-making and help identify opportunities. The tables consist of fields (columns) and the set of related data (rows) [7].

The main benefit of using relational databases is that it reduces redundancy and through that reduces the risk for insert, update, and delete anomalies. Reduced redundancy means that, in many cases, information only appears in one table and only once. Reducing redundancy often happens during the planning stages of a database, and is done by a database designer. The

process of doing this is called normalisation. The database designer often uses database schemas to start off building the database. A database schema is the structure of the database defined by formal SQL [7].

2.1.2 Database management systems

The DBMS is a program that is used to create and maintain a database. It also simplifies the process of defining, manipulating, and sharing a database with multiple users and applications. Defining the database specifies the constraints around it. What data types? What data structures are involved? What are some data constraints? Are all questions that are asked during this stage of the process. This information is generally stored as meta-data in the DBMS's catalogue - which is used by the DBMS software and database users to get information about the database's structure. This is done because of how a general-purpose DBMS is not customised for a database application, so the software needs to refer to the meta-data to find out what the structure is like. Constructing the database means storing data in a way that the DBMS can control, and sharing the database means that multiple users and/or applications can access and use the database concurrently [1, pg.5-10]. Other aspects that define a DBMS are insulation and the ability to have multiple views over the data. Insulation is an aspect that ensures that the structure of data - that is stored by the DBMS - when changed, does not affect how the program works. This is called program-data independence. The ability to have multiple views over data means that data from tables can be manipulated and put together with other tables to create other views over it. Another important database definition is the ability to reduce redundancy. Although in some cases, controlled redundancy can be used to improve query performance. The act of reintroducing redundancy into a database is called denormalisation [1, pg.10-12, 18].

The DBMS is what is used to optimise the database. This can be done through the handling of effective query processing - i.e how queries are executed and how data is fetched etc. Tuning hardware and creating indexes is done because of how the database often is stored on disk. This means that the DBMS needs to use special data structures, data types, and search techniques to quickly find the data that the query is requesting. The most common way to do this is by using indexes, as when a query is executed data needs to be retrieved from disk to main memory for processing. The entire purpose of indexes is to improve the search process for finding and retrieving data. There are other ways to improve this as well, such as by tuning the hardware or

switching to more efficient parts. For example, the DBMS often uses caching and buffers to improve performance. Caching means that the data retrieved from disk is stored for a while - there are different methods to decide for how long - with the prediction that it might be used again. This speeds up the process as if the cached data gets used again the Central Processing Unit (CPU) does not need to wait for retrieval from disk and can just use the cache instead. The buffer helps to pipeline the process of retrieving data from disk to main memory, it ensures that while the CPU works on data, the next data set can get loaded into the buffer, so when the CPU is done it can immediately get the new data. This is especially helpful if more data needs to be fetched than what can fit in main memory [1, pg.20, 541-558].

The DBMS consists of multiple parts. One of them is the query optimiser, which ensures that an appropriately effective execution plan is chosen for every query, based on some variables, such as storage system and indexes. The execution plan is the code that is built for the query, which decides what order different aspects of the query get executed in [1, pg.655-658]. This will be described further later on in this chapter.

2.2 Structured query language

SQL is the standard language for a relational DBMSs. It is a database language that has statements for data definitions, queries, and updates, hence it is both Data Definition Language (DDL) and Data Management Language (DML) [1, pg.178]. DDL means that the query language can deal with database schemas, their descriptions, and how the data resides in the database. DML on the other hand deals with the manipulation of data in the database, it consists of the most common SQL operations [8]. The query language is used to build the database schemas, query the relational database, and manage the database [1, ch.6].

A database schema describes the organisation and structure of the database. It contains all the database objects, such as tables, and can be visualised as the tables, their attributes, and how they are related to each other. In some DBMSs a database and a schema are equivalent and in others it is not [9]. A good comparison for this can be that the database schema can be seen as a java class, while the database objects are the methods in the class.

2.2.1 Relational algebra

Relational algebra provides a formal foundation for the relational model operations and is used as a basis to implement and optimise queries. It defines

a set of operations that can be used on a relational model. Most relational systems are based on relational algebra and some concepts are defined in SQL. Therefore, a query can be translated into a sequence of relational algebra operations, also called a relational algebra expression [1, ch.8].

It is assumed that the readers are familiar with relational algebra, which means the report will not go into detail about it.

2.2.2 PostgreSQL

PostgreSQL is an open-source object-relational database system that uses SQL, and offers features such as foreign keys - reference keys that link tables together - updatable views and more [10]. Views will be described in the next subsection.

PostgreSQL can also be extended by its users by adding new data types, functions, index methods, and more [10]. Its architecture is a client/server model, and a session consists of a server process - that manage database files, accepts connections to the database from the client-side, and performs database operations requested by the clients. And the client application that requests database actions for the server to perform. Like a typical client/server application, the server and client do not need to be connected to the same network and can communicate through normal internet procedures. This is important to keep in mind as files on the client-side might not be accessible on the server-side. PostgreSQL can handle multiple client connections to its servers [11] as most servers can.

Earlier it was mentioned that PostgreSQL is a relational database management system. This means that it is a system for managing data stored in relations - the mathematical term for a table. There are multiple ways of organising databases [12], but relational databases are what is the focus of this report. Each table in a relational database system contains a collection of named rows, and each row has a collection of named columns that contain a specified data type. These tables are then grouped into database schemas. There can be multiple databases in one server, just like there can be multiple schemas in a database. A collection of databases managed by one PostgreSQL server is called a database cluster [12]. Another aspect of PostgreSQL is that it supports automatic handling of foreign keys, through accepting or rejecting the value depending on its uniqueness. This means that PostgreSQL will warn if the value in the foreign key column is not unique, which is done to maintain the referential integrity of the data. The behaviour of the foreign key can be tuned to the application by the developer [13], this can be done through specifying

deletion of referenced objects, the order of deletion, and other things [14].

2.2.3 Queries

Here some query concepts used in the experiment will be explained.

Query operations

Two of the query operations that are used in the experiment need some closer examination. The LIKE and IN operations. To do this, the PostgreSQL tutorial's website is used. PostgreSQL tutorial is a website dedicated to teaching PostgreSQL concepts. They show examples and explanations of how to use operations and build a database [15].

The LIKE operation is used to pattern match strings to each other. This can be done using wildcards, which in PostgreSQL is '%' for any sequence of characters and '_' for any single character. A wildcard is used for pattern matching, as stated before. For example, matching the string 'Jen%' could give any string that starts with 'Jen'. While using 'Jen_' could match any string starting with 'Jen' and then a single character after [16].

The IN operator is used to match any string within a list of values. It does this by returning true if the comparing string matches one of the stated values in the IN operation. It is the equivalence of using equals and OR operations, although PostgreSQL executes the IN queries faster than the OR queries [17].

Nested queries

A query that executes multiple queries in one contains an inner query - also called a subquery - and an outer query [18]. Often these types of queries can be split into multiple separate queries. PostgreSQL executes these queries by first, executing the inner query, then getting the result and passing it to the outer query. Lastly, it executes the outer query [18].

A correlated inner query is evaluated for each row that is processed by the outer query, which differs from how a normal nested query executes according to Geeks for Geeks, a website dedicated to learning programming languages through examples [19]. As mentioned in the paragraph earlier, in a normal nested query the inner query gets executed first and then the outer query. It can also be said that the correlated query is driven by the outer query as the result of the inner query is dependent on the outer query [19]. This works similarly to how nested loops work in any other programming language.

2.2.4 Views and materialised views

A view is a named query that is often useful to have for queries that are run often. It is a key aspect of a good SQL database design. Views can be used in almost any place a real table can, and it is possible to build multiple views on each other [20]. Although, it is important to note that views are not stored as tables, and are instead stored as references to the queries. This means that every time a view is called on, the query that it is based on is executed [21].

The materialised view uses the same system as a view does but stores the result like a table. The main difference between a materialised view and a table is that the materialised view cannot be updated. Instead, the query that creates the materialised view is stored, so that it can be refreshed when the data needs to be updated. The data is often faster to access through a materialised view than a table, which can be useful in many cases even if the data is not entirely up to date [22].

2.3 Database tuning

The goal of database tuning is to dynamically evaluate the requirements - sometimes periodically - and to reorganise indexes and the file order to gain the best over-all performance. This makes changes to the database and its structure through normalisation or denormalisation, indexes, and the hardware aspect of the database - such as how files are physically ordered on disk, optimising Input/Output (I/O) operations, hardware upgrades et cetera [1, pg.459-461, 640].

Normalisation, denormalisation, and some aspects of hardware are outside of the scope of this report and will not be discussed further but some memory aspects are important to be aware of, this is discussed in the next subsection.

2.3.1 Database memory

A database is often too large to store in main memory, thus to manage performance a basic understanding of how database hardware works are necess-ary. The memory structure of a database is usually separated into three parts [1, pg.542]. The primary storage, which is what the CPU uses when executing operations. The secondary storage most usually consists of Hard Disk Drives (HDDs) or Solid State Drives (SSDs), and lastly the tertiary storage, which is offline storage such as Compact Disks (CDs) and magnetic tapes. The most important aspect for optimisation of memory access

is bringing data to the primary storage from the secondary storage, for the execution of operations on the database. In some cases, a database can be stored in the primary memory - a so-called main memory database - this is often done for real-time applications. But because databases often store persistent data, some of which needs to be read or handled multiple times while it is stored, it needs to use secondary storage. The databases are also generally too big to store on a single disk which means that multiple disks need to be used, and the benefits of secondary storage hardware often outweigh the benefits of the primary storage ones [1, pg.542-544].

Typically, the database application only needs small amounts of data to process from the database, hence, the data needs to be accessed on disk and effectively moved to main memory to increase the speed of execution. As mentioned earlier this is partly done through hardware by the use of buffers, as there is a noticeable difference between how quickly the CPU can process data and the moving of data from disk to main memory. Other ways to do this require a basic understanding of how the data is stored in the database and the hardware.

The data on disks are stored as something called files of records, in which a record is a set of data values that describe entities, their attributes, and relations - i.e a table [1, pg.560]. Files of records are often stored in data blocks - also called a page - which are fixed sizes of storage on a disk. This is important to note as the transmission of data from disk to main memory usually is done on a per-block basis. By physically storing data in contiguous blocks on disk performance can be improved as it puts related data near each other, which can prevent the arm on the disk (HDD) from having to move longer distances. This can be further improved by prediction, which is done through reading multiple blocks of data at once and putting it in main memory. This can reduce the search time on disk access. It only works if the application is likely to need consecutive blocks and the ordering of the file organisation allows it, though [1, pg.561-563].

How files are ordered in memory can be done in different ways. Storing the files in a specific order is called the file organisation [23], and it can be described as the auxiliary relationship between the records that build up the file. It is used to identify and access any given record [1, pg.545-546]. In the database, there are two ways to store files, the primary file organisation and the secondary file organisation. The primary file organisation decides how file records are physically placed on disk. This is done by using different data structures such as heaps, hash structures, and B-trees. For example, a heap file would not store the records in any particular order and instead place them as a

heap would order them. Unlike the primary file organisation, the secondary file organisation is a logical access structure that improves the access to file records based on other fields than what is used for the primary file organisation. This is often done through indexing [1, pg.545-546, 604-611].

There can be different types of records in a file, the type is decided by the collection of field names and their corresponding data types contained in the record. This means that records in files can be constant or of variable length. If a file has variable length records it can affect indexing and search algorithms efficiency. This is due to the way files consist of sequences of records. By having a constant length on records it is simpler to calculate the start of each field in a record based on the relative starting point of the record in the file. Therefore, algorithms handling variable-length records often need to be more complex, which can affect the speed of execution [1, pg.560-561]. The different ways of how variable-length files can look are as follows:

- The file record is of the same type but one or more of the fields have different sizes.
- The file record is of the same type but one or more of the fields have multiple values for each record, this is called a repeating field.
- The file record is of the same type but one or more of the fields are not mandatory.
- The file contains one or more records of different record types, this leads to the records being of different sizes. This often happens in clusters of related records.

[6, pg.]60-5611

As mentioned earlier, there are heap files and ordered files, which are the main ways of storing records on a file. The heap files store records in a heap structure, while the ordered files can use many different data structures for storage. The main benefit of using ordered files is that other search algorithms than linear search can be used when searching for a record. Although, ordered files are rarely used unless a primary index is implemented [1, pg.567-572]. The main data structures implemented for ordered files are hash tables, hash maps, and B-trees, which each have their pros and cons and are chosen depending on what the file is used for [1, pg.583]. These data structures are described in more detail later on in this chapter.

2.3.2 Indexing

An index is a supplementary access structure. It is used to quickly find and retrieve a record based on specified requirements. They are stored as files on disk and contain a secondary access path to reach records without having to physically order the files [1, pg.601-602]. Without an index, a query would have to scan an entire table to find the entries it is searching for. In a big table, having to go through every element sequentially would be very inefficient, especially in comparison to using an index. For example in a b-tree index, the search would only need to go a couple of levels deep in the tree [1, pg.601-602]. The index is handled by the DBMS in PostgreSQL. Which in part handles the updates for the index when a table changes. The downside to using indexes is that updating them as the tables change adds an overhead to the data manipulation operations. This means that updating a table indirectly adds to the execution time of the data manipulation operations [24], which is an important aspect to keep in mind when deciding if an index should be built on a table or not [1, pg.601].

The indexes are based on an index field, which can be any field in a file or multiple fields in the file. Multiple indexes can also be created on the same file. As mentioned earlier, indexes are data structures used to improve search performance. Therefore, many data structures can be used to construct them. The data structure is chosen depending on many different factors. One such factor is what queries are predicted to be used on the index. Indexes can be separated into two main areas, single-level indexing and multilevel indexing [1, pg.601], which will be described below.

Single-level indexes

Single-level indexing using ordered elements has the same idea as a book index, which has a text title and the page it can be found on. This can be compared to how the index has the index field and the field containing the pointers to where the data can be found on disk. The index field used for building the index on a file - with multiple fields - with a specified record structure, is usually only based on one field. Like earlier mentioned, the index stores the index field and a list of pointers to each disk block that contains a record with the same index field. The values - index fields - in an ordered index are also sorted so that a binary search can be performed to quickly find the desired data. How efficient is this? Well, if a comparison is made in the case of having both the data file and the index file sorted, the index file is often smaller than the data file. This means that searching through the index is still

faster than through the data file [1, pg.602].

As stated in the background, the index types are often separated into primary and secondary indexes. The single-level index can be either of these types [1, pg.602].

A primary index is a file containing ordered keys for a sorted file record. The primary index is used to physically order data on disk, which means that a primary index can only be a single-level index and that there can only be one primary index on a table. The field for the key is used to physically order the files, each record must have a unique value for that to be possible. The primary index only contains two fields, as stated earlier, which makes it effective for searching for data records in a file. The first field is a primary key and the second field is a pointer to a block address on disk. There is one index entry for each block in the data file. Although, a primary index does not have to use a key for the ordering field, and if it does not use a key it is called a clustered index instead [1, pg.602.605].

Indexes can also be defined as compact or sparse indexes. A sparse index has fewer entries than there are records on a file, which by definition makes a primary index a sparse index. The main issue with a primary index - as is the issue for most sorted data structures - is the insertion and deletion of elements. For example, inserting a new element in a filled array requires expansion of the array, and in a linked list, searching for where to insert the element takes time. Cluster indexes are used to quickly find groups of data. It is also an ordered index that has to deal with the issues of insertion and deletion of records. To solve this, clustered indexes often reserve space in blocks for insertion. Both cluster and primary indexes assume that the field for physical ordering of records on disk is the same as the index field [1, pg.602-605].

A secondary index offers a second logical ordering alternative for accessing a file when a primary option already exists. The records on the data file can be ordered, unordered, or hashed, as it does not deal with the physical ordering of records. The secondary index is also an ordered file with two fields, like a primary index. But it is created on a field with a candidate key or that has a unique value in each record. A candidate key is a field that could be a primary key, and a primary key is a field - or fields - that can be used to uniquely identify a row. This can be done by using counters, but also through other means. There can be multiple candidate keys, but only one primary key, which means that multiple secondary indexes can be created for the same file. In practice, it just adds access paths to the file based on different fields. Secondary indexes often take more memory space than primary indexes, although searching for arbitrary records is noticeably quicker [1, pg.609-611].

Multi-level indexes

The idea behind a multilevel index is to reduce the part of the index that is searched with the block factor (bfri) - also called the fan-out (fo) - for the index. During a multilevel index search, the area that is searched is reduced by fo, which if larger than two makes it more efficient than binary search. The way the multi-level index works is by viewing the index file as an ordered file with a distinct value for each entry. The index file counts as the first level of the multi-level index and the second level is defined as the primary index that is created on the first level. A block anchor is created for the second level so that it has an entry for each block of the first level. The block factor remains the same for every level of the multi-level index as the size for entries remains the same - a field value and a block address. This process is then be repeated, level three is another primary index created on the second level, et cetera. More levels are only needed if a level needs more than one block for storage as each level reduces the number of entries by a factor of fo, this means each level requires less storage. This also means that only one disk block is accessed per level, thus, for a multi-level index with t levels only t disk blocks are accessed during a search. Which increases the speed of searches. Lastly, the last level of the index is called the top index level, and the multi-level index can use primary, secondary and cluster indexes [1, pg.613-614]. Multi-level indexes still suffer from the issues of insertion and deletion of records. Dynamic multi-level indexes aim to solve this by leaving space in blocks for insertion of new entries and using appropriate insertion/deletion algorithms for creating/deleting index blocks when the data file grows/shrinks. This is often done by using B+-trees - which functions like a B-tree but has its leaf nodes connected as well - as a data structure [1, pg.613-614].

2.3.3 Index types

PostgreSQL provides multiple index types, among them are B-trees, Hash structures, Generalised Search Tree (GiST), Space Partitioned Generalised Search Tree (SP-GiST), Generalised Inverted Index (GIN) and Block Range Index (BRIN). The index types use different algorithms that are better suited for different types of queries. The B-tree usually suits the broadest range of queries which is why the default index type used for PostgreSQL is the B-tree [25].

B-trees

B-trees are balanced search trees that are useful for equality and range queries on data that can be ordered [25]. The PostgreSQL query planner will consider using a B-tree if any comparison operator is used in the query. B-tree indexes are also useful for retrieving data in sorted order, due to the nature of the B-trees [25]. PostgreSQL also supports multi-column B-trees. They are most effective when there are constraints on the leading columns but can be used for any subset of the index's columns. The rule is that when an equality constraint is used in the leading columns and any inequality constraint is used in the first column the part of the index that is scanned is more restricted. Column constraints to the right of these index columns are checked in the index so not as many accesses to the table is done, but there is no reduction of what parts of the index need to be scanned [26]. A visual representation of a B-tree index can be seen in Figure 2.1.

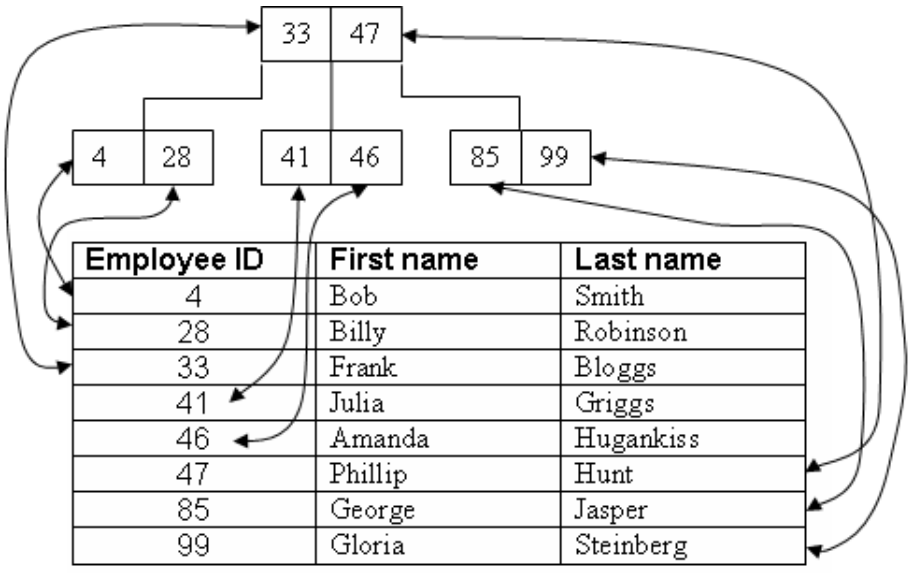

Figure 2.1: A B-tree index.
[27]

Hash indexes

Hash indexes are a secondary index structure that accesses a file through hashing a search key - which can not be the primary key for the file's organisation system [1, pg.633]. PostgreSQL supports persistent, on disk hash

indexes that are crash recoverable. One of the benefits of using a hash index is that any data type can be indexed by it as it only stores the hash value of the data being indexed, thus, there is no size constraint for the data column that is being indexed [28]. Although the use cases for the hash index are limited as hash indexes only support single-column indexes and cannot check uniqueness, nor can they perform range operations. They are best used for SELECT and UPDATE heavy operations that use equality scans over large tables. Another pitfall of the hash structure is the problem of overflow, therefore, hash indexes are most useful for mostly unique data. Because of the inherent nature of the hash structure causing difficulty with expansion, it is most useful for tables with few if any insertions [28].

A hash index can be implemented in different ways [1, pg.633], but in PostgreSQL, it is done by using buckets [28]. These buckets have a certain depth that is split when there are insertions into the index [1, pg.633-635]. An example figure of this can be seen in Figure 2.2.

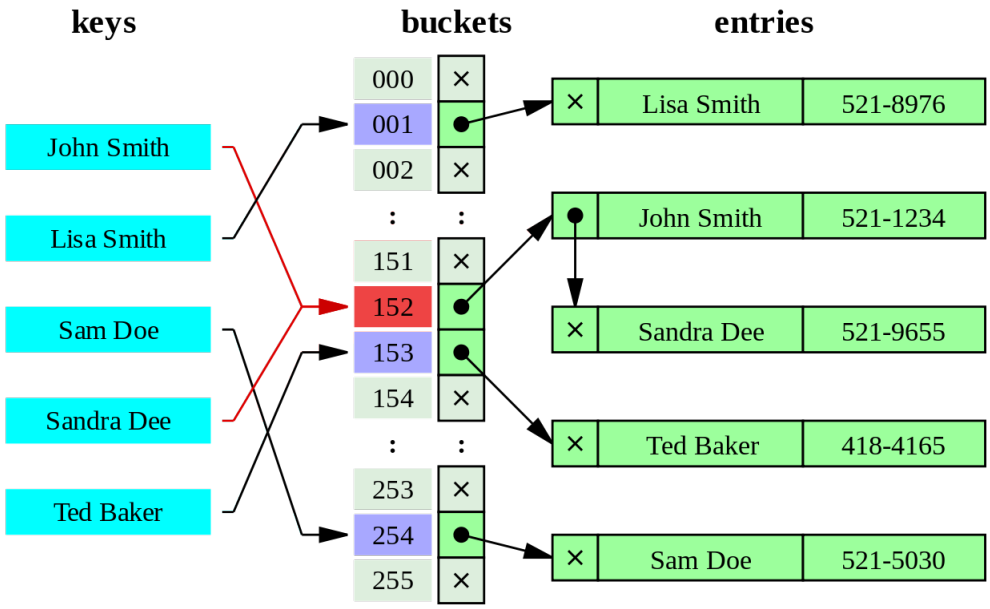

Figure 2.2: Hash index.
[29]

GiST indexes

A GiST index is a type of index that can be tweaked by the developer as there are many different kinds of index strategies that can be implemented [25]. It is based on the balanced tree access method to use for arbitrary indexing schemes. The main advantage to using GiST is that it allows for the development of a custom data type with an appropriate access structure by a data type expert - a programmer that does not have to be a database administrator [30]. How the GiST index is used depends on what operator class is implemented, but the standard for PostgreSQL is to include several two-dimensional geometric data types [25]. The operator class defines what operators can be used on the columns in the index, for example, comparison operations between different data types [31]. GiST indexes can optimise nearest-neighbour searches, but this is dependent on the operator classes defined [25]. A multi-column GiST can be used with query conditions that use any subset of the index's columns. Adding additional columns restricts the entries returned by the index. The way this works is that the first column is used to determine how much of the index need to be scanned. This index is not very effective if the first column only has a few distinct values [26].

