# Todo list

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 26, 2017

# Requirements for the thesis

**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)

**Dates** From 15/02/2017 to 15/06/2017

**Goals of the thesis**

- Identifying the factors that affect the performance of DBMS
- Showing the negative effects of these factors by benchmarking
- Proposing solutions to reduce their effect

**Main tasks**

- Identifying the factors that affect the performance of DBMS
- Identifying relevant benchmarks
- Benchmarking PostgreSQL with the benchmarks that were chosen
- Showing the impact of bad cardinality estimates on PostgreSQL performance

**Declaration of student**

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

Grenoble, 1 June 2017

Pham Tuan Hiep, author

**Attestation of the supervisor on the fulfillment of the requirements of the thesis**

Hanoi, 1 June 2017

Vu Tuyet Trinh, supervisor

# Abstract

Nowadays, the volume of data throughout the world has been exponentially increasing and has become extremely valuable. So it is challenging for us to manage and exploit it. Although many advanced database management systems have been developed to solve this challenge, they still struggle to control large volumes of data. Hence, it is neccessary for us to find 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, memory configuration, excessive indexing. It is hard to focus on all of these factors to solve the problem. Therefore, identifying the factor that has big impact on the performance of database management systems plays an important role.

This project focuses on identifying the factor which has big impact and showing its effect by benchmarking it in PostgreSQL and analyzing its effect. Besides, I also propose published solutions to reduce its effect.

In chapter 2, I introduce the steps that a query is processed. I identify and show the step that affects the query performance the most. And at the end of this chapter, I dig deeper into this step to identify the factor that has big impact on query performance.

In chapter 3, I identify common benchmarks for query performance in database management systems. And then i research them to find the best benchmarks for cardinality estimation in PostgreSQL.

In chapter 4, I benchmark cardinality estimation in PostgreSQL. Then I analyze the results to show the effect of cardinality estimation to query performance. Finally, I will propose some solutions that have been published to reduce its effect and improve the query performance.

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

# Acknowledgements

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

I would like to thank most sincerely Mr. Maxence Ahlouche, who patiently helped me get familiar with the new working environment, who gave me advice and always answer all questions as soon as I needed, and who guided me step by step during the process of implementing this project.

I am also thankful to Mrs. Christine Collet, Head of Heterogeneous and Adaptive Distributed dAta management Systems Team (HADAS) for her advice about my research and presentations.

I also wish to thank Mrs. Vu Tuyet Trinh for all her advice that helped me complete this project successfully.

I also thank all members of the HADAS team as well as the staff 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 Grenoble for their encouragements.

# Table of 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 generate unprecedented volumes of data. This data is stored in datasets that can reach petabytes or exabytes in size.

It is undoubted that these big volumes of data contain a significant value to our life. For example, desision making carried out by management teams and the creation of new business can be derived by these large amounts of information. However, to exploit these big volumes of data is not an easy task and there are many challenges ahead to achieve their great value. One of these challenges is the query performance of data management systems on big data sets.

Current data management systems have exposed their drawbacks, especially in query performance. When executing the queries with these DBMSs, there are long-running queries, which not only consumes system resources that lead the server and application to run slowly, but may also lead to table locking. So, query optimization becomes an important task.

There are many factors that affect to query optimization and it is hard to focus on all of them to improve the query performance. I identified that cardinality estimation has a big impact on query performance, and investigate this issue in particular.

### 1.1.2 Objective of my thesis

In this thesis, I have decided to focus primarily on cardinality estimation and showing its effect on 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.

- Identifying the factors in query optimization.
- Finding the best benchmarks for cardinality estimation.
- Benchmarking and analyzing the effect of cardinality estimation on query performance;
- Proposing published solutions.

# Chapter 2

# Query optimization

## 2.1 How is a query executed?

### 2.1.1 The main stages

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

**the parser** parses the query string;

**the rewriter** applies the rewrite rules (e.g. views);

**the optimizer** determines an efficient query plan;

**the executor** executes a query plan;

**the utility processor** processes DDL such as **CREATE TABLE**.

Figure 2.1 shows the architecture diagram of the query executor in PostgreSQL. In the first stage, the query string is parsed into a parse tree by the parser. After that, the query is then rewritten and checked semantically. In the next stage, PostgreSQL processes DDL (Data Definition Language) like **CREATE TABLE** or applies rewrite rules in the rewriter. The query tree is used to produce query plans and then the optimizer chooses the best plan based on a specific criterion, such as estimated cost in cost-based optimizers. Finally, the executor executes the plan chosen by the optimizer.

