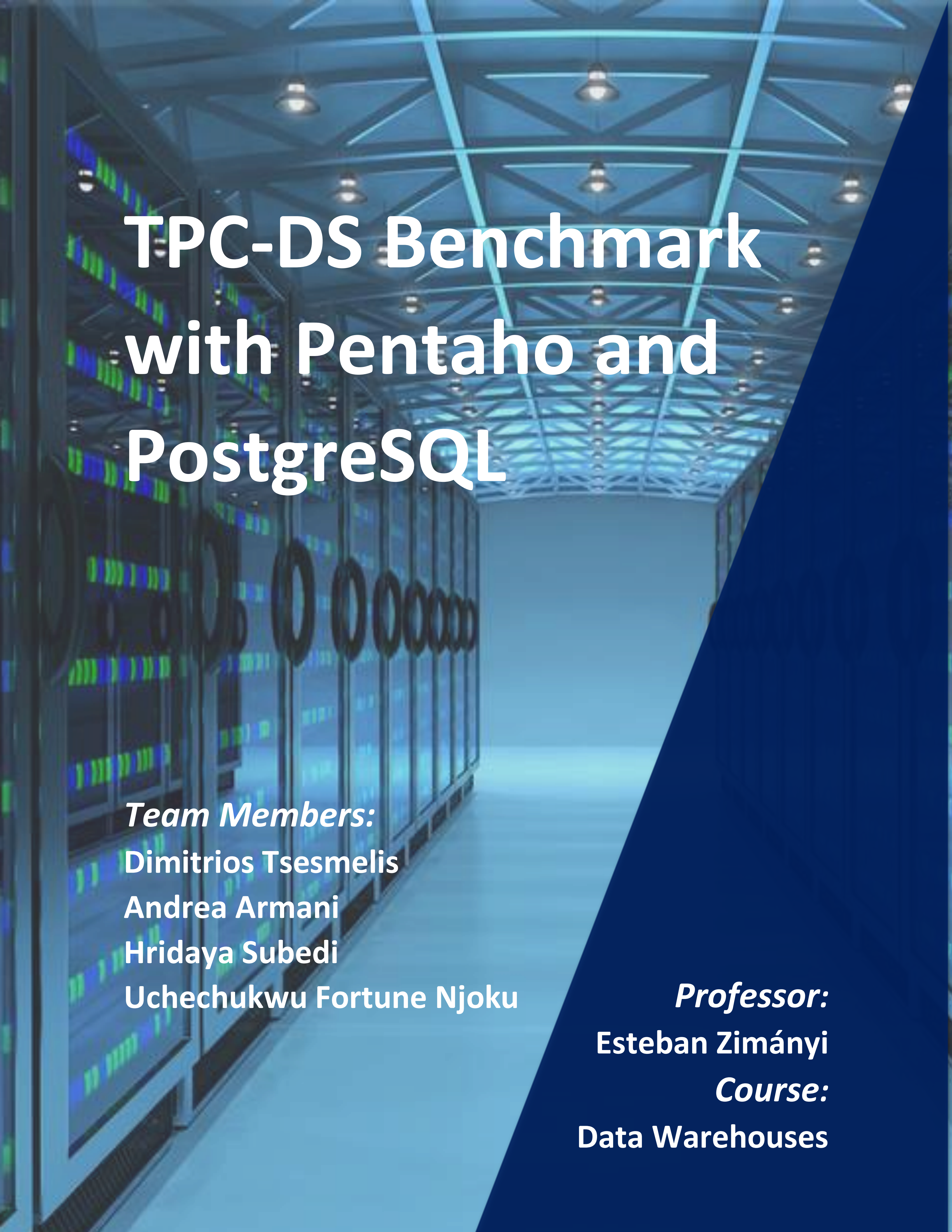


TPC-DS Benchmark with Pentaho and PostgreSQL

Team Members:

Dimitrios Tsesmelis

Andrea Armani

Hridaya Subedi

Uchechukwu Fortune Njoku

Professor:

Esteban Zimányi

Course:

Data Warehouses

Abstract

In our work we try to present concisely and comprehensively the basic features of the TPC-DS Benchmark, as well as our implementation using PostgreSQL server and Pentaho services. We performed multiple performance tests that can be found in chapter 3. The goal of these tests was to compare the execution time of the TPC-DS queries, by using only PostegreSQL server and a combination of PostegreSQL server and Pentaho Data Integration (Pentaho DI) services.

Considering the SQL queries produced by the provided software, we were challenged to translate them in PostgreSQL dialect, as the TPC-DS software does not export a compatible dialect. This process is explained in detail in chapter 1.6.

Finally, we note that our work is not a complete implementation of a TPC-DS Benchmark, as this is a very demanding and time-consuming job usually made by real RDBMS vendors. However, we tried to understand the whole phases of such a Benchmark, present them and materialize those that allow us to get the most import insights.

Contents

1) TPC-DS

1.1 Overview

The TPC Benchmark™DS (TPC-DS) is a decision support benchmark that models several generally applicable aspects of a decision support system, including queries and data maintenance. The benchmark provides a representative evaluation of the System Under Test's (SUT) performance as a general-purpose decision support system.

This benchmark illustrates decision support systems that:

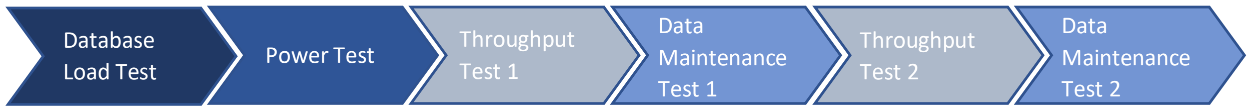
- Examine large volumes of data;
- Give answers to real-world business questions;
- Execute queries of various operational requirements and complexities (e.g., ad-hoc, reporting, iterative OLAP, data mining);
- Are characterized by high CPU and IO load;
- Are periodically synchronized with source OLTP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

A benchmark result measures query response time in single user mode, query throughput in multi user mode and data maintenance performance for a given hardware, operating system, and data processing system configuration under a controlled, complex, multi-user decision support workload.

All in all, a TPC-DS Benchmark could be summarized in the following sequence of tests:

1. Database Load Test
2. Power Test
3. Throughput Test 1
4. Data maintenance Test 1
5. Throughput Test 2
6. Data maintenance Test 2

The first test is the process of building the database in which the benchmark will be performed. The Power Test measures the ability of the system to process a sequence of queries in the least amount of time in a single stream fashion. The Throughput Tests measure the ability of the system to process the most queries in the least amount of time with multiple users. Finally, the Data Maintenance tests refresh the database, by performing updates (inserts and deletes) to the dataset that was loaded during the Load Test.