SP-GiST indexes

SP-GiST indexes expand on GiST indexes by permitting the implementation of different non-balanced disk-based data structures, such as radix trees, tries et cetera [25]. It supports partitioned trees which allow developing non-balanced tree structures. The generally desired feature for the structures is to use it to divide the search into pieces of equal size [32]. The standard operators for an SP-GiST index in PostgreSQL is to use an operator class for two-dimensional points [25].

GIN indexes

GIN indexes are similar to the previous two ones, although it differs by using the standard operator class for standard array operators [25]. GIN is specially designed to handle when the items to be indexed are composite values, and the queries performed need to search for the element values in the composite items. The word item refers to the composite values to be indexed and the key is the element value. The way the GIN works is that it stores sets of pairs - with the key and the posting list. The posting list is a set of rows Identity Documents (IDs) where the keys occur. Each key-value is only stored

once even though the same ID can occur multiple times [33]. Multi-column GIN indexes work similar to multi-column GiSTs, the main difference is that the search effectiveness is not dependent on what index column the query conditions use [26].

BRIN indexes

BRIN indexes store summaries of the values in a table in consecutive physical block ranges [25]. It is designed to handle very large tables that have columns with some natural correlation to where the columns are physically stored within the table. BRIN indexes can perform queries with regular bitmap index scans which returns all tuples in all pages - within a specified range - if the summary information stored by the index is part of the query conditions. This summary information needs to be updated when new pages of data are filled. This is not done when a new page is created, it is instead created when a summarisation run is invoked. On the other hand, values in a table changing can also cause the index tuple in the summary to be inaccurate. TO solve this, de-summarisation can be run [34]. The operator class that BRIN uses depends on the implemented strategies. For data with linear store order, the data in the index usually correspond to the minimum and maximum values of the columns for each block range, which makes some operations more suitable than others. But as different types of data can be stored in this type of index, the operations need to be chosen based on the type of data [25]. Multi-column BRIN indexes, like GIN has no dependence on what column is used in the query condition. Although there are few reasons as to why a multi-column BRIN would be used [26].

More about PostgreSQL indexes

PostgreSQL can combine multiple indexes, including multiple uses of the same type of index. This is useful when there are cases where a single index scan done by a query cannot directly use the index, which can happen if values are missing in the index that the query needs. To combine multiple indexes, the system creates a bitmap over each needed index. It maps the location of table rows that matches the index conditions, and the table rows are visited in physical order as that is how a bitmap works. This means that the ordering in the indexes is lost, and a separate sort needs to be applied if the query requests ordering of elements [35]. Another index that is supported by PostgreSQL is the partial indexes that are built over a subset of a table, which PostgreSQL also supports. Another reason to use partial indexes is that it can help avoid

indexing common values, since querying common values most often do not use indexes anyways. This reduces the size of the index so that many table operations are sped up when performed on the index [36].

All indexes are secondary indexes in PostgreSQL. This means that the table rows that are referenced can be anywhere on the PostgreSQL data heap. To access the data from an index scan, therefore, involves random access. Which depending on the disk drive can be slow. To make this more efficient, something called an index-only scan is supported. What this means is that a query can be answered without accessing the heap. The idea behind it is to return index entries instead of consulting with the heap entries. In PostgreSQL, only B-trees, GiSTs and SP-GiSTs can support index-only scans, and only B-trees always has built-in support for it [37]. One requirement to decide if an index-only scan is possible to form is that the query that wants to use the index-only scan must only reference columns that are stored in the index, otherwise, heap access is needed. Another requirement for index-only scans is that each row retrieved is visible to the query's Multi-Version Concurrency Control (MVCC) snapshot [37]. The MVCC is something that PostgreSQL uses for concurrency control. It works by showing each query and transaction a snapshot of how the database was some time ago, no matter how the data looks at the exact moment of querying. This protects the transaction from seeing inconsistent data that could be caused by other concurrent transactions [38]. The visibility information is not stored in the index, but PostgreSQL keeps track of the data that is old enough that it should be visible for all future transactions. This means that there is a loophole for data that does not change often to use index-only scans [37]. To effectively use this feature, a covering index can be used. This type of index is designed to include columns needed by a specified query. Sometimes some columns that are not part of the result is needed for a query, PostgreSQL supports this by adding a payload that is not part of the search key with the command INCLUDE [37], this can also be used to solve the problem of missing values in indexes like discussed for combining indexes.

2.3.4 Tuning variables

There are many factors to consider when building the physical database design to ensure efficiency. Among them are analysing queries to optimise the structure of tables and indexes. This is done to ensure that indexes are used and as efficient as predicted. the variables that each retrieval query looks at to map efficiency are: the relations accessed by the query, the attributes

on which a selection condition is specified, what type of condition it is, the attributes of any join or multiple tables, or objects that are linked and the attributes whose values will be retrieved by the query [1, pg.643-646]. As well as for each update operation or transaction: the updated files, the type of operations on each file, the attributes that the selection condition specifies, and the attributes whose values will be changed by the updates need to be assessed. The expected frequency of invocation of queries and transactions, as well as the time constraints of them also needs to be analysed. These aspects also need to be considered for update operations and uniqueness constraints on attributes [1, pg.643-646].

The initial choice of indexes might need to change for many different reasons, some of them might be due to the reasons listed in the previous paragraph. Other reasons are listed below:

- Queries might take too long to run due to lack of indexing.
- Some indexes might not be used by the queries.
- Some indexes are updated too frequently because the index attribute changes too often.

[1, pg.640]

To figure out if any of these issues apply to the database, many DBMSs have commands for tracing how a query is executed. After doing that the issues can be solved by either dropping, creating, or changing indexes (to or from cluster indexes), and rebuilding the indexes. All of these options can improve performance if the tracing is read correctly. The reason why rebuilding indexes can improve performance is because of how in the case of there being many deletions on the index key the index pages may contain space that is not used. This space can then be reclaimed during a rebuild. Rebuilding can also solve overflow issues caused by insertions [1, pg.640].

2.4 Query optimisation

Query optimisation is the action of finding the best possible way for a query to be executed, based on the physical database structure and indexes available. Although optimisation is not the best word for it, as there needs to be a limit for how long it can take before a query needs to be executed, which means that the optimal execution path might not be found. All of this is done by the query optimiser in the DBMS and can be implemented in different ways [1, pg.655].

2.4.1 The query optimiser

The purpose of the optimiser is to create a good query plan, as stated earlier. This is done by the DBMS to retrieve results from the database file. This plan is then translated to code by the code generator, which is done in three steps: the first step is to scan a query to identify all the query tokens. In the second step, the parser checks the syntax, and the validator checks all the attributes and relation names. Thirdly, a query tree structure or a Directed A-cyclical Graph (DAG) is created as an internal representation of the query. There are many different execution strategies for a query and the process of choosing one of them is what query optimisation is all about [1, pg.655-658].

As earlier mentioned, optimisation is not the best term for this process, as most of the time, the optimal plan is not chosen. Rather a reasonably efficient plan is. Finding the optimal strategy is too time-consuming - there is an exception for simple queries - as there are many variables involved when trying to find an optimal strategy. Such as detailed information about the sizes of the table, the distributions of column values, and the expected size of the result. Some of this information is not available for the DBMS. Despite this, optimisation is still needed in relational databases since SQL is a high-level query language. This means that there is only a specification of the intended result, not how to get there [1, pg.655-658].

To do all this the query optimiser first translates the query into an equivalent extended relational algebra expression. This is the tree mentioned for the query plan. It is used to transform the query into an optimised one. The way this is done is most often by deconstructing the query into query blocks, that then are translated into algebraic expressions [1]. After that, the optimiser can choose the best query plan for each block. This is done by improving on the algebraic expressions, and by following a set of heuristic rules. In which one of the most important rules is to preserve equivalence. This is due to there being many algebraic expressions to represent the same query. While the query is optimised it is not allowed to get switched into something else. The equivalence preservation rules ensure that the algebraic expressions for queries remain equivalent [1, ch.18].

2.4.2 The PostgreSQL optimiser

The PostgreSQL optimiser creates a query plan for every query it receives. With the EXPLAIN command, it is possible to access what plans the planner makes for any query. The structure of the planner is a plan tree with plan nodes, in which the leaf nodes of the tree are scan nodes that return rows from a table.

There are different types of scan nodes depending on the type of scan that is performed. If the query has other operations such as join, sorting, et cetera there will be nodes above the scan nodes - which means that the tree grows upwards [39]. As there are different ways to perform these operations, other nodes can also appear. The output of EXPLAIN shows a line for each node in the plan tree, its type, and the estimated cost of the execution of that node. The costs are estimated in arbitrary units that are dependent on the planner's cost parameters. The cost of an upper level-node includes the cost of all its children nodes.

An important thing to keep in mind is that the planner only will consider things it cares about in the cost, transmitting the result is not one of them. This is important to note as there can be other things that affect efficiency that the planner does not count on [39], which could mean that optimising a query is not the best solution to all efficiency problems.

To check the accuracy of the planners estimate the command EXPLAIN ANALYZE can be used. This causes the EXPLAIN command to execute the query and then display the row count and the run time for each plan node as well as their estimates. For the executed plans the unit is in milliseconds instead of an arbitrary unit, which is used by the statistics that EXPLAIN shows. EXPLAIN also has other options, among them is a BUFFER option that further can help with analysing run time statistics. This is done through helping with analysing what I/O operations are the most sensitive [39].

It is also important to note that with EXPLAIN ANALYZE the transactions need to be rolled back as the query is executed [39]. There are also other pitfalls to using EXPLAIN ANALYZE, such as the statistics deviating from normal run-time execution time. One reason as to why this happens is due to no output rows being delivered to a client. This means that there is no consideration to transmission time and I/O conversion costs. Another issue is that the overhead to EXPLAIN ANALYZE can be significant, this is because of how different operating systems can have different speeds for their gettimeofday() operations, so the operation can take longer than actual execution time due to this. The last pitfall to keep in mind is that EXPLAIN results cannot be generalised among different tables. This means that the same result cannot be expected to apply on a large table when tested on a small table [39].

The query planner looks at statistics to make good estimates, it does this for specific variables. For single-column statistics, important factors are the total number of entries in each table, and index, as well as the disk blocks they occupy. This information is kept as part of the table in the pg_class, under

the names reltuples and relpages. These two columns are not updated very often, so they often contain old values. VACUUM or ANALYZE can be used to update them on a per-use basis, which means that they are incrementally updated as they are used [40].

A common issue for slow queries is that the columns used in the query are correlated. The planner assumes that multiple conditions are independent [40]. PostgreSQL supports multivariate statistics to help with this. This is done by creating statistics objects with the CREATE STATISTICS command. Which facilitates an interest in a multivariate statistics object. The data collection is still done with ANALYZE. There are different ways to handle multivariate statistics, but the supported extended statistics in PostgreSQL are: functional dependencies, multivariate N-distinct counts and multivariate Most Common Value (MCV) lists [40]. The functional dependencies are the simplest of the extended statistics. A functional dependency is defined as 'if column a is functionally dependent on column b and if the knowledge of the value in b is sufficient to derive the knowledge in column a'. For example, having a column for social security number and also a birth month column, the birth month can be derived from the social security number, i.e the birth month is functionally dependent on the social security number. The reason as to why functional dependencies have their own statistics tool is due to how the existence of functional dependencies affects the accuracy of estimates in queries [40]. One important thing to note is that for PostgreSQL version 13 functional dependency statistics are limited to simple equality queries [40].

Multivariate N-distinct counts in PostgreSQL help improve the estimates for numbers of distinct values when combining more than one column - such as in GROUP BY(a, b) operations. It is only advisable to create these objects if combinations of columns are grouped, otherwise ANALYZE cycles are wasted. The multivariate MCV lists improve the accuracy of estimates for queries with conditions on multiple columns. This is done by ANALYZE collecting MCV lists on combinations of columns, so the MCV list contains the most common values collected by ANALYZE in the specified columns. This is not recommended to do very often as MCV lists are stored - unlike the information collected by N-distinct counts - which then can take up too much memory. It is advised to only use MCV lists on columns that are used in conditions together [40].

The planner can be controlled with JOIN clauses [41]. As there are many JOIN possibilities between tables to form the same result for queries, the more efficient ones need to be chosen. As JOINS deal with the cartesian product, the less calculation, and processing needed for the same result the better. The

number of JOIN possibilities grows exponentially the more tables are involved, and the PostgreSQL optimiser will then switch from exhaustive search to genetic probabilistic search by limiting the number of possibilities. This takes less time for the search but might not result in the best possible option [41]. There is less freedom for outer joins than inner joins for the planner [41].

2.5 Related works

This section describes some related works and is also the literature study. It starts with works that describe more theory about how indexing and query optimisation is done. It then moves onto related performed experiments.

2.5.1 Database performance tuning and query optimization

In the article 'Database performance tuning and query optimization' [42] Kamatkar. et al, describe database tuning as "minimising the response time for queries by making use of system resources". They further develop on this by describing how it is done through minimising network traffic, I/O operations, and CPU time. Doing this needs a good understanding of the data in the database and how the database - and its application - is supposed to function, the authors explain.

The article focuses on the tuning of a relational DBMS and it describes the typical issues encountered when it comes to databases as CPU bottlenecks, the memory structure, I/O capacity issues, design issues, and indexing issues. They state that indexing can be the solution to many performance issues, but indexing can become an issue if there are too many indexes on tables that update frequently. This is due to how the DBMS creates an overhead when a table is updated to ensure that the index is updated as well. Thus the cost for updates in a table becomes greater when indexes are involved. Maintaining the indexes can also increase CPU and I/O usage which would increase the cost of writing to disk [42].

The article then continues to describe the purpose of query optimisation and that query issues often are caused by bottlenecks, upgrade issues, design issues, large tables, bad indexing, issues with keys, bad coding et cetera [42]. Some techniques to solve efficiency issues are by using column names for SELECT statements instead of the '*' as arguments. As well as ensuring that the HAVING clause is executed after restricting the data with the SELECT

statement, as SELECT works as a filter. Another thing is to try and minimise the number of subquery blocks in a query. The article concludes by stating that creating a data flow diagram makes it easier to understand how a query should work, and then working on improving the queries based on the diagram makes sure that improvements are made [42].

2.5.2 Database tuning principles, experiments, and troubleshooting techniques

'Database tuning principles, experiments and troubleshooting techniques' [43] further develops on this topic. It should be noted that it was written in 2002 and might have some out-of-date aspects. But the material was cross-referenced so that the relevant and reliable facts are the only things presented in this report.

Sasha and Bonnet state that tuning is easy, as there are no difficult mathematical concepts that need to be understood. On the other hand, tuning can be incredibly difficult due to how knowledgeable the tuner needs to be about the database application. They state that there are five basic principles to tuning. First, think globally and fix locally. Which is done by moving data across disks or creating indexes. Creating indexes might be cheaper and more effective than getting more disk space. They state that improving specific queries and bench-marking them will not improve overall performance if the query is not executed frequently. Secondly, partitioning breaks bottlenecks. They describe this by stating that, often it is only one part of the system that limits the whole. A good local fix for this is by creating an index or rewriting the query. The global fix is to create more partitions, this causes the load to get spread out, either over more resources or over time. Although they warn that this might not always improve performance. Thirdly, start-up costs are high while running costs are low. The example they use is for this is that it is expensive to start a read operation on a disk but when the disk is reading, it can deliver data quickly. The authors also warn that to tune a database one must be prepared for trade-offs. Increasing the speed usually costs memory and/or processing power [43].

The book then continues to explain other aspects of index tuning. They describe the correct usage of indexes to have effects such as allowing queries to access one or more aspects in a table more quickly. And that improper use of indexes can lead to problems, such as indexes that are maintained but not used, files that are scanned to return a single record, and multi-table joins that run for a long time due to the wrong indexes being present.

To make more sense of how to implement indexes, as they are dependent

on the queries that are being executed, the authors have defined different query types, which are the following:

- Point queries return one record or parts of a record based on an equality selection.
- Multi-point queries return several records based on an equality selection.
- Range queries return a set of records whose values are within an interval.
- Prefix match queries are queries that use AND and LIKE statements, to match strings or sets of characters.
- Extremal queries are queries that obtain a set of records that return the minimum or maximum of attribute values.
- Ordering queries use the ORDER BY statement.
- Grouping queries use the GROUP BY statement.
- Join queries are queries that links two or more tables. There are different types of join queries. For joins that use an equality statement (equijoins), the optimisation process is simpler, for join queries that are not equijoins the system will try to execute the select statement before joining. This is due to non-equijoins often needing to do full table scans, even when there is an index present.

The authors then go on to describe index types, how they function and, what queries have the most use of them. There are clustering indexes - also called primary indexes - and non-clustering indexes. This has been described earlier in the background and will not be discussed further in this section.

They describe B-trees as good indexes for range, prefix match, and ordering queries. They state that one benefit of using a clustering B-tree the need for using an ORDER BY statement can be removed, this is good to keep in mind if sorting queries often are used on that table. Although, generally non-clustering indexes work best if the index covers all attributes necessary in a query. This is due to the fact that the query then can circumvent the need to access the table entirely if all information it needs is present in the index. They further develop that B-trees are useful for partial match, point, multipoint, range, and general join queries. And that hash indexes are good for point, multipoint and equijoin queries [43].

The authors then describe composite indexes and their benefits. A composite index is an index based on multiple attributes as its key. And having a dense

composite index can sometimes entirely answer a query without accessing the table. It is best used when a query is based on most of the key attributes in the index, rather than only one or a few of them. The main disadvantage for this type of index is the large key sizes as there are many more attributes that can potentially need to get updated when the table is updated. They conclude the chapter by stating that indexes should be avoided on small tables, dense indexes should be used on critical queries and indexes should not be used when the cost of updates and inserts exceed the time saved in queries [43].

The next part of the book describes query tuning and some tips on how to implement optimisation. They promote tuning over indexing by writing that inserting indexes can have a harmful global effect while rewriting a query only can have positive effects if done well. But what is a bad query? How is that determined? The authors state that a query is bad when it requires too many disk accesses and that it does not use the relevant indexes. They follow this up by describing some tips to use to improve queries. One of them is to not use DISTINCT as it creates an overhead due to sorting. DISTINCT is only needed if the fields returned do not contain a key as it then is a subset of the relation created by the FROM and WHERE clauses. It is not needed when every table mentioned returns fields that contain a key of the table by the select statement - a so-called privileged table. Or if every unprivileged table is joined with a privileged one - this is called that the unprivileged table reaches the privileged one [43]. They also caution that many systems do not handle subqueries well, and that the use of temporaries can cause operations to be executed in a sub-optimal manner. Complicated correlation sub-queries can often execute inefficiently and should be rewritten. But a benefit to using temporaries is that it can help with subverting the need of using an ORDER BY statement when there are queries with slightly different bind variables. They also warn against using HAVING statements if a WHERE statement is enough and encourage studying the idiosyncrasies of the system. Some systems might not use indexes when there is an OR statement involved, to circumvent this a union could be used. They state that the ordering of tables in the FROM statement can affect the order of joins, especially if more than five tables are used. They then discourage the use of views as it can lead to writing inefficient queries [43]. Rewriting nested queries is highly encouraged by the authors as query optimisers do not perform as well on many nested queries [43].

2.5.3 PostgreSQL query optimization: the ultimate guide to building efficient queries

The book 'PostgreSQL query optimization: the ultimate guide to building efficient queries' [21] continues to describe query tuning, but this time specifically for PostgreSQL. Dombrovskaya et al, state that an SQL query cannot be optimised outside the context of its purpose and outside its environment, therefore it is not possible to generalise a perfect method for query tuning. They also state that as a database application has many parts, optimising one of them might not improve global performance. For example, if network transactions are slow, optimising a query is not what would help global performance the most. They then go on to caution that PostgreSQL does not offer optimisation hints, like other DBMSs, but instead it offers one of the best query optimisers in the industry. This means that queries in PostgreSQL should be declarative - just stating what should be retrieved, not how to do it - so that the optimiser gets to do its job [21].

How does the PostgreSQL optimiser work though? The authors describes how it uses a cost theory for optimisation. It does this by using internal metrics that are based on the resources needed to execute a query or operation within a plan. The planner combines the primary metrics such as CPU cycles and I/O accesses to a single cost unit that is used for comparison of plans. There are different ways to access data and depending on different factors, and one way can be more efficient than another. The main factors used are full table scan, index-only scan, and index access, they write. For smaller values of selectivity - the percentage of rows in a table that the query selects - index access is preferable, as it is faster than a full table scan. But the best option is to use an index-only scan if the query allows it. This is not a general rule and is instead entirely dependent on what type of index is used [21].

The book then further develops on how the optimiser works. Such as the transformation and heuristics it uses to convert one plan to a better one. This is done in stages. The optimiser presents the plan as a tree that reads from the leaf nodes to the root. The first step of optimisation is to enhance the code by eliminating sub-queries, substituting views with their textual representation et cetera. The second step is to determine the possible order of operations, what execution algorithms are needed for the operations, and then compare the costs between the different plans to select the better one. Something specific to PostgreSQL is that it does not perform accessing and joining in the order they are presented in the FROM clause, so that is not something the query writer has to consider. The algorithm for the optimiser relies on the optimality principle,

which is that a sub-plan of an optimal plan is optimal for the corresponding sub-query. This means that for the optimisation tree, which consists of leaf nodes, - that represent file access - each node level contains more complex sub-queries. Heuristics are used to cut out the branches that are unlikely to be optimal and the cost for each node is calculated based on statistics that are represented as histograms. These histograms contain statistics of the existing data on tables, indexes, and distribution of values. The optimiser is not always correct though. Some pitfalls of it are mainly due to the histograms not being able to produce intermediate results, cost estimates are imprecise, and that heuristics might cut a plan too early to see if it was not optimal [21].

The authors then go on to describe short and long queries, what they are and how they can be optimised. A short query is a query that only needs a small number of rows to compute its output. This means it can read the entirety of a small table or about less than 10 % of a large one. Short queries benefit from using restrictive indexes and are most efficient with unique indexes, as these have fewer values to go through. Things to keep in mind when using short queries are that column transformations make it so that an index search cannot be performed on the transformed attribute. This means that in short queries column transformations should not be used. LIKE statements also do not utilise indexes, so they should also be avoided and can instead be replaced by equivalent OR statements [21].

Some other PostgreSQL-specific things the authors bring up are that PostgreSQL supports multi-index searches, which is done by creating bitmaps of blocks with matching records in main memory and then OR or AND-ing them together. When this is done, only blocks that match the search criterion remain. Since blocks are scanned in the order they are stored, the index order is lost. PostgreSQL also supports covering indexes that are used for extra support for index-only scans. These indexes are used so that other criteria do not need to be added to the index definition and can instead just INCLUDE the needed attributes. Excessive selection criteria can be added to a query to force the planner to use indexes or to reduce the size of joins. Another type of index that is supported by PostgreSQL is the partial index. It is an index that is built on a subset of a table and is used in a similar way to table partitioning but is instead to ensure that an index-only scan can be performed [21].

They conclude this chapter by stating that indexes should not be used when the table is small, or if the majority of the rows in a table are needed to execute a query, or a column transformation is used. To force a query to use an index the ORDER BY operation can be used.

A long query is described as when query selectivity is high for at least one

of the large tables. This means that almost all rows contribute to the output, even if the output size is small. The way to optimise these types of queries is by avoiding multiple full table scans and reducing the size of the result as soon as possible. Indexes are not needed here and should not be used, the authors state. For joins, a hash join is most likely the better algorithm for the job when dealing with long queries. If GROUP BY is used by a long query, the filtering needs to be applied first in most cases, to ensure efficiency. There are times that GROUP BY can reduce the size of the data-set, but the rule of thumb is to apply the SELECT statements first for the optimiser. Set operations can sometimes be used to prompt alternative execution plans. This can be done by replacing NOT EXIST and NOT IN with except, EXIST and IN with INTERSECT, and use UNION instead of multiple complex selection criteria with OR [21].