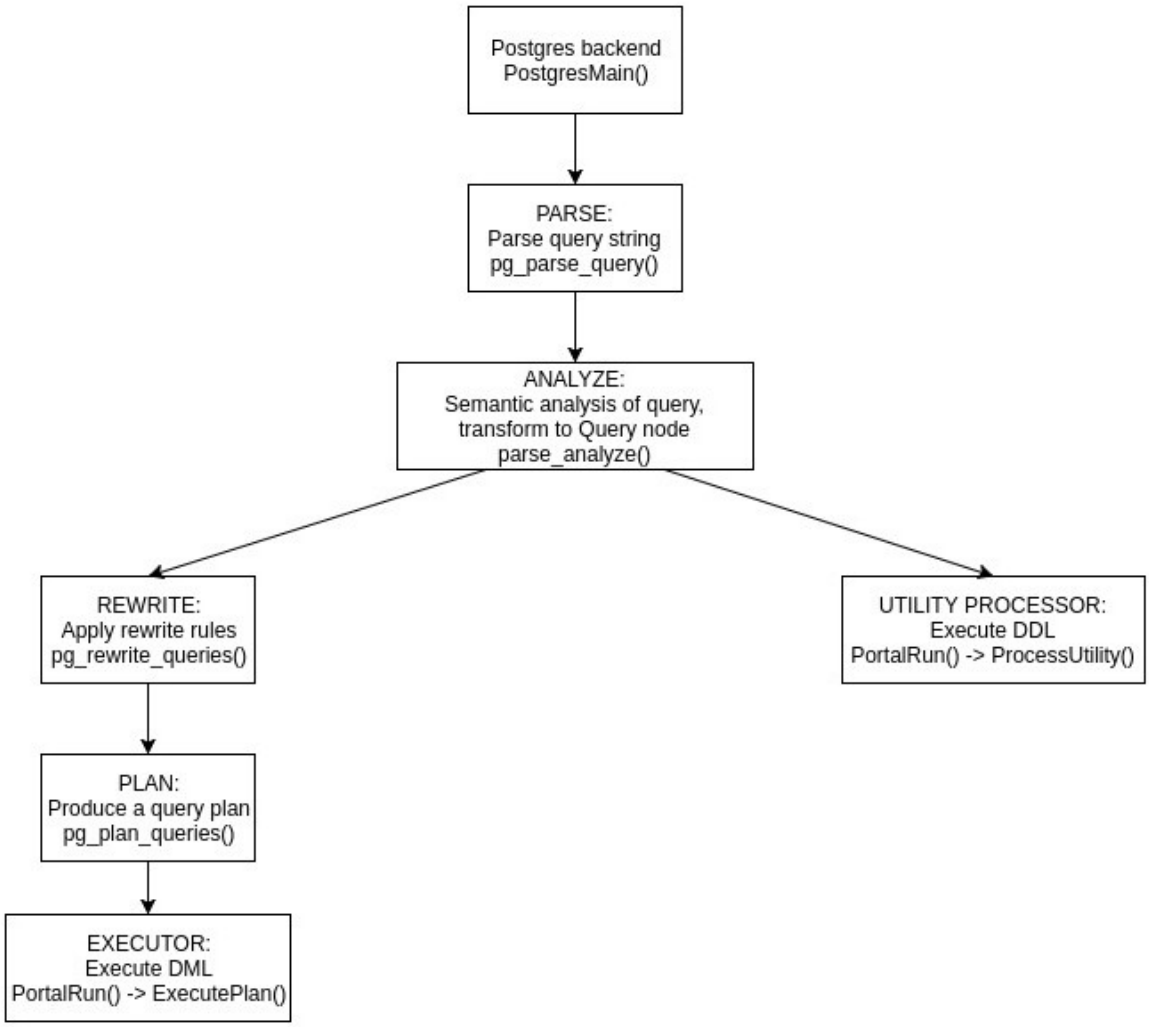


Figure 2.1: Architecture diagram of PostgreSQL

### 2.1.2 Parser

The aim of the parser [12] is to check the syntax of the query string. If the syntax of the query string is valid, the parser build a parse tree; whereas it returns an error. The syntax is defined in gram.y and scan.l which are built using the Unix tools yacc and lex.

PostgreSQL uses Bison, the parsing technology used by Ruby. A parser code based on a series of grammar rules is generated by Bison during the PostgreSQL C build process. The parser finds a corresponding pattern or syntax in the query string by checking each grammar rule, and then inserts a new C memory structure into the parse tree data structure.

Figure 2.2 is an example of the below query's parse tree:

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

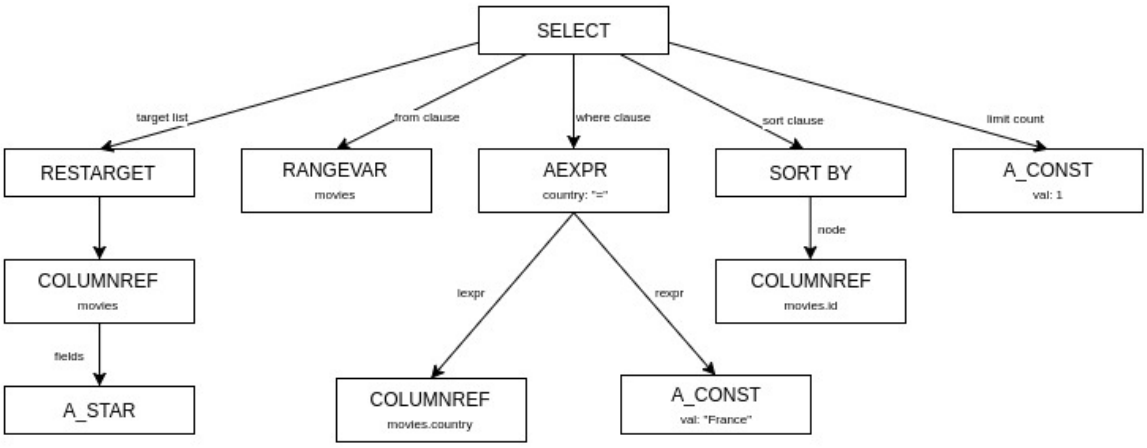


Figure 2.2: An example about parse tree.

### 2.1.3 Rewriter

After generating a parse tree, PostgreSQL converts this parse tree into another tree called as the query tree. A query tree contains parts [11]:

- The command type: it tells which command of the query string, such as SELECT, INSERT, UPDATE, DELETE.
- The range table: it contains the relations that are used in the query. For example, in a SELECT statement, the range table is the list of relations given after the FROM key word.
- The result relation: it is an index into the range table, which identifies the relation where the results of the query go. In INSERT, UPDATE and DELETE statements, the result relation is the relation where the changes are to take effect. There is not a result in SELECT queries.
- The target list: it contains expressions that define the result of the query. To be more specific, in SELECT statement, the target list is a list of expression between the key words SELECT and FROM. * is a special expression, it is just an abbreviation for all the column names of a relation.
- The qualification: it is an expression much like one of those contained in the target list entries. Its result value is a boolean, which tells whether the operation (INSERT, UPDATE, DELETE, or SELECT) for the final result row should be executed or not. In SQL statement, the qualification is the WHERE clause.
- The join tree: it shows the structure of the FROM clause.
- The others: the other parts of the query tree like the ORDER BY clause

Figure 2.3 is an example of the below query's query tree:

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

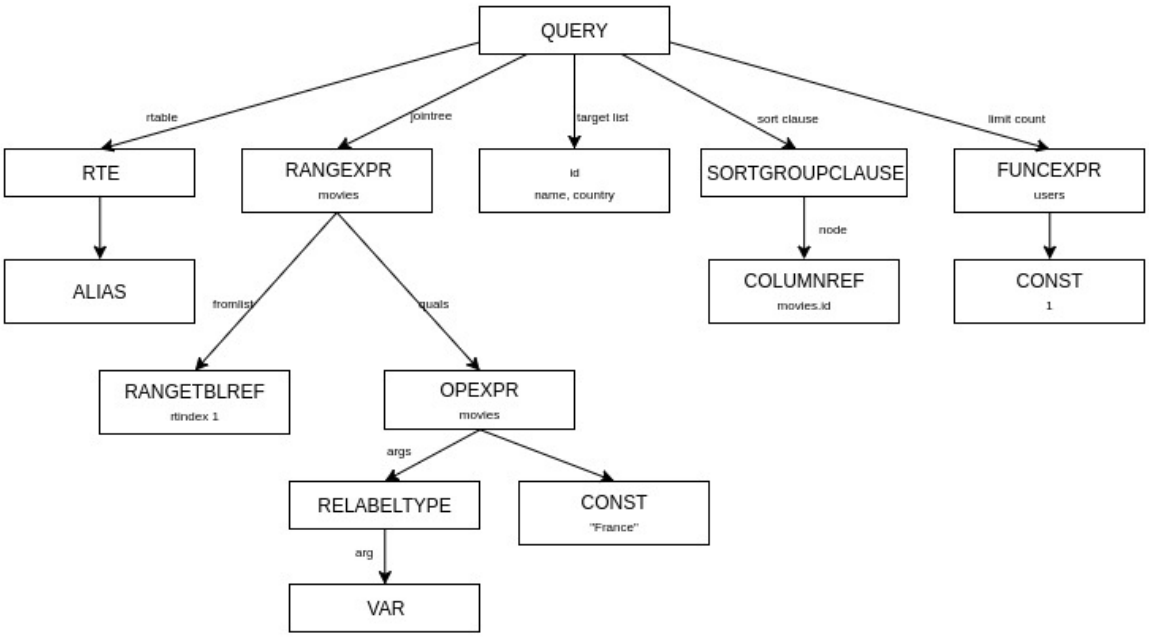


Figure 2.3: An example about query tree.

### 2.1.4 Planner/optimizer

Planner's main task is to create an optimal execution plan. There are many ways which produce the same results to execute a query tree of a SQL plan. The query optimizer will examine each these execution plans and finally choose the best execution plan based-on a specific criterion, such as estimated cost in cost-based optimizers (PostgreSQL's optimizer) .In PostgreSQL, after the cheapest path is determined, a full-fledged plan tree is built to pass to the executor.

Figure 2.4 is the below query's full-fledged plan tree:

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

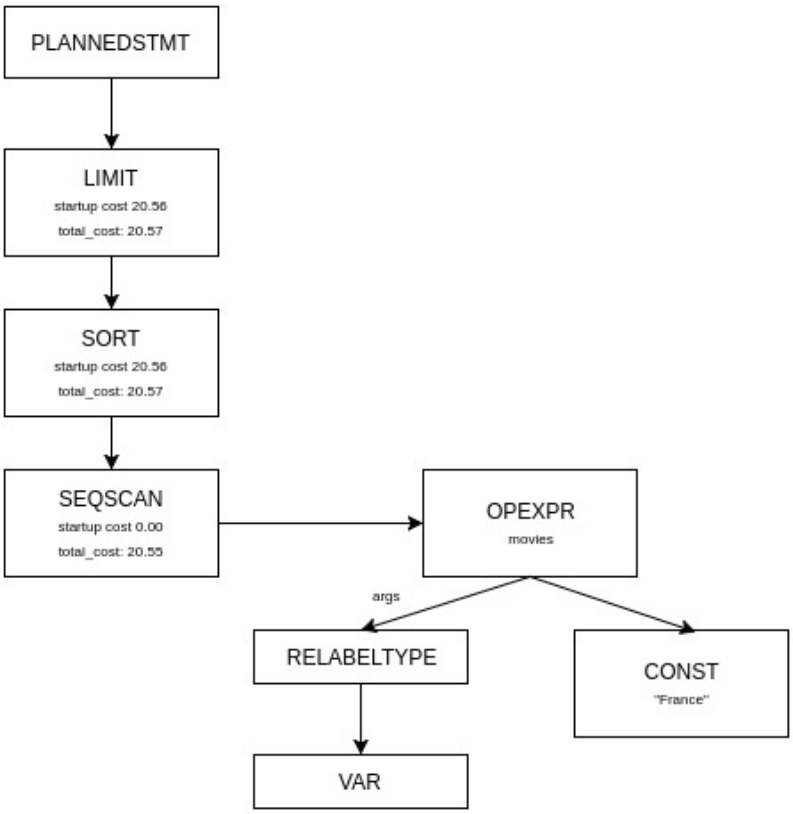


Figure 2.4: An example about full-fledged plan tree.

### 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.

When executing complex queries which involve many levels of plan nodes, 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 are two factors that impact on the decision of the optimizer: the amount of time spent figuring out the best query plan and the quality of the choice. 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.

### 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. Cardinality estimation has a bigger impact on query performance than cost model [3].

### 2.2.3 PostgreSQL's cardinality estimator and its limitations

#### PostgreSQL's cardinality estimator

Postgresql's optimizer follows the traditional textbook architecture. PostgreSQL's estimator uses histograms (quantile statistic), most common values with their frequences and domain cardinalities (distinct value counts) to estimate cardinalities of base tables. Histograms can not be applied to estimate cardinalities of complex predicates. To combine conjenctive predicates for the same table, independence assumption is used and PostgreSQL multiplies the selectivites of the individual selectivity estimates.

There are some assumptions [1] in PostgreSQL's estimator:

**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

**Principle 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 columns can lead to PostgreSQL estimator's bad cardinality estimates. To be more specific, I use the example below to show the impact of the correlation between relations on 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. But as i mentioned above, PostgreSQL's optimizer assumes uniformity, independence, inclusion, so it looks at each search predicate indepedently.

  In the first step the estimator makes the cardinality selectivity for the predicate **Company = 'Toyota'**. After that, the estimator produces a cardinality selectivity for the predicate **Brand = 'Civic'**. Finally, the estimator computes the cardinality selectivity of the query by multiplying the cardinality selectivities . For example, if 0.4, 0.4 are respectively the cardinality selectivity of the first and second predicate, 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

# IDENTIFYING BENCHMARKS

## 3.1 RESEARCHING BENCHMARKS

### 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. This benchmark illustrates decision support systems that

- Base on big volumes of data;
- Execute complex queries;
- Solve critical business questions.

