



Universiteit  
Leiden



Centrum Wiskunde & Informatica

# **Automated Detection of Performance Regression in New Versions of PostgreSQL.**

George-Alexander Anastasiou

Academic Supervisors: Stefan Manegold, Arno Knobbe

Centrum Wiskunde & Informatica Supervisors: Hannes Mühleisen,  
Mark Raasveldt

# Contents:

1 Introduction.....	5
2 Related Work.....	8
2.1 TPC-H Benchmarking.....	8
2.2 PostGreSQL and Benchmarking.....	11
2.3 Python with combination of Database Management systems.....	13
3 Implementation and Evaluation .....	14
3.1 - Analyzing the first tool and the results.....	14
3.2 - Analyzing the second tool and the results.....	30
4 Implementing Automation in Software Quality Assurance .....	33
5 Conclusions and Future work .....	38
6 Bibliography .....	40
7 Appendices .....	41

## **Dedications**

This thesis is dedicated to :

Mark Raasveldt, who has been a beacon of light in the darkness and complexity of database systems.

Michai Varga, who was always ready to unsheathe his Python sword, and help me tackle programming problems.

Stefan Manegold, for his trust, guidance, and for accepting me into CWI.

## **Abstract:**

Information Technology is an important structural part of organizations. Companies invest large amounts of capital and man power into their IT infrastructure. The majority of those investments are spent on maintaining and updating existing infrastructure. It is of vital importance to update software packages in order to fix security leaks and bugs in the software. Security leaks can cause immense damage to a firm and its business, and a loss of trust by clients if their private information is leaked.

However, updating software packages can introduce new problems as well. Whenever a software package is updated, the possibility of a regression occurring exists. For this reason, companies hire large Quality Assurance (QA) departments that test before every upgrade whether or not such regressions occur in the software. This costs companies a large amount of manpower and money.

A regression can occur in several forms. The most common form of a regression is that a new bug is introduced, where a workflow of a program that used to work correctly now no longer works correctly. For this type of regression, many types of automated tests and testing frameworks have been introduced. However, more subtle regressions also exist. In particular, one form of regression that is currently not tested for automation is the performance regression. This type of regression does not introduce actual faults in the program, but rather causes existing workflows to perform less efficiently than they did before the software update. This can cause severe problems in the business workflow, as severe performance regressions can slow down the operational system or introduce latency to the clients depending on where it occurs.

In this work, we focus on automatically detecting performance regressions to assist the quality assurance departments of businesses and help prevent severe performance regressions from occurring. In particular, we focus on Relational Database Management Systems (RDBMSs) which are used by the majority of businesses and where low latency is of vital importance to the business process.



# Chapter 1

## Introduction

The effectiveness of data management within firms has become more important due to the large increases in storage information capacity and the increasing amount of data generated by the firms. Data management includes many topics, such as data security, sharing, governance and warehousing. Data management is crucial regarding to how many businesses operate. To exploit the data a firm has, Relational Database Management Systems (RDBMS) are implemented and are used for maintaining databases, based on the relation model of data. Usually, all relationship database systems use the Structure Query Language (SQL) to query the data. SQL was particularly designed to address structured data that have relations between the different entities of the data. [1]

The development and maintenance of Database systems became imperative for the all kind of business. For example, Amazon uses an RDBMS in order to keep track of their customer base and inventory [13]. The maintenance of business operations and processes are mainly managed by the information systems a company has applied, in order to seize control of opportunities in the market regarding products or supplying services. Due to this dependence on information a firm has, the performance of those systems is of outmost importance. More stable systems that perform better have proved to eventually assist strategic decision making notable faster due to business intelligence, and avoiding opportunity costs. Because of this special relationship of a firm with its data, regression in systems can be associated with reducing the speed of processes and customer dissatisfaction. Eventually, regression in Information Systems can affect the decision makers and create multiple opportunity costs. An example of the impact a regression can create in the business performance is if a database request to answer a client search requires ten times more time to return results. This can result to an unresponsive website. An unresponsive website creates customer dissatisfaction. 88% of online consumers are less likely to return to a site after a bad experience and 47% of the users expect a maximum 2 seconds loading time for a website. Websites that are loading slow or are unresponsive cost retailers around 2.6 billion\$ in lost sales [2]. So, a performance regression in the RDBMS of a firm can have dire consequences.

Companies invest in QA (Quality assurance) in order to ensure quality and steady performance for their software products and existing infrastructure. Every new version of software must be tested before implemented for bugs and performance regression in different scenarios. Regarding performance regression in a software system, the automated tools that exist are few and usually not compatible with open source systems. The automation of the performance regression detection in an update of a software system can lead to a reduction in multiple costs and man power.

Regarding the database industry and performance testing, in order to help customers decide which database system performs better in a particular scenario, the Transaction Processing Performance Council was created. The TPC benchmark was standardized, starting with obsolete workloads for the present, like TPC-A, that kept evolving to the industry regulations and needs. The bench marking process however, can be difficult and time consuming. Implementing the TPC benchmark and studying the performance of a system by using a particular database vendor in comparison with another vendor or

another system, can lead to wrong conclusions and hasty mistakes. There are also many pitfalls regarding database performance comparison, such as non-reproducibility or failure to optimize the database for bench marking [3]. For serving the purposes of this thesis, the benchmark TPC-H will be applied. TPC-H is a decision support benchmark which consists of business oriented ad-hoc queries and the data generated are particularly chosen to have industry relevance. Using the TPC-H benchmark, users can come easier to a decision on which system they should implement, regarding their own hardware or particular scenario.

## **Purpose of this thesis**

In this thesis, due to today's high demand of high quality Database Systems, I propose the automation of the bench marking process, in order to detect performance regression between the current version of the system, and the next software update. This thesis is focused in the database system of PostgreSQL, an open source database that can be downloaded, installed and updated through open source version control systems, like Github.

One of the factors that performance regression can occur is the code of the new version of the system. More specifically, when a new commit is being pulled through GitHub, some code files are being deleted, and some new files are being added. In this thesis, I try not only to detect performance regression automatically, but also to specify which change of the code implemented, actually led to a regression.

By executing the tools, users can detect if the new version causes performance regression regarding the time of query execution, using an industry relevant data set, and choosing the scale factor for which is closer to their needs in order to perform those tests. Developers can recognize which particular file has led to a regression and the precise regression in time, in comparison with the previous version of the system.

For the implementation, I use Python for the scripting language, a general purpose scripting language that is widely used and accepted by data scientists, software engineers and developers, due to its large existing code base for data analysis, graph plotting and mathematics. Those combined factions make Python an ideal language for performing efficient operations in a database without forfeiting the advantages of a scripting language.

## **Thesis Outline**

The thesis is structured as follows. In Chapter 2, I introduce the related work that has been implemented by the TPC Benchmarking, the most relevant research in Relational Database Management Systems and their current ranking, the related research that has been done in PostgreSQL Benchmarking and the strengths of python as a scripting language for performing data analysis operations. In Chapter 3 I propose two tools in order to detect performance regression between the software updates of the database PostgreSQL. The first tool is producing the precise results of performance results based on the TPC-H benchmark, and produces graphs that demonstrate the time that was needed to execute all the twenty two queries using various hardware specifications that were available on the Scilens cluster of CWI.

The second tool's usability is to find the differences in the code between two different versions and track down the file that caused the most performance regression, after benchmarking and recompiling all the different test patches that are produced. In Chapter 4, I analyze the costs of Software Quality Assurance of Information Systems and the benefits of automating the process. Finally in Chapter 5, I present my conclusions regarding the evaluation of the automated performance regression tests as well as propositions for future expansions of this project, regarding both the QA and the IT sector. In the Appendix, I present the twenty two queries that were used as well as the flowcharts for the tools.

## Chapter 2

### Related work

In this section, I will present an overview of the related research that has been done in Relational Database Management Systems regarding benchmarking. More specifically, I will be analyzing the TPC Benchmarking process, the queries and the industry relevant data that are being produced by the TPC data generator.

### 2.1 The TPC Benchmarking

In the early 1980's, the industry started a competition that has accelerate over time, the automation of business transactions. One of the first applications that had worldwide focus was the Automated Teller Machine (ATM), and we have seen this online model of computing being applied to every aspect of the business, from groceries to gas stations to multinational companies. Users were involved for the first time in the creation of update transactions in an online database system. Thus, an online transaction processing industry was slowly manufactured, an industry that in the present represents billions of dollars in annual sales. So database and computer systems vendors began to make claims on the performance of their systems, based on the benchmark TP1, that was originally developed by IBM, the first attempt for a civilized test, or benchmark between the competition of which company produces the best system. As it was expected, the benchmark had flaws, as ignoring the network and user interaction components of an OLTP (On-Line Transaction Processing), it could generate inflated results. This situation also frustrated the vendors, because they felt that their competitors' claims were based on a flawed benchmark [4].

The Transaction Processing Performance Council was created on August 1988 and has two major organizational activities. The first is the creation of solid, reliable benchmarks and the second is creating a good process for reviewing and monitoring those benchmarks and their results. Those two organizational activities are quite important, because they lay the foundation for fair competition between the companies or entities that create the database systems. The first benchmark was created in 1989 by the TPC organization with the name of TPC-A. This benchmark was the foundation of later benchmarks, and it measured the total performance of a system, including the operating system, the database management system and other related components that are also involved in the transaction processing operation. Over the years, other TPC Benchmarks like TPC-B and TPC-C were developed and applied. The first TPC-C result that was published in 1992 had a 54 tpmC (Transactions per minute) with the cost per tpmC of \$188.562. A bit more than 6 years later, the best result was a 52.871 tpmC with the cost of \$135 per tpmC.

This tremendous improvement can be a reason of performance increase in hardware and software products and vendors improving their products in order to eliminate performance bugs that were exposed by the benchmark [6].

For the purpose of this thesis the TPC-H benchmark will be implemented for testing the performance of the database. The TPC-H is a decision support benchmark. It includes 22 business oriented queries written in SQL, and the data that are generated are implicitly chosen to have industry relevance. It is designed for decision support systems that have to thoroughly search through large volumes of data. The queries are designed to have a high degree of complexity and assist with the process of answering critical business questions. The performance metric is Query-per-Hour (QphH@Size), and it represents the capability of the system to process those queries. It reflects the query processing power when they are submitted by a single stream and also when they are submitted by multiple concurrent users. The price/performance metric is \$/QphH@Size. The queries are written in SQL-92 language and are annotated to specify the rows that must be returned when needed. In order to make the queries compatible for the PostgreSQL database management system, they had to be rewritten, while following all the compliance rules. No new query or variant of an existing query has been used during this project [5].

The functionality and use of its query is as follows. Pricing summary report query (Q1), Minimum cost supplier query (Q2), Shipping priority report query (Q3), Order priority checking query (Q4), Local supplier volume query (Q5), Forecasting revenue change query (Q6), Volume shipping query (Q7), National market share query (Q8), Product type profit measure query (Q9), Returned item reporting query (Q10), Important stock identification query (Q11), Shipping mode and order priority query (Q12), Customer distribution query (Q13), Promotion effect query (Q14), Top supplier query (Q15), Parts/Supplier relationship query (Q16), Small quantity order revenue query(Q17), Large Volume customer query (Q18), Discounted revenue query(Q19), Potential part promotion query (Q20), Suppliers who kept orders waiting query (Q21), Global Sales opportunity query (Q22) [5]. As demonstrated, every query is business relevant and has been designed in such a way that can represent the workloads of a company. The scale factor of the data can be chosen by the user, in order to achieve greater connection between the existing queries and real workloads.

An example of the business question and the code to execute it in SQL for Query 3 is as follows:

Business Question: Q3 retrieves the shipping priority and potential revenue,(which is defined as the sum of  $l\_extendedprice * (1-l\_discount)$  ), of the orders who have the largest revenue among those that had not been shipped yet, as of the given date. The orders are listed in decreasing order of the revenue. If more than 10 not shipped orders exist, only the 10 orders with the largest revenue are listed.

When it comes to fair bench marking, there are many common pitfalls. The first is the non-reproducibility of the experiment. The possibility of reproducing experiments is fundamental in science.

When other researchers cannot verify your results, by reproducing the experiment, your claims regarding a benchmark are not widely accepted, because nobody can verify that your results are indeed correct. Another pitfall is the failure to optimize the database for the precise benchmark. Finally it is imperative that the two systems that are being compared with the benchmark have the same functionality.

It is also important that the differentiation between the “hot” and the “cold” runs is made. The “cold” run represents the first time that the query is executed and the data are being loaded from a persistent storage, so it is significantly slower than the “hot” runs, which are loaded after the first time from a buffer pool [3].

In conclusion, fair bench marking can be hard. There are many processes that must be addressed, not only in the experiment itself and optimizing the database for the bench marking, but also the pre-processing is the same between the systems, automatic indexing is not turned on and the tests have been done in multiple data sets [7].

