ECOLE POLYTECHNIQUE DE BRUXELLES

2023-2024

INFO-H419 - Data Warehouse

Project 2 TPC-DI Benchmark Using DuckDB

Group:

Jintao Ma(000586557)

Linhan Wang(000585737)

Iyoha Peace Osamuyi(000583313)

Hieu Nguyen Minh(000583782)

Professor

Esteban Zimányi

December 2023

Contents

Listings

List of Figures

Abstract

In this project, we present an in-depth implementation of the TPC-DI Benchmark using DuckDB, a versatile, free, and open-source database from CWI. Our comprehensive report is organized into six key chapters, each addressing crucial aspects of the benchmarking process:

Chapter 1: Sets the stage by outlining the project's goals, requirements, and introduces the programming tools and databases utilized.

Chapter 2: Delves into the fundamentals of ETL (Extract, Transform, and Load) and the intricacies of TPC-DI benchmarking.

Chapter 3 and 4: Detail our setup and configuration, alongside a comprehensive guide on executing tests via Python scripts. This section includes various assumptions and practical code implementations.

Chapter 5: Focuses on data-driven insights, employing numerous charts for comparison, analysis and interpretation,

Chapter 6: Concludes with a summary of our findings and reflections on the project.

For those interested in a more granular exploration of our experimental and analytical approaches, we have made all related codes and scripts accessible through our GitHub repository at `https://github.com/facingfrost/tpcdi-duckdb`.

1 Introduction

1.1 Overview

The Project entails performing TPC-DI Benchmarking on a selected Database Management System which include DuckDB, PostgreSQL, SparkSQl, MySQL etc. We can execute the benchmarking using a chosen data integration tool such as Python scripts, Apache Airflow, Pentaho Data Integration, Oracle Data Integrator, Talend Data Studio, or SQL scripts. The Benchmarking is done with several scale factors which determine the size of the resulting data warehouse and then implement some metrics to evaluate the performance of the database.
The project is carried out in groups of 4 persons and delivers a report in pdf file explaining the essential aspects of your implementation, and a zip file containing the code of your implementation, with all necessary instructions to be able to replicate your implementation. This report uses Python script for the data integration tool and DuckDB as the preferred choice of Database Management System on which the TPC-DI benchmark is performed.

1.2 Aim and Objectives

The aim of this project is to execute the TPC-DI benchmark, which involves the extraction, transformation, and loading (ETL) of data from an On-Line Transaction Processing (OLTP) system and various other data sources into a chosen data warehouse (DuckDB), with the following objectives include:

- To learn and understand the steps required for a complete ETL (Extract, Transform, Load) implementation.
- Evaluate benchmarking performance and conducting an analysis of the outcomes.

1.3 Technology Used

1.3.1 DuckDB

DuckDB is an in-process SQL OLAP database management system.[1] The first open-source version of it was released by Hannes Mühleisen and Mark Raasveldt,who were researchers in the Database Architectures research group at Centrum Wiskunde & Informatica (CWI) in 2019.[RM19]It was the first purpose built in-process Online Analytical Processing(OLAP)-database management system.[2]It was designed to be fast, reliable and easy to use. There have been some important features of this tool:

1.3.2 Simple and portable

For DuckDB, there is no DBMS server software to install, update and maintain. It does not run as a separate process, but completely embedded within a host process. For the analytical use cases that DuckDB targets, this has the additional advantage of high-speed data transfer to and from the database. In some cases, DuckDB can process foreign data without copying. We can use DuckDB without installing, updating, or maintaining separate DBMS server software. It doesn't run as an independent process but is entirely integrated within a host process. This approach provides added benefits for analytical use cases, notably enabling high-speed data transfer to and from the database.

[1] https://duckdb.org/

[2] https://www.cwi.nl/en/news/cwi-spin-off-company-duckdb-labs-provides-solutions-for-fast-database-analytics

Feature-rich

DuckDB enables users to run complex SQL queries and provides APIs for Java, C, C++, and more. It is also deeply integrated into Python and R, enabling users to conduct efficient interactive data analysis; thus, you can interact with DuckDB from your preferred programming language. There is also access to extra SQL keywords that make SQL queries easier to write, such as EXCLUDE, REPLACE, and ALL.

Fast

DuckDB can support the Online analytical processing (OLAP).It features a query execution engine that employs columnar vectorization. While a few queries may still be interpreted, a substantial batch of values, known as a "vector," is processed in a single operation. This approach significantly minimizes the overhead found in conventional systems like PostgreSQL, MySQL, or SQLite, which handle each row sequentially. The utilization of vectorized query execution significantly enhances performance in OLAP queries. They used an interactive SQL shell to demonstrate the capabilities of an analytical database in the browser[Koh+22].

Free and Extensible

Because the DuckDB is Open Source, all of the source code is freely available on GitHub. You can acess it by click the link: https://github.com/duckdb They also invite developers from anyone to take issues.

1.3.3 Python

Python is an interpreted, general-purpose, interactive, object-oriented programming language. It incorporates modules, exceptions, dynamic typing, very high level dynamic data types, and classes. It supports multiple programming paradigms beyond object-oriented programming, such as procedural and functional programming. Python combines remarkable power with very clear syntax. It has interfaces to many system calls and libraries, as well as to various window systems, and is extensible in C or C++ [Pyt23]. In this project, we used python to write scripts for creating

the database, loading of tables in our database, querying and performing various evaluation metrics. Also, Jupyter notebook with matplotlib was used to visualize our data. Python version 3.10 was used in this project.