After the end of the execution of the above tests, we need some metrics in order to access our results. TPC-DS defines three primary metrics:

a) A Performance Metric, **QphDS@SF**, reflecting the TPC-DS query throughput.
b) A Price-Performance metric, **$/QphDS@SF**.
c) System availability date.

The primary performance metric of the benchmark is QphDS@SF, defined as:

$$QphDS@SF = \left\lfloor \frac{SF * Q}{\sqrt[4]{T_{PT} * T_{TT} * T_{DM} * T_{LD}}} \right\rfloor$$

Where:

- SF is based on the scale factor used in the benchmark.
- Q is the total number of weighted queries: $Q=S_q*99$, with S_q being the number of streams executed in a Throughput Test. A stream is defined as the sequential execution of a permutation of queries submitted by a single emulated user. A stream consists of the 99 queries.
- $T_{PT}=T_{Power}*S_q$, where T_{Power} is the total elapsed time to complete the Power Test.
- $T_{TT}= T_{TT1}+T_{TT2}$, where T_{TT1} is the total elapsed time of Throughput Test 1 and T_{TT2} is the total elapsed time of Throughput Test 2.
- $T_{DM}= T_{DM1}+T_{DM2}$, where T_{DM1} is the total elapsed time of Data Maintenance Test 1 and T_{DM2} is the total elapsed time of Data Maintenance Test 2.
- T_{LD} is the load factor computed as $T_{LD}=0.01*S_q*T_{Load}$, where T_{Load} is the time to finish the load.
- T_{PT}, T_{TT}, T_{DM} and T_{LD} quantities are in units of decimal hours with a resolution of at least 1/3600th of an hour.

The price-performance metric for the benchmark is defined as:

$$\$/QphDS@SF = \frac{P}{QphDS@SF}$$

Where:

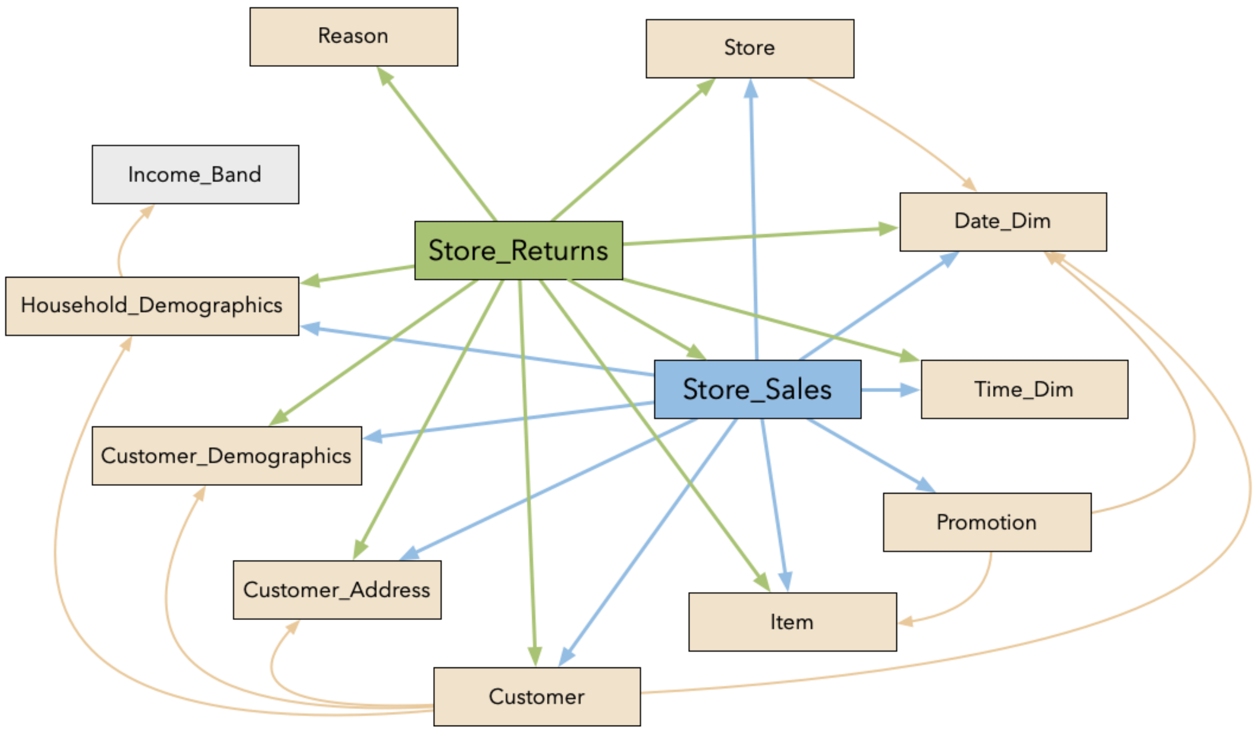
- P is the price of the Priced System.
- QphDS@SF is the reported performance metric.

1.2 Schema

The TPC-DS models a real data warehouse and focuses on Online Analytical Processing Tasks (OLAP). It provides a complete database that simulates the decision support functions of a retail product supplier so that users can relate intuitively to the components of the benchmark.

The schema of the database is not build using a 3rd Normal Form (3NF) schema. However, the TPC-DS implements a multiple snowflake schema, which is a hybrid approach between a 3NF and a pure Star schema. This schema consists of Fact and Dimension tables. Fact tables (ex: Store_Returns) are used to represent the measures of a database, while Dimension tables (ex: Store) provide descriptive information for all the measurements recorded in Fact tables.

The *Create* statements for the database exist in the directory *v2.11.0rc2/tools* of the provided software, inside the folders *tpcds.sql* and *tpcds_ri.sql.*



Graphical representation of two Fact and some Dimension tables of the database.

1.3 Data Population

The created databases should now be populated with data. The TPC-DS has a tool that automatically generates different size of datasets (scale factor). It is worth to mention here that the Fact tables have Tuple scaling, which means that their tuple number is linearly growing as the scale factor grows, while the Dimension tables are Domain scaling as the size of these tables is sub-linearly growing.

The **dsdgen** program produces 25 .csv files (1 file for each table) that contain the data to be populated into the database.

Command to be executed:

```
./dsdgen -scale 4 -dir ./produced_dataset/
```

In every file, the content of each field is terminated by '|'. A '|' in the first position of a row indicates that the first column of the row is empty. Two consecutive '|' indicate that the given column value is empty. Empty column values are only generated for columns that are NULL-able as specified in the logical database design. Empty column values, as generated by dsdgen, must be treated as NULL values in the data processing system.