## 2.2 PostgreSQL and Bench marking

PostgreSQL is one of the most used open source database management system. It currently ranks as fourth in the database industry and second when it comes to open source database systems as shown in Figure 1.

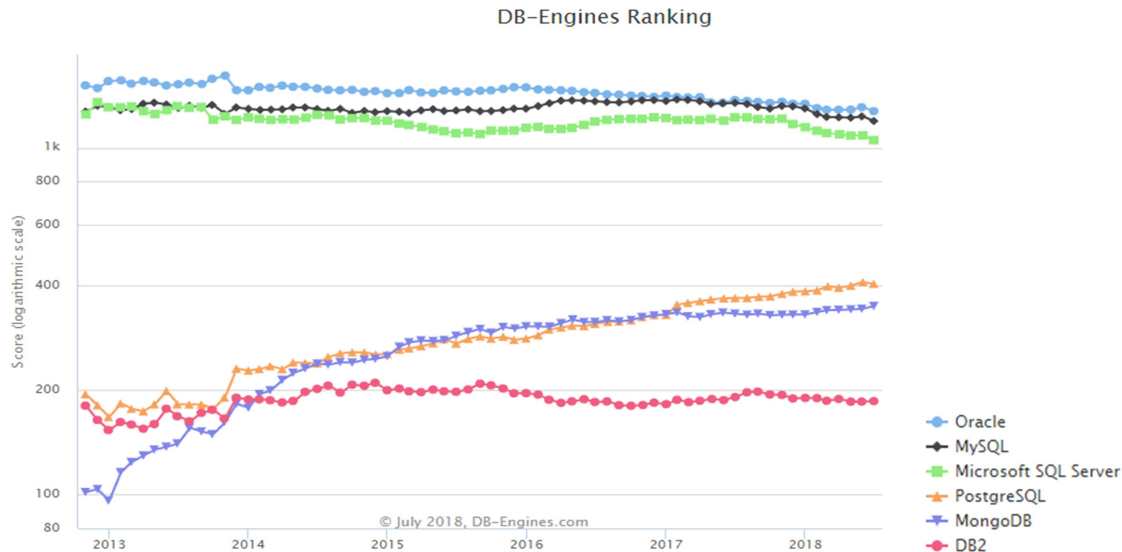


Figure 1: Current Ranking of Database Engines regarding usability (Source: [www.DB-Engines.com](http://www.DB-Engines.com) , July 2018)

Every database system is based on a model, with the exception of schema-less ones. The model is responsible for handling the data using applications or libraries in order to manage databases of various sizes and sorts. PostgreSQL is a relational database system. The relational model reforms all the data to be stored by defining relationships and related entities with unique attributes. Some of the known advantages of PostgreSQL in comparison with other database systems are its strong and experienced community, which can be accessed through free knowledge bases, it has a strong third party support, it can be extended programmatically with stored procedures and finally PostgreSQL is an objective Relational Database System, with support for nesting [8]. Some of its disadvantages are that for simple read heavy operations PostgreSQL might perform slightly worse than other database systems like MySQL and it is sometimes hard to find service providers that can supply managed instances and support. For the purposes of this thesis, my choice to work with PostgreSQL derives from its abilities to deliver reliability and data integrity, the extensibility that it offers and the effortless integration of Python scripts.

For some cases, like the TPC-B benchmark, some tools already exist that help with the benchmarking processes. The most popular of those tools is the PgBench. The target areas of PgBench are the hardware of the system, the PostgreSQL core operators and the identification of performance regression.

For TPC-H, there is no tool that allows a user to automate the process of benchmarking or optimize the database for the benchmark. In general, while there are tools for analyzing OLTP (Online Transaction Processing) models for measuring transactions per second, there are no OLAP (Online Analytically Processing) tools. Usually, OLAP models are more complicated, due to the escalated complexity of the queries and the involvement of aggregations. For OLAP systems the response time is the most important element, and it demonstrates how effective is the system regarding a workload, and regarding the hardware [5], [9].

When it comes to PostgreSQL optimization, there is not a standardized way to optimize the database system, precisely for the benchmark. Index scans are usually preferable over sequential scans, yet again, if the “SELECT” statement will return more than 10% of all rows in a table, a sequential scan is much faster from an index scan. The main reason for this behavior is because an index scan can require several In/Out operations for every row, look for the row in the index and finally retrieve the row from a heap, while a sequential scan will require only a single In/Out for every row. Sometimes, even less, because of the fact that a block (page) on the disk most probably contains more than one row. There are some statements like “EXPLAIN” and “EXPLAIN ANALYZE” in order to help you determine the approach that PostgreSQL is following to execute the queries and the actual performance of the precise approaches. The main problem is that query plans are not easy to read, with the information being closer to machine language. In order to visualize the problem better, there is a tool with the name Postgres Explain Viewer (PEV) that simplifies query plans. It demonstrates a horizontal tree with nodes representing query plan. It provides the error amount in the planned time versus the actual execution time, and information about the most “expensive” nodes or “bad estimates” as shown in Figure 2.

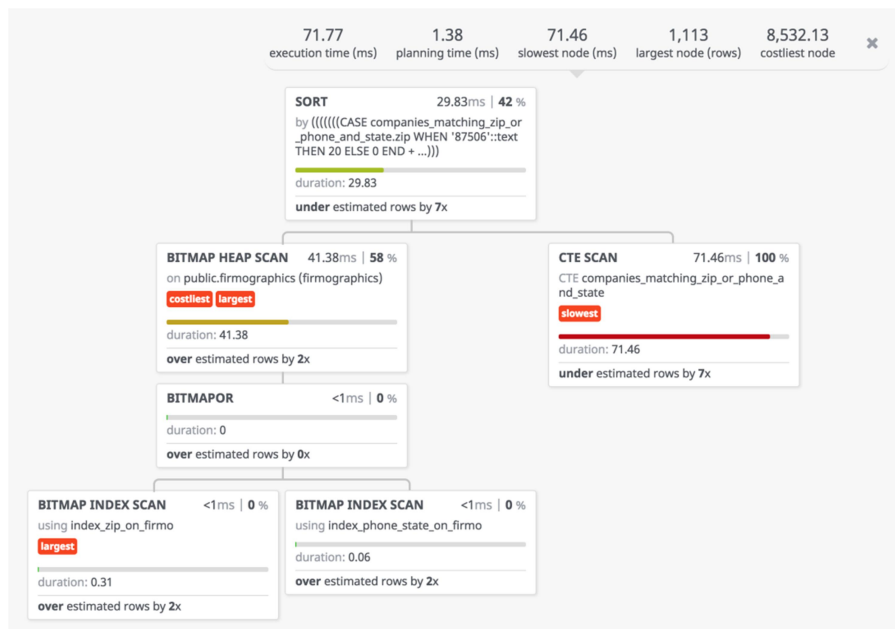


Figure 2: Postgres Explain Viewer horizontal tree



Another option regarding performance increase is to create views and query the views precisely. A view is a tool for storing partial queries, and they can be treated as tables, having to look through a much more limited amount of data. Also, a view can be materialized, with the results being stored by PostgreSQL. The statements are “CREATE MATERIALIZED VIEW” and “REFRESH MATERIALIZED VIEW”. So in read-heavy operations, like the 17<sup>th</sup> query of the TPC-H benchmark, the cost of the partial query is compensated. Materialized views are especially helpful when someone is performing identical operations such as “SUM” or “COUNT” and when joining additional tables. In conclusion, a materialized view is way more “cheaper” regarding the resources of the system than a full statement [9].

For the purposes of this thesis, I have decided not to perform any other optimization regarding the queries other than that the allowed one index per table, due to compliance with the TPC benchmark regulations and policies

## **2.3 Python with combination of Database Management systems**

Python is one of the most popular and widely accepted scripting language regarding databases. One strength of python is its use with relational and also NoSQL (Not Only SQL) databases. NoSQL databases are undergoing substantial growth in web applications and Big Data due to their agility in permitting data to be stored in a flexible manner than the relational model allows. On the other hand, this flexibility has its downsides such as limited support for consistent transactions [11]

The integration features of Python are also one of its advantages. It integrates the Enterprise Application Integration which makes it extremely convenient to develop Web Services and call on databases by invoking COM components. As a result of Python’s extensive libraries support and its productivity in comparison with other scripting languages, the programming part of this thesis is mostly occupied by python, and a small part of SQL for the creation and querying the bench marking tables.

## Chapter 3

### Implementation and Evaluation

All information of an organization derives from the data its produces and manipulates. The usage of high quality Database systems is one of the most important functions that interest any firm in any business sector. I have chosen to apply the concepts of automation in one of the most used open sources databases, PostGreSQL. More precisely, the automation of globally accepted and acknowledged performance tests is an attractive idea to the current market, because of the value business intelligence adds to a firm.

The purpose of the tools is to automate the benchmarking process between the software updates of PostGreSQL. The tools use Github in order to install and pull new updates of PostGreSQL. They compare the performance of two versions of PostGreSQL regarding time and produce graphical representations of each of the 22 queries from the TPC-H organization, as well as the total Power@Size comparison. The second tool, if regression is detected, will also track down the file that caused the most regression.

In this section, I will give an in depth description of the tools and their usability. I will be explaining the implementation process and the decisions that the user must take regarding the bench marking process and how those decisions will influence the results. I will be analyzing my decisions regarding the creation and architecture of the tools. A precise flowchart of both of the tools can be found in the appendices.

### 3.1 Analyzing the first tool and the results

Regarding the queries of the TPC-H benchmark, each query is defined by the business question, the functional query definition, the substitution parameters, which describe how to enter values that are needed to complete the syntax, and the query validation. For example the initial syntax from TPC-H for query 13 is shown in figure 3.

```
Select
c_count, count(*) as custdist
from ( select c_custkey, count(o_orderkey)
      from customer left outer join orders on c_custkey = o_custkey
      and o_comment not like '%[WORD1]%[WORD2]%'
      group by c_custkey)
as c_orders (c_custkey, c_count)
group by c_count
order by custdist desc, c_count desc;
```

Figure 3: Query 13 from the TPC-H specifications (Source : URL : [http://www.tpc.org/tpc\\_documents\\_current\\_versions/pdf/tpc-h\\_v2.17.3.pdf](http://www.tpc.org/tpc_documents_current_versions/pdf/tpc-h_v2.17.3.pdf))

For the values “Word1” and “Word2”, the documentation of TPC-H clarifies that Word1 must be randomly selected from four possible values: “Special”, “pending”, “unusual” and “express”. Word2 must be randomly selected from the four possible values: “packages”, “requests”, “accounts” and “deposits”. For the sake of speed I implemented the queries from the script made from Hannes Muehleisen for the MonetDB/PostgreSQL comparison [10], as shown in figure 4.

```
1  select
2      c_count,
3      count(*) as custdist
4  from
5      (
6          select
7              c_custkey,
8              count(o_orderkey)
9          from
10             customer left outer join orders on
11                 c_custkey = o_custkey
12                 and o_comment not like '%special%requests%'
13          group by
14              c_custkey
15      ) as c_orders (c_custkey, c_count)
16  group by
17      c_count
18  order by
19      custdist desc,
20      c_count desc;
```

Figure 4: TPC-H Query 13

All queries represent business scenarios that tend to manifest themselves in everyday business situations and business relevant queries. The output of the queries has been tested for correctness, as non-integer expressions including prices are expressed in decimal notation with at least two digits behind the decimal point and dates are expressed in a format that includes the year month and day in an integer form (YYYY-MM-DD). All the results of the queries match the validate output of data, specified in in the TPC-H documentation.

The first tool is focused between the current commit of PostgreSQL and the next update. It installs PostgreSQL, if it is already installed it recognizes the installation, downloads the TPC-H benchmark and generate the data for the tables for a scale factor that the user will provide. The data are being generated with the use of the QGEN Program. The QGEN program is a TPC provided software package that must be used in order for the results of benchmarking to be considered valid. Next, the tool creates the data directory that the tables will reside, inserts the data into the tables and runs the 22 queries, five times each.

Next, after storing the results, it drops the tables and upgrades to the next commit of PostgreSQL by pulling it from the source on Github. After that, it recreates the tables, regenerate the data and insert the data into the tables. The time loading the data is also being compared between the two versions. Next, it runs the TPC-H queries again and compares the results from the previous version of each query individually as well as the total Power@size that it is defined by the following mathematical expression from the TPC organization:

$$\text{TPC-H Power@Size} = \sqrt[24]{((3600 * \text{SF}) / \sum Q_i(i, 0))}$$

Where SF is the scale factor for the data to be produced from QGEN and Q is the product of the 22 queries. TPC-H implementations represent the raw query execution power of the system in the least amount of time [5].