1.3.4 Code Editor: Microsoft Visual Code

Visual Studio Code is a lightweight but powerful source code editor which runs on your desktop and is available for Windows, macOS and Linux. It comes with built-in support for JavaScript, TypeScript and Node.js and has a rich ecosystem of extensions for other languages and runtimes (such as C++, CSharp, Java, Python, PHP, Go, .NET) [Cod23]. It was useful in this project to write our python scripts and running dsdgen and dsqgen on command line.

1.3.5 Tools and materials

The main reading material for conducting our experiments is TPC BenchMark DI Standard Specification Version 1.1.0 [TPC14]. Besides, we follow the tutorial in DuckDB documentation[3].

1.4 Limitations

One Limitation was that the SQL Keyword "**ALTER**" hasn't been fully implemented to handle referential integrity constraints. Time was a limiting factor for us to thoroughly expose the objectives stated above.

[3] https://duckdb.org/docs/archive/0.9.1/

2 ETL and TPC-DI Benchmarking

2.1 ETL

According to Judith Awiti, Alejandro A. Vaisman and Zimányi [JZ20], ETL is defined as the Extraction, Transformation and Load process, in which the data go through three steps: extracting data from internal and external sources of an organization, transforming these data, and loading them into a data warehouse. The Data Integration (DI) should combine data from various sources into a single one to solve possible conflicts of data meaning and data shape.

The ETL process plays a pivotal role in data warehouse mantenance. By making use of ETL on the data, organizations can consolidate information from diverse sources with different formats, and get a unified view of their data. Besides facilitating the data integration, ETL is a foundation for analytics, reporting, and business intelligence, enabling enterprises to extract valuable insights from their huge data. The diagram 2.1 below shows the ETL process flow:

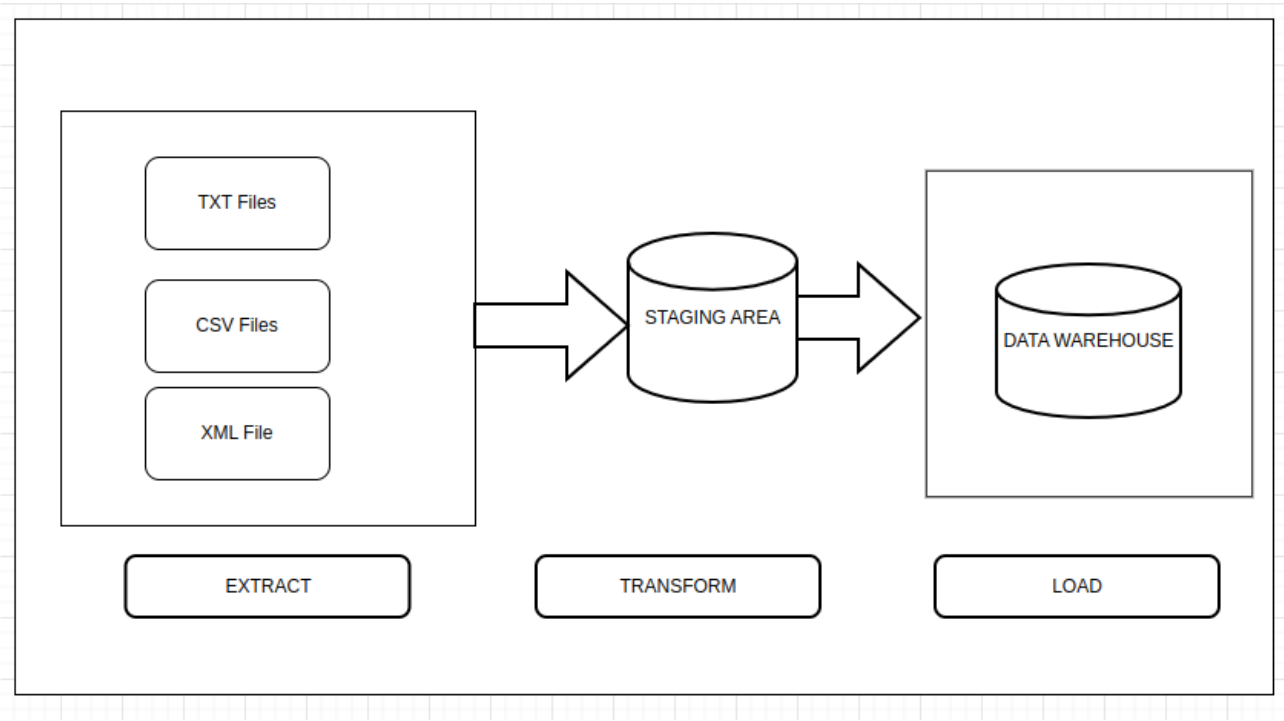


Figure 2.1: ETL Process FLow

2.2 Benchmarking

Benchmarking is a process of comparing and evaluating an organization's performance against an external standard or best practice model. It involves measuring and analyzing various aspects of an organization's processes, performance, and practices against those of other similar organizations, with the goal of identifying areas for improvement and setting goals for achieving better results [Mah21]. Benchmarking a database is the process of performing well defined tests on that particular database for the purpose of evaluating its performance [Sam21]. Benchmark test results facilitate means for cross platform comparisons of various database management systems by providing valuable information to database professionals on whether to utilise a particular database product. Hence, we are carrying and evaluating the performance of DuckDB against the TPC-DS Benchmarking standard.

