UNIVERSITÉ LIBRE DE BRUXELLES

TPC-DS Benchmarking with Postgres Database

Data Warehouses (INFO-H413)

Erasmus Mundus Joint Master's Degree
in
Big Data Management and Analytics

by

Ahmad (ahmad@ulb.be)
Rishika Gupta (rishika.gupta@ulb.be)
Mir Wise Khan (mir.khan@ulb.be)
Chidiebere Ogbuchi (chidiebere.ogbuchi@ulb.be)

under the guidance of

Prof. Esteban Zimanyi

École polytechnique de Bruxelles
Université Libre de Bruxelles
October 2022

Contents

Abstract **i**

1 Introduction **1**
1.1 Overview . . . 1
1.2 Aim and Objectives . . . 1
1.3 Tools Used . . . 1
1.4 Limitations and Justifications . . . 2

2 Technology Fundamentals **3**
2.1 Postgres SQL . . . 3
2.1.1 Why PostgreSQL . . . 4
2.2 Other Tools . . . 4
2.2.1 Docker Desktop . . . 4
2.2.2 Visual Studio Code . . . 4
2.2.3 Python Interpreter . . . 5

3 Benchmarking and Implementation **6**
3.1 Introduction to Benchmarking . . . 6
3.2 TPC-DS . . . 6
3.3 Implementing TPC-DS on PostgreSQL . . . 7
3.3.1 Scaling and Database Population . . . 8
3.3.1.1 Scaling Model . . . 8
3.3.1.2 Test Database Scaling . . . 9
3.3.1.3 DSDGEN and Database Population . . . 9
3.3.2 Queries Overview . . . 10
3.3.2.1 Query Definition . . . 10
3.3.2.2 Query Modifications . . . 10
3.3.2.3 Query Ordering . . . 11
3.3.3 Query Execution and Optimisation . . . 11
3.3.3.1 Query Execution . . . 11
3.3.3.2 Optimisation . . . 11

4 Results and Discussions **12**
4.1 Benchmarking results . . . 12
4.1.1 Overall performance across all scale factors . . . 12
4.1.2 Individual performance across all scale factors . . . 13
4.1.3 Comparison of optimized and original queries . . . 15
4.2 Analysis of irregular pattern results . . . 15
4.2.1 Demystifying the performance variation on Query 04 . . . 15

4.2.2 Understanding the catalyst for queries with exponential run time . . . 18
4.3 Summary of Results . . . 20

5 Conclusion **21**

Bibliography **i**

A Appendix **iii**
A.1 Python Scripts . . . iii
A.2 Indexes . . . xi
A.3 Optimised Queries . . . xii

List of Figures

3.1 TPC-DS Process Diagram 7
3.2 Initial Query For Interval Syntax 10
3.3 Updated Query For Interval Syntax 10

4.1 Total Average Runtime (99 Queries) by Scale Factor 12
4.2 [Query 01-20] Average Runtime (Mins) by Scale Factor (SF) 13
4.3 [Query 21-40] Average Runtime (Mins) by Scale Factor (SF) 13
4.4 [Query 41-60] Average Runtime (Mins) by Scale Factor (SF) 13
4.5 [Query 61-80] Average Runtime (Mins) by Scale Factor (SF) 13
4.6 [Query 81-99] Average Runtime (Mins) by Scale Factor (SF) 14
4.7 Queries without exponential runtime (majority) 14
4.8 Optimised Queries Average Runtime (Mins) by Scale Factor (SF) . . 15
4.9 Original Queries Average Runtime (Mins) by Scale Factor (SF) . . . 15
4.10 Query 04 - Average Runtime (Mins) by Scale Factor (SF) 15
4.11 Query 04 - Query Plan 5 SF 16
4.12 Query 04 - Query Plan 15 SF 17
4.13 Query 04 - Query Statistics 5 SF 17
4.14 Query 04 - Query Statistics 15 SF 18
4.15 Exponential run-time detailed analysis for Q14, Q30, Q95 - All SFs . 19
4.16 Query 14 - Query Statistics 19
4.17 Query 23 - Query Statistics 19

List of Tables

1.1 Tools Used for TPC-DS Benchmarking with Postgres SQL 2

3.1 Local Machine Specifications 8
3.2 Implemented Scale Factors 8
3.3 Tuple Count Summary 9

Listings

A.1 Preprocessing_DBSetup_DataLoad Script iii
A.2 Query_Run_Test Script vi
A.3 Query_Performance_Test Script viii
A.4 Index_Setup Script xi
A.5 Query_04 xii
A.6 Query_04 xii
A.7 Query_06 xv
A.8 Query_11 xvi
A.9 Query_74 xviii
A.10 Query_81 xx

List of Abbreviations

CTE	Common Table Expressions
DB	Database
DBMS	Database Management Systems
DS	Decision Support
GB	Gigabytes
MS	Milli-seconds
OLAP	Online Analytical Processing
OLTP	Online Transaction Processing
ORDBMS	Object Relational Database Management System
RDBMS	Relational Database Management Systems
SF	Scale Factor
SQL	Structured Query Language
SUT	System Under Test
TPC	Transactional Processing Council
TPC-DS	Transactional Processing Council's Decision Support Benchmark
WSL	Windows Subsystem for Linux

Abstract

TPD-DS is a profound decision support benchmark presently developed by the Transaction Processing Performance Council (TPC). It comprises elements that can be utilized to evaluate a wide spectrum of implementation methodologies mapped to a conventional business setting. This report establishes a concept that briefly outlines the business modeling systems and performance aspects adopted into this benchmark. The TPC-DS provides decision support functions of a retail product supplier, as well as data loading, query generation and data maintenance. The database contains several snowflake schemas with common tabling dimensions with a huge bag of queries. In general, the benchmarks model very essential aspects of a typical decision support system which entails transformation of transactional data into business intelligence as well as synchronization and maintenance processes of data structures.

Chapter 1

Introduction

1.1 Overview

This project entails selecting a suitable Database Management System (DBMS) tool on which the TPC-DS benchmark will be implemented such as SQL Server Analysis Services, SparkSql, PostgreSQL, MariaDB, etc.

The benchmark is performed with several scale factors which relatively influence the data warehouse size. A reference scale factors is estimated, including other factors at different dimensions in order to evaluate its performance.

The project is performed in a groups of 4 persons and delivers a self-explanatory report of the major significant parts of the implementation. This report employs PostgreSQL as the preferred choice of DBMS on which the TPC-DS benchmark is performed.

1.2 Aim and Objectives

- The main aim of this report is to implement and evaluate the TPC-DS benchmark on a DBMS tool in order to learn how to efficiently perform a benchmark.
- Illustrating the objective further, with this project we will understand and implement the TPC-DS benchmark on a DBMS tool, learn how to perform a benchmark that helps us choose the best available database tool for our business.
- Another objective is to evaluate the benchmarking performance and analyse the results.

1.3 Tools Used

The tools installed and utilized to perform the benchmark operation are summarized in the table 1.1.

Tool	Version	Description
TPC-DS standard benchmark tool	3.2.0	The official tools set offered by TPC-DS for data generation, query generation and an answer set to compare results.
PostgreSQL	14.0	Open-source PostgreSQL relational database to store and query data.
JupyterLab	3.3.2	IDE for Python and iPython notebook.
Docker Desktop	20.10.17	Docker was used to run Ubuntu on top of Windows to run the TPC-DS tool in a Linux environment.
Python	3.10.5	Python was used as the main programming language for running our scripts.
MS PowerBI	2.110	Power BI was utilized to create visualizations of benchmarking results for further analysis.
GitHub	2.38.1	GitHub Desktop was used to share code files as well as images conveniently with the team members.

Table 1.1: Tools Used for TPC-DS Benchmarking with Postgres SQL

1.4 Limitations and Justifications

Since the entire project was implemented on a local machine, it was associated with a certain limit on the resources (tools that could be used), thereby preventing scaling higher benchmarks. It is definitely possible for us to procure cloud-based services like Google Cloud, and Amazon Azure and implement Postgres in their environments. With the provision of more cores as well as storage, we could have certainly benchmarked until 100 GB's least. Despite the complexity associated with local resources, we tried to benchmark 20 GB as well as 25 GB but haven't included them in the report due to abstract results.

Chapter 2

Technology Fundamentals

2.1 Postgres SQL

PostgreSQL is a free enterprise open-source object relational database management system (ORDBMS) akin to a relational database, bar that it is object-oriented such that it offers classes and objects models including inheritance in query-language and database schemas [Bartolini et al., 2017]. Initially developed at the University of California, Berkeley by the Database Research Team of the computer science department, is now adapted and developed by a vast horde of contributory developers. It provides a huge diversity of support languages ranging from C, Python, PHP, C++, Perl and Java amongst others that permits a variety selection of constructs that can proffer solutions to problems [The PostgreSQL Global Development Group, 2022]. In benchmarks, PostgreSQL is fast and provides similar excellent performance as when compared to other proprietary and open source databases [Obe & Hsu, 2017]. Also, it shoulders a huge part of the SQL standard and offers advanced present-day features such as but not limited to:

- Complex queries
- Transactional integrity Triggers
- Multiversion concurrency control
- Foreign keys
- Updatable views [Matthew & Stones, 2005]

Furthermore, PostgreSQL allows user extension in several ways such as adding and connecting new:

- operators
- data types
- index methods
- procedural languages
- aggregate functions
- functions

As as a result of the open license, PostgreSQL can be utilized, distributed & modified by any individuals without charge for any reason [The PostgreSQL Global Development Group, 2022].

2.1.1 Why PostgreSQL

PostgreSQL has numerous benefits including:

- Outstanding SQL standards compliance.
- Client-server architectural structure.
- High degree of synchronous interface and design where users don't interfere with each other.
- High extent of configuration and extensions for several kinds of applications
- Outstanding scalability and performance with high-level tuning and optimization features.
- Excellent support for different types of data formats including relational, post-relational (arrays, nested relations via record types) documents (JSON, CSV and XML), and dictionary keys/values.

In addition, the PostgreSQL system is a robust and high-quality tool with rich documentation, maintainability, interoperability and high availability. It requires low maintenance as well as provides excellent performance, security and compatibility for major operating systems on both enterprise and embedded usage [Bartolini et al., 2017]. In this project, PostgreSQL shall be used as a database management tool for implementing the TPC-DS benchmark.

2.2 Other Tools

2.2.1 Docker Desktop

This application allows for the transformation and optimization of workflows by allowing users to connect to a collection of pre-built developer tools and systems from the Docker Extension Marketplace. It allows for the creation and sharing of customized tools with other team members in its dev environment.
Also, Docker provides a fast way to build solutions and projects in containers as well as offers flexible control, secure access and management of container images [Install Docker Desktop, 2022]. For this coursework, Docker was used as a replacement to WSL (Windows Subsystem for Linux) to run the latest version of TPC-DS tool on Linux (Ubuntu) for the purpose of generating executable SQL queries for PostgreSQL from query templates.

2.2.2 Visual Studio Code

Visual Studio Code is a compact but extremely powerful source code editor that runs on computer desktops and is accessible on macOS, Windows and Linux operating systems. It has a built-in interface standard for Typescript, Node.js and JavaScript as well as a offers a wide array of extensions for other programming languages (Python, C++, C, Java, etc.). In action, visual studio code has an impressive UX and allows the customization of workflows [Visual Studio Code, 2022]. This project was useful for building and verifying the entire solution on dsdgen.sln.