The tool uses multiple Python libraries as shown in Figure 5, as NumPy which adds support for multidimensional arrays and matrices, and is accompanied with a collection of mathematical functions in order to be able to operate on these matrices. The matplotlib.pyplot library was used for developing the graphical representations of the results. Psycpg2 was used in order to establish a connection with the database PostgreSQL.

```
3  import numpy as np
4  import matplotlib.pyplot as plt
5  import psycpg2
6  import getpass
7  import os
8  import time
9  import subprocess
10 import math
11 import statistics
12
13 from os.path import expanduser
14 from operator import sub
15 from numpy import prod
```

Figure 5: Libraries used.

The adapter between the database and the Operating System based scripts is the module Psycopg2 for establishing the initial connection and the manipulation of the database, as creating the tables, loading the data that was generated into the tables and write the queries as shown in Figure 6.

```
conn=psycopg2.connect("dbname='%s' user='%s' host='localhost'" % (DBname,Uservar))

cur=conn.cursor()

cur.execute("""Create table nation (n_nationkey integer, n_name text, n_regionkey integer, n_comment text)""")
cur.execute("""Create index GRulez0 on nation (n_nationkey);""")

cur.execute("""Create table supplier (s_suppkey integer, s_name text, s_address text, s_nationkey integer, s_phon
cur.execute("""Create index GRulez1 on supplier (s_suppkey);""")

cur.execute("""Create table region (r_regionkey integer, r_name text, r_comment text)""")
cur.execute("""Create index GRulez2 on region (r_regionkey);""")
```

Figure 6: Psycopg2 connection

In order to update to the newest version of PostgreSQL, it uses the pull method from the source of PostGre in Github. The version that is being tested is displayed. In the case of absence of a new commit, a “sleep” command will be initiated for a small amount of time, and then retry to update the software as shown in Figure 7.

```
os.chdir(pathupdatecommit)
ComValue1=os.popen("git rev-parse HEAD").read()
print ("Current commit version is : %s" %ComValue1)
print "Updating to the new commit of Postgresql.."
os.system("git pull https://github.com/postgres/postgres")
print "-----"
ComValue2=os.popen("git rev-parse HEAD").read()

while ComValue1==ComValue2:
    print ("There is no new commit available, waiting for 3 minutes..")
    os.system("sleep 3m")
    os.system("git pull https://github.com/postgres/postgres")
    ComValue2=os.popen("git rev-parse HEAD").read()
```

Figure 7: Updating to the newest version

After the queries have been executed in both of the commits, the matplotlib.pyplot library is being used to produce the graphical representation of the results and finally the TPC-H Power@size difference between those two commits, shown in figure 8.

```
means_sizepower=(SizePowerMed, SizePowerCold)
means_sizepowerb=(SizePowerMedb, SizePowerColdb)
fig, ax=plt.subplots()
index=np.arange(2)
bar_width=0.35
opacity=0.8
rects1=plt.bar(index,means_sizepower,bar_width,alpha=opacity,color='blue', label='TPC-H Power@Size before update')
rects2=plt.bar(index+bar_width,means_sizepowerb,bar_width,alpha=opacity,color='green', label='TPC-H Power@Size after update')
plt.xlabel('')
plt.ylabel('')
plt.title("TPC-H Power@size")
plt.xticks(index+bar_width,('Power@Size Hot Runs Median', 'Power@Size Cold runs'))
plt.legend()
plt.tight_layout()
plt.show()
```

Figure 8: Code for graphical representations.

Hardware and computer specifications play an important role in benchmarking, so the script was executed in the Scilens Cluster of Centrum Wiskunde & Informatica, using various machines and specifications. The scripts were executed on the Diamonds, Stones, Bricks and Gems configurations, each representing a different tier of hardware specification. The precise hardware configuration follows in Table 1.

	<b>Diamonds</b>	<b>Gems</b>	<b>Stones</b>	<b>Bricks</b>	<b>Rocks</b>
<b>Type</b>	Intel Xeon E5-4657L v2	Intel Xeon e5-2650 v2	Intel Xeon E5-2650 v2	Intel Xeon 2650	Intel Core i7-2600k
<b>Architecture</b>	x86_64	x86_64	x86_64	x86_64	x86_64
<b>CPU op-mode(s)</b>	32-bit, 64-bit	32-bit, 64-bit	32-bit, 64-bit	32-bit, 64-bit	32-bit, 64-bit
<b>Byte Order</b>	Little Endian	Little Endian	Little Endian	Little Endian	Little Endian
<b>CPU(s)</b>	96	32	32	32	8
<b>Thread(s) per core</b>	2	2	2	2	2
<b>Core(s) per socket</b>	12	8	8	8	4
<b>Sockets</b>	4	2	2	2	1
<b>NUMA node(s)</b>	4	2	2	2	1
<b>CPU family</b>	6	6	6	6	6
<b>Model</b>	62	62	62	45	42
<b>Stepping</b>	4	4	4	7	7
<b>Clockspeed</b>	2.4 GHz	2.6 GHz	2.6 GHz	2.0 GHz	3.4 GHz
<b>Virtualization</b>	VT-x	VT-x	VT-x	VT-x	VT-x
<b>L1d cache</b>	32 KB	32 KB	32 KB	32 KB	32 KB
<b>L1i cache</b>	32 KB	32 KB	32 KB	32 KB	32 KB
<b>L2 cache</b>	256 KB	256 KB	256 KB	256 KB	256 KB

Table 1: CWI Scilens-configuration standard (Source: URL: <https://www.monetdb.org/wiki/Scilens-configuration-standard>)

The results of the queries are separated in “cold” and “hot” runs. The “cold” run represents the first time a query is being executed after the tables are created and populated. It takes CPU time to figure out how to run a query. That is because PostGre needs to compile an execution plan in order to figure out the best way to run the query. PostgreSQL uses memory to cache execution plans in order to save time the next time the same query is executed. A “hot” run of a query is considered a query execution, when the query has been executed already in the past. Therefore there is no CPU time loss to execute the query, because there is no need to compile an execution plan again.

Two versions of PostGre were selected randomly in order to execute the test and present the results for the first tool. The versions are “f9fe269ca21808c1f6a3d0d239365fa4eaf2b389” and “af63926cf577f4c30q43b7651e93e3a5eaa262e0”. The tool was executed in all tiers of the Scilens cluster for scale factor 1, which means that the data that are being produced by the QGEN software will have a size of 1 Gigabyte. The results in the diamonds tier, for the commits regarding the “cold run and “hot” runs average differences are represented in table 2:



	Version: <b>f9fe269ca21808c1f6a3d0d239365fa4eaf2b389</b>		Version: <b>af63926cf577f4c30q43b7651e93e3a5eaa262e0</b>	
	“Hot” Run”(seconds)	“Cold” Run (seconds)	“Hot” Run(seconds)	“Cold” Run (seconds)
Query 1	5.746	5.967	5.773	5.789
Query 2	8.158	8.259	6.156	6.265
Query 3	1.194	1.795	1.183	1.96
Query 4	1.145	1.245	1.145	1.258
Query 5	2.624	2.683	2.645	2.696
Query 6	8.528	8.565	0.526	0.696
Query 7	9.598	9.696	9.540	10.129
Query 8	8.713	8.769	6.708	6.968
Query 9	2.893	2.963	2.118	2.369
Query 10	0.842	0.965	0.872	0.963
Query 11	0.189	0.192	0.199	0.269
Query 12	0.624	0.631	0.662	0.774
Query 13	0.565	0.567	0.560	0.591
Query 14	0.535	0.597	0.540	0.553
Query 15	1.215	1.369	1.116	1.124
Query 16	0.974	1.236	0.933	1.124
Query 17	0.212	0.458	0.212	0.365
Query 18	5.130	5.362	5.328	5.698
Query 19	0.180	0.189	0.093	0.154
Query 20	0.081	0.096	0.001	0.0013
Query 21	3.045	3.178	3.058	3.069
Query 22	0.313	0.369	0.296	0.495
Time Loading data	52.397		60.803	

Table 2

The results for the stones tier are represented in Table 3:

	Version: <b>f9fe269ca21808c1f6a3d0d239365fa4eaf2b389</b>		Version: <b>af63926cf577f4c30q43b7651e93e3a5eaa262e0</b>	
	“Hot” Run”(seconds)	“Cold” Run (seconds)	“Hot” Run(seconds)	“Cold” Run (seconds)
Query 1	5.127	5.569	5.369	5.698
Query 2	7.945	8.693	7.536	7.969
Query 3	1.096	1.125	1.140	1.254
Query 4	1.006	1.012	1.014	1.019
Query 5	2.216	2.396	2.225	2.239
Query 6	0.391	0.478	0.399	0.475
Query 7	8.453	9.019	8.597	8.602
Query 8	0.511	0.698	0.512	0.632
Query 9	1.629	1.785	1.596	1.637
Query 10	0.659	0.796	0.658	0.661
Query 11	0.145	0.249	0.149	0.178
Query 12	0.485	0.501	0.478	0.511
Query 13	0.447	0.494	0.569	0.571
Query 14	0.389	0.391	0.201	0.421
Query 15	0.852	0.969	0.365	0.474
Query 16	0.747	0.878	1.092	1.147
Query 17	0.169	0.256	0.009	0.019
Query 18	4.211	4.563	4.107	4.117
Query 19	0.071	0.124	0.009	0.017
Query 20	0.002	0.003	0.087	0.103
Query 21	2.501	2.695	2.963	3.145
Query 22	0.225	0.334	0.109	0.295
Time Loading data	49.329		51.968	

Table 3

The results for the bricks tier are represented in Table 4:

	Version: <b>f9fe269ca21808c1f6a3d0d239365fa4eaf2b389</b>		Version: <b>af63926cf577f4c30q43b7651e93e3a5eaa262e0</b>	
	“Hot” Run”(seconds)	“Cold” Run (seconds)	“Hot” Run(seconds)	“Cold” Run (seconds)
Query 1	5.890	5.920	5.891	5.930
Query 2	0.150	0.154	0.147	0.154
Query 3	1.077	1.093	1.081	1.129
Query 4	1.069	1.075	1.069	1.095
Query 5	2.768	2.801	2.741	2.894
Query 6	0.451	0.453	0.463	0.465
Query 7	10.868	11.410	10.510	10.671
Query 8	0.617	0.638	0.632	0.682
Query 9	1.974	2.003	2.082	2.083
Query 10	0.784	0.799	0.806	0.813
Query 11	0.169	0.714	0.173	0.174
Query 12	0.559	0.564	0.579	0.580
Query 13	0.563	0.573	0.582	0.589
Query 14	0.455	0.457	0.470	0.474
Query 15	1.030	1.114	1.022	1.034
Query 16	0.924	1.009	0.926	0.906
Query 17	0.212	0.258	0.216	0.250
Query 18	4.937	5.125	4.843	5.134
Query 19	0.057	0.060	0.059	0.062
Query 20	0.001	0.003	0.002	0.003
Query 21	3.065	3.094	3.127	3.107
Query 22	0.242	0.274	0.279	0.283
Time Loading data	228.103		195.719	

Table 4

The results for the rocks tier are represented in Table 5:

	Version: <b>f9fe269ca21808c1f6a3d0d239365fa4eaf2b389</b>		Version: <b>af63926cf577f4c30q43b7651e93e3a5eaa262e0</b>	
	“Hot” Run”(seconds)	“Cold” Run (seconds)	“Hot” Run(seconds)	“Cold” Run (seconds)
Query 1	4.536	4.896	4.566	4.573
Query 2	0.095	0.110	0.094	0.100
Query 3	0.816	0.832	0.783	0.783
Query 4	0.776	0.778	0.776	0.778
Query 5	2.109	2.116	2.110	2.119
Query 6	0.327	0.328	0.329	0.336
Query 7	7.770	7.780	7.812	7.889
Query 8	0.449	0.558	0.454	0.460
Query 9	1.507	1.514	1.518	1.523
Query 10	0.538	0.584	0.591	0.584
Query 11	0.122	0.122	0.122	0.122
Query 12	0.405	0.407	0.407	0.430
Query 13	0.426	0.445	0.429	0.430
Query 14	0.321	0.322	0.320	0.328
Query 15	0.718	0.726	0.717	0.722
Query 16	0.615	0.623	0.606	0.614
Query 17	0.144	0.172	0.144	0.170
Query 18	3.669	3.703	3.705	3.801
Query 19	0.050	0.049	0.050	0.059
Query 20	0.001	0.001	0.001	0.001
Query 21	2.257	2.240	2.821	2.258
Query 22	0.192	0.206	0.203	0.217
Time Loading data	53.251		56.551	

Table 5

The results are being represented in the form of graphical representations for the user, for each query separately and the total Power@Size comparison between the two versions. For the sake of the reader's interest I will present the results of only two queries that were selected randomly, for the gems tier of the CWI Scilens Cluster. The versions of PostGre that are being examined are "f9fe269ca21808c1f6a3d0d239365fa4eaf2b389" and "af63926cf577f4c30q43b7651e93e3a5eaa262e0" and the results represent query number seven and query number two. The results represent the "cold run" of the query, the minimum of the "hot" runs, the maximum of the "hot" runs and the average of the "hot" runs as shown in Figure 10 and 11. The scale factor selected is 1:

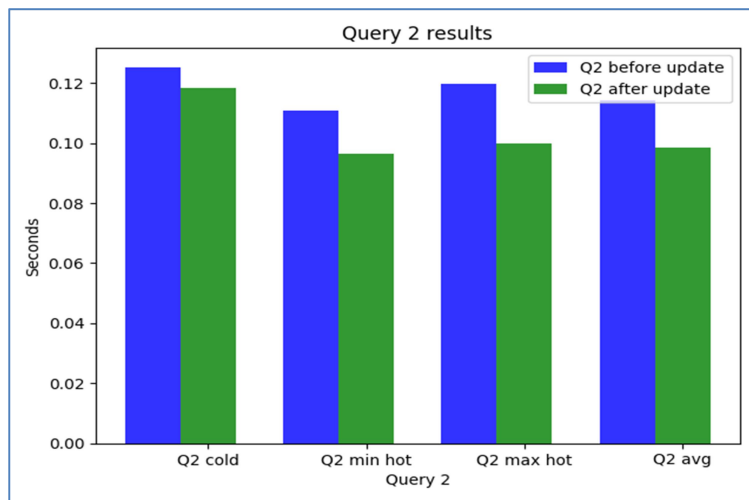


Figure 10: Query 2 performance regression test on Gems tier for scale factor 1

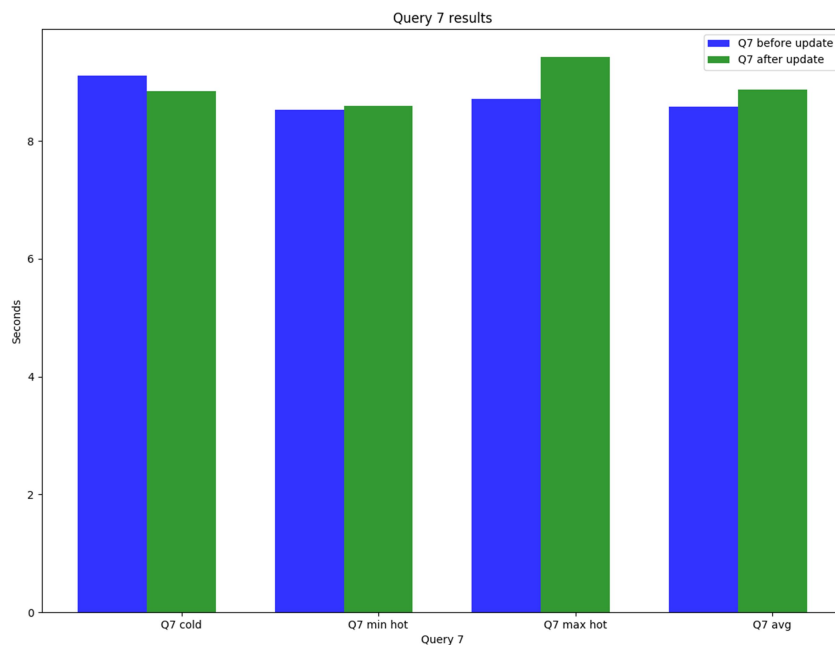


Figure 11: Query 7 performance regression test on Gems tier for scale factor 1.

The tool was executed for a scale factor of ten in the stones and bricks tier. Again I present the results for query number two and seven, so the reader can make a comparison regarding different workloads. The results are demonstrated in figures 12-15.

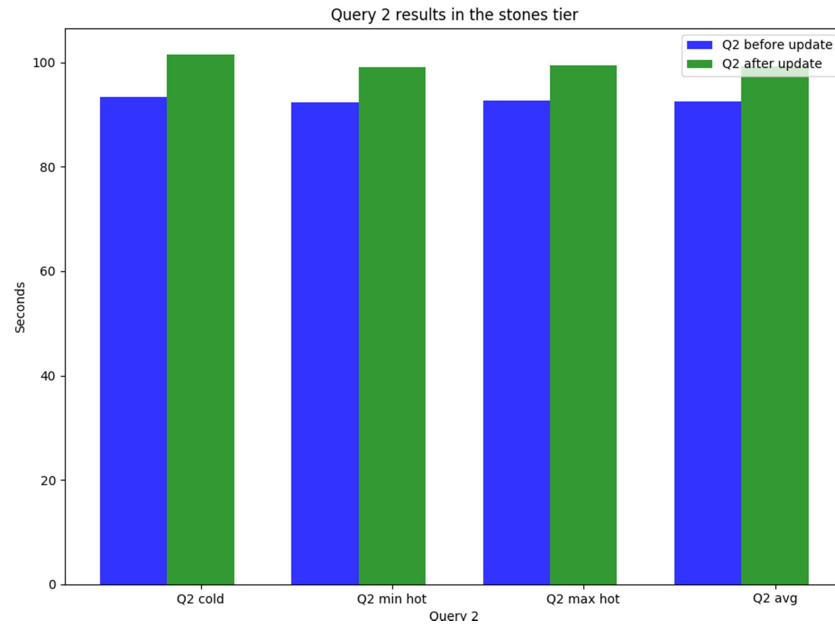


Figure 12: Query 2 performance regression test between 2 commits on Stones tier.

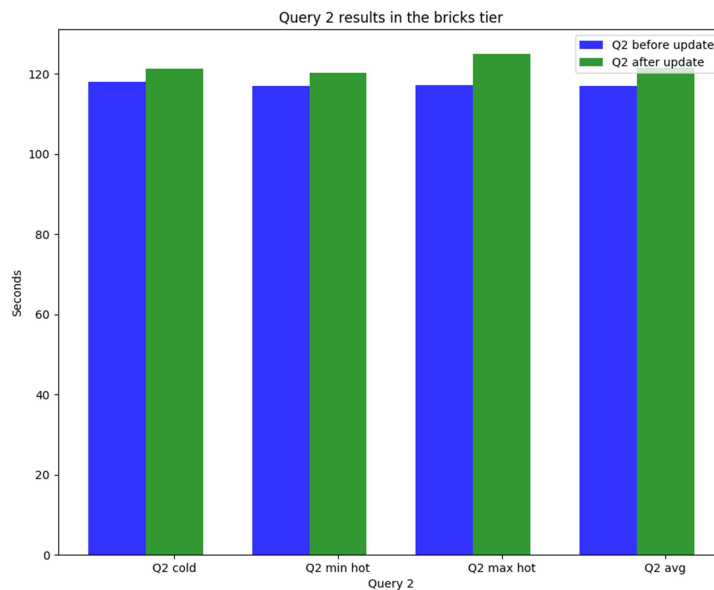


Figure 13: Query 2 performance regression test between 2 commits on Bricks tier.

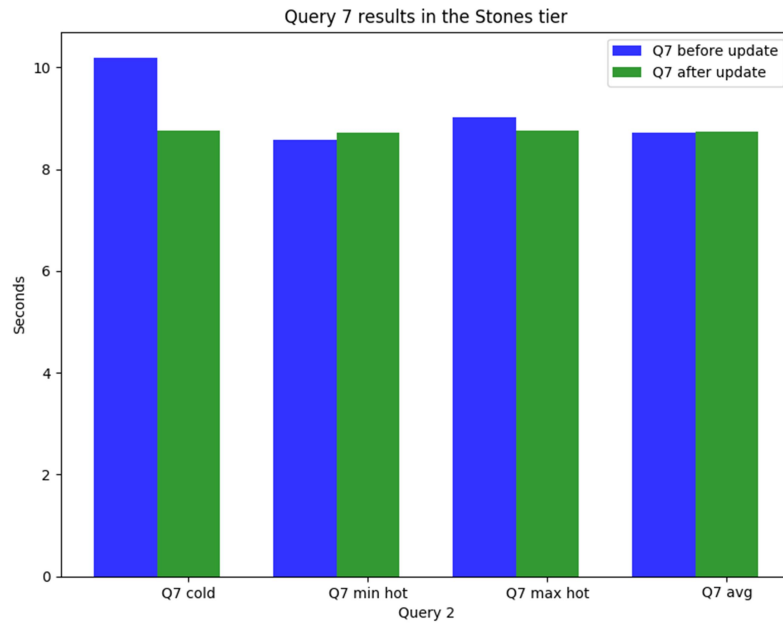


Figure 14: Query 7 performance regression test between 2 commits on the Stones tier.

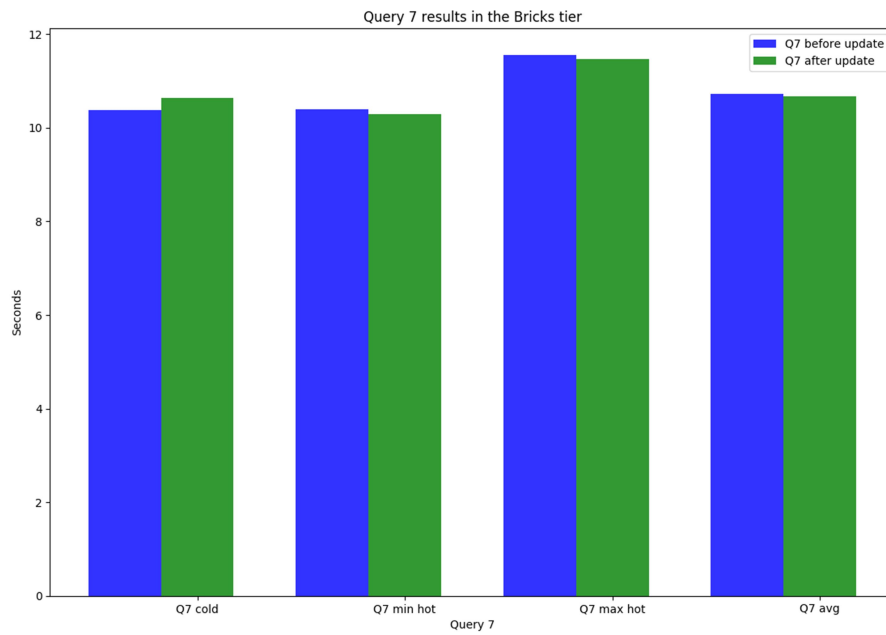


Figure 15: Query 7 performance regression test between 2 commits on the Bricks tier.

An interesting observation regarding the TPC-H queries is regarding the query number seventeen. The TPC organization is very strict regarding partitioning and indexing plans that a vendor can use in order to produce valid results. While this is to some point understandable because some techniques like materialized views would be used in order to tune the workload. While the standard allows the database to automatically pick index types, for example MySQL automatically creates indexes in any column that is declared as foreign key, it is not allowed to create join indexes, materialized views or indexed views and computed columns. The results of the 17<sup>th</sup> query on the Diamonds tier without indexing the table lineitem can be seen in figure 16.

```
Elapsed time for the 1 run of the 17th query is 4366.55145717 seconds
Elapsed time for the 2 run of the 17th query is 4360.88024211 seconds
Elapsed time for the 3 run of the 17th query is 4365.52146912 seconds
Elapsed time for the 4 run of the 17th query is 4362.6419549 seconds
Elapsed time for the 5 run of the 17th query is 4361.01572418 seconds
The cold run for the 17th query was: 4366.55145717 seconds
The fastest time for the 17th query hot runs was : 4360.88024211 seconds
The slowest time for the 17th hot runs was : 4365.52146912
Average time for the 17th query hot runs is : 4362.51484758 seconds
-----
```

Figure 16: Query 17 results without indexes on Diamonds tier for scale factor 1.

After adding a B-tree index on the l\_partkey column, the results were tremendously faster, as shown in figure 17:

```
Elapsed time for the 1 run of the 17th query is 0.280045986176 seconds
Elapsed time for the 2 run of the 17th query is 0.209609985352 seconds
Elapsed time for the 3 run of the 17th query is 0.20432806015 seconds
Elapsed time for the 4 run of the 17th query is 0.205332994461 seconds
Elapsed time for the 5 run of the 17th query is 0.204468011856 seconds
The cold run for the 17th query was: 0.280045986176 seconds
The fastest time for the 17th query hot runs was : 0.20432806015 seconds
The slowest time for the 17th hot runs was : 0.209609985352
Average time for the 17th query hot runs is : 0.205934762955 seconds
```

Figure 17: Query 17 results after indexing the l\_partkey column for scale factor 1 on Diamonds tier.