In our work, we use the **COPY (*[field_names]*) FROM *[path]* DELIMITER *[delimiter]*** to insert the data from the csv files to our PostgreSQL Server. All the insert scripts that were used during the Load Test can be found in the path *src/insert_dataset_to_db.SQL*. We note that before running the scripts to the server, we first removed the last delimiter character '|' from every row of every csv file.

1.4 Data Qualification

After the dataset is loaded to our RDBMS, we need to validate the consistency of our data, by ensuring that the dataset in our database is identical to the one produced by dsdgen. For this reason, we need to make a Data Qualification test. In this phase, we are going to use the **Qualification database**. The intent is that the functionality exercised by running the validation queries against the qualification database be the same as that exercised against the test database during the performance test. To this end, the qualification database must be identical to the test database.

Our Data Qualification test can be summarized in the below steps:

1) Create the Qualification Database, which has the same structure as the Test Database and contains only 1GB of data.

2) Modify every template as it is described in the Appendix B chapter of the specification document. We changed every parameter with the right one. The produced templated can be found in *v2.11.0rc2/qualification_query_templates/* directory.

3) Run the dsqgen program with the parameter -QUALIFY Y in order to get the right permutation of the queries.

Command to be executed:

```
./dsqgen -input../qualification_query_templates/templates.lst -directory
../qualification_query_templates/ -dialect ../query_templates/netezza -output_dir
../produced_query/ -QUALIFY Y
```

The above program produced a flat file that contains all the 99 sql queries that will be used for the validation phase.

4) We execute every one of these queries and we save each output to a file named "queryX".

5) We compare every file we produced in the previous step with the equivalent file that exists in the *v2.11.0rc2/answer_sets* directory. Every pair of files must be the same so that the Qualification Test is successful.

1.5 Query overview

As we have already mentioned, the TPC-DS provides some queries that should be executed to access the performance of our system. These queries simulate some real reporting queries, ad-hoc queries, iterative OLAP queries and data mining queries. Consequently, we need many different types of queries, the functional definition of which is defined by the query templates (*v2.11.0rc2/query_templates*).

Dsqgen translates the query templates into fully functional SQL, which is known as executable query text (EQT). The source code for dsqgen is provided as part of the electronically downloadable portion of TPC-DS.

The program takes as a parameter the dialect (-DIALECT flag) which determines to which SQL version the output will be. The available versions are:

- Db2
- Oracle
- Sqlserver
- Netezza

We are using PostgreSQL and the most similar language is Netezza, so we first create the 99 queries using Netezza as dialect and then we translate every query in PostgreSQL, as it is described in chapter 1.6.

Command to be executed:

```
./dsqgen -input ../query_templates/templates.lst -directory ../query_templates -dialect
../query_templates/netezza -scale 1 -output_dir ../produced_query/
```

1.6 Query translation to PostgreSQL

In order to automate the process of translation of the queries to PostgreSQL, we created a new set of templates that can be found in *v2.11.0rc2/query_templates* (the original templates are in *v2.11.0rc2/ query_templates_original* directory). In this way, each time we need to create a new file containing the 99 queries, we should not make every change to the output file, as the output is generated using the correct templates that produce queries in PostgreSQL.

The changes we made are listed below:

1) We slightly modified every template that subtracted or added days. This change was applied to templates 5, 12, 16, 20, 21, 32, 37, 40, 77, 80, 82, 92, 94, 95, and 98.
   Old: `and (cast('2000-02-28' as date) + 30 days)`
   New: `and (cast('2000-02-28' as date) + '30 days'::interval )`

2) Change the queries that used alias tables in ORDER BY clause. This change was applied to templates 36, 70 and 86.
   Old:
```
SELECT   Sum(ss_net_profit) AS total_sum ,
        s_state ,
        s_county ,
        Grouping(s_state)+Grouping(s_county) AS lochierarchy ,
        Rank() OVER ( partition BY Grouping(s_state)+Grouping(s_county),
        CASE
                WHEN Grouping(s_county) = 0 THEN s_state
        END ORDER BY Sum(ss_net_profit) DESC) AS rank_within_parent
…
…
…
GROUP BY rollup(s_state,s_county)
ORDER BY lochierarchy DESC ,
        CASE
                WHEN lochierarchy = 0 THEN s_state
        END ,
        rank_within_parent limit 100
```
   New:

```
SELECT   * --added                                                        line
FROM    (                              --added                            line

            SELECT   Sum(ss_net_profit) AS total_sum ,
                 s_state ,
                 s_county ,
                 Grouping(s_state)+Grouping(s_county) AS lochierarchy ,
                 Rank() OVER ( partition BY Grouping(s_state)+Grouping(s_county),
                 CASE
                       WHEN Grouping(s_county) = 0 THEN s_state
                 END ORDER BY Sum(ss_net_profit) DESC) AS rank_within_parent
…
…
…
)                                  --added                                line
GROUP BY rollup(s_state,s_county) ) AS sub
ORDER BY lochierarchy DESC ,
      CASE
            WHEN lochierarchy = 0 THEN s_state
      END ,
      rank_within_parent limit 100;
```

3) Query templates were modified to exclude columns not found in the query. This change was applied to template 30 (c_last_review_date_sk and ctr_total_return does not exist and are not used in the query).

4) In some template we had to add table aliases to some sub-queries. This change was applied to template: 2, 14, and 23.

   Old:

```
WITH wscs AS
(
    SELECT sold_date_sk ,
        sales_price
    FROM  (
            SELECT ws_sold_date_sk    sold_date_sk ,
                ws_ext_sales_price sales_price
            FROM   web_sales
            UNION ALL
            SELECT cs_sold_date_sk    sold_date_sk ,
                cs_ext_sales_price sales_price
            FROM   catalog_sales)),
…
```

   New:

```
WITH wscs AS
(
    SELECT sold_date_sk ,
        sales_price
    FROM  (
            SELECT ws_sold_date_sk    sold_date_sk ,
                ws_ext_sales_price sales_price
            FROM   web_sales
            UNION ALL
            SELECT cs_sold_date_sk    sold_date_sk ,
                cs_ext_sales_price sales_price
            FROM   catalog_sales) AS a),
…
```

1.7 Data maintenance

Data maintenance operations are performed as part of the benchmark execution. TPC-DS recognizes these operations as an integral part of the data warehouse lifecycle. The update dataset is generated by dsqgen program, which produces some flat files that represent the required information.