The authors then describe the pitfalls of views and that their main use, which is for encapsulation purposes. Materialised views on the other hand can help improve performance. This is due to the fact that data is actually stored and because indexes can be created on them. A materialised view should be created if the data it is based on does not update often if it is not very critical to have up-to-date data, the data in the materialised view is read often, and if many queries could make use of it.

After this section, the authors discuss partitioned tables. The main use for them is to optimise table scanning. If a query uses values in the range of the partitioning, only one partition would need to get scanned. This means that the key should be chosen to satisfy a search condition. Indexes can be applied on these tables, and they are beneficial for short queries [21].

After this, multidimensional and spatial searches are discussed. The authors state that spatial data often require range queries. Which means finding all the data located at a certain distance or closer to a specified point in space. And nearest-neighbour queries, which is to find a variable number of objects closest to the specified point. These queries cannot be supported by one-dimensional indexes or even multiple indexes. This is when GiST indexes come into play. They describe GiST indexes as points and search conditions are represented as a rectangle and that all points within the rectangle or polygon are returned as the result [21].

Lastly, the book concludes with the ultimate optimisation algorithm for queries which summarises the points brought up in this section.

2.5.4 Comparison of physical tuning techniques implemented in two opensource DBMSs

The report ‘Comparison of physical tuning techniques implemented in two opensource DBMSs [44] questions if there are different tuning techniques between MySQL and PostgreSQL, what techniques they support, and if they improve performance. The goal of this study was to compare the two open-source performance tuning techniques with each other and to answer the problem of if there are any significant differences in the tuned and untuned performance of queries between MySQL and PostgreSQL with regards to indexes, BLOB management, and denormalisation. Only indexes are relevant for this report so the other aspects of the result will be omitted. It is also worth noting that this report was written in 2005 some aspects of it might be outdated.

Only b-tree indexes and hash indexes were investigated as those were the only indexes that the DBMSs had in common. The result showed that the average time reduced for PostgreSQL with a B-tree index was 67.4% and that the hash index increased query time for the queries tested [44].

2.5.5 PostgreSQL database performance optimization

In the report ‘PostgreSQL database performance optimization’ [45], the question of how well indexes perform for certain queries and also if updating the query statistic mattered. The result is shown in Figure 2.3.

Steps	Without index	With index	Difference
Retrieve task data	1402.842	261.275	1142.567
Retrieve worker data	1197.555	910.242	287.313
Hash worker records	328.803	335.627	-6.824
Join task and worked	611.711	532.272	79.439
Retrieve costumer data	2386.164	1358.244	1027.92
Retrieve company data	170.747	279.589	-108.842
Join customer and company	194.505	170.244	24.26
Hash costumer and company records	184.086	103.039	81.047
Join task and costumer	444.445	450.632	-6.187
Sort	407.123	433.888	-26.765

Figure 2.3: Table of collected data for execution time of queries with and without indexes.

The steps are the queries performed, the second column shows the time of execution without indexes, and the third column the time after indexes and clustering were implemented. The result for the prepared query execution was that no major difference was noticeable on single queries. The prepared query was done by running EXPLAIN ANALYZE to ensure that the optimisers statistics were up to date [45].

2.5.6 MongoDB vs PostgreSQL: a comparative study on performance aspects

The report 'MongoDB vs PostgreSQL: a comparative study on performance aspects' [46] compares the two DBMSs on their available indexes for spatio-temporal data. It investigates the performance of B-trees and GiST, but also how queries are affected by indexes. The result was that PostgreSQL performed on average 89 times faster with an index applied [46].

2.5.7 Comparing Oracle and PostgreSQL, performance and optimization

In the report 'Comparing Oracle and PostgreSQL, performance and optimization' [47] the optimisation strategies between the Oracle DBMS and PostgreSQL were compared. This was done using benchmarks with a strategy of adding column-based indexes to improve query execution. The result showed that PostgreSQL can improve up to 91% with indexes, which means that it is more sensitive to optimisation and shows better performance with them. By only adding primary and foreign keys the performance was improved by 38% and by adding indexes it was improved by 88% [47].

2.5.8 Space-partitioning Trees in PostgreSQL: Realization and Performance

This report [48] focuses on comparing different implementations of SP-GiST indexes compared to B+-trees and tries. The SP-GiSTs was implemented with PostgreSQL was extended to include prefixes and regular expression matches, as well as a generic incremental NN search (nearest-neighbour search). The result showed that a disk based SP-GiST trie performed two orders of magnitudes better than a B+-tree when it comes to regular expression match searches, while a disk based SP-GiST kd-tree performed more than 300% better for a point match than a trie. A disk based suffix tree was also implemented for substring match purposes and it performed around three orders of magnitude better than the existing technique (text scan) at the time of this report. These implementations were made based on using different SP-GiST operators, which the report describes as external methods to support different types of queries.

The SP-GiST trie implementation was compared to the B+-tree in the context of text string data, while the SP-GiST kd-tree and PMR quadtree was compared to R-trees in the context of point and line segment data, respectively. The suffix tree was compared to sequential scanning as there is was no other method to support substring matches.

The result shows that a disk based SP-GiST trie performs two orders of magnitudes better than a B+-tree when it comes to regular expression match searches this was due to how B+-trees are sensitive to where single character wildcards appear. A wildcard is used with the character '?'. It retrieves multiple data sets of a string for example: ?ove = cove, love, dove, etc. The reason for the result was because the B+-tree used the wildcard in the search,

so if the wildcard appears in the first character then a full table search has to be made, it explores all the avenues without filtering. The trie on the other hand uses non-wildcard characters in the search for filtering.

In this report experimenting, the trie had better search performance than the B+-tree when it came to exact match, around 150% better, it also scaled better than the B+-tree. For prefix matches the B+-tree outperformed the trie, this was due to the inherent nature of having the keys sorted in the leaf nodes. Which allows the tree to answer prefix match queries very efficiently. For exact matches the B+-tree scales better as well, this is due to how the trie consists of more nodes and more node splits than the B+-tree.

Kd-tree and R-tree comparison was done over a two-dimensional point data set. The kd-tree performed 300% better than the R-tree when it came to point search and 125% better when it came to range search, although the R-tree has better insertion time and better index size. This is due to how the kd-tree has a node size (bucket size) of one and every insertion causes a node split. This leads to the number of nodes being very large and the clustering technique that SP-GiST uses to reduce the tree page height costs the index page utilisation.

PMR quadtree in comparison to R-tree for indexing was done on line-segment data sets. The R-tree had better insertion and search performance.

The nearest neighbour search for the kd-tree and the point quadtree was better than for the trie. This is due to how the trie performs the NN search character by character while for the kd-tree and the point quadtree the NN search is based on partitions.

Chapter 3

Method

This chapter describes the research methods and methods used for the testing of optimisation methods. The first section describes the methodologies used for the research and how they were used for the project. The sub-questions for the project are then presented as well as the research approach.

3.1 Research methods

This section describes the chosen research methods and why they were chosen.

3.1.1 Quantitative and qualitative methods

These two methods are typically applied to projects that are either numerical or non-numerical. One method needs to be chosen to show what the research is based on. Quantitative research verifies or falsifies what is being tested or built based on variables that can be measured with quantifications. These methods need to use large data sets and use statistics to make the research project valid. Qualitative research on the other hand is used to try and discern meanings to develop theories for a conclusion. This method uses smaller data sets that are trustworthy enough to reach a reliable result [49].

Using a qualitative method often has the purpose of creating an understanding of why things are the way they are. While quantitative methods is a research approach that is an objective, formal and systematic process that often uses empirical data. It often describes, tests, and examines cause and effect in relationships by using a deductive process. The difference between qualitative and quantitative in that sense is that the qualitative approach develops a theory inductively. Other differences can be seen in the sampling of data.

Qualitative methods often choose data sets that are small and selective, while the quantitative approach uses large and random data. The purpose of the random collection of data is to be able to draw general conclusions [50].

3.1.2 Inductive and deductive approach

The inductive approach is used to formulate theories by using explanations from observation. Data is usually collected with qualitative methods and by analysing the data to provide explanations for it, to understand what is happening. The result is based on experiences and needs to contain enough data to explain the phenomenon [49].

The deductive approach is used to verify or falsify a hypothesis, it is most commonly used with a quantitative approach. The hypothesis needs to use measurable terms and explain the variables measured, as well as express the expected result. The result from this approach is a generalisation that is based on the collected data, and the explanation of how variables are related to understanding what is happening [49].

The purpose of the inductive approach is to allow findings from frequent, dominant, or significant themes to be found in raw data without putting many restrictions on it [51]. Like, for example, when using a deductive approach the restrictions of the wording formulating the hypotheses can cause the key themes of the research left invisible or obscured. Therefore the inductive approach is better suited to describe the actual effects, not just the planned effects. Other purposes of the inductive approach are, to establish clear links between the objectives and the findings from the raw data, and develop a theory about the underlying structure that the data shows. In conclusion, the inductive approach aims to question the core meanings there are for the research area. This should then be presented by describing the most important themes [51].

3.1.3 Subquestions

The research question posed in chapter one is "how do indexing and query optimisation affect response time for a PostgreSQL database?". To explain what this means the question can be divided into sub-questions.

- What methods of indexing are there and what are their use cases?
- How does query optimisation work and how can queries be optimised?
- What is the overlap between indexing and query optimisation?

- How does indexing, the query optimiser, and query tuning compare to each other?

3.2 Applied methods and research process

This section explains why the methods were chosen, as well as how they are applied for this thesis by describing the process.

3.2.1 The chosen methods

The methods chosen for this project is the qualitative and quantitative method with an inductive approach.

Finding information for the background and literature has to be done qualitatively as some information is harder to find than others. Making sure that the information is reliable is of higher priority than collecting a large quantity of it. It is also suitable as there are some aspects of the research field that are more explored than others, which means that the sample data in some fields are smaller than others. This is not a big deal in qualitative research as long as the data is reliable enough and relevant.

The quantitative method is chosen for the experiment. The experiment deals with a large set of data - although in a specific case - and is performed similarly to a laboratory experiment. Such as by including specified variables to measure, that are measured in a specific environment and with specified tools and scripts.

The inductive approach is chosen partly because of the qualitative method, but also because of the purpose of the project. The goal is to observe the behaviour of indexing and query optimisation to reach an understanding of what is happening. Which matches the purpose of an inductive approach well. Especially when there is no hypothesis and due to the delimitations that restrict the amount of generalisation that can be made. The theories are formulated from the inductive result are then tested with the quantitative method.

3.2.2 The process

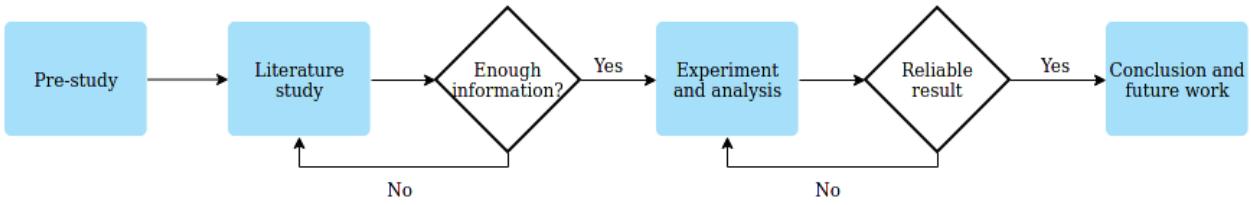

Figure 3.1: Flowchart of the method.

Pre-study

As seen in Figure 3.1, the first step of the method is to conduct the pre-study. This is conducted to gain a basic understanding of the research area as well as to develop the aim and research question that are formed in this report. This is the information found and presented in the background of this report. This is done to ensure that the necessary skills and knowledge for this report are known and that the researcher in question has that knowledge. By learning more detail about databases and how to optimise them, delimitations are formed, and the research question is worded in such a way that it included the sub-questions.

The literature study

The second step is to conduct the literature study. Which was done by following the research question and keeping track of the purpose of this thesis. This is done with a qualitative and inductive approach as explained earlier. Information is filtered by relevance and then analysed for reliability. The reliability comes from seeing if there are multiple sources, as well as comparing the studies found with each other to see if it follows a trend or if the discussions of the result provides a viable explanation for it. If an explanation is viable is deduced based on if these explanations also could be found or deduced from other sources.

Experiment and analysis

The third step is done after the literature study was conducted and the theories formed by the inductive approach, an experiment is developed to test these theories. The planning of the experiment is, in part, based on the findings in the literature study. This is done to be able to get a more reliable result as well as to be able to compare the findings of the experiments to the literature studies. The experiment conducted follows a quantitative approach by using a large data-set in a database as well as measuring execution time over different sizes of data-sets, and how the execution time changes with implementations of indexes and query tuning.

Conclusion and future work

After interpreting the result, it is presented in this thesis and explained and analysed in the discussion chapter. A conclusion is drawn and implications of what was found is discussed, as well as reflecting on proposals towards further studies. The result is also analysed for reliability. In this experiment this is handled through performing the planned queries on the database multiple times to get an average execution time that would be more reliable than only performing it once. As well as comparing the result of the testing to the results of the experiments performed in the literature studies. The procedure followed to ensure quality is described in the next part.

3.2.3 Quality assurance

To ensure the quality of the experiment the following criterion need to be met:

- Ensuring validity. Making sure that the research has been conducted according to the rules of the project and that the meaning of the result can be easily discernible. As well as making sure that any testing instruments measure the correct things [49].
- Ensuring dependability. Judging the correctness of conclusions, by reviewing the content, scrutinising it, and making sure to note down the consistency of the result for each testing instance [49].
- Ensuring replicability. This means that there should be sufficient information in this report to be able to replicate the study and get similar results [49].

Chapter 4

Experiment

This chapter goes into detail about the experiment performed for this project. It tells of the hardware and software used, the database design and queries performed. The improved queries can be seen in a section below. The database schema, and the index design can be found in the appendixes A and C at the end of this report. The details in this chapter should be sufficient enough to ensure replicability.

4.1 Experiment design

This section describes under what hardware conditions the experiment was conducted, as well as what software was used. It then moves onto describing how the experiment was conducted.

4.1.1 Hardware

The following list presents the relevant hardware used to run the database environment, for the purpose of replicability of the experiment:

- Motherboard: ROG STRIX X99 gaming
- Random Access Memory (RAM):Corsair Vengeance LED DDR4, 4x8 GB, 3400MHz
- CPU:Intel core i7-6850K, 3.6 GHz, 15MB Cache
- SSD: Kingston A400 SSD, 960 GB, 500MB/s read and 450MB/s write

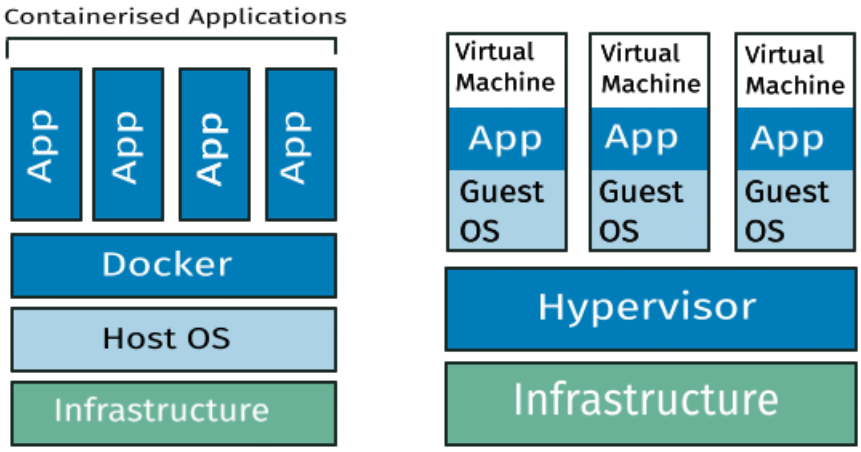

Figure 4.1: Comparison of containers and virtual machine.

4.1.2 Docker and the docker environment

Docker is an open source containerisation platform. A containerisation platform allows the developers to package applications into containers, which are standardised executable components that combine the source code for the application with the operating systems libraries and dependencies. The purpose of this is to simplify the process of delivering distributed applications [52]. The way this works is due to the nature of what a container is. A container uses the isolation and virtualisation capabilities that are built into the Linux kernel. This allows the parts of the application to share the resources of one instance of the host operating system, similarly to how the relationship between hypervisor and virtual machines work. This means that a container have many of the same abilities as a virtual machine does, plus some additional benefits. Such as, being more light weight than a virtual machine, being more efficient than a virtual machine, and being faster to deploy, provision, and restart [52]. A comparison of containers and virtual machines can be seen in Figure 4.1.

The benefits of using docker specifically as a containerisation program is that it allows automated container creation based on source code that can be found on the docker official website - so called docker images [52].

The information about the docker version is Docker version 20.10.9, build c2ea9bc90b. With the PostgreSQL version being: psql (PostgreSQL) 14.0, and the operating system: Debian 14.0-1.pgdg110+1. The image version is the latest version as of this report (2021-10-07), with the id: 6ce504119cc8. The additions that are added to this is: after the image was pulled, Debian is

updated to the latest version as of 2021-10-06 and the time package is installed (apt install time).

4.1.3 Other software

Other software used to handle the data produced and to manipulate the data in the database file are software that comes with using a Linux system. The 'sed' command is used to create smaller databases from the original SQL dump file. For each version created the number of rows are divided by 10. So the first n rows are taken from the dump file and placed in another file. The row numbers for the tables in the database can be seen in Figure 4.3.

A small python script is also used to quickly calculate the mean of the data-points collected in the files.

4.1.4 Method and purpose

The purpose of the experiment is to gather data to measure the difference in execution time based on query improvement, implemented indexes and the ANALYZE command for the optimiser. This is done to compare the difference in efficiency, and if one of these methods show greater difference than the others. Another thing measured is the scalability of these methods.

To do this, first a baseline measurement is taken by running the queries - explained later in this chapter - on all database sizes. The queries are run 100 times to gather 100 data points that are then used to calculate a mean for each query. This is repeated four times to calculate the mean of the mean, for a more reliable result. The first time running the queries should not be used in the mean calculations as the caches should be warmed up first, for a more reliable result. The result gathered from this is plotted to show the execution times for the queries and how they scale when the database grows. Thereafter, the same procedure is followed to gather measurements for the tuned queries, the implemented indexes, and the ANALYZE command. The usage of ANALYZE is restricted to only be ran before the original queries are run, and also not used as a data-point, as ANALYZE only is used to update the statistics for the optimiser. This means that the ANALYZE command is run for the queries before the looping of them.

4.1.5 Database design

The database can be seen in Figure 4.2 and the database schema can be seen in the appendix A. The database is based on the IMDb-database (link in appendix

F, which is a database filled with movies, games, tv-shows and other media. It contains information about people who have worked in the media, and how it is rated. The ratings are collected from users on the IMDb website. As the figure shows, the database only has six tables, with a couple of attributes in each table. The person table contains the person_id which is just a string of characters to identify the row in the table - and is also the primary key. It also contains the surname and last name of the person, their date of birth, and death date - which is null if the person is still alive. This table has a one-to-many relationship with the crew table as one person can be multiple crew members. The crew table has a title_id - which is the id of the media the crewmember worked on. The person_id to link to who the person is, a category - which is the title of their work, i.e actor, director, writer, etc - and job. From looking at the data in the database file the job column is mostly null values, with the exception for producers who have a repeat of 'producer', and writers which have what they write - poem, play, book - and the title of it in a string.

The next group of tables is the ones that contain information about the media in the database. The akas table is an overview of the media in the database, it contains the basic information about the media such as the title, the region it was produced in, what language it contains, what type it is - shows IMDb display, original, alternative or null. What attributes it has - information about the title, mostly null - and a boolean value for if the title is an original title or not. This then gets further divided into an episodes table, that shows information about the episodes in a show, i.e the episode number and season number. More information about the titles can be found in the titles table. It shows what type of media it is, the original and the primary title of the media, if it is adult rated, when it premiered, when it ended (mainly for shows), how long the runtime for the media is, as well as what genres the media is in. The genre column contains a string of all the genres the media belongs to. The titles table also contains the primary key, which is on the title_id column. The last table is the ratings table, which contains the average rating for the title and how many votes it has received, it also has a primary key on the title_id column. Thus, there is a one-to-one relationship between titles and ratings, a one-to-many relationship between titles and akas, and a one-to-many relationship between titles and episodes.

It is important to note that the database does not contain any foreign keys, the only key constraints that exist are the primary keys that can be seen in Figure 4.2.

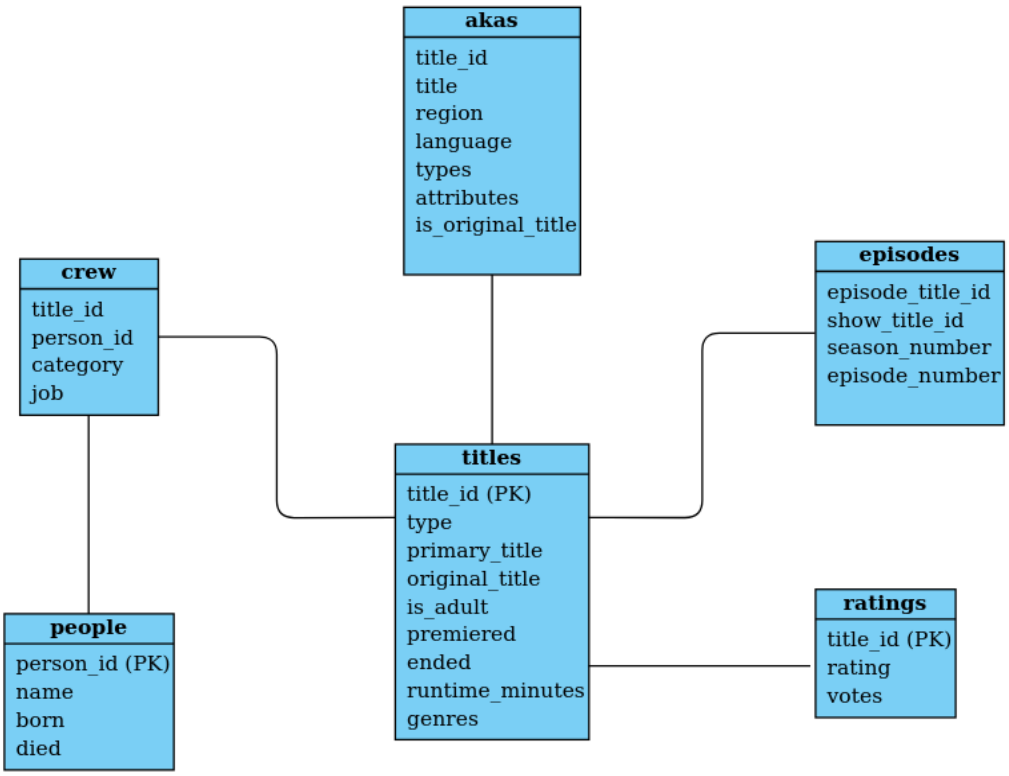

Figure 4.2: The IMDb-database table relations.

In Figure 4.3 the different amount of data can be seen for the testing. The data is calculated by dividing the original amount of rows - seen under the column named 1 - in each table by ten for each iteration. The first n rows are taken from the original file and placed in separate files to fill the database with.

Table name	1	2	3	4
Akas	1436745 rows	143675 rows	14368 rows	1437 rows
Crew	9990049 rows	999005 rows	99901 rows	9990 rows
Episodes	593366 rows	59337 rows	5934 rows	593 rows
People	3571826 rows	357183 rows	35718 rows	3572 rows
Ratings	362285 rows	36229 rows	3623 rows	362 rows
Titles	2294719 rows	229472 rows	22947 rows	2295 rows

Figure 4.3: The table sizes in the database.