Following the same policy for the other queries, there was no major difference regarding time performance, with the exception of Query two, that was executed significantly faster for scale factor 1 in the Diamonds tier, as shown in figures 18 and 19.

```
Elapsed time for the 1 run of the 2nd query is : 90.2082688808 seconds
Elapsed time for the 2 run of the 2nd query is : 91.9842910767 seconds
Elapsed time for the 3 run of the 2nd query is : 93.889688015 seconds
Elapsed time for the 4 run of the 2nd query is : 89.472121954 seconds
Elapsed time for the 5 run of the 2nd query is : 88.8233411312 seconds
The cold run for the 2nd query was: 90.2082688808 seconds
The fastest time for the 2nd query hot runs was : 88.8233411312 seconds
The slowest time for 2nd query hot runs was : 93.889688015
Average time for the 2nd query hot runs is : 91.0423605442 seconds
```

Figure 18: Query 2 before using a B-tree index for scale factor 1 on Diamonds tier.

```
Elapsed time for the 1 run of the 2nd query is : 0.127872943878 seconds
Elapsed time for the 2 run of the 2nd query is : 0.100706100464 seconds
Elapsed time for the 3 run of the 2nd query is : 0.098592042923 seconds
Elapsed time for the 4 run of the 2nd query is : 0.0993111133575 seconds
Elapsed time for the 5 run of the 2nd query is : 0.0981168746948 seconds
The cold run for the 2nd query was: 0.127872943878 seconds
The fastest time for the 2nd query hot runs was : 0.0981168746948 seconds
The slowest time for 2nd query hot runs was : 0.100706100464
Average time for the 2nd query hot runs is : 0.0991815328598 seconds
```

Figure 19: Query 2 after using a B-tree index for scale factor 1 on Diamonds tier.

Finally, after producing the graphical representations for each query the tool will create the final comparison of the versions regarding the Power@Size each version “cold” run and “hot” runs as shown in Figure 20.

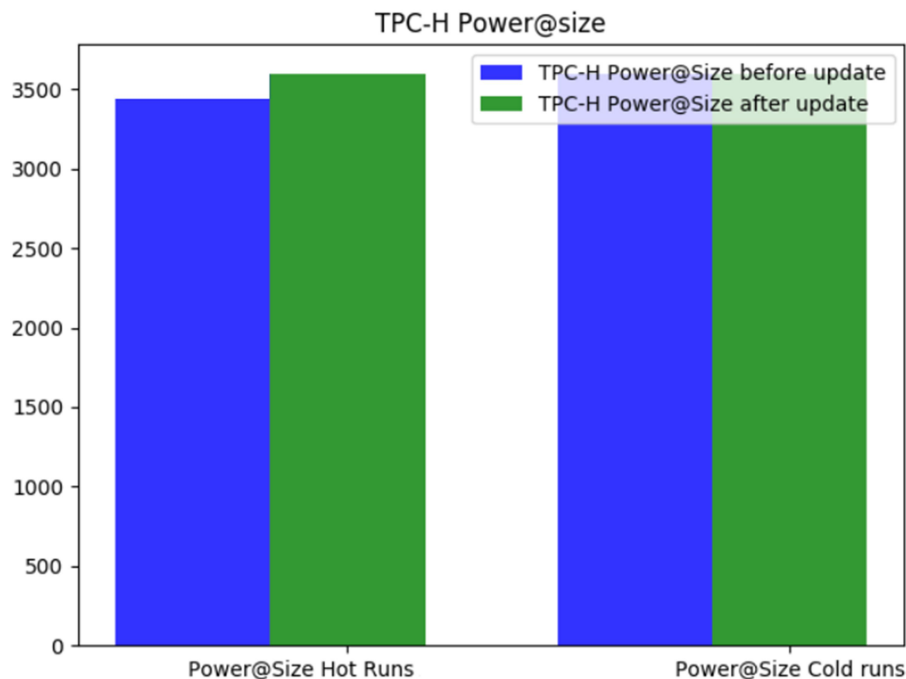


Figure 20: TPC-H Power@Size comparison between two versions of PostGre

The precise code for the first tool can be found at:

URL: <https://github.com/GAlexAnastasiou/PostGreAutomatedBenchmarking>

### 3.2 Analyzing the second tool

The second tool has the function of initially tracking time regression between two versions of PostGre. In case that regression is detected, it will track down the change of the code that caused the regression. The purpose of this tool is to help testers and DevOps track down the file of the code that caused regression in the system faster and make the necessary changes. After following the initial installation procedure of PostgreSQL and producing the data with the QGEN program, it will count the total versions that have been uploaded in the source in Github at the past, allowing the user to choose how many versions he would like to roll back in order to test for regression as shown in Figure 21:

```
The current commit is : c55de5e5123ce58ee19a47c08425949599285041
Number of total commits : 45223
Rolling back 4 commits...
HEAD is now at 56b4da8c9d Use more modern instructions for creating a new dev cycle
```

Figure 21: Total current versions, and the rolling back option.

The twenty-two queries are executed separately for every commit, six times each. The “cold” and the “hot” runs are printed for each of the versions. The comparison and the results are stored in a file for every two versions, moving forward. Every time that a new version is tested, the tables are dropped and recreated, in order to compare the loading time of the data sets. After the older version has been tested against the new one for regression regarding time, the script calculates if there was indeed regression while executing the queries and loading the data into the tables.

After we have obtained the benchmark results for a new release we have to automatically decide whether or not a performance regression has occurred. This sounds easy, but it is not. The new version of the software could perform worse simply because of a random variance. This risk increases when we test more queries, as the chance that the software will perform worse on any single one of the queries increases with the amount of queries we test. Another consideration is that the measurements itself can be imprecise. The timers use to measure only have a certain time resolution, and when we measure queries that take only several milliseconds to run the timer itself can introduce variance as well.

For this reason, I take the standard deviation of the five runs into account when attempting to classify whether or not a particular version introduces a performance regression. When the difference between the new version and the old version exceeds the standard deviation I consider that a performance regression as shown in Figure 22.

```
>>>>Performance regression detected!<<<<
Current commit is : c55de5e5123ce58ee19a47c08425949599285041
the previous commit is : 56b4da8c9d11f685f1fe2e11cf015e850913b6b8
Number of files in this commit : 13
-----
```

Figure 22: Performance regression detection between two versions and the number of files of differences in those versions.

In case of a regression detection the tool starts analyzing the changes that occurred between the versions. More precisely, it will produce the differences between the code lines of those versions, delete the last change of the differences and try to recompile PostgreSQL. In case of a successful recompilation, the queries are run again, and the time that was needed to execute them is once more compared with the initial commit. If the difference between the new version and the old version exceeds the standard deviation, the changes of the code are being stored in a file, as well as the commit name. Eventually, the file of the changes of the code that was responsible for most of the regression in comparison with the other files between those 2 versions is stored in the file:

```
The most suspicious file is : diff --git a/doc/src/sgml/libpq.sgml b/doc/src/sgml/libpq.sgml
index d67212b831..515bcb6779 100644
--- a/doc/src/sgml/libpq.sgml
+++ b/doc/src/sgml/libpq.sgml
@@ -8440,7 +8440,7 @@ main(int argc, char **argv)
/* Set always-secure search path, so malicious users can't take control. */
res = PQexec(conn,
    "SELECT pg_catalog.set_config('search_path', '', false);");
- if (PQresultStatus(res) != PGRES_TUPLES_OK)
+ if (PQresultStatus(res) != PGRES_COMMAND_OK)
{
    fprintf(stderr, "SET failed: %s", PQerrorMessage(conn));
    PQclear(res);
@@ -8610,7 +8610,7 @@ main(int argc, char **argv)
/* Set always-secure search path, so malicious users can't take control. */
res = PQexec(conn,
    "SELECT pg_catalog.set_config('search_path', '', false);");
- if (PQresultStatus(res) != PGRES_TUPLES_OK)
+ if (PQresultStatus(res) != PGRES_COMMAND_OK)
{
    fprintf(stderr, "SET failed: %s", PQerrorMessage(conn));
    PQclear(res);
The current commit is : 56b4da8c9d11f685f1fe2e11cf015e850913b6b8
```

Figure 29: The file that caused performance regression for version "56b4da8c9d11f685f1fe2e11cf015e850913b6b8"

Same policies with the first tool are being applied regarding indexing the tables. Finally, after storing the file that caused the most regression into a file with the version name, the tool proceeds to the next version, repeating the same procedure, until it reaches the final commit of PostgreSQL in Github. A flowchart of the tool can be found in the appendix.

The code for the second tool can be found at: URL: <https://github.com/GAlexAnastasiou/PostgreSQL-Benchmarking-automated-detection-of-regression-file>

## 4. Implementing automation in Software Quality Assurance

The above tools can be used in implementing automation in the testing of the newest software releases. The department that is responsible for testing the software before approving it for production status is the Software Quality Assurance department. The software is being tested for security and for complying with standards and certain policies.

According to Capgemini, Sogeti and Hewlett Packard, 35% of the IT investments of any firm are directed in testing. The prediction for the end of 2018 is that testing will claim 40% of the total IT investments as shown in Figure 30 [14]. The survey found 39% of respondents to declare that the reliance on manual testing is the most important technological challenge in application development.

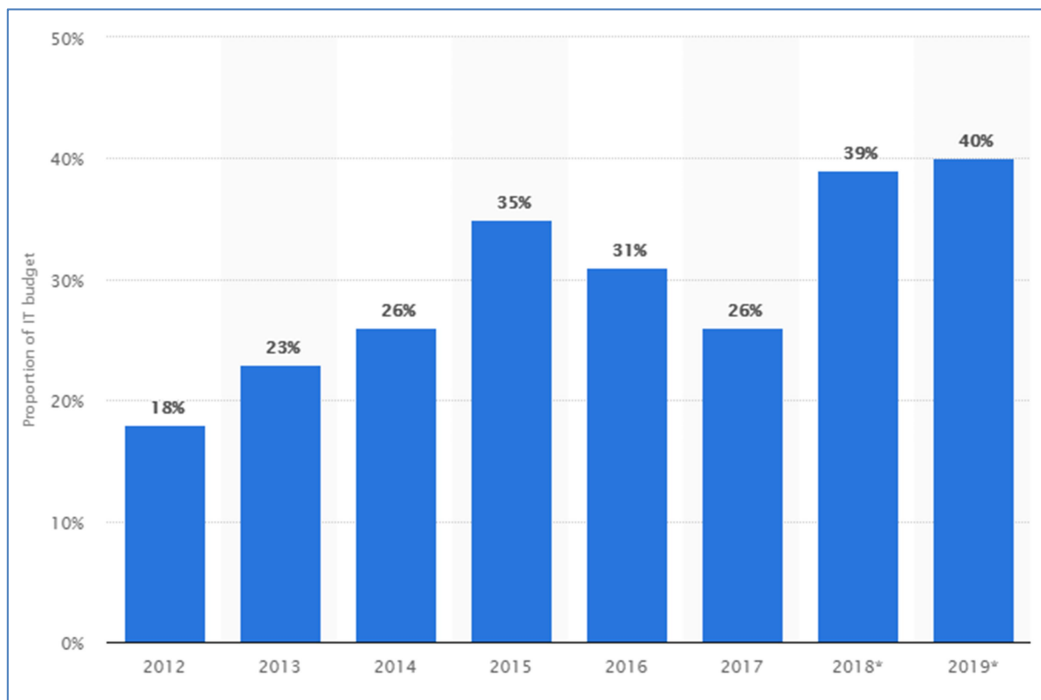


Figure 30: Proportion of IT budget spend in Quality Assurance (Source: Statista 2018)

The last release of World Quality Report 2018-19 most important finding is that end user satisfaction is now at the top of the testing strategy goals. Testing is becoming increasingly aligned with the business goals of firms. The second important finding was that 99% of the respondents use DevOps in at least some of their projects. The DevOps title derives from development and operations and aims in unifying software integration, testing, and infrastructure management. DevOps aim to deliver value to the end user as fast as possible, while keeping quality high.

The benefits of Software Quality Assurance in an organization are the higher reliability on software products, the reduced cost of overall software lifecycle and providing greater customer satisfaction while gradually reducing the maintenance costs. The main principles of Software Quality Assurance are “fir for purpose” and “right at the first time”. In order to achieve those two principles, cooperation with development is essential. The depth of testing is categorized in three levels. The first level is called “Black box” testing, which only tests if the functionality works as planned. The second level is called White Box texting, in which the internal structure of the software is tested, with code reviews. The third level is called Gray Box Testing, in which the testers have knowledge of the internal data structures and have designed the tests for that particular data structures. The tests are executed at user level. Two methodologies are currently being used. The first is Scrum, in which each part of the software is tested differently and after integration the whole product is tested. The second is the Waterfall or the V-Model, in which everything is tested at the end.

The costs that are associated with Software Quality Assurance are separated in two major types: Conformance costs and non-conformance costs. The costs of Conformance are costs of prevention of defects before they happen and the appraisal costs that include measuring and evaluating the software products in order to assure quality standards are implemented. For the prevention process, examples of costs are the training of staff in design methodologies and quality improvement meetings. Examples of appraisal costs are code inspections and testing activities. So, the conformance costs are mostly associated with the amount invested to achieve quality software products. The non-conformance costs are all the expenses that are included when things do not go as planned. Those are the costs in reprogramming and retesting [15].