TPC-H queries :

- Answer real-world business questions;
- Simulate generated ad-hoc queries
- Are far more complex than most OLTP transactions;
- Include a rich breadth of operators and selectivity constraints;
- Generate on the database server component of the system under test;
- Are executed against a database complying to specific population and specific scales. There are fixed scale factors: 1GB , 10GB, 30GB, 100GB, 300GB, 1000GB, 3000GB, 10000GB, 30000GB, 100000GB.
- Are implemented with constraints derived from staying closely synchronized with an on-line production database;

- Contain few joins, between 0 to 4 joins each query. For example, 1.sql doesn't have any join, 2.sql contains 3 joins. There are not columns in realtions.

**TPC-DS**

TPC-DS [16] models several generally applicable aspects of a decision support system (DSS). It includes:

- 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.

TPC-DS illustrates decision support systems that:

- Bases on big volumes of data;
- Answers real-world business questions;
- Execute complex queries;
- 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.

User queries contain four broad classes of queries:

- Reporting queries: The "reporting" nature of a DSS system is captured by these queries Their aim is to answer well-known, pre-defined questions about the financial and operational health of a business. They tend to be static.
- Ad hoc queries: Whereas reporting queries, these queries capture the dynamic of a DSS system. Their aim is to answer immediate and specific business questions.
- Iterative Online Analytical Processing (OLAP) queries: OLAP queries can discover new and meaningful relationships and trends by exploring and analyzing business data. A scenario-based user session in which a sequence of queries is submited is used to distinguish they with Ad hoc queries.
- Data mining queries: Their aim is the prediction of future trends and behaviors; besides, they are also used to allow business to make proactive, knowledge-driven decisions. They consists of joins and large aggregation that return large data result sets for possible extration.

User queries in TPC-DS are executed against a database to specific population and specific scales. The offical scale factors are 1TB, 3TB, 10TB, 30TB, 30TB and 100TB. 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 simulates 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. TPC-E is characterized by:

- The simultaneous execution of multiple transaction types of various 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 size;
- Contention on data access and update;
- Moderate system and application execution time.

There are limitations in this benchmark:

- There are few joins in each query and the queries totally focus on the transactions. So, it is not used to benchmark the parameters in DBMSs.
- 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.

### 3.1.2 Star Schema Benchmark

The Star Schema Benchmar (SSB) [10] was designed to test star schema optimization. There are four query flights, four dimensions in SSB. The SSB is significantly based on the TPC-H benchmark but is improved by implementing a traditional pure star-schema and allowing column and table compression. Although model queries in SSB benchmar are based on the TPC-H query set, they are modified to vary the selectivity.

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.

### 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 and consists 21 tables. 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 . There are many correlated columns in relations of this data set. Its size reaches currently to around 20 Gigabytesafter being transformed into relational databases.
- JOB queries: They are based on the IMDB data set and are constructed as analytical SQL queries. 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 3.1 is the typical query grapth 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.

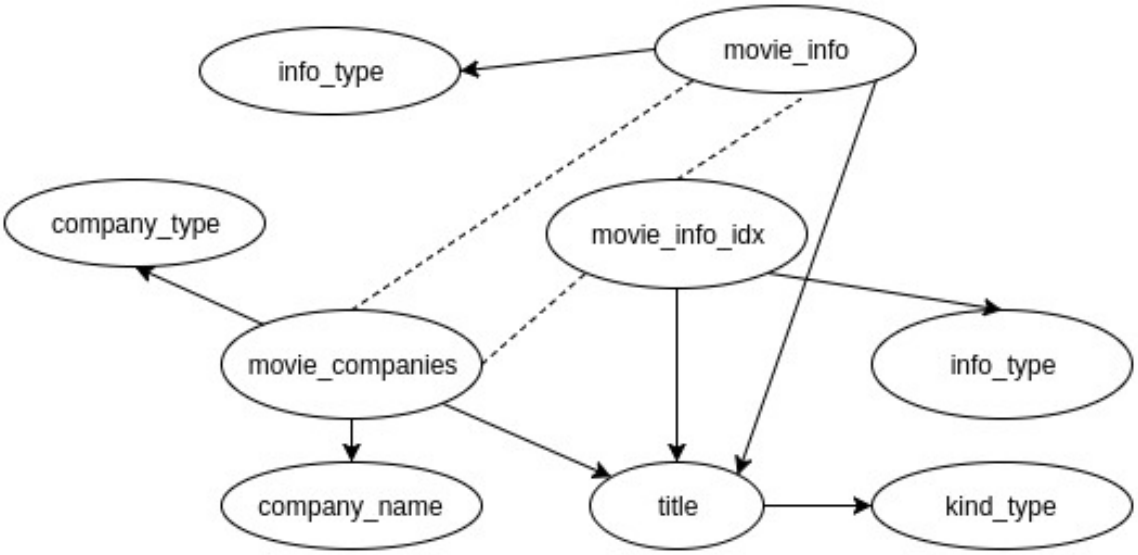


Figure 3.1: Typical query graph of JOB workload [1].

## 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 the data generators use the assumptions (uniformity, independence, principle of inclusion). These assumptions are the reasons leading to the limitaions of cardinality estimator of PostgresSQL. However, There

are full of correlations and non-uniform data distributions in current real-world data sets, which makes cardinality estimation much harder. Therefore, I don't use these benchmarks for benchmarking cardinality estimation in PostgreSQL.

Besides, JOB benchmark solved the limitation of the benchmarks above. IMDB data set is real-world data set with full of correlations and non-uniform data distributions and its queries contain many joins, which makes cardinality estimator of PostgreSQL encounter hardships. So it is very suitable to be used for benchmarking cardinality estimation in PostgreSQL.