2.2.3 Python Interpreter

Python is a general-purpose programming language that allows quick working and integration of systems effectively. This high-level language is dynamically input and supports procedural, functional and object-oriented programmed. It can be compiled using an interactive development emulator [Python, 2022]. For this benchmark project, Jupyter notebook was used to create and compile python scripts. Python allowed us to cleanse and transform the initial load data generated and push them into the database. Also, it was used to wrangle the generated query templates for effective accessibility on PostgreSQL.

Chapter 3

Benchmarking and Implementation

3.1 Introduction to Benchmarking

Benchmarking involves comparing performance indicators and processes to industry best practices usually in relation to time, quality and cost metrics. It is generally used to estimate similarities and contrast between a specific performance metric. In databases, benchmarking may be difficult especially if it follows different relational and object model approach. Despite this fact, organizations and individuals still experience the challenge of selecting a suitable DBMS platform for implementing models, as most databases offer many similar features on many fronts. However, performance is a great differentiator when choosing between available databases for decision support. Leveraging benchmarks can be used in recommending a suitable selection of a given technology [Tortosa, 2020].

In other words, benchmarking a database is the process of performing well-defined tests on that particular database for the purpose of evaluating its performance [Kabangu, 2009]. The performance evaluation can help an organization decide if the particular choice of the database can meet the business needs of the organization in the long run.

3.2 TPC-DS

TPC Benchmark™DS (TPC-DS) is basically a decision support benchmark model that fashions various relevant areas of a simple decision support structure, entailing queries as well as data maintenance. The TPC-DS benchmark offers a comprehensive decision-making system that represents a typical appraisal of the System Under Test's (SUT) performance model. Generally, it depicts a typical decision support platform that:

- Evaluates huge amounts of data;
- Provides solutions to business challenges in reality;
- Performs queries that meet the need of several operational complexities such as ad-hoc, data mining and reporting requirements;

- Delineate a large CPU usage and IO loading;
- Typifies periodic database maintenance activities especially with OLTP database synchronization;
- Executes on ORDBMS and RDBMS based systems.

In addition, a benchmark result assesses various aspects including the query response time output in an isolated user level, query throughput in multiple user levels and data maintenance evaluation for a designated hardware, data processing and operating system setting under a monitored and controlled decision support workload [Transaction Processing Performance Council (TPC), 2021].

3.3 Implementing TPC-DS on PostgreSQL

Figure 3.1 shows a business process model depicting a brief rundown of the implementation the TPC-DS benchmark on PostgreSQL.

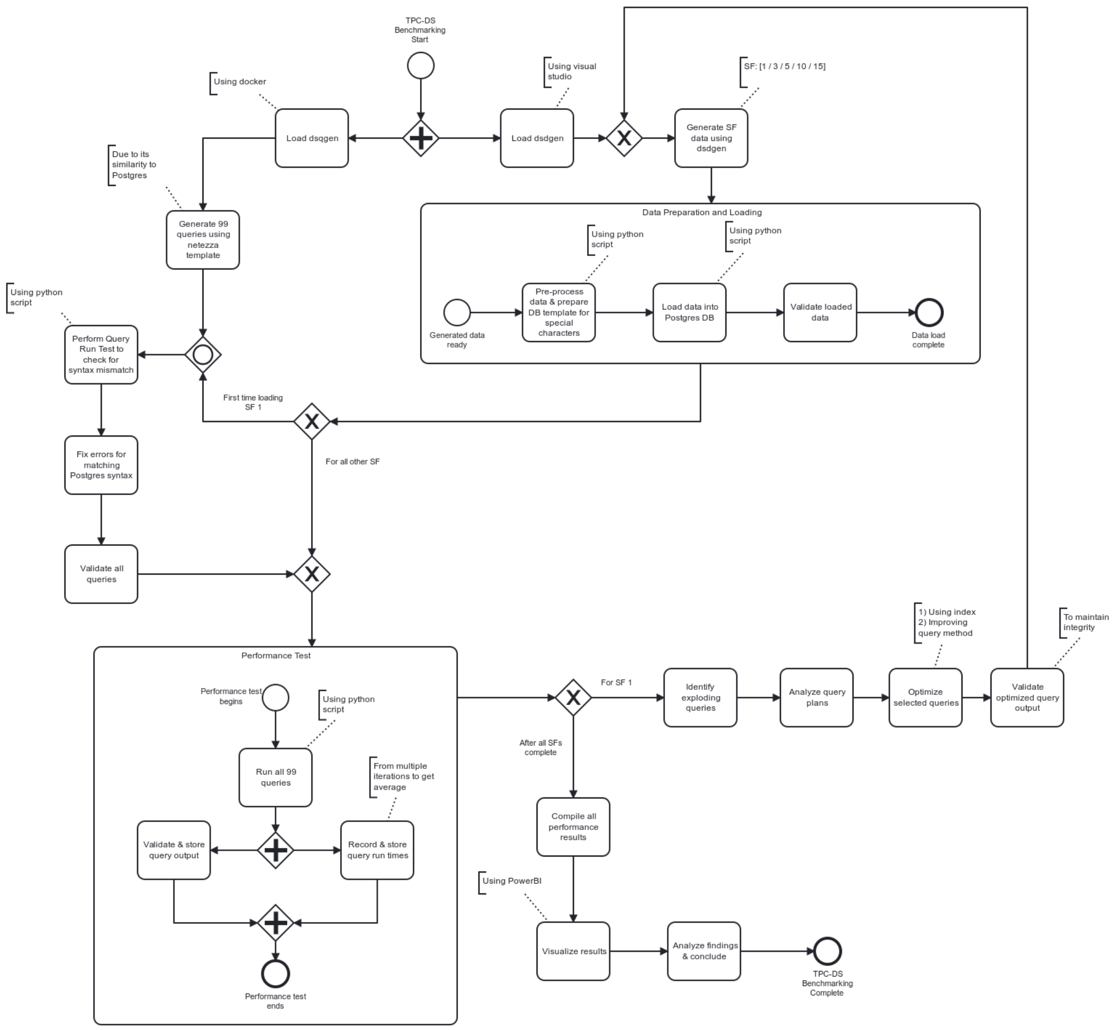

Figure 3.1: TPC-DS Process Diagram

Some of the major processes undertaken are summarized as follows:

- Building the solution of the dsdgen and dsqgen from the TPC-DS tool kit using visual studio and docker desktop respectively to obtain input data for the initial loading of the warehouse via the dsdgen utility.
- Generating scale factors using the dsdgen for the load data as well as generating 99 queries via the dsqgen using the Netezza SQL template.
- Cleaning, wrangling and transforming the data using suitable python commands to prepare and load the data into the database.
- Matching the queries with PostgreSQL by performing a query run test to check for mismatches and error.
- Using python scripts, perform tests on the queries and compile all results.
- Discover exploding queries in order to analyze and optimize them.
- Re-run all scale factors and evaluate final results.
- Visualize all outputs and provide detailed reports.

The benchmark was implemented on a local machine with specifications illustrated in table 3.1:

CPU (AMD Ryzen 7 6800HS)	**RAM (DDR5 SODIMM)**	**GPU (NVIDIA GeoForceRTX3060)**
• 8 cores, 16 threads • Base clocking speed at 3.2GHz and can overclock up to 4.7GHz • 16MB L3 Cache	• 16 GB memory • 4800MHz speed	• Dedicated graphics • 6GB VRAM

Table 3.1: Local Machine Specifications

3.3.1 Scaling and Database Population

3.3.1.1 Scaling Model

TPC-DS Benchmark identifies a set of distinct points used for scaling, hence called the scale factors that depend on the dsdgen file (from the TPC-DS toolkit), which is influenced by the software and hardware on which it is run [Transaction Processing Performance Council (TPC), 2021].

Scale Factors set for this project are listed in the table 3.2, wherein gigabyte (GB) is equivalent to 2^{10} bytes.

SF	1	3	5	10	15
Scale Factor	1 GB	3 GB	5 GB	10 GB	15 GB

Table 3.2: Implemented Scale Factors

As each scale factor has a corresponding SF which has no units and is almost equal to the bytes stored in the database. For this project the various scale factors and SFs are presented in the table 3.2

3.3.1.2 Test Database Scaling

For each scale factor, the total number of tuples stored in the data warehouse differs. This variation is depicted as in the table 3.3.

DB Table: Tuple Count Summary (for each scale factor)					
Table Name	**SF 1**	**SF 3**	**SF 5**	**SF 10**	**SF 15**
call_center	6	10	14	24	6
catalog_page	11718	11718	11718	12000	11718
catalog_returns	144067	432000	720174	1439749	2160757
catalog_sales	1441548	4319367	7199490	14401261	21602679
customer	100000	188000	277000	500000	183000
customer_address	50000	94000	138000	250000	91000
customer_demographics	1920800	1920800	1920800	1920800	1920800
date_dim	73049	73049	73049	73049	73049
household_demographics	7200	7200	7200	7200	7200
income_band	20	20	20	20	20
inventory	11745000	28188000	49329000	133110000	14356305
item	18000	36000	54000	102000	22000
promotion	300	344	388	500	327
reason	35	37	39	45	35
ship_mode	20	20	20	20	20
store	12	32	52	102	28
store_returns	287514	862834	1437911	2875432	4315222
store_sales	2880404	8639377	14400052	28800991	43197400
time_dim	86400	86400	86400	86400	86400
warehouse	5	6	7	10	5
web_page	60	90	122	200	162
web_returns	71763	215477	359991	719217	1079028
web_sales	719384	2160165	3599503	7197566	10795812
web_site	30	32	34	42	30

Table 3.3: Tuple Count Summary

3.3.1.3 DSDGEN and Database Population

The data is loaded into the database using the following command:

```
.\dsdgen.exe /scale 10 /dir .\tmp /suffix .csv /parallel 4 /child 1
/delimiter "^" /terminate n &

.\dsdgen.exe /scale 10 /dir .\tmp /suffix .csv /parallel 4 /child 2
/delimiter "^" /terminate n &

.\dsdgen.exe /scale 10 /dir .\tmp /suffix .csv /parallel 4 /child 3
/delimiter "^" /terminate n &
```

```
.\dsdgen.exe /scale 10 /dir .\tmp /suffix .csv /parallel 4 /child 4
/delimiter "^" /terminate n
```

- Scale factor varied as per the table 3.2
- Data was dumped as CSV file (Default file output extension is .dat)
- For fast and parallel data generation, the process is run parallelly on 4 threads
- Each child process command is concatenated using and
- End of line (EOL) is indicated by "|" and if "|" appears consecutively twice, it indicates a NULL value in the column. This is handled using:
  - `"^" as the delimiter -> /delimiter "^"`
  - `End of line (EOL) character to be NULL -> /terminate n`
- To support special international characters throughout the entire workflow, ISO/IEC 8859-1 is used. For Postgres - the Win1252 template supports the requisite character sets [The PostgreSQL Global Development Group, 2022]

3.3.2 Queries Overview

3.3.2.1 Query Definition

Queries in TPC-DS basically answer various business questions, pertaining to the data warehouse for example - "What are the total sales through each channel in the year 2009?" TPC-DS toolkit includes several templates for query generation. In this project, we generated queries as per the Netezza template due to its close resemblance with PostgreSQL.