4.1.6 Queries

The queries used for the experiment are listed below, including a small description as to what they are testing and why they are chosen. The reason for there being a smaller amount of queries to test things is due to there being more extensive testing, both to see scaling and performance, rather than trying to find queries that have specific cases.

```
1     --how many movies are in the database?
2     SELECT COUNT(DISTINCT title_id)
3     FROM titles
4     WHERE type IN ('movie', 'video');
```

Query 1

Query 1 is used to test multi-point queries. This query is chosen due to the fact that both hash indexes and B-tree indexes are good choices for point queries, so seeing a difference in performance would be noticeable here. This query is also easy to make improvements to when it comes to tuning, which is another factor in the choosing of it.

```
1     --how much content in each type is on the
    database and what are the types?
2       SELECT type, COUNT(*)
3       FROM titles
4       GROUP BY(type)
5       ORDER BY (type) ASC;
```

Query 2

Query 2 looks at all the types that there are in the table and then how many of each type there are. This is a large query which is why it was chosen. Seeing

if it can be improved by any of the methods for this query type would be very useful.

```
--list all actors/actresses playing in
 a spiderman movie
SELECT DISTINCT name
FROM (SELECT primary_title, original_title,
    crew.title_id, person_id, category
    FROM crew
    INNER JOIN titles ON
    titles.title_id = crew.title_id
    WHERE primary_title
    LIKE 'Spider-Man%'
    OR original_title LIKE 'Spider-Man%') as a
INNER JOIN people ON
a.person_id = people.person_id
WHERE a.category = 'actor'
OR a.category = 'actress';
```

Query 3

Query 3 lists all the actors and actresses that have played in a Spider-Man movie. This query is chosen in part because of how it has an inner query, that could easily be transformed into a materialised view, so comparing that in performance is of interest.

```
--get the second-highest rating
SELECT DISTINCT rating
FROM ratings
WHERE rating  = (
SELECT MAX(rating) FROM ratings
WHERE rating != (
SELECT MAX(rating) FROM ratings));
```

Query 4

Query 4 gets the second-highest rating for the media in the database, this query is chosen due to how it has a correlated inner query and see if any of the methods can improve this is of interest as correlated inner queries can be likened to inner loops in other programming languages. Seeing if this could be improved by the optimiser or an index is of great interest.

```
--find all movies made between 2000 and 2010
SELECT primary_title, premiered
FROM titles
WHERE type LIKE 'movie'
AND premiered BETWEEN 2000 AND 2010
ORDER BY premiered ASC;
```

Query 5

Query 5 gets all the movies that are premiered between 2000 and 2010. This query is chosen to test range queries.

4.1.7 Improved queries

This part describes the improved queries and how they are improved.

```
1     --how many movies are in the database?
2     SELECT COUNT(title_id)
3     FROM titles
4     WHERE type = 'movie' OR type = 'video';
```

Improved query 1

The difference between improved query 1 and query 1 can be seen in the SELECT statement and the WHERE statement. The improved query does not use DISTINCT, as it is deemed unnecessary due to the nature of primary keys being unique. DISTINCT would just add extra overhead to the filtering. The WHERE clause differs in how the improved query uses OR instead of IN. This is done to see if IN and OR had any difference in performance. As IN checks the column value and matches it to a list of values. Technically, as stated in the background, the IN statement should be executed faster than OR, so it is not an improvement, rather a difference in the query to see if it makes a difference in performance.

```
1     --how much content in each type are on the
    database and what are the types?
2       SELECT type, COUNT(title_id)
3       FROM titles
4       GROUP BY(type)
5       ORDER BY (type) ASC;
```

Improved query 2

The improved query 2 differs from query 2 by counting the column title_id instead of *. This is done to test the the statement in the literature study. The source states that by switching * to the column to be counted, performance would be improved.

```
--list all actors/actresses playing in
 a spiderman movie
 CREATE MATERIALIZED VIEW q3
 AS
 SELECT primary_title, original_title,
    crew.title_id, person_id, category
    FROM crew
    INNER JOIN titles ON
    titles.title_id = crew.title_id
    WHERE primary_title
    LIKE 'Spider-Man%'
    OR original_title LIKE 'Spider-Man%';

 SELECT DISTINCT name FROM people
 INNER JOIN q3 ON
 q3.person_id = people.person_id
 WHERE q3.category = 'actor'
 OR q3.category = 'actress';
```

Improved query 3

The improved query 3 uses builds a materialised view instead of using an inner query. As source from the literature study states that the query planner can have a difficult time optimising queries with inner loops. It is also stated in the background that running queries on materialised views can cause better query performance.

```
--find all movies made between 2000 and 2010
SELECT primary_title, premiered
FROM titles
WHERE type = 'movie'
AND premiered BETWEEN 2000 AND 2010
ORDER BY premiered ASC;
```

Improved query 5

The improved query 5 differs from query 5 by switching the LIKE operation to an equals operation which is done to test if there is a difference between them. As LIKE uses pattern matching for the characters. It can be used to use wildcards, but as the only thing that is matched is 'movies' it was deemed unnecessary if performance differs.

4.1.8 Keys and indexing structure

As there are primary keys in the database already no key constraints needed to be added. Due to how generic indexes often are placed on foreign and primary

key constraints that are how indexes are decided to be placed.

The indexes tested are the B-tree index and the hash index, they are tested one at a time and are placed in what is deemed a generic way, which is by placing them on the primary keys and what would be the foreign keys of the tables. The index on the titles and episodes table is sorted based on the title_id. Crew is sorted on title_id, and the people index is sorted based on the person_id. The akas and ratings indexes are sorted on title_id, as well.

After this, more personalised indexes are created to see how the queries would interact with them. The personalised index is used in this report for a lack of official wording. It is defined as an index that is tuned specifically toward a query. The index on the titles table is on the type column and another index on the premiered column. The episodes table is sorted based on the show_title_id. Crew is sorted on category, and the people index is sorted based on the person_id. The akas indexes are sorted on title_id, and the ratings table on rating. These indexes replace the old general indexes.

The full indexing schema can be seen in the appendix C.

4.1.9 The experiment tests

The following list presents the experiments that are run on the IMDb database.

- The queries, this is used for baseline measuring and is used to decide if the other results are slower or faster.
- Improved queries, the original queries that have been tuned for better performance.
- The DBMS optimiser, this is done by running the ANALYZE command with the queries, before looping them to ensure that the optimiser statistics are up to date.
- General indexes, running the baseline queries with indexes built based on key constraints.
- Personalised indexes, running the baseline queries with indexes that are built based on columns used by the queries.

Chapter 5

Results and Analysis

This chapter summarises the result of the literature study, as well as presents the result from the experiment.

5.1 Literature study result

This section describes the results from the literature study, and can also be seen as a summary of the main points of the Related works section.

5.1.1 Theory

In the report 'Database performance tuning and query optimization' [42] the main take-away points are that indexing can be the solution to many performance issues, but maintaining an index can cause overhead when updating tables. It can also cause CPU and I/O usage to increase which also increases the cost of writing data to disk [42]. The book 'Database tuning principles, experiments, and troubleshooting techniques [43] further develops on this. First, it describes how a database administrator should think to improve a database with a three-step technique:

- Think globally, fix locally: Creating indexes can be a good solution as it can be cheaper than creating more disk space.
- Partitioning break bottlenecks: a local fix to breaking bottlenecks is to create indexes.
- Start-up costs are high, running cost is low: improving execution time often costs memory or processing power.

The book then continues to describe how queries can be divided up into types, and what indexes suit which query type. The query types are described as:

- Point queries return one record or parts of a record based on an equality selection.
- Multi-point queries return several records based on an equality selection.
- Range queries return a set of records whose values are within an interval.
- Prefix match queries are queries that use AND and LIKE statements, to match strings or sets of characters.
- Extremal queries are queries that obtain a set of records that return the minimum or maximum of attribute values.
- Ordering queries use the ORDER BY statement.
- Grouping queries use the GROUP BY statement.
- Join queries are queries that links two or more tables. There are different types of join queries. For joins that use an equality statement (equijoins), the optimisation process is simpler, for join queries that are not equijoins the system will try to execute the select statement before joining. This is due to non-equijoins often needing to do full table scans, even when there is an index present.

B-tree indexes are in particular good for range, prefix match, partial match, point, multipoint, general join, and ordering queries. Clustering B-trees are good for getting rid of the ORDER BY statement, due to the ordering nature of B-trees in combination with physical storage. And for non-clustering indexes, covering all the attributes necessary for a query is the best way to use them, as then it is possible for the DBMS to use an index-only scan. Another type of index is the composite index, whose use-cases are mainly to ensure minimal table accesses for queries that use many of the key attributes in the index. Although, there can become an issue with updates, as this type of index use many attributes for its key, the chance of the index having to update when the table does is higher. The major tip from this book is that indexes should be avoided on smaller tables, dense indexes should be used on critical queries to make use of the index-only scan, and building an index is dependent on if the time saved in execution time is larger than the cost of updating the index [43].

Some methods for improving queries are getting rid of the * and instead using the column name in operations. Make sure that the HAVING clause is

executed after restricting the data with the SELECT statements. As well as by minimising the number of subquery blocks that are in a nested query [42]. Query tuning should be considered before implementing indexes, as inserting indexes can have harmful global effects. In comparison, rewriting a query can only have positive effects, if done correctly [43]. Tips for rewriting queries are:

- Do not use DISTINCT unnecessarily as it creates an overhead due to sorting.
- Avoid subqueries as much as possible, as many systems do not handle them well.
- Complicated correlation sub-queries can often execute in an inefficient way and should be rewritten.
- The use of temporaries can cause operations to be executed in a sub-optimal manner, but it can also help with subverting the need of using an ORDER BY statement.
- Do not use HAVING statements if a WHERE statement is enough.
- Study the idiosyncrasies of the system. Some systems might not use indexes when there is an OR statement involved, to circumvent this a union could be used.
- They state that the ordering of tables in the FROM statement can affect the order of joins, especially if more than five tables are used.
- The use of views as it can lead to writing inefficient queries.

[43]

The book 'PostgreSQL query optimization: the ultimate guide to building efficient queries' [21] continues on building the theory for creating optimised queries. The book begins by stating that a database application has many parts, optimising one of them might not improve global performance. And also, that PostgreSQL has one of the best query optimisers in the industry, so declarative queries should be used. The way the planner works is by combining the primary metrics such as CPU cycles and I/O accesses to a single cost unit that is used for comparison of plans. Some inaccuracies of the optimiser are mainly due to the stored histograms not being able to produce intermediate results, cost estimates being imprecise, and that heuristics might cut a plan too early to see if it really was not optimal [21].

A query can access data in different ways. The main ways are full table scan, index-only scan, and index access. For smaller values of selectivity, index access is preferable, as it is faster than a full table scan. This also means that if selectivity is high, using a full table scan is preferable. But the best option is to use an index-only scan if the query allows it. Although this is entirely dependent on the index used [21].

The book then describes short and long queries and how they can be tuned. Short queries benefit from using restrictive indexes and are most efficient with unique indexes, as these have fewer values to go through. Things to keep in mind when using short queries are that column transformations make it so that an index search cannot be performed on the transformed attribute. LIKE statements also do not utilise indexes, so they should also be avoided and can instead be replaced by equivalent OR statements. Some tips for indexing on the other hand are that indexes should not be used when the table is small, or if the majority of the rows in a table is needed to execute a query, or a column transformation is used. A tip to force a query to use an index is to use the ORDER BY operation [21].

The way to optimise long queries is by avoiding multiple full table scans and reducing the size of the result as soon as possible. Indexes are not needed here and should not be used. Another tip is that hash join is most likely the better algorithm for joining long queries. And if GROUP BY is used by a long query, the filtering needs to be applied first in most cases. There are times that GROUP BY can reduce the size of the data-set, but the rule of thumb is to apply the SELECT statements first for the optimiser. Lastly, set operations can sometimes be used to prompt alternative execution plans. Another tip to improve execution time for queries is to use materialised views, but a materialised view should only be created if the data it is based on does not update often. This means that if it is not very critical to have up-to-date data, the data in the materialised view is read often, and if many queries could make use of it [21].

Lastly, multidimensional and spatial searches are discussed. Spatial data often require range queries. Which means finding all the data located at a certain distance or closer to a specified point in space. And nearest-neighbour queries, which is to find a variable number of objects closest to the specified point. These queries cannot be supported by one-dimensional indexes or even multiple indexes, and must instead use special indexes, such as the GiST.

5.1.2 Other experiments

In the report 'Comparison of physical tuning techniques implemented in two opensource DBMSs' [44] B-tree indexes and hash indexes were investigated to see how they affected execution time in two DBMSs. The result showed that the average time reduced for PostgreSQL with a B-tree index was 67.4% and that the hash index increased execution time for the queries tested [44]. This study is complemented by the report 'PostgreSQL database performance optimization' [45], whose result can be seen in Figure 2.3. What can be seen is that in most cases indexes improved performance, although hashing, joining, retrieving, and sorting sometimes increased execution time. The PostgreSQL optimiser was also investigated by running the ANALYZE command for queries, the result was that no major difference existed by doing this, compared to just running the queries [45].

Another study, investigating indexes and how they affect execution time is the report 'MongoDB vs PostgreSQL: a comparative study on performance aspects' [46]. The result showed that PostgreSQL performed on average 89 times faster with an index applied [46]. A similar study was done in the report 'Comparing Oracle and PostgreSQL, performance and optimization' [47] which showed that PostgreSQL can improve up to 91% with indexes and by only adding primary and foreign keys the performance was improved by 38% and by adding indexes it was improved by 88% [47].

Lastly, the report 'Space-partitioning Trees in PostgreSQL: Realization and Performance' [48] investigated how SP-GiST indexes compared to other tree-based indexes. The result showed that a disk based SP-GiST trie performed two orders of magnitudes better than a B+-tree when it comes to regular expression match searches. The reason for the result was due to the fact that the B+-tree used the wildcard in the search. The trie on the other hand uses non-wildcard characters in the search for filtering. The SP-GiST trie had better search performance than the B+-tree when it came to exact match, around 150% better, it also scaled better than the B+-tree. For prefix matches the B+-tree outperformed the SP-GiST trie, this was due to the inherent nature of having the keys sorted in the leaf nodes. Which allows the tree to answer prefix match queries very efficiently. For exact matches the B+-tree scales better as well, this is due to how the SP-GiST trie consists of more nodes and more node splits than the B+-tree. The SP-GiST kd-tree performed 300% better than the R-tree when it came to point search and 125% better when it came to range search, although the R-tree has better insertion time and better index size. This is due to how the kd-tree has a node size (bucket size) of one and

every insertion causes a node split. The R-tree had better insertion and search performance than the SP-GiST PMR quadtree. Lastly, the nearest neighbour search for the kd-tree and the point quadtree was better than for the trie. This is due to how the trie performs the NN search character by character while for the kd-tree and the point quadtree the NN search is based on partitions.

5.2 Results

In Figure 5.1, Figure 5.2 and, Figure 5.5 the baseline, improved query, and using the baseline query on the personalised B-tree schema can be seen. The result for using the ANALYZE command is omitted due to how it remained the same as the query result, the same reason applies to the generic B-tree and Hash indexes.

In Figure 5.3 there is also the addition of the generic B-tree and Hash indexes, the optimiser result is once again omitted for the same reason stated before.

In Figure 5.4 only the baseline query and the personalised B-tree results can be seen. The same reason for omitted results stands as for the first paragraph. The reason that there is no improved query result, is due to how that was not tested here.

Detailed results for how the queries were executed, the EXPLAIN output, and more detailed graphs can be seen in the appendix E and D.

5.2.1 Other results

The materialised view took less than a minute to build for all data sets. It should be noted that different materialised views were tried and some of them took too long to build in the largest data set. Towards 6 minutes before being manually terminated.

Using the ANALYZE command for the optimiser did not show a very big difference compared to executing the queries normally. Some queries on some data sets had the same execution time, others were a millisecond more or less.

The Hash index could not be built for certain columns, due to the nature of Hash structures, so that could not be tested. And for the general indexes, the index was not used in most queries, which is why that result is omitted.

Further information about the result can be seen in the appendix, both more detailed graphs -in appendix D - for each query executed and the explanation from the DBMS, using the EXPLAIN command - in appendix E.

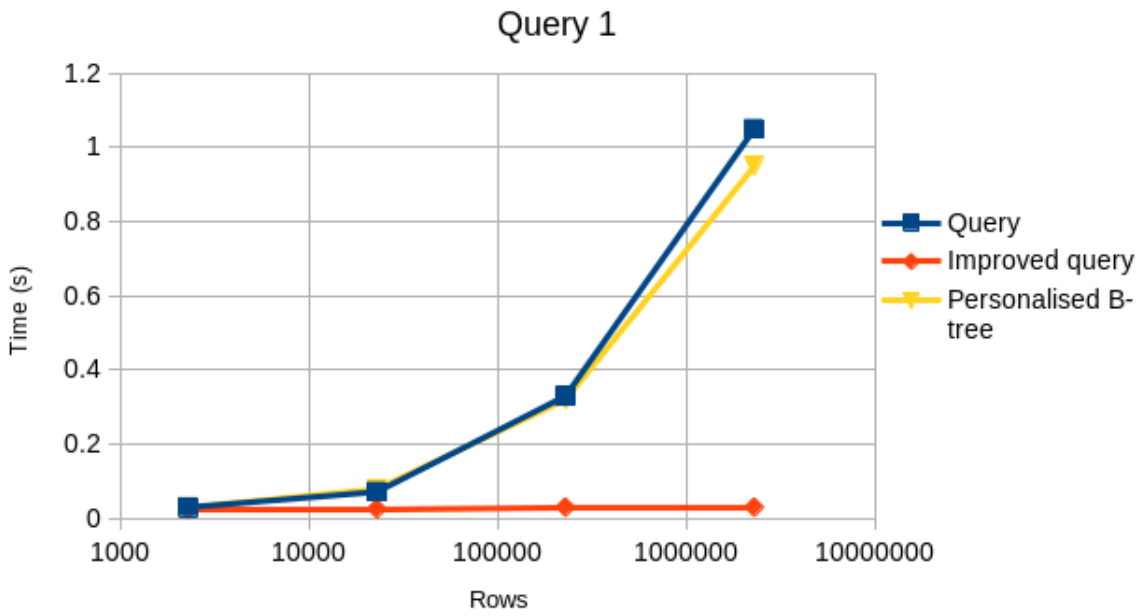

Figure 5.1: Execution time comparison for query 1 versions.

The graph shows the execution times (y-axis) for the query executed, the improved query, and the query executed with a B-tree index and how they scale over increased data in the table. The x-axis shows the rows in the table with logarithmic growth. The query executed with the B-tree shows similar performance to just executing the query, but has a slight improvement when scaled to a larger data set. The result for the general indexes and the query executed with the ANALYZE command were omitted due to how they did not show any difference from just executing the query.

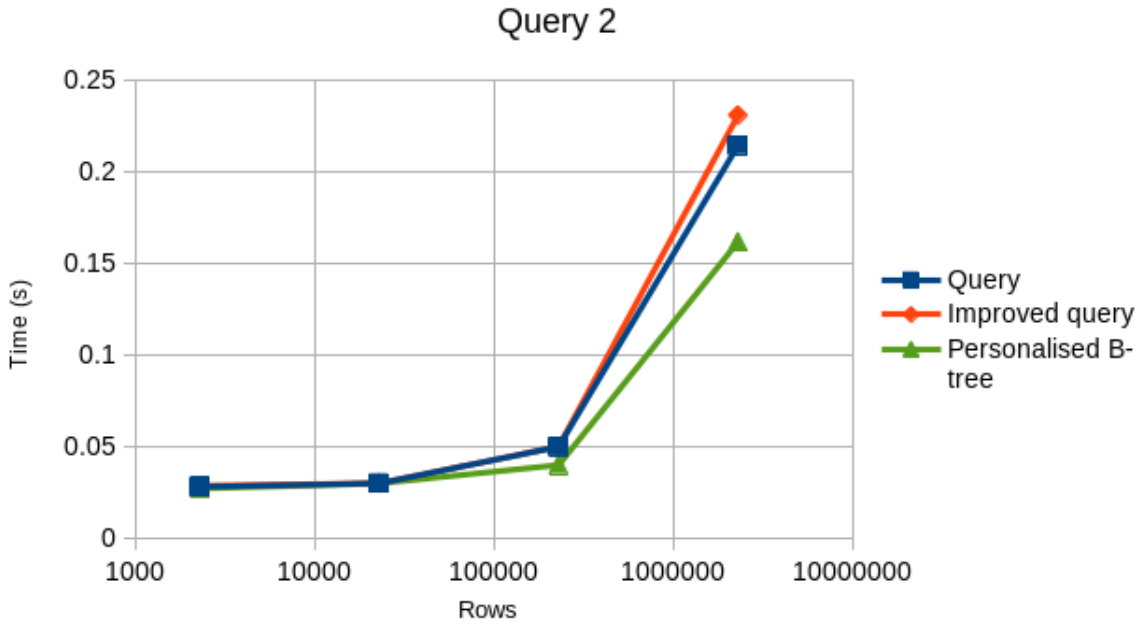

Figure 5.2: Execution time comparison for query 2 versions.

The graph shows the execution times (y-axis) for the query executed, the improved query, and the query executed with a B-tree index and how they scale over increased data in the table. The x-axis shows the rows in the table with logarithmic growth. The query executed with the B-tree shows similar performance to just executing the query, but has a slight improvement when scaled to a larger data set. The improved query also has similar performance but scales worse. The result for the general indexes and the query executed with the ANALYZE command were omitted due to how they did not show any difference from just executing the query.

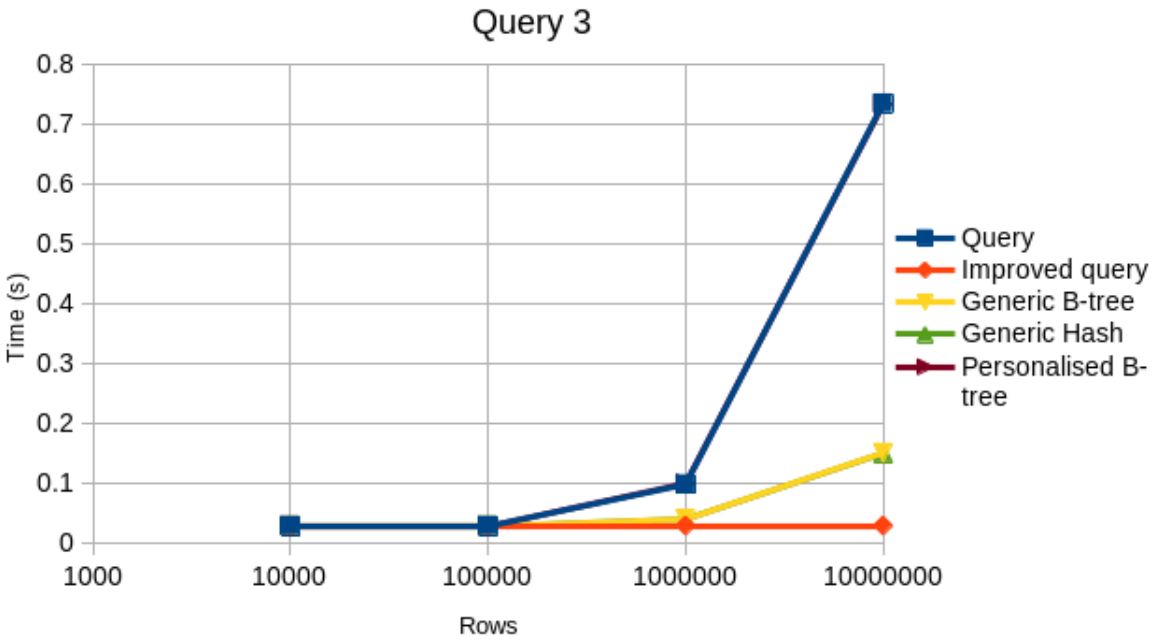

Figure 5.3: Execution time comparison for query 3 versions.