# Chapter 4

# BENCHMARKING CARDINALITY ESTIMATION ON POSTGRESQL

## 4.1 BASIC THEORIES AND IMPORTANT CONCEPTS

### 4.1.1 Q-error

#### The Definition of Q-error

In [2], R and A are respectively a relation and its attributes, and $\{x_1, ..., x_m\} = \Pi_A(R)$ is the set of distinct values of A. The frequency density is a set of pairs $(x_i, f_i)$ with $f_i = |\sigma_{A=x_i}(R)|$, $1 \leq i \leq m$

Define $F$ as a function to approximate this set of paris; $F(x_i) =: F_i$ gives the estimate $F_i$ of $f_i$. And $F$ returns value in a range between 0 and 1.

Typically, norms are used to define the error. $\vec{b} = (f_1, ..., f_m)^T \in \mathbb{R}^m$ organizes the correct values 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

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

for z = $(z_1, ..., z_m)^T \in \mathbb{R}^m$. While the $l_2$ error does not give bound on estimates, $l_\infty$ 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:

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

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:

$$\|z\|_Q = \max_{i=1}^{m} \|z_i\|_Q$$

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

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

To measure the quality of base table cardinality estimates, q-error is the factor by which an estimate differs from the true cardinality. For example, if the true cardinality of an expression is 1000, the estimates of 10 or 100000 both have a q-error of 100.

### 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

It looks for two rows that 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.

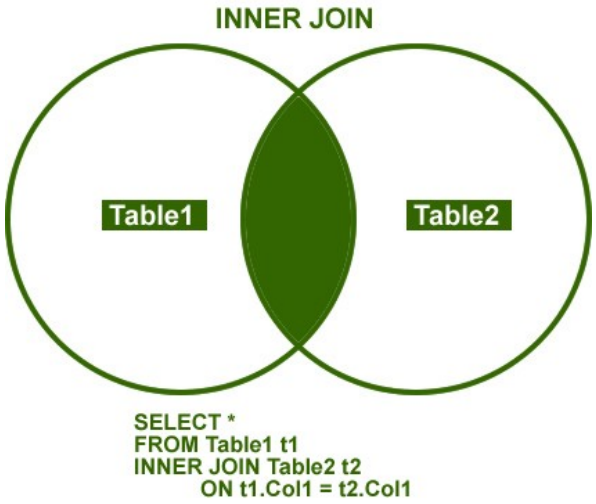


Figure 4.1: An example about inner join [17].

**Outer join**

- Left outer join: Its result contains all rows of the ”left” table, even if the join-condition does not find any matching row in the ”right” table. Figure 4.2 makes more specific about that.

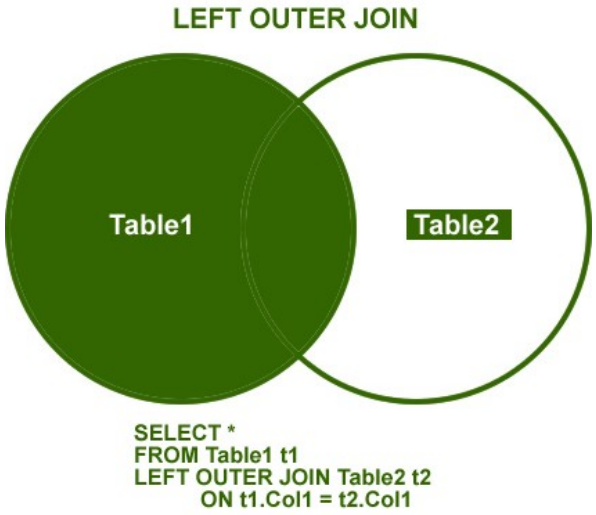


Figure 4.2: An example about left outer join [17].

- Right outer join: In contrast with left outer join, the result of right outer join contains all rows of the ”right” table, even if the join-condition does not find any matching row in the ”left” table. Figure 4.3 shows that:

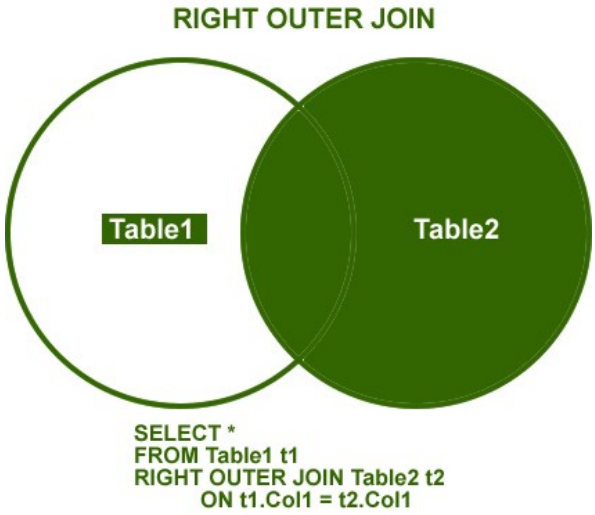


Figure 4.3: An example about right outer join [17].

- Full outer join: A full outer join combine left outer join and right outer join. So, the result of full outer join contains all rows of both tables. With the lack of a matching row, the result set will have NULL values for every column of the table.

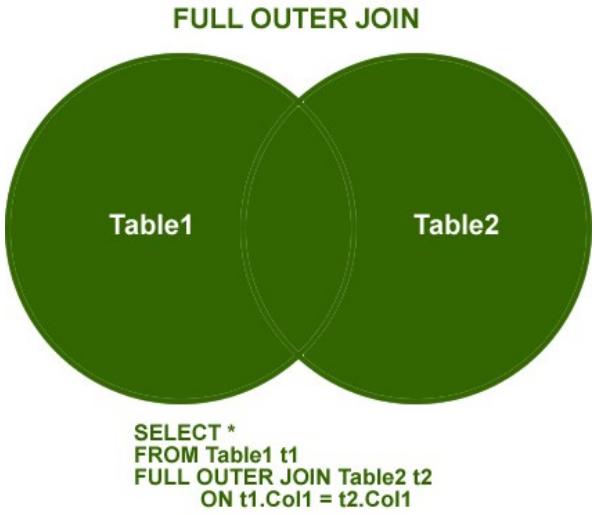


Figure 4.4: An example about full outer join [17].

**Cross join**

If where clause is not used, the result set of the cross join is the number of rows in the first table multiplied by the number of rows in the second table; whereas it functions like an INNER JOIN.