Transaction Processing Performance Council (TPC) is a non-profit organization founded to define transaction processing and database benchmarks and to disseminate objective, verifiable TPC performance data to the industry [1]. In terms of data warehouses, TPC provide two benchmarks: TPC-DS and TPC-DI for the evaluation of decision support and data integration, respectively.

[1] https://en.wikipedia.org/wiki/Transaction_Processing_Performance_Council

2.3 TPC-DI

The TPC Benchmark™ DI (TPC-DI) serves as a model for data integration benchmarking, covering all aspects of a fundamental data integration setup [TPC14]. This encompasses the complete Extract, Transform, Load (ETL) process and regular data upkeep. TPC-DI offers a detailed framework for data integration, effectively assessing the performance of the System Under Test (SUT). It is a typical data integration platform, aligning with industry standards and distinguished by its thorough method of evaluating data integration proficiency. These components are characterized by:

- Handling and transferring substantial amounts of data.
- Diverse transformations applied to the dataset.
- Incorporation of both historical and incremental data updates.
- Adherence to data consistency standards.
- Integration of various data sources in different formats.
- Management of numerous data tables, each with distinct data types, attributes, and interrelations

Conversely, the operational model for the TPC-DI benchmark is structured in the following manner:

- The **source data** is autonomously generated and stored in flat file formats.
- The **transformation phase** kicks off with the System Under Test (SUT) accessing and reading this source data.
- During transformation, the s**ource data undergoes validation** and is methodically structured for efficient loading into a Data Warehouse (DWH).
- This **process reaches completion** once all the source data has been transformed and is successfully stored in the DWH.

The **performance metric** of the benchmark is a throughput measure, the **Number of Source data rows processed per second**.

2.4 DIGEN

DIGen, the automated data generator supplied by the TPC-DI benchmark, serves a crucial role in generating Source Data and audit details, and it's the exclusively endorsed method for this purpose. It provides detailed statistics regarding the data generation process.

The data is generated from the TPC-DI tool in the following way:

- **Requirement**: DIGen - Data Generation utility v1.1.0 used for generating source data for the TPC-DI benchmark.
- **Dependencies**:
  - Requires Java SE 8 or above
  - PDGF which is located in the same directory as the DIGen
- Command line Usage:

```
$ java -jar DIGen.jar -sf <scalefactor: 3,5,7,9>  -o <directory:>
```

Listing 2.1: Data generation.

2.5 Phases of operation

The TPC-DI benchmark, reflecting real-world Data Integration (DI) applications, involves two primary processes: a historical load and incremental updates. The historical load typically occurs during the initial creation or restructuring of a data warehouse, while incremental updates represent the ongoing addition of new data. The TPC-DI benchmark specifically models daily updates within a constrained time frame, often overnight, to fit with other maintenance activities. These processes are crucial in handling large volumes of historical data within limited time windows. The benchmark encompasses several phases executed sequentially:

- Preparation Phase:
  - **Data Generation:** Utilizing a data generator (as detailed in Clause 6), this phase involves creating data either directly in the Staging Area or elsewhere before transferring it to the Staging Area. This process is not timed for the benchmark.

  - **Data Warehouse Creation:** Involves setting up the Data Warehouse database and tables, including disk space allocation. This setup is not considered a DI operation and thus is not timed.
  - **Data Integration Preparation:** This step varies among implementations and includes preparing and configuring the data integration software. It is not timed for the benchmark.

- **Historical Load Phase:** This phase, which is timed for the benchmark, involves different transformations than incremental updates. It populates initially empty destination tables with new data. Source files may differ in their ordering properties compared to incremental updates, and the Historical Load typically uses a larger data set. After this phase, a Validation Query is run to collect information for the automated audit phase.

- **Incremental Update Phase:** Distinct from the Historical Load, this phase deals with transformations of CDC extracts from the OLTP database, representing changes since the last extract. It includes two required phases to ensure repeatability, each with a completion time of 30-60 minutes. A Validation Query follows each update for correctness verification in the audit phase.

- **Automated Audit Phase:** This final phase, not timed, involves querying the Data Warehouse to conduct extensive tests on the resultant data and generate a report. All tests must pass for the run to be considered valid.

Each of these phases is integral to the TPC-DI benchmark, emulating the complexities and constraints of real-world data integration tasks. However, for this study, we implement only the Historical load phase which contains the following and is integrated in no particular order:

1. DimDate
2. DimTime
3. StatusType
4. TaxRate
5. TradeType
6. DimBroker

7. DimCompany
8. DimCustomer
9. DimAccount
10. DimSecurity
11. DimTrade
12. FactCashBalances
13. FactMarketHistory
14. FactWatches
15. Industry
16. Financial
17. Prospect

3 System Setting and Data Generation

3.1 Hardware Specification

Some members of our group used Windows OS, MacOS and Ubuntu OS so we installed DuckDB directly on our computers . The Apple MacBook was the main machine for the benchmark implementation as all the results were obtained from it.