3.3.2.2 Query Modifications

To ensure complete compatibility of queries with PostgreSQL, a few minor modifications were done.

- Intervals: Postgres requires the specific keyword - interval to identify the interval of days, Query Syntax were modified from 3.2 to 3.3.

```
and d_date between (cast ('2002-05-18' as date) - 30 days)
               and (cast ('2002-05-18' as date) + 30 days)
```

Figure 3.2: Initial Query For Interval Syntax

```
and d_date between (cast ('2002-05-18' as date) - interval '30 day')
               and (cast ('2002-05-18' as date) + interval '30 day')
```

Figure 3.3: Updated Query For Interval Syntax

- Aliases - A few of the sub-queries and columns were given an alias "x".
- Joins - Instead of generally selecting from Table A, Table B and specifying the JOIN condition in WHERE Clause, explicit joins (inner/left/right) were used.

3.3.2.3 Query Ordering

Queries are ordered in sequences by the dsqgen file. Due to the difference in the query sequences, -QUALIFY was used as the sample command below to generate them in the same order as the answer templates.

```
./dsqgen -DIRECTORY ../query_templates -INPUT
../query_templates/templates.lst \ -VERBOSE Y -QUALIFY Y
-DIALECT netezza
```

Note: The above command was executed on a Docker (Linux) environment.

3.3.3 Query Execution and Optimisation

3.3.3.1 Query Execution

- After query generation by TPC-DS templates, queries were run to identify the syntax errors. Modifications to the queries were done as in 3.3.2.2 for 23 erroneous queries in this project.
- Queries were first run on SF-1 and the average run-time for 5 iterations was considered as the Query run-time.
- The previous step was essential to identify the set of potential queries that might explode (the run-time of that particular query is very high in comparison to the others) as data increases (higher SFs) because of correlated sub-queries, sub-queries, etc.
- Optimisation (as in 3.5.2) was done for a few queries and run against all SFs.

3.3.3.2 Optimisation

- Optimized queries are detailed in Appendix A.
- The ways used in this project to optimize are:
  - Query plans were studied thoroughly to analyze possible points of optimisation.
  - Correlated Sub-queries were rewritten using Common Table Expressions (CTE).
  - Indexes were added for fast retrieval of table tuples.
  - Tables in joins were re-ordered in the increasing order of their counts ensuring efficient joins.
  - Distinct keyword was eliminated by rewriting queries in the form of Common Table Expressions (CTE).

Chapter 4

Results and Discussions

As mentioned in the previous section, the benchmark was performed on a local machine, with a total of five different scale factors (1, 3, 5, 10 and 15). The results will be focusing on a few different criteria, namely, the evolution of the run times as the scale factor of the database increases, comparison between optimized and original query performances, analysis of queries with exponential run time growth, and the SQL methods that typically hampers the performance of the query execution time.

4.1 Benchmarking results

4.1.1 Overall performance across all scale factors

Inspecting the total average run time for all the 99 queries which were run sequentially for multiple iterations, it is evident that the performance is nowhere close to being linear. In fact, as the scale factor increases further, an exponential pattern starts to emerge, as seen in Figure 4.1.

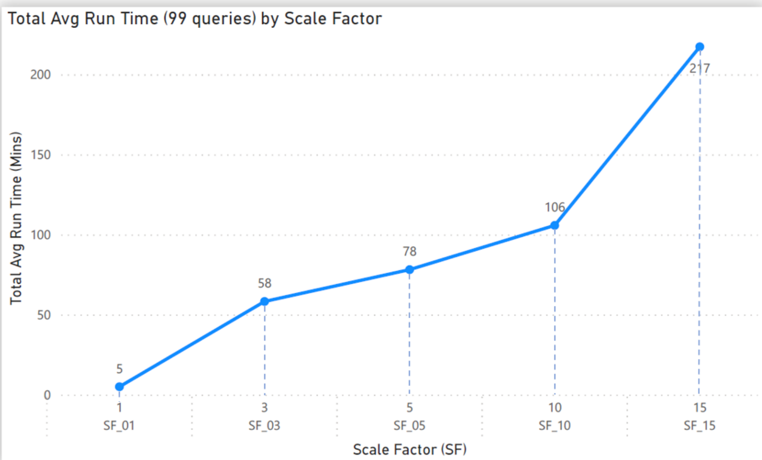

Figure 4.1: Total Average Runtime (99 Queries) by Scale Factor

At a scale factor (SF) of 1 (equivalent to a database size of 1GB), all of the 99 queries were managed to complete within a span of 5 minutes (on average). This quickly rises to a total of 58 minutes for SF 3, which then follows along with a smaller increase in run time for SF 5 and 10 respectively, but rises again steeply when progressing to SF 15.

Although this is the case, looking only at the overall run time for all 99 queries together, gives a biased assumption regarding the performance of many of the individual queries. Thus, the performance results are further broken down into individual queries in the next section.

4.1.2 Individual performance across all scale factors

Due to a large number of queries, they are split into five charts containing 20 queries each, as seen in Figures 4.2 to 4.6. Firstly, it can be observed that the majority of the queries are not running exponentially as the scale factor increases and it is actually due to a few specific queries such as query 14, 30 and 95 that causes the total overall run time to massively increase as seen previously in Figure 4.1. Furthermore, an unusual pattern is evident with query 4, where it initially rises in run time for SF 3 and 5, but reduces as the scale factor increases more towards SF 10 and 15. Both of these unique observations are further discussed in Section 4.2.

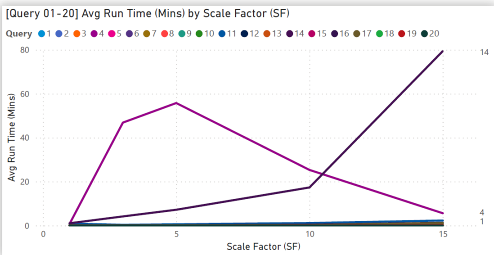

Figure 4.2: [Query 01-20] Average Runtime (Mins) by Scale Factor (SF)

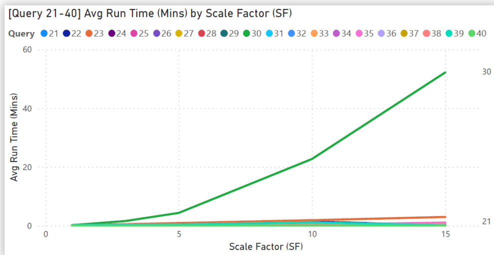

Figure 4.3: [Query 21-40] Average Runtime (Mins) by Scale Factor (SF)

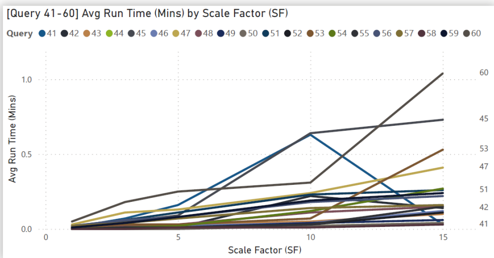

Figure 4.4: [Query 41-60] Average Runtime (Mins) by Scale Factor (SF)

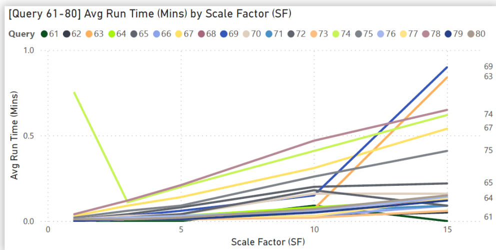

Figure 4.5: [Query 61-80] Average Runtime (Mins) by Scale Factor (SF)

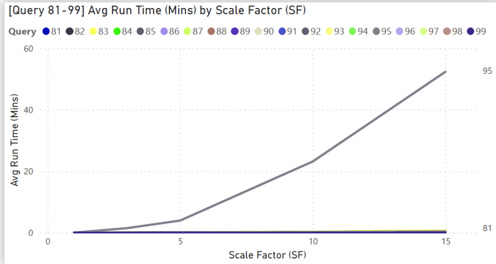

Figure 4.6: [Query 81-99] Average Runtime (Mins) by Scale Factor (SF)

Given that there were a few queries with exponential run times, the scale for the charts above was stretched due to these outliers, resulting in less visibility on the patterns for some of the queries with much less run time. Therefore, an additional chart is generated below (Figure 4.7), which excludes these outliers and shows the overall non-exponential trend for the majority of the queries (95%), and mostly managing to complete within 60 seconds at a scale factor of 15.

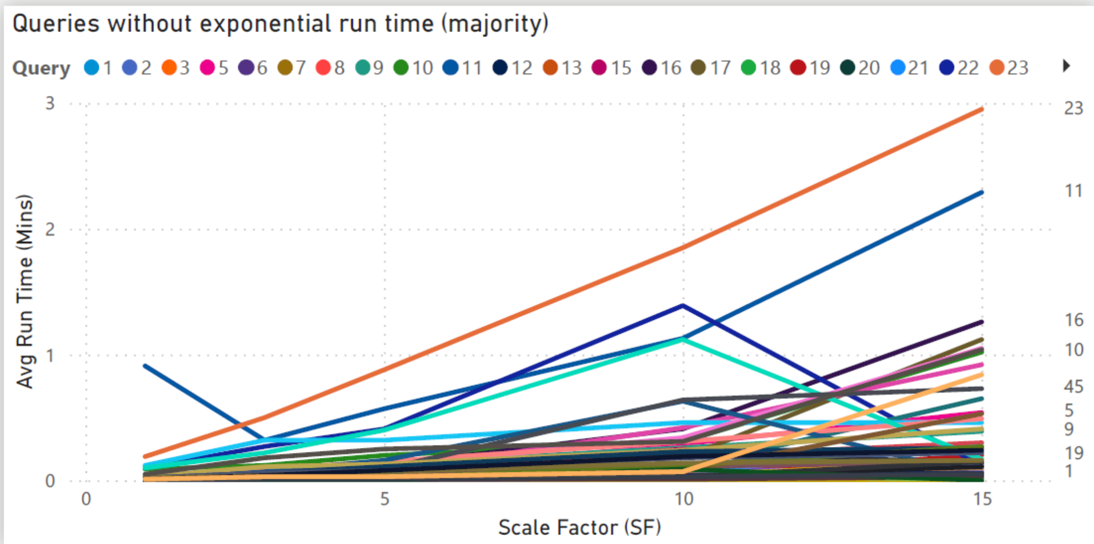

Figure 4.7: Queries without exponential runtime (majority)

4.1.3 Comparison of optimized and original queries

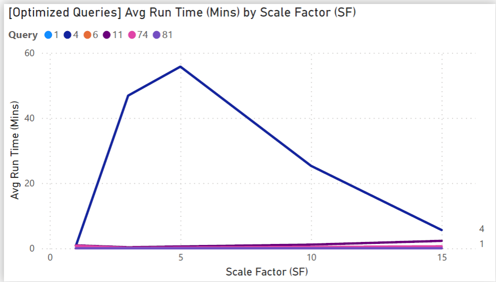

Figure 4.8: Optimised Queries Average Runtime (Mins) by Scale Factor (SF)

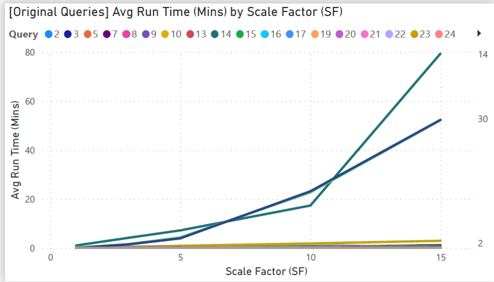

Figure 4.9: Original Queries Average Runtime (Mins) by Scale Factor (SF)

As discussed in the benchmarking implementation (Section 3), a total of six queries (1, 4, 6, 11, 74 and 81) were optimized by improving the structure and SQL method-/approach used (while maintaining the same output as original). From Figure 4.8 (optimized queries), it is evident that none of the six optimized queries continued to explode exponentially as the scale factors progressed up to SF 15. Although, as mentioned previously, query 4 experienced a unique pattern which is discussed in Section 4.2. In contrast, if we observe Figure 4.9 (original queries), it can be seen that there are a few queries (14, 30 and 95) that kept increasing exponentially in run time as the scale factor increased (also discussed in Section 4.2). Nonetheless, most of the remaining queries did not explode possibly due to the help of indices as well.