### 4.1.3 Physical join operators

Physical join operators are used by DBMSs internally to implement these logical join operators, DBMSs use three different physical operators. Three physical join operators are respectively Nested Loops Join, Hash Join, Merged Join

#### Nested Loops Join

In Nested Loops join [15], for each iteration of the outer loop all the iterations of the inner loop are executed. To be more specific, for each row of the outer table, all rows from the inner table are examined with this row. If matched, the row from the inner table is included in the result-set; whereas it is ignored. The next row from the outer table is picked up and the same process is repeated.

In terms of complexity, If 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). 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).

#### Merge Join

In Merge join [15], the inputs are required to be sorted on join keys/merge columns. As a consequence of being sorted, this join operator reads a row from one input and then compares it with the row of another input. If the rows match, that matched row is included in result set and then it reads the next row from the input table; whereas the lesser of the two rows is ignored and the process continues this way until all rows have been processed.

In terms of complexity, it is assumed that N, M are the row numbers of tables, 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 not choose the Merge join type due to the the cost of pre-sorting process.

#### Hash join

There are two phases in a hash join [15]. They are the build phase and the probe phase; so it has two inputs called build input and proble input. It is considered that the smaller table is build input and the other one is probe input.

In the build phase, join keys of all the rows from the build table are scanned and then hashes are generated and stored in hash table. In the probe phase, join keys of each row of the probe table are scanned and hashes are generated again, then compared against the corresponding hash table for a match.

Hash table requires memory resource to store and a hash function requires significant amount of CPU cycles to generate hashes. To achieve high performance, the PostgreSQL optimizer may parallelize a Hash join to scale better than any other join.

In term of complexity, it is assumed that $h_c$ 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 comparison between actual execution time and estimated execution time of queries.

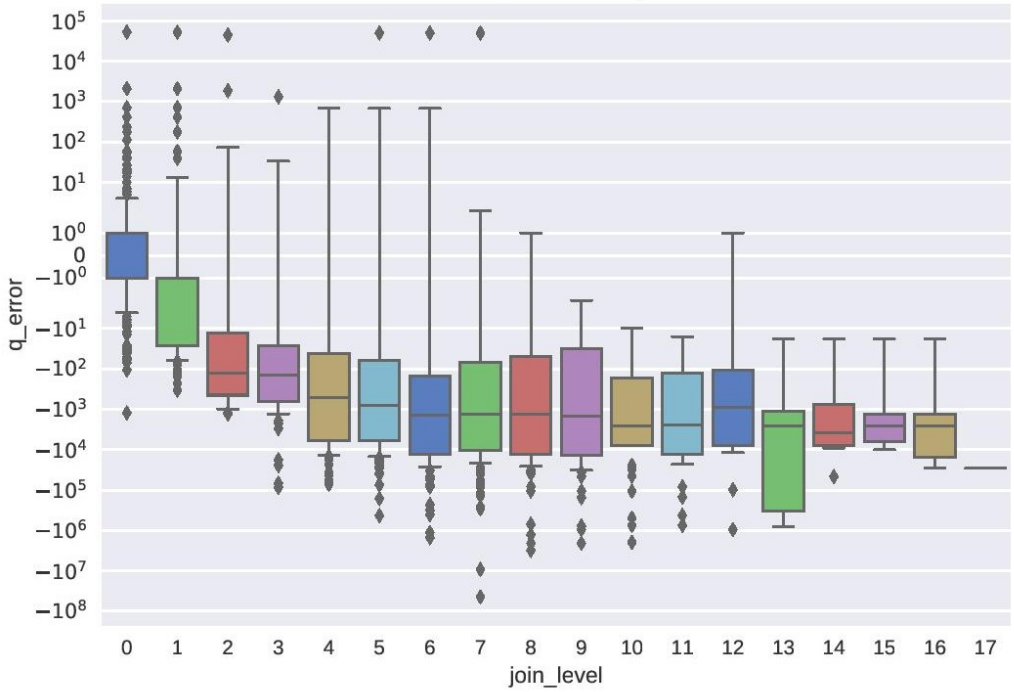


Figure 4.5: Q-error of plan node and its join level.