So, in order to reduce the costs in Software Quality Assurance, Campanella proposes to remove the failure costs by investing in prevention activities and continue to evaluate and alter preventative efforts [16]. The approach of this idea is that software failure costs can be reduced by identifying and permanently fixing defects early in the software lifecycle because the cost of corrections increases the later the defect is recognized. The activities that are proposed in order to remove the failure costs are the creation of lifecycle development standards, the creation of detailed documentation and finally the implementation of automation in software configuration and testing processes.

Implementing automation in testing is usually done after manual testing. Manual testing includes several testers executing scenarios without using any automation tools. It is considered primitive as a method, but it is necessary in order to track unique bugs in a software system and eventually lead to an automated test. The main goal of manual testing is functionality, in all requirements. The use of automated tools is for test cases that would normally require a big amount of human intervention for the execution as well as reducing the labor force on this field.

According to most Software Quality Assurance related firms like Intland Software [17] the best practices regarding automation are test cases that are executed repeatedly, test cases that are tedious or difficult to perform manually and test cases that are time consuming. The cases that are to be avoided in automation are the cases that the requirements are changed frequently.

The tools that were created for the automation of benchmarking of PostgreSQL Database in this thesis followed this precise model. Executing 22 queries manually and benchmarking them in comparison with the newer update of the software is tedious to perform. Also testing for different scenarios and different workloads within two versions of a system is difficult to accomplish manually, since it is time consuming to test for a large amount of data, like a scale factor of 100. The process of creating the tables manually and implementing indexes on them, insert the data into the system and then run all the queries 5 times in a row while gathering the precise results regarding time performance, is time consuming and mostly repetitive. Yet it is necessary in order to provide a quality software product to the users. The tools provide a precise result for the time of execution of each query regarding the comparison of the current and previous model of the software as well as the total Power@Size. The second tool also provides an answer as to where the performance regression can be tracked in the file of the code, saving time for the testers from manually tracking down the file. The main purpose of the tools are speeding up the testing process and reduce the human labor that is required to test for performance regression.

The benefits of automation in benchmarking with those tools are the following. First and most important the optimization of time required for testers. The frequency of updates can reach two or three updates per day, which usually, the last update is a fix for the previous one. For frequent execution manual testing takes even more time for bigger systems. The testing team can be deployed in the results of the tool, rather than handle repetitive tests. The second benefit is the increase of efficiency by reducing the human error possibility. Manual testing can be mundane, and can wear testers out. The automation of the test can allow the execution without user interaction. Instead, testers can now focus on the results. The third benefit is the increase of the test coverage by allowing the user to choose how many versions of the system he would like to examine. The fourth benefit is the user environment simulation by allowing the tester to choose the scale factor. This way the tester can also simulate user satisfaction, regarding on the user's work load. Finally, the last benefit of the tools is the reusability of the tests, while the TPC-H benchmarking is still considered relevant.

The need of automation in testing processes continues to rise. However, automation is the biggest bottleneck in Software Quality Assurance. While testing was initially designed in order to purely remove bugs and evaluate the functionality of a software system, with the use of automation in repetitive tests, testers can now focus on the customer experience and business functions of the system.



The current suggestion from is the use of DevOps when possible. Testers need to be involved earlier in the processes and be more aligned with the development team, in order to be able to create automated tests for different scenarios. However the implementation of DevOps for Software Quality Assurance is part of the Agile methodology. Software Quality Assurance is currently following the Waterfall mode. The Waterfall model has five stages in software development life cycle, which are the requirements definition, the system design, the implementation, the integration and finally the operation and maintenance of the system. The Software Quality Assurances role in the Waterfall model starts after the implementation with code reviews and inspections. Using the Agile model, Software Quality Assurance has to take an active role earlier in the stages of development. The Agile model has three stages, which are the Requirement definitions and the system design using stories, the implementation and the integration. Testing must be performed in every stage of the development life cycle. In the first stage with customer feedback, in the second stage with code reviews and meetings and in the third stage by ensuring that all the functionality requirements are met [18]. That, as a consequence can increase the costs of Software Quality Assurance teams. By applying automation in the initial stages of the development, using previous test cases and user stories, can reduce the costs of Software Quality Assurance. According to Alberts, design errors in the start of the development life cycle have double the impact than do coding errors as shown in table [number]. He supports that usually the 66,% of errors in development life cycle is a product of poor design, 16,6% of errors are logic errors and 16,5% of errors are Syntax Errors [19].

Error type	% Total Errors	Severity	%Total cost of error
Design	66%	2.5	83%
Logic	16,5%	1	8%
Syntax	16,5%	1	8%
	Development phase	Operations phase	Both
% of Total Life Costs	47,5%	50%	97,5%
%Costs Due to Errors	48%	50%	-
% of Total Life Cycle Costs attributed to errors	22,6%	25%	47,6%

Table 6: Errors and Software Life Cycle Costs [19]



Based on those estimations the return of investing in better design and earlier Software Quality Assurance involvement is a multiplicative factor of five. For example, a euro more spend in design and gathering user stories and automated cases from the start of the development life cycle would have saved five euros spent on corrections and maintenance at later stages of the software life cycle. This is also visible in the Database system PostgreSQL. The regression that can be accidentally implemented with a new patch can have severe consequences in the time required to execute queries in large data sets, which will lead to slower processes and ultimately harm the user satisfaction. By tracking the regression before the newer version is implemented using the automated tools, the testers or DevOps can make the necessary changes, and avoid the costs of creating another patch, which is usually a quick fix for the previous patch. Using the automated tools, they can also get an indication of which code file caused the most regression, so they can focus their efforts in a much more targeted part of the code.

## **5. Conclusions and Future Work**

### **5.1 Conclusion**

As it was discussed, performance evaluation of databases is something deeply dependent on hardware, indexing and generally database optimization. Arriving to a conclusion regarding which database performs better, tends to be difficult and it is always correlated with the precise scenario and workload the performance test is associated with. For business in all kind of sectors, quality database systems and RDBMS's is a necessity due to the large value they add regarding information processing, market analysis and assisting management in the decision making process.

While automation of bench marking is a subject that can be controversial because of today's current market distribution, automated tools that provide testing with valid results and detailed reports, that apply different scenarios which are globally acknowledged and accepted, can shift the current trends and redistribute the database industry financially. Therefore, the automation of database bench marking can provide rapid results and acquire significantly less manpower in order to conclude which database is the best for the firm's workload and everyday querying scenarios. This decision is important to any organization since the Big Data concept started to be applied. The value of better performance in database systems is translated to faster processes, reduction of opportunity costs, better managerial decisions and ultimately customer satisfaction and loyalty.

### **5.2 Future work**

Our approach for classifying results as performance regressions is simple and effective, but still has some problems. As we only ever compare two adjacent versions for performance regressions and we have a margin for error on detecting them, performance regressions could silently bypass our system if every consecutive version very slightly reduces the performance of a query. If we do not detect a one millisecond difference in two consecutive versions as performance regression, for example, every version could reduce the performance of a query by 1 millisecond until a significant performance regression is introduced. To avoid this we would have to take account the performance of a query over time instead of just looking at consecutive versions and detecting if performance regressions slowly occur.

These regressions are more difficult to detect. It is especially challenging to find out which code is responsible for the regression, as likely many minor changes have introduced many small performance regressions instead of a single line or body of code causing the problem.

Another improvement that could be made is the amount of times we run the benchmarks. Currently we run the queries six times in the first tool and five times in the second tool, since the extra run was not necessary for the second tool. However we could choose how many queries to run adaptively based on the standard deviation. If after five runs the performance is still volatile, we might want to perform more runs. On the other hand, if three runs have exactly the same time, maybe we can already stop running the query.

While those tools can detect the regression between the new versions of the database and eventually track down the difference that added the most regression in time performance of PostGreSQL, there is also the possibility of difference files to cause regression in a precise combination. For example, file number two showed no regression between the two commits, but the combination of file two and nine may have caused most of the regression. Also, if the order that the files are listed is of importance, because of compilation problems, the permutations must be analyzed. The mathematical expression of determining the permutations within a list of files is shown below [12]:

$$P(n, r) = {}^nP_r = \frac{n!}{(n - r)!}$$

An example for applying the above mathematical type in the case of permutation with repetition inside a list that contain ten files and the user chooses to detect regression for the combination of three files, the total possibilities will be one thousand. In the case of permutations without allowing the repetition of the files, the possibilities that are produced are still of a large number. For example, a typical file that contains the differences between the commits can easily amount to sixteen differences. If the user chooses to test for regression for that particular file that contains sixteen differences and choose to test for permutations that group up to three differences, the possibilities that will be produced are 3.360. Keeping in mind that every time the new test version of PostGreSQL has to be recompiled, all the tables dropped and reloaded again and executing the twenty-two queries of the TPC-H benchmark five times each, the completion of the script can take a large amount of time. An interesting research question would be to determine a valid way to produce this test including the combinations and permutations of the differences between the commits regarding the time that the script has to conclude and the mathematical and programming validity of this idea.

## Bibliography

- [1] Chatham, Mark (2012). Structured Query Language By Example - Volume I: Data Query Language
- [2] URL : <https://www.sweor.com/firstimpressions>
- [3] Mark Raasveldt, Pedro Holanda, Tim Gubner & Hannes Mühleisen. 2018. Fair Benchmarking Considered Difficult: Common Pitfalls In Database Performance Testing. In Proceedings of 7th International Workshop on Testing Database Systems (DBTEST'18). ACM, New York, NY, USA
- [4] Darrell Huff and Irving Geis. 1993. How to Lie With Statistics. W. W. Norton & Company.
- [5] TPC Benchmark H (decision support), Standard Specification, Revision 2.17.3, Transaction Processing Performance Council
- [6] Jim Gray. 1992. Benchmark Handbook: For Database and Transaction Processing Systems. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA
- [7] Jean-Yves Le Boudec. 2010. Performance Evaluation of Computer and Communication Systems. EPFL Press.
- [8] URL: <https://www.postgresql.org/docs/>
- [9] PostgreSQL Server Programming Krosing, Hannu ; Roybal, Kirk Mlodgenski, Jim Olton: Packt Publishing 25 June 2013
- [10] Monetdb Postgre compare, Hannes Muehleisen , URL: <https://github.com/MonetDB/monetdb-postgres-compare>
- [11] URL: <https://docs.python.org/>
- [12] Permutation, Matrices and generalized young Tableaux, Donald E. Knuth, Pacific Journal of Mathematics
- [13] URL: <https://aws.amazon.com/rds/>
- [14] URL : <https://www.capgemini.com/service/world-quality-report-2017-18/>
- [15] Evaluating the Cost of Software Quality, (Sandra A. Slaughter, Donald E. Harter, and Mayuram S. Krishnan)]
- [16] Campanella, J. Principles of Quality Costs, 2nd ed., ASQC Press, Milwaukee, 1990.
- [17] URL : <https://intland.com/quality-assurance-software-testing/>
- [18] Software Quality and Agile Methods Ming Huo, June Verner, Liming Zhu, Muhammad Ali Babar National ICT Australia Ltd. and University of New South Wales, Australia
- [19] The economics of software quality assurance by David S. Alberts, The Mitre Corporation McLean, Virginia

## Appendices

### Appendix A

#### TPC Membership (April 2017)

In this appendix all the firms and organizations that are of full membership status to the TPC organization are presented in Figure 31 while the associate members are presented in Figure 32.

Full Members				
				
				
				
	 An SAP Company			

Figure 31 – Full status membership firms of the organization TPC.




Associate Members		
		

Figure 32 – Associate status firms of the TPC organization.

## Appendix B

### TPC-H Queries Business Questions and Functional Definitions

In this appendix I present all the queries business questions representations and all the queries functional definitions for PostGreSQL [10].

Query 1 Business Question: The Pricing Summary Report Query provides a summary pricing report for all lineitems shipped as of a given date. The date is within 60 - 120 days of the greatest ship date contained in the database. The query lists totals for extended price, discounted extended price, discounted extended price plus tax, average quantity, average extended price, and average discount. These aggregates are grouped by RETURNFLAG and LINESTATUS, and listed in ascending order of RETURNFLAG and LINESTATUS. A count of the number of lineitems in each group is included.

