KTH Computer Science
and Communication

Evaluating the effect of cardinality estimates on two state-of-the-art query optimizer's selection of access method

Studying how MariaDB's and PostgreSQL's respective query optimizers select access method in correlation to the sample size used when estimating cardinality.

MARTIN BARKSTEN

Master's Thesis at NADA
Supervisor: John Folkesson
Examiner: Patric Jensfelt

TRITA xxx yyyy-nn

Abstract

This master thesis concern relational databases and their query optimizer's sensitivity to cardinality estimates and the effect the quality of the estimate has on the number of different access methods used for the same relation. Two databases are evaluated — PostgreSQL and MariaDB — on a real-world dataset to provide realistic results. The evaluation was done via a tool implemented in Clojure and tests were conducted on a query and subsets of it with varying sample sizes used when estimating cardinality.

The results indicate that MariaDB's query optimizer is less sensitive to cardinality estimates and for all tests select the same access methods, regardless of the quality of the cardinality estimate. This stands in contrast to PostgreSQL's query optimizer which will vary between using an index or doing a full table scan depending on the estimated cardinality. Finally, it is also found that the predicate value used in the query affects the access method used. Both PostgreSQL and MariaDB are found sensitive to this property, with MariaDB having the largest number of different access methods used depending on predicate value.

Referat

En utvärdering av kardinalitetsuppskattningens påverkan på två state-of-the-art query optimizers val av metod för att hämta data

Detta masterexamensarbete behandlar relationella databaseer och hur stor påverkan kvaliteten på den uppskattade kardinaliteten har på antalet olika metoder som används för att hämta data från samma relation. Två databaser testades — PostgreSQL och MariaDB — på ett verkligt dataset för att ge realistiska resultat. Utvärderingen gjordes med hjälp av ett verktyg implementerat i Clojure och testerna gjordes på en query, och delvarianter av den, med varierande stora sample sizes för kardinalitetsuppskattningen.

Resultaten indikerar att MariaDBs query optimizer påverkas lite av kardinalitetsuppskattningen, för alla testerna valde den samma metod för att hämta datan. Detta skiljer sig mot PostgreSQLs query optimizer som varierade mellan att använda sig av index eller göra en full table scan beroende på den uppskattade kardinaliteten. Slutligen, pekade även resultaten på ett båda databasernas query optimizers varierade metod för att hämta data beroende på värdet i predikatet som används för att filtrera bort rader.

Contents

List of Figures

Glossary

The terminology of databases often varies across literature and database vendors, this section therefore defines the terms used in this thesis. When it is the case that something commonly goes by other names as well, they will be included to a great an extent as possible.

Database

No distinction is made between the database and the Database Management Systems (DBMS) in this thesis as it is not relevant to separate them. If it is relevant to distinguish between the two it will be done explicitly.

Data

The *data* in a database is the values stored in the rows and columns of the database. This does not include additional information stored in the database such as indexes.

Dataset

A *dataset* is all information stored in the database, including both data and additional information such as indexes, primary and foreign keys etc.

Query optimizer

The terms query optimizer and optimizer are used interchangeably throughout the thesis. If an optimizer of some other kind is used this will be made explicit.

Host variable

A *host variable* is a variable declared in the program in which the SQL statement is embedded [6, p. 151]. Host variables are by the fact that they begin with a colon. An example of a host variable is `:HEIGHT` in Figure 0.1.

```
SELECT   NAME
FROM     PERSONS
WHERE    HEIGHT = :HEIGHT
```

Figure 0.1: A simple query using a host variable.

Predicate

In order to get the data you need, you must be able to specify what conditions the data should fulfill to be relevant, this is done by specifying predicates. To illustrate, consider Figure 0.2 taken from [24].

```
WHERE    SEX = 'M'
AND
(WEIGHT = 90
OR
HEIGHT > 190)
```

Figure 0.2: The **WHERE** clause an SQL query containing three predicates and two compound predicates.

The **WHERE** clause in Figure 0.2 contains three predicates:

1. `SEX = 'M'`
2. `WEIGHT = 90`
3. `HEIGHT > 190`

A *compound predicate* is two or more predicates that are tied together in the form of **AND**, **OR** or other similar operators. The **WHERE** clause in Figure 0.2 can be considered to have two different compound predicates:

1. `WEIGHT = 90 OR HEIGHT > 190`
2. `SEX = 'M' AND (WEIGHT = 90 OR HEIGHT > 190)`

Index slice

The term *index slice* comes from [24] and is defined as the number of index rows that need to be read for a predicate; the thinner the slice the less amount of index rows that need to be read, and consequently the number of reads to the table.

The thickness of the index slice will depend on the number of *matching columns* — the number of columns that exist both in the **WHERE** clause and the index. To illustrate why, consider the query in Figure 0.3.

```
WHERE   WEIGHT = 90
AND
HEIGHT > 190
```

Figure 0.3: The **WHERE** clause of a query with two potential matching columns.

If there exists an index on only HEIGHT, no values for WEIGHT can be discarded in the index slice. If an index is added for WEIGHT, the thickness of the index slice will decrease as only values fulfilling both the HEIGHT and WEIGHT requirements remain.

Indexable predicate

A *indexable predicate* is a predicate that can be evaluated when the index is accessed (allowing a matching index scan) [39, 20]. Revisiting the example from earlier, both of the predicates in Figure 0.3 are examples of indexable predicates.

Matching predicate

A *matching predicate* is an indexable predicate with the corresponding necessary indexes [20]. In Figure 0.3 both predicates are indexable and would be matching if there exists an index for WEIGHT and HEIGHT respectively.

Non-indexable predicate

A *difficult predicate* (also sometimes called *nonsearch arguments*, *index suppression*, *difficult predicate*) is the opposite of an indexable predicate, and can as a consequence not define the index slice [24]. What predicates are non-indexable varies from database to database, but a typical example of one can be seen in Figure 0.4.

```
COL1 NOT BETWEEN :hv1 AND :hv2
```

Figure 0.4: A example of a commonly used non-indexable predicate.

Boolean term predicate

A *boolean term predicate* (BT predicate) is one that can reject a row because it does not match the predicate [24]. Conversely a non-BT predicate is a predicate that cannot reject a row. Non-BT predicates are typically the result of using **OR**. To illustrate when a predicate is BT respectively non-BT consider, assume there is an index (A, B) on **TABLE** and consider Figure 0.5 and Figure 0.6.

For the query in Figure 0.5 if the first predicate A > :A evaluates to false for a row the row can be rejected instantly, making it a BT predicate. For the query in

```
SELECT  A, B
FROM    ATABLE
WHERE   A > :A
AND
B > :B
```

Figure 0.5: A query with a BT predicate.

```
SELECT  A, B
FROM    ATABLE
WHERE   A > :A
OR
B > :B
```

Figure 0.6: An query with no BT predicates.

Figure 0.6 on the other hand it might be the case that `B > :B` evaluates to true even if `A > :A` does not, making both predicates non-BT predicates.

Index screening

A column may be in both the **WHERE** clause and the index, yet be unable to participate in defining the index slice due to other reasons [24]. Even if this is the case the column may still be able to reduce the amount of reads to the table anyway. A column fulfilling these criteria is a *screening column* as the presence of it in the index allows not reading from the table. The process of determining which columns might fulfill this is called *index screening*.

Cardinality

The *cardinality* is the number of distinct values for a column, or combination of columns [24]. The cardinality of the data is usually used when the query optimizer estimates the cost of different access paths.

Filter factor

The *filter factor* specifies what proportion of the rows that satisfy the conditions in a predicate [24]. The filter factor can be seen as the selectivity of a predicate and the lower it is, the more the number of rows that are filtered out by a predicate. For a predicate such as `HEIGHT = :HEIGHT` there are three ways to talk about filter factor:

- The *value specific filter factor* is the filter factor for one specific value of `:HEIGHT`;

- The *average filter factor* is the average value for all value specific filter factors;
- And the *worst-case filter factor* is the highest possible filter factor for a given value of `:HEIGHT`

Access path

The query optimizers output is an *access path*, which is an abstract representation of the path to access the data.

Query plan

The *query plan* will be used to refer to the concrete representation given by an database to describe the underlying access path.

Execution plan

The *execution plan* corresponds to an access path but describes how to physically access the data.

Relation

The words *relation* and *table* are synonymous and used interchangeably throughout the thesis.

Chapter 1

Introduction

> The person who gave us this book told us that the book describes a secret technology called a database.
> We hear that the database is a system that allows everyone to share, manage, and use data.
>
> *The King of Kod, from [40, p. 6]*

If you want to save data, you need a database. And almost every computer program need to save some form of data, consequently requiring them to use a database. The trend is also going towards generating more and more data, putting higher strain on databases and requiring better performance. To improve and develop databases is therefore a topic of much relevance in today's society.

One key component of databases is the query optimizer, the part of the database that analyses the users query and finds the optimal path to access it. Or rather, theoretically it finds the optimal path. Work has been done improving query optimizers since the early '70s [7], yet optimizers often select a bad access path, causing slow queries [25].

Guy Lohman identifies the cardinality estimate of the data to be main cause for bad plans:

"The root of all evil, the Achilles Heel of query optimization, is the estimation of the size of intermediate results, known as cardinalities"

The estimates can often be wrong by several orders of magnitude [26]. These incorrect estimates then propagate through the query and grow at an exponential rate [22], making the query optimizer base its decisions on false grounds.

The topic of improving the estimations has seen some study, yet the evaluation of new methods is often done on data that is easy to estimate, being uniformly distributed. It is only recently that a study has been done to analyze the performance of the optimizer end-to-end on complex real-world data [25]. As one of the

results, the study found that PostgreSQL's optimizer performs unnaturally well on the typically tested uniform data.

This thesis will aim to provide further insight into performance of query optimizers by studying a previously unstudied metric and analysing the performance of state-of-the-art optimizers. The evaluation will be conducted on complex real-world data.

1.1 Problem statement

In this thesis two open-source state-of-the-art databases are evaluated: MariaDB [27] and PostgreSQL [35]. One real-world dataset containing a large amount of data and with a complex schema and setup will be used in the evaluation. The database will be analyzed to measure the performance of the query optimizer in order to answer the question:

How much effect does the cardinality estimate have on the query optimizers selection of access method during the join enumeration?

To evaluate this tests will be done in order to identify a correlation between the quality of the cost estimation and the number of different access methods used. To simulate varying, but realistic, quality of the cost estimation the sample size used to estimate the cardinality will be varied. For a more in-depth description of how this evaluation is conducted, see Section 4.1.

The exact setup and metrics of the dataset is described in Section 4.2. A motivation to why the databases are used is described in Section 4.1.1.

1.2 Purpose

Query optimizers make bad cardinality estimates, and as a consequence bad cost estimations of access paths [25] – but how much does this affect the actual selection of access method? Even though the errors may be large, they may not be large enough to actually cause the optimizer to generate different access paths.

There are three steps to the optimization – search space expansion, cost estimation and join enumeration (more about those in Section 3.3) – this thesis will focus on measuring what effect bad performance in the first two steps has on the third and final one. The focus of the study will be to identify if cost estimation does affect the access methods used and if the behavior differs between the databases evaluated. Finding an answer to these two will give a good indication to what effect the cost estimation has on the final step in terms of access methods used.

Studying this is of relevance for the following reasons:

1. The tool developed for evaluation can be used in the future to measure the performance of query optimizers;

2. The evaluation will provide insight into what steps in the optimization process produce bad access paths;

3. The performance of query optimizers has not seen much study using actual real-world data;
4. The actual performance of the databases right now will be evident;
5. Since both databases compared are open-source, one performing better may guide development for the other.

Of primary interest for academia are probably reasons 1, 2 and 3 whereas database vendors might be more interested in 4 and 5.

1.3 Outline

Below is a brief outline of the chapters in the report and what can be expected to be found in each:

- The Introduction chapter gives an introduction to the subject, the problem statement discussed, the purpose of the thesis and why it is novel and relevant.
- The Related work chapter contains an overview of what previous and relevant work has been done in the area of improving and evaluating the performance of query optimizers.
- The Theory chapter gives a background and the information necessary to understand the thesis. The chapter begins with a background on how a modern query optimizer works. It then continues with a description of the individual characteristics of the databases analysed: MariaDB and PostgreSQL. It also defines some important terms used throughout the thesis.
- The Method chapter describes the evaluations done and how they were implemented. It also describes more in-detail the data used for the databases, the database configurations used and the environment the tests were run in.
- The Results chapter displays the results from the performance test in the form of graphs. It also gives some brief commentary on them.
- The Discussion chapter discusses the performance of the query optimizers and the consequences of it, as well as the validity of the results. It also answers the problem statement and provides suggestions for future research.

Chapter 2

Related work

Improving the query optimizer is a topic naturally tied to that of evaluating the query optimizer's performance. In spite of this, the optimizer's performance has not seen much study, while improving them on the other hand has. This section will start with a section about some of the more recent evaluations that have been done and then continue with a section about some improvements done to the query optimizer relating cardinality estimation. A final section describes some implementations to improve the performance of relational operators and make them more resistant to bad plans.

2.1 Evaluating the query optimizer's performance

In [25] Leis et. al. perform what they claim is:

"the first end-to-end study of the join ordering problem using a real-world data set and realistic queries".

In the study they create the a database setup based on the Internet Movie Database (IMDb), create a set of realistic queries for it and call it the Join Order Benchmark (JOB). Using this benchmark they then measure how PostgreSQL, HyPer and three unnamed commercial databases perform in terms of cardinality estimates, cost modelling and general performance. They also compare the results to TPC-H, the database setup previously mostly used for evaluation and show that the PostgreSQL optimizer performs unrealistically well for TPC-H because of the uniform data distribution. The results of their study show that relational databases produce large estimation errors and that primarily the cardinality estimate is to blame.

Another article that has evaluated optimizers is [47] where Wu et al. analyzed if the optimizer's cost model can be used to estimate the actual run-time of the query. They find that the optimizer consistently makes bad cost estimates, but show that for analyzing actual run-time a more costly and precise analysis can be conducted on the selected access path.

Evaluating a query optimizer's cardinality estimation is often done through a

comparison with the actual cardinality. Calculating the exact cardinality can however be very costly for complex queries and datasets. A more efficient method that can find the exact cardinalities by studying a subset of all expressions is presented in [8].

One novel way of studying and analyzing the plans chosen by the query optimizer is a tool called Picasso, which allows query plans to be visualized as two-dimensional diagrams [19]. The tool provides a visualisation of the performance across the entire query plan space, thus providing another way of analysing queries or query optimizers.

An example of a visualisation done with Picasso can be seen in Figure 2.1. The colored regions represent a specific execution plan, the X and Y-axis represent the selectivity for the attributes `SUPPLIER.S_ACCTBAL` and `LINEITEM.L_EXTENDEDPRICE` respectively. The percentages in the legend correspond to the area covered by each plan.

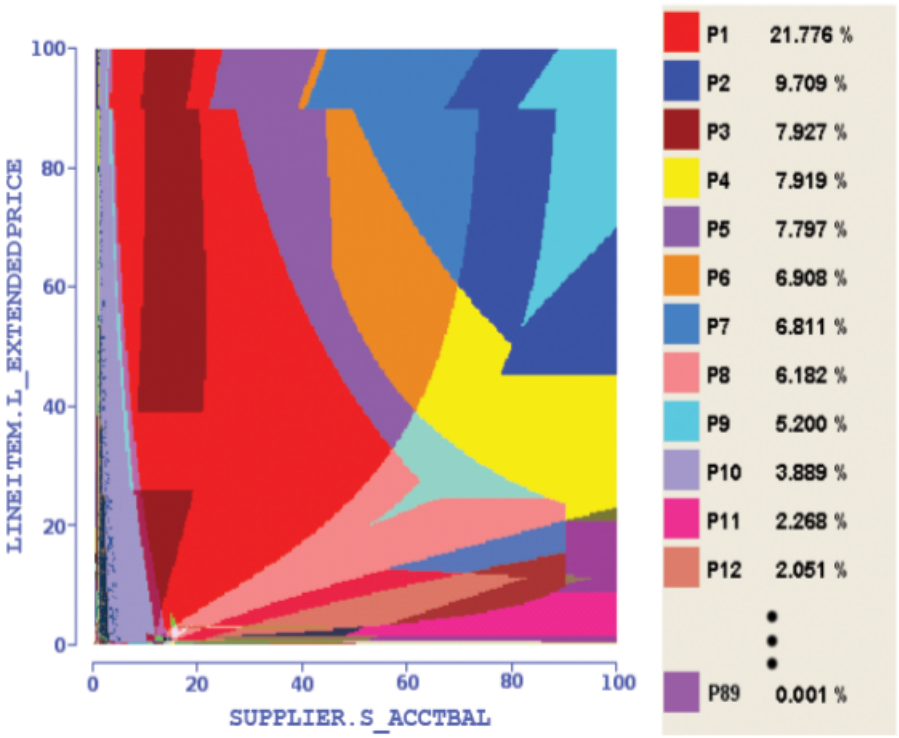

Figure 2.1: A query plan visualisation done by Picasso, image taken from [19].