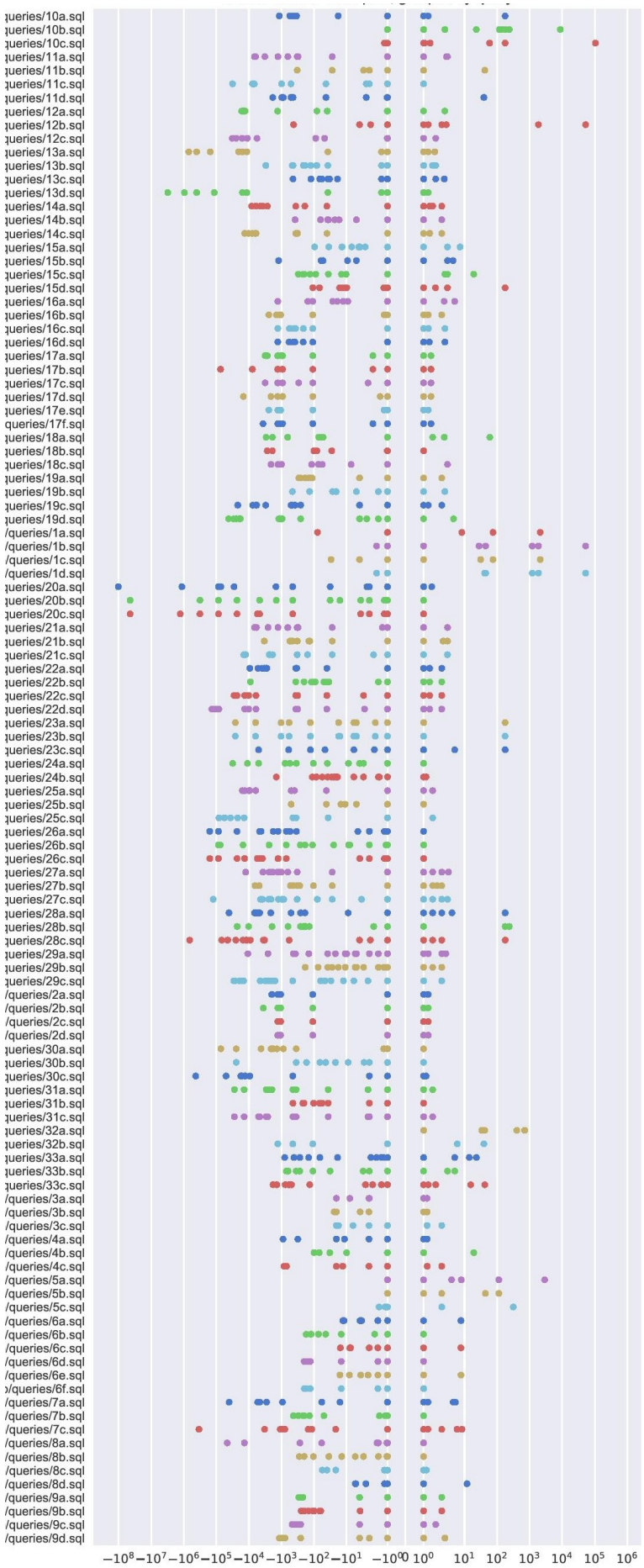


Figure 4.6: Q-error of plan node per query.

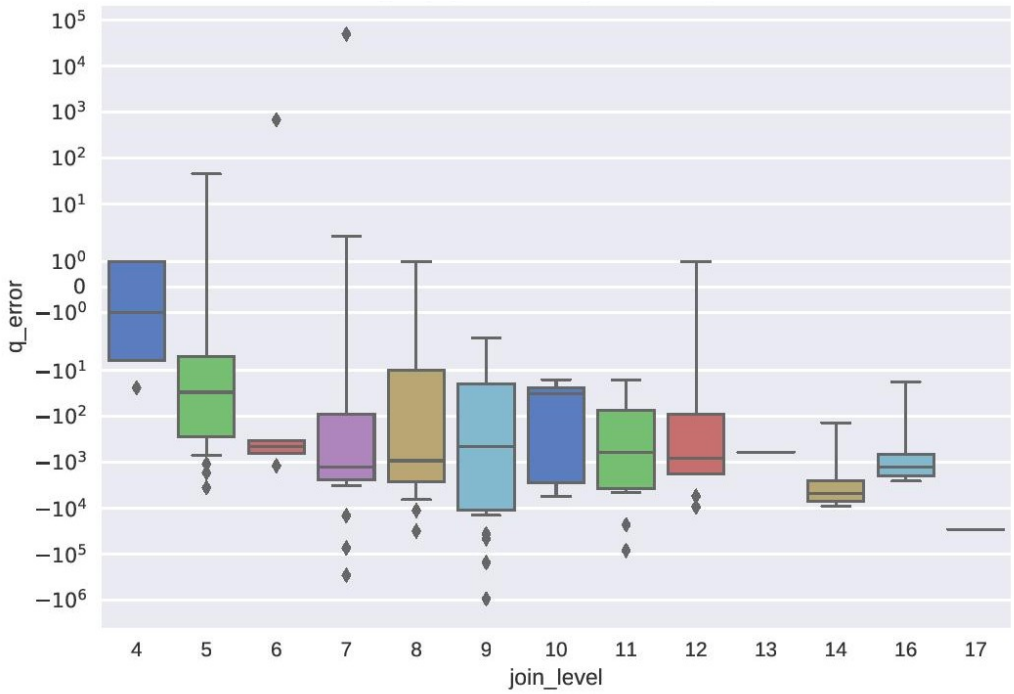


Figure 4.7: Q-error of plan node and join tree depth.