Query 1 Functional definition:

```
select
    l_returnflag,
    l_linestatus,
    sum(l_quantity) as sum_qty,
    sum(l_extendedprice) as sum_base_price,
    sum(l_extendedprice * (1 - l_discount)) as sum_disc_price,
    sum(l_extendedprice * (1 - l_discount) * (1 + l_tax)) as sum_charge,
    avg(l_quantity) as avg_qty,
    avg(l_extendedprice) as avg_price,
    avg(l_discount) as avg_disc,
    count(*) as count_order
from
    lineitem
where
    l_shipdate <= date '1998-12-01' - interval '90' day
group by
    l_returnflag,
    l_linestatus
order by
    l_returnflag,
    l_linestatus;
```

Query 2 Business Question: The Minimum Cost Supplier Query finds, in a given region, for each part of a certain type and size, the supplier who can supply it at minimum cost. If several suppliers in that region offer the desired part type and size at the same (minimum) cost, the query lists the parts from suppliers with the 100 highest account balances. For each supplier, the query lists the supplier's account balance, name and nation; the part's number and manufacturer; the supplier's address, phone number and comment information.

Query 2 Functional definition:

```
1  select
2      s_acctbal,
3      s_name,
4      n_name,
5      p_partkey,
6      p_mfgr,
7      s_address,
8      s_phone,
9      s_comment
10 from
11     part,
12     supplier,
13     partsupp,
14     nation,
15     region
16 where
17     p_partkey = ps_partkey
18     and s_suppkey = ps_suppkey
19     and p_size = 15
20     and p_type like '%BRASS'
21     and s_nationkey = n_nationkey
22     and n_regionkey = r_regionkey
23     and r_name = 'EUROPE'
24     and ps_supplycost = (
25         select
26             min(ps_supplycost)
27         from
28             partsupp,
29             supplier,
30             nation,
31             region
32         where
33             p_partkey = ps_partkey
34             and s_suppkey = ps_suppkey
35             and s_nationkey = n_nationkey
36             and n_regionkey = r_regionkey
37             and r_name = 'EUROPE'
38     )
39 order by
40     s_acctbal desc,
41     n_name,
42     s_name,
43     p_partkey
44 limit 100;
```

Query 3 Business Question: The Shipping Priority Query retrieves the shipping priority and potential revenue, defined as the sum of  $l\_extendedprice * (1 - l\_discount)$ , of the orders having the largest revenue among those that had not been shipped as of a given date. Orders are listed in decreasing order of revenue. If more than 10 unshipped orders exist, only the 10 orders with the largest revenue are listed

Query 3 Functional definition:

```
select
    l_orderkey,
    sum(l_extendedprice * (1 - l_discount)) as revenue,
    o_orderdate,
    o_shippriority
from
    customer,
    orders,
    lineitem
where
    c_mktsegment = 'BUILDING'
    and c_custkey = o_custkey
    and l_orderkey = o_orderkey
    and o_orderdate < date '1995-03-15'
    and l_shipdate > date '1995-03-15'
group by
    l_orderkey,
    o_orderdate,
    o_shippriority
order by
    revenue desc,
    o_orderdate
limit 10;
```



Query 4 Business Question: The Order Priority Checking Query counts the number of orders ordered in a given quarter of a given year in which at least one lineitem was received by the customer later than its committed date. The query lists the count of such orders for each order priority sorted in ascending priority order.

Query 4 Functional definition:

```
select
    o_orderpriority,
    count(*) as order_count
from
    orders
where
    o_orderdate >= date '1993-07-01'
    and o_orderdate < date '1993-07-01' + interval '3' month
    and exists (
        select
            *
        from
            lineitem
        where
            l_orderkey = o_orderkey
            and l_commitdate < l_receiptdate
    )
group by
    o_orderpriority
order by
    o_orderpriority;
```

Query 5 Business Question: The Local Supplier Volume Query lists for each nation in a region the revenue volume that resulted from lineitem transactions in which the customer ordering parts and the supplier filling them were both within that nation. The query is run in order to determine whether to institute local distribution centers in a given region. The query considers only parts ordered in a given year. The query displays the nations and revenue volume in descending order by revenue. Revenue volume for all qualifying lineitems in a particular nation is defined as  $\text{sum}(l\_extendedprice * (1 - l\_discount))$ .

Query 5 Functional definition:

```
select
    n_name,
    sum(l_extendedprice * (1 - l_discount)) as revenue
from
    customer,
    orders,
    lineitem,
    supplier,
    nation,
    region
where
    c_custkey = o_custkey
    and l_orderkey = o_orderkey
    and l_suppkey = s_suppkey
    and c_nationkey = s_nationkey
    and s_nationkey = n_nationkey
    and n_regionkey = r_regionkey
    and r_name = 'ASIA'
    and o_orderdate >= date '1994-01-01'
    and o_orderdate < date '1994-01-01' + interval '1' year
group by
    n_name
order by
    revenue desc;
```

Query 6 Business Question: The Forecasting Revenue Change Query considers all the lineitems shipped in a given year with discounts between DISCOUNT-0.01 and DISCOUNT+0.01. The query lists the amount by which the total revenue would have increased if these discounts had been eliminated for lineitems with l\_quantity less than quantity. Note that the potential revenue increase is equal to the sum of [l\_extendedprice \* l\_discount] for all lineitems with discounts and quantities in the qualifying range

Query 6 Functional definition:

```
select
    sum(l_extendedprice * l_discount) as revenue
from
    lineitem
where
    l_shipdate >= date '1994-01-01'
    and l_shipdate < date '1994-01-01' + interval '1' year
    and l_discount between 0.06 - 0.01 and 0.06 + 0.01
    and l_quantity < 24;
```

Query 7 Business Question: The Volume Shipping Query finds, for two given nations, the gross discounted revenues derived from lineitems in which parts were shipped from a supplier in either nation to a customer in the other nation during 1995 and 1996. The query lists the supplier nation, the customer nation, the year, and the revenue from shipments that took place in that year. The query orders the answer by Supplier nation, Customer nation, and year (all ascending).

Query 7 Functional definition:

```
1  select
2      supp_nation,
3      cust_nation,
4      l_year,
5      sum(volume) as revenue
6  from
7      (
8          select
9              n1.n_name as supp_nation,
10             n2.n_name as cust_nation,
11             extract(year from l_shipdate) as l_year,
12             l_extendedprice * (1 - l_discount) as volume
13         from
14             supplier,
15             lineitem,
16             orders,
17             customer,
18             nation n1,
19             nation n2
20         where
21             s_suppkey = l_suppkey
22             and o_orderkey = l_orderkey
23             and c_custkey = o_custkey
24             and s_nationkey = n1.n_nationkey
25             and c_nationkey = n2.n_nationkey
26             and (
27                 (n1.n_name = 'FRANCE' and n2.n_name = 'GERMANY')
28                 or (n1.n_name = 'GERMANY' and n2.n_name = 'FRANCE')
29             )
30             and l_shipdate between date '1995-01-01' and date '1996-12-31'
31         ) as shipping
32  group by
33      supp_nation,
34      cust_nation,
35      l_year
36  order by
37      supp_nation,
38      cust_nation,
39      l_year;
```

Query 8 Business Question: The market share for a given nation within a given region is defined as the fraction of the revenue, the sum of  $[l\_extendedprice * (1 - l\_discount)]$ , from the products of a specified type in that region that was supplied by suppliers from the given nation. The query determines this for the years 1995 and 1996 presented in this order.

Query 8 Functional definition:

```
select
    o_year,
    sum(case
        when nation = 'BRAZIL' then volume
        else 0
    end) / sum(volume) as mkt_share
from
    (
        select
            extract(year from o_orderdate) as o_year,
            l_extendedprice * (1 - l_discount) as volume,
            n2.n_name as nation
        from
            part,
            supplier,
            lineitem,
            orders,
            customer,
            nation n1,
            nation n2,
            region
        where
            p_partkey = l_partkey
            and s_suppkey = l_suppkey
            and l_orderkey = o_orderkey
            and o_custkey = c_custkey
            and c_nationkey = n1.n_nationkey
            and n1.n_regionkey = r_regionkey
            and r_name = 'AMERICA'
            and s_nationkey = n2.n_nationkey
            and o_orderdate between date '1995-01-01' and date '1996-12-31'
            and p_type = 'ECONOMY ANODIZED STEEL'
        ) as all_nations
group by
    o_year
order by
    o_year;
```

Query 9 Business Question: The Product Type Profit Measure Query finds, for each nation and each year, the profit for all parts ordered in that year that contain a specified substring in their names and that were filled by a supplier in that nation. The profit is defined as the sum of  $[(l\_extendedprice * (1 - l\_discount)) - (ps\_supplycost * l\_quantity)]$  for all lineitems describing parts in the specified line. The query lists the nations in ascending alphabetical order and, for each nation, the year and profit in descending order by year (most recent first).

Query 9 Functional definition:

```
select
    nation,
    o_year,
    sum(amount) as sum_profit
from
    (
        select
            n_name as nation,
            extract(year from o_orderdate) as o_year,
            l_extendedprice * (1 - l_discount) - ps_supplycost * l_quantity as amount
        from
            part,
            supplier,
            lineitem,
            partsupp,
            orders,
            nation
        where
            s_suppkey = l_suppkey
            and ps_suppkey = l_suppkey
            and ps_partkey = l_partkey
            and p_partkey = l_partkey
            and o_orderkey = l_orderkey
            and s_nationkey = n_nationkey
            and p_name like '%green%'
        ) as profit
group by
    nation,
    o_year
order by
    nation,
    o_year desc;
```

Query 10 Business Question: The Returned Item Reporting Query finds the top 20 customers, in terms of their effect on lost revenue for a given quarter, who have returned parts. The query considers only parts that were ordered in the specified quarter. The query lists the customer's name, address, nation, phone number, account balance, comment information and revenue lost. The customers are listed in descending order of lost revenue. Revenue lost is defined as  $\text{sum}(l\_extendedprice * (1 - l\_discount))$  for all qualifying lineitems.

Query 10 Functional definition:

```
select
    c_custkey,
    c_name,
    sum(l_extendedprice * (1 - l_discount)) as revenue,
    c_acctbal,
    n_name,
    c_address,
    c_phone,
    c_comment
from
    customer,
    orders,
    lineitem,
    nation
where
    c_custkey = o_custkey
    and l_orderkey = o_orderkey
    and o_orderdate >= date '1993-10-01'
    and o_orderdate < date '1993-10-01' + interval '3' month
    and l_returnflag = 'R'
    and c_nationkey = n_nationkey
group by
    c_custkey,
    c_name,
    c_acctbal,
    c_phone,
    n_name,
    c_address,
    c_comment
order by
    revenue desc
limit 20;
```

Query 11 Business Question: The Important Stock Identification Query finds, from scanning the available stock of suppliers in a given nation, all the parts that represent a significant percentage of the total value of all available parts. The query displays the part number and the value of those parts in descending order of value.

Query 11 Functional definition:

```
select
    ps_partkey,
    sum(ps_supplycost * ps_availqty) as value
from
    partsupp,
    supplier,
    nation
where
    ps_suppkey = s_suppkey
    and s_nationkey = n_nationkey
    and n_name = 'GERMANY'
group by
    ps_partkey
having
    sum(ps_supplycost * ps_availqty) >
    (
        select
            sum(ps_supplycost * ps_availqty) * 0.0001000000
        from
            partsupp,
            supplier,
            nation
        where
            ps_suppkey = s_suppkey
            and s_nationkey = n_nationkey
            and n_name = 'GERMANY'
    )
order by
    value desc;
```



Query 12 Business Question: The Shipping Modes and Order Priority Query counts, by ship mode, for lineitems actually received by customers in a given year, the number of lineitems belonging to orders for which the l\_receiptdate exceeds the l\_commitdate for two different specified ship modes. Only lineitems that were actually shipped before the l\_commitdate are considered. The late lineitems are partitioned into two groups, those with priority URGENT or HIGH, and those with a priority other than URGENT or HIGH.

Query 12 Functional definition:

```
1  select
2      l_shipmode,
3      sum(case
4          when o_orderpriority = '1-URGENT'
5              or o_orderpriority = '2-HIGH'
6              then 1
7          else 0
8      end) as high_line_count,
9      sum(case
10         when o_orderpriority <> '1-URGENT'
11             and o_orderpriority <> '2-HIGH'
12             then 1
13         else 0
14     end) as low_line_count
15  from
16      orders,
17      lineitem
18  where
19      o_orderkey = l_orderkey
20      and l_shipmode in ('MAIL', 'SHIP')
21      and l_commitdate < l_receiptdate
22      and l_shipdate < l_commitdate
23      and l_receiptdate >= date '1994-01-01'
24      and l_receiptdate < date '1994-01-01' + interval '1' year
25  group by
26      l_shipmode
27  order by
28      l_shipmode;
```

Query 13 Business Question: This query determines the distribution of customers by the number of orders they have made, including customers who have no record of orders, past or present. It counts and reports how many customers have no orders, how many have 1, 2, 3, etc. A check is made to ensure that the orders counted do not fall into one of several special categories of orders. Special categories are identified in the order comment column by looking for a particular pattern.