Finally, on a more practical note Lahdenmäki et al. describe of how to identify queries where the selected access path is bad, and how to solve the problem [24]. The book gives a thorough introduction to many important aspects of the database and the chapter "Optimizers Are Not Perfect" focuses on incorrect cardinality estimates

and other common query optimizer errors.

2.2 Bad statistics and cardinality estimates

In [22] Ionnidis et al. develop a framework to study how cardinality estimate errors propagate in queries. Their results indicate that the error increases exponentially with the number joins.

There are two different methods of improving how optimizers handle cardinality estimation:

- Reducing the effect of incorrect estimations by making plans more *robust*, which means they perform better over large regions of the search space;
- Or by improving the estimations.

2.2.1 Improving robustness

Harish et al. present a way to make plans more robust by allowing the optimizer to select the most robust plan that is not "too slow" compared to the calculated optimal path. [18]. They develop an external tool for this purpose and find that the tool indeed does improve performance by reducing the effect of selectivity errors.

A similar study is done by Abhirama et al. but they implement the selection directly in the PostgreSQL query optimizer [1]. Their results agree with those found by Harish et al. in that the performance is improved. The results they present show that robut plans often reduced the adverse effects of selectivity errors by more than two-thirds, while only providing a minor overhead in terms of time and memory.

2.2.2 Improving cardinality estimates

The most studied problem of cardinality estimate is that of finding the right balance between calculation time, memory overhead and correctness. One common method used in current state-of-the-art databases is histograms that assume attributes are independent of each other, an assumption that often is not correct [21]. Recent studies have been done to find alternatives method that do not assume independence.

Tzoumas et al. present one method that instead of the usual one-dimensional statistical summary, saves it as a small, two-dimensional distribution [45]. Their results show an small overhead, and an order of magnitude better selectivity estimates.

In [48] Yu et al. develop a method called *Correlated Sampling* that does not sample randomly, but rather save correlated sample tuples that retain join relationships. They further develop an estimator, called reverse estimator, that use correlated sample tuples for query estimation. Their results indicate that the estimator is fast to construct and provides better estimations than existing methods.

In [46] Vengerov et al. once again study Correlated Sampling, but improve on it by allowing it to only make a single pass over the data. They compare the algorithm to two other sampling approaches (independent Bernoulli Sampling and End-Biased Sampling, which is described in [12]) and find Correlated Sampling to give the best estimates in a large range of situations.

2.3 Improving operators

In [28] Müller et al. study the two implementations of relational operators: hashing and sorting (more about these in Section 3.1.4). Their study find that the two paradigms are in terms of cache efficiency actually the same and from this observation develop a relational aggregation algorithm that outperform state-of-the-art methods by up to 3.7 times.

A problem described in [25] is that the optimizer tends to pick nested-loop joins (more about these in Section 3.1.4) even though they provide a high risk but only a very small payoff. In [16] Goetz Graefe provides a generalized join algorithm that can replace both merge joins and hash joins in databases, thus avoiding the danger of mistaken join algorithm choices during query optimization.

Chapter 3

Theory

> An SQL query walks into a bar
> and sees two tables.
> He walks up to them and asks
> "Can I join you?"
>
> – *Source: Unknown, from [41]*

In this chapter a background is given to relational databases with more focus on the areas of interest for this thesis. The first section will give a high-level introduction to relational databases and how they work in general. Following this is a section with a more in-depth description of indexes. After this comes the final section covering the query optimizer, detailing how it works, how it can be monitored and its limitations.

3.1 Relational databases

A database is a computerized record-keeping system, a way to save computerized data [11, p. 6]. The data stored in the database can then be accessed and modified by the user. Accessing and modifying the data is typically done through a layer of software called the database management system, which provides a method for accessing and modifying the data.

A database stores data in the form of rows in different tables. These rows are also sometimes referred to as tuples. In a relational database these tuples can then have relations between each other in the form of for example "Tuple A has one or more of Tuple B".

The following sections will describe some components of the database which are of the most relevant for this thesis. First comes a section describing the most common method to access data in a relational database, SQL, after this is is a section about how the query described by SQL is executed and finally a section about one of the most fundamental operations in SQL – the join operation.

3.1.1 SQL

SQL, or Structured Query Language in full, is the computer language most commonly used to query and modify the database, it is formally defined by ISO/IEC 9075 [15, p. 29][23]. The language came from research into manipulating databases in the early '70s and it is now one of the most popular database languages in existence [38].

3.1.2 Query execution

The execution of a query in the form of an SQL statement is split into four phases [37]:

1. *Parsing*, in which the input text is transformed into query blocks;
2. *Optimization*, in which an optimized way to access the data is found, called an *access path*;
3. *Code generation*, in which the access path is transformed into a way to execute it, the *execution plan*;
4. And *execution*, when the code is executed;

The *parsing*, *code generation* and *execution phases* are all fairly trivial compared to the *optimization*. The optimization process is also the phase that has the potentially most effect on the execution time for the query. The query optimization process is performed by the query optimizer, which is described in more detail in Section 3.3.

3.1.3 The join operation

One of the most fundamental operations in SQL is that of joining two tables. An example of an inner join on the tables `EMPLOYEES` and `DEPARTMENTS` can be found in Figure 3.1.

```
SELECT *
FROM   EMPLOYEES, DEPARTMENTS
WHERE  EMPLOYEES.DEPARTMENT_ID =
↪ DEPARTMENTS.DEPARTMENT_ID
AND    EMPLOYEES.NAME = 'John';
```

Figure 3.1: An SQL query that will find the all employees by the name of John and info about their department.

There are four kinds of joins typically supported in databases. To illustrate this assume we have the following `Join(A, B)`, where `A` and `B` are relations and `Join` is one of the join operations.

- An inner join will return all rows in common between `A` and `B`;
- A left outer join will return all rows in `A` and all common rows in `B`;
- A right outer join will return all rows in `B` and all common rows in `A`;
- And a full outer join will return all rows in `A` and all rows in `B`.

See Figure 3.2 for a visualization of the joins using Venn diagrams.

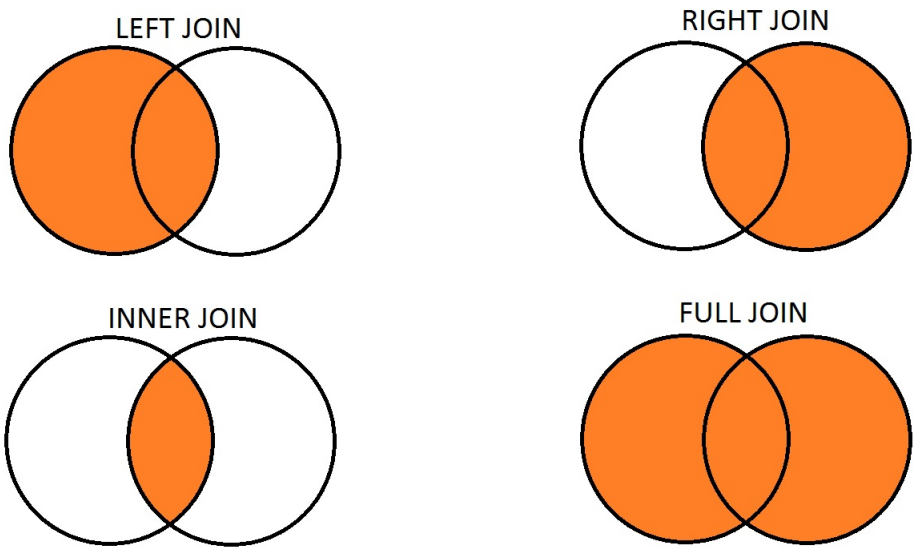

Figure 3.2: The four join operations illustrated using Venn diagrams, image taken from [5].

3.1.4 Implementation of operators

Operators can be divided into three classes:

1. *Sorting-based* methods;
2. *Hash-based* methods;
3. And *index-based* methods.

In general index-based methods are variations of class 1 or 2 that utilize indexes to speed up parts of the algorithm. Most notably when there exists an index — for example B-trees — that allows the data to be accessed sorted, joins can be done very efficiently.

Furthermore the algorithms for the operators can be grouped by the number of passes the algorithm does:

1. *One-pass algorithms* read the data from disk only once;

2. *Two-pass algorithms* read the data once, process and then save it before doing another pass;

3. And algorithms that do three or more passes and are essentially generalizations of two-pass algorithms.

There are several operators to implement in a database, but the most relevant algorithms for this thesis are those implementing the join operator. Implementation of the join operator is typically done with three fundamental algorithms: nested loop join, merge join and hash join [33]. Many databases support more join algorithms, but they are typically variations of one of these three algorithms.

For the descriptions of the algorithms assume we have a query joining the tables `A` and `B`, `Join(A, B)`. The first table of the join operation, `A`, is referred to as the *outer table* and the second table, `B`, as the *inner table*.

Nested loop join

A nested loop join is essentially two nested loops, one over the outer table and inside it one over the inner table [15, p. 718-722]. The nested loop join is a bit of a special case as it is not necessarily of any of the classes. The number of passes it does can also be considered to be "one-and-a-half" as the outer table's tuples is read only once, while the inner table's tuples are read repeatedly.

If an index exists for the inner table the nested loop join could be considered to be of class 3 and is then called a *index nested loop join*.

Merge join

A merge join (sometimes also called *sort-merge join*) is a variation of a nested loop join that requires both tables to be sorted. The two tables can then be scanned in parallel, allowing each row to only be scanned once [15, p. 723-730]. The sorting of the tables can be achieved through an explicit sort step or through an index.

A merge join can be considered to be a two-pass algorithm of class 1.

Hash join

There are several types of hash joins but the general principle remains the same: build a hash table for the outer table, then scan the inner table to find rows that match the join condition [15, p. 732-738].

Hash joins are two-pass algorithms of class 2.

3.2 Indexes

This section will cover the basics behind indexes as described by Ramakrishnan et. al. in [36, Ch. 8].

An index is a data structure that allows data stored in the database to be accessed quicker through some retrieval operations. The data stored in the index is called the *data entry* and the value that is indexed – the value in the column – is called the *search key*. There are three alternatives for what to store as the data in the data entry:

1. A data entry is a the actual data saved;
2. A data entry is a pair containing a search key and record id;
3. A data entry is a pair containing a search key and a list of record ids corresponding to the key.

Which alternative is used depends on how the index is created and what kind of an index it is.

3.2.1 Compound indexes

Compound indexes are indexes containing more than one fields. A compound index can support a broader range of queries than a normal index and since they also contain more columns, they contain more information about the data saved.

3.2.2 Clustered index

A clustered index is an index on a column which is sorted in the same way as the index; otherwise it is an unclustered index. An index using Alternative 1 is sorted by definition, whereas Alternative 2 and 3 require the data stored to be sorted.

3.2.3 Data structures

The two most common indexes used are *hash-based indexes* and *tree-based indexes*. Below is a more detailed description of both.

Hash-based indexes

A hash-based index is implemented as a hash table, mapping the hashed value of a search key to a bucket containing one or more values. To search in the index the search key is hashed and the corresponding bucket is identified, all values in the bucket are then examined to identify the matching one.

Tree-based indexes

Tree-based indexes save the data as hierarchical sorted trees where the leaf nodes contain the values. To find a value the search starts at the root and all non-leaf nodes direct the search toward the correct leaf node. In practice the trees are often implemented as B^+-trees, which is a data structure that ensure that paths from the root to a leaf node are of the same length [10]. The efficiency of a B-tree index depends on the number of levels the B-tree has [15, p. 645].

3.3 The query optimizer

In order to access the data in an as efficient way as possible, the query is optimized by a built-in tool in the databases called the query optimizer. Below is a description of the fundamental operations performed by query optimizer, taken mostly from C., Surajits article on the topic [7].

Query evaluation is handled by two components: the *query optimizer* and the *query execution engine*. The input to the query optimizer is a parsed representation of the SQL query and the output is an access path, that is transformed into a query plan that the query execution engine then performs.

In order for the query optimizer to find an access path it must be able to:

1. Expand the *search space* to find all access paths that are valid transformations of the query;
2. Perform a *cost estimation* for each access path to calculate its cost;
3. And finally *enumerate* the access paths to find which is the best.

A good query optimizer is one that does not cause too much overhead in the query execution in calculating the access path, while still finding a good access path. In order to do this each step must fulfill the criteria:

1. The search space includes plans with a low cost;
2. The cost estimation is accurate;
3. And the enumeration algorithm is efficient.

3.3.1 Expanding the search space

The first task of the query optimizer is that of taking the original search space containing just the original query, and expanding it through transformation rules. The expansion will thus generate a larger search space containing valid permutations of the join order. The output is a set of *operator trees*, which are binary trees where nodes represent operations and leaves values.

There are multiple rules that can be applied, most of which are complex and work only under some specific conditions. The most relevant rules for index-selection are described below, for a more in-depth description see [7].

Join reordering

One important rule is that both inner join and full join are:

- *Commutative*, `Join(R1, R2)` is equivalent to `Join(R2, R1)`;
- And *associative*, `Join(R1, Join(R2, R3))` is equivalent to `Join(Join(R1, R2), R3)`.

This means that the joins can be grouped and reordered as the optimizer finds best. Another consequence of this is that the operators that can be seen as a single node with many children, see Figure 3.3 for an example.

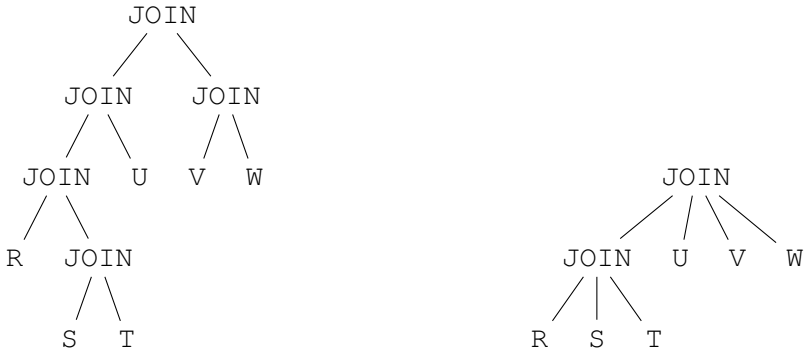

Figure 3.3: The `JOIN` operators are assumed to be associative and commutative, allowing Figure 3.3a to be transformed into Figure 3.3b, example taken from [15, p. 791].

Pushing operations up and down the tree

Another fundamental rule used by optimizers is that of pushing an operator down the expression tree in order to reduce the cost of performing it [15, p. 768-792]. For the example the selection operators tend to reduce the size of the relations, meaning pushing them down as far down the tree as possible is beneficial.

Another rule that can be applied is to pull an operator up the tree, in order to then be able to push it down again to reduce the size of more relations. See Figure 3.4, which illustrate how pulling a selection up the tree allows it to then be pushed down more branches.

The conditions for when these rules are applicable naturally varies a lot depending on the operator and it is beyond the scope of this thesis to list them all. See [15, p. 768-779] for an in-depth description of the rules and conditions.

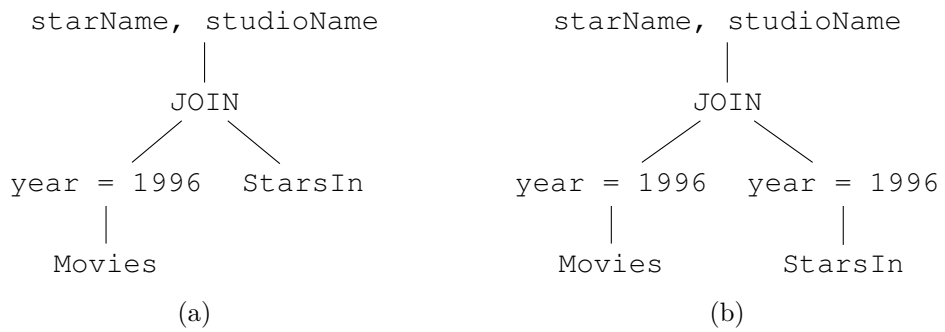

Figure 3.4: An expression tree representing a query to find which stars worked for which studios in 1996, example taken from [15, p. 774]. The selection operator in Figure 3.4a is first pulled up the tree, allowing it to then be pushed down an additional branch as illustrated in Figure 3.4b.