Data maintenance functions perform insert and delete operations that are defined in pseudo code. Depending on which operation they perform, and on which type of table, they are categorized as Method1 through Method3:

1. Method 1: Fact Table Load

   The below pseudo code assures that whenever a new record is going to be inserted to a Fact table, we will replace the Business keys (bkc) of this record with the correspondent surrogate keys of each Dimension tables.

```
for every row v in view V corresponding to fact table F:
  get row v into local variable lv
  for every type 1 business key column bkc in v:
    get row d from dimension table D corresponding to bkc
        where the business keys of v and d are equal
              update bkc of lv with surrogate key of d
  end for
  for every type 2 business key column bkc in v
    get row d from dimension table D corresponding to bkc
      where the business keys of v and d are equal and rec_end_date is NULL
    update bkc of lv with surrogate key of d
  end for
  insert lv into F
end for
```

2. Method 2: Sales and Returns Fact Table Delete

```
Delete rows from R with corresponding rows in S where d_date between Date1 and Date2
Delete rows from S where d_date between Date1 and Date2
```

D_date is a column of the date_dim dimension. D_date has to be obtained by joining to the date_dim dimension on sales date surrogate key. The sales date surrogate key for the store sales is ss_sold_date_sk, for catalog it is cs_sold_date_sk and for web sales it is ws_sold_date_sk

3. Method 3: Inventory Fact Table Delete

```
Delete rows from I where d_date between Date1 and Date2
```

D_date is a column of the date_dim dimension. D_date has to be obtained by joining to the date_dim dimension on inv_date_sk.

The data maintenance functions must be implemented in SQL or procedural SQL and they should respect the following rules.

Each data maintenance function inserting or updating rows in dimension and fact tables is defined by the following components:

a) Descriptor, indicating the name of the data maintenance function in the form of DM_<abbreviation of data warehouse table> for dimensions and LF_<abbreviation of the data warehouse fact table> for fact tables. The extension indicates the data warehouse table that is populated with this data maintenance function.
b) The data maintenance method describes the pseudo code of the data maintenance function.
c) A SQL view V describing which tables of the source schema need to be joined to obtain the correct rows to be loaded.
d) The column mapping defining which source schema columns map to which data warehouse columns.

Each data maintenance function deleting rows from fact tables is defined by the following components:

1. Descriptor, indicating the name of the data maintenance function in the form of DF_<abbreviation of data warehouse fact table>. The extension indicates the data warehouse fact table from which rows are deleted.
2. Tables: S and R, or I in case of inventory.
3. Two dates: Date1 and Date2.
4. The data maintenance method indicates how data is deleted.

1.8 Executing the queries using Batch file

We created a .bat file in order to be able to execute the queries in batches, without using the PgAdmin. This .bat file can be found in the directory data/run_queries.bat.

Firstly, the use needs to config the variables that exist in the first lines:

- PGHOST: The host of our database.
- PGDATABASE: The database name.
- PGUSER: The username.
- PGPASSWORD: The user's password

It is also required that the user will specify the directory in which the queries he/she wants to execute are placed (line 14: C:\Users\Alex\Desktop\TPC-DS\data\queries_to_execute\).

This program reads every query that exists in the specified directory, executes it using psql command and returns the corresponding result and the execution time in a file in the directory data/outputs (e.g.: for the query_1.sql file, it will create the file output_query_1).

2) Pentaho Services

2.1 Overview

Pentaho is a Business Intelligence suite developed by Hitachi Vantara which provides a combination of data integration, OLAP, reporting, information dashboards, data mining and ETL solutions. The software is available in community and enterprise editions. For the TPC-DS benchmark, we used the community editions and briefly describe some of the components below.

Pentaho Business Analytics Platform makes the core of the suite, connecting the other components including plug-ins. It has features that secure the suite as demonstrated by the login requirement.

Pentaho Analysis Services (Mondrian) is the online analytical processing (OLAP) server which supports multidimensional expressions (MDX) query language, XML for analysis and olap4j (open Java API for OLAP) interface specifications.

Pentaho Data Mining searches for patterns in data by using Waikato Environment for Knowledge Analysis (WEKA) which consists of machine learning algorithms for various data mining tasks.

Pentaho Data Integration (PDI) sometimes referred to as kettle is made up of the data integration engine for extraction, transformation and loading (ETL) data and a GUI interface called Spoon for defining data integration jobs and transformation.

Pentaho Report Designer is a report writer using sub reports, charts and graphs. It consists of a core reporting engine, capable of generating reports based on an XML definition file. It can query and use data from many sources including SQL and MDX.

Pentaho Schema Workbench provides a graphical interface for designing OLAP cubes. This is helpful for inspecting a cube visually to determine how to maintain or expand it for better analysis.

Pentaho Aggregation Designer generates precalculated, aggregated answers to speed up analysis work and MDX queries executed against Mondrian. It operates on Mondrian XML schema files and the database with the underlying tables described by the schema.

Pentaho Metadata Editor is used to build business models and act as an abstraction layer from the underlying data sources.

Pentaho Data Access Wizard allows users to create new data sources for use throughout the system from other databases or CSV files uploaded to the server while using a setup wizard.

Other tools available solely in the enterprise edition include Pentaho Mobile, Pentaho Interactive Reporting (PIR), Pentaho Dashboard Designer (PDD) and Pentaho Analysis (Analyzer) (PAZ). There are several community driven plug-ins for achieving various things like the popular Saiku-reporting tool.

2.2 Our approach

In order to benchmark Pentaho as a data warehouse tool, we had PostgreSQL as our underlaying data source with the needed data loaded and executed the 99 SQL queries in Spoon, which is the Graphical User Interface (GUI) for defining query execution jobs and transformation in PDI.

A transformation is a network of logical tasks called steps. There are two main components associated with transformations:

1) Steps are the building blocks of a transformation. There are over 140 steps available in PDI e.g. insert, delete, CSV file input, execute SQL script and so on. These steps are grouped according to the function they perform for example; input output, scripting and so on. Each step in a transform is designed to perform the tasks the user requires. For our benchmarking of the Pentaho tool, we used the scripting group of steps.

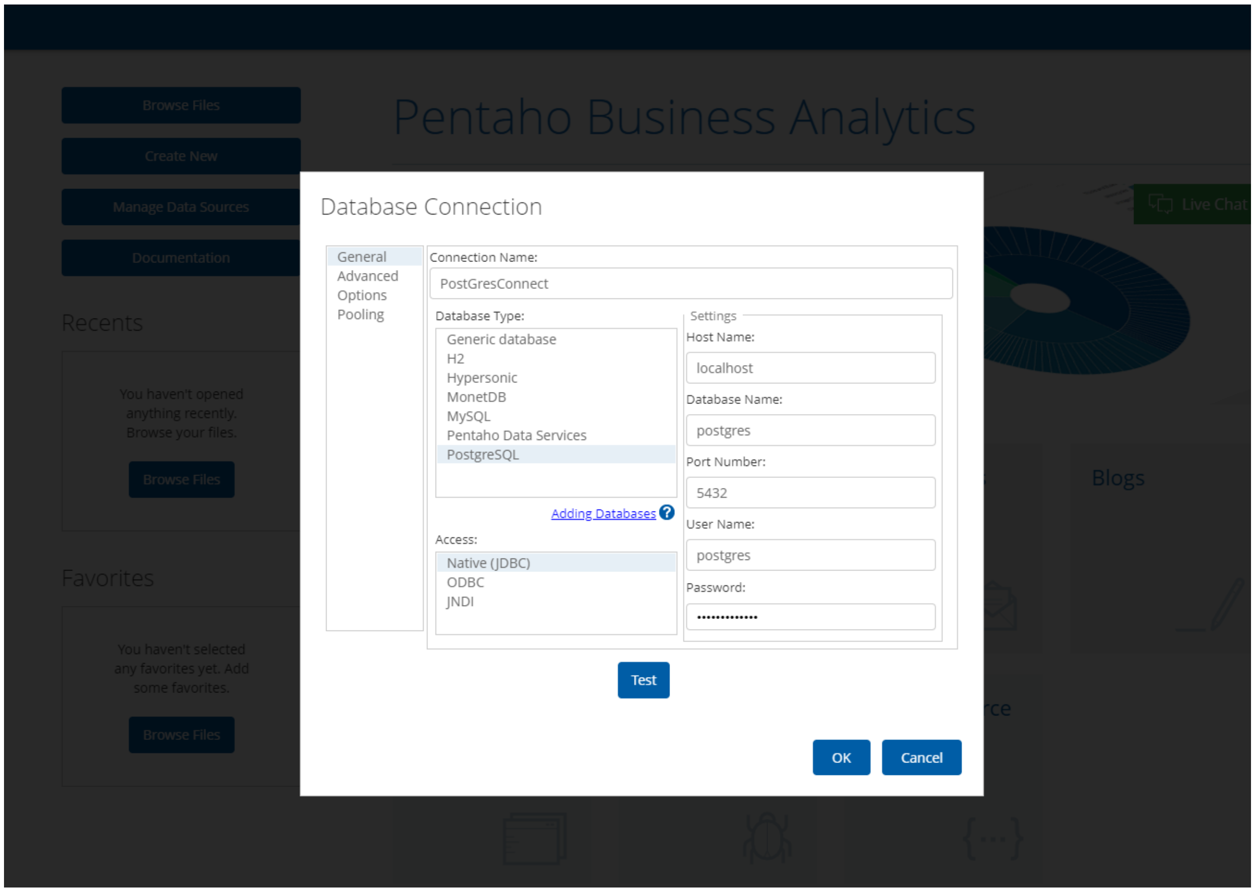
2) Hops are the data pathways that connect steps together and allows schema metadata to pass from one step to another. Hops determine the flow of data through the steps of a transformation. In Spoon, hops are represented as arrows.

After running transformations, the execution results panel, which appears at the bottom part of Spoon, contains several different tabs useful for analysing how the transformation ran. Particularly, the metrics tab displays a Gantt chat showing how long it takes to connect to a database, how much time it spent executing a SQL query, or how long it takes to load a transformation. For this project, our interest is how much time was spent executing the SQL queries.

Stepwise, we did the following to measure the execution time of the queries using Pentaho PDI and PostgreSQL server:

Step 1: Launch Pentaho server, by running the the shell script in the following path [pentaho_directory]/pentaho-server/start-pentaho.sh.

Step 2: Add Postgres database as a data source using the data access wizard.



Database Connection

Connection Name: PostgresConnect

Database Type: Generic database, H2, Hypersonic, MonetDB, MySQL, Pentaho Data Services, PostgreSQL

Access: Native (JDBC), ODBC, JNDI

Settings

Host Name: localhost

Database Name: postgres

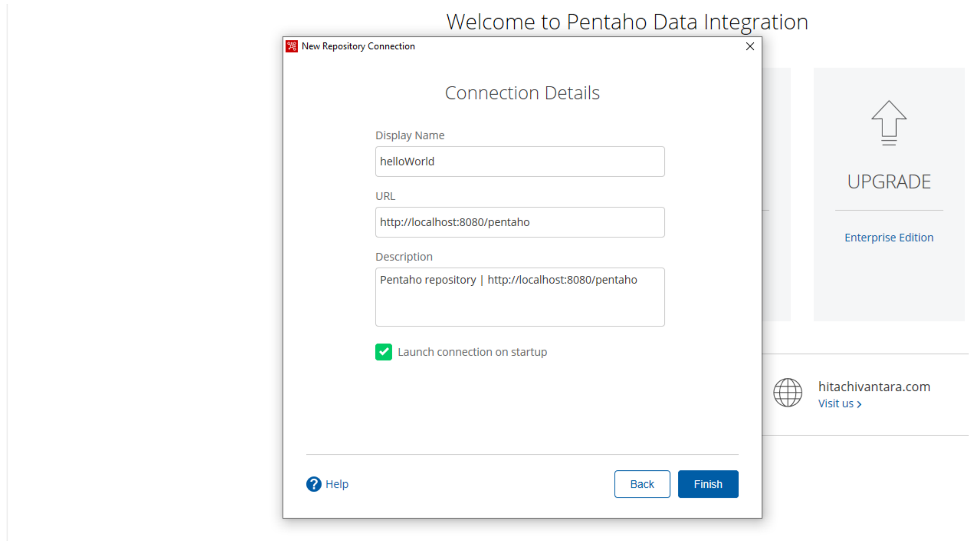
Port Number: 5432

User Name: postgres

Password: ••••••••••••••

Step 3: Launch Spoon, which is a component of Pentaho data integration, by running the batch file in the following path [data_integration_path]/data-integration/Spoon.bat.

<u>Step 4:</u> Connect Pentaho repository (i.e. Pentaho server).



Connection Details

Display Name: helloWorld

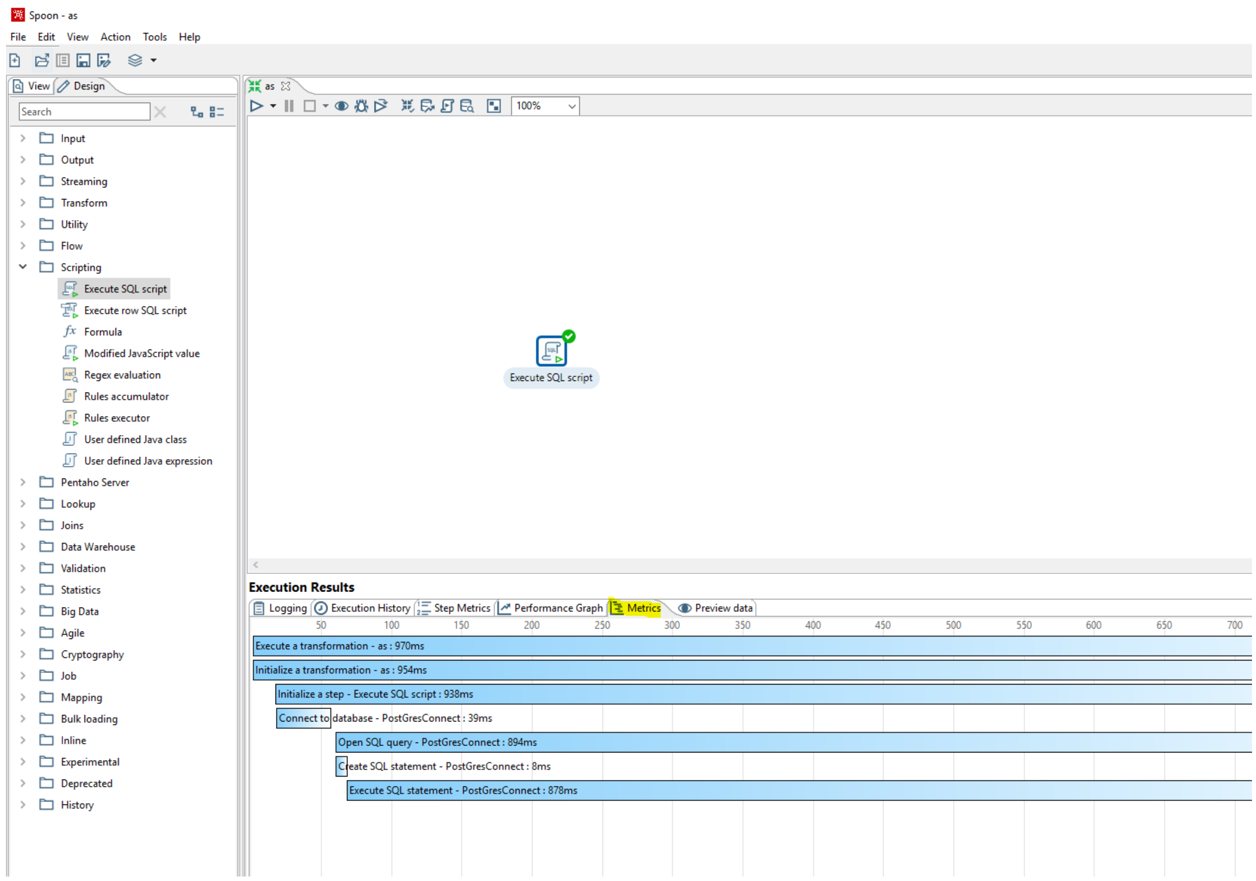
URL: http://localhost:8080/pentaho

Description: Pentaho repository | http://localhost:8080/pentaho

☑ Launch connection on startup

<u>Step 5:</u> Define a new job.

<u>Step 6:</u> Run job.



Execution Results

```
Execute a transformation - as : 970ms
Initialize a transformation - as : 954ms
Initialize a step - Execute SQL script : 938ms
Connect to database - PostGresConnect : 39ms
Open SQL query - PostGresConnect : 894ms
Create SQL statement - PostGresConnect : 8ms
Execute SQL statement - PostGresConnect : 878ms
```

3) Results

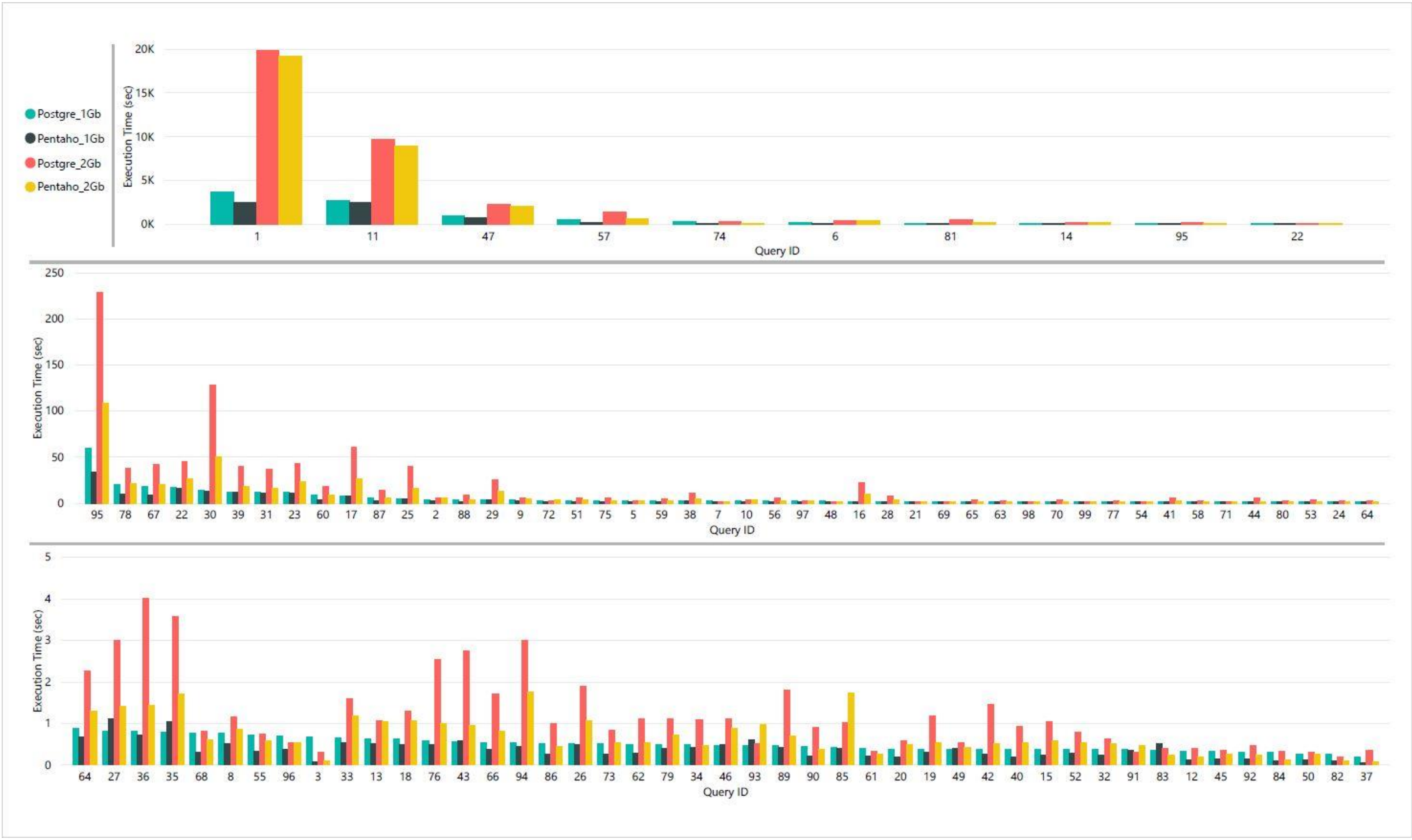
3.1 Graphical representation of our results