The graph shows the execution times (y-axis) for the query executed, the improved query, the generic B-tree and hash index, and the query executed with a personalised B-tree index and how they scale over increased data in the table. The x-axis shows the rows in the table with logarithmic growth. The generic indexes show similar performance, they scale better than just the query, the same can be said for the improved query. The personalised B-tree line is difficult to see but it is behind the query line, which means that they had similar performance. The ANALYZE performance was omitted due to how it executed like just running the query.

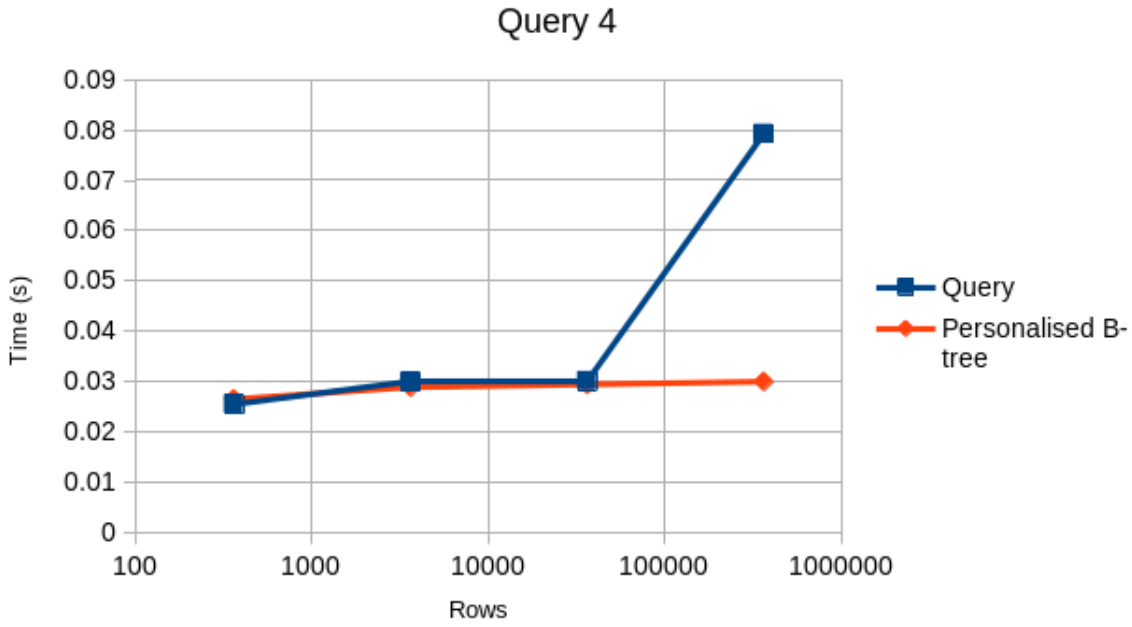

Figure 5.4: Execution time comparison for query 4 versions.

The graph shows the execution times (y-axis) for the query executed, and the query executed with a B-tree index, and how they scale over increased data in the table. The x-axis shows the rows in the table with logarithmic growth. The query executed with the B-tree shows similar performance to just executing the query at first, but shows a large improvement when it comes to scaling. The result for the general indexes and the query executed with the ANALYZE command were omitted due to how they did not show any difference from just executing the query. This query was not tuned, which is why there is no improved query result.

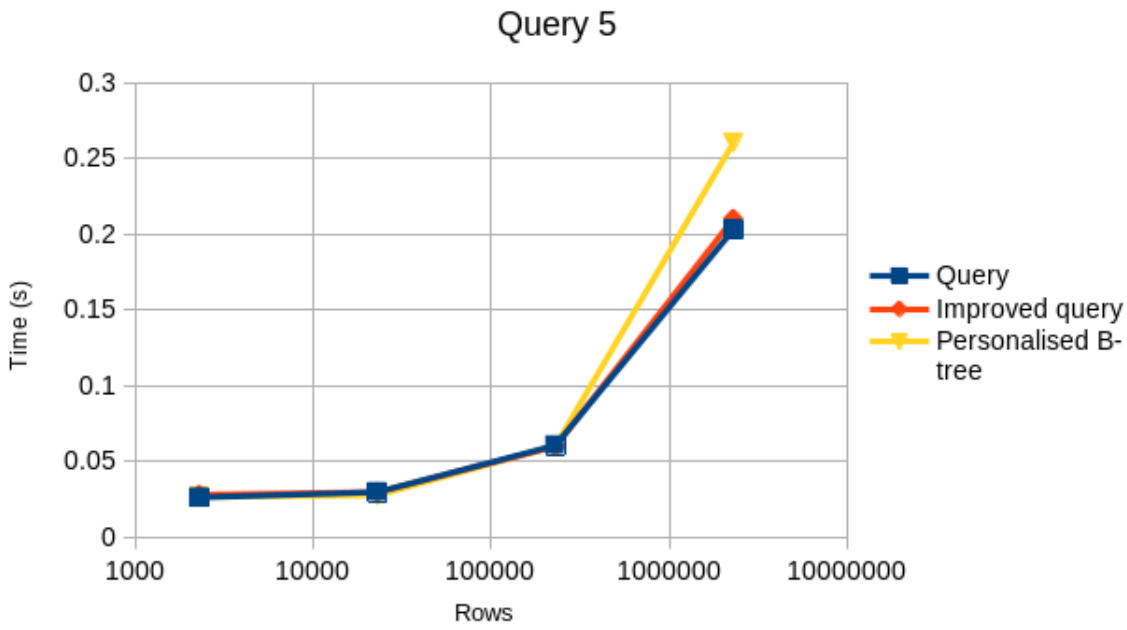

Figure 5.5: Execution time comparison for query 5 versions.

The graph shows the execution times (y-axis) for the query executed, the improved query, and the query executed with a B-tree index and how they scale over increased data in the table. The x-axis shows the rows in the table with logarithmic growth. The query executed with the B-tree shows similar performance to just executing the query, but scales worse. The result for the general indexes and the query executed with the ANALYZE command were omitted due to how they did not show any difference from just executing the query.

Chapter 6

Discussion

This chapter discusses and provides explanations for the result in the experiment by using the information provided in the background as well as the appendix. It compares the result to the literature studies and analyses the reliability and validity of the result. It also discusses the problems faced during the thesis, how they were solved and what problems could not be solved. It brings up the sources of errors to consider as well as the limitations of the result and reiterates what sustainability and ethical effects this result may have.

6.1 The result

This section describes the result of the experiment, combines it with the EXPLAIN output to shine light onto why the result looks like it does, and compares it to other the result from the literature study.

As mentioned in the background, EXPLAIN is used to provide an output of the query plan that the query planner in the DBMS has provided. The output is read bottom-up, and it uses an abstract measurement

Query 1

As can be seen in Figure 5.1, the query and the query executed on the B-tree have similar execution times, but the B-tree query scales slightly better. The improved query on the other hand shows a very big improvement. To better see the result see Figure D.6 in the appendix D. The difference between the normal query and the improved query is the usage of DISTINCT, and IN and OR. From the EXPLAIN output file (in the appendix E), it shows that it filters the IN statement first for any strings that contain any of 'movie' or 'video', as

it contains it as listed values. Then a sequential scan is performed on titles. In comparison, the improved query filters the text using OR statements. A parallel sequential scan is run with two workers, which means that it most likely uses one worker each to scan for either movie or video at the same time. The OR statement seemed to have allowed the planner to use two workers instead of one, which could have led to better performance, or the lack of DISTINCT could have done this further tests would be needed to be sure of what the cause was.

The query using the B-tree index functions similarly to how just running the query works. What differs is how the query uses the B-tree index by performing a bitmap index scan followed by a bitmap heap scan rather than a sequential scan. The index it uses has the column type on the table titles as the indexing column, which means that it has a faster time finding where the correct rows are placed on file, but it still needs to access the data, which is why the bitmap heap scan also is performed.

The Hash index could not be implemented on the types column, mainly due to the nature of Hash structures and how they do not work well with many entries that use the same hash key. The result for the generalised indexes was also omitted from the result as the query did not use the index.

Query 2

In Figure 5.2 the result for executing the query, the improved query, and the query on the B-tree index can be seen. What is most notable is that the improved query was not improved at all. The query differs from the original query by using SELECT(title_id) instead of SELECT(*). The difference in performance is minimal, yet the sources from the literature study stated that changing the * should improve performance. The query performed on the index scaled better.

The EXPLAIN output shows that for the query first a parallel sequential scan is performed, then it is partially hashed based on type, this is then sorted on the type as well, and then finally merged into the result. The improved query shows the same output as the normal query, which means that technically both of them should have the same performance. This can be noted as until the largest data-set was used, the performance was incredibly similar, and for the largest data-set, the performance only differs by less than a millisecond. Which could be a source of error, potentially due to caching. Or it could be an effect of having implemented a materialised view, which might be hogging some memory.

The query performed on the B-tree index uses the index with the types column as the indexing column. It first performs an index-only scan, then groups the types, and finally merges the result. It does this with two workers as well. This explains why this is faster than performing the query without indexes, as an index-only scan is faster than a sequential scan. It also does not perform any type of sorting as the B-tree already has the data sorted which also saves time.

As stated before, the Hash index could not be implemented on the types column, mainly due to the nature of Hash structures and how they do not work well with many entries that use the same Hash key. The result for the generalised indexes was also omitted from the result as the query did not use the index.

Query 3

As Figure 5.3 states, the query executed on the generic B-tree and Hash indexes have the same execution times, and they both show slight improvement compared to executing the query without an index. The tuned query also shows slight improvement when it comes to scaling. The other B-tree index has the same performance as just executing the query. The improved query shows a big improvement compared to the other tests. It should also be noted that building the materialised view for the improved query took less than a minute.

The EXPLAIN output is quite long but can be summarised as the following. The query uses an index condition on the primary key for the people table, it then does an index scan using the primary key on people. After this, the query filters the movie titles and gathers the titles that are searched for. This is done with a parallel sequential scan on titles and then hashed. Another parallel sequential scan is run on crew, the result is then hashed and then a hash join is applied to form the result of the nested loop. This is then sorted by name and the result is merged.

The improved query on the other hand uses an index scan on the primary key for the people table, it uses the cached key for the materialised view (as it also contains person_id) for memoisation purposes. The set is then filtered on the crew conditions and a nested sequential scan is performed on the materialised view. The result is then sorted.

The query on the generic B-tree and Hash index shows similar outputs. They both do an index scan based on the person_id column, they then filter the category on the crew table and use the index condition for title_id (as it is a primary key for titles). They then both do a parallel sequential scan on titles

and uses two nested loop - in comparison to just executing the query which only uses one nested loop - and then sorts to gather the result.

Lastly, the personalised B-tree performs an index scan using the person_id column, it then filters the title and performs a parallel sequential scan on the titles table. It then parallel hashes the result from this part of the query. Then it continues on to filter the crew table for the rows needed, does a parallel hash join - like just performing the query does - and then sorts. This means that the personalised B-tree performs almost exactly like just performing the query, except for the usage of the implemented index instead of using the primary key constraint. Which would explain why they have the same execution times.

Query 4

Figure 5.4 shows the result for the query and the query using the B-tree index. What can be seen is that by using the index the query scales a lot better.

The query begins with performing a parallel sequential scan on the ratings table with a variable filter ($3). This is then repeated again, but the result is saved as ratings_2 that compares the result from the previous sequential scan to find the second highest rating. After this another parallel sequential scan is performed to gather all the media with this rating.

When the query is executed with the index, the same thing happens but instead of using sequential scans, index only scans backwards are used instead, and only done twice. This explains how the performance could improve by so much. Like stated earlier, index only scans are a lot quicker than sequential scans.

Query 5

In Figure 5.5 the query, improved query and the query performed on the B-tree index can be seen. The B-tree index scales worse than the query. Whilst the improved query scales slightly better.

The query is executed by first filtering the the premiered column on the desired values, it then performs a parallel sequential scan on titles with this. After this it sorts the result based on the premier dates. The improved query does the exact same thing, which means that technically they should have the same performance.

With the index on the other hand, a bitmap index scan is performed on the types column, the result is then filtered on the premiered column as specified in the query and applied to a bitmap heap scan on the titles table. The result is then sorted. The cost of the bitmap index scan and the bitmap heap

scan combined is lower than the parallel sequential scan, which means that there should be slight improvement. The reason as to why there is not any improvement can be due to different things, it could be that the query planner is inaccurate in the execution times of the planning due to not updated statistic. Other issues, as mentioned in related works, could be that there are elements to this query that the query optimiser does not take into consideration, something that might have to be fixed manually with a statistics object or something else.

The most likely explanation for why the execution time was higher, is due to how at a certain point for selectivity, the heap access (also called index access) has a higher cost than doing a full table scan. Which means that for this result, despite doing an index scan to find the correct entries to save time, the heap scan takes more time than doing a table scan would.

Optimiser results

As stated in the result, the optimiser result was omitted from the graphs. This is due to how they performed almost exactly like just performing the query, sometimes some milliseconds better and sometimes some milliseconds worse. The explanation for that is assumed to be measurement errors as the measuring were done on different times and memory and cache usage could have changed between them. The lack of change in execution time could be because of how the statistics remained the same when just executing the query, so the need to change query plan did not have to happen. For further research, testing the ANALYZE command when executing queries on indexes might be of more interest, as inserting an index could maybe cause the statistics in the query planner to be out of date.

Comparing with the literature study

Overall, the result gathered from the experiment conducted in this thesis matches well with the general consensus of the experiment conducted in the literature study material. In general, the indexes - when used by the query - improved performance. Although, what differed is that there was no difference between using a B-tree index and a Hash index during the specific case tested in the experiment. This is not to say that there cannot be a difference as there was only one case where the Hash index could be tested in the performed experiment. Another difference is that the B-tree index, in two cases, worsened performance. Which can be explained by the theory in the literature study result, and was explained earlier. What was not tested was adding primary and foreign keys, to see how that would affect performance, but what could be seen

in the result was that by having primary keys, a type of index search could still be performed, so it would be far-fetched, to believe that by implementing more of these key constraints in the correct places would improve query execution time. Although that is something that should be further tested. Lastly for the experiment section of the litterateur study, the ANALYZE result showed that there was no major difference between using it or not, which was the same result gathered in this thesis' experiment.

Some of the methods gathered in the literature study were tested when tuning queries to optimise performance. Removing DISTINCT in Query 1 greatly improved performance. Changing * to the column to be counted did not improve performance for Query 2. In the experiment the result showed that it actually worsened performance, although the EXPLAIN output showed that the execution strategy and the predicted time would remain the same - dismissing any idea of there being errors - in this case doing this did not improve the query. Avoiding subqueries was also something that was tested. This was done in Query 3, by creating a materialised view. This improved performance greatly, especially when it came to scaling the query over a larger data-set.

Although idiosyncrasies were not studied in detail, through the result for Query 1, either the lack of DISTINCT or the addition of OR (or both) could have caused the planner to choose a plan that made use of two workers instead of one, which is the most likely reason for improved performance. As that was the main difference in the EXPLAIN output. Another thing that could be seen was the described relation between full table scan, index only scan and index access in Query 3 and Query 5. As the most likely reason as to why the indexes did not improve performance was due to high selectivity, which makes the index access (also called heap access) more inefficient than a full table scan.

6.1.1 Reliability Analysis

As mentioned in the method, the queries were looped 100 times each, four times (the way they were looped can be seen in the appendix for the script in appendix B). The first time the loop was run was excluded as it counted as warming up the cache. This ensured that a mean could be taken from all three of the runs and then that a mean could be calculated from the three data points. This should improve reliability, as if any of the means was vastly different from the others, the data files generated could be inspected to see if one of the runs had drastically different data points than the other runs.

The queries were also tested as they were constructed to ensure that they gathered what they were supposed to so that the measuring would be accurate.

6.1.2 Dependability Analysis

To ensure the correctness of conclusions, comparisons between the experiment and literature study was made to ensure that there was a reasonable explanation for similarities or differences in results. In the experiment this was further ensured by noting down and viewing each testing instance to see if any major discrepancies could be found when timing the execution of the queries.

6.1.3 Validity Analysis

The result measures the execution time for the queries in different circumstances and how it scales to larger data sets. The script shown in the appendix B does this. It used the /user/bin/time package to do this. The queries were tested so that they gathered the intended data in all the cases beforehand so that they were valid. The package to measure time was studied to ensure that it would measure the execution time correctly and was compared to how it was measured when using the Linux time command, as well as the built-in PostgreSQL command for measuring the execution time of queries to ensure that it was accurate, the ANALYZE command was also useful to see the accuracy of measuring. To ensure further validity of the result, the literature study and the experiment result were compared to see if there were any major differences, and explanations were provided based on the information gathered in the background. As well as by providing a print-out of the EXPLAIN command for the result, explanations as to why the result looks like it does has been provided.

6.2 Problems and sources of error

This section discusses the problems that came up during the thesis, as well as the error sources that should be considered when analysing the result.

6.2.1 Problems

One issue at the beginning, that was realised at a later point, was the delimitations. At first, they were too few and too imprecise. Both of these problems were solved as the project went along and the research for the

background and the literature study was found and analysed. The background showed the extent of the area of database optimisation, which caused more delimitations to be formed, and the literature study showed how other studied formulated problems and described the problem area which made it easier to form the research questions for this thesis.

Another issue found during this study was that there are many different indexes for PostgreSQL, as mentioned in the background. The issue was that the information found about them pointed towards their main use case is for a specific type of data - spatio-temporal - or specific types of operations - nearest neighbour search, find coordinates within an area, etc. Due to the time constraints of this thesis, there was no time to experiment with these types of indexes which meant that they were also put as a delimitation for the experiment. Although, a couple of research papers were found within the area of spatio-temporal indexing, in which a few of them contained tests done with PostgreSQL. One of these reports was more relevant than the others and was then put in the literature study result. This was done to have some relevant information about the use cases of one of the indexes, and also how it compares to other indexes used in the same problem area.

More problems were found when the experiment was being planned. First, there was no documentation of the data in the database, therefore, some of the data was looked over to check what each attribute actually meant. As well as to see if there were any key constraints. At first, the constraints were not found at all, because they were at the end of the file. When they were found a plan was made to test to see if they could be changed to include more foreign keys. Due to the nature of the data in the database, and that the data was decided to be split for the multiple versions of the database, making foreign keys was difficult, and had to be foregone entirely. Which meant that only the original key constraints were used for the database and the indexes implemented. Another issue found was some issues with query tuning, since that was somewhat of a novel concept, formulating better queries was a bit difficult and in some cases almost impossible. This can be seen in the case of Query 4, which was not tuned due to a lack of knowledge.

6.2.2 Sources of error

One major source of an error discovered was the lack of ability to clear the cache between query runs. In reality, a database would not be running the same queries on loop, which means that the cache may not have the data needed at all times. There was not a way to do this, due to the nature of Docker only having

a readable operating system environment, which is why the exact times should be taken with a grain of salt. Clearing the cache between queries would not simulate a real database, but would instead measure the worst-case benchmark, which would mean that most likely in a real-life scenario the benchmark would be better than measured. Which means that the result would be more accurate. As mentioned earlier, there were also issues with query tuning, which means that the queries might not have been tuned very well, which means that the result might not show the extent of how the query tuning can improve execution time.

Another smaller source of error is that the running of tests happened during different days, and times of days (query and improved query on one day, and index and optimiser on another). This could cause errors in that cache and memory performance can differ, which could lead to minor execution time differences.

It should also be noted that for Query 3 the two smaller data sets did not return any result, so the scaling could be inaccurate for them. And for Query 4, the ratings table is a lot smaller than the other tables, which means that scaling differences, in the beginning, could be a lot smaller than for the other results. It would be of interest to have a larger ratings table to see if that reasoning is correct or not.

6.3 Limitations

There are some limitations to the result that are worth mentioning. One of them is what was brought up in the background. As mentioned, a database is often not a standalone product, it most often is connected to some sort of application as an interface of sorts, and a server. The result of this thesis does not take into account how application or server issues play into efficiency. Neither does it test hardware, to see how that affects efficiency. Both are limitations that should be taken into consideration when optimising a database.

Another limitation is the fact that other PostgreSQL indexes than the B-tree and Hash index could not be tested. The Hash index could not be tested in all cases either. This means that the extent of improvement that indexes have on a database could not be accurately measured. Although it could be argued that by PostgreSQL using the B-tree as the standard indexing structure, there could be something in that the B-tree most often is suitable for indexing.

6.4 Sustainability

As mentioned in the introduction, optimising a database system has an environmental effect as it reduces the resources a database uses. Shorter response time and efficient use of hardware lead to lessening the total computing time and could reduce the wear on hardware as well as a reduction in energy usage. And an ethical problem that is related to database efficiency, is the potential that people more easily can manage to compile data from different data sets. This can then be presented or used to discern information that can cause privacy issues.

Chapter 7

Conclusions and Future work

This chapter summarises the result and the information discussed in the discussion chapter, as well as answering the research questions, and sub-questions posed.

7.1 Conclusion

The purpose of this thesis was to investigate how indexing and query optimisation affect the response time for a PostgreSQL database, with the purpose of furthering research in the area, as well as providing information for database administrators and students alike. As one of the aims was to provide course material for database courses.

To summarise the findings of the experiment and the literature study, the research question and the subquestions are answered below.

7.1.1 Answering the subquestions

The subquestions posed for this thesis is the following:

1. What methods of indexing are there and what are their use cases?
2. How does query optimisation work and how can queries be optimised?
3. What is the overlap between indexing and query optimisation?
4. How does indexing, the query optimiser, and query tuning compare to each other?

Subquestion 1

As discovered in the background and literature study, there are many types of indexes in PostgreSQL. The methods of implementing indexes differ depending on if they are primary or secondary indexes. As in PostgreSQL, only secondary indexes are used, the focus will lay there to answer this question. The method to implement indexes is to look at the queries, analyse their frequency, and what type of queries they are. As well as looking at the table to see what type of data there is on there and how often it gets updated. Depending on the data types and index structure that should be chosen, this also depends on the types of queries that are supposed to use the index. Thereafter it can be determined what should go into the index if it should be sparse or dense. And also if the index needs to index all data in a table, if it does not a partial index can be used. If the query uses multiple tables, or other columns than what is indexed, determine if a composite index can be used.

The use-cases of indexes are mainly determined by what indexing structures are used. In most cases, the type of query or data type can determine what index should be used. For example, as mentioned in the background, the Hash index is suitable for point, and multi-point queries. Which B-trees also are good for but are also extended to include range queries, prefix matches, and ordering queries. SP-GiST, GiST, GIN and BRIN are mostly used for implementing special data-types into a database. In the literature study SP-GiST was described to mainly be used for spatio-temporal data, and depending on how these indexes are implemented - i.e what data structures are used - they can be useful for different types of queries. This result recommended their implementations of SP-GiST trie for regular expression matches, exact matches B+-trees for prefix match queries, the SP-GiST kd-tree for point search and range searches, but if insertion and index size is of critical nature, the R-tree works better. This also is the reason to use an R-tree over a SP-GiST PMR quadtree. Nearest neighbour searches also benefit from using a kd-tree implementation.

A generalisation based on the gathered result would be that using a B-tree index is more versatile and suits more situations than using a hash index would be, but if implemented incorrectly could instead slow down the execution time. Removing DISTINCT from a query where possible makes the scaling of a query a lot better than using the operation. On smaller data sets (in this thesis tables with less than 100 000 rows) rarely show a difference in execution time no matter if an index is implemented or if a query is tuned.