4.2 Analysis of irregular pattern results

4.2.1 Demystifying the performance variation on Query 04

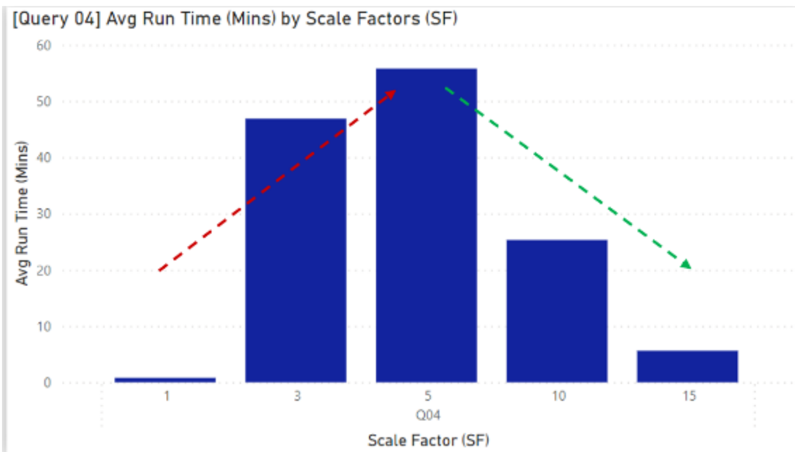

Figure 4.10: Query 04 - Average Runtime (Mins) by Scale Factor (SF)

As mentioned above, query 04 was one of the queries that were optimized to further improve its performance as the database is scaled. This is due to the query making use of all three large fact tables (catalog_sales, store_sales, web_sales), which can result in very expensive steps for the query to run. As seen from Figure 4.10, the query worked very well for SF 1, but as the scale factor increased to 3 and 5, the run time increased exponentially, but then interestingly reduced when scaling to SF 10 and 15.

In order to understand what changed between SF 5 and SF 15, a deeper dive was done into the query planner and statistics in terms of how the query was executed in steps for both of these scale factors.

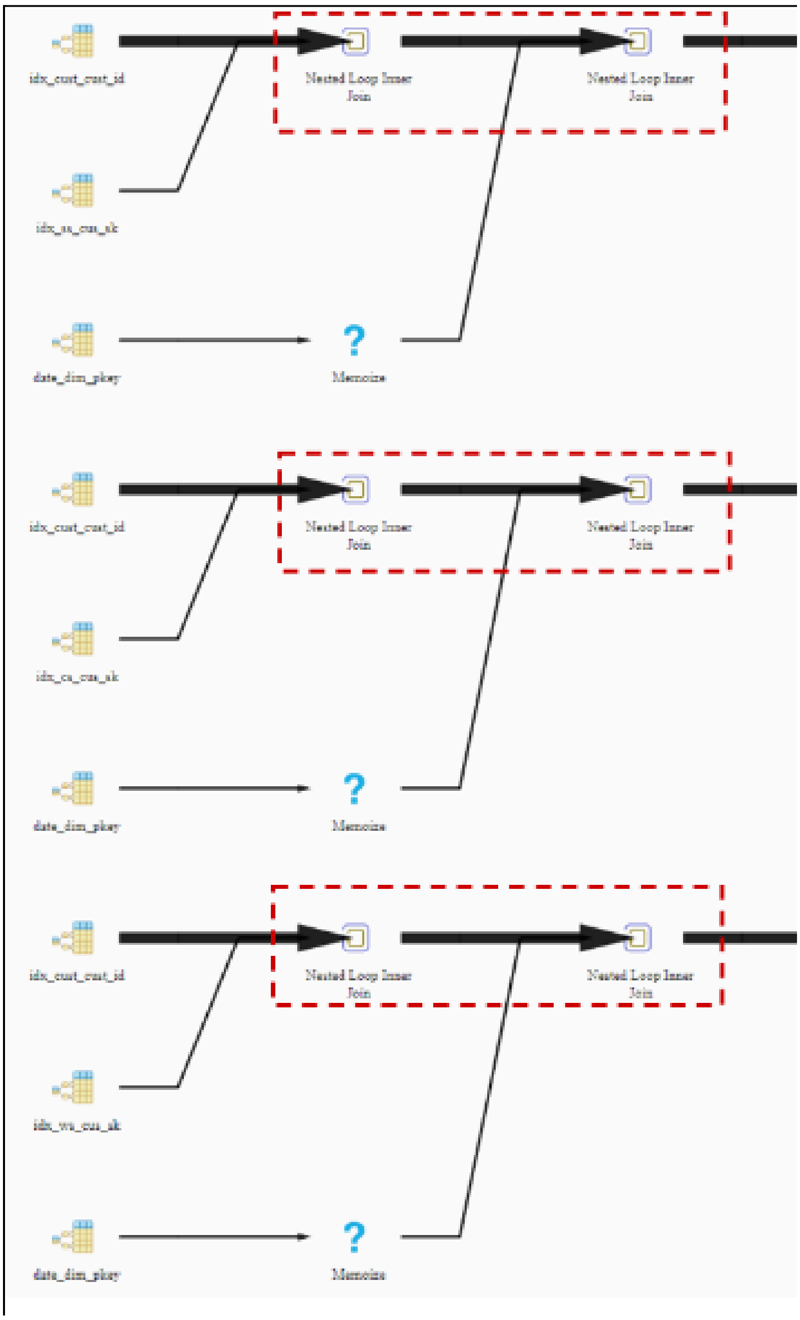

Figure 4.11: Query 04 - Query Plan 5 SF

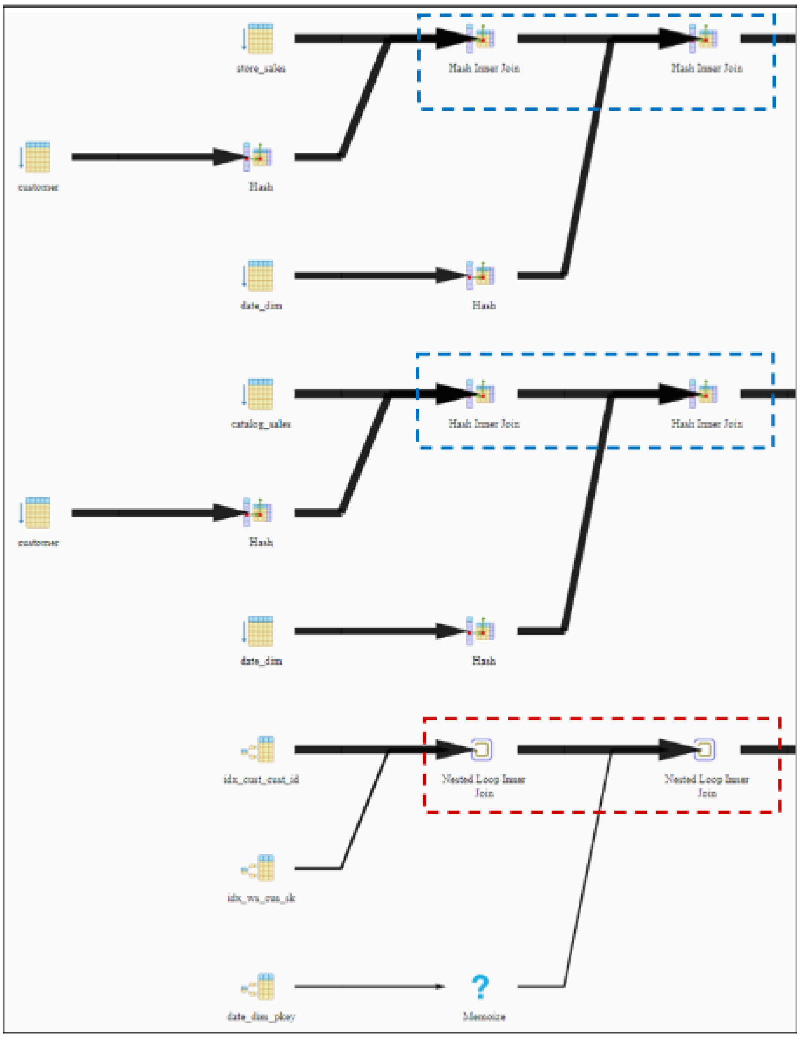

Figure 4.12: Query 04 - Query Plan 15 SF

Statistics per Node Type			
Node type	Count	Time spent	% of query
Aggregate	6	7355.196 ms	0.21%
CTE Scan	6	111459.894 ms	3.12%
Gather Merge	3	47002.046 ms	1.32%
Incremental Sort	4	9329.735 ms	0.27%
Index Scan	9	571.891 ms	0.02%
Limit	1	3471878.941 ms	96.94%
Materialize	2	15.91 ms	0.01%
Memoize	3	0 ms	0%
Merge Inner Join	4	397.699 ms	0.02%
Nested Loop Inner Join	7	3514598.933 ms	98.13%
Sort	5	551.103 ms	0.02%

Statistics per Relation			
Relation name	Scan count	Total time	% of query
Node type	Count	Sum of times	% of relation
catalog_sales	1	0.05 ms	0.01%
Index Scan	1	0.05 ms	100%
customer	3	571.742 ms	0.02%
Index Scan	3	571.742 ms	100%
date_dim	3	0.003 ms	0.01%
Index Scan	3	0.003 ms	100%
store_sales	1	0.072 ms	0.01%
Index Scan	1	0.072 ms	100%
web_sales	1	0.024 ms	0.01%
Index Scan	1	0.024 ms	100%

Figure 4.13: Query 04 - Query Statistics 5 SF

Statistics per Node Type

Node type	Count	Time spent	% of query
Aggregate	6	17101.377 ms	4.59%
CTE Scan	6	101151.402 ms	27.13%
Gather Merge	3	34629.793 ms	9.29%
Hash	4	234.046 ms	0.07%
Hash Inner Join	4	21792.748 ms	5.85%
Incremental Sort	2	4976.232 ms	1.34%
Index Scan	3	193.253 ms	0.06%
Limit	1	131542.595 ms	35.28%
Materialize	3	0.78 ms	0.01%
Memoize	1	0 ms	0%
Merge Inner Join	3	4.698 ms	0.01%
Nested Loop Inner Join	4	293517.387 ms	78.71%
Seq Scan	6	5145.489 ms	1.38%
Sort	6	135467.997 ms	36.33%

Statistics per Relation

Relation name	Scan count	Total time	% of query
Node type	**Count**	**Sum of times**	**% of relation**
catalog_sales	1	3653.605 ms	0.98%
Seq Scan	1	3653.605 ms	100%
customer	3	265.619 ms	0.08%
Index Scan	1	193.142 ms	72.72%
Seq Scan	2	72.477 ms	27.29%
date_dim	3	7.408 ms	0.01%
Index Scan	1	0.001 ms	0.02%
Seq Scan	2	7.407 ms	99.99%
store_sales	1	1412 ms	0.38%
Seq Scan	1	1412 ms	100%
web_sales	1	0.11 ms	0.01%
Index Scan	1	0.11 ms	100%

Figure 4.14: Query 04 - Query Statistics 15 SF

Firstly, inspecting the query plan for both scale factors, it's evident that different initial steps were taken. As seen from Figure 4.11 for SF 5, the query planner decided to make use of all indexes available, but then proceeded to the next step with multiple nested loop inner joins. On the other hand, in Figure 4.13 for SF 15, the query planner felt that the customer table was small enough in comparison to two out of three of the large fact tables (catalog_sales store_sales) and decided to perform a sequential scan for them instead of using an index. This resulted in less usage of nested loop inner joins overall for the next steps.

Looking further into the statistics for the query run, it can be observed that the additional nested loop inner joins resulted in a much more expensive execution overall. Thus, for SF 5, the multiple nested loop inner joins cost a total of 58 minutes (3514598.933 ms), while for SF 15, it cost only about 5 minutes (293517.387 ms) as there were much fewer of them (specifically avoiding the large fact tables). Detailed statistics are available in Figures 4.13 and 4.14.

This observation further confirms that the index can play the hero and the devil in different situations, given that the query plan's choices could result in more expensive decisions for the steps that follow that.

4.2.2 Understanding the catalyst for queries with exponential run time

Three queries demonstrated exponential run times, namely query 14, 30 and 95 (Figure 4.15), and this creates the opportunity to further understand what were the factors that influenced this the most.

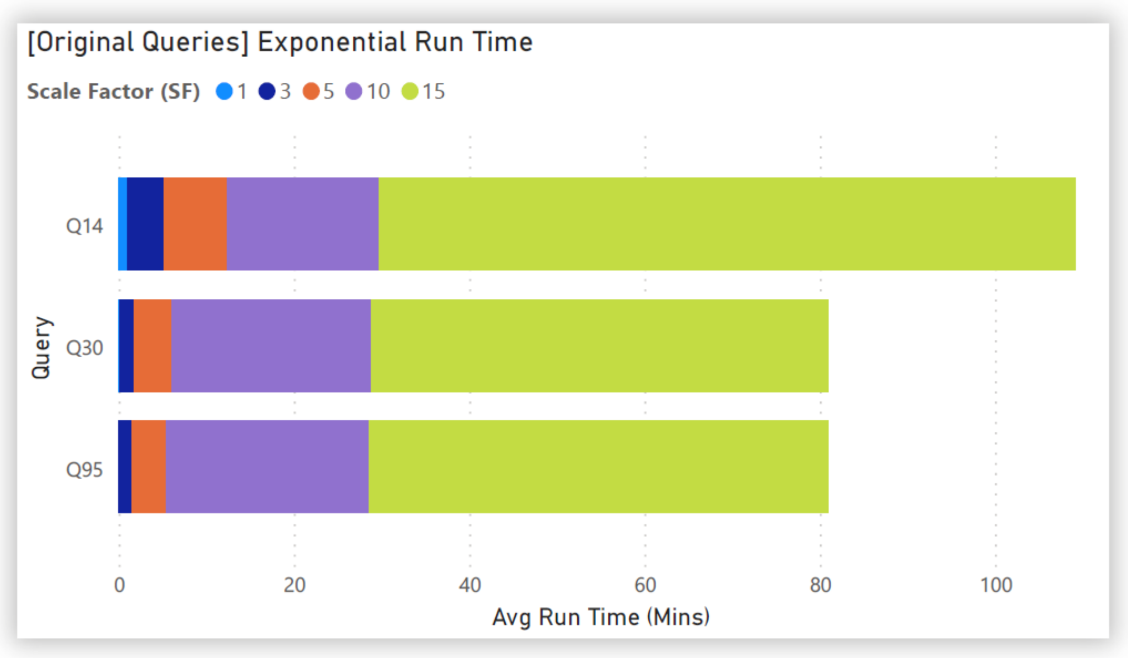

Figure 4.15: Exponential run-time detailed analysis for Q14, Q30, Q95 - All SFs

To begin with, we have query 14, which is a unique query that actually has two outputs. In order to make a fair comparison, query 23 was used as it also has two outputs and uses all the three large fact tables as well. Similar to the previous analysis, a deeper dive was done into the query planner and statistics, which resulted in a identical discovery where excessive usage of nested loop inner joins on large tables for query 14 (Figure 4.16) resulted in a much longer run time (exponential), as compared to query 23 (Figure 4.17) which had more of a linear run time as the scale factors progressed.

Statistics per Node Type

Node type	Count	Time spent	% of query
Aggregate	9	3076.298 ms	0.41%
Append	4	2404.09 ms	0.32%
Bitmap Heap Scan	1	21.22 ms	0.01%
Bitmap Index Scan	1	0.001 ms	0.01%
CTE Scan	6	27243.692 ms	3.58%
Gather	4	6100.239 ms	0.81%
Hash	13	7.764 ms	0.01%
Hash Inner Join	13	24067.091 ms	3.17%
Hashed Intersect	2	5170.589 ms	0.68%
Index Scan	14	16.466 ms	0.01%
Limit	1	734149.669 ms	96.43%
Nested Loop Inner Join	6	723638.081 ms	95.05%
Result	1	0.337 ms	0.01%
Seq Scan	10	5490.01 ms	0.73%
Sort	4	1422.78 ms	0.19%
Subquery Scan	4	3224.136 ms	0.43%

Statistics per Relation

Relation name	Scan count	Total time	% of query
Node type	Count	Sum of times	% of relation
catalog_sales	3	1906.409 ms	0.26%
Index Scan	1	8.822 ms	0.47%
Seq Scan	2	1897.587 ms	99.54%
date_dim	9	1.793 ms	0.01%
Index Scan	9	1.793 ms	100%
item	7	24.547 ms	0.01%
Index Scan	3	0.081 ms	0.33%
Seq Scan	4	24.466 ms	99.68%
store_sales	3	1592.449 ms	0.21%
Bitmap Heap Scan	1	21.22 ms	1.34%
Seq Scan	2	1571.229 ms	98.67%
web_sales	3	2002.498 ms	0.27%
Index Scan	1	5.77 ms	0.29%
Seq Scan	2	1996.728 ms	99.72%

Statistics per Node Type

Node type	Count	Time spent	% of query
Aggregate	6	2685.649 ms	0.25%
Append	3	2384.906 ms	0.22%
Bitmap Heap Scan	2	37.525 ms	0.01%
Bitmap Index Scan	2	0.002 ms	0.01%
CTE Scan	4	26818.017 ms	2.47%
Gather	4	5923.643 ms	0.55%
Hash	10	6.722 ms	0.01%
Hash Inner Join	10	15378.268 ms	1.42%
Hashed Intersect	2	4920.412 ms	0.46%
Index Scan	12	49.591 ms	0.01%
Limit	1	1058950.316 ms	97.54%
Merge Inner Join	2	1590.288 ms	0.15%
Nested Loop Inner Join	5	1365941.567 ms	125.81%
Result	1	0.292 ms	0.01%
Seq Scan	10	5643.236 ms	0.52%
Sort	5	77203.501 ms	7.12%
Subquery Scan	4	3104.948 ms	0.29%

Statistics per Relation

Relation name	Scan count	Total time	% of query
Node type	Count	Sum of times	% of relation
catalog_sales	2	1881.136 ms	0.18%
Seq Scan	2	1881.136 ms	100%
date_dim	10	49.523 ms	0.01%
Index Scan	10	49.523 ms	100%
item	6	20.248 ms	0.01%
Index Scan	2	0.068 ms	0.34%
Seq Scan	4	20.18 ms	99.67%
store_sales	4	1577.421 ms	0.15%
Bitmap Heap Scan	2	37.525 ms	2.38%
Seq Scan	2	1539.896 ms	97.63%
web_sales	2	2202.025 ms	0.21%
Seq Scan	2	2202.025 ms	100%

Figure 4.16: Query 14 - Query Statistics

Statistics per Node Type

Node type	Count	Time spent	% of query
Aggregate	12	15354.644 ms	14.91%
Append	1	0.816 ms	0.01%
Bitmap Heap Scan	2	3.715 ms	0.01%
Bitmap Index Scan	2	0.002 ms	0.01%
CTE Scan	4	84671.148 ms	82.18%
Gather Merge	3	2772.679 ms	2.7%
Hash	9	36.637 ms	0.04%
Hash Inner Join	9	15175.104 ms	14.73%
Index Only Scan	2	4.705 ms	0.01%
Index Scan	4	1.292 ms	0.01%
Limit	1	18862.237 ms	18.31%
Nested Loop Inner Join	2	17991.501 ms	17.47%
Seq Scan	4	2661.907 ms	2.59%
Sort	3	48535.991 ms	47.11%

Statistics per Relation

Relation name	Scan count	Total time	% of query
Node type	Count	Sum of times	% of relation
catalog_sales	1	2.637 ms	0.01%
Bitmap Heap Scan	1	2.637 ms	100%
customer	2	4.705 ms	0.01%
Index Only Scan	2	4.705 ms	100%
date_dim	4	1.292 ms	0.01%
Index Scan	4	1.292 ms	100%
item	1	5.831 ms	0.01%
Seq Scan	1	5.831 ms	100%
store_sales	3	2656.076 ms	2.58%
Seq Scan	3	2656.076 ms	100%
web_sales	1	1.078 ms	0.01%
Bitmap Heap Scan	1	1.078 ms	100%

Statistics per Node Type

Node type	Count	Time spent	% of query
Aggregate	13	14984.768 ms	15.12%
Append	1	0.04 ms	0.01%
Bitmap Heap Scan	2	3.526 ms	0.01%
Bitmap Index Scan	2	0.002 ms	0.01%
CTE Scan	4	81472.66 ms	82.21%
Gather Merge	3	2804.833 ms	2.83%
Hash	9	23.889 ms	0.03%
Hash Inner Join	9	14076.363 ms	14.21%
Index Only Scan	2	7.477 ms	0.01%
Index Scan	6	0.55 ms	0.01%
Limit	1	18152.302 ms	18.32%
Nested Loop Inner Join	4	17260.938 ms	17.42%
Seq Scan	4	2530.438 ms	2.56%
Sort	6	46907.968 ms	47.33%

Statistics per Relation

Relation name	Scan count	Total time	% of query
Node type	Count	Sum of times	% of relation
catalog_sales	1	2.472 ms	0.01%
Bitmap Heap Scan	1	2.472 ms	100%
customer	4	7.516 ms	0.01%
Index Only Scan	2	7.477 ms	99.49%
Index Scan	2	0.039 ms	0.52%
date_dim	4	0.511 ms	0.01%
Index Scan	4	0.511 ms	100%
item	1	6.48 ms	0.01%
Seq Scan	1	6.48 ms	100%
store_sales	3	2523.958 ms	2.55%
Seq Scan	3	2523.958 ms	100%
web_sales	1	1.054 ms	0.01%
Bitmap Heap Scan	1	1.054 ms	100%

Figure 4.17: Query 23 - Query Statistics

Query 30 and 95 also demonstrated similar performance issues as seen above, and therefore will not be elaborated further.

4.3 Summary of Results

Running a benchmark test on a large variety of queries, on multiple scale factors provided great insights into how Postgres deals with both increasing volume of data, indices and the alteration of node steps chosen depending on the size of the table data. Although majority of the queries demonstrated non-exponential run-time performance (more than 90%), there were a few that stood out. An important influential factor was identified from these queries, which is that nested loop inner joins are one of the most taxing operations that can be done during query execution, and Postgres struggles with that once the data volume increases.

Chapter 5

Conclusion

In conclusion, the TPC-DS framework has provided a well-rounded model, including pragmatic components to perform a fair and transparent benchmark on Postgres DB. Some queries were optimized to further improve their performance given they were struggling in the initial run test and a few index were also added to full utilize the capabilities of the database. This enabled the performance test to have even more variety, allowing additional analysis in terms of comparison between original and optimized queries, the identification of exploding queries with exponential run time performance and understanding unique behaviours between different scale factors. A key component that has been identified, was the heavy computational cost of nested loop inner joins which caused a few of the queries to struggle and have large execution run times as the scales increased. Therefore, it is crucial for analysts to design their OLAP queries in such a way that they can avoid correlated sub-queries which results in the usage of nested loop inner joins. Furthermore, it is important to always consider how the query planner is going to form the execution steps, in order to fully utilized the capabilities of its resources and avoid settling for higher computational steps as much as possible.

Bibliography

[Bartolini et al., 2017] Bartolini, G., Ciolli, G., & Riggs, S. (2017). PostgreSQL Administration Cookbook - Third Edition. Packt Publishing, Limited.

[Install Docker Desktop, 2022] install Docker Desktop. (2022, January 22). Docker. Retrieved October 27, 2022, from https://www.docker.com/products/docker-desktop/

[Kabangu, 2009] Kabangu, S. (2009). Benchmarking Databases.

[Matthew & Stones, 2005] Matthew, N., & Stones, R. (2005). Beginning Databases with PostgreSQL: From Novice to Professional. Apress.

[Obe & Hsu, 2017] Obe, R. O., & Hsu, L. S. (2017). PostgreSQL: Up and Running : a Practical Guide to the Advanced Open Source Database. In (pp. 11 - 14). O'Reilly Media, Incorporated.

[The PostgreSQL Global Development Group, 2022] The PostgreSQL Global Development Group. (2022). PostgreSQL 14.5 Documentation. PostgreSQL. Retrieved October 26, 2022, from https://www.postgresql.org/docs/14

[Python, 2022] Python. (2022, February). Welcome to Python.org. Retrieved October 27, 2022, from https://www.python.org/

[Tortosa, 2020] Tortosa, Á. H. (2020). Performance Benchmark Postgresql / Mongodb. Ongres.com. Retrieved October 28, 2022, from https://info.enterprisedb.com/rs/069-ALB-339/images/PostgreSQL_MongoDB_Benchmark-WhitepaperFinal.pdf

[Transaction Processing Performance Council (TPC), 2021] TPC-DS. (2021, January). tpc.org. Retrieved October 28, 2022, from https://www.tpc.org/tpcds/

[Visual Studio Code, 2022] Visual Studio Code. (2022, January). Visual Studio Code - Code Editing. Redefined. Retrieved October 27, 2022, from https://code.visualstudio.com/

[Zimanyi, n.d.] Zimanyi, E. (n.d.). INFO-H-419: Data Warehouses. INFO-H-419: Data Warehouses [Université Libre de Bruxelles - Service CoDE - Laboratoire WIT]. Retrieved October 27, 2022, from https://cs.ulb.ac.be/public/teaching/infoh419

Appendix A

Appendix

A.1 Python Scripts

```
# TPCDS: Preprocessing, DB Setup and Data Load Script

# importing Libraries
import sys, os, re
import psycopg2
import numpy as np
import pandas as pd
from psycopg2 import Error
from psycopg2.extensions import ISOLATION_LEVEL_AUTOCOMMIT

# set up connection variables
db_host = "localhost"
db_port = "5432"
db_user = "postgres"
db_pass = "password"
db_name = "postgres"

# function to connect with postgres
def connect_postgres(db_host, db_port, db_user, db_pass, db_name):
    try:
        # Connect to an existing database
        connection = psycopg2.connect(host = db_host,
                                      port = db_port,
                                      user = db_user,
                                      password = db_pass,
                                      database = db_name)
        # Set auto-commit
        connection.set_isolation_level(ISOLATION_LEVEL_AUTOCOMMIT);
        # Create a cursor to perform database operations
        cur = connection.cursor()
        # Print PostgreSQL details
        print("PostgreSQL server information")
        print(connection.get_dsn_parameters(), "\n")
        # Executing a SQL query
        cur.execute("SELECT version();")
        # Fetch result
        record = cur.fetchone()
        print("You are connected to - ", record, "\n")

    except (Exception, Error) as error:
```

```python
        print("Error while connecting to PostgreSQL", error)
    else:
        return cur

# connect to postgres
cur = connect_postgres(db_host, db_port, db_user, db_pass, db_name)

# drop tpcds db
db_name = "tpcds"

cur.execute(
    f"DROP DATABASE IF EXISTS {db_name} WITH (FORCE);"
)
print("SQL Status Output:\n", cur.statusmessage)


# change win1252 encoding temp db to normal before drop
try:
    cur.execute(
        "ALTER DATABASE win1252_temp is_template false;"
    )
except Exception as e:
    print(e)
else:
    print("SQL Status Output:\n", cur.statusmessage)

# drop win1252 encoding temp db (after set to normal db)
cur.execute(
    "DROP DATABASE IF EXISTS win1252_temp WITH (FORCE);"
)
print("SQL Status Output:\n", cur.statusmessage)

# create win1252 encoding temp db
cur.execute(
    """

    CREATE DATABASE win1252_temp
        WITH
        OWNER = postgres
        TEMPLATE = template0
        ENCODING = 'WIN1252'
        CONNECTION LIMIT = -1
        IS_TEMPLATE = True;

    """
)
print("SQL Status Output:\n", cur.statusmessage)

# create tpcds db
cur.execute(
    f"""

    CREATE DATABASE {db_name}
        WITH
        OWNER = postgres
        TEMPLATE = win1252_temp
        ENCODING = 'WIN1252'
        CONNECTION LIMIT = -1
```

```
        IS_TEMPLATE = False;

    """
)
print("SQL Status Output:\n", cur.statusmessage)

# connect to tpcds db
cur = connect_postgres(db_host, db_port, db_user, db_pass, db_name)

# create tables for db
cur.execute(open("tools/tpcds.sql", "r").read())
print("SQL Status Output:\n", cur.statusmessage)
cur.execute(open("tools/tpcds_source.sql", "r").read())
print("SQL Status Output:\n", cur.statusmessage)

# get dir path
path = os.getcwd() + '\\tools\\tmp\\'
files = os.listdir(path)
print(path)

# function to get full abosolute path of csv files containing data
def get_absolute_path(d):
    return [os.path.join(d, f) for f in os.listdir(d)]

# get full abosolute path of csv files containing data
files_abs_path = [p.replace('\\', '/') for p in get_absolute_path(path)]
print("Total files:", len(files_abs_path))
print("First few files...")
files_abs_path[:5]

# exclude extra delimiter for dbgen_version file
file_count = 0
for iteration in range(0, 1):
    for file in files_abs_path:
        file_open = open(file,'r')
        all_text = file_open.read().replace(" ", "")
        file_open.close()

        if (all_text[-13] == '^' and 'dbgen_version' in file):
            file_open_read = open(file, 'r', encoding = 'latin-1')
            string_list = file_open_read.readlines()
            file_open_read.close()

            for i in range(len(string_list)):
                last_delimeter_index = string_list[i].rfind("^")
                string_list[i] = string_list[i][:last_delimeter_index] + "" + string_list[i][last_delimeter_index + 1:]

            file_open_write = open(file, 'w', encoding = 'latin-1')
            new_file_contents = ''.join(string_list)
            file_open_write.write(new_file_contents)
            file_open_write.close()

            file_count += 1
        else:
            pass
```

```
    print(f'\nIteration {iteration + 1} done!')
    print(f'{file_count} file(s) updated for extra column exclusion.')
    file_count = 0

# generate sql commands for loading data from csv to postgres db
# considers that csv files were generated in parallel stream
sql_commands_file = open('data_load_script.sql','w')

for file in files:
    underscore_index = [underscore_ind.start() for underscore_ind in re.finditer('_', file)]
    file_name = file[:underscore_index[-2]]
    file_path = path+file
    sql_command = "COPY public."+file_name+" FROM '"+file_path+"' delimiter '^' CSV;\n"
    sql_commands_file.write(sql_command)

sql_commands_file.close()

# load csv data into db
cur.execute(open("data_load_script.sql", "r").read())
print("SQL Status Output:\n", cur.statusmessage)

# add constraints to db
cur.execute(open("tools/tpcds_ri.sql", "r").read())
print("SQL Status Output:\n", cur.statusmessage)

# close connection to db
cur.close()

# End of script.
```

Listing A.1: Preprocessing_DBSetup_DataLoad Script

```
# TPCDS: Query Run Test Script

# importing libraries
import sys, os
import psycopg2
from psycopg2 import Error
from psycopg2.extensions import ISOLATION_LEVEL_AUTOCOMMIT
from IPython.display import clear_output
from datetime import datetime

# set up connection variables
db_host = "localhost"
db_port = "5432"
db_user = "postgres"
db_pass = "password"
db_name = "tpcds"

# function to connect with postgres
def connect_postgres(db_host, db_port, db_user, db_pass, db_name):
    try:
        # Connect to an existing database
        connection = psycopg2.connect(host = db_host,
                                      port = db_port,
                                      user = db_user,
```

```
                                                password = db_pass,
                                                database = db_name)
          # Set auto-commit
          connection.set_isolation_level(ISOLATION_LEVEL_AUTOCOMMIT);
          # Create a cursor to perform database operations
          cur = connection.cursor()
          # Print PostgreSQL details
          print("PostgreSQL server information")
          print(connection.get_dsn_parameters(), "\n")
          # Executing a SQL query
          cur.execute("SELECT version();")
          # Fetch result
          record = cur.fetchone()
          print("You are connected to - ", record, "\n")

     except (Exception, Error) as error:
          print("Error while connecting to PostgreSQL", error)
     else:
          return cur

# connect to postgres
cur = connect_postgres(db_host, db_port, db_user, db_pass, db_name)

# get dir path
path = os.getcwd() + '\\all_queries\\initial_queries'
files = os.listdir(path)
print(path)

# function to get full abosolute path files in directory
def get_absolute_path(d):
     return [os.path.join(d, f) for f in os.listdir(d)]

# get full abosolute path files in directory
files_abs_path = [p.replace('\\', '/') for p in get_absolute_path(
    path)]
print("Total files:", len(files_abs_path))
print("First few files...")
files_abs_path[:5]

# printing start datetime
now = datetime.now()
current_time = now.strftime("%H:%M:%S")
print("Run Test Start =", current_time)

# perform run test on each query
# save results in text file
script_num = 1
script_errors = 0
for sql_script in files_abs_path:
     textfile = open("query_run_test_result.txt", "a")
     textfile2 = open("query_run_test_query_errors.txt", "a")
     clear_output(wait = True)
     try:
          cur.execute(
               open(sql_script, "r").read()
          )
     except Exception as e:
          script_errors += 1
```

```python
        outcome = f"Error, Message: {e}"
        print(sql_script)
        print(outcome)
        textfile.write(sql_script + "\n")
        textfile.write(outcome + "\n\n")
        # for tracking errors alone
        textfile2.write(sql_script + "\n")
        textfile2.write(outcome + "\n\n")
    else:
        outcome = f"Success, Message: {cur.statusmessage}"
        print(sql_script)
        print(outcome)
        textfile.write(sql_script + "\n")
        textfile.write(outcome + "\n\n")

    script_num += 1
    textfile.close()
    textfile2.close()


# printing end datetime
now = datetime.now()
current_time = now.strftime("%H:%M:%S")
print("Run Test End =", current_time)

# close connection to db
cur.close()

# check total amount of query errors
print(f"We have a total of {script_errors} queries with error")

# End of script.
```

Listing A.2: Query_Run_Test Script

```python
# TPCDS: Query Performance Test Script

# importing libraries
import sys, os
import psycopg2
import numpy as np
import pandas as pd
from psycopg2 import Error
from psycopg2.extensions import ISOLATION_LEVEL_AUTOCOMMIT
from datetime import datetime
from IPython.display import clear_output

# scale factor being tested
sf = 'sf_1'

# set up connection variables
db_host = "localhost"
db_port = "5432"
db_user = "postgres"
db_pass = "password"
db_name = "tpcds"

# function to connect with postgres
def connect_postgres(db_host, db_port, db_user, db_pass, db_name):
```

```
    try:
        # Connect to an existing database
        connection = psycopg2.connect(host = db_host,
                                      port = db_port,
                                      user = db_user,
                                      password = db_pass,
                                      database = db_name)
        # Set auto-commit
        connection.set_isolation_level(ISOLATION_LEVEL_AUTOCOMMIT);
        # Create a cursor to perform database operations
        cur = connection.cursor()
        # Print PostgreSQL details
        print("PostgreSQL server information")
        print(connection.get_dsn_parameters(), "\n")
        # Executing a SQL query
        cur.execute("SELECT version();")
        # Fetch result
        record = cur.fetchone()
        print("You are connected to - ", record, "\n")

    except (Exception, Error) as error:
        print("Error while connecting to PostgreSQL", error)
    else:
        return cur

# connect to postgres
cur = connect_postgres(db_host, db_port, db_user, db_pass, db_name)

# get dir path
path = os.getcwd() + '\\all_queries\\optimized_queries_final'
files = os.listdir(path)
print(path)

# function to get full abosolute path files in directory
def get_absolute_path(d):
    return [os.path.join(d, f) for f in os.listdir(d)]

# get full abosolute path files in directory
files_abs_path = [p.replace('\\', '/') for p in get_absolute_path(path)]
print("Total files:", len(files_abs_path))
print("First few files...")
files_abs_path[:5]

# setup dataframe for recording query execution run times
query_name_list = []
for i in range(len(files)):
    query_name_list.append("Q" + files[i][-6:-4])
query_name_dict = {'query':query_name_list}
exec_details_df = pd.DataFrame(query_name_dict)

# get the date-time before all 99 queries have run (with iterations if chosen)

run_start_default = datetime.now()
# dd/mm/YY H:M:S
run_start = run_start_default.strftime("%d/%m/%Y %H:%M:%S")
print("Overall Run Start:", run_start)
```

```python

# run all 99 queries in sequence, and multiple iterations if chosen
# save query result table output
# save query execution run time (for all iterations)
q_errors = 0
exec_details = []
# choose number of iterations to run
n_iterations = 3

for i in range(1, n_iterations + 1):
    clear_output(wait = True)
    print(f'Iteration {i}\n')
    q_index = 0
    exec_details = []
    iteration_start = datetime.now()
    for sql_script in files_abs_path:

        exec_start = datetime.now()
        try:
            cur.execute(
                open(sql_script, "r").read()
            )
        except Exception as e:
            q_errors += 1
            outcome = "Error"
        else:
            outcome = "Success"

        exec_end = datetime.now()
        exec_run_time = "{:.2f}".format((exec_end - exec_start).total_seconds())
        query_num = query_name_list[q_index]
        print(f'{query_num}: Success, Execution Time: {exec_run_time}s')
        exec_details.append(exec_run_time)

        # load table output to csv file (on first iteration only)
        if i == 1:
            df = pd.DataFrame(cur.fetchall(), columns = [desc[0] for desc in cur.description])
            df.to_csv(f'performance_test/{sf}/{query_num}.csv', index = False)
        else:
            pass

        q_index += 1

    iteration_end = datetime.now()
    iteration_run_time = "{:.2f}".format(((iteration_end - iteration_start).total_seconds()) / 3600)
    print(f'\n{sf.upper()}, Iteration {i}, Total run time for the 99 queries: {iteration_run_time}hr')

    # append iteration execution details to dataframe
    exec_details_df[f'exec_time_iter_{i}'] = np.array(exec_details)

# check total amount of query errors
print(f"We have a total of {q_errors} queries with error")
```

```
# get the date-time after all 99 queries have run (with iterations
    if chosen)
run_end_default = datetime.now()
# dd/mm/YY H:M:S
run_end = run_end_default.strftime("%d/%m/%Y %H:%M:%S")
print(f"Overall Run End (with {n_iterations} iterations):", run_end
    )


# get the total run time (in hours) for all 99 queries to complete
    (with iterations if chosen)
total_run_time = "{:.2f}".format(((run_end_default -
    run_start_default).total_seconds()) / 3600)
print(f'Total run time for the 99 queries (with {n_iterations}
    iterations): {total_run_time}hr')

# full details on query execution times (including iterations &
    average)
# load execution details to csv
exec_details_df['avg_exec_time'] = np.round(exec_details_df.iloc[:,
     1:].apply(pd.to_numeric).mean(axis = 1), 2)
exec_details_df.to_csv(f'performance_test/{sf}/exec_time_details_{
    sf}.csv', index = False)
exec_details_df

# close connection to db
cur.close()

# End of script.
```

Listing A.3: Query_Performance_Test Script

A.2 Indexes

```
create index if not exists idx_cs_cus_sk
on public.catalog_sales
using hash (cs_bill_customer_sk);

create index if not exists idx_cs_sold_date_sk
on public.catalog_sales
using hash (cs_sold_date_sk);

create index if not exists idx_cust_cust_id
on public.customer
using btree (c_customer_id);

create index if not exists idx_date_dyear
on public.date_dim
using btree (d_year);

create index if not exists idx_ss_cus_sk
on public.store_sales
using hash (ss_customer_sk);

create index if not exists idx_ss_sold_date_sk
```

```
on public.store_sales
using hash (ss_sold_date_sk);

create index if not exists idx_ws_cus_sk
on public.web_sales
using hash (ws_bill_customer_sk);

create index if not exists idx_ws_sold_date_sk
on public.web_sales
using hash (ws_sold_date_sk);
```

Listing A.4: Index_Setup Script

A.3 Optimised Queries

```
-- Query 01

with customer_total_return as (
  select sr_customer_sk as ctr_customer_sk,
         sr_store_sk as ctr_store_sk,
         sum(sr_return_amt_inc_tax) as ctr_total_return
  from store_returns,
       date_dim
  where sr_returned_date_sk = d_date_sk
    and d_year = 1999
  group by sr_customer_sk,
         sr_store_sk
)

, average_cust_returns as (
  select
    ctr_store_sk as store_sk,
    avg(ctr_total_return) * 1.2 as ctr_avg_return
  from customer_total_return
  group by ctr_store_sk
)

select
  c_customer_id
from customer_total_return
inner join average_cust_returns
  on ctr_store_sk = store_sk
inner join store
  on ctr_store_sk = s_store_sk
inner join customer
  on ctr_customer_sk = c_customer_sk
where s_state = 'TN'
and ctr_total_return > ctr_avg_return
order by c_customer_id
limit 100;
```