Our results consist of 3 bar charts. In each chart, we compare every query based on 4 different aspects, namely its execution time using PostgreSQL Server with 1gb of data, PostgreSQL server and Pentaho DI services with 1gb, PostgreSQL Server with 2gb and PostgreSQL server and Pentaho DI services with 2gb.

We are presenting the results in order, from the slowest to the fastest queries. The execution times of the query have different scales, as the maximum execution time is 3660 seconds while the minimum is less than 1 second. As a result, we are plotting the times in 3 different charts, **using different scales**, so that the visual representation is more precise.

The measurements were taken with our personal computer (Dell xps 15 9570), the hardware specifications of which are listed below:

- Processor: intel core i7-8750H CPU @ 2.2GHz 2.21GHz
- RAM: 16GB
- SSD: 500 GB



Queries execution time in seconds:

query	Postgre_1Gb	Postgre_2Gb	Pentaho_1Gb	Pentaho_2Gb
1	3660	19800	2460	19176.505
2	3.981	6.252	2.64	5.808
3	0.667	0.313	0.076	0.111
5	2.607	2.451	1.117	2.294
6	201	406	115.234	392.455
7	2.348	1.574	0.741	1.359
8	0.764	1.166	0.507	0.857
9	3.356	5.458	2.401	4.674
10	2.322	3.412	1.275	3.174
11	2763	9720	2507.32	8910
12	0.324	0.39	0.125	0.189
13	0.641	1.064	0.52	1.036
14	95	242	86.606	218.487
15	0.378	1.04	0.25	0.597

16	1.989	22.858	1.765	10.162
17	8.104	60.903	7.732	26.995
18	0.628	1.295	0.5	1.080
19	0.387	1.185	0.31	0.547
20	0.389	0.577	0.203	0.485
21	1.825	1.948	0.719	1.067
22	17.653	45.394	16.558	26.986
23	11.477	42.789	11.373	22.877
24	0.898	2.776	0.844	1.723
25	5.023	40.322	4.749	16.584
26	0.521	1.907	0.484	1.078
27	0.826	3.012	1.124	1.414
28	1.857	8.287	1.953	3.702
29	3.515	25.214	3.546	12.849
30	13.912	128	12.653	50.512
31	11.593	36.426	10.67	15.935
32	0.372	0.624	0.25	0.513
33	0.647	1.6	0.547	1.19
34	0.491	1.091	0.422	0.474
35	0.784	3.567	1.036	1.706
36	0.808	4.018	0.734	1.443
37	0.204	0.344	0.062	0.09
38	2.409	11.209	2.327	4.994
39	12.078	40.455	12.033	18.468
40	0.38	0.936	0.204	0.545
41	1.155	5.32	0.984	2.4
42	0.381	1.47	0.264	0.506
43	0.568	2.749	0.578	0.96
44	1.104	5.49	1.14	1.968
45	0.322	0.362	0.157	0.274
46	0.477	1.114	0.5	0.895
47	1016	2286	800.923	2041.371
48	2.124	1.702	1.001	1.362
49	0.384	0.529	0.391	0.432
50	0.271	0.306	0.125	0.273
51	2.793	5.863	1.937	3.43
52	0.378	0.787	0.281	0.545
53	0.924	3.689	0.625	1.518
54	1.19	1.457	0.625	1.282
55	0.718	0.755	0.328	0.594
56	2.249	5.583	1.312	2.752

57	527	1434	197.044	626.778
58	1.15	3.107	0.609	1.269
59	2.572	4.981	1.203	2.619
60	8.481	18.188	4.059	9.212
61	0.399	0.335	0.222	0.267
62	0.504	1.114	0.282	0.537
63	1.725	2.232	0.688	1.512
64	0.896	2.256	0.672	1.31
65	1.776	3.894	0.999	2.071
66	0.551	1.714	0.375	0.811
67	18.115	41.664	8.967	20.247
68	0.781	0.81	0.312	0.617
69	1.798	1.835	0.703	1.114
70	1.619	3.184	0.796	1.396
71	1.132	1.569	0.484	0.833
72	2.855	2.813	0.641	3.252
73	0.513	0.845	0.265	0.529
74	285	375	115.927	151.68
75	2.731	5.5	1.516	2.789
76	0.589	2.545	0.484	0.99
77	1.453	2.542	0.75	1.332
78	20.378	37.733	9.826	21.146
79	0.504	1.114	0.406	0.718
80	1.052	2.329	0.688	1.534
81	128	538	56.237	228.679
82	0.266	0.204	0.094	0.11
83	0.347	0.393	0.515	0.25
84	0.298	0.321	0.109	0.125
85	0.43	1.017	0.41	1.747
86	0.527	0.989	0.266	0.438
87	5.523	14.506	2.577	5.783
88	3.835	8.916	2.015	3.585
89	0.459	1.799	0.422	0.7
90	0.457	0.916	0.218	0.38
91	0.369	0.314	0.344	0.459
92	0.306	0.469	0.156	0.245
93	0.463	0.506	0.609	0.969
94	0.529	2.99	0.438	1.766
95	60.02	229	33.69	108.39
96	0.693	0.544	0.375	0.547
97	2.137	2.386	1.14	2.422

98	1.693	0.642	0.312	0.516
99	1.489	1.143	0.5	1.093

3.2 Comments

3.2.1 Slowest queries

From the above bar charts, we can observe that the slowest queries are query_1 and query_11. Regarding query_11 (can be found in the directory data/queries_to_execute/query11.sql), the fact that the table *year_total* is joined with its self 4 times (*t_s_firstyear, t_s_secyear, t_w_firstyear and t_w_secyear),* makes its execution time higher.
Regarding query_1, the fact that we are self joining the table *customer_total_return* in the *where* clause, dramatically reduces the performance of the query.