In order to carry out the benchmark experimentation, Apple Macbook and HP EliteBook 840 G5 were used. Table 3.1 shows the hardware specification of this computer:

Name	**HP EliteBook 840 G5**	**Apple Macbook**
Operating System	Ubuntu 22.04.3 LTS	MacOS 13.0
System Type	64 Bits	64 Bits
CPU	Intel® Core™ i7-8650U	Apple M1 Pro
CPU Frequency	1.90GHz × 8	3.2 GHz
RAM capacity	24 GB	16 GB
RAM type	DDR4	SDRAM
Hard disk capacity	512 GB	512 GB

Table 3.1: System Specifications.

3.2 Setup for Project

Our implementation was inspired by a partial implementation of TPC-DI in PostgreSQL [SG22].

3.2.1 DuckDB installation

We use `pip`[1] - the package installer for Python - to install the DuckDB.

```
pip install duckdb==0.9.1
```

Listing 3.1: DuckDB installation.

3.2.2 TPC-DI kit installation

We install the TPC-DS kit from the tpcds- toolkit downloaded from TPC website[2].

```
git clone https://github.com/gregrahn/tpcds-kit.git
cd tpcds-kit/tools
make OS=MACOS
```

Listing 3.2: TPC-DS kit installation.

The first command creates a folder tpcds-kit which is a replicate of the tpcds-kit repository. Then we move into the `tools` folder, and run the `make` file to build TPC-DS tools.

3.3 Database schema creation

The first step in the process is to connect to DuckDB and create a database:

```
import duckdb
```

Listing 3.3: Database setup

Then we use the scripts in `tpcds-kit/tools/tpcdi.sql` to create the database scheme.

[1] https://duckdb.org/docs/installation/index
[2] https://www.tpc.org/tpcdi

3.4 Data generation

3.4.1 Jre and JVM Setting

Before generating data, we need to install Jre and JVM[3].The version shoulde be higher than 1.8.0.

```
C:\Users\Jintao1999>java -version java version "1.8.0_172"
Java(TM) SE Runtime Environment(build 1.8.0_172-b11)
Java HotSpot(TM) 64-Bit Server VM (build 25.172-b11, mixed mode)
```

Listing 3.4: Jre and JVM Setting.

3.4.2 Data generation

We generated data for evaluation with the TPC-DI program dsdgen. At this step, we are in the `tpcds-kit/tools` folder. We needed to Unzip and change the 'PDGF' to'pdgf'. We found scale factors must be in range:3-2147.So we set the minimum scale factors as 3.

Then We ran the following command lines to generate data in 3 scale factors and store them in the `generated-data` folder.

```
run java -jar DIGen.jar -o D:\tpcdi\data -sf 3  # SF 3
run java -jar DIGen.jar -o D:\tpcdi\data -sf 5  # SF 5
run java -jar DIGen.jar -o D:\tpcdi\data -sf 7  # SF 7
run java -jar DIGen.jar -o D:\tpcdi\data -sf 9  # SF 9
```

Listing 3.5: Data generation.

[3]https://www.java.com/en/download/manual.jsp

4 Benchmark Implementation Process

4.1 Implementing TPC-DI on DuckDB

Figure 4.1 shows a business process model depicting a brief rundown of the implementation the TPC-DI benchmark on DuckDB.