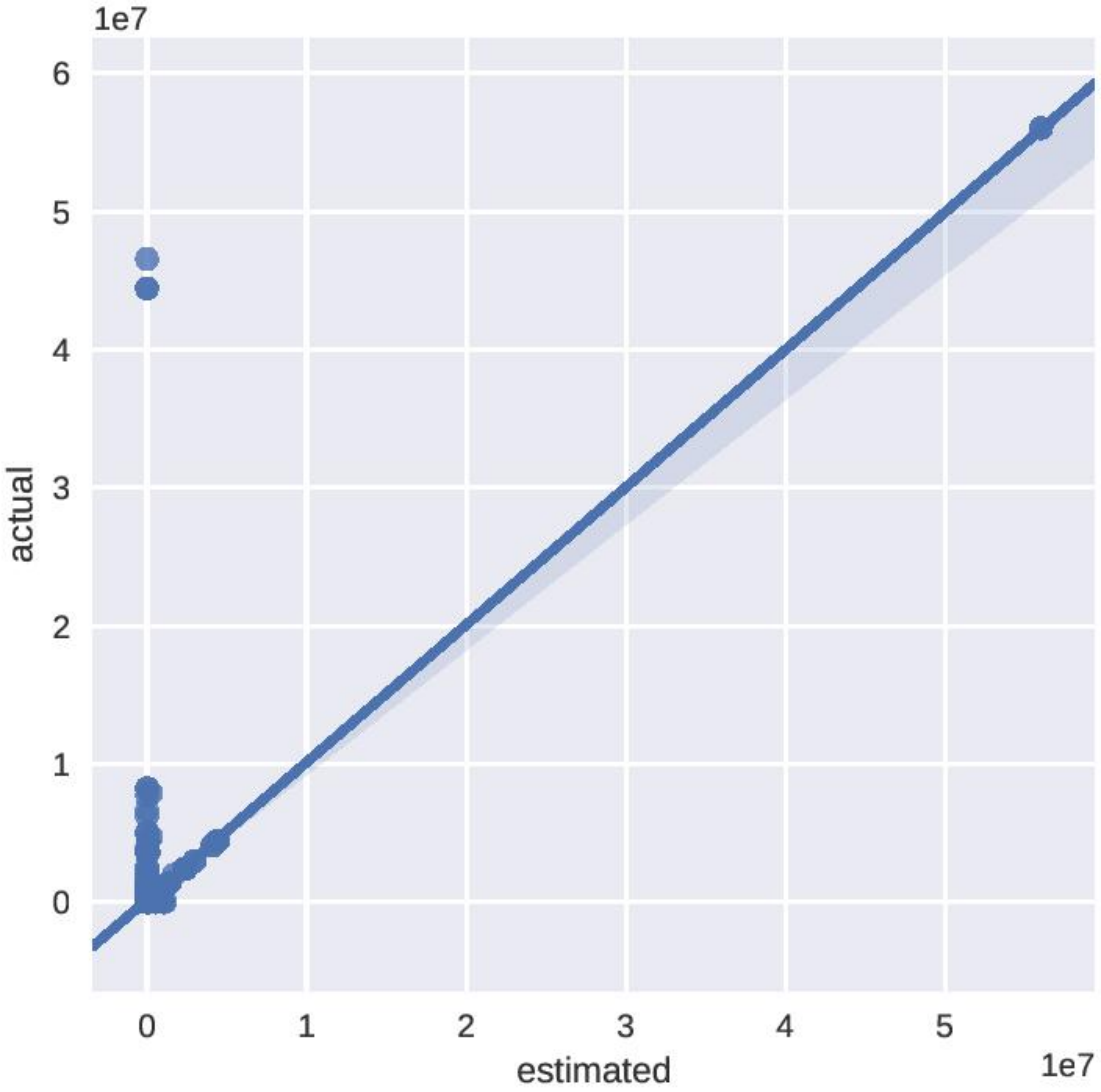


Figure 4.8: actual execution time and estimated execution time.

## 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 doesn't choose the plan with smallest cost. The example below shows the effect of estimation to which plan the PostgreSQL's optimizer chooses :

- 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 queries 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. Therefore, query performance is decreased remarkably. To demonstrate this, i make the comparision between query performance between PostgreSQL with and without nestedloop join on JOB benchmarks.

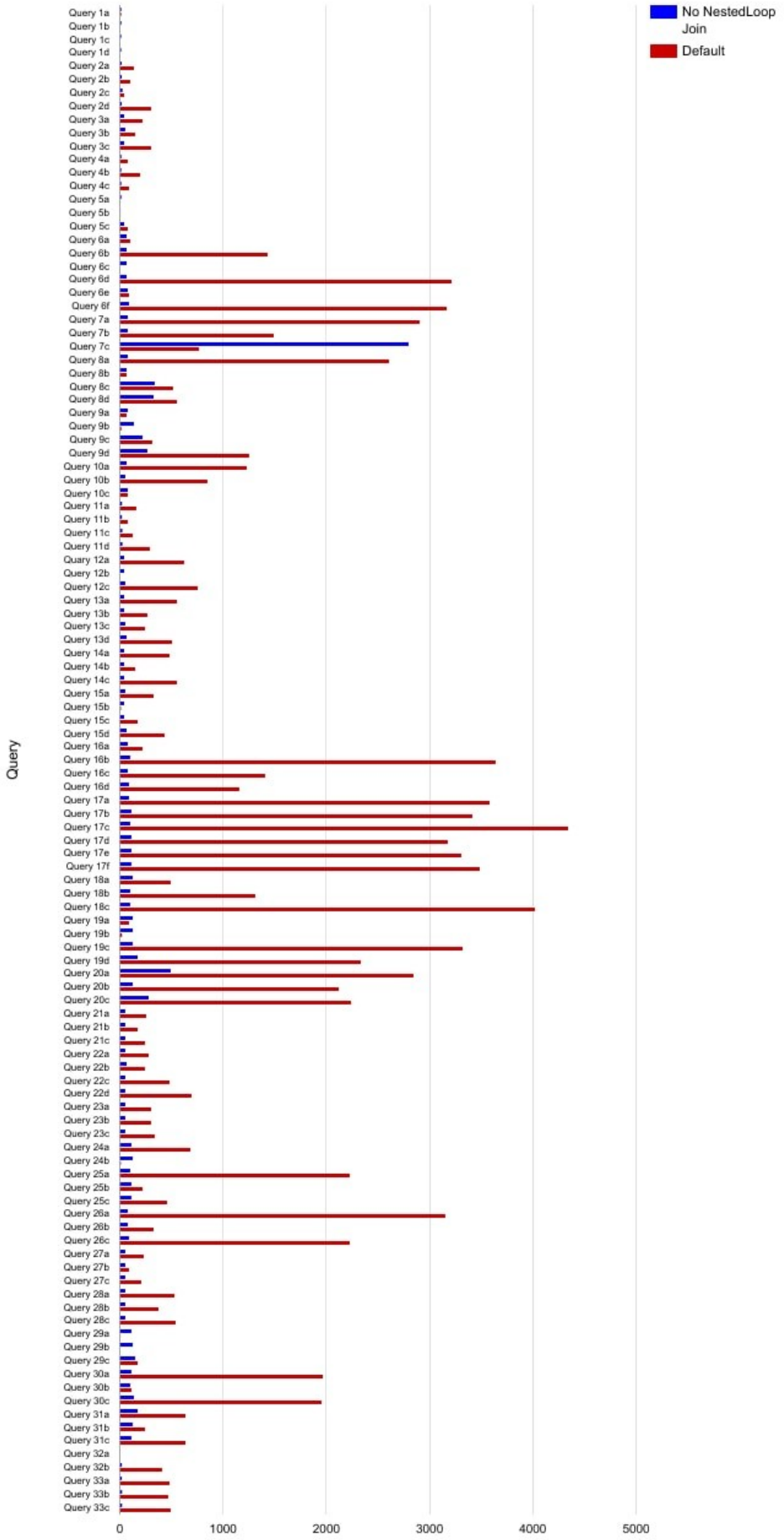


Figure 4.9: Query execution time (s) between PostgreSQL with and without nestedloop join.

As Figure 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 declines from about 3213 seconds to nearly 73 seconds.

## 4.4 THE PUBLISHED SOLUTIONS

There have been many solution published to improve the query performance. There are some lines of these solutions.

- 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 to estimate 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]. Instead of using single histogram as in DBMSs, multidimensional histograms are used. 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 identify the factor that has big impact on query performance and to show its effect by benchmarking it with JOB benchmark.

Through my thesis, I studied 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 implementing those.
- Consolidation of C programming, Python programming, Bash Script programming, SQL database.

## 5.2 FUTURE WORK

In short term, I will extend the showing 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 solutions that were published and then create a new solution to reduce the effect of bad 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

[17] JOINing data in R using data.table.
https://rstudio-pubs-static.s3.amazonaws.com/52230_5ae0d25125b544ca