It is obvious that self-joins are not performing well and they should be avoided in order to get good performance.

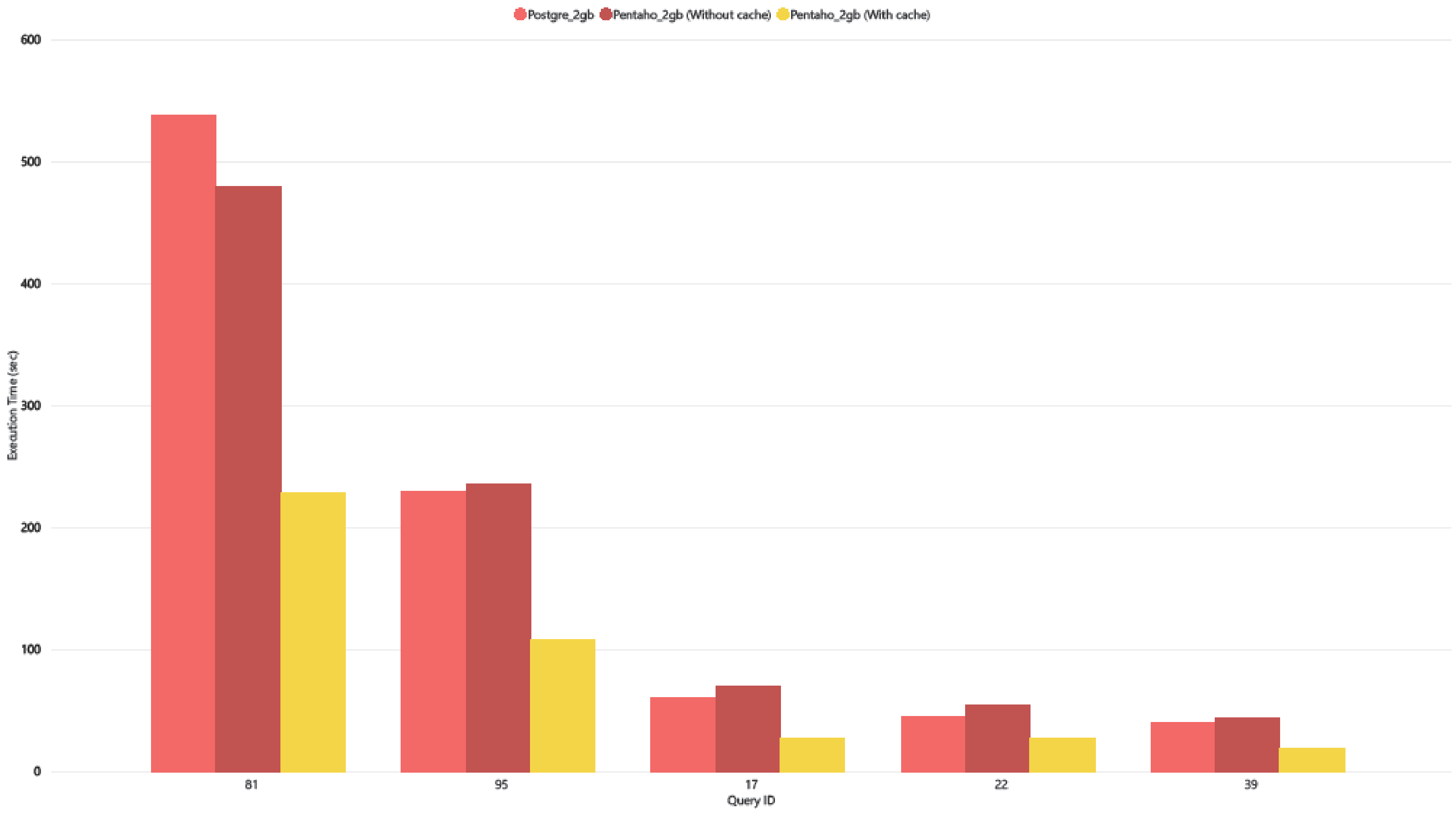
3.2.2 1gb execution time > 2gb execution time

Depending on the particular availability of resources during the execution of a query, the time is slightly affected. That could lead to paradoxical situation in which the same query results faster when launched with a bigger scale. For instance, query_96 is slower (0.693 sec) when launced for 1gb than 2gb (0.375).

3.2.3 Database Cache

Database cache is enabled by default in Pentaho. It improves the performance of the data service on subsequent follow-up queries. Running queries when data service results or some related results from the same table are cached, Pentaho Data Integration will fetch data from the cached dataset instead of querying against the database. In this project, we tabled all the execution time of queries in Pentaho DI when database cache was set to true, which resulted in less execution time than that of PostgreSQL Server execution time.

To prove this, we cleared our database cache, we ran a few selected queries many times and we measured the mean time. As we can observe in the below chart, the execution time is similar to that of PostgreSQL Server. Hence, we can optimize our execution time by enabling the Pentaho Service Cache and modifying the Cache duration as per our requirements.



Postgre_2gb
Pentaho_2gb (Without cache)
Pentaho_2gb (With cache)
600
500
400
300
200
100
0
Execution Time (sec)
81
95
17
22
39
Query ID

Conclusion

This report presents a TPC-DS Benchmark of PostgreSQL Server and Pentaho Services. We came to the conclusion that the benchmark of these technologies can be successfully performed in a local machine for scale factors up to 2gb. However, for bigger scale factors, it is vital to use Cloud technologies that provide more resources.

From the above results we see that the mean execution time of all the queries for 2gb of data is almost 3 times bigger than the mean execution time for 1gb of data. This means that in a local machine using PostgreSQL, we can not achieve a good scale, as the execution time is not linearly growing.

Finally, it is obvious that using a tool like Pentaho on the top of our PostgreSQL Server notably improves the results. Further works could benefit from the use of Analysis Services tools, like Pentaho Mondrian, to further increase the performance of the queries, due to the fact that most of them are making aggregations on the data. This can be achieved by creating hyper cubes that significantly speed up the aggregations.