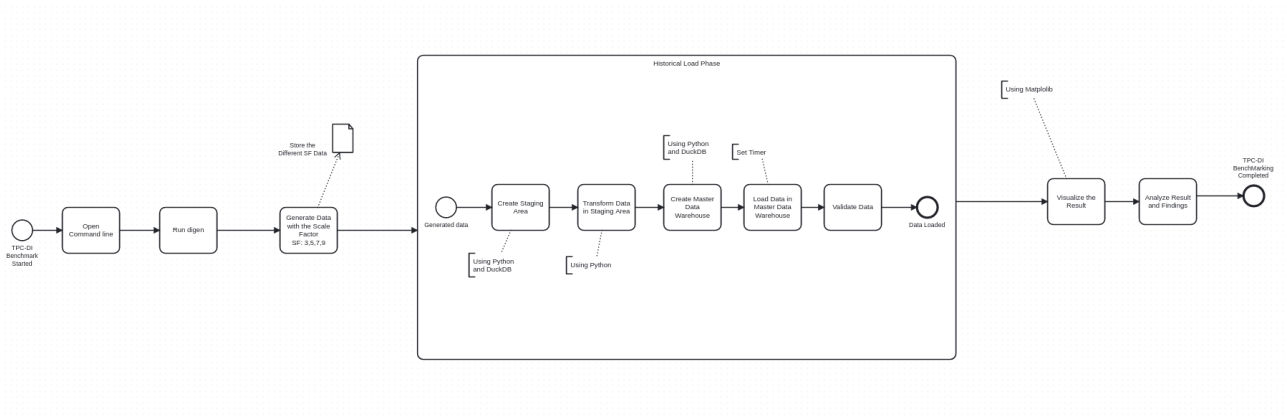


Figure 4.1: TPC-DI Benchmarking BPMN Diagram

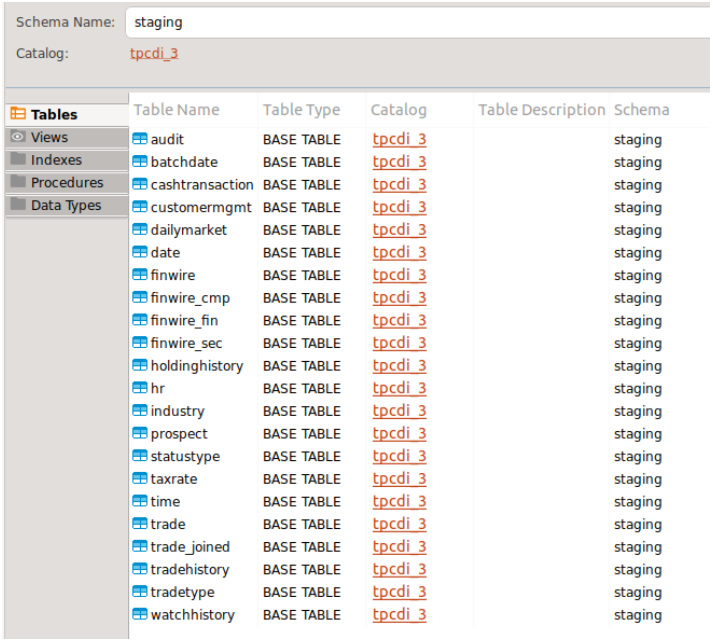
From Figure 4.1, some of the process that was done include:

- Using DIGen Utility from the TPC-DI tool kit to obtain load data for all our scale factor
- DuckDB is created for each selected SF with 2 schemas:
  - Staging Area: holds the initially transformed data
  - Master data warehouse
- Python Scripts was used for running the ETL tasks

- Visualize all outputs.
- Benchmarking results are obtained and analyzed.

The implementation of the TPC-DI benchmarking were done in different stages, which include:

1. Create staging schema as seen in figure 4.2.
2. Create master schema as seen in figure 4.3.
3. Extract data and populate into staging schema.
4. Transform and load into master schema.



Schema Name: staging
Catalog: tpcdi_3

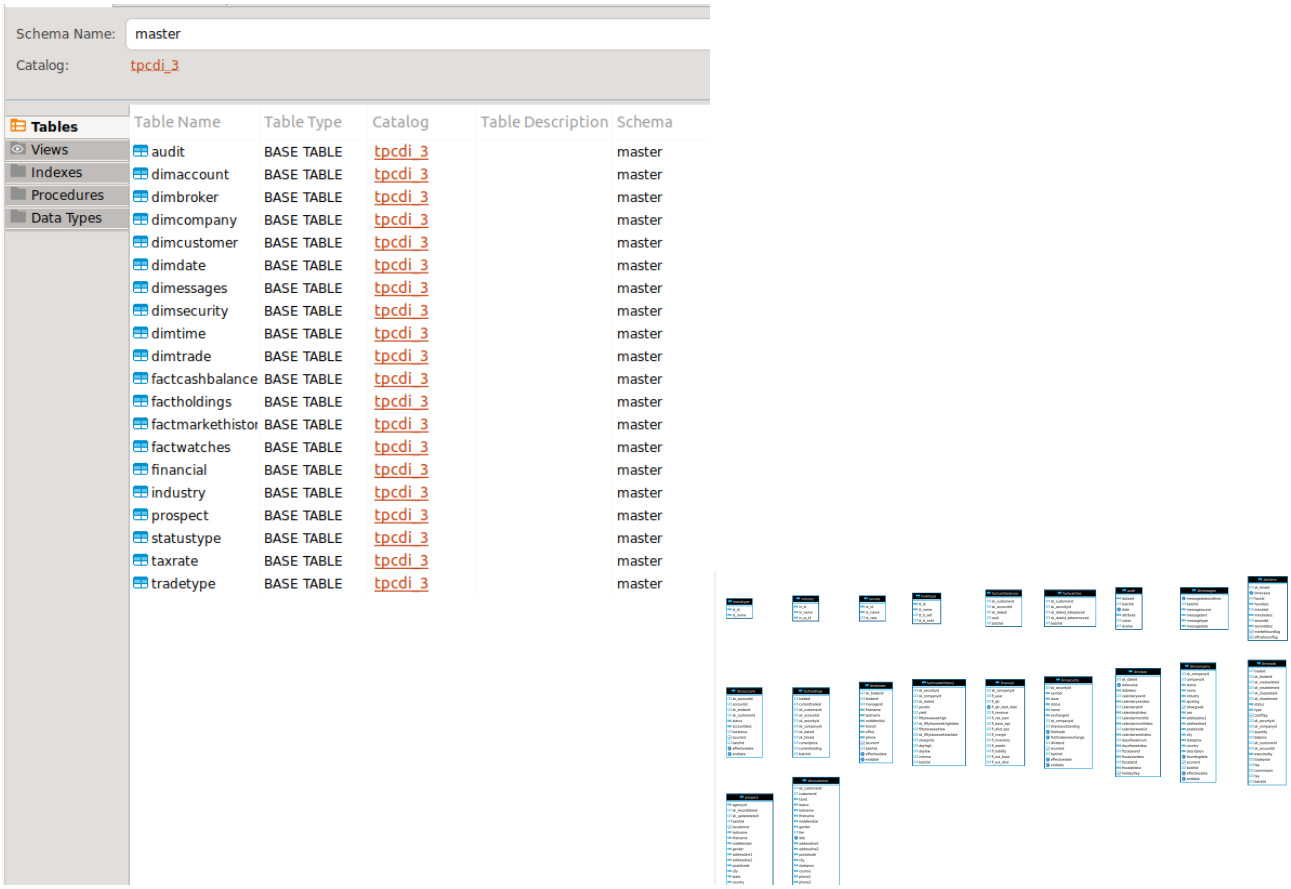
Tables
Views
Indexes
Procedures
Data Types

Table Name	Table Type	Catalog	Table Description	Schema
audit	BASE TABLE	tpcdi_3		staging
batchdate	BASE TABLE	tpcdi_3		staging
cashtransaction	BASE TABLE	tpcdi_3		staging
customermgmt	BASE TABLE	tpcdi_3		staging
dailymarket	BASE TABLE	tpcdi_3		staging
date	BASE TABLE	tpcdi_3		staging
finwire	BASE TABLE	tpcdi_3		staging
finwire_cmp	BASE TABLE	tpcdi_3		staging
finwire_fin	BASE TABLE	tpcdi_3		staging
finwire_sec	BASE TABLE	tpcdi_3		staging
holdinghistory	BASE TABLE	tpcdi_3		staging
hr	BASE TABLE	tpcdi_3		staging
industry	BASE TABLE	tpcdi_3		staging
prospect	BASE TABLE	tpcdi_3		staging
statustype	BASE TABLE	tpcdi_3		staging
taxrate	BASE TABLE	tpcdi_3		staging
time	BASE TABLE	tpcdi_3		staging
trade	BASE TABLE	tpcdi_3		staging
trade_joined	BASE TABLE	tpcdi_3		staging
tradehistory	BASE TABLE	tpcdi_3		staging
tradetype	BASE TABLE	tpcdi_3		staging
watchhistory	BASE TABLE	tpcdi_3		staging

(a) Staging Schema

(b) Staging ERD

Figure 4.2: Staging Database Schema



Schema Name: master
Catalog: tpcdi_3

Tables
Views
Indexes
Procedures
Data Types

Table Name	Table Type	Catalog	Table Description	Schema
audit	BASE TABLE	tpcdi_3		master
dimaccount	BASE TABLE	tpcdi_3		master
dimbroker	BASE TABLE	tpcdi_3		master
dimcompany	BASE TABLE	tpcdi_3		master
dimcustomer	BASE TABLE	tpcdi_3		master
dimdate	BASE TABLE	tpcdi_3		master
dimessages	BASE TABLE	tpcdi_3		master
dimsecurity	BASE TABLE	tpcdi_3		master
dimtime	BASE TABLE	tpcdi_3		master
dimtrade	BASE TABLE	tpcdi_3		master
factcashbalance	BASE TABLE	tpcdi_3		master
factholdings	BASE TABLE	tpcdi_3		master
factmarkethistor	BASE TABLE	tpcdi_3		master
factwatches	BASE TABLE	tpcdi_3		master
financial	BASE TABLE	tpcdi_3		master
industry	BASE TABLE	tpcdi_3		master
prospect	BASE TABLE	tpcdi_3		master
statustype	BASE TABLE	tpcdi_3		master
taxrate	BASE TABLE	tpcdi_3		master
tradetype	BASE TABLE	tpcdi_3		master

(a) Maser Schema (b) Master ERD

Figure 4.3: Master Data Warehouse

To make it more clear, we follow tpc-di-benchmark[1] to divide the whole ETL process into 30 tasks. They are shown in table 4.1 .

4.2 Tasks

4.2.1 Task 1

Task 1 creates staging schema, storing all data from the source file. The staging schema includes 21 tables: batchdate, cashtransaction, customermgmt, dailymarket, date, finwire, finwire_cmp, finwire_sec, finwire_fin, holdinghistory, hr, industry, prospect, statustype, taxrate, time, tradehistory, trade, tradetype, watchhistory and audit.