Subquestion 2

Query optimisation can be separated into two parts, query tuning, and using the query optimiser. The query optimiser is part of the DBMS and works with statistics over the database, and the query planner to ensure that a good query plan is chosen. This is done by looking at specific factors, such as CPU cycles and I/O accesses, combining them to a single unit, and then comparing this unit between plans. The PostgreSQL optimiser can update the statistics by running the ANALYZE command for a query, as well as be improved by implementing supported statistical objects - for multivariate statistics. This is necessary as there are use cases where the optimiser does not work well, such as for correlated columns in queries, etc. The query planner optimises a query by setting up a plan tree, with plan nodes, in which each plan node contains the cost of planned execution (in the special unit). To not have infinite plans, and ensure that the optimised query is the equivalent of the starting query, heuristics rules are used.

Query tuning on the other hand uses techniques and the skills of the query writer. It is done by manually rewriting queries, to better make use of the database resources. This is entirely based on the knowledge that the query writer has about the database and the query language used. As different types of queries benefit from different optimisation techniques. Summarised techniques from the literature study result are:

- Do not use DISTINCT unless necessary.
- Avoid subqueries as much as possible, especially correlated subqueries.
- Temporaries can cause execution to be slow, but can also subvert the need for using ORDER BY operations.
- Do not use HAVING if WHERE is enough.
- Depending on the system, some operations can cause the query to not use indexes. These idiosyncrasies need to be studied.
- Ordering in the FROM statement can influence the ordering of JOINs, especially if more than five tables are joined.
- The use of views can lead to writing inefficient queries.
- Index-only scans are always faster than full table scans, but index access can be slower than full table scans if the selectivity of the query is high.

- Short queries benefit from using restrictive indexes, especially when the indexes are unique as well.
- Doing a column transformation can cause indexes to not be used.
- ORDER BY can force the query to use an index.
- Long queries, do not benefit from indexes, and instead are optimised by ensuring that few full table scans are done. It is also beneficial to reduce the size of the result as soon as possible.
- Materialised views are good for improving execution time if it is not critical for the query to have fully up-to-date data.

Based on the experiment result, using an OR statement instead of an IN operation could also potentially improve performance, although more tests would be needed to verify that.

Subquestion 3

From the information stated, indexing and queries are incredibly entwined. The purpose of both query optimisation and indexing is to improve efficiency. Although, this can be done in different ways. Indexing can be used for the ordering of files, which would be one of the main differences. Another difference is that because of how indexes need to be implemented on the database as an auxiliary structure, query optimisation can be a less invasive procedure to use when improving execution time on a sensitive database. Such as for databases that cannot afford more memory allocation, or have their tables changing often. From the experiment, it can also be seen that, in the case of the experiment, it is only so much an index can do if the query is bad. So query optimisation and indexing have areas where they both are entwined to have good execution time. The query optimiser is always running as well, although the accuracy can be improved by specific operations based on data type, query, and other factors. This means that the optimiser overlap with both indexes and query tuning.

Subquestion 4

From subquestion 3, it can then be argued to mean that one method cannot be superior to the others, as something like this cannot be generalised. It all depends on the situation. How the database looks if the database structure can change, how much memory is available, and if there is a priority to

queries. Although, based on the result we can split it up into some cases. Implementing indexes for spatio-temporal data improves execution time for queries - this should be complemented with seeing how query optimisation affects it though. B-tree indexes are more well-rounded in their use cases, and from the experiment worked really well for improving a correlated subquery. Query tuning worked really well for a nested query (by using a materialised view), as well as for a large query (selecting many rows in a table) - which was also stated in the literature study as long queries benefit more from query optimisation than indexes. Based on the literature study result, in cases of column transformation query optimisation works better. And for short queries, using indexes is more beneficial.

7.1.2 The research question

The research question summarises the subquestions. Indexing and query optimisations affect the response time positively if implemented correctly, as can be seen, both in the literature study and the experiment conducted for this thesis report. Although, there are cases where indexes can increase execution time. In the literature study, this happened during hashing records, retrieving records from a specific table, joining certain tables, and sorting. In the case of the experiment conducted in this thesis: Query 5 (range query) and also, in part Query 3 (nested join query), which happened due to incorrect usage of indexes. Tuning queries, on the other hand, had two cases of showing great performance improvement, one of them being Query 3 (with the materialised view) and Query 1 (removing DISTINCT), which concludes that materialised views improves the execution time, and scales very well, but needs to be weighed against the cost of creating and maintaining it. The other two cases showed a lack of improvement, but this most likely was due to a lack of tuning knowledge. The query optimiser (ANALYZE) on the other hand did not affect response time majorly. This means that depending on the case response time can be affected positively or negatively - or not at all - by implementing indexes or query optimisation techniques.

7.2 Future work

For further research, ideas of interest can be seen in the following list.

- Testing more of the different query types that are mentioned in [43], to see how they interact with indexes, the optimiser, and query tuning.

- Testing the different statistical elements in the optimiser.
- Have larger data sets and different types of data, to be able to generalise conclusions.
- See how normalisation affects execution time.
- See how key constraints affect execution time.
- Implement one of the other PostgreSQL indexes, to see how they affect performance.
- Testing the cost of indexes by using update or remove operations on a table, as well as testing the cost of updating a materialised view. To better understand their use-cases.

This is mainly motivated by filling in the gaps for the limitations of the result in this thesis. Having this information, and more information in general, would make a stronger case for the conclusions of this thesis. As well as, further mapping out more information about optimisation techniques in general. As of this thesis, it was somewhat difficult to find other published research focusing specifically on PostgreSQL and how to optimise a database within it. Complementing this thesis with any of the above suggestions would contribute to having more detailed and specific information for the PostgreSQL community.

7.3 Reflections

This chapter describes some reflections of the works, suggestions towards others, what I would change about the works, and the impact of the work done. As well as some other thoughts about the project.

7.3.1 Thoughts about the work

During the course of this works, I found out and learned a lot more in-depth about databases, as well as how to conduct a research project. What I also found, was a lack of official, or published research about this area in particular. It was difficult to conduct the literature study as most materials were not very similar to the work being done in this thesis. So my suggestion for others working within database systems would be to publish more detailed information about optimisation techniques and their explanations. I would

say that doing the pre-study was an integral part of this thesis, so for other thesis students, I would recommend conducting a pre-study to collect basic knowledge about what information is out there within their research area. To ensure that what they are doing is possible, and within the delimitations.

Some things I would change if I were to redo this work are to summarise the literature study result before conducting the experiment, as it would have saved me more time than having to go back and forth in the report to find the information I need. As well as, when writing, spending that time actually formulating and editing as I go, instead of writing the necessary information and then having to go back and edit large sections at a time. I believe that it would have been faster if I had spent the time writing it better the first time. This potentially could have given me more time for the experiment, so that I could have tested more scenarios.

7.3.2 Impact

The impact of the result of this thesis, I believe is somewhat small on a socio-econimic scale. I think it could have a larger impact on students as it summarises a lot of information, and test it on a specified database language. Which they then could use for their own learning purposes. I also believe that it could potentially help database administrators that have started working with PostgreSQL. I believe that if continued, this research has the potential of having a high impact on the PostgreSQL community in the sense of making it even more available. This could lead to more people, and companies using PostgreSQL for their relational databases.

As mentioned in the discussion and the background, the impact of optimisation can improve environmental costs. Partly by less usage of hardware leads to less wear, and also software optimisation could lead to needing to upgrade hardware less. Another environmental improvement would be that needing less time for execution could lead to less energy usage overall. This could also be argued to help companies to keep unnecessary costs down.

References

[1] R. Elmazri and N. B. Shamkant, *The fundamentals of database systems*. Pearson, 2016.

[2] H. E. Williams and D. Lane, *Web Database Applications with PHP & MySQL*. O'Reilly Media, 2002-04-16, [Online] https://www.oreilly.com/library/view/web-database-applications/0596005431/ch01.html, (Accessed: 2021-09-01).

[3] M. Bakni. (2017-08-02) Client-server 3-tier architecture. [Online] https://commons.wikimedia.org/wiki/File:Client-Server_3-tier_architecture_-_en.png, (Accessed: 2021-10-06).

[4] N. Fialkovskaya. (2021-01-08) Speed test. [Online] https://sitechecker.pro/speed-test/, (Accessed: 2021-09-01).

[5] S. O'dea. Average internet connection speed in the us. [Online] https://www.statista.com/statistics/616210/average-internet-connection-speed-in-the-us/, (Accessed: 2021-08-24).

[6] Oracle. What is a database? [Online] https://www.oracle.com/database/what-is-database/, (Accessed: 2021-08-25).

[7] IBM Cloud Education. Relational databases. [Online] https://www.ibm.com/cloud/learn/relational-databases, (Accessed: 2021-09-01).

[8] GeeksforGeeks. (2021-06-28) Dbms set 1. [Online] https://www.geeksforgeeks.org/introduction-of-dbms-database-management-system-set-1/, (Accessed: 2021-09-17).

[9] Ian. (2016-06-06) What is a database schema? [Online] https://database.guide/what-is-a-database-schema/, (Accessed: 2021-09-17).

[10] PostgreSQL Global Development Group. Postgresql documentation introduction. [Online] https://www.postgresql.org/docs/13/intro-whatis.html, (Accessed: 2021-09-03).

[11] ——. Architectural fundamentals. [Online] https://www.postgresql.org/docs/13/tutorial-arch.html, (Accessed: 2021-09-03).

[12] ——. Sql concepts. [Online] https://www.postgresql.org/docs/13/tutorial-concepts.html, (Accessed: 2021-09-03).

[13] ——. Advanced features: foreign keys. [Online] https://www.postgresql.org/docs/13/tutorial-fk.html, (Accessed: 2021-09-03).

[14] ——. Constraints. [Online] https://www.postgresql.org/docs/8.3/ddl-constraints.html#DDL-CONSTRAINTS-FK, (Accessed: 2021-09-17).

[15] PostgreSQL Tutorial. Postgresql tutorial. [Online] https://www.postgresqltutorial.com/, (Accessed: 2021-10-21).

[16] ——. Postgresql like. [Online] https://www.postgresqltutorial.com/postgresql-like/, (Accessed: 2021-10-21).

[17] ——. Postgresql in. [Online] https://www.postgresqltutorial.com/postgresql-in/, (Accessed: 2021-10-21).

[18] ——. Postgresql subquery. [Online] https://www.postgresqltutorial.com/postgresql-subquery/, (Accessed: 2021-10-21).

[19] Geeks for geeks. Sql correlated subqueries. [Online] https://www.geeksforgeeks.org/sql-correlated-subqueries/, (Accessed: 2021-10-21).

[20] PostgreSQL Global Development Group. Views. [Online] https://www.postgresql.org/docs/13/tutorial-views.html, (Accessed: 2021-09-04).

[21] H. Dombrovskaya, B. Novikov, and A. Bailliekova, *PostgreSQL query optimization: the ultimate guide to building efficient queries*. Apress, 2021.

[22] PostgreSQL Global Development Group. Materialised views. [Online] https://www.postgresql.org/docs/current/rules-materializedviews.html, (Accessed: 2021-09-04).

[23] GeeksforGeeks. (2021-09-07) File organization in dbms. [Online] https://www.geeksforgeeks.org/file-organization-in-dbms-set-1/, (Accessed: 2021-09-17).

[24] PostgreSQL Global Development Group. Indexes: introduction. [Online] https://www.postgresql.org/docs/13/indexes-intro.html, (Accessed: 2021-09-03).

[25] ——. Index types. [Online] https://www.postgresql.org/docs/13/indexes-types.html, (Accessed: 2021-09-03).

[26] ——. Multicolumn indexes. [Online] https://www.postgresql.org/docs/13/indexes-multicolumn.html, (Accessed: 2021-09-03).

[27] Ta bu shi da yu . (2005-06-17) B-tree index. [Online] https://en.wikipedia.org/wiki/File:Btree_index.PNG, (Accessed: 2021-09-27).

[28] PostgreSQL Global Development Group. Hash indexes. [Online] https://www.postgresql.org/docs/13/hash-intro.html, (Accessed: 2021-09-04).

[29] J. Stolfi. (2009-04-10) Hash table. [Online] https://commons.wikimedia.org/wiki/File:Hash_table_5_0_1_1_1_1_1_LL.svg, (Accessed: 2021-10-06).

[30] PostgreSQL Global Development Group. Gist indexes. [Online] https://www.postgresql.org/docs/13/gist-intro.html, (Accessed: 2021-09-03).

[31] ——. Operator classes and operator families. [Online] https://www.postgresql.org/docs/9.5/indexes-opclass.html, (Accessed: 2021-09-18).

[32] ——. Sp-gist indexes. [Online] https://www.postgresql.org/docs/13/spgist-intro.html, (Accessed: 2021-09-03).

[33] ——. Gin indexes. [Online] https://www.postgresql.org/docs/13/gin-intro.html, (Accessed: 2021-09-03).

[34] ——. Brin indexes. [Online] https://www.postgresql.org/docs/13/brin-intro.html, (Accessed: 2021-09-03).

[35] ——. Combining indexes. [Online] https://www.postgresql.org/docs/13/indexes-bitmap-scans.html, (Accessed: 2021-09-03).

[36] ——. Partial indexes. [Online] https://www.postgresql.org/docs/13/indexes-partial.html, (Accessed: 2021-09-03).

[37] ——. Index-only scans. [Online] https://www.postgresql.org/docs/13/indexes-index-only-scans.html, (Accessed: 2021-09-03).

[38] ——. Multi-version concurrency control. [Online] https://www.postgresql.org/docs/7.1/mvcc.html, (Accessed: 2021-09-18).

[39] ——. Query planner. [Online] https://www.postgresql.org/docs/13/using-explain.html, (Accessed: 2021-09-04).

[40] ——. Query planner statistics. [Online] https://www.postgresql.org/docs/13/planner-stats.html, (Accessed: 2021-09-04).

[41] ——. Oins and the query planner. [Online] https://www.postgresql.org/docs/13/explicit-joins.html, (Accessed: 2021-09-04).

[42] S. J. Kamatkar, A. Kamble, A. Viloria, L. Hernandez-Fernandez, and E. Garcia, "Database performance tuning and query optimization," in *Lecture notes in computer science 10943 - Data mining and big data*, 2018, pp. 3–11.

[43] D. Sasha and P. Bonnet, *Database tuning principles, experiments and troubleshooting techniques*. Morgan Kaufman, 2002.

[44] F. Oyvind, "Comparison of physical tuning techniques implemented in two open source dbmss," 2005.

[45] Q. Wang, "Postgresql database performance optimization," 2011.

[46] A. Makris, K. Tserpes, G. Spiliopoulus, D. Zissis, and D. Anagnostopoulos, "Mongodb vs postgresql: a comparative study on perfomance aspects," 2020.

[47] P. Martins, P. Tomé, C. Wanzeller, F. A. Sá, and M. Abbasi, "Comparing oracle and postgresql, performance and optimization," in *Trends and applications in information systems and technologies, vol. II*, 2021, pp. 3–11.

[48] M. Y. Eltabakh, R. Eltarras, and W. G. Aref, "Space-partitioning trees in postgresql: Realization and performance," in *Proceedings of the 22nd International Conference on Data Engineering*, 2006, [Online] https://www.cerias.purdue.edu/assets/pdf/bibtex_archive/01617468.pdf, (Accessed: 2021-10-21).

[49] A. Håkansson, "Portal of research methods and methodologies for research projects and degree projects," in *WORLDCOMP'13 - The 2013 World Congress in Computer Science, Computer Engineering, and Applied Computing*, 2013.

[50] N. B. Nkomo and J. Lihanda, "Qualitative and quantitative methodology," 2010-05-21, [Online] https://www.academia.edu/44204575/QUALITATIVE_AND_QUANTITATIVE_METHODOLOGY, (Accessed: 2021-09-21).

[51] D. R. Tomas, "A general inductive approach for analyzing qualitative evaluation data," 2006-06, [Online] https://journals.sagepub.com/doi/pdf/10.1177/1098214005283748, (Accessed: 2021-09-21).

[52] IBM Cloud Education. (2021-06-23) Docker. [Online] https://www.ibm.com/cloud/learn/docker, (Accessed: 2021-10-07).

Appendix A

The database schema

```
--
-- PostgreSQL database dump
--

-- Dumped from database version 13.0 (Debian
   13.0-1.pgdg100+1)
-- Dumped by pg_dump version 13.0 (Debian 13.0-1.
   pgdg100+1)

SET statement_timeout = 0;
SET lock_timeout = 0;
SET idle_in_transaction_session_timeout = 0;
SET client_encoding = 'UTF8';
SET standard_conforming_strings = on;
SELECT pg_catalog.set_config('search_path', '',
   false);
SET check_function_bodies = false;
SET xmloption = content;
SET client_min_messages = warning;
SET row_security = off;

SET default_tablespace = '';

SET default_table_access_method = heap;

--
-- Name: akas; Type: TABLE; Schema: public; Owner:
   postgres
--

CREATE TABLE public.akas (
```

```
28      title_id character varying NOT NULL, --PRIMARY
    KEY
29      title character varying,
30      region character varying,
31      language character varying,
32      types character varying,
33      attributes character varying,
34      is_original_title integer
35  );
36
37
38  ALTER TABLE public.akas OWNER TO postgres;
39
40  --
41  -- Name: crew; Type: TABLE; Schema: public; Owner:
    postgres
42  --
43
44  CREATE TABLE public.crew (
45      title_id character varying, --REFERENCES public
    .akas
46      person_id character varying, --REFERENCES
    public.people
47      category character varying,
48      job character varying
49  );
50
51
52  ALTER TABLE public.crew OWNER TO postgres;
53
54  --
55  -- Name: episodes; Type: TABLE; Schema: public;
    Owner: postgres
56  --
57
58  CREATE TABLE public.episodes (
59      episode_title_id character varying NOT NULL, --
    PRIMARY KEY
60      show_title_id character varying, --REFERENCES
    public.akas
61      season_number integer,
62      episode_number integer
63  );
64
65
66  ALTER TABLE public.episodes OWNER TO postgres;
67
```

```
68 --
69 -- Name: people; Type: TABLE; Schema: public; Owner
       : postgres
70 --
71
72 CREATE TABLE public.people (
73     person_id character varying NOT NULL, --PRIMARY
       KEY
74     name character varying,
75     born integer,
76     died integer
77 );
78
79
80 ALTER TABLE public.people OWNER TO postgres;
81
82 --
83 -- Name: ratings; Type: TABLE; Schema: public;
       Owner: postgres
84 --
85
86 CREATE TABLE public.ratings (
87     title_id character varying NOT NULL, --
       REFERENCES public.akas
88     rating double precision,
89     votes integer
90 );
91
92
93 ALTER TABLE public.ratings OWNER TO postgres;
94
95 --
96 -- Name: titles; Type: TABLE; Schema: public; Owner
       : postgres
97 --
98
99 CREATE TABLE public.titles (
100     title_id character varying NOT NULL, --
       REFERENCES public.akas
101     type character varying,
102     primary_title character varying,
103     original_title character varying,
104     is_adult integer,
105     premiered integer,
106     ended integer,
107     runtime_minutes integer,
108     genres character varying
```

```
109 );
110
111
112 ALTER TABLE public.titles OWNER TO postgres;
113
114 --
115 -- Data for Name: akas; Type: TABLE DATA; Schema:
    public; Owner: postgres
116 --
```

Keys

```
1 --
2 -- Name: people people_pkey; Type: CONSTRAINT;
    Schema: public; Owner: postgres
3 --
4
5 ALTER TABLE ONLY public.people
6     ADD CONSTRAINT people_pkey PRIMARY KEY (
    person_id);
7
8
9 --
10 -- Name: ratings ratings_pkey; Type: CONSTRAINT;
    Schema: public; Owner: postgres
11 --
12
13 ALTER TABLE ONLY public.ratings
14     ADD CONSTRAINT ratings_pkey PRIMARY KEY (
    title_id);
15
16
17 --
18 -- Name: titles titles_pkey; Type: CONSTRAINT;
    Schema: public; Owner: postgres
19 --
20
21 ALTER TABLE ONLY public.titles
22     ADD CONSTRAINT titles_pkey PRIMARY KEY (
    title_id);
```

Appendix B

The script template

The commented lines (5-7) were used when the ANALYZE command was run, this was to ensure that the latest statistics were used every time the script was run.

The commented line 21 did not work due to permission errors, as mentioned in the report.

```
1  #!/bin/bash
2
3  #execute ./loop1 when in the right docker image
4  LIMIT=100
5  #-------------------------------------------#
6  # Uncomment to execute the sql files that has the
       ANALYZE
7  # command
8  #-------------------------------------------#
9
10 #psql -U postgres -d imdb -f amovies.sql > /dev/
       null 2>&1
11 #psql -U postgres -d imdb -f atypes.sql > /dev/null
        2>&1
12 #psql -U postgres -d imdb -f ajoin.sql > /dev/null
       2>&1
13 #psql -U postgres -d imdb -f asecondhigh.sql > /dev
       /null 2>&1
14 #psql -U postgres -d imdb -f ainterval.sql > /dev/
       null 2>&1
15
16 for (( i = 0; i < LIMIT; i++ ));
17
18 do
19 #FORMAT BELOW
20 #/usr/bin/time -o <outputfile> -a -f %e psql -U <
```

```
    username docker> -d <database name in docker> -f
     <name of query file> > /dev/null 2>&1
#-a -f %e flags has to do with the /usr/bin/time
    package and how it formats time output
# > /dev/null 2>&1 throws the sql output into null,
     so it does not show in the terminal

/usr/bin/time -o movies1.txt -a -f %e psql -U
    postgres -d imdb -f movies.sql > /dev/null 2>&1
/usr/bin/time -o types1.txt -a -f %e psql -U
    postgres -d imdb -f types.sql > /dev/null 2>&1
/usr/bin/time -o join1.txt -a -f %e psql -U
    postgres -d imdb -f join.sql > /dev/null 2>&1
/usr/bin/time -o secondhigh1.txt -a -f %e psql -U
    postgres -d imdb -f secondhigh.sql > /dev/null
    2>&1
/usr/bin/time -o interval1.txt -a -f %e psql -U
    postgres -d imdb -f interval.sql > /dev/null
    2>&1

#sync && echo 1 > /proc/sys/vm/drop_caches #drops
    cache?
done;
echo -ne '\n'
```

Appendix C

Indexes

B-tree indexes

The commented indexes are the generic indexes that were first tested.

```
--CREATE INDEX titles_b ON public.titles USING BTREE(title_id);
--CREATE INDEX akas_b ON public.akas USING BTREE(title_id);
--CREATE INDEX crew_b ON public.crew USING BTREE(title_id);
--CREATE INDEX people_b ON public.people USING BTREE(person_id);
--CREATE INDEX ratings_b ON public.ratings USING BTREE(title_id);
--CREATE INDEX episodes_b ON public.episodes USING BTREE(show_title_id);

CREATE INDEX titles_b ON public.titles USING BTREE(type);
CREATE INDEX titlesprem_b ON public.titles USING BTREE(premiered);
CREATE INDEX akas_b ON public.akas USING BTREE(title_id);
CREATE INDEX crew_b ON public.crew USING BTREE(category);
CREATE INDEX people_b ON public.people USING BTREE(person_id);
CREATE INDEX ratings_b ON public.ratings USING BTREE(rating);
CREATE INDEX episodes_b ON public.episodes USING BTREE(show_title_id);
```

Hash indexes

The commented indexes are the personalised indexes that could not be generated for the large database.

```
1  CREATE INDEX titles_b ON public.titles USING HASH(
       title_id);
2  CREATE INDEX akas_b ON public.akas USING HASH(
       title_id);
3  CREATE INDEX crew_b ON public.crew USING HASH(
       title_id);
4  CREATE INDEX people_b ON public.people USING HASH(
       person_id);
5  CREATE INDEX ratings_b ON public.ratings USING HASH
       (title_id);
6  CREATE INDEX episodes_b ON public.episodes USING
       HASH(show_title_id);
7
8  --CREATE INDEX titles_b ON public.titles USING HASH
       (type);
9  --CREATE INDEX titlesprem_b ON public.titles USING
       HASH(premiered);
10 --CREATE INDEX akas_b ON public.akas USING HASH(
       title_id);
11 --CREATE INDEX crew_b ON public.crew USING HASH(
       category);
12 --CREATE INDEX people_b ON public.people USING HASH
        (person_id);
13 --CREATE INDEX ratings_b ON public.ratings USING
       HASH (rating);
14 --CREATE INDEX episodes_b ON public.episodes USING
       HASH (show_title_id);
```

Appendix D

Detailed graphs

D.0.1 Baseline test

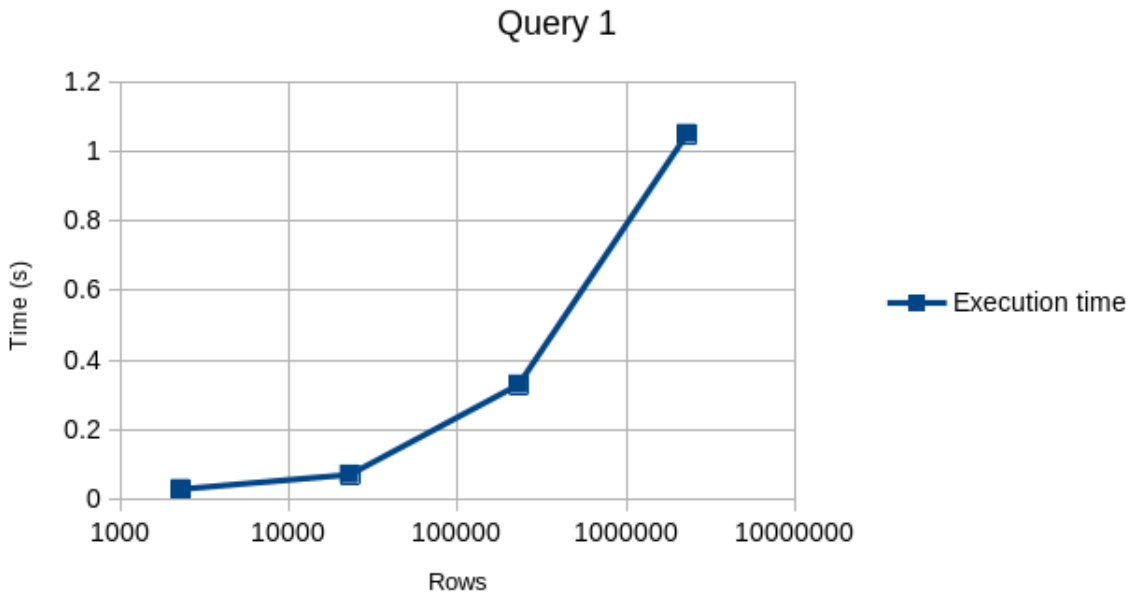

Figure D.1: Execution time for query 1.