Eliminating operators via an index

Several operations such as `GROUP BY`, `ORDER BY`, `MAX` etc can be eliminated because the the relation is known to already fulfill these criteria [15, p. 777-779]. One of the more common criteria is the existence of an index that allows the data to be retrieved sorted.

In combination with the ability to move operators up and down the tree this can be very powerful as potentially costly operations can be eliminated.

Accessing table data

For each read from a table there will be generated one access path per usable index, as well as one for a full table scan [15, p. 827-829].

3.3.2 Cost estimation

The second task of the query optimizer is to be able to assign a cost to a given search plan. This cost estimation will be repeated several times for each operator tree in the search space that the query optimizer considers relevant, thus it is important that the estimation is efficient. The cost estimation is done in three steps:

1. Collect statistics of stored data;
2. For each node in the tree calculate the cost of applying the operation;
3. And then the calculate the statistics for the resulting output.

There are two important statistics that need to be calculated for the data: the number of tuples in a relation and the number of different values in the column of a relation for an attribute [15, p. 807-808]. Common methods of saving the data is through a histogram describing the data distribution for an attribute. If

histograms are not used the second highest and lowest values are used – this is because the highest and lowest values often are outliers

Modern databases can contain huge amounts of data and thus calculating the exact histograms is impossible, requiring them to be estimated, usually through *sampling* [15, p. 807-808]. When estimating via sampling only a subset of the data is read and used to estimate. Furthermore the statistics are usually only computed periodically as they:

- Tend not to change radically in a short time;
- Even if inaccurate, they are used consistently for all cost estimations;
- And keeping them up to date may cause their calculations to take up the most of the database's time.

Finally the statistics need to be propagated upwards in the tree. There are several ways of performing this propagation and they are described in more detail in [7].

3.3.3 Enumeration

The enumeration is the final task of the optimizer and the one that will perform the actual expansion and cost estimation, as it selects in what way to expand the search space. It would be too costly to expand the entire search space and for each plan estimate the cost, instead the search space is expanded heuristically in a way that the optimizer believes will give cheap plans.

When expanding the search space the general principle is to find paths where the size of relations is reduced as early as possible. For example pushing an operation down the tree, as shown in Figure 3.4, can reduce the size of the relation and thus reduce the cost of performing a join later on [15, p. 772-774].

If the enumeration algorithm estimates the cost for a new plan to be more expensive than a previously found one, it can discard it right away. The main goal for the enumeration algorithm is therefore to expand the search space in such a way that the best plans are generated early, so that the more expensive plans can be discarded quickly later on [29].

Most modern query optimizers use a dynamic programming algorithm first proposed in 1979 for the System R database [37]. The algorithm is built on the observation that the `JOIN`s are independent — the ith `JOIN` is independent of how the first $i - 1$ relations were joined. Based on this it is possible to construct a tree of all permutations of joins by searching from smaller to successively larger subsets.

One final optimization done during this step is to heuristically prune subtrees that the are deemed too bad to even consider [30].

3.3.4 Monitoring

It is often useful to monitor and see what decisions the query optimizer make and why; most databases implement the ability to do so via an SQL statement [24, p. 34]. In PostgreSQL and MariaDB the statement is called **EXPLAIN** [34] [13], but it is also called **SHOW** PLAN or **EXPLAIN** PLAN. For an example of how **EXPLAIN** is used see Figure 3.5, which shows the code for a query, and Figure 3.6, which shows the corresponding trace.

```
EXPLAIN
SELECT  title.title
FROM    movie_info, title
WHERE   movie_info.info IN ('Bulgaria') AND
↪ movie_info.movie_id=title.id;
```

Figure 3.5: An example of a query done on the IMDb dataset, requesting the title of all movies filmed in Bulgaria. See Figure 3.6 for the output.

```
Merge Join  (cost=2.25..921735.28 rows=19682252 width
   =17)
Merge Cond: (title.id = movie_info.movie_id)
->  Index Scan using title_pkey on title  (cost
   =0.43..155884.96 rows=3572150
width=21)
->  Index Only Scan using movie_info_idx_mid on
   movie_info
(cost=0.44..511110.22 rows=19682252 width=4)
(4 rows)
```

Figure 3.6: The access path as shown by PostgreSQL's EXPLAIN statement, corresponding to the the query in Figure 3.5.

3.3.5 Limitations

Even though much work has been done on improving query optimizers they may not always choose the correct access path. This section will describe some of the primary reasons why an incorrect access path path is chosen, as described in [24, Ch. 14].

The optimizer can't see the best path

One reason the query optimizer cannot find the best path is because it is unable to see all alternatives because the query is too complicated.

- If a predicate is non-indexable it cannot by definition participate in defining the index slice. Furthermore it might also be the case when the predicate is even more difficult that the the optimizer is unable to perform an index screening, forcing it to read a table row.
- If a compound predicate contains **OR** it may become non-BT, which in turn means the predicate cannot be used to define the index slice. This means the query optimizer cannot make full use of potential indexes that exist.
- Sometimes the optimizer will add an **ORDER BY** to data that is already sorted thanks to an index.

The optimizer's cost estimate is wrong

Even if the optimizer is able to see all alternatives it might be the case that the filter factor is incorrectly estimated, resulting in an incorrect access path.

- If the filter factor is not estimated for a host variable, it must use a default value which often results in a poor estimate. However, if the filter facotr is estimated evewry time the query is executed it adds a large overhead.
- If the optimizer is unaware of the true distribution of the data it is forced to guess cardinality, a guess that tends to be wrong if the distribution of the data is skewed;
- In a compound predicate such as `HEIGHT = :HEIGHT AND WEIGHT = :WEIGHT` the optimizer can only produce a good estimate of the filter factor if it knows the cardinality of the combination of the `HEIGHT` and `WEIGHT` columns. If it does not, it must estimate this.

Chapter 4

Method

In this chapter, the method used to investigate the problem statement is presented. First, the choice of method is described, motivating the dataset and technologies used. Following this the problems used for benchmarking are presented, including a more in-depth description of the dataset. After these motivations, the actual implementation details are presented. Finally the evaluations done are described, providing the parameters used for the tests conducted.

4.1 Choice of method

This section will motivate the methods' three primary questions:

1. What databases are evaluated?
2. What dataset is used to evaluate the databases?
3. How are the databases evaluated?

The following three sections will answer these questions, motivating choice of databases, dataset and implementation.

4.1.1 Choice of databases

The two databases chosen to evaluated are PostgreSQL and MariaDB. The choice was made based on the the fact that they are:

1. Modern databases with widespread use and active development;
2. Open-source projects allowing anyone to read and modify the source code;
3. And they implement state-of-the-art algorithms and methods.

In addition to this they both cover two common use-cases: academia and enterprises. All research papers mentioned in Chapter 2 that have implemented new

algorithms or modified old ones have done so in PostgreSQL. On the other hand MariaDB is compatible with MySQL, making it a common alternative for companies to use.

An evaluation of both of these database will give a good indication of the performance of a modern state-of-the-art query optimizer. Furthermore, as mentioned in Section 1.2 since both of them are open-source, if one performs better than the other the code can be studied to identify areas of improvement.

4.1.2 Choice of dataset

The primary focus when selecting the dataset was to use a dataset which could capture the complexity of a real-world database and provide a realistic challenge for the query optimizer. The primary requirements are that the dataset feature:

1. Many relations with multiple indexes each;

2. More than just trivial indexes on a single row;

3. Skewed and non-uniform data for which the cardinality is not trivially estimated;

4. And a sufficiently large amount of data such that the database must estimate cardinality.

The datasets most commonly used for evaluation of database implementations are TPC-H [43], TPC-DS [42] and more recently JOB [25]. However, none of these datasets meet requirement 2 making the problem of selecting access methods for these databases trivial.

Instead, the dataset chosen was one taken from the real world: the dataset for TriOptima's product triReduce [32] which fulfill all the requirement. The metrics of the dataset are presented in more detail in Section 4.2.

Another important aspect of the dataset is the queries used for evaluation. Selecting these was done based on the following critera:

- The relations involved in the query must be sufficiently large as to require the cardinality to be estimated via sampling;

- The data most be accessible via one or more index so that the actual index selection is not trivial for the query optimizer.

The two criteria are not fulfilled for more than a few relations in a database, reducing the amount of queries relevant for evaluation. However, the queries that do fulfill the above requirements are also those that are most interesting to study as they are the ones that will have the longest execution time.

4.1.3 Choice of implementation

The focus when implementing the tool used to evaluate the databases was to find a tool that would allow a high-level description of the data transformations necessary. Additionally the language must be sufficiently stable and be able to handle potentially large amounts of data.

The language chosen that fulfill these requirements is Clojure. Clojure compiles to bytecode that runs on the JVM, which is stable and well-used. Additionally the language is well-suited to describing data transformations as it provides many high-level functions for doing so.

More information regarding Clojure, the tool developed and how the data is transformed in practice is presented in Section 4.3

4.2 Benchmark problems

This section describes the problems used for benchmarking, starting with specification of the hardware that the tests were ran on. Following this is a description of the metrics of the dataset used. Finally the metrics for the queries used for the evaluation are presented.

4.2.1 Hardware specs

All evaluations were done on a dedicated computer running only the databases. The most important part of the hardware is to ensure that there is sufficient data for both the databases and the results of the tests. For all evaluations three hard drives were used, one for each database and one for the tool itself.

The exact specifications are:

- 2 *Intel® Xeon® Processor E52643 (10M Cache, 3.30 GHz, 8.00 GT/s Intel® QPI)*, featuring 4 cores each;
- 1 *Seagate Savvio 15K.3 ST9146853SS 146GB 15000 RPM 64MB Cache SAS 6Gb/s 2.5"*, used to store the project itself on;
- 1 *Seagate Constellation ES.3 ST4000NM0023 4TB 7200 RPM 128MB Cache SAS 6Gb/s 3.5"*, used to store the PostgreSQL database on;
- And 1 *Seagate Constellation ES ST2000NM0001 2TB 7200 RPM 64MB Cache SAS 6Gb/s 3.5"*, used to store the MariaDB database on.

As a final note it is worth pointing out that the effect of the hardware should have none, or very little, effect on the query optimizer's plan selection.

4.2.2 The dataset

As detailed in Section 4.1.2 the dataset should be sufficiently complex in terms of indexes, table size and table values. Table 4.1 presents the number of indexes, the

	#index	#rows	size (MB)
database total	1130	305	1165290
mm	6	64882651	9448
book	6	51709	10
resamb	3	40598	5
cmm	2	17335822	1219
cmt	9	52808814	12811
t	35	115851469	92633
est	32	33726190	19434
ct	23	115751571	72320
mt	9	21721256	4284

Table 4.1: The metrics for the dataset used for evaluation of the databases. Both the metrics for the entire database and those of individual relations are shown. Note that the relation names have been anonymized and are only referred to by an identifier.

number of rows and the size in MB of the entire database and all relations involved in the benchmarking, the names of the relations have been anonymized and are referred to by an identifier such as *mm* or *book*.

The original dataset was stored in a MySQL database and was ported to PostgreSQL and MariaDB. MariaDB is made as an add-on to MySQL and thus required no tooling, the data was just copied into a fresh install of MariaDB using a *mysqldump*. For PostgreSQL *py-mysql2pgsql* [31] was used to create a dump of the MySQL database with all MySQL specific data types converted to their most similar PostgreSQL equivalents. The data was then read into a fresh install of PostgreSQL.

In the copying process all indexes and relations were maintained, thus maintaining essentially the same metrics for both of the copies as for the original.

4.2.3 The queries

To evaluate the databases only one query was used. The reason is that, as described in Section 4.1.2, there are few relations that can be involved in the query as they must fulfill the important critera in terms of indexes and size. As such one query covering all of the most complex relations was constructed. This query can be considered to be the most complex query for the dataset and subsets of it are used to simulate simpler scenarios.

The original query used for evaluation can be seen in Figure 4.1, the relation names are once again anonymized. The query is constructed so that the predicate used for filtering is an indexable one. For the metrics of the relations involved see Figure 4.1. The variations of the query simply remove one or more relations

involved.

```
SELECT *
FROM ct JOIN t JOIN mt JOIN mm JOIN book JOIN cmt JOIN cmm
 ↪  JOIN est JOIN resamb
WHERE ct.key = :KEY
```

Figure 4.1: The query used for evaluation, simplified and anonymized. The relations are joined on rows in common.

4.3 Implementation

This section will cover the implementation details of the tool used for evaluation. The section starts with a general overview of the process of evaluation, breaking it down into steps. Following this is a description of the tool developed, showing how it is used and what technologies are used. After this section comes three sections describing each of the three steps of evaluation. Finally implementation details are provided for PostgreSQL and MariaDB.

Evaluating a database for a given query can be broken down into three primary steps:

1. Repeatedly forcing the database to re-estimate the cardinalities and then generate query plans;
2. Parsing the query plans to find what access methods are used for all relations;
3. And finally analyzing the parsed plans to find the number of unique access methods used for each relation.

These three steps must be executed in order, but they can be executed independently of each other — it is possible to only generate plans and save them for later parsing and analysis.

4.3.1 Overview of the tool

The tool was implemented so as to be executed from the command line, providing all the necessary parameters via flags. As mentioned in Section 4.1.3 the tool was developed using Clojure and is therefore typically ran via Leiningen, as shown in Figure 4.2.

As the steps can be executed one at a time, the results for each are saved and the tool allows the execution of only a specific set of steps at a time. An example of this can be seen in Figure 4.3 where a previously generated plan is parsed and analyzed.

The tool is open-source and can be found at [2]. The repository has further documentation regarding project structure etc.

```
    lein run steps='generate parse analyze' query=queryid
↪   repetitions=100 samplesizes='10 100'
↪   --database=postgresql
```

Figure 4.2: An example of using the tool to generate, parse and analyze a query with some given parameters, such as the sample sizes to use.

```
    lein run plans/xxx-000000000 steps='parse analyze'
```

Figure 4.3: An example of how the tool can be used to parse and analyze a previously generated file containing query plans.

4.3.2 Generating plans

The task of generating query plans can be broken down in the following steps:

1. Set the sample size used to estimate cardinalities;
2. Generate new statistics, and thus new cardinality estimates, for all relations involved in the query used for evaluation;
3. Find the query plan for each possible value of the query.

These steps are then repeated a number of times to ensure that all query plans are found.

The most relevant parts of the code used to generate plans can be found in Figure 4.4. In the figure two functions can be seen: `generate-plans` and `sample-and-query`.

The generation step is handled by `generate-plans`, which is provided the options for the evaluation. This function will then for each sample size to use, repeated the number of times specified, call `sample-and-query`.

The `sample-and-query` function will in turn perform all of the steps outlined above; set the statistics target and generate new statistics via `resample-with!` and then find the query plan for all possible predicate values via repeated calls to `explain-query`.

Each plan found is directly saved to file for later use in parsing.

4.3.3 Parsing the plans

Parsing the generated plans is done by simply stripping all information but the access methods from the query plans, after this is done the access methods are grouped by what relation they access. The code for the parsing can be seen in Figure 4.5.

Parsing a plan is done with the function `parse-plan`, which will find all access methods with `find-relation-accesses` and group these by their relation with

```
(defn- sample-and-query [save-plan options]
  (resample-with! options)
  (doseq [param param-range]
    (save-plan (explain-query options param))))

(defn generate-plans [opts save-plan]
  (j/with-db-connection [db-con (opts->db-info opts)]
    (doseq [sample-size (:samplesizes opts)]
      (dotimes [i (:repetitions opts)]
       (sample-and-query save-plan
                         (assoc
                          opts
                          :sample-size sample-size
                          :connection db-con))))))
```

Figure 4.4: The relevant parts of the Clojure code used to generate the query plans. Some function definitions have been removed to improve readability.

`group-by-relation`. Finding the access methods in the query plan is done by traversing the plan as a tree and calling `save-if-relation-access` on each value — storing it, if it describes an access method.

Each generated plan is parsed and the new plan saved to another file. The main purpose is to transform the data into something more easily analyzed. Additionally the size of the query plans are reduced in size, reducing the time taken to analyze all plans.

4.3.4 Analyzing the plans

Analyzing the plans is done by merging all access methods found when parsing the plans, keeping the distinct methods for each relation. The code for analysis can be seen in Figure 4.6.

Analyzing all generated plans for a sample size is done by the function `analyze-plans`, which is provided an identifier for the database evaluated, a function to read the next plan and the total number of plans to read. The analysis is done by reading the next plan and merging it with the previous one. If the same relation is found in both the function `conj-distinct` will add only the new access methods found that are distinct from those previously found. Only the index used is considered in terms of two plans being distinct from each other.

The analysis is done for each sample size, and the resulting map of relation to access methods is saved for later study.

```
(defn- save-if-relation-access [db-id o]
  (if (and (map? o) (contains? o db-id))
    (swap! relation-accesses conj o))
  o)

(defn- find-relation-accesses [db-id plan]
  (reset! relation-accesses [])
  (postwalk #(save-if-relation-access db-id %) plan)
  @relation-accesses)

(defn- group-by-relation [db-id accesses]
  (group-by
   #(get % db-id)
   accesses))

(defn parse-plan [db plan]
  (let [db-id (access-key db)]
    (group-by-relation db-id (find-relation-accesses db-id
↪  plan))))
```

Figure 4.5: The Clojure code used to parse the query plan output from the generation step.

4.3.5 PostgreSQL

In PostgreSQL the SQL commands used are quite straightforward, one is used to delete all previously gathered statistics, one to set the sample size and finally an **ANALYZE** is called for each relation involved in the query being evaluated. The specific commands used can be seen in Figure 4.7, the value of the sample size is provided as a host variable, as it will depend on the options used when evaluating a database and query.

4.3.6 MariaDB

In MariaDB the commands used to estimate cardinality are storage engine specific, in this case InnoDB is the storage engine used. It is worth noting that InnoDB only provides MariaDB with statistics — it is MariaDB that does the actual query optimization.

It is necessary in InnoDB to ensure that the statistics used are those generated, to do this no persistent statistics are saved. Furthermore, no data is deleted between estimates as it is neither possible nor necessary — the old statistics are overwritten by the new. Finally the relations are all analyzed in one single **ANALYZE** call.

The specific commands used can be seen in Figure 4.8.

```
(defn conj-distinct [f x y]
  (reduce
   (fn [coll v]
     (if (some #(= (f %) (f v)) coll)
       coll
       (conj coll v)))
   x y))

(defn analyze-plans [db next-plan plans-to-read]
  (loop [m {} plans-left plans-to-read]
    (if (zero? plans-left)
      m
      (recur
       (merge-with
        #(conj-distinct (fn [access] (get access (idx-key
↪  db)))
                        %1 %2)
        m (next-plan))
       (dec plans-left)))))
```

Figure 4.6: The Clojure code used to analyze the parsed output. The code will merge the maps generated, only keeping the unique access methods for each relation.

```
DELETE FROM pg_statistics;
SET default_statistics_target TO :SAMPLE_SIZE;
ANALYZE table1;
ANALYZE table2;
```

Figure 4.7: The SQL commands used to first delete all statistics in PostgreSQL, set the statistics target and finally analyze all relations involved in the query.

4.4 Evaluation

In order to evaluate the database the dataset described in 4.1.2 and query #1, shown in Figure 4.1. The tool supports several options to be set and the values used for the evaluation can be seen in Table 4.2, which shows the initial tests conducted. The results from these tests warranted further evaluation and the options for these tests can be seen in Table 4.3. The same values were used when testing both PostgreSQL and MariaDB.

In order to find what effect the cardinality estimate has on the access methods used the databases the tests use varying sample sizes used when generating the estimates. When using a low sample size the generated cardinality estimate tends

```
SET GLOBAL innodb_stats_persistent='OFF';
SET GLOBAL innodb_stats_auto_recalc='OFF';
SET GLOBAL innodb_stats_transient_sample_pages =
↪ :SAMPLE_SIZE;
ANALYZE TABLE table1, table2;
```

Figure 4.8: The SQL commands used to first ensure that MariaDB will not use some other stats than those we gather, then set the statistics target and finally analyze all relations involved in the query.

query	repetitions	sample sizes
#1	50	1
#2	50	1
#1	50	100
#2	50	100

Table 4.2: The number of repetitions and sample size used for the evaluation. The tests are conducted with a low and high value for the sample size in order to capture the difference when analyzing cardinality with few samples versus many samples.

to be worse than when a high sample size is used. The tests were done on the original query and a subset of it, seen in Figure 4.9. The simpler query was used to evaluate the effect cardinality estimate errors propagating when a large number of joins is involved. The options used for the tests can be seen in Table 4.2. The repetition size is set to 50 due to time constraints when evaluating — for example, evaluating PostgreSQL with a sample size of 100 and 50 repetitions takes roughly 100 hours.

Additional tests were conducted on subsets of the query to further evaluate the behavior identified in the first evaluation. As can be seen in the table the number of repetitions and sample size is set to 1, this to test the effect of predicate value rather than sample size. For a further discussion about this see Section 6.2, where the results found for the initial tests are discussed in detail.

```
SELECT *
FROM ct JOIN t JOIN mt JOIN mm JOIN book JOIN cmt
WHERE ct.key = :KEY
```

Figure 4.9: Query #2, a subset of the original query used to provide a more simple scenario for the query optimizer. While still containing the relation *cmt*, which is relevant for PostgreSQL.

```
SELECT *
FROM ct
WHERE ct.key = :KEY
```

Figure 4.10: Query #3, a subset of the original query and one that should be trivial for the query optimizer as it does no joins and only selects a single relation.

queries	repetitions	sample sizes
#1	1	1
#2	1	1
#3	1	1

Table 4.3: The number of repetitions and sample size used for the evaluation. Two tests with a large number of repetitions were done for the original, and the other queries evaluated with a lesser number of repetitions.

Chapter 5

Results

This chapter contains the results of using the tool to evaluate the two databases. The chapter starts with a section showing the results when evaluating the effect of the cardinality estimate for the access methods chosen by the databases. Following this is a section containing the results of the second evaluation, which focus on evaluating what factors other than cardinality estimate cause the databases to select different access methods.

5.1 The effect of cardinality estimate

This section contains the results of the evaluation which focus on the effect of cardinality estimates. The section is split into two sections, one describing the tests with the low sample size and the other describing the tests with a high sample size. For the exact parameters used see Table 4.2, which describes queries tested, repetitions done and sample sizes used.

The results are presented in the form of graphs showing the number of unique access methods used for each relation and database. To complement these graphs are also excerpts of the most relevant parts of the tool's output for the test. The excerpts focus on the parts of the output showing which access methods are used for the relations with many access methods.

5.1.1 Low sample size

The low sample size was set to 1 and the process of generating new cardinality estimates and retrieving query plans for them repeated 50 times.

The first test was done on query #1 and the results can be seen in Figure 5.1. It is clear from the figure that both databases use different access methods for the same relation; for MariaDB the relation *ct* is accessed using three different methods and PostgreSQL for the relations *mt*, *cmt*, *cmm* and *est* are all accessed with two different methods.

The access methods used for the relations with many access methods can be seen in Figure 5.2 for MariaDB and Figure 5.3 for PostgreSQL. From these it can be seen that MariaDB varies between three different indexes when accessing *ct*. For PostgreSQL it is instead the case that the it varies between using a full table scan (called a *seq scan*) or an index.

The results of the second test — using query #2 instead — can be seen in Figure 5.4. The query is simpler in terms of the number of joins and tables involved, yet it is still the case that PostgreSQL will select multiple access methods for the relation *mt*.

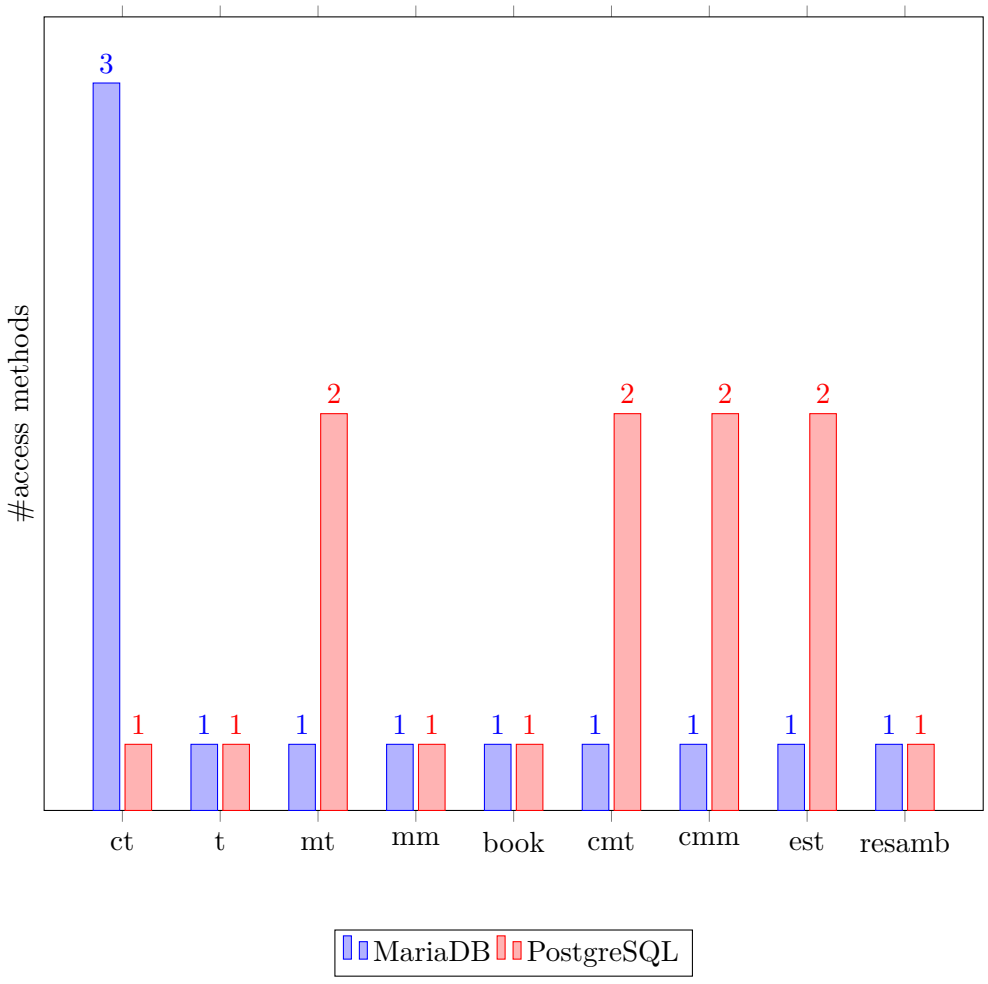

Figure 5.1: The test conducted in the first evaluation using a sample size of 1 and 50 repetitions. The graph shows that MariaDB varies between three different access methods for the relation *ct* and PostgreSQL between two for the relations *mt*, *cmt*, *cmm* and *est*.

```
{
    "ct": [
        {
            "table": "ct",
            "key": "match_view_ref",
            "possible_keys": "match_view_ref,match_view_ref_2,CycleTrade
 ↪  _master_trade_ref,CycleTrade_trade_ref,CycleTrade_match_view_ref"
        },
        {
            "table": "ct",
            "key": "match_view_ref_2",
            "possible_keys": "match_view_ref,match_view_ref_2,CycleTrade
 ↪  _master_trade_ref,CycleTrade_trade_ref,CycleTrade_match_view_ref"
        },
        {
            "table": "ct",
            "key": "CycleTrade_match_view_ref",
            "possible_keys": "match_view_ref,match_view_ref_2,CycleTrade
 ↪  _master_trade_ref,CycleTrade_trade_ref,CycleTrade_match_view_ref"
        }
    ]
}
```


Figure 5.2: Excerpt of the tool's output when evaluating MariaDB with 50 repetitions and a sample size of 1. The relation shown, *ct*, is the one which MariaDB use different access methods for.

5.1.2 High sample size

The second set of tests done was done using a sample size of 100, the number of repetitions was set to the same value as before — 50.

The first test was done on query #1 and the results of it can be seen in Figure 5.5. Compared to the access methods used for the low sample size, seen in Figure 5.1, PostgreSQL has become more consistent in its choice of access methods. However, MariaDB remains consistent in using three different access methods for the relation ct.

An excerpt of the tool's output for the test on PostgreSQL can be seen in Figure 5.7. From the output it is clear that effect of improved cardinality estimated made PostgreSQL never choose to do a full table scan rather than use an index for all relations shown.

The output for MariaDB can be seen in Figure 5.6 and comparing it to the output from the test with a low sample size, seen in Figure 5.2, it is clear that the same three access methods are used — regardless of sample size used when estimating cardinality.

Query #2 was also tested with a high sample size and the results, seen in Figure 5.8, show the same results as those for PostgreSQL and query #1 — the relation *mt* is now accessed with only one access method.

5.2 Evaluating subsets of the query

This section contains the results for the second evaluation conducted. The focus of this evaluation was to identify what other factors might affect the choice of access method if it was not the cardinality estimate. Thus, the tests are done with only 1 repetition to see if the access methods are different even if the cardinality estimate is the same for all query plans generated.

Three queries are tested, the original query, a subset of the original query with less tables involved and a trivial query accessing only the relation *ct*.

5.2.1 Query #1

The first test was done on query #1 in order to see which access methods varied, event though only one query plan was retrieved and the results can be seen in Figure 5.9. The results show that even though the cardinality estimate is fixed, different query plans are still generated.

The tool's output shows that MariaDB accesses the relation *ct* with three different access methods (as previously observed in Figure 5.2 and Figure 5.6).

The output for PostgreSQL can be seen in Figure 5.11 and shows that relations *cmt*, *cmm* and *est* are accessed with different access methods. Unlike in the previous evaluation the relation *mt* is not accessed differently.

5.2.2 Query #2

The second query tested is simpler than the original as it involves less tables and thus `JOIN` operations. The results of the test can be seen in Figure 5.12, which shows that PostgreSQL still use multiple access methods for the relation *cmt*.

The output of the tool for PostgreSQL can be seen in Figure 5.13. Comparing the output to that of the test on the original query, shown in Figure 5.11, shows that *cmt* is accessed with the same two access methods for both.

5.2.3 Query #3

The final query used used to test the databases with was the most simple variant of query #1 — selecting and filtering only the relation *ct*. The results of the test can be seen in Figure 5.14, which shows that MariaDB still use three access methods. Furthermore, the output of the tool, shown in Figure 5.15, shows that it is the same methods as those used for the full query, shown in Figure 5.10.

```
{
    "cmm": [
        {
            "Node Type": "Index Scan",
            "Index Name":
 ↪  "\"NoneCycleMasterMatch_cycle_master_match_key_pkey\"",
            "Alias": "cmm"
        },
        {
            "Node Type": "Seq Scan",
            "Alias": "cmm"
        }
    ],
    "cmt": [
        {
            "Node Type": "Index Scan",
            "Index Name":
 ↪  "\"NoneCycleMasterTrade_cycle_master_trade_key_pkey\"",
            "Alias": "cmt"
        },
        {
            "Node Type": "Seq Scan",
            "Alias": "cmt"
        }
    ],
    "est": [
        {
            "Node Type": "Index Scan",
            "Index Name": "\"NoneExternalServiceTrade_cycle_master_trade
 ↪  _ref_external_servic\"",
            "Alias": "est"
        },
        {
            "Node Type": "Seq Scan",
            "Alias": "est"
        }
    ],
    "mt": [
        {
            "Node Type": "Seq Scan",
            "Alias": "mt"
        },
        {
            "Node Type": "Index Scan",
            "Index Name": "\"NoneMasterTrade_master_trade_key_pkey\"",
            "Alias": "mt"
        }
    ]
```


Figure 5.3: Excerpt of the tool's output when evaluating PostgreSQL with 50 repetitions and a sample size of 1. The relation shown — cmm, cmt, est and mt — are those which PostgreSQL use different access methods for. The output shows that compared to MariaDB, PostgreSQL only varies between using an index or doing a full table scan — never between several different indexes.

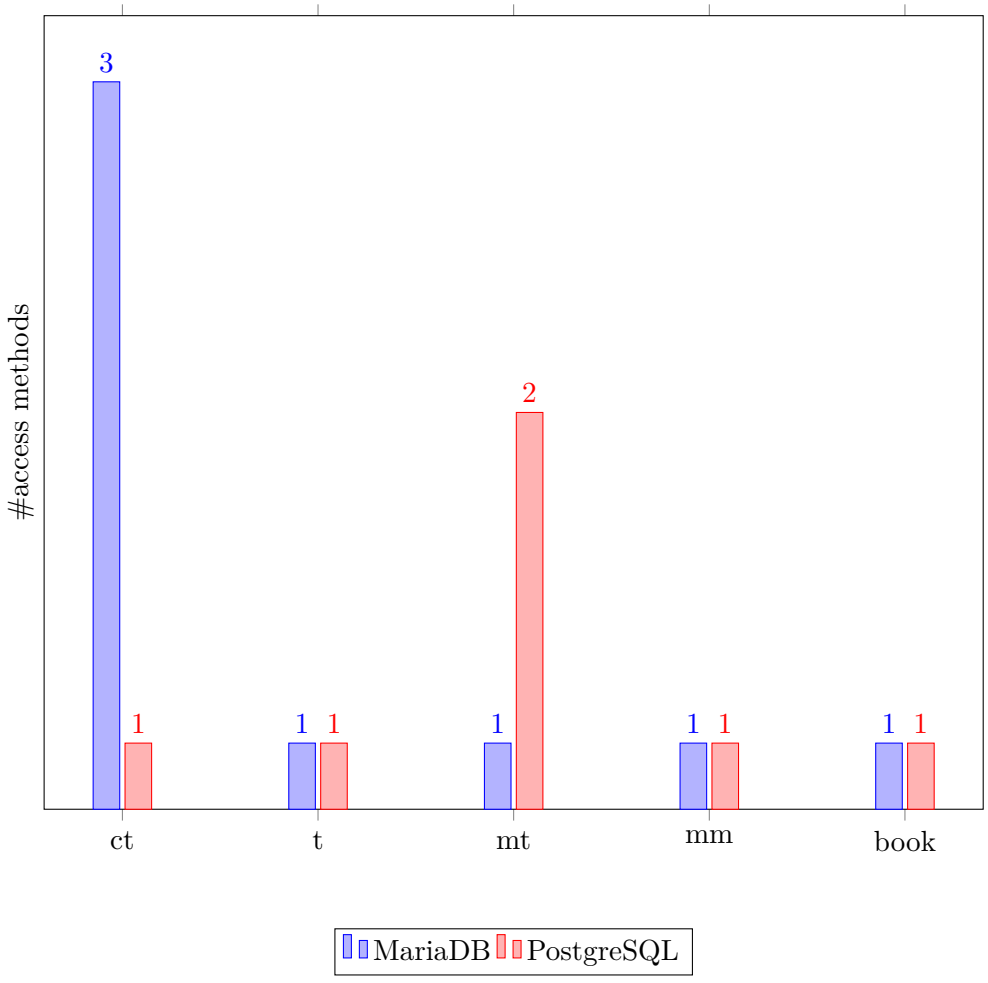

Figure 5.4: The test conducted using query #2, 50 repetitions and a sample size of 1 to simulate a cardinality estimate of low quality. The graph shows that PostgreSQL will vary between using two different access methods for the relation *mt*, even though the query is simpler in terms of the number of joins used compared to query #1.

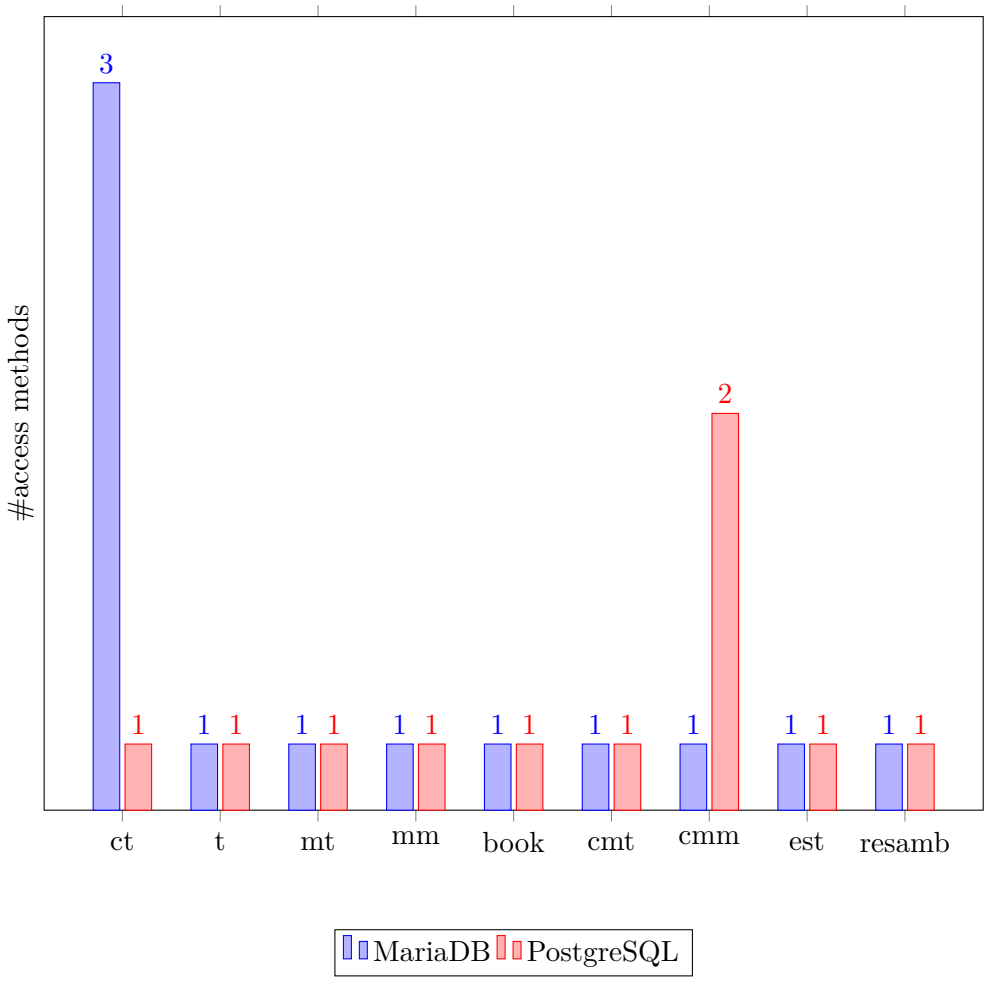

Figure 5.5: The test conducted using query #1, 50 repetitions and a sample size of 100 to simulate a cardinality estimate of high quality. The graph once again shows that MariaDB use three different access methods for the relation *ct*. PostgreSQL on the other hand now only use different access methods for the relation *cmm* and none else.

```
{
    "ct": [
        {
            "table": "ct",
            "key": "match_view_ref",
            "possible_keys": "match_view_ref,match_view_ref_2,CycleTrade
 ↪  _master_trade_ref,CycleTrade_trade_ref,CycleTrade_match_view_ref"
        },
        {
            "table": "ct",
            "key": "match_view_ref_2",
            "possible_keys": "match_view_ref,match_view_ref_2,CycleTrade
 ↪  _master_trade_ref,CycleTrade_trade_ref,CycleTrade_match_view_ref"
        },
        {
            "table": "ct",
            "key": "CycleTrade_match_view_ref",
            "possible_keys": "match_view_ref,match_view_ref_2,CycleTrade
 ↪  _master_trade_ref,CycleTrade_trade_ref,CycleTrade_match_view_ref"
        }
    ]
}
```


Figure 5.6: Excerpt of the tool's output when evaluating MariaDB with query #1, 50 repetitions and a sample size of 100. Comparing this output to that given when using a sample size of 1, shown in Figure 5.2, shows that the same access methods are used — regardless of sample size used when estimating cardinality.

```
{
    "cmm": [
        {
            "Node Type": "Index Scan",
            "Index Name":
 ↪  "\"NoneCycleMasterMatch_cycle_master_match_key_pkey\"",
            "Alias": "cmm"
        },
        {
            "Node Type": "Seq Scan",
            "Alias": "cmm"
        }
    ],
    "cmt": [
        {
            "Node Type": "Index Scan",
            "Index Name":
 ↪  "\"NoneCycleMasterTrade_cycle_master_trade_key_pkey\"",
            "Alias": "cmt"
        }
    ],
    "est": [
        {
            "Node Type": "Index Scan",
            "Index Name": "\"NoneExternalServiceTrade_cycle_master_trade
 ↪  _ref_external_servic\"",
            "Alias": "est"
        }
    ],
    "mt": [
        {
            "Node Type": "Index Scan",
            "Index Name": "\"NoneMasterTrade_master_trade_key_pkey\"",
            "Alias": "mt"
        }
    ]
}
```


Figure 5.7: Excerpt of the tool's output when evaluating PostgreSQL with query #1, 50 repetitions and a sample size of 100. The relations shown are the same as those for the test with a sample size of 1. The output shows that all but the relation *cmm* are now accessed by using an index.

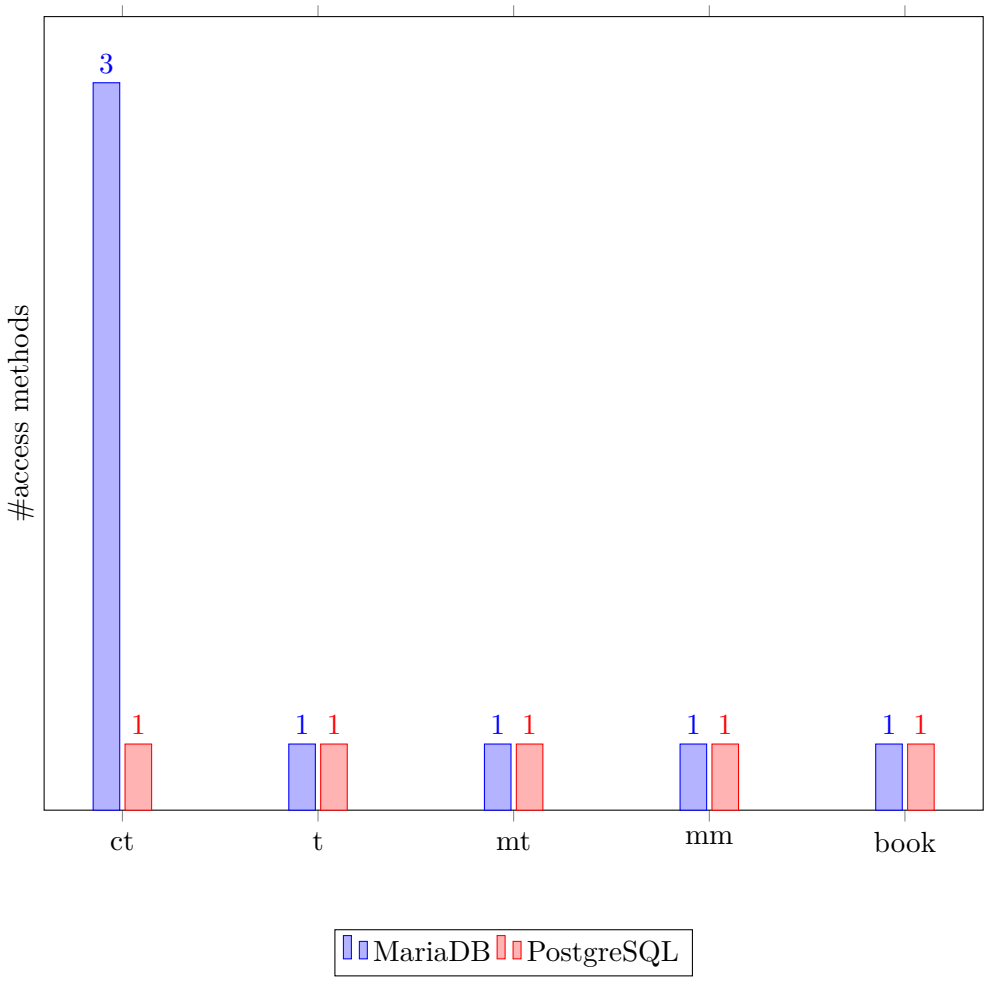

Figure 5.8: The test conducted using query #2, 50 repetitions and a sample size of 100 to simulate a cardinality estimate of high quality. The graph should be compared that of the test with a sample size of 1, which can be seen in Figure 5.4. The notable difference is that *mt* is selected using only one access method, rather than two.

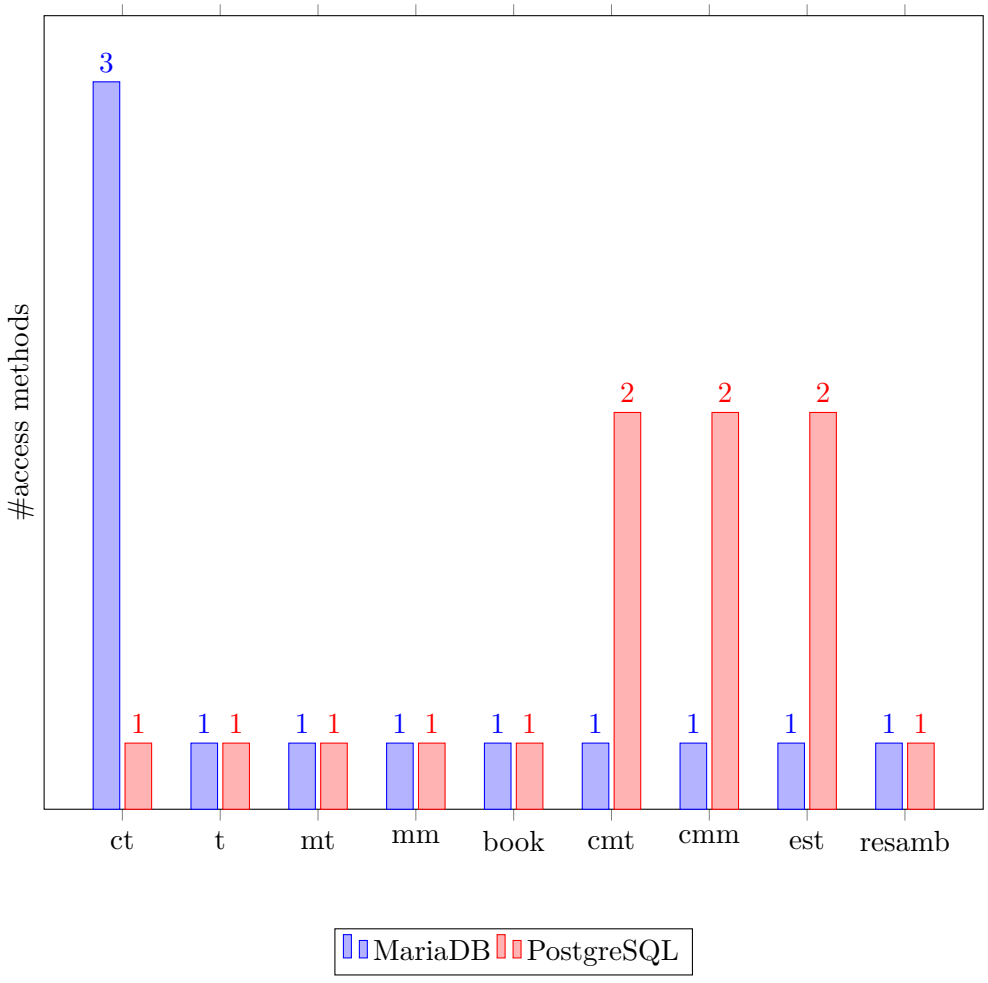

Figure 5.9: The access methods when evaluating query #1 with only 1 repetition. The graph shows that even though the estimated cardinality is the same for all retrieved query plans, MariaDB still use different access methods for *ct* and PostgreSQL for *cmt*, *cmm* and *est*.

```
{
    "ct": [
        {
            "table": "ct",
            "key": "match_view_ref",
            "possible_keys": "match_view_ref,match_view_ref_2,CycleTrade
 ↪  _master_trade_ref,CycleTrade_trade_ref,CycleTrade_match_view_ref"
        },
        {
            "table": "ct",
            "key": "match_view_ref_2",
            "possible_keys": "match_view_ref,match_view_ref_2,CycleTrade
 ↪  _master_trade_ref,CycleTrade_trade_ref,CycleTrade_match_view_ref"
        },
        {
            "table": "ct",
            "key": "CycleTrade_match_view_ref",
            "possible_keys": "match_view_ref,match_view_ref_2,CycleTrade
 ↪  _master_trade_ref,CycleTrade_trade_ref,CycleTrade_match_view_ref"
        }
    ]
}
```


Figure 5.10: Excerpt of the tool's output when testing MariaDB with query #1 and 1 repetition. Only the relation which MariaDB select multiple different access methods for, *ct*, is shown.


```
    {
    "cmt": [
        {
            "Node Type": "Index Scan",
            "Index Name":
 ↪  "\"NoneCycleMasterTrade_cycle_master_trade_key_pkey\"",
            "Alias": "cmt"
        },
        {
            "Node Type": "Seq Scan",
            "Alias": "cmt"
        }
    ]
}
```


Figure 5.11: Excerpt of the tool's output when testing PostgreSQL with query #1 and 1 repetition. The relation *cmt* is shown as PostgreSQL selects multiple access methods for it and the relation is part of query #2, which is used for the second test.

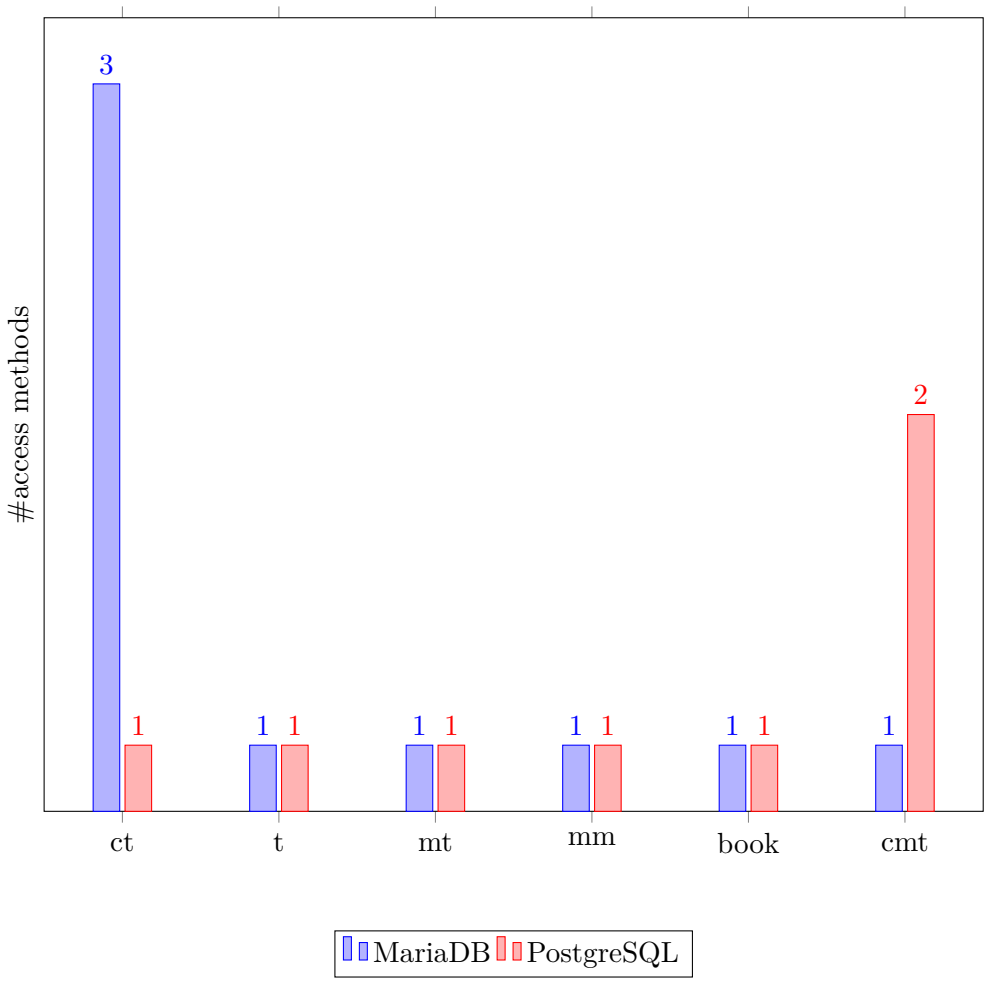

Figure 5.12: The access methods for query #2 with 1 repetition. The graph shows that even though the query is simpler than the original query, PostgreSQL still use different access methods for the relation *cmt*.

```
{
  "cmt": [
  {
    "Node Type": "Index Scan",
    "Index Name":
↪  "\"NoneCycleMasterTrade_cycle_master_trade_key_pkey\"",
    "Alias": "cmt"
  },
  {
    "Node Type": "Seq Scan",
    "Alias": "cmt"
  }
  ]
}
```

Figure 5.13: Excerpt of the tool's output when evaluating PostgreSQL with query #2 and 1 repetition. The relation shown is the same as in Figure 5.11 and it can once again be seen that the same two access methods are used by PostgreSQL.

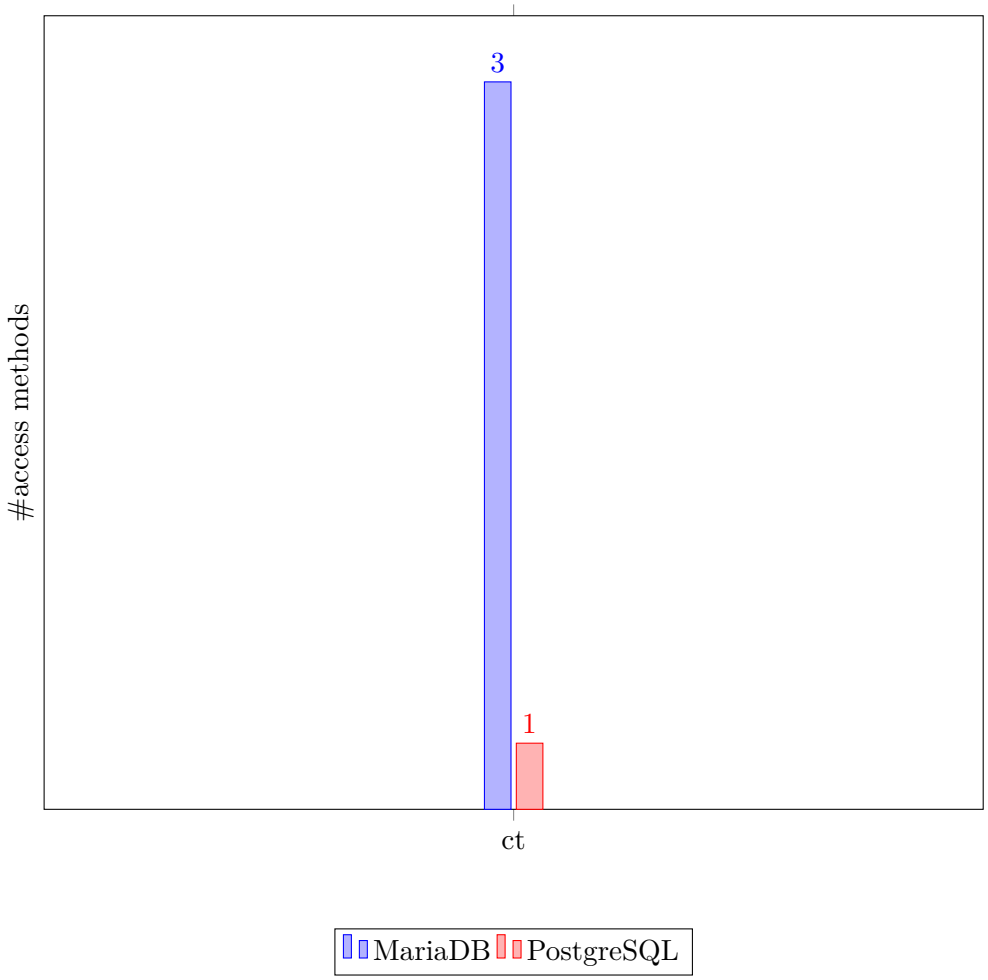

Figure 5.14: The access methods used for query #3 with 1 repetition. Only one relation is accessed, making the query the most simple variant of query #1 — yet MariaDB will still use three different access methods to access *ct*.

```
{
    "ct": [
        {
            "table": "ct",
            "key": "match_view_ref",
            "possible_keys":
 ↪  "match_view_ref,match_view_ref_2,CycleTrade_match_view_ref"
        },
        {
            "table": "ct",
            "key": "match_view_ref_2",
            "possible_keys":
 ↪  "match_view_ref,match_view_ref_2,CycleTrade_match_view_ref"
        },
        {
            "table": "ct",
            "key": "CycleTrade_match_view_ref",
            "possible_keys":
 ↪  "match_view_ref,match_view_ref_2,CycleTrade_match_view_ref"
        }
    ]
}
```


Figure 5.15: Excerpt of the tool's output when evaluating MariaDB with query #3 and 1 repetition. The output shows that the access methods for the relation *ct* are the same for query #3 and query #1.

Chapter 6

Discussion

This chapter starts with a discussion regarding the validity of the results and the possibility of using these to draw more general conclusions. Furthermore the results and what conclusions can be drawn from these are discussed. Finally, some suggestions for future research is given.

6.1 Validity of the results

Two main criticisms concerning the validity of the results can be raised:

1. Only a few number of queries were used for evaluation;

2. And only one dataset was used for evaluation.

Both of these criticisms will be answered in turn below. Finally a motivation is provided as to why the results can be considered possible to draw general conclusions from.

6.1.1 Only a few queries were used

Only one single query and subsets of that query were used, this is hardly a large set of tests for evaluation.

To answer this criticism it is important to consider the fact that the possibility for multiple access methods for the same relation depends on the criteria outlined in Section 4.1.2. The tables fulfilling these criteria are few, reducing the possibility to use multiple queries as they will only involve the same tables anyway.

Furthermore, the query constructed can be considered to be sufficiently complex to be problematic for the query optimizer, while not being unreasonably contrived. This means that if the problem does not arise for the query, it is highly unlikely to do so for a simpler query — and of little interest if it does for a more complex query.

So while only one query and subsets of it are used, they cover a good interval of simple to reasonably complex for the query optimizer to handle. This means that the results from these can be considered valid.

6.1.2 Only one dataset was used

Only one dataset was used for evaluation, the results found for the problem studied could be very different for another dataset.

This criticism is one related to the problem of evaluating databases in general: the performance of the database is often dependent on the dataset used for testing. While it is correct that the results could vary depending on dataset, this problem applies to all studies of databases.

Furthermore, the dataset used to test the databases is one taken from the real world, making it more realistic than those often otherwise used. As such, the validity of the results of this study are well on par with those of other studies using less realistic datasets like TPC-H.

So while only one dataset is used, the results found are realistic and well on par with that of other studies on the subject of databases.

6.1.3 Applicability

The results found in this study do, as all other studies of databases do, suffer from the problem of the possibility of the results depending on the dataset used. However, the results found in this study show two things:

- Different access methods are used to access the same relation;
- And the behavior of when this is done differs between MariaDB and PostgreSQL.

Neither of these results is tied to the specific dataset used, instead they are both show behavior that will very likely arise for more datasets. Adding more datasets could only answer the question of the prevalence of the behavior.

6.2 Selection of access method

This section will discuss the main findings of the evaluation, starting with a discussion regarding the correlation between sample size and the number of different access methods used for the same relation. Following this a section describing the second set of results identified: the correlation between predicate value and access methods used.

6.2.1 The effect of cardinality estimate

The main purpose of the study was to identify the effect of the cardinality estimate on the access methods selected. As a way to evaluate this the number of samples

used when analyzing and estimating the cardinality is varied from a low to a high value. The results found from the evaluation show the following:

1. PostgreSQL will vary between doing a full table scan or accessing the data via an index depending on the cardinality estimate;
2. PostgreSQL becomes more consistent in using the same access method when the cardinality is better estimated thanks to an increased sample size;
3. The cardinality estimate rather than the number of joins seem to be the main cause of PostgreSQL varying access method;
4. MariaDB remains unaffected by the cardinality estimate for the tests done;
5. The primary reason why MariaDB remains unaffected seems to be the fact that it will always use an index if one exists, never doing a full table scan.
6. And finally, MariaDB and PostgreSQL will sometimes use different access methods, even for a cardinality estimate of high quality.

PostgreSQL varies between full table scan and using an index

That PostgreSQL varies between full table scans and using an index depending on the cardinality estimated can be seen in Figure 5.1 where the relations *mt*, *cmt*, *cmm* and *est* are all accessed with two different methods. The output of the tool, seen in Figure 5.3, shows that these accesses are either using an index or a full table scan.

It is also the case that this does not happen because of the varying predicate value, as the relation *mt* has varying access methods — which is not the case when testing with only one repetition, as can be seen in Figure 5.9.

PostgreSQL is more consistent for better cardinality estimates

When testing PostgreSQL with a better cardinality estimate, as shown in Figure 5.5, the number of relations which have different access methods decrease. With the high sample size it is only the relation *cmm* that is accessed in multiple ways.

As can be seen in Figure 5.7, PostgreSQL will also choose to use an index when the cardinality is better estimated. All of the relations but *cmm* are now accessed only using an index. This indicates that the behavior PostgreSQL's query optimizer considers correct is using an index for these relations, but when the cardinality estimate varies due to a low sample size it will sometimes opt for an incorrect full table scan.

Cardinality estimate is the reason rather than complexity caused by joins

The tests done with the simpler query, query #2, showed that even though the query is simpler in terms of the number of joins and tables involved, PostgreSQL

will still vary in what access methods its using, as can be seen in Figure 5.4. This can be seen as a further motivation that the cardinality estimate, rather than other factors, is what is causing PostgreSQL to vary between what access methods its using.

MariaDB is unaffected by cardinality estimate

Unlike PostgreSQL, MariaDB seems to remain unaffected by the cardinality estimates as can be seen when comparing the results in Figure 5.1 to those in Figure 5.5. It is in both only one relation, *ct*, that is accessed with multiple different access methods and that relation has the same number of different access methods — three — for both tests. It is also the same access methods that are used, as can be seen in Figure 5.2 and Figure 5.6.

MariaDB always use an index if one exists

Another difference to PostgreSQL is that MariaDB will always use an index if one exists. This can be seen in Appendix A where the full output of the tool is shown. For all relations which have only two possible access methods — via an index on the primary key or through a full table scan — MariaDB will only ever consider using the index. It is therefore only the case that relations *ct* and *mt* can have several access methods as they have more than one index.

Both databases use many access methods for high quality cardinality estimates

The final discovery that can be drawn from the results is that both databases use multiple access methods even for high quality estimates of the cardinality. As an example consider Figure 5.5, the relation *ct* for MariaDB and *cm*m for PostgreSQL are both accessed with multiple access methods even though a high sample size is used when estimating the cardinality.

Furthermore, as MariaDB remains consistent in choosing between the three different methods regardless of the sample size used when estimating cardinality it seems that the choice is a deliberate one. This behavior was therefore further evaluated by using only 1 repetition to see if the reason for varying behavior depended on another factor than varying cardinality estimates. The results for this evaluation will be discussed in the next section, Section 6.2.2.

6.2.2 The effect of predicate value

The second evaluation done was focused on identifying if a factor other than cardinality estimate was the reason for the varying use of access methods. This evaluation was therefore done by using only 1 repetition and so on only one cardinality estimate is the result for all the query plans generated.

The evaluation found the following results:

1. Both databases will use different access methods for the same estimated cardinality;

2. The factor that is the reason for the different access methods used is the predicate used to filter rows;

3. The behavior is deliberate and will remain regardless of query complexity.

Both databases use different access methods

The first notable result is that both databases will use different access methods even when the estimated cardinality is the same. This can be clearly seen in Figure 5.9 where the relation *ct* for MariaDB and relations *cmt*, *cmm* and *est* are all accessed with multiple different access methods.

The predicate value used is important

The reason that these relations are accessed in different ways can not be because of a varying cardinality estimate, as the same estimated cardinality is used when generating all query plans. The only other thing that varies in the tests is that all possible values for `:KEY` in the query, seen in Figure 4.1, are used to generate query plans. From this it can be concluded that the the value of `:KEY` in the predicate `ct.key = :KEY` will affect the access method for the relations.

It can be further noted that for PostgreSQL the predicate will affect relations that are not directly involved in it, but also those that are joined together with it (for example *cmt*). Since MariaDB always use an index if one exists this behavior is hard to observe as the only relation it will use different access methods for is the one in the predicate, *ct*.

The behavior remains for simpler queries.

Finally, the behavior of being sensitive to predicate value was evaluated for two simpler queries to see if the result remains the same when not `JOIN`ing so many tables together.

The first test was done on query #2 and the results can be seen in Figure 5.12. The main focus was on evaluating PostgreSQL to see if it behaved the same and varied access method for the relation *cmt*. As can be seen in the figure and when comparing the output, shown in Figure 5.11 and Figure 5.13 respectively, PostgreSQL does continue to use different access methods.