Part of Task 1 SQL is shown in listing 4.1.

```
1 . . . . . .
```

[1] https://github.com/risg99/tpc-di-benchmark

Number	Name	Type
1	Create staging schema	Create Schema
2	Load txt and csv sources to staging	Extract & Load
3	Load finwire source to staging	Extract & Load
4	Parse finwire and load to seperate tables	Extract & Load
5	Convert customer management source from xml to csv	Extract
6	Load customer management source to staging	Extract & Load
7	Create master schema	Create schema
8	Direct load master.tradetype	Load
9	Direct load master.statustype	Load
10	Direct load master.taxrate	Load
11	Direct load master.industry	Load
12	Transform & load master.dimdate	Transform & Load
13	Transform & load master.dimtime	Transform & Load
14	Transform & load master.dimcompany	Transform & Load
15	Load master.dimessages with alert from master.dimcompany	Transform & Load
16	Transform & load master.dimbroker	Transform & Load
17	Transform & load master.prospect	Transform & Load
18	Transform & load master.dimcustomer	Transform & Load
19	Load master.dimessages with alert from master.dimcustomer	Transform & Load
20	Update master.prospect	Transform & Load
21	Transform & load master.dimaccount	Transform & Load
22	Transform & load master.dimsecurity	Transform & Load
23	Transform & load master.dimtrade	Transform & Load
24	Load master.dimessages with alert from master.dimtrade	Transform & Load
25	Transform & load master.financial	Transform & Load
26	Transform & load master.factcashbalances	Transform & Load
27	Transform & load master.factholdings	Transform & Load
28	Transform & load master.factwatches	Transform & Load
29	Transform & load master.factmarkethistory	Transform & Load
30	Load master.dimessages with alert from master.factmarkethistory	Transform & Load

Table 4.1: Implementation tasks

```
drop table if exists staging.customermgmt;
create table staging.customermgmt(
  --action element
  actiontype char(9) check(actiontype in ('NEW','ADDACCT','UPDCUST','UPDACCT
   ','CLOSEACCT','INACT')),
  actionts varchar check(length(actionts) > 0),
  --action.customer element
  c_id numeric(11) not null check(c_id >= 0),
  c_tax_id char(20) check((actiontype = 'NEW' and length(c_tax_id) > 0) or (
   actiontype != 'NEW')),
  c_gndr char(1) check(length(c_gndr) > 0),
  c_tier numeric(1) check(c_tier >= 0)
 ......
);
```

Listing 4.1: Part of Task 1 SQL

4.2.2 Task 2-3

Task 2 and Task 3 simply load data in txt and csv to staging schema, with the delimiter as "|". Part of Task 1 SQL is shown in listing 4.2.

```
......
COPY staging.batchdate FROM '/Users/wanglinhan/Desktop/BDMA/ULB/INFO-H419/
   Project/P2/generated_data/Batch1/BatchDate.txt';

COPY staging.cashtransaction FROM '/Users/wanglinhan/Desktop/BDMA/ULB/INFO-
   H419/Project/P2/generated_data/Batch1/CashTransaction.txt' delimiter '|';

COPY staging.dailymarket FROM '/Users/wanglinhan/Desktop/BDMA/ULB/INFO-H419/
   Project/P2/generated_data/Batch1/DailyMarket.txt' delimiter '|';
......
```

Listing 4.2: Part of Task 2 SQL

4.2.3 Task 4

Task 4 parse finwire and load to seperate tables. Because there are three record types in FINWIRE files: 'CMP', 'SEC', and 'FIN', we need to seperate them into three tables finwire_cmp, finwire_sec, finwire_fin with selected attributes.

As DuckDB doesn't support Postgresql, we need to define a Python function and register it to the connection. During the parsing process, the function need to connect to database several times and execute sql queries several times, instead of finishing every step in one function.

This feature may show the disadvantage of DuckDB to handle complicated queries and transformation.

Part of the customized function is shown in listing 4.3

```
def staging_finwire_split(conn):
    conn.execute("DELETE FROM staging.finwire_cmp")
    conn.execute("DELETE FROM staging.finwire_sec")
    conn.execute("DELETE FROM staging.finwire_fin")

    # Fetch data from staging.finwire
    result = conn.execute("SELECT * FROM staging.finwire").fetchall()
    for row in result:
        rectype = row[0][15:18]
        if rectype == 'CMP':
            # Logic for CMP
            exe_args = [row[0] for i in range(16)]
            conn.execute("""
                INSERT INTO staging.finwire_cmp
                SELECT
          nullif(trim(both from substring(?,1,15)), '') as pts,
          nullif(trim(both from substring(?,16,3)), '') as rectype,
                    ......
                LIMIT 1
            """, exe_args)
        elif rectype == 'SEC':
            # Logic for SEC
            exe_args = [row[0] for i in range(12)]
            conn.execute("""
                INSERT INTO staging.finwire_sec
```

```
                SELECT
          nullif(trim(both from substring(?,1,15)), '') as pts,
          nullif(trim(both from substring(?,16,3)), '') as rectype,
                    ......
                LIMIT 1
            """, exe_args)
        elif rectype == 'FIN':
            # Logic for FIN
            exe_args = [row[0] for i in range(17)]
            conn.execute("""
                INSERT INTO staging.finwire_fin
                SELECT
          nullif(trim(both from substring(?,1,15)), '') as pts,
          nullif(trim(both from substring(?,16,3)), '') as rectype,
                    ......
                LIMIT 1
            """, exe_args)
```

Listing 4.3: Part of Task 4 SQL

4.2.4 Task 5-6

Task 5 converts customer management source from xml to csv using a Python function without connecting to database. The output csv has 36 columns.