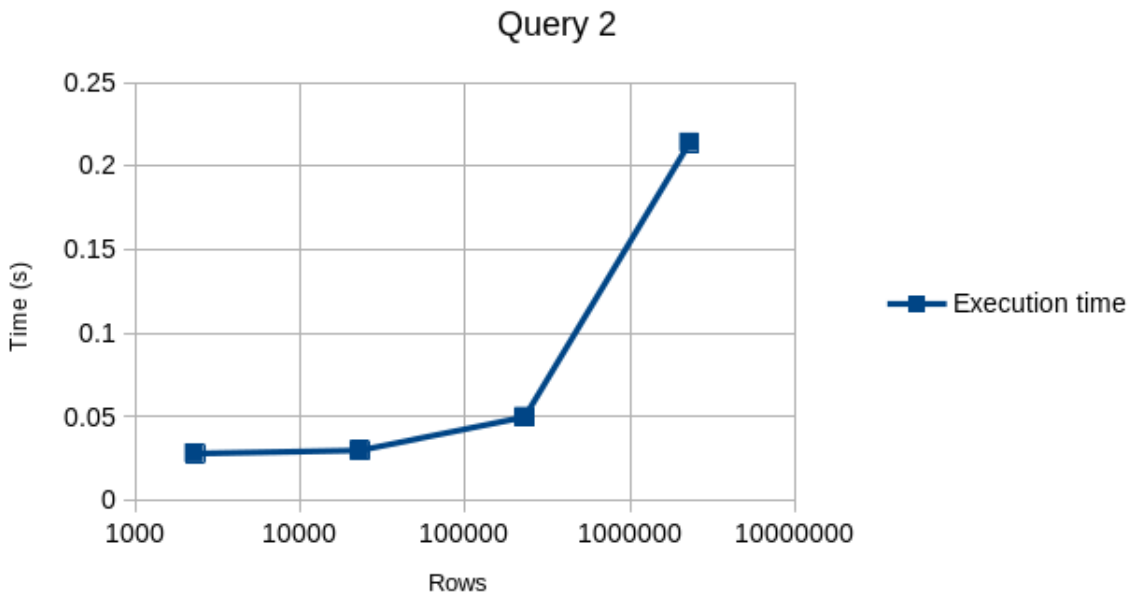

Figure D.2: Execution time for query 2.

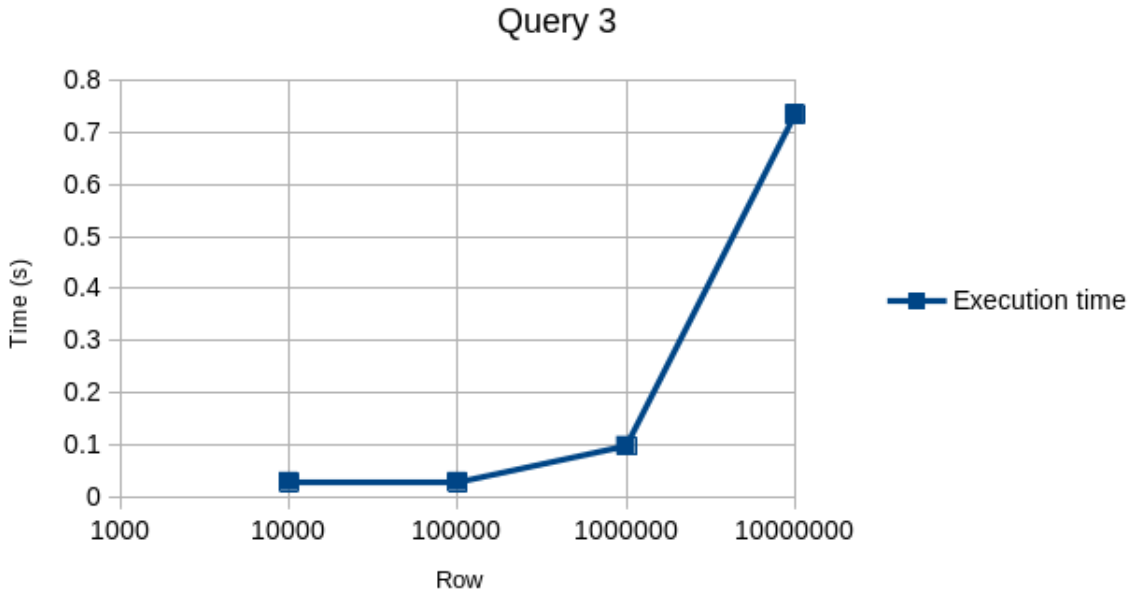

Figure D.3: Execution time for query 3.

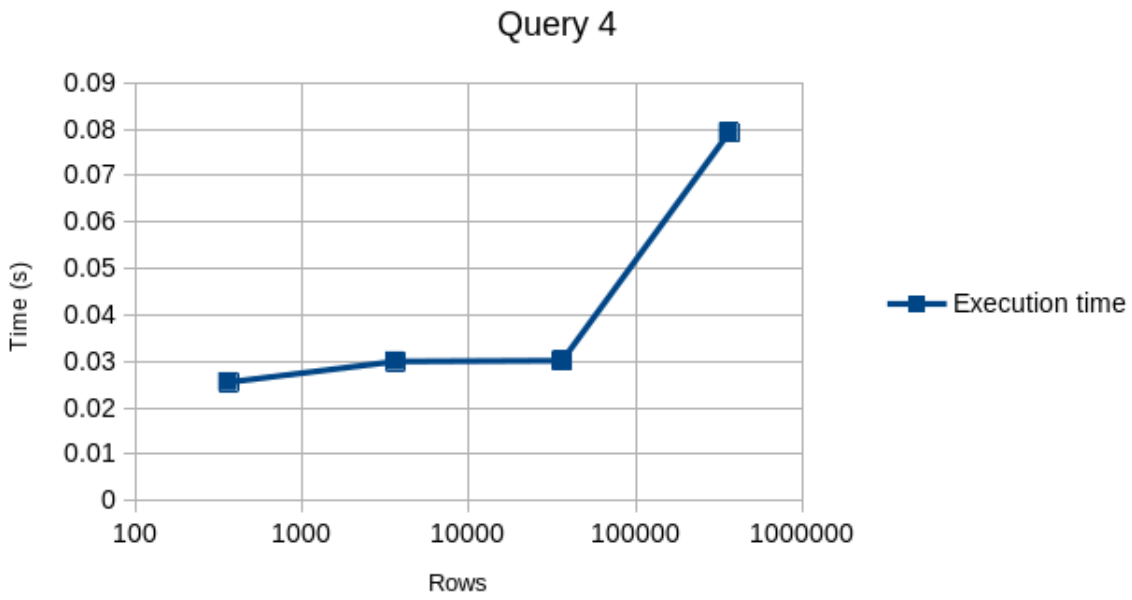

Figure D.4: Execution time for query 4.

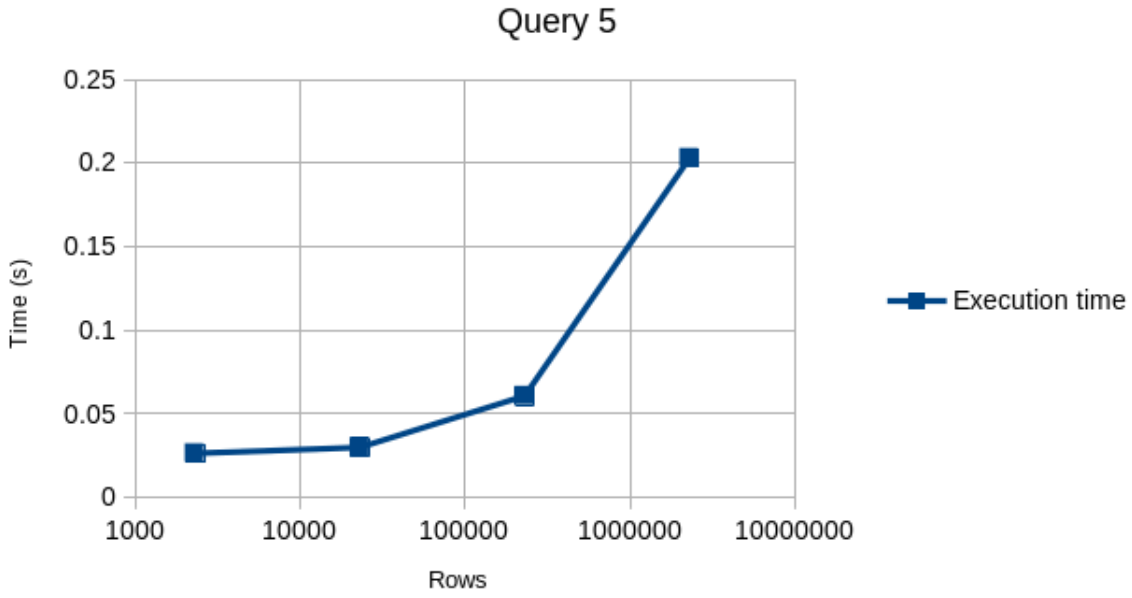

Figure D.5: Execution time for query 5.

D.0.2 Improved queries

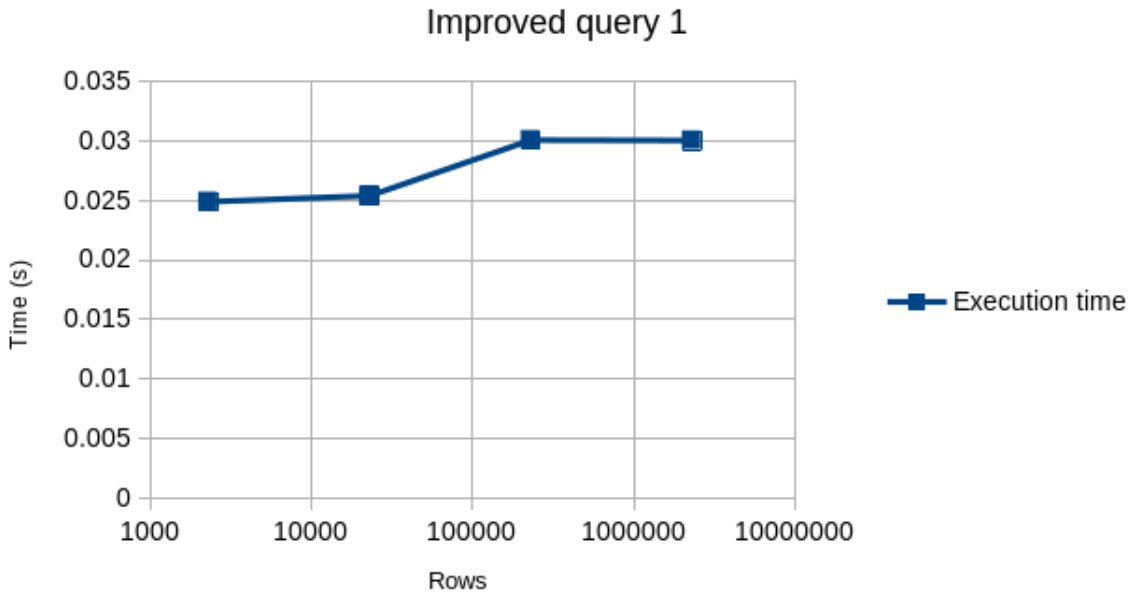

Figure D.6: Execution time for the improved query 1.

Figure D.7: Execution time for the improved query 2.

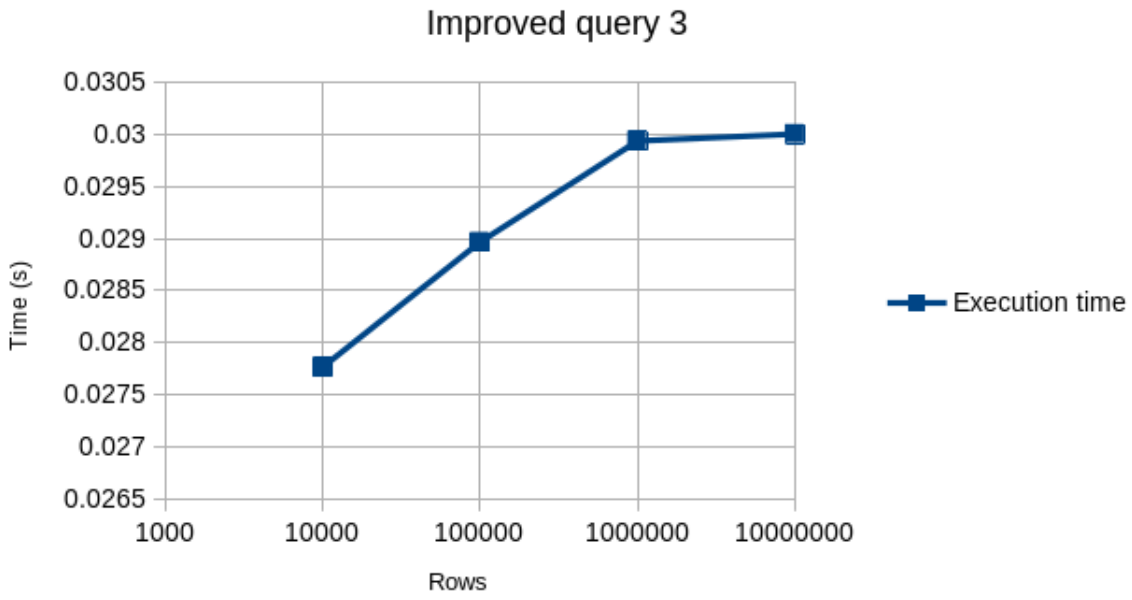

Figure D.8: Execution time for the improved query 3.

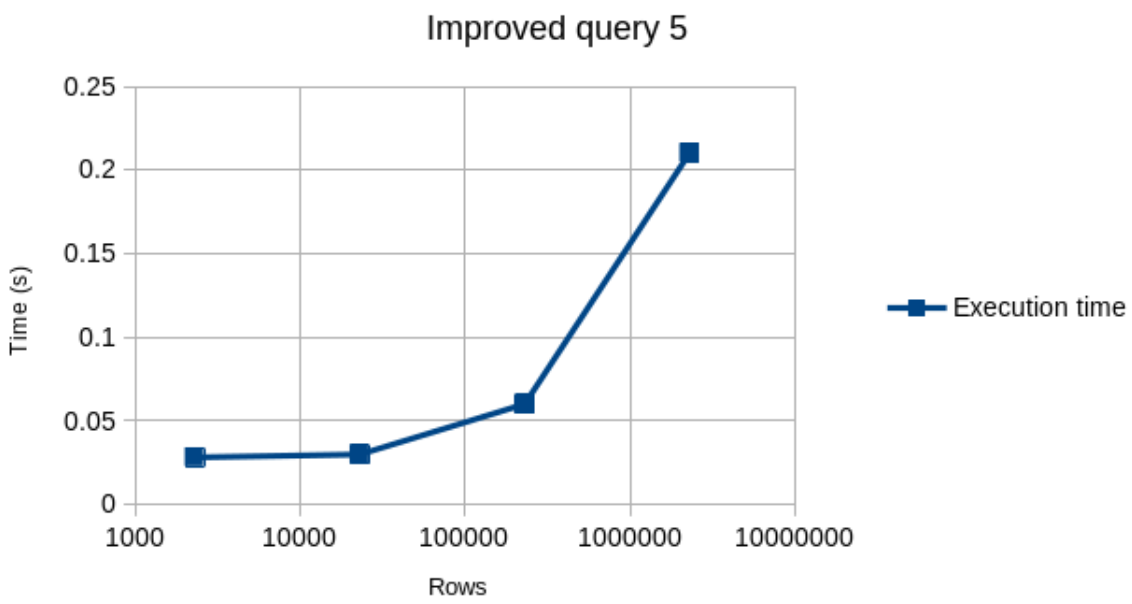

Figure D.9: Execution time for the improved query 5.

D.0.3 Hash index

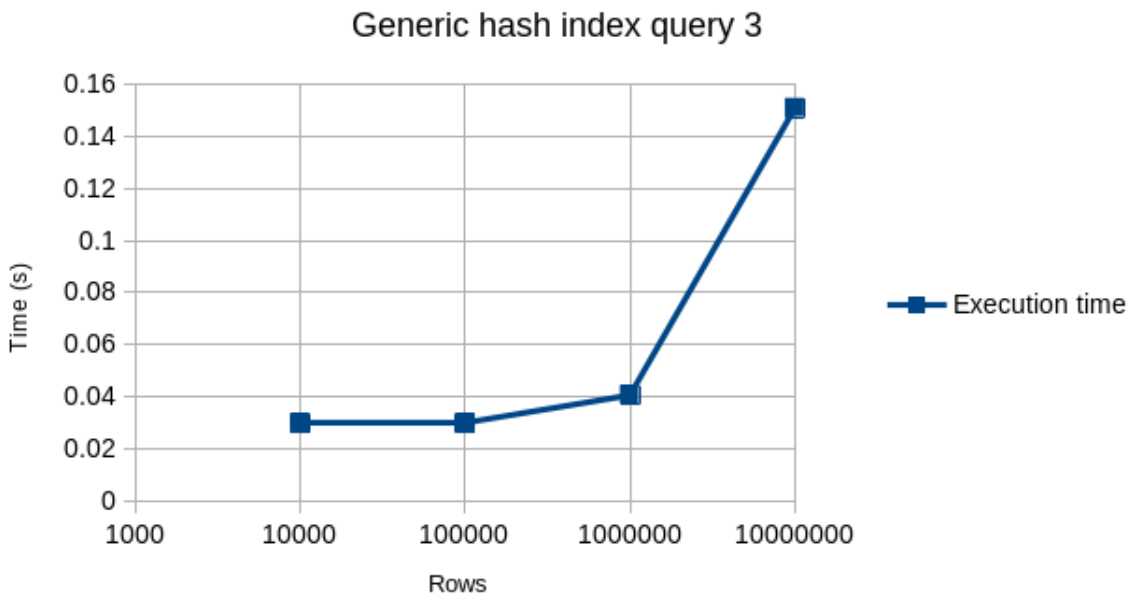

Figure D.10: Execution time for query 3 with Hash index.

D.0.4 B-tree index

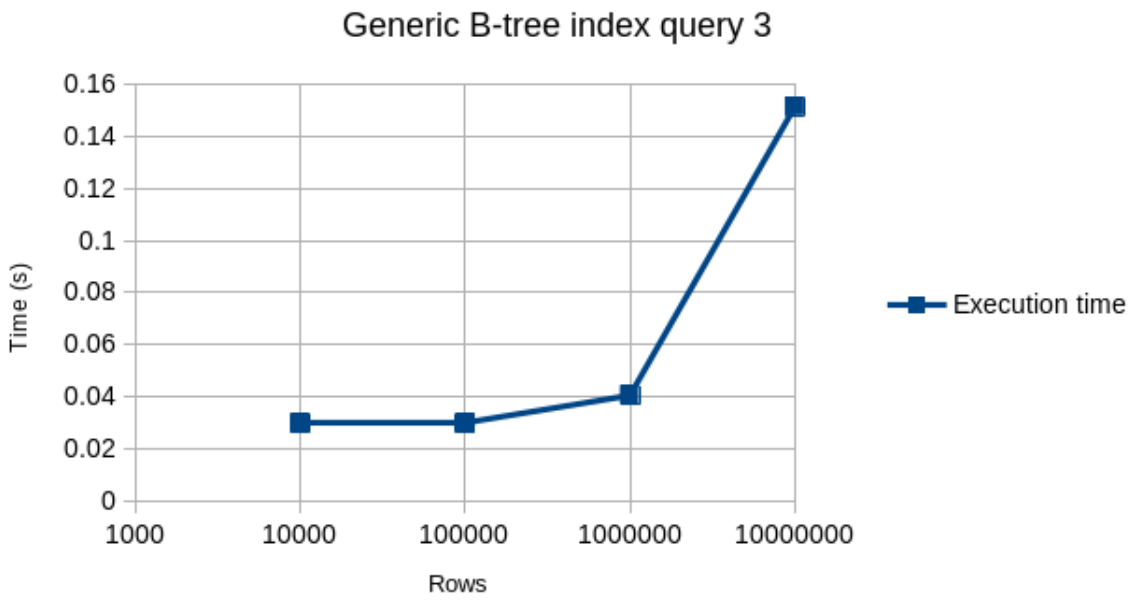

Figure D.11: Execution time for query 3 with B-tree.