Query 13 Functional Definition:

```
select
    c_count,
    count(*) as custdist
from
    (
        select
            c_custkey,
            count(o_orderkey)
        from
            customer left outer join orders on
                c_custkey = o_custkey
                and o_comment not like '%special%requests%'
        group by
            c_custkey
    ) as c_orders (c_custkey, c_count)
group by
    c_count
order by
    custdist desc,
    c_count desc;
```

Query 14 Business Question: The Promotion Effect Query determines what percentage of the revenue in a given year and month was derived from promotional parts. The query considers only parts actually shipped in that month and gives the percentage. Revenue is defined as  $(l\_extendedprice * (1 - l\_discount))$ .

Query 14 Functional definition:

```
select
    100.00 * sum(case
        when p_type like 'PROMO%'
            then l_extendedprice * (1 - l_discount)
        else 0
    end) / sum(l_extendedprice * (1 - l_discount)) as promo_revenue
from
    lineitem,
    part
where
    l_partkey = p_partkey
    and l_shipdate >= date '1995-09-01'
    and l_shipdate < date '1995-09-01' + interval '1' month;
```

Query 15 Business Question: The Top Supplier Query finds the supplier who contributed the most to the overall revenue for parts shipped during a given quarter of a given year. In case of a tie, the query lists all suppliers whose contribution was equal to the maximum, presented in supplier number order.

Query 15 Functional definition:

```
create view revenue0 (supplier_no, total_revenue) as
    select
        l_suppkey,
        sum(l_extendedprice * (1 - l_discount))
    from
        lineitem
    where
        l_shipdate >= date '1996-01-01'
        and l_shipdate < date '1996-01-01' + interval '3' month
    group by
        l_suppkey;

select
    s_suppkey,
    s_name,
    s_address,
    s_phone,
    total_revenue
from
    supplier,
    revenue0
where
    s_suppkey = supplier_no
    and total_revenue = (
        select
            max(total_revenue)
        from
            revenue0
    )
order by
    s_suppkey;

drop view revenue0;
```

Query 16 Business Question: The Parts/Supplier Relationship Query counts the number of suppliers who can supply parts that satisfy a particular customer's requirements. The customer is interested in parts of eight different sizes as long as they are not of a given type, not of a given brand, and not from a supplier who has had complaints registered at the Better Business Bureau. Results must be presented in descending count and ascending brand, type, and size.

Query 16 Functional definition:

```
select
    p_brand,
    p_type,
    p_size,
    count(distinct ps_supkey) as supplier_cnt
from
    partsupp,
    part
where
    p_partkey = ps_partkey
    and p_brand <> 'Brand#45'
    and p_type not like 'MEDIUM POLISHED%'
    and p_size in (49, 14, 23, 45, 19, 3, 36, 9)
    and ps_supkey not in (
        select
            s_supkey
        from
            supplier
        where
            s_comment like '%Customer%Complaints%'
    )
group by
    p_brand,
    p_type,
    p_size
order by
    supplier_cnt desc,
    p_brand,
    p_type,
    p_size;
```

Query 17 Business Question: The Small-Quantity-Order Revenue Query considers parts of a given brand and with a given container type and determines the average lineitem quantity of such parts ordered for all orders (past and pending) in the 7-year database. What would be the average yearly gross (undiscounted) loss in revenue if orders for these parts with a quantity of less than 20% of this average were no longer taken?

Query 17 Functional definition:

```
select
    sum(l_extendedprice) / 7.0 as avg_yearly
from
    lineitem,
    part
where
    p_partkey = l_partkey
    and p_brand = 'Brand#23'
    and p_container = 'MED BOX'
    and l_quantity < (
        select
            0.2 * avg(l_quantity)
        from
            lineitem
        where
            l_partkey = p_partkey
    );
```

Query 18 Business Question: The Large Volume Customer Query finds a list of the top 100 customers who have ever placed large quantity orders. The query lists the customer name, customer key, the order key, date and total price and the quantity for the order.

Query 18 Functional definition:

```
select
    c_name,
    c_custkey,
    o_orderkey,
    o_orderdate,
    o_totalprice,
    sum(l_quantity)
from
    customer,
    orders,
    lineitem
where
    o_orderkey in (
        select
            l_orderkey
        from
            lineitem
        group by
            l_orderkey having
                sum(l_quantity) > 300
    )
    and c_custkey = o_custkey
    and o_orderkey = l_orderkey
group by
    c_name,
    c_custkey,
    o_orderkey,
    o_orderdate,
    o_totalprice
order by
    o_totalprice desc,
    o_orderdate
limit 100;
```

Query 19 Business Question: The Discounted Revenue query finds the gross discounted revenue for all orders for three different types of parts that were shipped by air and delivered in person. Parts are selected based on the combination of specific brands, a list of containers, and a range of sizes.

Query 19 Functional definition:

```
select
    sum(l_extendedprice* (1 - l_discount)) as revenue
from
    lineitem,
    part
where
    (
        p_partkey = l_partkey
        and p_brand = 'Brand#12'
        and p_container in ('SM CASE', 'SM BOX', 'SM PACK', 'SM PKG')
        and l_quantity >= 1 and l_quantity <= 1 + 10
        and p_size between 1 and 5
        and l_shipmode in ('AIR', 'AIR REG')
        and l_shipinstruct = 'DELIVER IN PERSON'
    )
    or
    (
        p_partkey = l_partkey
        and p_brand = 'Brand#23'
        and p_container in ('MED BAG', 'MED BOX', 'MED PKG', 'MED PACK')
        and l_quantity >= 10 and l_quantity <= 10 + 10
        and p_size between 1 and 10
        and l_shipmode in ('AIR', 'AIR REG')
        and l_shipinstruct = 'DELIVER IN PERSON'
    )
    or
    (
        p_partkey = l_partkey
        and p_brand = 'Brand#34'
        and p_container in ('LG CASE', 'LG BOX', 'LG PACK', 'LG PKG')
        and l_quantity >= 20 and l_quantity <= 20 + 10
        and p_size between 1 and 15
        and l_shipmode in ('AIR', 'AIR REG')
        and l_shipinstruct = 'DELIVER IN PERSON'
    );
```



Query 20 Business Question: The Potential Part Promotion query identifies suppliers who have an excess of a given part available; an excess is defined to be more than 50% of the parts like the given part that the supplier shipped in a given year for a given nation. Only parts whose names share a certain naming convention are considered.

Query 20 Functional definition:

```
select
    s_name,
    s_address
from
    supplier,
    nation
where
    s_suppkey in (
        select
            ps_suppkey
        from
            partsupp
        where
            ps_partkey in (
                select
                    p_partkey
                from
                    part
                where
                    p_name like 'forest%'
            )
        and ps_availqty > (
            select
                0.5 * sum(l_quantity)
            from
                lineitem
            where
                l_partkey = ps_partkey
                and l_suppkey = ps_suppkey
                and l_shipdate >= date '1994-01-01'
                and l_shipdate < date '1994-01-01' + interval '1' year
        )
    )
    and s_nationkey = n_nationkey
    and n_name = 'CANADA'
order by
    s_name;
```

Query 21 Business Question: The Suppliers Who Kept Orders Waiting query identifies suppliers, for a given nation, whose product was part of a multi-supplier order (with current status of 'F') where they were the only supplier who failed to meet the committed delivery date.

Query 21 Functional definition:

```
select
    s_name,
    count(*) as numwait
from
    supplier,
    lineitem l1,
    orders,
    nation
where
    s_suppkey = l1.l_suppkey
    and o_orderkey = l1.l_orderkey
    and o_orderstatus = 'F'
    and l1.l_receiptdate > l1.l_commitdate
    and exists (
        select
            *
        from
            lineitem l2
        where
            l2.l_orderkey = l1.l_orderkey
            and l2.l_suppkey <> l1.l_suppkey
    )
    and not exists (
        select
            *
        from
            lineitem l3
        where
            l3.l_orderkey = l1.l_orderkey
            and l3.l_suppkey <> l1.l_suppkey
            and l3.l_receiptdate > l3.l_commitdate
    )
    and s_nationkey = n_nationkey
    and n_name = 'SAUDI ARABIA'
group by
    s_name
order by
    numwait desc,
    s_name
limit 100;
```

Query 22 Business Question: this query counts how many customers within a specific range of country codes have not placed orders for 7 years but who have a greater than average “positive” account balance. It also reflects the magnitude of that balance. Country code is defined as the first two characters of c\_phone.

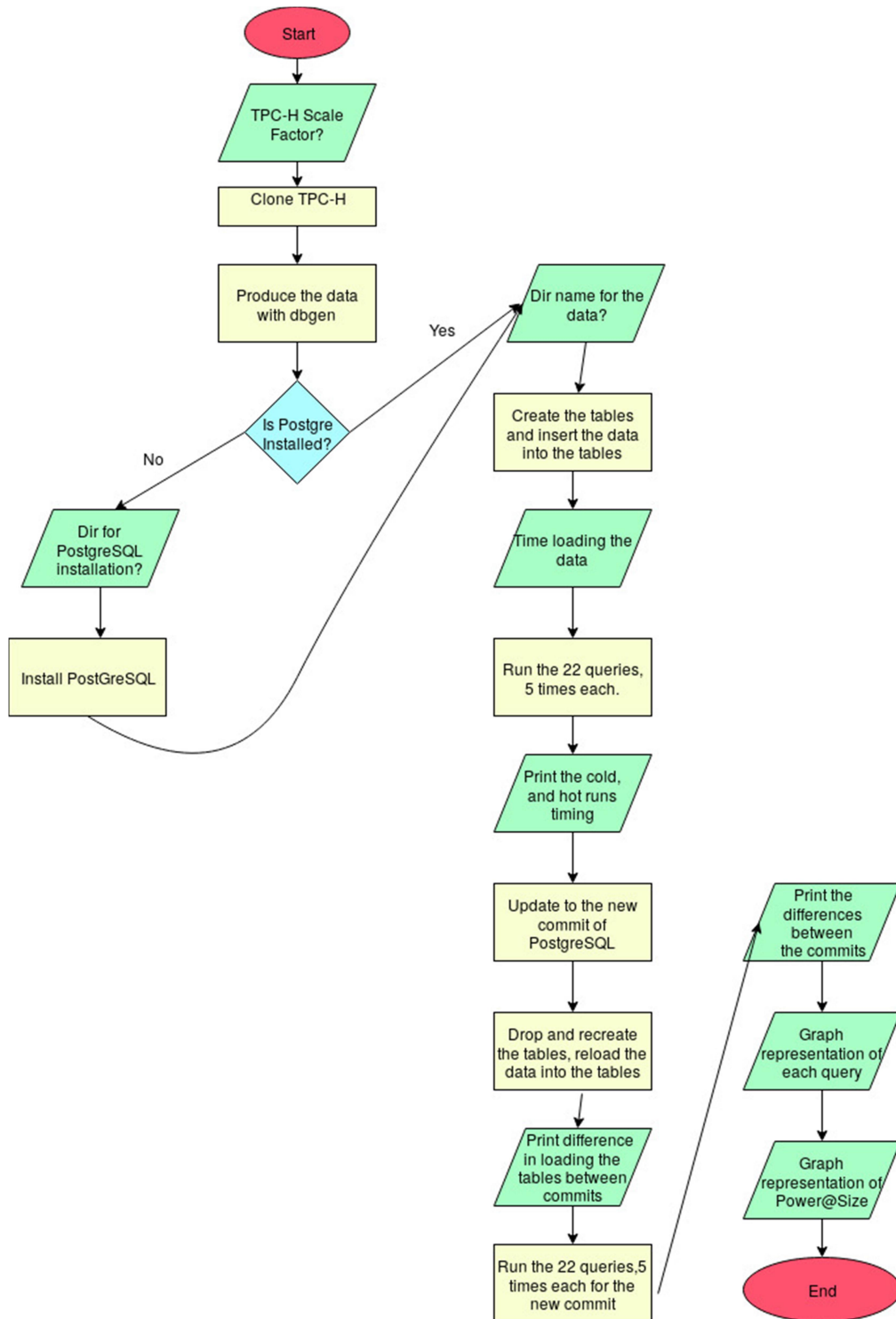
Query 22 Functional definition:

```
select
    centrycode,
    count(*) as numcust,
    sum(c_acctbal) as totacctbal
from
    (
        select
            substring(c_phone from 1 for 2) as centrycode,
            c_acctbal
        from
            customer
        where
            substring(c_phone from 1 for 2) in
                ('13', '31', '23', '29', '30', '18', '17')
            and c_acctbal > (
                select
                    avg(c_acctbal)
                from
                    customer
                where
                    c_acctbal > 0.00
                    and substring(c_phone from 1 for 2) in
                        ('13', '31', '23', '29', '30', '18', '17')
                )
            and not exists (
                select
                    *
                from
                    orders
                where
                    o_custkey = c_custkey
            )
        ) as custsale
group by
    centrycode
order by
    centrycode;
```

## Appendix C

### Flowcharts of the tools

Tool 1:



## Tool 2:

