HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF INFORMATION AND COMMUNICATION TECHNOLOGY

THESIS

SUBMITTED FOR PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

ENGINEER
IN

INFORMATION TECHNOLOGY

BENCHMARKING CARDINALITY IN POSTGRESQL

Author:	**PHAM Tuan Hiep**
	Class SIC-PFIEV K57
Supervisor:	**Mr. Christophe Bobineau**
	Associate Professor of Grenoble Institute of Technology
	Mrs. Vu Tuyet Trinh
	Doctor of Hanoi University of Science and Technology
	Mr. Maxence Ahlouche
	PhD student at Grenoble Institute of Technology

HANOI May 19, 2017

REQUIREMENT FOR THE THESIS

1. Student information

Student name: Pham Tuan Hiep
Tel: +84 918 632 952 Email: hieppso194@gmail.com
Class: System of Information and Communication Program - PFIEV
This thesis is performed at: Grenoble Computer Science Laboratory (LIG)
From 15/02/2017 to 15/06/2017

2. Goals of the thesis

- Finding the factors that affect to the performance of DBMS execution
- Proving the negatively effects of these factors by benchmarking
- Proposing the solutions to reduce their effects

3. Main tasks

- Finding the factors that affect to the performance of DBMS execution
- Finding the benchmarks.
- Benchmarking PostgreSQL with the benchmarks that were choosen .
- Proving the effect of the bad cardinality estimation to PostgreSQL performance.

4. Student information

I Pham Tuan Hiep - hereby warrant that the work and presentation in this thesis are performed by myself under the supervisor of Mr. Christophe Bobineau, Mr. Vu Tuyet Trinh and Mr. Maxence Alouche. All results presented in this thesis are truthful and are not copied from any other work

Grenoble, 1 June 2017
Author

Pham Tuan Hiep

5. Attestation of the supervisor on the fulfillment of the requirements of the thesis

Hanoi, 1 June 2017
Supervisor

Vu Tuyet Trinh

ABSTRACTION OF THESIS

Nowadays, The data throughout the world has been exponentially increasing and become extremely valuable. So it is challenging for us to manage and exploit it. Although there are many advanced database management systems which have been developed to solve this challenge, they have still exposed their drawbacks in managing the big volume data sets. Their default executor showed bad performance in executing the query on these data sets. Hence, it is neccessary for us to find the solutions to improve the performance of database management systems.

There are many factors that affect the performance of database management systems, such as: cost model, cardinality estimation, etc. It is hard to focus on all of these factors to solve the problem. Therefore, Finding the factor that has the most significant effect to the performance of database management systems plays an important role.

This project focus on finding this main factor and proving its effect by benchmarking it in PostgreSQL and analyzing its effect. Besides, I also propose the published solutions to reduce its effect.

First of all, I introduce the context about the way that a query is executed. I will focus on the steps that a query is processed. And after that, I find and prove the step that affects to the query performance the most. And then i dig deeper into this step to find the factors that affect to query performance by researching query optimization.

Before benchmarking the factors, I find the benchmarks that are used to benchmark executing query performance of database management systems. And then i try and research them to find the best benchmarks for benchmarking the factor in PostgreSQL.

Next, I benchmark the factor in PostgreSQL by using the benchmarks that i already choose. And then i analyze the results to show the effects of the factor to query performance. And finally, I will propose some solutions that have been published to reduce their effects and improve the query performance.

This project is made at HADAS team, LIG Laboratory under the supervisions of Dr. Christophe Bobineau - Associate Professor of Grenoble Institute of Technology, Maxence Alouche - PhD student at Grenoble INP and Dr. Vu Tuyet Trinh - SOICT.

ACKNOLEDGEMENT

Firstly, I would like to send all my thanks to Mr. Christophe Bobineau - who gives me chances for doing the internship in team HADAS of LIG, for all of his advises and supports during my internship as well as for helping me to correct this thesis.

I would like to thank most sincerely to Mr. Maxence Alouche who patiently helps me get familiar with the new working environment, who gives me advice and always answers all question as soon as i need, who guides me step by step during the process of implementing this project.

I am also thankful to Mrs. Christine Collet - Header of Heterogenous Autonomous distributed DAta Services Team (HADAS) for her advises about my research and presentation.

I also wish to thank Mrs. Vu Tuyet Trinh for all her advises to help me complete successfully.

Other thanks are for all members of team HADAS as well as staffs of LIG who supported me during my internship.

Finally, I would like to express my gratitude and my thanks to my family and my friends in both Hanoi and in Grenoble for their encouragements.

Contents

LIST OF FIGURES

LIST OF TABLES

Chapter 1

INTRODUCTION

1.1 CONTEXT AND OBJECTIVE OF THE THESIS

1.1.1 CONTEXT

Nowadays, new information sources, such as social networks, mobile devices, sensors, recreationals have been already generated unprecedented volumes of data. These sources are stored in data sets that can reach petabytes or exabytes.

According to IBM, 90% of currently generated information has been created in the last two years. As a consequence of this growth, the term Big Data has been popularized, which is destined to become one of the most promising technology trends in the coming years.

More and more companies are realizing that the large amounts of information that they accumulate can play a critical role in the decision making carried out by management teams and the creation of new business. However, this is not an easy task and there are some challenges ahead to achieve it. One of these challenges is the query performance of data management systems on the big datasets.

For any production database, Query performance becomes an issue sooner or later. Having long-running queries not only consumes system resources that lead the server and application run slowly, but also may lead to table locking and data corruption issues. So, query optimization becomes an important task.

There has been many factors that affect to query optimization and it is hard to focus on all of them to satisfy query optimization aim. Therefore, it is extremely necessary to find the factor that affects query performance the most.

1.1.2 OBJECTIVE OF MY THESIS

In this thesis, I have decided to focus primarily on finding the factor that affects query performance the most and showing its effect to query performance. Besides, I will also propose some published solutions to reduce its effects. There are steps in my thesis:

- Understanding how a query is executed.
- Finding the factors in query optimization.

- Finding the benchmarks to benchmark this factor.
- Benchmarking and analyzing the effect of this factor.
- Proposing the published solutions to reduce the effect.

Chapter 2

QUERY OPTIMIZATION.

2.1 HOW IS A QUERY EXECUTED?

2.1.1 The main steps to execute a query.

There are five main components in Postgresql architecture [11]:

- The parser - parse the query string
- The rewriter - apply rewrite rules
- The optimizer - determine an efficient query plan
- The executor - execute a query plan
- The utility processor - process DDL like **CREATE TABLE**

Firgure 2.1 shows the architecture diagram of query executor in Postgresql. In the first step, query string is parsed to parse tree in the parser. After that, parse tree is analyzed its semantic and transformed to Query node. In the next step PostgreSQL processes DDL (Data Definition Language) like **CREATE TABLE** or apply rewrite rules in rewriter. Before going to executor, query tree is used to produce a query plan.

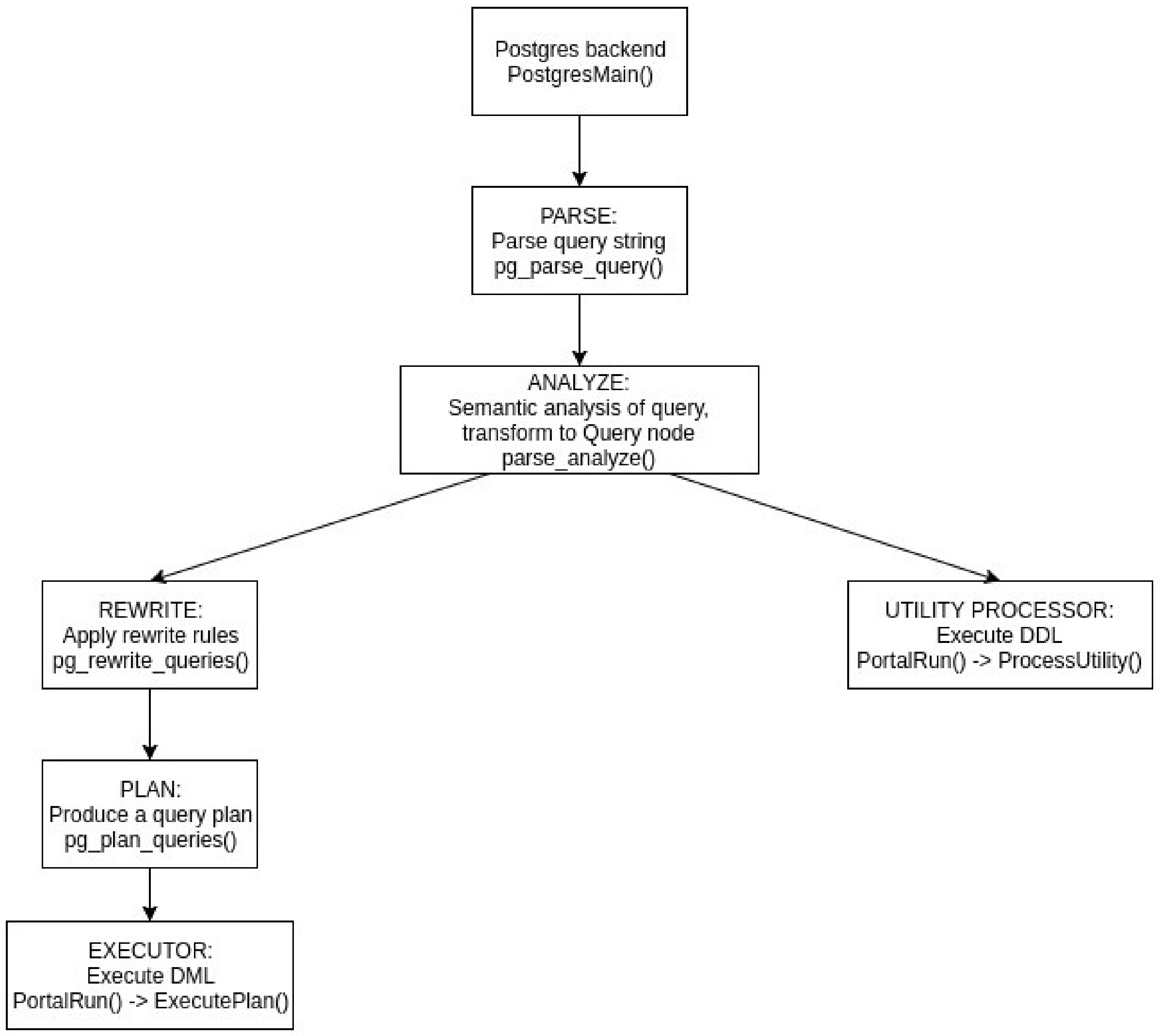


Figure 2.1: Architecture diagram of Postgresql.

2.1.2 Parser

The parser [12] has to check the query string (which arrives as plain ASCII text) for valid syntax. If the syntax is correct a parse tree is built up and handed back; otherwise an error is returned. It is defined in gram.y and scan.l which are built by using the Unix tools yacc and lex.

It turns out that PostgreSQL uses the same parsing technology that Ruby does, a parser generator called **Bison**. Bison runs during the PostgreSQL C build process and generates parser code based on a series of grammar rules. The generated parser code is what runs inside of PostgreSQL when we send it SQL commands. Each grammar rule is triggered when the generated parser finds a corresponding pattern or syntax in the SQL string, and inserts a new C memory structure into the parse tree data structure.

Figure 2.2 is an example about the parse tree of the query:

```
SELECT * FROM movies
WHERE country = 'France' ORDER BY id ASC LIMIT 1
```

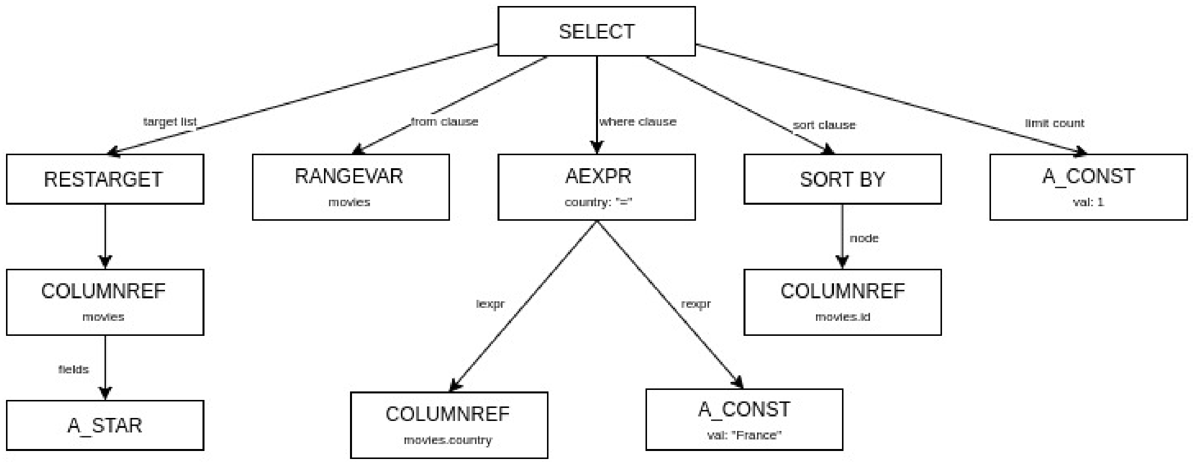


Figure 2.2: An example about parse tree.

2.1.3 Rewriter

Once PostgreSQL has generated a parse tree, it then converts it into another tree using a different set of nodes. This is known as the query tree. The parts of a query tree [11] are:

- The command type: This is a simple value telling which command (SELECT, INSERT, UPDATE, DELETE) produced the query tree.
- The range table: The range table is a list of relations that are used in the query. In a SELECT statement these are the relations given after the FROM key word.
- The result relation: This is an index into the range table that identifies the relation where the results of the query go.
- The target list: The target list is a list of expressions that define the result of the query. In the case of a SELECT, these expressions are the ones that build the final output of the query. They correspond to the expressions between the key words SELECT and FROM (* is just an abbreviation for all the column names of a relation. It is expanded by the parser into the individual columns, so the rule system never sees it.).
- The qualification: The query's qualification is an expression much like one of those contained in the target list entries. The result value of this expression is a Boolean that tells whether the operation (INSERT, UPDATE, DELETE, or SELECT) for the final result row should be executed or not. It corresponds to the WHERE clause of an SQL statement.
- The join tree: The query's join tree shows the structure of the FROM clause. For a simple query like SELECT ... FROM a, b, c, the join tree is just a list of the FROM items, because we are allowed to join them in any order. But when JOIN expressions, particularly outer joins, are used, we have to join in the order shown by the joins. In that case, the join tree shows the structure of the JOIN expressions. The restrictions associated with particular JOIN clauses (from ON or USING expressions) are stored as qualification expressions attached to those join-tree nodes. It turns out to be convenient to store the top-level WHERE expression as a qualification attached to the top-level

join-tree item, too. So really the join tree represents both the FROM and WHERE clauses of a SELECT.

- The others: The other parts of the query tree like the ORDER BY clause

Figure 2.3 is an example about the query tree of the query:

```
SELECT * FROM movies
WHERE country = 'France' ORDER BY id ASC LIMIT 1
```

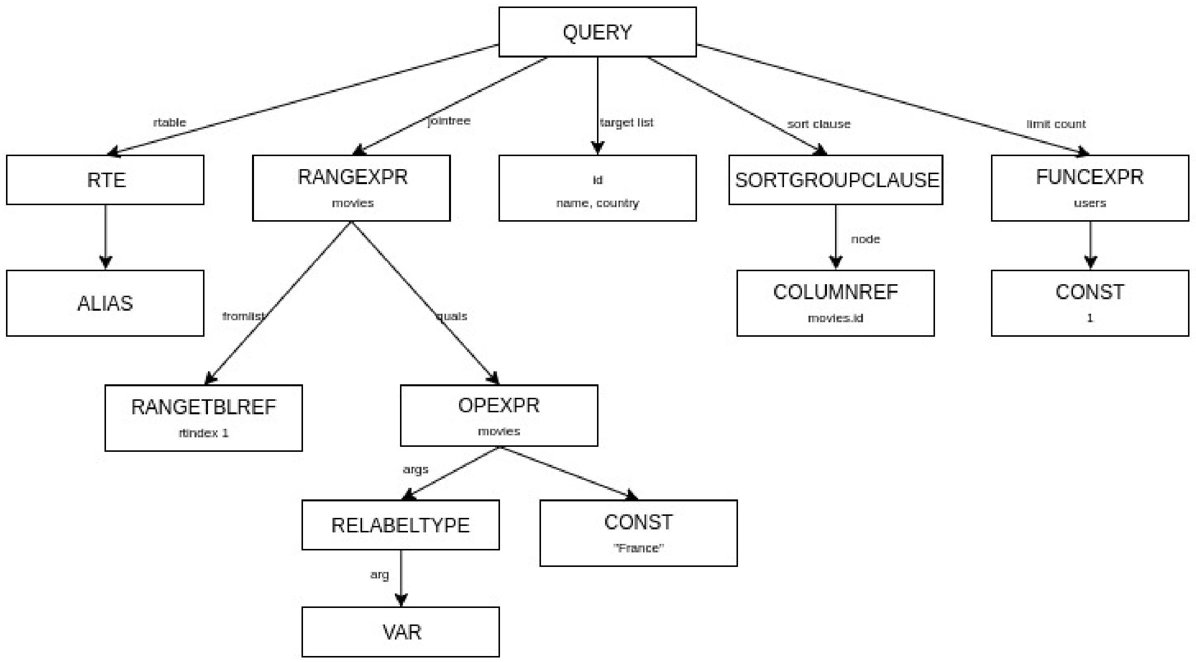


Figure 2.3: An example about query tree.

2.1.4 Planner

The task of the planner [12] is to create an optimal execution plan. A query tree of a SQL plan can be actually executed in a wide variety of different ways, each of them will produce the same of results. If it is computationally feasible, the query optimizer will examine each of these possible execution plans, ultimately selecting the execution plan that is expected to run the fastest.

The planner's search procedure actually works with data structures called paths, which are simply cut-down representations of plans containing only as much information as the the planner needs to make its decisions. After the cheapest path is determined, a full-fledged plan tree is built to pass to the executor. This represents the desired execution plan in sufficient detail for the executor to run it.

Firgure 2.4 is the full-fledged plan tree of the query:

```
SELECT * FROM movies
WHERE country = 'France' ORDER BY id ASC LIMIT 1
```

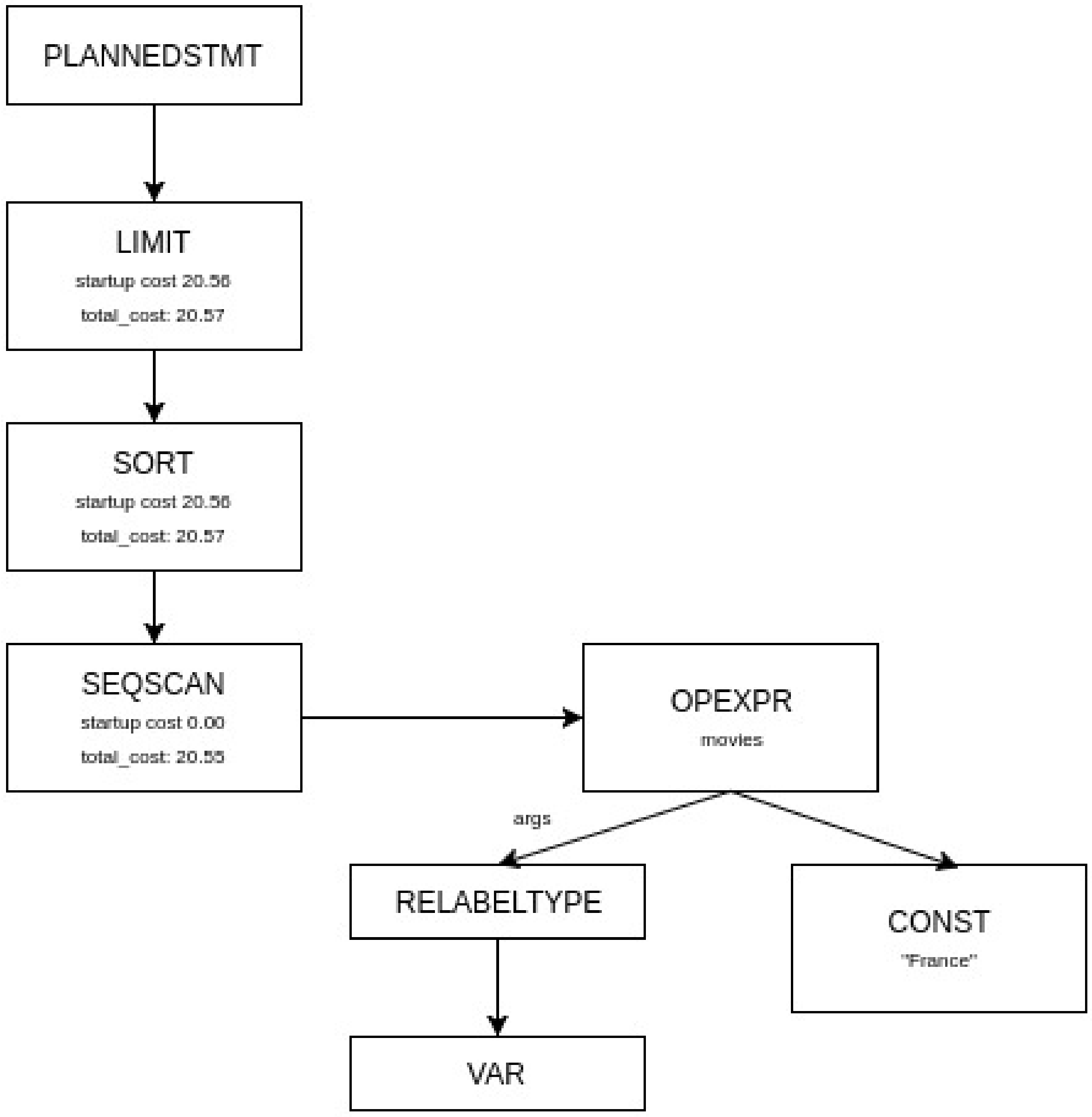


Figure 2.4: An example about full-fledged plan tree.

2.1.5 Executor

The executor [11] takes the plan handed back by the planner and recursively processes it to extract the required set of rows. This is essentially a demand-pull pipeline mechanism. Each time a plan node is called, it must deliver one more row, or report that it is done delivering rows.

To provide an example, assume that the top node is a MergeJoin node. Before any merge can be done two rows have to be fetched (one from each subplan). So the executor recursively calls itself to process the subplans (it starts with the subplan attached to lefttree). The new top node (the top node of the left subplan) is, let's say, a Sort node and again recursion is needed to obtain an input row. The child node of the Sort might be a SeqScan node, representing actual reading of a table. Execution of this node causes the executor to fetch a row from the table and return it up to the calling node. The Sort node will repeatedly call its child to obtain all the rows to be sorted. When the input is exhausted (as indicated by the child node returning a NULL instead of a row), the Sort code performs the sort, and finally is able to return its first output row, namely the first one in sorted order. It keeps the remaining rows stored so that it can deliver them in sorted order in response to later demands.

The MergeJoin node similarly demands the first row from its right subplan. Then it compares the two rows to see if they can be joined; if so, it returns a join row to its caller. On the next call, or immediately if it cannot join the current pair of inputs, it advances to the next row of one table or the other (depending on how the comparison came out), and again checks for a match. Eventually, one subplan or the other is exhausted, and the MergeJoin node returns NULL to indicate that no more join rows can be formed.

Complex queries can involve many levels of plan nodes, but the general approach is the same: each node computes and returns its next output row each time it is called. Each node is also responsible for applying any selection or projection expressions that were assigned to it by the planner.

2.2 QUERY OPTIMIZATION.

2.2.1 Introduction.

Query optimization is a function of relational database management systems. The query optimizer attempts to determine the most efficient way to execute a given query by considering the possible query plans. Therefore, it is belong to planner stage.

There is a trade-off between the amount of time spent figuring out the best query plan and the quality of the choice; the optimizer may not choose the best answer on its own. Different qualities of database management systems have different ways of balancing these two. Cost-based query optimiers, such as PostgreSQL optimizer, evaluate the resource footprint of various query plans and use this as the basis for plan selection. They assign an estimated ”cost” to each possible query plan, and choose the plan with the smallest cost. Costs are used to estimate the runtime cost of evaluating the query, in terms of the number of I/O operations required, CPU path length, amount of disk buffer space, disk storage service time, and interconnect usage between units of parallelism. The set of query plans examined is formed by examining the possible access paths (e.g., primary index access, secondary index access, full file scan) and various relational table join techniques (e.g., merge join, hash join). The search space can become quite large depending on the complexity of the SQL query. There are two types of optimization. These consist of logical optimization which generates a sequence of relational algebra to solve the query and physical optimization which is used to determine the means of carrying out each operation.

2.2.2 The query optimizer of PostgreSQL

As i mentioned above, PostgreSQL’s optimizer is a cost-based optimizer, so it chooses the plan that have the lowest estimated processing cost to execute. PostgreSQL’s optimizer determines the cost of executing a query plan based on two main factors:

- Cardinality estimation: the total number of rows processed at each level of a query plan, referred to as the cardinality of the plan.
- Cost model: the cost model of the algorithm dictated by the operators used in the query

The first factor, cardinality, is used as an input parameter of the second factor, the cost model. Therefore, improving cardinality estimation leads to better estimated costs and, in turn, faster execution plans.

2.2.3 PostgreSQL's cardinality estimator and its limitations.

PostgreSQL's cardinality estimator

Postgresql's optimizer follows the traditional textbook architecture. The cardinalities of base tables are estimated using histograms (quantile statistics), most common values with their frequencies, and domain cardinalities (distinct value counts). These per-attribute statistics are computed by the analyze command using a sample of the relation. For complex predicates, histograms can not be applied. To combine conjunctive predicates for the same table, PostgreSQL simply assumes independence and multiplies the selectivities of the individual selectivity estimates

The assumptions that PostgreSQL's cardinality estimator based on:

- Independence: predicates on attributes (in the same table or from joined tables) are independent.
- Uniformity: all values, except for the most-frequent ones, are assumed to have the same number of tuples
- Priciple of inclusion: the domains of the join keys overlap such that the keys from the smaller domain have matches in the larger domain.

The formula [1] that is used to estimate result sizes of joins show these assumptions:

$$|T_1 \bowtie_{x=y} T_2| = \frac{|T_1||T_2|}{max(dom(x),dom(y))}$$

Where T_1 and T_2 are arbitrary expresions and dom(x) is the domain cardinality of attribute x, i.e., the number of distinct values of x.

The limitation of PostgreSQL's cardinality estimator

- Correlated data : Correlated data [13] between relations can lead to the bad cardinality estimation of PostgreSQL executor. To be more specific, i use the example below to demonstrate the impact of the correlation between relations to cardinality estimation of PostgreSQL:

  ```
  SELECT * FROM Products where Company = 'Toyota' and Brand = 'Civic'
  ```

  When you look at this query, you will know immediately how many rows are returned. The number of returned results is zero because the company Toyota doesn't have Civic brand car. But as i metioned above, the PostgreSQL's optimizer uses the assumptions (uniformity, independence, inclusion, ad hoc constants), so it looks at each search predicate indepedently.

  In the first step the cardinality estimation is done for the predicate **Company = 'Toyota'**. After that, the optimizer produces a cardinality estimation for other predicate **Brand = 'Civic'**. And finally both estimations are multiplied by each other to produce the final estimation. When the first predicate produces a cardinality selectivity of 0.4 and the second one produces a cardinality selectivity of 0.4, the final cardinality selectivity will be 0.16 (0.4*0.4) instead of 0.0 as the fact. The PostgreSQL's optimizer handles every predicate on its own without any correlation between relations.

Chapter 3

FINDING BENCHMARKS

3.1 REASEARCHING BENCHMARK.

3.1.1 TPC benchmarks

TPC-H

The TPC-H [16] is a decision support benchmark. It consists of a suite of business oriented to ad-hoc queries and concurrent data modifications. The queries and the data populating the database have been chosen to have broad industry-wide relevance while maintaning a sufficient degree of ease of implementation. This benchmark illustrates decision support systems that

- Examine large volumes of data.
- Execute queries with a high degree of complexity.
- Give answers to critical business questions.

TPC-H evaluates the performance of various decision support systems by the execution of sets of queries against a standard database under controlled conditions. The TPC-H queries:

- Give answers to real-world business questions.
- Simulate generated ad-hoc queries (e.g., via a point and click GUI interface);
- Are far more complex than most OLTP transactions.
- Include a rich breadth of operators and selectivity constraints;
- Generate intensive activity on the part of the database server component of the system under test.
- Are executed against a database complying to specific population and scaling requirements. The relations in this database don't have the correlated data together.
- Are implemented with constraints derived from staying closely synchronized with an on-line production database.
- contains few joins, between 0 to 4 joins each query. For example, 1.sql doesn't have any join in this query, 2.sql contains 3 joins.

TPC-DS

TPC-DS [16] is a decision support benchmark that models several generally applicable aspects of a decision support system (DSS), including queries and data maintenance:

- User queries, which convert operational facts into business intelligence.
- Data maintenance, which synchronizes the process of managenment analysis with the operational external data source on which it relies.

The benchmark provides a representative evaluation of the System Under Test's(SUT) performance as a general purpose decision support system.

TPC-DS illustrates decision support systems that:

- Examine large volumes of data.
- Give answers to real-world business questions.
- Execute queries of various operational requirements and complexities
- Are characterized by high CPU and IO load
- Are periodically synchronized with source Online Transaction Porcessing (OLTP) databases through database maintenance functions.
- Run on "Big Data" solutions, such as RDBMS as well as Hadoop/Spark based systems.

There are four broad classes of queries that characterize most decision support queries:

- Reporting queries: These queries capture the "reporting" nature of a DSS system. They include queries that are executed periodcally to answer well-known, pre-defined questions about the financial and operational health of a business. They tend to be static.
- Ad hoc queries: These queries capture the dynamic nature of a DSS system in which impromtu queries are constructed to answer immediate and specific business questions.
- Iterative Online Analytical Processing (OLAP) queries: These queries are used for the exploration and analysis of business data to discover new and meaningful relationships and trends. It is distinguished with Ad hoc queries by a scenario-based user session in which a sequence of queries is submitted.
- Data mining queries: These queries can be used to predict future trends and behaviors, allowing business to make proactive, knowledge-driven decisions. This class of queries typically consists of joins and large aggregation that return large data result sets for possible extraction.

There are joins in each query, especially the queries which are belong to the class of data mining.

- For example: query 91 contains 6 joins

TPC-E

TPC-E [16] is an OLTP workload. It is a mixture of read-only and update intensive transactions that simulate the activities found in complex OLTP application environments. The database schema, data population, transactions, and implementation rules have been designed to be broadly representative of modern OLTP systems. It exercises a breadth of system components associated with such environments, which are characterized by:

- The simultaneous execution of multiple transaction types that span a breadth of complexity.
- A balanced mixture of disk input/output and processor usage.
- Transaction integrity.
- A mixture of uniform and non-uniform data access through primary and secondary keys.
- Databases consisting of many tables with a wide variety of sizes, attributes, and relationships with realistic content.
- Contention on data access and update.
- Moderate system and application execution time.

The goal of TPC-E is simulating the OLTP workload of a brokerage firm. The focus of the benchmark is the central database that executes transactions related to the firm's customer accounts. In keeping with the goal of measuring the performance characteristics of the database system, the benchmark does not attempt to measure the complex flow of data between multiple application systems that would exist in a real environment.

There are limitations in this benchmark:

- Benchmark results are significantly dependent upon workload, specific application requirements, and systems design and implementation. Relative system performance will vary because of those and other factors. Therefore, it should not used as a substitue or specific customer application benchmarking when critical capacity planning and product evaluation decisions are contemplated.
- There are few joins in each query and the queries totally focus on the transactions.

3.1.2 Star Schema Benchmark

The Star Schema Benchmar (SSB) [10] was designed to test star schema optimization to address the issues outlined in TPC-H with the goal of measuring performance of database products and to test a new materialization strategy. It is a simple benchmark that consists of four query flights, four dimensions. The SSB is significantly based on the TPC-H benchmark with improvements that implements a traditional pure star-schema and allows column and table compression.

The SSB is designed to measure performance of database products against a traditional data warehouse scheme. It implements the same logical data in a tradional star schema, whereas TPC-H models the data in pseudo 3NF schema.

Some model queries in SSB benchmar are based on the TPC-H query set, but need to be modified to vary the selectivity. There are the queries that don't have an equal conterpart in TPC-H.

- Query Flights: Compared to the TPC-H's 22 queries, SSB contains four query flights that each consist of three to four queries that vary selectivity. Each flight consists of a sequence of queries that someone working with data warehouse systems.

- Caching: One other issue arises in running the Star Schema Benchmark queries, and that is the caching effect that reduces the number of disk accesses necessary when query Q2 follows query Q1, because of overlap of data accessed between Q1 and Q2. SSB attempts to minimize overlap, however in situations where this cannot be done, SSB will take whatever steps are needed to reduce caching effects of one query on another.

3.1.3 JOB benchmark

JOB bechmark [1] contains two main components:

- IMDB Data set: It contains a plethora of information about movies and related facts related actors, directors, production companies. The data is freely available for non-commercial use as text files. Besides, there is an open source, namely imdbpy, to transform the text files into a relational database with 21 tables. As most real-world data sets, IMDB is full of correlations and non-uniform data distributions, and is therefore much more challenging than most synthetic data sets. Now, The size of this data set is about 20 Gigabytes.

- JOB queries: They are based on the IMDB data set and is constructed as analytical SQL queries. They focus on join ordering, which argualy is the most important query optimization problem. They were designed to have between 3 and 16 joins, with an average of 8 joins per query. For example, Query 13d, which finds the ratings and release dates for all movies produced by US companies, contains 11 joins.

Figure 5 is the typical query gapth of JOB workload. In this graph, The solid edges in the graph represent key/foreign key edges (1 : n) with the arrow head pointing to the primary key side. Dotted edges represent foreign key/foreign key joins (n : m), which appear due to transitive join predicates.

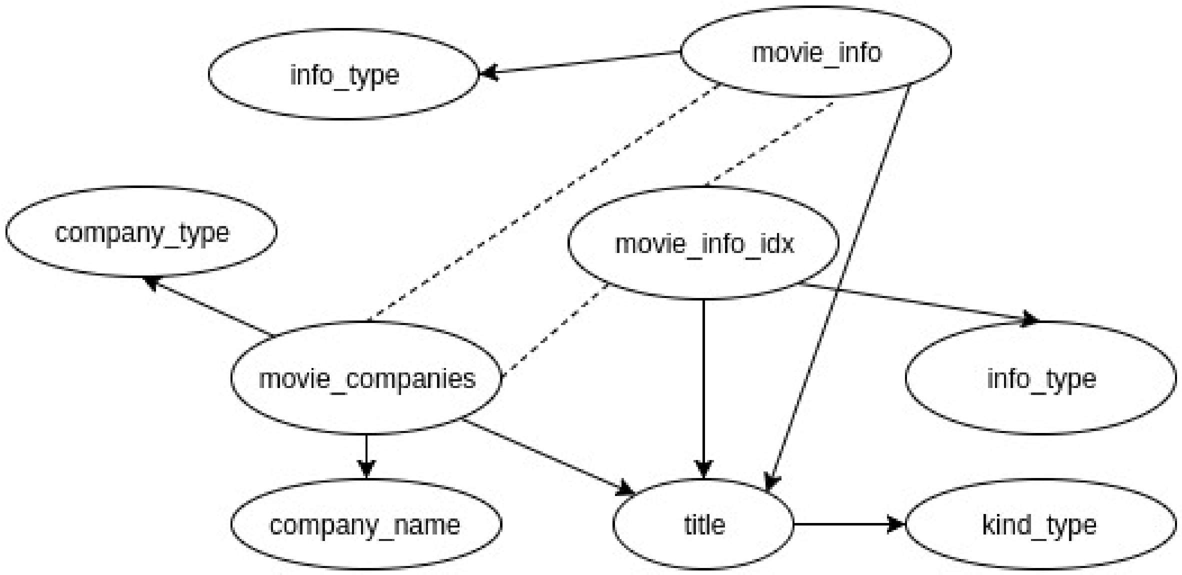


Figure 3.1: Typical query graph of JOB workload.

For cardinality estimators the queries are challenging due to the significant number of joins and the correlations contained in the dataset. However, The queries did not try to trick the query optimizer, e.g., by picking attributes with extreme correlations. Also, they intentionally did not include more complex join predicates like inequalities or non-surrogate-key predicates, because cardinality estimation for this workload is already quite challenging.

3.2 CONCLUSION

Although TPC-H, TPC-E, TPC-DS or the Star Schema Benchmark (SSB) have proven their value for evaluating query engines, I argue that they are not good bechmark for the cardinality estimation component of query optimizers. The reason is that in order to easily be able to scale the benchmark data, the data generators are using the very same simplifying assumptions (uniformity, independence, principle of inclusion) which are the reasons leading to the limitaions of cardinality estimator of PostgresSQL and that query optimizers make. Real-world data sets, in contrast, are full of correlations and non-uniform data distributions, which makes cardinality estimation much harder. Therefore, I don't use these benchmarks for benchmarking the cardinality estimation component of PostgreSQL's query optimizer.

Besides, JOB benchmark solved the limitations of the benchmarks above. Their queries contain many joins and their data set are full of correlations and non-uniform data distributions, which makes cardinality estimator of PostgreSQL encounter many hardships. So it is very suitable to be used for benchmarking cardinality estimation component of PostgreSQL's optimizer.

Chapter 4

BENCHMARKING CARDINALITY ESTIMATION ON POSTGRESQL

4.1 BASIC THEORIES AND IMPORTANT CONCEPTS.

4.1.1 Q-error

The Definition of Q-error [2].

Let R be a relation and A be one of its attributes, and $\{x_1, ..., x_m\} = \Pi_A(R)$ is the set of distinct values of A. Then, the frequency density is a set of pairs (x_i, f_i) with $f_i = |\sigma_{A=x_i}(R)|$, $1 \leq i \leq m$

F is a function to approximate this set of paris; $F(x_i) =: F_i$ gives the estimate F_i of f_i. And F is a polynomial of degree 0 or 1.

Typically, norms are used to define the error. So, the correct values are organized into a vector $\vec{b} = (f_1, ..., f_m)^T \in \mathbb{R}^m$ and the estimates into a vector $\vec{B} = (F_1, ..., F_m)^T \in \mathbb{R}^m$. l_p error metrics are based on l_p norms as in

$$\|b - B\|_p$$

where $1 \leq p \leq \infty$ and the most common norms are

$$\|z\|_2 = \sqrt{(z_1)^2 + ... + (z_m)^2}$$
$$\|z\|_\infty = \overset{m}{\underset{i=1}{max}}|z_i|$$

for z $= (z_1, ..., z_m)^T \in \mathbb{R}^m$. While the l_2 error does not give bound on estimates, l_∞ does. Define $\Delta = \|b - B\|_\infty$. Then

$$f_i - \Delta \leq F_i \leq f_i + \Delta.$$

However, this error bounds are not really useful in the context of query optimization. For $z \in \mathbb{R}$, a multiplicative error:

$$\|z\|_Q = \begin{cases} \infty \quad if \quad z \leq 0 & (4.1a) \\ \dfrac{1}{z} \quad if \quad 0 < z \leq 1 & (4.1b) \\ z \quad if \quad 1 \leq z & (4.1c) \end{cases}$$

For $z > 0$, this is the same as saying $\|z\|_Q = max\left(z, \frac{1}{z}\right)$. Thus, it treats over and underestimates symmetrically. For a vector $z \in \mathbb{R}^m$, define:

$$\|z\|_Q = \overset{m}{\underset{i=1}{max}} \|z_i\|_Q$$

Let $\vec{a}$ and $\vec{b}$ be two vectors in $\mathbb{R}^m$ where b_i. Define $\vec{a}/\vec{b} = \frac{\vec{a}}{\vec{b}} = (a_1/b_1, ..., a_n/b_n)^T$. Then, we can define the q-error of an estimation B of b as

$$\|B/b\|_Q$$

As l_∞, l_q produces valid, symmetric bounds for individual estimates. Define q = $\|B/b\|_Q$. Then,

$$\left(\frac{1}{q}\right) f_i \leq F_i \leq q f_i$$

Note that the error bounds are symmetric and multiplicative. The q-error is rarely used in the literature. The only exception are from the area of sampling

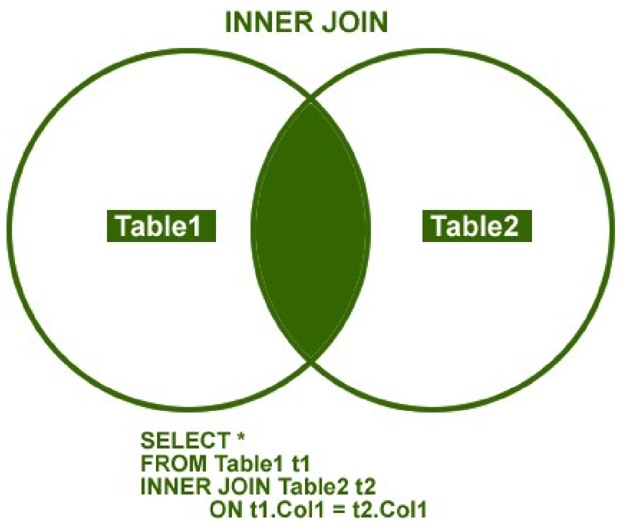
Q-error which is used to measure the quality of base table cardinality estimates is the factor by which an estimate differs from the true cardinality. For example, if the true cardinality of an expression is 100, the estimates of 10 or 1000 both have a q-error of 10. Using the ratio instead of an absolute or quadratic difference captures the intuition that for making planning decisions only relative differences matter. The q-error furthermore provides a theoretical upper bound for the plan quality if the q-errors of a query are bounded.

4.1.2 Logical join operators

Logical operators [14] decribe the relational algebraic operation used to process a statement. In other words, logical operators decribe conceptually what operation needs to be performed. In PostgreSQL. there are logical join operators, such as Inner join, Outer join, Cross join.

Inner join

An inner join simply looks for two rows that put together satisfy a join predicate. For example, this query used the join predicate ”customer.id=salary.customerid” to find all Customer and Sale rows with the same customerid.

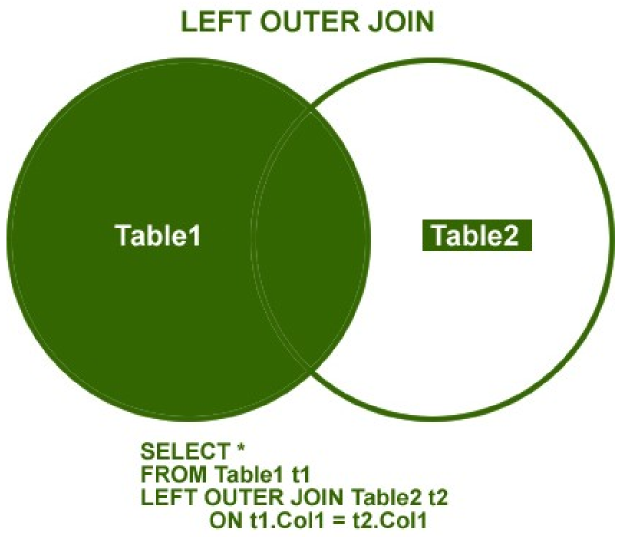


```
SELECT *
FROM Table1 t1
INNER JOIN Table2 t2
        ON t1.Col1 = t2.Col1
```

Figure 4.1: An example about inner join.

Outer join

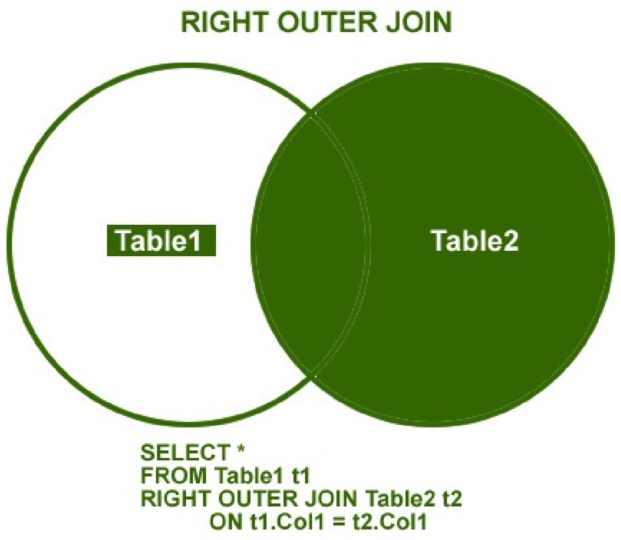
- Left outer join: The result of a left outer join for tables A and B always contains all rows of the ”left” table (A), even if the join-condition does not find any matching row in the ”right” table (B). This means that if the **ON** clause matches 0 rows in B (for a given row in A), the join will still return a row in the result (for that row) - but with NULL in each column from B. A left outer join returns all the values from an inner join plus all values in the left table that do not match to right table, including rows with NULL (empty) values in the link column.



```
SELECT *
FROM Table1 t1
LEFT OUTER JOIN Table2 t2
        ON t1.Col1 = t2.Col1
```

Figure 4.2: An example about left outer join.

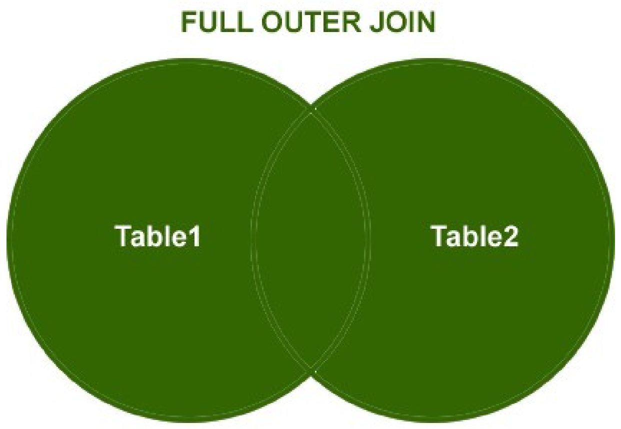
Right outer join: A right outer join (or right join) closely resembles a left outer join, except with the treatment of the tables reversed. Every row from the "right" table (B) will appear in the joined table at least once. If no matching row from the "left" table (A) exists, NULL will appear in columns from A for those rows that have no match in B. A right outer join returns all the values from the right table and matched values from the left table (NULL in the case of no matching join predicate).



```
SELECT *
FROM Table1 t1
RIGHT OUTER JOIN Table2 t2
      ON t1.Col1 = t2.Col1
```

Figure 4.3: An example about right outer join.

- Full outer join: Conceptually, a full outer join combines the effect of applying both left and right outer joins. Where rows in the FULL OUTER JOINED tables do not match, the result set will have NULL values for every column of the table that lacks a matching row. For those rows that do match, a single row will be produced in the result set (containing columns populated from both tables).



```
SELECT *
FROM Table1 t1
FULL OUTER JOIN Table2 t2
      ON t1.Col1 = t2.Col1
```

Figure 4.4: An example about full outer join

Cross join

The SQL Cross join produces a result set which is the number of rows in the first table multiplied by the number of rows in the second table if no Where clause is used along with Cross join. If where clause is use with Cross join, it functions like an INNER JOIN

4.1.3 Physical join operators

We use logical operators when we write queries to define a relational query at the conceptual level. To implement these logical join operators, DBMS use three different physical operators. Three physical join operators are respectively Nested Loops Join, Hash Join, Merged Join

Nested Loops Join

A Nested Loops join [15] is a logical structure in which one loop (iteration) sedides inside another one, that is to say for each iteration of the outer loop all the iterations of the inner loop are executed/processed. To be more specific, For each row of the outer table, all the rows from the inner table are matched one by one if the row matches it is included in the result-set otherwise it is ignored. Then the next row from the outer table is picked up and the same process is repeated and so on.

In terms of complexity, If we assume that N is the number of rows from the outer output and the number of rows that table is M. The complexity of this join operator is O(NlogM). Therefore, The PostgreSQL's optimizer might choose a Nested Loops join when one of the joining tables is small (considered as the outer table) and another one is large (consider as the inner table which is indexed on the column that is in the join).

Merge Join

The first thing about a Merge join [15] is that it requires both inputs to be sorted on join keys/merge columns (or both input tables have clustered indexes on the column that joins the tables). Because the rows are pre-sorted, a Merge join immediately begins the matching process. It reads a row from one input and then compares it with the row of another input. If the rows match, that matched row is considered in the result-set (then it reads the next row from the input table, does the same comparison/match) or else the lesser of the two rows is ignored and the process continues this way until all rows have been processed.

In terms of complexity, If we assume that N is the number of rows from the outer output and the number of rows that table is M. The complexity of this join operator is O(N+M). If the inputs are not both sorted on the join key, the PostgreSQL optimizer will often choose the Merge join type due the the cost of pre-sorting process.

Hash join.

A Hash join [15] is performed in two phases; the Build phase and the Probe phase and so the hash join has two inputs i.e, build input and probe input. The smaller of the input tables is considered as the build input and the other one is probe input

During the build phase, joining keys of all the rows of the build table are scanned. Hashes are generated and placed in an in memory hash table. During the probe phase, joining keys of each row of the the probe table scanned. Again hashes are generated and compared against the corresponding hash table for a match.

A Hash function requires significant amount of CPU cycles to generate hashes and memory resources to store the hash table. If there is memory pressure, some of the partitions of the hash table are swapped to tempdb and whenever there is a need, it is brought back into cache. To achieve high performance, the PostgreSQL optimizer may parallelize a Hash join to scale better than any other join

In term of complexity, well assume h_c is the complexity of the hash table creation, and h_m is the complexity of the hash match function. Therefore, the complexity of the Hash join will be O(N*h_c + M*h_m + J) where N is the smaller data set, M is the larger data set and J is a joker complexity addition for the dynamic calculation and creation of the hash function.

4.2 BENCHMARKING.

I use JOB benchmark queries to examine the q-error of Postgresql estimator's cardinality estimation. The volume of JOB data set is about 20GB. Figure 4.5 shows the q-error of plan node by its join level. Figure 4.6 shows the q-error of plan node per query. Figure 4.7 and figure 4.8 show respectively the q-error of plan node by join tree depth, the actual execution time and estimated execution time of queries.

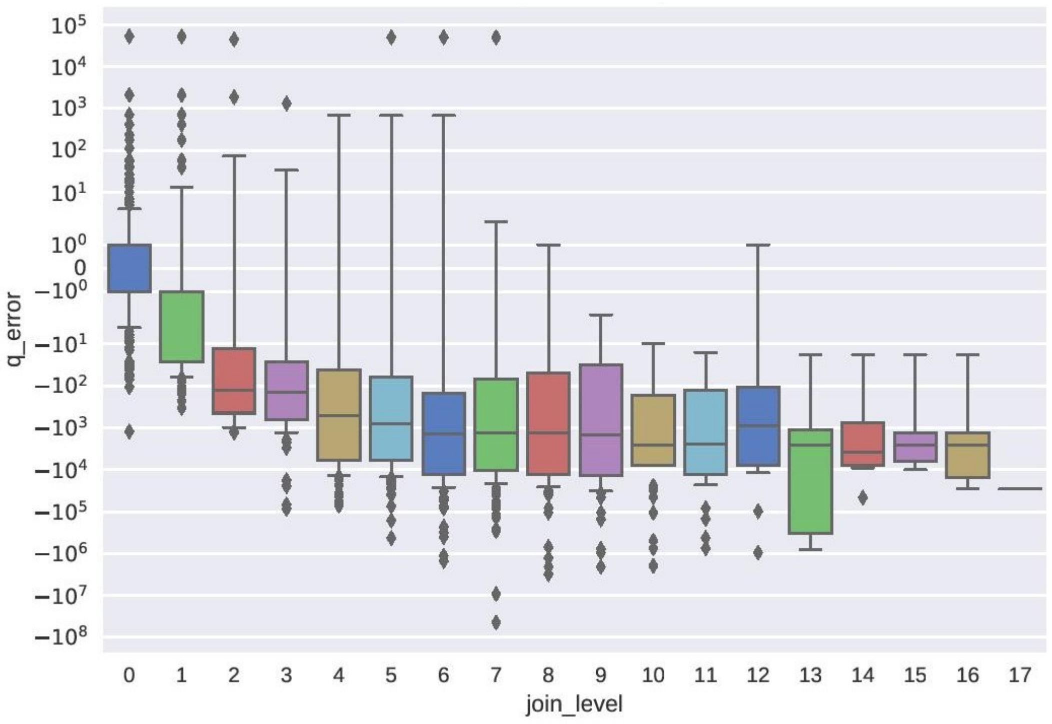


Figure 4.5: Q-error of plan node and its join level.

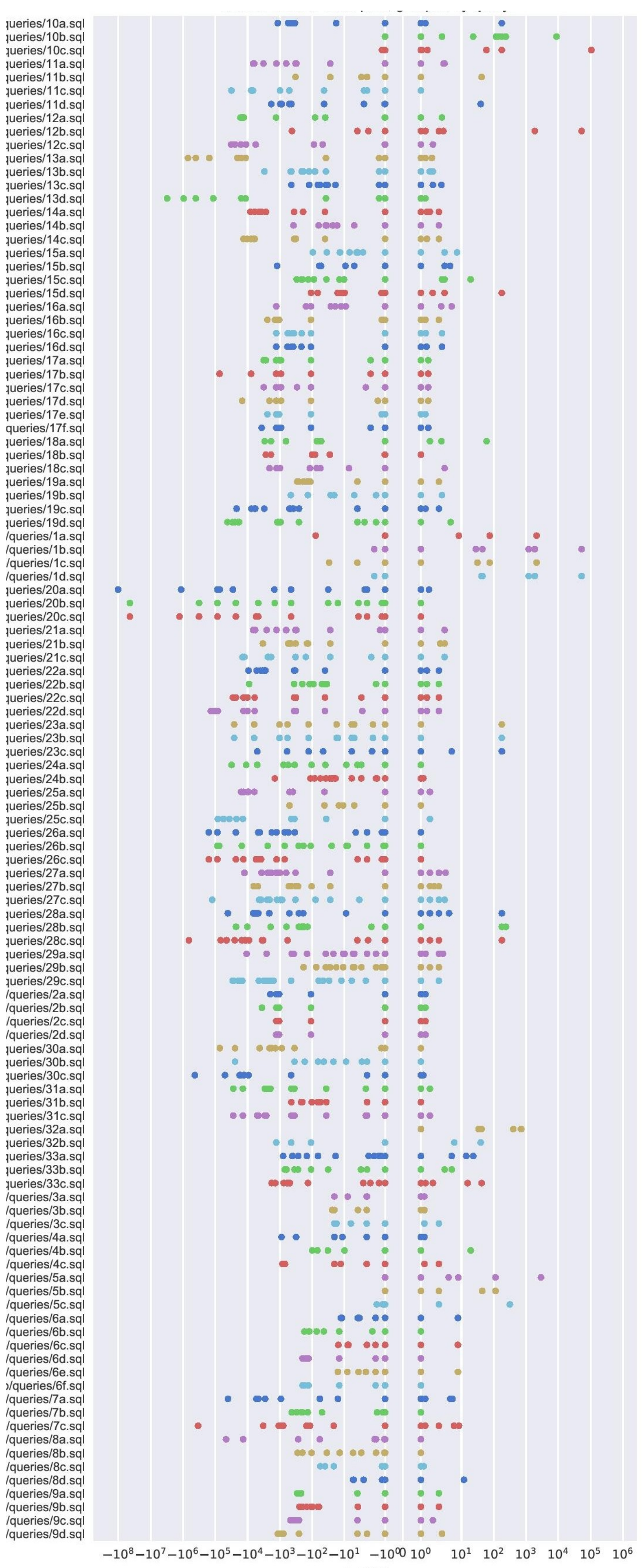


Figure 4.6: Q-error of plan node per query.

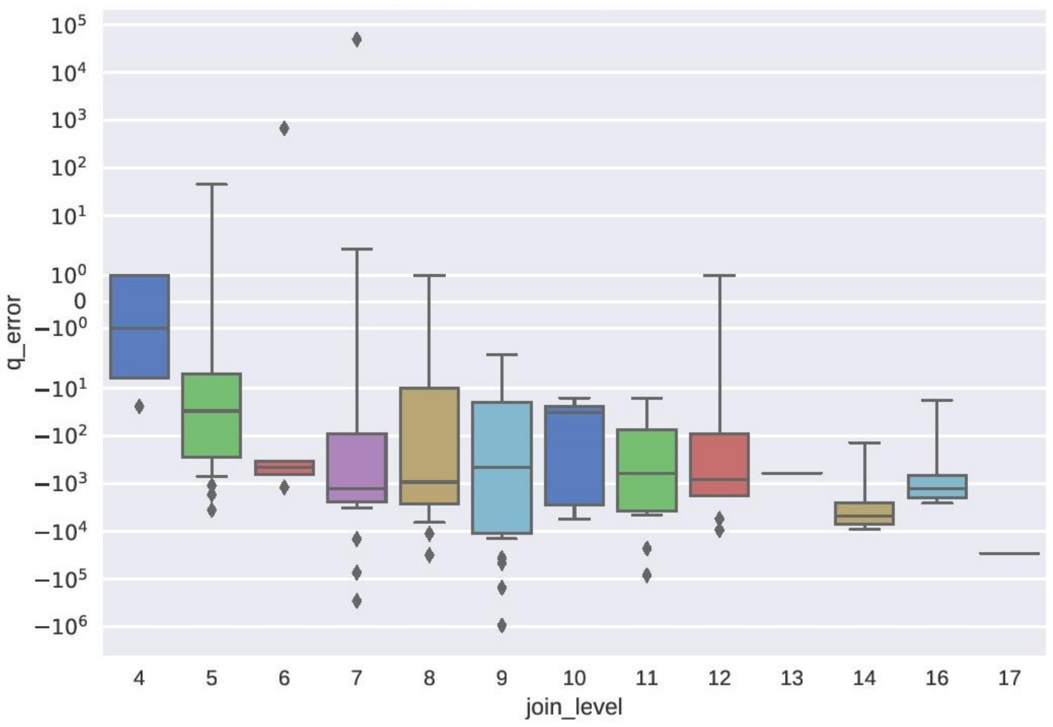


Figure 4.7: Q-error of plan node and join tree depth.

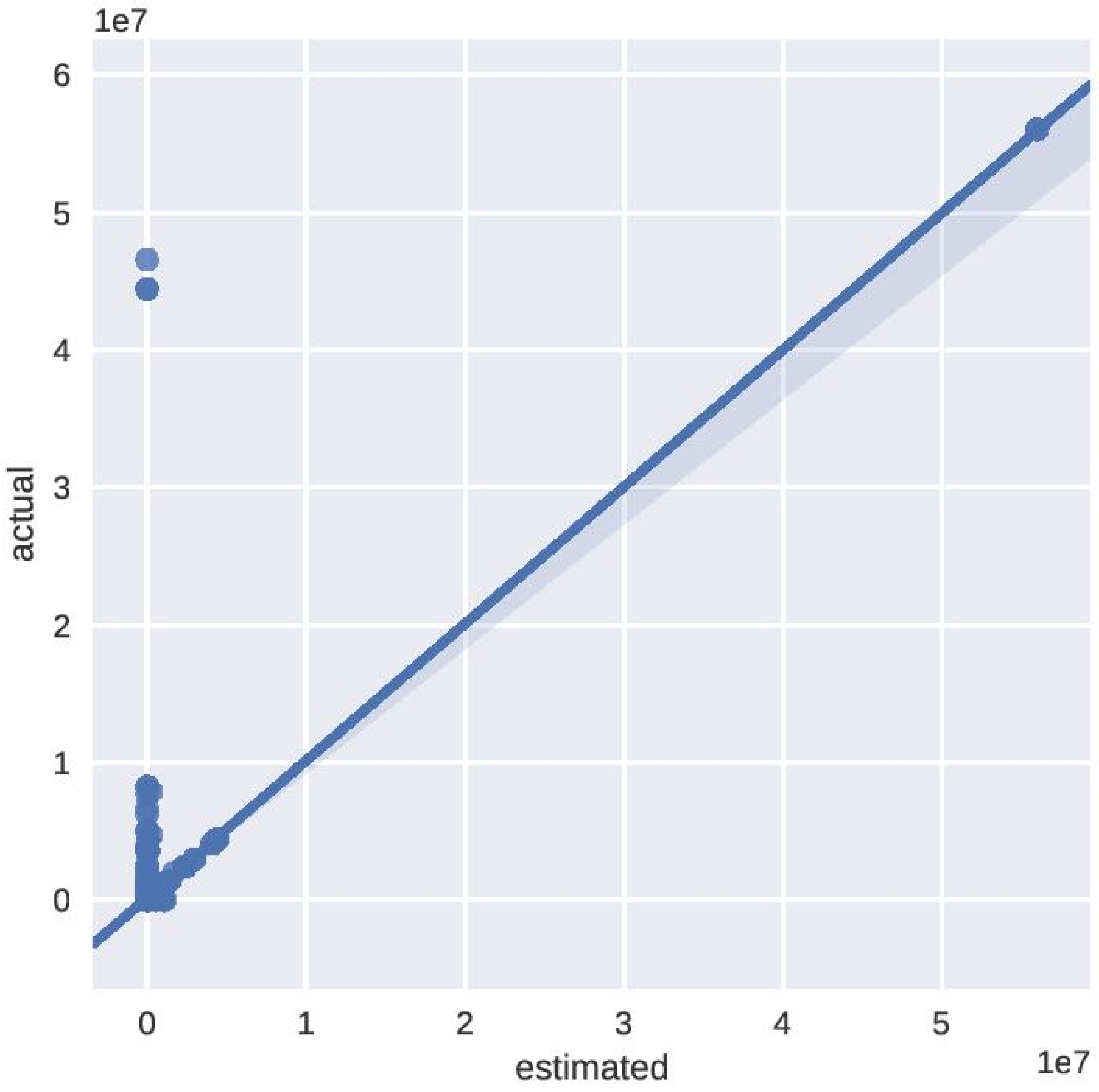


Figure 4.8: actual execution time and estimated execution time.

4.3 ANALYZING THE EFFECT OF BAD CARDINALITY ESTIMATION.

As i mentioned in chapter 2, PostgreSQL optimizer is a purely cost-based optimizer. Therefore, when the cost is misestimated, it leads to that PostgreSQL choose the plan which is not the best plan to execute. The example below shows the effect of estimation to choosing the plan of the PostgreSQL's optimizer :

- A, B are two relations; $card\,(A)_{est}$, $card\,(B)_{est}$ are respectively cardinality estimation of A,B; $card\,(A)_{real}$, $card\,(A)_{real}$ are respectively true cardinality of A, B.

$$card\,(A)_{est} = 1000,\ card\,(B)_{est} = 1$$
$$card\,(A)_{real} = 1000,\ card\,(B)_{real} = 1000$$

A nested loop join B	A merge join B
complexity = O(n*m)	complexity = O(n+m)
estimated cost = 1000 * 1 = 1000	estimated cost = 1000 + 1 = 1001
actual cost = 1000 * 1000 = 1000000	actual cost = 1000 + 1000 = 2000

Table 4.1: Operators sensitivity to estimation errors.

The underestimation of PostgreSQL's estimator in this example is 1000. Look at the table, we find that the estimated cost of nestedloop join is smaller than the estimated cost of merge join, so the planner's choice is nestedloop join instead of merge join whose actual cost is smaller about 1000 times than nestedloop join.

Figure 4.5, figure 4.6, figure 4.7, we find that most of the cardinality estimation of PostgreSQL estimator on JOB benchmark is underestimation. As a consequence of this underestimation, PostgreSQL's optimizer decides to introduce a nestedloop join because of a very low cardinality estimate, whereas in fact the true cardinality is larger, which lead PostgreSQL's optimizer to choose worse plan. To demonstrate the effect of this underestimation of PostgreSQL's estimator, i make the comparision between query performance between PostgreSQL with and without nestedloop join on JOB benchmarks.

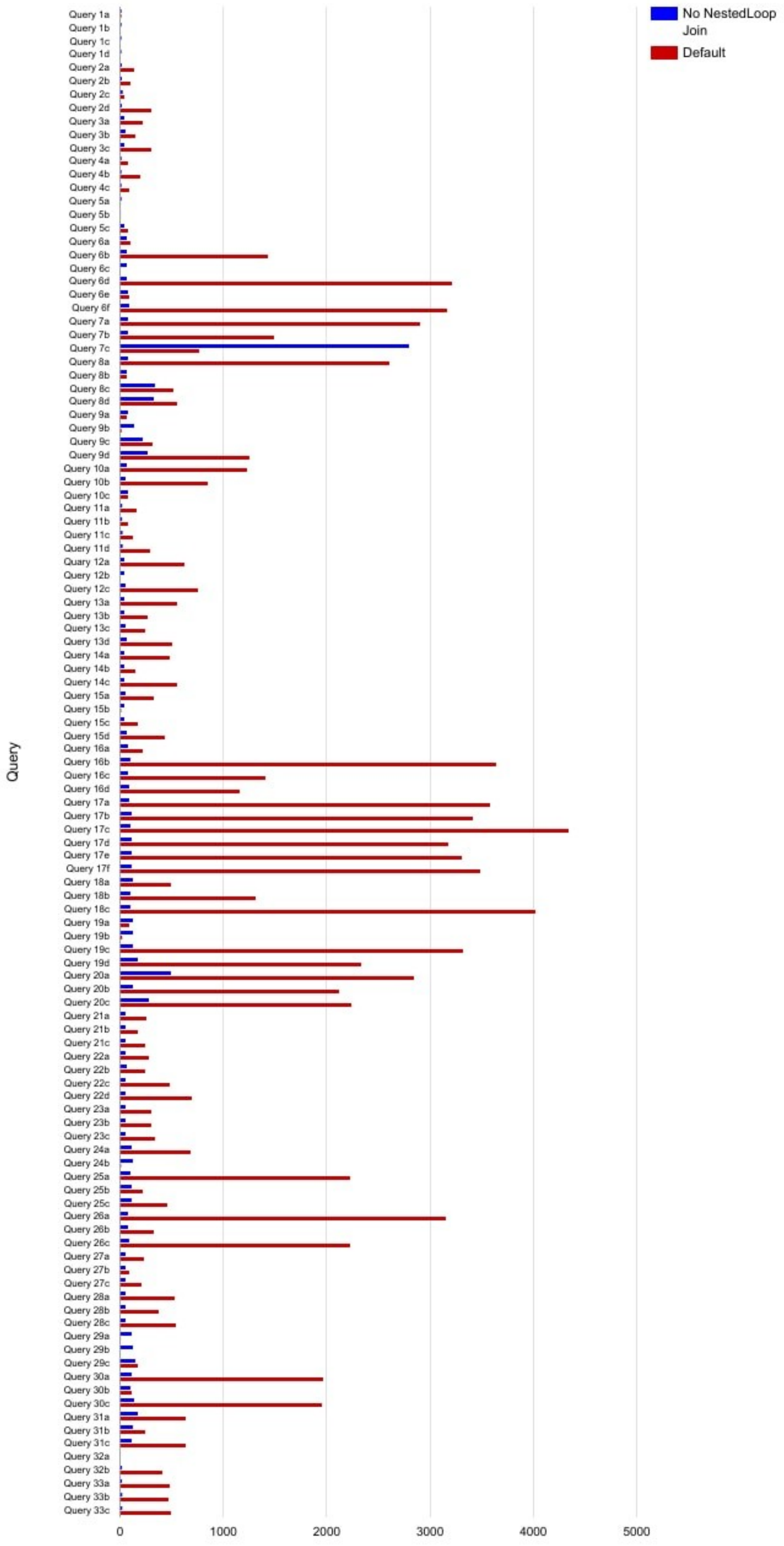


Figure 4.9: Query execution time (s) between PostgreSQL with and without nestedloop join.

As Firgure 4.9 shows, when rerunning all queries of JOB benchmarks without the risky nested-loop joins, we observed that the queries's execution time are improved significantly. For example, The execution time of query 6d is smaller 44 times after rerunning all queries of JOB benchmarks without the risky nested-loop joins, which increases from about 3213 seconds to nearly 73 seconds.

4.4 THE PUBLISHED SOLUTIONS.

Today, there is no perfect solution for query optimization problem, so a lot of paper on existing algorithms improvement are published. There are different lines of research in query optimization field.

- Use machine learning methods for cardinality estimation. For example:
  - In [3] the main point of their approach is using query execution statistics of previously queries to improve cardinality estimations. It is called adaptive cardinality estimation.
  - In [4] neural networks are used ti estunate the selectivity for user defined functions and data types. Nevertheless, this paper address the problem of estimation the selectivity for a single clause.
  - In [5] it is proposed to estimate node selectivity using inference in automatically constructed Bayesian networks.
- Use sampling-based approach [6, 7]. They do not improve query optimizer's cost estimator directly, but propose sampling-based ways to rectify standard query optimizer's errors if they are. Sampling-based approches are good on low-relational queries, but on queries with lots of joins they may have too large variance.
- Use multidimensional statistics for correct cardinality estimation [8, 9]. The most popular kind of such statistics are multidimensional histograms. It is the most popular way to improve standard cardinality estimation method in DBMS community.

Chapter 5

CONCLUSION AND FUTURE WORK

5.1 CONCLUSION

Remind that the objective of my work is to find the main factor that effect the most to query performance and to prove its effect by benchmarking it with JOB benchmark.

Through my thesis, I gain new technologies, the architecture of PostgreSQL as well as the way to benchmark a parameter in DBMS. I summarize the obtained results:

- Studying techniques, technologies such as: PostgreSQL and well-known bencmarks such as: TPC benchmarks, JOB benchmark.
- Studying the theory about the query optimization, the structure of benchmarks and implementation those.
- Consolidation of C programming, Python programming, Bash Script programming, SQL database.

5.2 FUTURE WORK

In short term, I will extend the proving of cardinality estimation's effect by creating an extension of PostgreSQL. This extension will make a comparision between PostgreSQL with default cardinality estimation, with true cardinality and with some published solutions.

In long term, I will research more solution that were publish and then create a new solution to reduce the negatively effect of cardinality estimation to query performance.

REFERENCES

[1] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter Boncz, Alfons Kemper, Thomas Neuman. *How Good Are Query Optimizers, Really?*. Leis et al., VLBD 2015.

[2] G. Moerkotte, T. Neumann, and G. Steidl. *Preventing bad plans by bounding the impact of cardinality estimation errors.* PVLDB, 2(1):982993, 2009.

[3] Oleg Ivanov and Sergey Bartunov. *Adaptive cardinality estimation.*

[4] M. S. Lakshmi and S. Zhou. *Selectivity estimation in extensible databases - a neural network approach.* In Proceedings of the 24rd International Conference on Very Large Data Bases, VLDB 98, pages 623627, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[5] L. Getoor, B. Taskar, and D. Koller. *Selectivity estimation using probabilistic models.* SIGMOD Rec., 30(2):461472, May 2001.

[6] W. Wu, J. F. Naughton, and H. Singh. *Sampling-based query reoptimization.* CoRR, abs/1601.05748, 2016.

[7] W. Wu, Y. Chi, S. Zhu, J. Tatemura, H. Hacigms, and J. F. Naughton. *Predicting query execution time: Are optimizer cost models really unusable?*. In Data Engineering (ICDE), 2013 IEEE 29th International Conference on, pages 10811092, April 2013.

[8] N. Bruno, S. Chaudhuri, and L. Gravano. *Stholes: A multidimensional workload-aware histogram, 2001.*

[9] D. Gunopulos, V. J. Tsotras, and C. Domeniconi. *Selectivity estimators for multidimensional range queries over real attributes.* The VLDB Journal,14:137154, 2005.

[10] Pat O'Neil, Betty O'Neil, Xuedong Chen. *Star Schema Benchmark.* Revision 3, June 5, 2009.

[11] Overview of PostgreSQL Internals.
`https://www.postgresql.org/docs/9.6/static/overview.html`

[12] Following a Select Statement Through Postgres Internals.
`http://patshaughnessy.net/2014/10/13/following-a-select-statement-through-postgres-internals`

[13] Cardinality Estimation Limitations.
`https://www.sqlpassion.at/archive/2017/05/01/cardinality-estimation-limitations`

[14] Introduction to Joins.
`https://blogs.msdn.microsoft.com/craigfr/2006/07/19/introduction-to-joins`

[15] LOOP, HASH and MERGE Join Types.
`http://www.madeiradata.com/loop-hash-and-merge-join-types`

[16] TPC Current Specifications.
`http://www.tpc.org/tpc_documents_current_versions/current_specifications.asp`