Personalised B-tree index

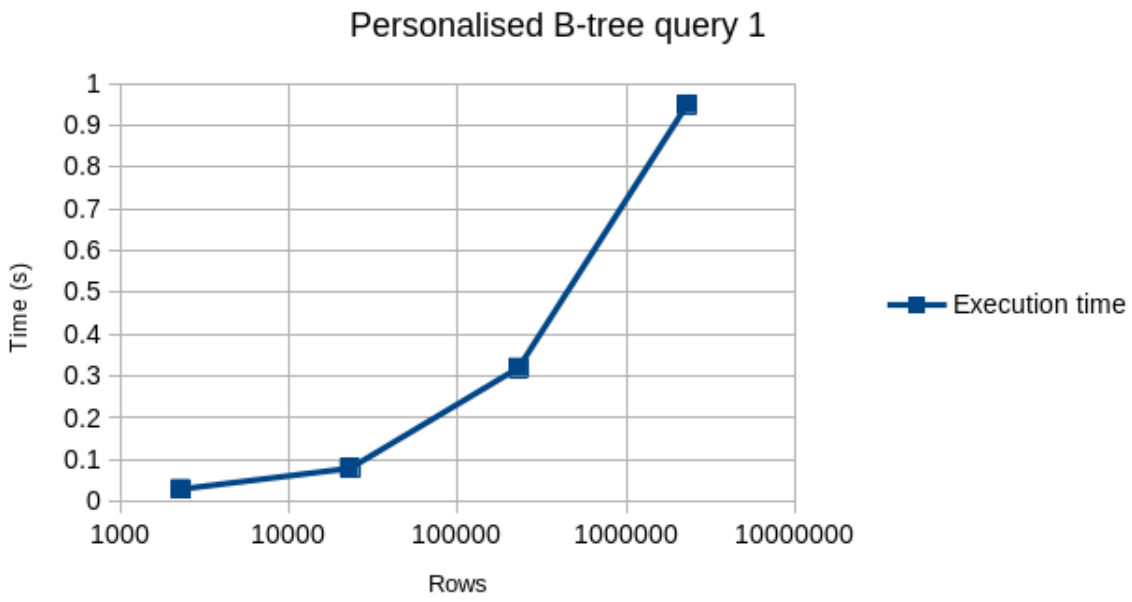

Figure D.12: Execution time for the B-tree index implemented for query 1.

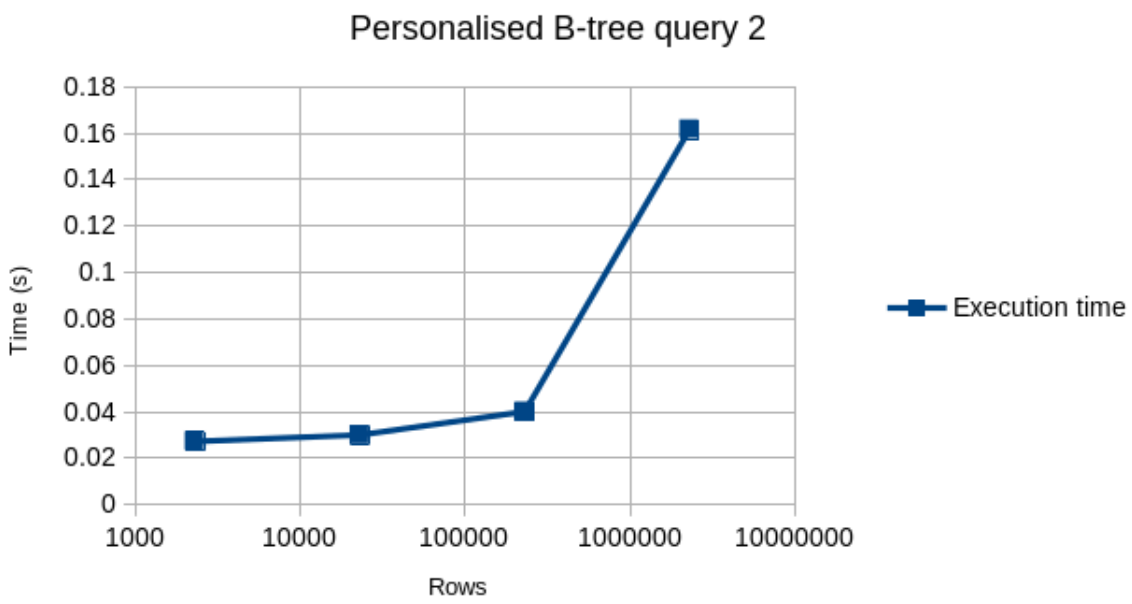

Figure D.13: Execution time for the B-tree index implemented for query 2.

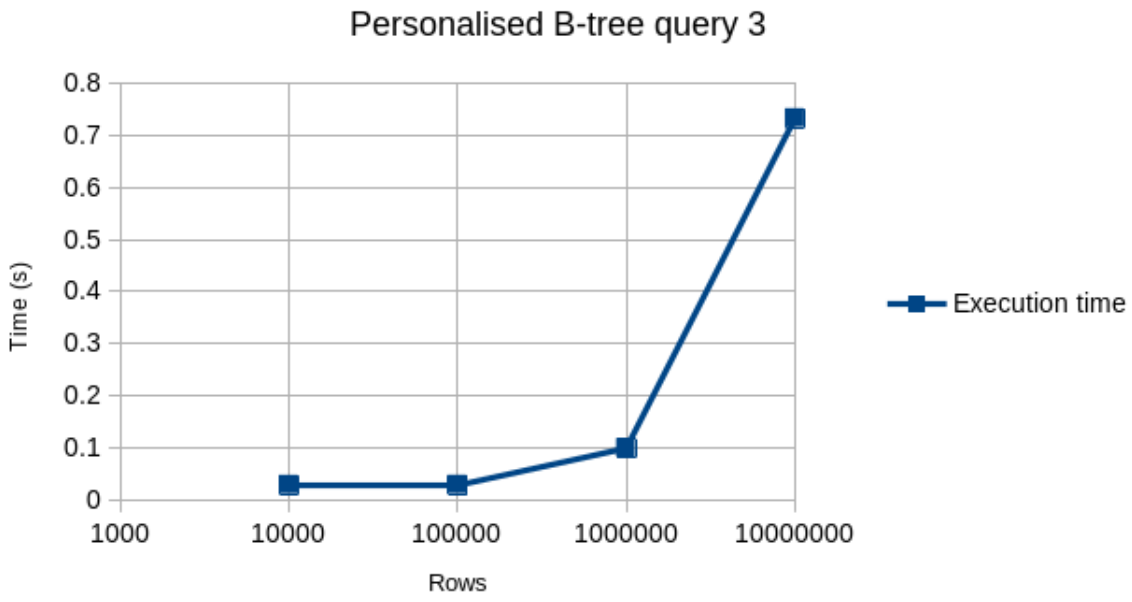

Figure D.14: Execution time for the B-tree index implemented for query 3.

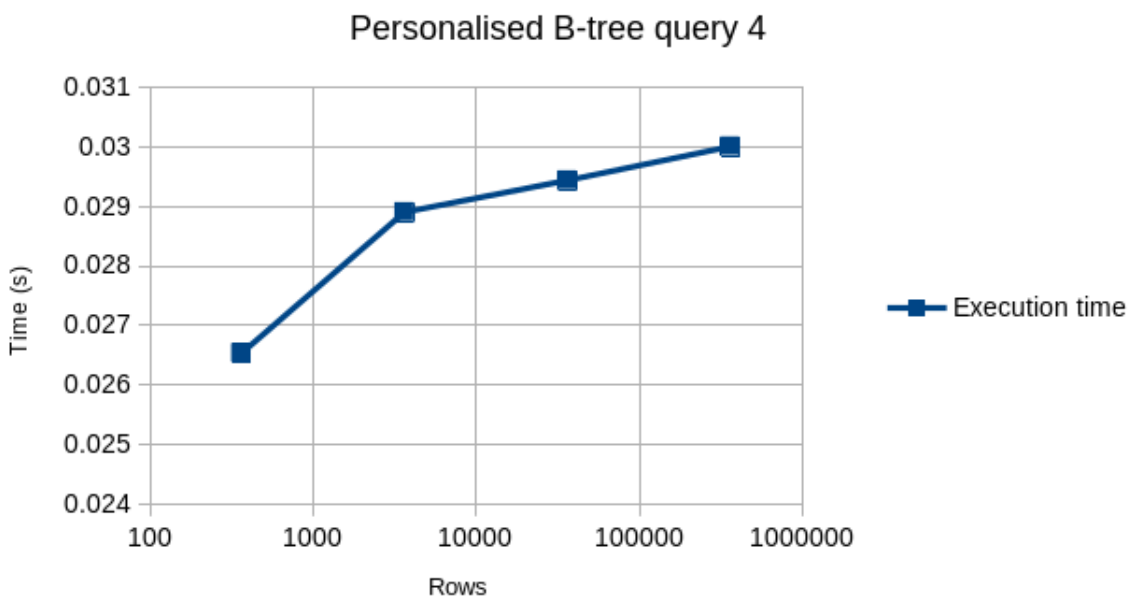

Figure D.15: Execution time for the B-tree index implemented for query 4.

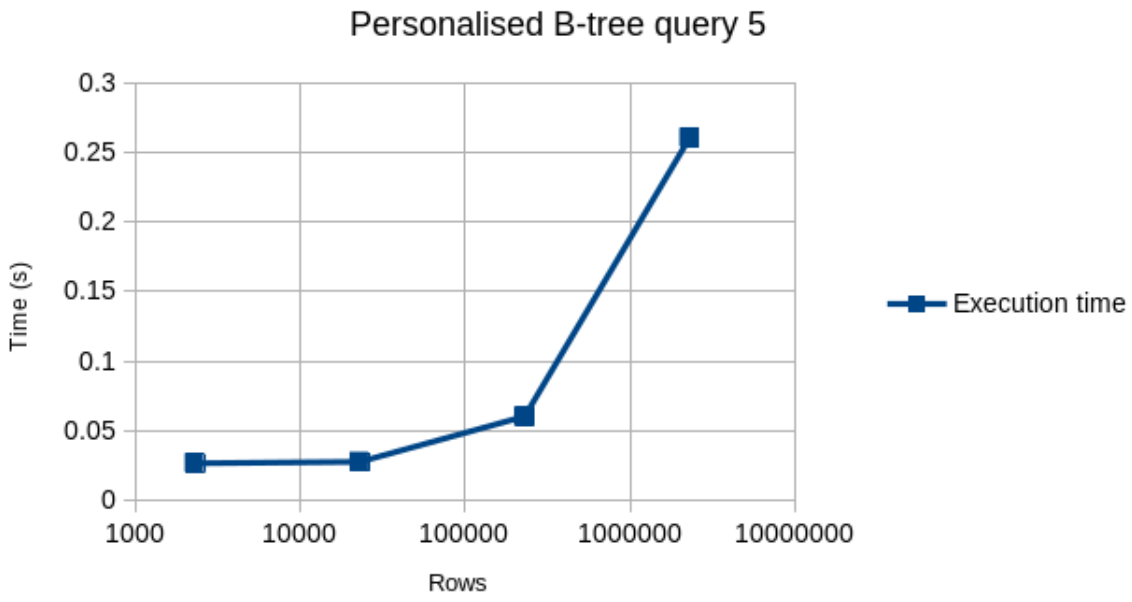

Figure D.16: Execution time for the B-tree index implemented for query 5.

Appendix E

EXPLAIN output

The EXPLAIN output was generated for the largest database only. The reason as to why not all queries show all the tests is due to how information is repeated. For example if an index was created for Query 1 but not used by the query execution plan, then the execution plan would remain the same as running the query without the index.

```
movies (q1)
Aggregate  (cost=63728.93..63728.94 rows=1 width=8)
    ->  Seq Scan on titles  (cost=0.00..63007.91
    rows=288405 width=10)
          Filter: ((type)::text = ANY ('{movie,video
    }'::text[]))

improved:
Finalize Aggregate  (cost=49960.60..49960.61 rows=1
     width=8)
    ->  Gather  (cost=49960.39..49960.60 rows=2
    width=8)
     Workers Planned: 2
          ->  Partial Aggregate  (cost
    =48960.39..48960.40 rows=1 width=8)
                ->  Parallel Seq Scan on titles  (cost
    =0.00..48667.96 rows=116973 width=10)
                      Filter: (((type)::text = 'movie'::
    text) OR ((type)::text = 'video'::text))

personalised btree:
Aggregate  (cost=41827.01..41827.02 rows=1 width=8)
    ->  Bitmap Heap Scan on titles  (cost
    =3172.32..41105.89 rows=288446 width=10)
     Recheck Cond: ((type)::text = ANY ('{movie,
    video}'::text[]))
```

```
18          ->  Bitmap Index Scan on titles_b  (cost
    =0.00..3100.20 rows=288446 width=0)
19          Index Cond: ((type)::text = ANY ('{movie,
    video}'::text[]))
20 ------------------------------------------------------------

21
22 types (q2):
23 Finalize GroupAggregate  (cost=49668.25..49670.78
    rows=10 width=16)
24 Group Key: type
25     ->  Gather Merge  (cost=49668.25..49670.58 rows
    =20 width=16)
26     Workers Planned: 2
27          ->  Sort  (cost=48668.22..48668.25 rows=10
    width=16)
28          Sort Key: type
29              ->  Partial HashAggregate  (cost
    =48667.96..48668.06 rows=10 width=16)
30              Group Key: type
31                  ->  Parallel Seq Scan on titles  (
    cost=0.00..43887.97 rows=955997 width=8)
32
33 improved:
34 Finalize GroupAggregate  (cost=49668.25..49670.78
    rows=10 width=16)
35 Group Key: type
36     ->  Gather Merge  (cost=49668.25..49670.58 rows
    =20 width=16)
37     Workers Planned: 2
38          ->  Sort  (cost=48668.22..48668.25 rows=10
    width=16)
39          Sort Key: type
40              ->  Partial HashAggregate  (cost
    =48667.96..48668.06 rows=10 width=16)
41              Group Key: type
42                  ->  Parallel Seq Scan on titles  (
    cost=0.00..43887.97 rows=955997 width=18)
43
44 personalised btree:
45 Finalize GroupAggregate  (cost=1000.45..34694.65
    rows=10 width=16)
46 Group Key: type
47     ->  Gather Merge  (cost=1000.45..34694.45 rows
    =20 width=16)
48     Workers Planned: 2
```

```
        ->  Partial GroupAggregate  (cost
    =0.43..33692.12 rows=10 width=16)
        Group Key: type
            ->  Parallel Index Only Scan using
    titles_b on titles  (cost=0.43..28911.35 rows
    =956133 width=8)
---------------------------------------------------------------


join (q3)
Unique  (cost=192351.95..192429.28 rows=650 width
    =14)
  ->  Gather Merge  (cost=192351.95..192427.66
    rows=650 width=14)
  Workers Planned: 2
        ->  Sort  (cost=191351.93..191352.61 rows
    =271 width=14)
        Sort Key: people.name
            ->  Nested Loop  (cost
    =48670.51..191340.98 rows=271 width=14)
            ->  Parallel Hash Join  (cost
    =48670.08..191208.32 rows=271 width=10)
            Hash Cond: ((crew.title_id)::text = (
    titles.title_id)::text)
                ->  Parallel Seq Scan on crew  (
    cost=0.00..138537.62 rows=1524040 width=20)
                Filter: (((category)::text = 'actor
    '::text) OR ((category)::text = 'actress'::text)
    )
                    ->  Parallel Hash  (cost
    =48667.96..48667.96 rows=170 width=10)
                    ->  Parallel Seq Scan on titles
      (cost=0.00..48667.96 rows=170 width=10)
                    Filter: (((primary_title)::text
     ~~ 'Spider-Man%'::text) OR ((original_title)::
    text ~~ 'Spider-Man%'::text))
                        ->  Index Scan using
    people_pkey on people  (cost=0.43..0.49 rows=1
    width=24)
                        Index Cond: ((person_id)::
    text = (crew.person_id)::text)
                        JIT:
                        Functions: 19
                        Options: Inlining false,
    Optimization false, Expressions true, Deforming
    true
```

```
improved:
Unique  (cost=524.38..524.70 rows=64 width=14)
  ->  Sort  (cost=524.38..524.54 rows=64 width=14)
      Sort Key: people.name
      ->  Nested Loop  (cost=0.44..522.46 rows=64 width=14)
          ->  Seq Scan on q3  (cost=0.00..5.17 rows=64 width=10)
          Filter: (((category)::text = 'actor'::text) OR ((category)::text = 'actress'::text))
          ->  Memoize  (cost=0.44..8.46 rows=1 width=24)
              Cache Key: q3.person_id
              ->  Index Scan using people_pkey on people  (cost=0.43..8.45 rows=1 width=24)
                  Index Cond: ((person_id)::text = (q3.person_id)::text)

generic btree:
Unique  (cost=51267.26..51344.59 rows=650 width=14)
  ->  Gather Merge  (cost=51267.26..51342.97 rows=650 width=14)
      Workers Planned: 2
      ->  Sort  (cost=50267.24..50267.92 rows=271 width=14)
          Sort Key: people.name
          ->  Nested Loop  (cost=0.86..50256.29 rows=271 width=14)
              ->  Nested Loop  (cost=0.43..50123.63 rows=271 width=10)
              ->  Parallel Seq Scan on titles  (cost=0.00..48669.99 rows=170 width=10)
                  Filter: (((primary_title)::text ~~ 'Spider-Man%'::text) OR ((original_title)::text ~~ 'Spider-Man%'::text))
              ->  Index Scan using crew_b on crew  (cost=0.43..8.53 rows=2 width=20)
                  Index Cond: ((title_id)::text = (titles.title_id)::text)
                  Filter: (((category)::text = 'actor'::text) OR ((category)::text = 'actress'::text))
                  ->  Index Scan using people_b on people  (cost=0.43..0.49 rows=1 width=24)
                      Index Cond: ((person_id)::text = (crew.person_id)::text)
```

```
101
102 generic hash:
103 Unique  (cost=54417.46..54494.79 rows=650 width=14)
104     ->  Gather Merge  (cost=54417.46..54493.16 rows
    =650 width=14)
105     Workers Planned: 2
106         ->  Sort  (cost=53417.43..53418.11 rows=271
     width=14)
107         Sort Key: people.name
108             ->  Nested Loop  (cost=0.00..53406.48
    rows=271 width=14)
109             ->  Nested Loop  (cost=0.00..53389.68
    rows=271 width=10)
110             ->  Parallel Seq Scan on titles  (cost
    =0.00..48669.99 rows=170 width=10)
111             Filter: (((primary_title)::text ~~ '
    Spider-Man%'::text) OR ((original_title)::text
    ~~ 'Spider-Man%'::text))
112                 ->  Index Scan using crew_b on crew
      (cost=0.00..27.74 rows=2 width=20)
113                 Index Cond: ((title_id)::text = (
    titles.title_id)::text)
114                 Filter: (((category)::text = 'actor
    '::text) OR ((category)::text = 'actress'::text)
    )
115                     ->  Index Scan using people_b
    on people  (cost=0.00..0.06 rows=1 width=24)
116                     Index Cond: ((person_id)::text
    = (crew.person_id)::text)
117
118 personalised btree:
119 Unique  (cost=192356.32..192433.64 rows=650 width
    =14)
120     ->  Gather Merge  (cost=192356.32..192432.02
    rows=650 width=14)
121     Workers Planned: 2
122         ->  Sort  (cost=191356.29..191356.97 rows
    =271 width=14)
123         Sort Key: people.name
124             ->  Nested Loop  (cost
    =48672.55..191345.34 rows=271 width=14)
125             ->  Parallel Hash Join  (cost
    =48672.12..191212.68 rows=271 width=10)
126             Hash Cond: ((crew.title_id)::text = (
    titles.title_id)::text)
127                 ->  Parallel Seq Scan on crew  (
    cost=0.00..138539.81 rows=1524093 width=20)
```

```
128                 Filter: (((category)::text = 'actor
    '::text) OR ((category)::text = 'actress'::text)
    )
129                       ->  Parallel Hash  (cost
    =48669.99..48669.99 rows=170 width=10)
130                       ->  Parallel Seq Scan on titles
      (cost=0.00..48669.99 rows=170 width=10)
131                       Filter: (((primary_title)::text
     ~~ 'Spider-Man%'::text) OR ((original_title)::
    text ~~ 'Spider-Man%'::text))
132                             ->  Index Scan using
    people_b on people  (cost=0.43..0.49 rows=1
    width=24)
133                             Index Cond: ((person_id)::
    text = (crew.person_id)::text)
134                             JIT:
135                             Functions: 19
136                             Options: Inlining false,
    Optimization false, Expressions true, Deforming
    true
137 ------------------------------------------------------------

138
139 secondhigh (q4):
140 Gather  (cost=14182.88..19908.84 rows=3981 width=8)
141 Workers Planned: 1
142 Params Evaluated: $3
143 InitPlan 2 (returns $3)
144     ->  Finalize Aggregate  (cost
    =13182.87..13182.88 rows=1 width=8)
145     InitPlan 1 (returns $1)
146           ->  Finalize Aggregate  (cost
    =6327.97..6327.98 rows=1 width=8)
147           ->  Gather  (cost=6327.86..6327.97 rows=1
    width=8)
148           Workers Planned: 1
149                 ->  Partial Aggregate  (cost
    =5327.86..5327.87 rows=1 width=8)
150                 ->  Parallel Seq Scan on ratings
    ratings_1  (cost=0.00..4795.09 rows=213109 width
    =8)
151                 ->  Gather  (cost=6854.78..6854.89 rows
    =1 width=8)
152                 Workers Planned: 1
153                 Params Evaluated: $1
154                       ->  Partial Aggregate  (cost
    =5854.78..5854.79 rows=1 width=8)
```

```
155                 ->  Parallel Seq Scan on ratings
    ratings_2  (cost=0.00..5327.86 rows=210767 width
    =8)
156                 Filter: (rating <> $1)
157                     ->  Parallel Seq Scan on
    ratings  (cost=0.00..5327.86 rows=2342 width=8)
158                     Filter: (rating = $3)
159
160 personalised btree:
161 Unique  (cost=1.35..115.01 rows=91 width=8)
162 InitPlan 4 (returns $3)
163     ->  Result  (cost=0.91..0.92 rows=1 width=8)
164     InitPlan 2 (returns $1)
165         ->  Result  (cost=0.45..0.46 rows=1 width
    =8)
166         InitPlan 1 (returns $0)
167             ->  Limit  (cost=0.42..0.45 rows=1
    width=8)
168             ->  Index Only Scan Backward using
    ratings_b on ratings ratings_1  (cost
    =0.42..10328.41 rows=362285 width=8)
169             Index Cond: (rating IS NOT NULL)
170             InitPlan 3 (returns $2)
171                 ->  Limit  (cost=0.42..0.45 rows=1
    width=8)
172                 ->  Index Only Scan Backward using
    ratings_b on ratings ratings_2  (cost
    =0.42..11234.12 rows=358304 width=8)
173                 Index Cond: (rating IS NOT NULL)
174                 Filter: (rating <> $1)
175                     ->  Index Only Scan using
    ratings_b on ratings  (cost=0.42..114.09 rows
    =3981 width=8)
176                     Index Cond: (rating = $3)
177 ------------------------------------------------------------
178
179 interval (q5):
180 Gather Merge  (cost=53532.97..58336.94 rows=41174
    width=24)
181 Workers Planned: 2
182     ->  Sort  (cost=52532.95..52584.42 rows=20587
    width=24)
183     Sort Key: premiered
184         ->  Parallel Seq Scan on titles  (cost
    =0.00..51057.95 rows=20587 width=24)
```

```
185          Filter: (((type)::text ~~ 'movie'::text)
     AND (premiered >= 2000) AND (premiered <= 2010))
186
187 improved:
188 Gather Merge  (cost=53532.97..58336.94 rows=41174
     width=24)
189 Workers Planned: 2
190      ->  Sort  (cost=52532.95..52584.42 rows=20587
     width=24)
191       Sort Key: premiered
192            ->  Parallel Seq Scan on titles  (cost
     =0.00..51057.95 rows=20587 width=24)
193            Filter: ((premiered >= 2000) AND (premiered
      <= 2010) AND ((type)::text = 'movie'::text))
194
195 personalised btree:
196 Sort  (cost=43904.87..44028.40 rows=49413 width=24)
197 Sort Key: premiered
198      ->  Bitmap Heap Scan on titles  (cost
     =2210.69..40052.48 rows=49413 width=24)
199      Filter: (((type)::text ~~ 'movie'::text) AND (
     premiered >= 2000) AND (premiered <= 2010))
200            ->  Bitmap Index Scan on titles_b  (cost
     =0.00..2198.34 rows=200788 width=0)
201            Index Cond: ((type)::text = 'movie'::text)
```

Appendix F

Database link

https://canvas.kth.se/courses/19966/files/3413108/download

TRITA-EECS-EX-2021:821