For MariaDB the behavior was tested using query #3, which is a simple select only on the relation *ct*. The results can be seen in Figure 5.14 and it is clear that the behavior remains — three different access methods are used. Comparing the output, seen in Figure 5.10 and Figure 5.15, also shows that the methods are the same.

6.2.3 Conclusions from the discussion

This section will discuss what conclusions can be drawn from the discussions regarding the effect of cardinality and predicate value.

First of all, cardinality estimate has a clear effect on PostgreSQL and the access methods chosen by its query optimizer. When the sample size used to estimate cardinality is low the number of different access methods used for relations will vary much more than when the sample size used increases. This result provides a good indication as to the question that this thesis asks, and shows that cardinality estimate will affect access method chosen for PostgreSQL.

The effect of cardinality estimate can however only be observed for PostgreSQL, for MariaDB no such correlation can be seen. The primary reason that MariaDB is not as sensitive to cardinality estimate is that it never considers doing a full table scan if it can use an index. However, it is worth noting that it can not be concluded that MariaDB is unaffected by cardinality estimates. Instead it might be concluded that it is less likely for MariaDB to vary between different access methods because of different cardinality estimates, as it must be the case that it is different indexes it varies between — rather than an index or a full table scan, as is the case for PostgreSQL.

When evaluating the databases it is also found that another factor that affects the choice of access method is the predicate value used to filter rows. For PostgreSQL till affects whether it uses an index or does a full table scan, whereas it for MariaDB means it will switch between using different indexes. Since it is usually the case that one index should be the correct one for a given query this behavior might be seen as unintuitive. Furthermore, this behavior might cause a query that was first fast to become fast with the Introduction of a seemingly unrelated index, a story of this happening to a company is described in [24][Ch. 14].

One final result that is noteworthy is that the two query optimizers behave considerably different for the tests done. PostgreSQL never considers many different indexes for the same relation, unlike MariaDB. MariaDB on the other hand never considers a full table scan, unlike PostgreSQL. This is interesting as it shows that there seems to be no clear best practice for query optimizers if they differ on such fundamental levels. This also indicates that further research and evaluation of query optimizer is necessary in order to find what is the best behavior.

6.3 Future research

This section will cover some suggestions for future research on the topic of databases, both in general and specifically for the problem of selecting access method.

As discussed in Section 6.1 one problem when evaluating databases is the dataset used for evaluation. In this study a dataset based on a product for the company TriOptima was used to evaluate the databases with a real-world dataset. However, this dataset can't be made public. Other datasets, like TPC-H or the more recent JOB, suffer from the problem that they are simpler than a database used by a

company; as an example both of them have only one index per relation on the primary key.

One important area of research in databases would therefore be to create one or more realistic datasets with complex data, relations and indexes that could be used for research. Using these datasets for evaluation would then provide results that could be considered more general and correct.

The focus of this thesis was the effect of the cardinality estimate on the access methods used by the query optimizer. The results show that PostgreSQL is more sensitive to this and will more often do a full table scan when the cardinality estimate is done with a lower sample size. MariaDB on the other hand is observed to be more robust as it never does a full table scan if an index exists. This leads to the question of which behavior is the best — should the query optimizer allow full table scans or is it better to always skip them?

This study also show that both database's query optimizers select different access methods depending on predicate value and that the two do so differently. This leads to the question of whether this is the correct behavior or if the query optimizers should try to use only one access method always, regardless of predicate value.

Finally it can be noted that the results from this study show the need for further research into query optimizers as two state-of-the-art query optimizers behave considerably different. The topic of query optimization clearly needs much further study to arrive at what is best practices.

Chapter 7

Conclusions

This thesis concerns the evaluation of two modern, state-of-the-art database's, and the effect of cardinality estimates on the access methods used; the thesis question posed was: *How much effect does the cardinality estimate have on the query optimizers selection of access method during the join enumeration?*

Two databases — PostgreSQL and MariaDB — were chosen as good representations of state-of-the-art query optimizers. A large enterprise dataset was ported to both of these two databases. Finally, a query was constructed using the most complex tables in the dataset.

A tool was then implemented in Clojure to allow the two databases to be evaluated. With the tool the access methods used for each relation can be identified and tests conducted with different queries, sample sizes and repetitions.

An evaluation of the two databases was done using the query constructed and testing their choice of access methods for varying sample sizes. The results from this evaluation warranted a second evaluation using the same estimated cardinality to identify other factors than cardinality estimate that might cause the databases to select different access methods.

The results show that PostgreSQL is affected by bad cardinality estimates, and will vary between using an index or doing a full table scan for more relations when the sample size is kept small. MariaDB on the other hand was found to be less sensitive to cardinality estimate, mainly because it would never consider doing a full table scan if it could instead use an index.

The second evaluation found that MariaDB will instead vary between using different indexes depending on the predicate value used when filtering rows. This was found to be true regardless of the complexity of the query and the behavior remained consistent even for a trivial query involving only one table. PostgreSQL also displayed a similar behavior, but instead switched between using an index or doing a full table scan depending on predicate value.

In conclusion, we have found results showing that the cardinality estimate affects PostgreSQL and smaller samples used when estimating will increase the risk of doing a full table scan instead of using the correct index. MariaDB was found to be more

robust to varying cardinality estimates, because it will always use an index if one exists.

Bibliography

[1] M. Abhirama et al. “On the stability of plan costs and the costs of plan stability”. In: *Proceedings of the VLDB Endowment* 3.1-2 (2010), pp. 1137–1148. URL: http://dl.acm.org/citation.cfm?id=1920983 (visited on 01/21/2016).

[2] Martin Barksten. *mbark/ambiguous-index-finder*. GitHub. URL: https://github.com/mbark/ambiguous-index-finder (visited on 05/02/2016).

[3] Srikanth Bellamkonda et al. “Adaptive and big data scale parallel execution in oracle”. In: *Proceedings of the VLDB Endowment* 6.11 (2013), pp. 1102–1113. URL: http://dl.acm.org/citation.cfm?id=2536235 (visited on 02/08/2016).

[4] Renata Borovica-Gajic et al. “Smooth scan: Statistics-oblivious access paths”. In: *Data Engineering (ICDE), 2015 IEEE 31st International Conference on*. IEEE, 2015, pp. 315–326. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=7113294 (visited on 02/08/2016).

[5] Brian. *Better Queries: Joins*. Bounded Index. 2014-07-01T03:38:40+00:00. URL: http://bilquist.com/better-queries-joins/ (visited on 02/10/2016).

[6] Don Chamberlin. *A Complete Guide to DB2 Universal Database*. Morgan Kaufmann, 1998. 820 pp. ISBN: 978-1-55860-482-7.

[7] Surajit Chaudhuri. “An overview of query optimization in relational systems”. In: *In PODS*. 1998, pp. 34–43.

[8] Surajit Chaudhuri, Vivek Narasayya, and Ravi Ramamurthy. “Exact cardinality query optimization for optimizer testing”. In: *Proceedings of the VLDB Endowment* 2.1 (2009), pp. 994–1005. URL: http://dl.acm.org/citation.cfm?id=1687739 (visited on 01/21/2016).

[9] *Clojure*. URL: https://clojure.org/ (visited on 05/03/2016).

[10] Douglas Comer. “Ubiquitous B-Tree”. In: *ACM Comput. Surv.* 11.2 (June 1979), pp. 121–137. ISSN: 0360-0300. DOI: 10.1145/356770.356776. URL: http://doi.acm.org/10.1145/356770.356776 (visited on 02/02/2016).

[11] C.J. Date. *An Introduction to Database Systems*. 8th ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2003. ISBN: 0-321-19784-4.

[12] Cristian Estan and Jeffrey F. Naughton. "End-biased samples for join cardinality estimation". In: *Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference on*. IEEE, 2006, pp. 20–20. URL: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1617388 (visited on 02/08/2016).

[13] *EXPLAIN - MariaDB Knowledge Base*. URL: https://mariadb.com/kb/en/mariadb/explain/ (visited on 02/03/2016).

[14] Nicholas L. Farnan et al. "PAQO: a preference-aware query optimizer for PostgreSQL". In: *Proceedings of the VLDB Endowment* 6.12 (2013), pp. 1334–1337. URL: http://dl.acm.org/citation.cfm?id=2536309 (visited on 01/21/2016).

[15] Hector Garcia-Molina, Jeffrey D Ullman, and Jennifer Widom. *Database systems: the complete book*. Upper Saddle River, NJ: Prentice Hall, 2002. ISBN: 0-13-031995-3 978-0-13-031995-1.

[16] Goetz Graefe. "A Generalized Join Algorithm." In: *BTW*. Citeseer, 2011, pp. 267–286. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.401.1510&rep=rep1&type=pdf#page=285 (visited on 02/08/2016).

[17] Peter J. Haas et al. "Sampling-based estimation of the number of distinct values of an attribute". In: *VLDB*. Vol. 95. 1995, pp. 311–322. URL: http://www.vldb.org/conf/1995/P311.PDF (visited on 02/08/2016).

[18] D. Harish, Pooja N. Darera, and Jayant R. Haritsa. "Identifying robust plans through plan diagram reduction". In: *Proceedings of the VLDB Endowment* 1.1 (2008), pp. 1124–1140. URL: http://dl.acm.org/citation.cfm?id=1453976 (visited on 01/21/2016).

[19] Jayant R. Haritsa. "The Picasso database query optimizer visualizer". In: *Proceedings of the VLDB Endowment* 3.1-2 (2010), pp. 1517–1520. URL: http://dl.acm.org/citation.cfm?id=1921027 (visited on 02/08/2016).

[20] *IBM Knowledge Center - Indexable and non-indexable predicates*. Jan. 1, 2013. URL: https://www-01.ibm.com/support/knowledgecenter/SSEPEK_10.0.0/com.ibm.db2z10.doc.perf/src/tpc/db2z_indexablepredicates.dita (visited on 02/03/2016).

[21] Yannis Ioannidis. "The history of histograms (abridged)". In: *Proceedings of the 29th international conference on Very large data bases-Volume 29*. VLDB Endowment, 2003, pp. 19–30. URL: http://dl.acm.org/citation.cfm?id=1315455 (visited on 02/08/2016).

[22] Yannis E. Ioannidis and Stavros Christodoulakis. "On the Propagation of Errors in the Size of Join Results". In: *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*. SIGMOD '91. New York, NY, USA: ACM, 1991, pp. 268–277. ISBN: 0-89791-425-2. DOI: 10.1145/115790.115835. URL: http://doi.acm.org/10.1145/115790.115835 (visited on 02/08/2016).

[23] *ISO/IEC 9075-1:2011 - Information technology – Database languages – SQL – Part 1: Framework (SQL/Framework)*. URL: http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=53681 (visited on 02/01/2016).

[24] Tapio Lahdenmäki and Michael Leach. *Relational database index design and the optimizers: DB2, Oracle, SQL server, et al.* Hoboken, NJ: Wiley, 2005. 310 pp. ISBN: 978-0-471-71999-1.

[25] Viktor Leis et al. "How good are query optimizers, really?" In: *Proceedings of the VLDB Endowment* 9.3 (2015), pp. 204–215. URL: http://dl.acm.org/citation.cfm?id=2850594 (visited on 01/21/2016).

[26] Guy M. Lohman. *Is Query Optimization a "Solved" Problem?* ACM SIGMOD Blog. URL: http://wp.sigmod.org/?p=1075 (visited on 02/08/2016).

[27] *MariaDB.org*. MariaDB.org. URL: https://mariadb.org (visited on 01/27/2016).

[28] Ingo Müller et al. "Cache-Efficient Aggregation: Hashing Is Sorting". In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD '15. New York, NY, USA: ACM, 2015, pp. 1123–1136. ISBN: 978-1-4503-2758-9. DOI: 10.1145/2723372.2747644. URL: http://doi.acm.org/10.1145/2723372.2747644 (visited on 02/09/2016).

[29] A. Nica, I. Charlesworth, and M. Panju. "Analyzing Query Optimization Process: Portraits of Join Enumeration Algorithms". In: *2012 IEEE 28th International Conference on Data Engineering (ICDE)*. 2012 IEEE 28th International Conference on Data Engineering (ICDE). Apr. 2012, pp. 1301–1304. DOI: 10.1109/ICDE.2012.132.

[30] Kiyoshi Ono and Guy M. Lohman. "Measuring the Complexity of Join Enumeration in Query Optimization." In: *VLDB*. 1990, pp. 314–325. URL: http://www.researchgate.net/profile/Kiyoshi_Ono/publication/221311358_Measuring_the_Complexity_of_Join_Enumeration_in_Query_Optimization/links/0fcfd50bfe86d75dfc000000.pdf (visited on 01/27/2016).

[31] *philipsoutham/py-mysql2pgsql*. GitHub. URL: https://github.com/philipsoutham/py-mysql2pgsql (visited on 05/03/2016).

[32] *Portfolio compression service for OTC derivative dealers - triReduce | TriOptima*. URL: http://www.trioptima.com/services/triReduce.html (visited on 05/03/2016).

[33] *PostgreSQL: Documentation: 9.0: Planner/Optimizer*. URL: http://www.postgresql.org/docs/9.0/static/planner-optimizer.html (visited on 01/26/2016).

[34] *PostgreSQL: Documentation: 9.3: EXPLAIN*. URL: http://www.postgresql.org/docs/9.3/static/sql-explain.html (visited on 02/03/2016).

[35] *PostgreSQL: The world's most advanced open source database*. URL: http://www.postgresql.org/ (visited on 01/27/2016).

[36] Raghu Ramakrishnan and Johannes Gehrke. *Database management systems*. 3. ed., international ed. Boston, Mass.: McGraw-Hill, 2003. 1065 pp. ISBN: 978-0-07-246563-1 978-0-07-115110-8 978-0-07-123057-5.

[37] P. Griffiths Selinger et al. "Access Path Selection in a Relational Database Management System". In: *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data*. SIGMOD '79. New York, NY, USA: ACM, 1979, pp. 23–34. ISBN: 0-89791-001-X. DOI: 10.1145/582095.582099. URL: http://doi.acm.org/10.1145/582095.582099 (visited on 01/29/2016).

[38] *SQL | computer language*. Encyclopedia Britannica. URL: http://www.britannica.com/technology/SQL (visited on 02/01/2016).

[39] *Summary of predicate processing*. Oct. 2014. URL: https://www-01.ibm.com/support/knowledgecenter/api/content/nl/en-us/SSEPEK_10.0.0/com.ibm.db2z10.doc.perf/src/tpc/db2z_summarypredicateprocessing.dita (visited on 02/03/2016).

[40] Mana Takahashi, Shoko Azuma, and Ltd Trend-Pro Co. *The Manga Guide to Databases*. 1 edition. Tokyo, Japan; San Francisco, Calif.: No Starch Press, Feb. 7, 2009. 224 pp. ISBN: 978-1-59327-190-9.

[41] *The Join Operation*. URL: http://Use-The-Index-Luke.com/sql/join (visited on 02/09/2016).

[42] *TPC-DS - Homepage*. URL: http://www.tpc.org/tpcds/ (visited on 05/02/2016).

[43] *TPC-H - Homepage*. URL: http://www.tpc.org/tpch/ (visited on 01/26/2016).

[44] Immanuel Trummer and Christoph Koch. "Multi-Objective Parametric Query Optimization". In: VLDB. 2015. URL: https://infoscience.epfl.ch/record/206967 (visited on 01/21/2016).

[45] Kostas Tzoumas, Amol Deshpande, and Christian S. Jensen. "Lightweight graphical models for selectivity estimation without independence assumptions". In: *Proceedings of the VLDB Endowment* 4.11 (2011), pp. 852–863. URL: http://www.vldb.org/pvldb/vol4/p852-tzoumas.pdf (visited on 02/08/2016).

[46] David Vengerov et al. “Join size estimation subject to filter conditions”. In: *Proceedings of the VLDB Endowment* 8.12 (2015), pp. 1530–1541. URL: http://dl.acm.org/citation.cfm?id=2824051 (visited on 02/09/2016).

[47] Wentao Wu et al. “Predicting query execution time: Are optimizer cost models really unusable?” In: *2013 IEEE 29th International Conference on Data Engineering (ICDE)*. 2013 IEEE 29th International Conference on Data Engineering (ICDE). Apr. 2013, pp. 1081–1092. DOI: 10.1109/ICDE.2013.6544899.

[48] Feng Yu et al. “CS2: a new database synopsis for query estimation”. In: *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM, 2013, pp. 469–480. URL: http://dl.acm.org/citation.cfm?id=2463701 (visited on 02/09/2016).

Appendix A

Program output

A.1 PostgreSQL

A.1.1 Evaluation 1


```
{
    "mm": [
        {"Node Type": "Index Only Scan", "Index Name":
 ↪  "\"NoneMasterMatch_master_match_key_pkey\"", "Alias": "mm"}
    ],
    "book": [
        {"Node Type": "Seq Scan", "Alias": "book"}
    ],
    "resamb": [
        {"Node Type": "Seq Scan", "Alias": "resamb"}
    ],
    "bti": [
        {"Node Type": "Seq Scan", "Alias": "bti"}
    ],
    "cmm": [
        {"Node Type": "Index Scan", "Index Name":
 ↪  "\"NoneCycleMasterMatch_cycle_master_match_key_pkey\"", "Alias":
 ↪  "cmm"},
        {"Node Type": "Seq Scan", "Alias": "cmm"}
    ],
    "cmt": [
        {"Node Type": "Index Scan", "Index Name":
 ↪  "\"NoneCycleMasterTrade_cycle_master_trade_key_pkey\"", "Alias":
 ↪  "cmt"},
        {"Node Type": "Seq Scan", "Alias": "cmt"}
    ],
    "t": [
        {"Node Type": "Index Scan", "Index Name":
 ↪  "\"NoneTrade_trade_key_pkey\"", "Alias": "t"}
    ],
    "est": [
```

```
        {"Node Type": "Index Scan", "Index Name": "\"NoneExternalService
↪  Trade_cycle_master_trade_ref_external_servic\"", "Alias":
↪  "est"},
        {"Node Type": "Seq Scan", "Alias": "est"}
    ],
    "ct": [
        {"Node Type": "Index Scan", "Index Name":
↪  "\"NoneCycleTrade_match_view_ref\"", "Alias": "ct"}
    ],
    "mt": [
        {"Node Type": "Seq Scan", "Alias": "mt"},
        {"Node Type": "Index Scan", "Index Name":
↪  "\"NoneMasterTrade_master_trade_key_pkey\"", "Alias": "mt"}
    ]
}
```


Listing A.1.1: The output when testing PostgreSQL with query #1, a sample size of 1 and 50 repetitions.


```
{
    "mm": [
        {"Node Type": "Index Only Scan", "Index Name":
↪  "\"NoneMasterMatch_master_match_key_pkey\"", "Alias": "mm"}
    ],
    "book": [
        {"Node Type": "Seq Scan", "Alias": "book"}
    ],
    "resamb": [
        {"Node Type": "Seq Scan", "Alias": "resamb"}
    ],
    "bti": [
        {"Node Type": "Seq Scan", "Alias": "bti"}
    ],
    "cmm": [
        {"Node Type": "Index Scan", "Index Name":
↪  "\"NoneCycleMasterMatch_cycle_master_match_key_pkey\"", "Alias":
↪  "cmm"},
        {"Node Type": "Seq Scan", "Alias": "cmm"}
    ],
    "cmt": [
        {"Node Type": "Index Scan", "Index Name":
↪  "\"NoneCycleMasterTrade_cycle_master_trade_key_pkey\"", "Alias":
↪  "cmt"}
    ],
    "t": [
        {"Node Type": "Index Scan", "Index Name":
↪  "\"NoneTrade_trade_key_pkey\"", "Alias": "t"}
    ],
    "est": [
        {"Node Type": "Index Scan", "Index Name": "\"NoneExternalService
↪  Trade_cycle_master_trade_ref_external_servic\"", "Alias":
↪  "est"}
    ],
```

```
    "ct": [
        {"Node Type": "Index Scan", "Index Name":
↪   "\"NoneCycleTrade_match_view_ref\"", "Alias": "ct"}
    ],
    "mt": [
        {"Node Type": "Index Scan", "Index Name":
↪   "\"NoneMasterTrade_master_trade_key_pkey\"", "Alias": "mt"}
    ]
}
```

Listing A.1.2: The output when testing PostgreSQL with query #1, a sample size of 1 and 50 repetitions.

A.1.2 Evaluation 2

```
{
 "mm": [
        {
         "Node Type": "Index Only Scan",
         "Index Name": "\"NoneMasterMatch_master_match_key_pkey\"",
         "Alias": "mm"
         }
        ],
 "book": [
          {
           "Node Type": "Seq Scan",
           "Alias": "book"
           }
          ],
 "resamb": [
            {
             "Node Type": "Seq Scan",
             "Alias": "resamb"
             }
            ],
 "bti": [
         {
          "Node Type": "Seq Scan",
          "Alias": "bti"
          }
         ],
 "cmm": [
         {
          "Node Type": "Index Scan",
          "Index Name":
↪   "\"NoneCycleMasterMatch_cycle_master_match_key_pkey\"",
          "Alias": "cmm"
          },
         {
          "Node Type": "Seq Scan",
          "Alias": "cmm"
          }
```

```
        ],
"cmt": [
        {
         "Node Type": "Index Scan",
         "Index Name":
↪  "\"NoneCycleMasterTrade_cycle_master_trade_key_pkey\"",
         "Alias": "cmt"
         },
        {
         "Node Type": "Seq Scan",
         "Alias": "cmt"
         }
        ],
"t": [
       {
        "Node Type": "Index Scan",
        "Index Name": "\"NoneTrade_trade_key_pkey\"",
        "Alias": "t"
        }
       ],
"est": [
        {
         "Node Type": "Index Scan",
         "Index Name": "\"NoneExternalServiceTrade_cycle_master_trade_r⌋
↪  ef_external_servic\"",
         "Alias": "est"
         },
        {
         "Node Type": "Seq Scan",
         "Alias": "est"
         }
        ],
"ct": [
        {
         "Node Type": "Index Scan",
         "Index Name": "\"NoneCycleTrade_match_view_ref\"",
         "Alias": "ct"
         }
        ],
"mt": [
        {
         "Node Type": "Seq Scan",
         "Alias": "mt"
         }
        ]
}
```


Listing A.1.3: The output when testing PostgreSQL with query #1, a sample size of 1 and 1 repetitions.


```
{
    "mt": [
```

```
            {
                "Node Type": "Seq Scan",
                "Alias": "mt",
            }
        ],
        "ct": [
            {
                "Node Type": "Index Scan",
                "Index Name": "\"NoneCycleTrade_match_view_ref\"",
                "Alias": "ct",
            }
        ],
        "t": [
            {
                "Node Type": "Index Scan",
                "Index Name": "\"NoneTrade_trade_key_pkey\"",
                "Alias": "t",
            }
        ],
        "mm": [
            {
                "Node Type": "Index Only Scan",
                "Index Name": "\"NoneMasterMatch_master_match_key_pkey\"",
                "Alias": "mm",
            }
        ],
        "book": [
            {
                "Node Type": "Seq Scan",
                "Alias": "book",
            }
        ]
}
```


Listing A.1.4: The output when testing PostgreSQL with query #2, a sample size of 1 and 1 repetitions.


```
{
    "steps": [
        "generate",
        "parse",
        "analyze"
    ],
    "query": "postgresql9",
    "samplesizes": [
        1
    ],
    "repetitions": 1,
    "database": "postgresql"
}
{
    "ct": [
```

```
        {
            "Node Type": "Index Scan",
            "Index Name": "\"NoneCycleTrade_match_view_ref\"",
            "Alias": "ct",
        }
    ]
}
```


Listing A.1.5: The output when testing PostgreSQL with query #3, a sample size of 1 and 1 repetitions.

A.2 MariaDB

A.2.1 Evaluation 1


```
{
    "mm": [
        {"table": "mm", "key": "PRIMARY", "possible_keys": "PRIMARY"}
    ],
    "book": [
        {"table": "book", "key": "PRIMARY", "possible_keys": "PRIMARY"}
    ],
    "resamb": [
        {"table": "resamb", "key": "master_trade_ref", "possible_keys":
 ↪  "master_trade_ref"}
    ],
    "bti": [
        {"table": "bti", "key": "cycle_trade_ref", "possible_keys":
 ↪  "cycle_trade_ref"}
    ],
    "cmm": [
        {"table": "cmm", "key": "PRIMARY", "possible_keys": "PRIMARY"}
    ],
    "cmt": [
        {"table": "cmt", "key": "PRIMARY", "possible_keys": "PRIMARY"}
    ],
    "t": [
        {"table": "t", "key": "PRIMARY", "possible_keys": "PRIMARY"}
    ],
    "est": [
        {"table": "est", "key": "cycle_master_trade_ref",
 ↪  "possible_keys": "cycle_master_trade_ref"}
    ],
    "ct": [
        {"table": "ct", "key": "match_view_ref", "possible_keys":
 ↪  "match_view_ref,match_view_ref_2,CycleTrade_master_trade_ref,CycleT⌋
 ↪  rade_trade_ref,CycleTrade_match_view_ref"},
        {"table": "ct", "key": "match_view_ref_2", "possible_keys":
 ↪  "match_view_ref,match_view_ref_2,CycleTrade_master_trade_ref,CycleT⌋
 ↪  rade_trade_ref,CycleTrade_match_view_ref"},
```

```
        {"table": "ct", "key": "CycleTrade_match_view_ref",
↪   "possible_keys": "match_view_ref,match_view_ref_2,CycleTrade_master
↪   _trade_ref,CycleTrade_trade_ref,CycleTrade_match_view_ref"}
    ],
    "mt": [
        {"table": "mt", "key": "PRIMARY", "possible_keys":
↪   "PRIMARY,MasterTrade_master_match_ref"}
    ]
}
```

Listing A.2.1: The output when testing MariaDB with query #1, a sample size of 1 and 50 repetitions.

```
{
    "mm": [
        {"table": "mm", "key": "PRIMARY", "possible_keys": "PRIMARY"}
    ],
    "book": [
        {"table": "book", "key": "PRIMARY", "possible_keys": "PRIMARY"}
    ],
    "resamb": [
        {"table": "resamb", "key": "master_trade_ref", "possible_keys":
↪   "master_trade_ref"}
    ],
    "bti": [
        {"table": "bti", "key": "cycle_trade_ref", "possible_keys":
↪   "cycle_trade_ref"}
    ],
    "cmm": [
        {"table": "cmm", "key": "PRIMARY", "possible_keys": "PRIMARY"}
    ],
    "cmt": [
        {"table": "cmt", "key": "PRIMARY", "possible_keys": "PRIMARY"}
    ],
    "t": [
        {"table": "t", "key": "PRIMARY", "possible_keys": "PRIMARY"}
    ],
    "est": [
        {"table": "est", "key": "cycle_master_trade_ref",
↪   "possible_keys": "cycle_master_trade_ref"}
    ],
    "ct": [
        {"table": "ct", "key": "match_view_ref", "possible_keys":
↪   "match_view_ref,match_view_ref_2,CycleTrade_master_trade_ref,CycleT
↪   rade_trade_ref,CycleTrade_match_view_ref"},
        {"table": "ct", "key": "match_view_ref_2", "possible_keys":
↪   "match_view_ref,match_view_ref_2,CycleTrade_master_trade_ref,CycleT
↪   rade_trade_ref,CycleTrade_match_view_ref"},
        {"table": "ct", "key": "CycleTrade_match_view_ref",
↪   "possible_keys": "match_view_ref,match_view_ref_2,CycleTrade_master
↪   _trade_ref,CycleTrade_trade_ref,CycleTrade_match_view_ref"}
    ],
```

```
    "mt": [
        {"table": "mt", "key": "PRIMARY", "possible_keys":
↪   "PRIMARY,MasterTrade_master_match_ref"}
    ]
}
```

Listing A.2.2: The output when testing MariaDB with query #1, a sample size of 1 and 50 repetitions.

A.2.2 Evaluation 2


```
{
 "mm": [
        {
         "table": "mm",
         "key": "PRIMARY",
         "possible_keys": "PRIMARY"
         }
        ],
 "book": [
          {
           "table": "book",
           "key": "PRIMARY",
           "possible_keys": "PRIMARY"
           }
          ],
 "resamb": [
            {
             "table": "resamb",
             "key": "master_trade_ref",
             "possible_keys": "master_trade_ref"
             }
            ],
 "bti": [
         {
          "table": "bti",
          "key": "cycle_trade_ref",
          "possible_keys": "cycle_trade_ref"
          }
         ],
 "cmm": [
         {
          "table": "cmm",
          "key": "PRIMARY",
          "possible_keys": "PRIMARY"
          }
         ],
 "cmt": [
         {
          "table": "cmt",
          "key": "PRIMARY",
          "possible_keys": "PRIMARY"
```

```
          }
         ],
"t": [
      {
       "table": "t",
       "key": "PRIMARY",
       "possible_keys": "PRIMARY"
       }
      ],
"est": [
        {
         "table": "est",
         "key": "cycle_master_trade_ref",
         "possible_keys": "cycle_master_trade_ref"
         }
        ],
"ct": [
       {
        "table": "ct",
        "key": "match_view_ref",
        "possible_keys": "match_view_ref,match_view_ref_2,CycleTrade_ma⌋
↪  ster_trade_ref,CycleTrade_trade_ref,CycleTrade_match_view_ref"
        },
       {
        "table": "ct",
        "key": "match_view_ref_2",
        "possible_keys": "match_view_ref,match_view_ref_2,CycleTrade_ma⌋
↪  ster_trade_ref,CycleTrade_trade_ref,CycleTrade_match_view_ref"
        },
       {
        "table": "ct",
        "key": "CycleTrade_match_view_ref",
        "possible_keys": "match_view_ref,match_view_ref_2,CycleTrade_ma⌋
↪  ster_trade_ref,CycleTrade_trade_ref,CycleTrade_match_view_ref"
        }
       ],
"mt": [
       {
        "table": "mt",
        "key": "PRIMARY",
        "possible_keys": "PRIMARY,MasterTrade_master_match_ref"
        }
       ]
}
```


Listing A.2.3: The output when testing MariaDB with query #1, a sample size of 1 and 1 repetitions.


```
{
    "steps": [
        "generate",
        "parse",
```

```
        "analyze"
    ],
    "query": "mariadb4",
    "samplesizes": [
        1
    ],
    "repetitions": 1,
    "database": "mariadb"
}
{
 "ct": [
        {
         "table": "ct",
         "key": "match_view_ref",
         "possible_keys": "match_view_ref,match_view_ref_2,CycleTrade_ma⌋
 ↪  ster_trade_ref,CycleTrade_trade_ref,CycleTrade_match_view_ref"
        },
        {
         "table": "ct",
         "key": "match_view_ref_2",
         "possible_keys": "match_view_ref,match_view_ref_2,CycleTrade_ma⌋
 ↪  ster_trade_ref,CycleTrade_trade_ref,CycleTrade_match_view_ref"
        },
        {
         "table": "ct",
         "key": "CycleTrade_match_view_ref",
         "possible_keys": "match_view_ref,match_view_ref_2,CycleTrade_ma⌋
 ↪  ster_trade_ref,CycleTrade_trade_ref,CycleTrade_match_view_ref"
        }
       ],
 "mt": [
        {
         "table": "mt",
         "key": "PRIMARY",
         "possible_keys": "PRIMARY,MasterTrade_master_match_ref"
        }
       ],
 "mm": [
        {
         "table": "mm",
         "key": "PRIMARY",
         "possible_keys": "PRIMARY"
        }
       ],
 "t": [
       {
        "table": "t",
        "key": "PRIMARY",
        "possible_keys": "PRIMARY"
       }
      ],
 "book": [
          {
           "table": "book",
```

```
            "key": "PRIMARY",
            "possible_keys": "PRIMARY"
            }
          ]
}
```

Listing A.2.4: The output when testing MariaDB with query #2, a sample size of 1 and 1 repetitions.

```
{
    "ct": [
        {
            "table": "ct",
            "key": "match_view_ref",
            "possible_keys":
 ↪  "match_view_ref,match_view_ref_2,CycleTrade_match_view_ref"
        },
        {
            "table": "ct",
            "key": "match_view_ref_2",
            "possible_keys":
 ↪  "match_view_ref,match_view_ref_2,CycleTrade_match_view_ref"
        },
        {
            "table": "ct",
            "key": "CycleTrade_match_view_ref",
            "possible_keys":
 ↪  "match_view_ref,match_view_ref_2,CycleTrade_match_view_ref"
        }
    ]
}
```

Listing A.2.5: The output when testing MariaDB with query #3, a sample size of 1 and 1 repetitions.