Listing A.5: Query_04

```
-- Query 04

with store_year_total as (
 select c_customer_id customer_id
```

```
       ,c_first_name customer_first_name
       ,c_last_name customer_last_name
       ,c_preferred_cust_flag customer_preferred_cust_flag
       ,c_birth_country customer_birth_country
       ,c_login customer_login
       ,c_email_address customer_email_address
       ,d_year dyear
       ,sum(((ss_ext_list_price-ss_ext_wholesale_cost-
   ss_ext_discount_amt)+ss_ext_sales_price)/2) year_total
 from customer
     ,store_sales
     ,date_dim
 where c_customer_sk = ss_customer_sk
   and ss_sold_date_sk = d_date_sk
 group by c_customer_id
         ,c_first_name
         ,c_last_name
         ,c_preferred_cust_flag
         ,c_birth_country
         ,c_login
         ,c_email_address
         ,d_year
)

, catalog_year_total as (
 select c_customer_id customer_id
       ,c_first_name customer_first_name
       ,c_last_name customer_last_name
       ,c_preferred_cust_flag customer_preferred_cust_flag
       ,c_birth_country customer_birth_country
       ,c_login customer_login
       ,c_email_address customer_email_address
       ,d_year dyear
       ,sum((((cs_ext_list_price-cs_ext_wholesale_cost-
   cs_ext_discount_amt)+cs_ext_sales_price)/2) ) year_total
 from customer
     ,catalog_sales
     ,date_dim
 where c_customer_sk = cs_bill_customer_sk
   and cs_sold_date_sk = d_date_sk
 group by c_customer_id
         ,c_first_name
         ,c_last_name
         ,c_preferred_cust_flag
         ,c_birth_country
         ,c_login
         ,c_email_address
         ,d_year
)

, web_year_total as (
 select c_customer_id customer_id
       ,c_first_name customer_first_name
       ,c_last_name customer_last_name
       ,c_preferred_cust_flag customer_preferred_cust_flag
       ,c_birth_country customer_birth_country
       ,c_login customer_login
       ,c_email_address customer_email_address
```

```
       ,d_year dyear
       ,sum((((ws_ext_list_price-ws_ext_wholesale_cost-
   ws_ext_discount_amt)+ws_ext_sales_price)/2) ) year_total
 from customer
     ,web_sales
     ,date_dim
 where c_customer_sk = ws_bill_customer_sk
   and ws_sold_date_sk = d_date_sk
 group by c_customer_id
         ,c_first_name
         ,c_last_name
         ,c_preferred_cust_flag
         ,c_birth_country
         ,c_login
         ,c_email_address
         ,d_year
)

, t_s_firstyear as (
  select customer_id
       ,customer_first_name
       ,customer_last_name
     ,customer_email_address
       ,year_total
  from store_year_total
  where dyear = 2001
  and year_total  > 0
)

, t_s_secyear as (
  select customer_id
       ,customer_first_name
       ,customer_last_name
     ,customer_email_address
       ,year_total
  from store_year_total
  where dyear = 2001+1
)

, t_c_firstyear as (
  select customer_id
       ,customer_first_name
       ,customer_last_name
     ,customer_email_address
       ,year_total
  from catalog_year_total
  where dyear = 2001
  and year_total  > 0
)

, t_c_secyear as (
  select customer_id
       ,customer_first_name
       ,customer_last_name
     ,customer_email_address
       ,year_total
  from catalog_year_total
  where dyear = 2001+1
```

```sql
)

, t_w_firstyear as (
  select customer_id
        ,customer_first_name
        ,customer_last_name
      ,customer_email_address
        ,year_total
  from web_year_total
  where dyear = 2001
  and year_total  > 0
)

, t_w_secyear as (
  select customer_id
        ,customer_first_name
        ,customer_last_name
      ,customer_email_address
        ,year_total
  from web_year_total
  where dyear = 2001+1
)

select
  t_s_secyear.customer_id
  ,t_s_secyear.customer_first_name
  ,t_s_secyear.customer_last_name
  ,t_s_secyear.customer_email_address
 from t_s_firstyear
 inner join t_s_secyear
  on t_s_firstyear.customer_id = t_s_secyear.customer_id
 inner join t_w_firstyear
  on t_s_firstyear.customer_id = t_w_firstyear.customer_id
 inner join t_w_secyear
  on t_w_firstyear.customer_id = t_w_secyear.customer_id
 inner join t_c_firstyear
  on t_s_firstyear.customer_id = t_c_firstyear.customer_id
 inner join t_c_secyear
  on t_s_firstyear.customer_id = t_c_secyear.customer_id
where
  (t_c_secyear.year_total / nullif(t_c_firstyear.year_total, 0))
  >
  (t_s_secyear.year_total / nullif(t_s_firstyear.year_total, 0))
  and
  (t_c_secyear.year_total / nullif(t_c_firstyear.year_total, 0))
  >
  (t_w_secyear.year_total / nullif(t_w_firstyear.year_total, 0))
order by t_s_secyear.customer_id
        ,t_s_secyear.customer_first_name
        ,t_s_secyear.customer_last_name
        ,t_s_secyear.customer_email_address
limit 100;
```

Listing A.6: Query_04

```sql
-- Query 06

with average_item_price as (
  select
```

```sql
    i_category as category,
    avg(i_current_price) * 1.2 as avg_item_price
  from item
  group by i_category
)

select  a.ca_state state, count(*) cnt
from customer_address a
inner join customer c
  on  a.ca_address_sk = c.c_current_addr_sk
inner join store_sales s
  on c.c_customer_sk = s.ss_customer_sk
inner join date_dim d
  on s.ss_sold_date_sk = d.d_date_sk
inner join item i
  on s.ss_item_sk = i.i_item_sk
inner join average_item_price aip
  on i.i_category = aip.category
where
  d.d_year = 1998
  and d.d_moy = 3
  and i.i_current_price > aip.avg_item_price
group by a.ca_state
having count(*) >= 10
order by cnt, a.ca_state
limit 100;
```

Listing A.7: Query_06

```sql
-- Query 11

with store_year_total as (
 select c_customer_id customer_id
       ,c_first_name customer_first_name
       ,c_last_name customer_last_name
       ,c_preferred_cust_flag customer_preferred_cust_flag
       ,c_birth_country customer_birth_country
       ,c_login customer_login
       ,c_email_address customer_email_address
       ,d_year dyear
       ,sum(ss_ext_list_price-ss_ext_discount_amt) year_total
 from customer
     ,store_sales
     ,date_dim
 where c_customer_sk = ss_customer_sk
   and ss_sold_date_sk = d_date_sk
 group by c_customer_id
         ,c_first_name
         ,c_last_name
         ,c_preferred_cust_flag
         ,c_birth_country
         ,c_login
         ,c_email_address
         ,d_year
)

, web_year_total as (
 select c_customer_id customer_id
       ,c_first_name customer_first_name
```

```
       ,c_last_name customer_last_name
       ,c_preferred_cust_flag customer_preferred_cust_flag
       ,c_birth_country customer_birth_country
       ,c_login customer_login
       ,c_email_address customer_email_address
       ,d_year dyear
       ,sum(ws_ext_list_price-ws_ext_discount_amt) year_total
 from customer
     ,web_sales
     ,date_dim
 where c_customer_sk = ws_bill_customer_sk
   and ws_sold_date_sk = d_date_sk
 group by c_customer_id
         ,c_first_name
         ,c_last_name
         ,c_preferred_cust_flag
         ,c_birth_country
         ,c_login
         ,c_email_address
         ,d_year
)


, t_s_firstyear as (
  select customer_id
       ,customer_first_name
       ,customer_last_name
     ,customer_email_address
       ,year_total
  from store_year_total
  where dyear = 1999
  and year_total  > 0
)

, t_s_secyear as (
  select customer_id
       ,customer_first_name
       ,customer_last_name
     ,customer_email_address
       ,year_total
  from store_year_total
  where dyear = 1999+1
)

, t_w_firstyear as (
  select customer_id
       ,customer_first_name
       ,customer_last_name
     ,customer_email_address
       ,year_total
  from web_year_total
  where dyear = 1999
  and year_total  > 0
)

, t_w_secyear as (
  select customer_id
       ,customer_first_name
```

```
        ,customer_last_name
      ,customer_email_address
        ,year_total
  from web_year_total
  where dyear = 1999+1
)

select
  t_s_secyear.customer_id
  ,t_s_secyear.customer_first_name
  ,t_s_secyear.customer_last_name
  ,t_s_secyear.customer_email_address
 from t_s_firstyear
 inner join t_s_secyear
  on t_s_firstyear.customer_id = t_s_secyear.customer_id
 inner join t_w_firstyear
  on t_s_firstyear.customer_id = t_w_firstyear.customer_id
 inner join t_w_secyear
  on t_w_firstyear.customer_id = t_w_secyear.customer_id
where
  case
    when t_w_firstyear.year_total > 0 then t_w_secyear.year_total /
    t_w_firstyear.year_total
  else 0.0
  end
    >
  case
    when t_s_firstyear.year_total > 0 then t_s_secyear.year_total /
    t_s_firstyear.year_total
    else 0.0
  end
order by t_s_secyear.customer_id
         ,t_s_secyear.customer_first_name
         ,t_s_secyear.customer_last_name
         ,t_s_secyear.customer_email_address
limit 100;
```

Listing A.8: Query_11

```
-- Query 74

with store_year_total as (
 select c_customer_id customer_id
        ,c_first_name customer_first_name
        ,c_last_name customer_last_name
        ,d_year as year
        ,stddev_samp(ss_net_paid) year_total
 from date_dim
      ,store_sales
      ,customer
 where d_date_sk = ss_sold_date_sk
  and ss_customer_sk = c_customer_sk
   and d_year in (2001,2001+1)
 group by c_customer_id
          ,c_first_name
          ,c_last_name
          ,d_year
)

```

```sql
, web_year_total as (
 select c_customer_id customer_id
       ,c_first_name customer_first_name
       ,c_last_name customer_last_name
       ,d_year as year
       ,stddev_samp(ws_net_paid) year_total
 from date_dim
     ,web_sales
     ,customer
 where d_date_sk = ws_sold_date_sk
  and c_customer_sk = ws_bill_customer_sk
   and ws_sold_date_sk = d_date_sk
   and d_year in (2001,2001+1)
 group by c_customer_id
         ,c_first_name
         ,c_last_name
         ,d_year
)

, t_s_firstyear as (
  select customer_id
       ,customer_first_name
       ,customer_last_name
       ,year_total
  from store_year_total
  where year = 2001
  and year_total  > 0
)

, t_s_secyear as (
  select customer_id
       ,customer_first_name
       ,customer_last_name
       ,year_total
  from store_year_total
  where year = 2001+1
)

, t_w_firstyear as (
  select customer_id
       ,customer_first_name
       ,customer_last_name
       ,year_total
  from web_year_total
  where year = 2001
  and year_total  > 0
)

, t_w_secyear as (
  select customer_id
       ,customer_first_name
       ,customer_last_name
       ,year_total
  from web_year_total
  where year = 2001+1
)

select
```

```
t_s_secyear.customer_id, t_s_secyear.customer_first_name,
    t_s_secyear.customer_last_name
 from t_s_firstyear
 inner join t_s_secyear
  on t_s_firstyear.customer_id = t_s_secyear.customer_id
 inner join t_w_firstyear
  on t_s_firstyear.customer_id = t_w_firstyear.customer_id
 inner join t_w_secyear
  on t_w_firstyear.customer_id = t_w_secyear.customer_id
where
  (t_w_secyear.year_total / nullif(t_w_firstyear.year_total,0))
  >
  (t_s_secyear.year_total / nullif(t_s_firstyear.year_total,0))
order by
   t_s_secyear.customer_last_name
  ,t_s_secyear.customer_first_name
  ,t_s_secyear.customer_id
limit 100;
```

Listing A.9: Query_74

```
-- Query 81

with customer_total_return as (
  select
    cr_returning_customer_sk as ctr_customer_sk ,
    ca_state as ctr_state,
    sum(cr_return_amt_inc_tax) as ctr_total_return
  from
    catalog_returns
    inner join date_dim on cr_returned_date_sk = d_date_sk
    inner join customer_address on cr_returning_addr_sk =
   ca_address_sk
    where
      d_year = 1998
    group by
      cr_returning_customer_sk
      ,ca_state)

, cust_average_return as (
  select
    ctr_state as ctr_state
  ,avg(ctr_total_return) * 1.2 as ctr_avg_return
  from customer_total_return ctr1
  group by ctr_state
  )

select c_customer_id,
  c_salutation,
  c_first_name,
  c_last_name,
  ca_street_number,
  ca_street_name ,
  ca_street_type,
  ca_suite_number,
  ca_city,
  ca_county,
  ca_state,
  ca_zip,
```

```
  ca_country,
  ca_gmt_offset ,
  ca_location_type,
  ctr_total_return
from customer_total_return ctr1
  inner join cust_average_return ctr2 on ctr1.ctr_state = ctr2.
   ctr_state
  inner join customer on ctr_customer_sk = c_customer_sk
  inner join customer_address on c_current_addr_sk = ca_address_sk
where
  ctr1.ctr_total_return > ctr2.ctr_avg_return
  and ca_state = 'TX'
order by c_customer_id,
  c_salutation,
  c_first_name,
  c_last_name,
  ca_street_number,
  ca_street_name ,
  ca_street_type,
  ca_suite_number,
  ca_city,
  ca_county,
  ca_state,
  ca_zip,
  ca_country,
  ca_gmt_offset ,
  ca_location_type,
  ctr_total_return
limit 100;
```
Listing A.10: Query_81