Part of Task 5 funtion is shown in listing 4.4

```
def customermgmt_convert(filepath):
    with open(filepath+'/CustomerMgmt.xml') as fd:
        doc = xmltodict.parse(fd.read())
        fd.close()

    with open(filepath+"/CustomerData.json", "w") as outfile:
        outfile.write(json.dumps(doc))
        outfile.close()

    f = open(filepath+'/CustomerData.json','r')

    cust = json.load(f)
    actions = cust['TPCDI:Actions']
```

```
    action = actions['TPCDI:Action']
    cust_df = pd.DataFrame(columns = np.arange(0, 36))


    for a in action:

        cust_row = {}

        # action element
        cust_row.update({0: [f"{a.get('@ActionType')}"]})
        cust_row.update({1: [f"{a.get('@ActionTS')}"]})
        ......

        # append to dataframe
        cust_df = pd.concat([cust_df, pd.DataFrame.from_dict(cust_row)],
axis = 0)

    cust_df.replace(to_replace = np.NaN, value = "", inplace = True)
    cust_df.replace(to_replace = "None", value = "", inplace = True)
    cust_df.to_csv(filepath+'/CustomerMgmt.csv', index = False)
    print('Customer Management data converted from XML to CSV')
```

Listing 4.4: Part of Task 5 SQL

Task 6 directly loads the output csv into staging schema.

4.2.5 Task 7

Task 7 creates the master schema, which is the data warehouse we need. The master schema includes 20 tables: tradetype, statustype, taxrate, industry, dimdate, dimtime, dimcompany, dimbroker, dimcustomer, dimaccount, dimsecurity, dimtrade, financial, factcashbalances, factholdings, factmarkethistory, factwatches, prospect, audit and dimessages.

4.2.6 Task 8-11

Task 8-11 direct load master.tradetype, master.statustype, master.taxrate, master.industry.

In these 4 tables, the data is the same in staging schema and master schema.

Part of Task 8 SQL is shown in listing 4.5

```
delete from master.tradetype;
insert into master.tradetype
  select * from staging.tradetype;
```

Listing 4.5: Part of Task 8 SQL

4.2.7 Task 12-15

Task 12 loads date related attributes from staging.date to master.dimdate.

Task 13 loads time related attributes from staging.time to master.dimtime.

Task 14 loads company related attributes from staging.finwire_cmp, staging.statustype and staging.industry to master.dimcompany. At the same time, it classifies whether some company is low-grade.

Task 14 SQL is shown in listing 4.6.

```
-- dimcompany
delete from master.dimcompany;
insert into master.dimcompany
  select
  row_number() over(order by cik) as sk,
  cik::numeric(11) as companyid,
  s.st_name as status,
  companyname as name,
  i.in_name as industry,
  (CASE
    WHEN sprating not in ('AAA','AA','AA+','AA-','A','A+','A-','BBB','BBB+',
   'BBB-','BB','BB+','BB-','B','B+','B-','CCC','CCC+','CCC-','CC','C','D')
      THEN null
    ELSE f.sprating END) as sprating,
  (CASE
    WHEN sprating not in ('AAA','AA','AA+','AA-','A','A+','A-','BBB','BBB+',
   'BBB-','BB','BB+','BB-','B','B+','B-','CCC','CCC+','CCC-','CC','C','D')
      THEN null
```

```
    WHEN f.sprating like 'A%' or f.sprating like 'BBB%'
      THEN false
    ELSE
      true
  END) as islowgrade,
  ceoname as ceo,
  addressline1,
  addressline2,
  postalcode,
  city,
  stateprovince,
  country,
  description,
  CAST(SUBSTRING(foundingdate, 1, 4) || '-' || SUBSTRING(foundingdate, 5, 2)
    || '-' || SUBSTRING(foundingdate, 7, 2) AS DATE),
  case when lead( (select batchdate from staging.batchdate) ) over (
   partition by cik order by pts asc ) is null then true else false end as
   iscurrent,
  1 as batchid,
  CAST(SUBSTRING(left(f.pts, 8), 1, 4) || '-' || SUBSTRING(left(f.pts, 8),
   5, 2) || '-' || SUBSTRING(left(f.pts, 8), 7, 2) AS DATE) as effectivedate
   ,
  '9999-12-31'::date as enddate
  from
    staging.finwire_cmp f,
    staging.statustype s,
    staging.industry i
  where
    f.status = s.st_id
  and f.industryid = i.in_id;
```

Listing 4.6: Task 14 SQL

Task 15 inserts master.dimessages from dimcompany results. A record will be inserted in the DImessages table if a company's SPRating is not one of the valid values for Standard and Poor long-term credit-ratings. The valid values are: AAA, AA[+/-], A[+/-], BBB[+/-], BB[+/-], B[+/-], CCC[+/-], CC, C, D.

Task 15 SQL is shown in listing 4.7.

```
-- dimessages alert for dimcompany
```

```
delete from master.dimessages;
insert into master.dimessages
  select
  now(),
  1 as batchid,
  'DimCompany' as messagesource,
  'Invalid SPRating' as messagetext,
  'Alert' as messagetype,
  'CO_ID = ' || cik::varchar || ', CO_SP_RATE = ' || sprating::varchar
  from staging.finwire_cmp
  where sprating not in ('AAA','AA','AA+','AA-','A','A+','A-','BBB','BBB+','
   BBB-','BB','BB+','BB-','B','B+','B-','CCC','CCC+','CCC-','CC','C','D');
```

Listing 4.7: Task 15 SQL

4.2.8 Task 16

Task 16 transforms and loads master.dimbroker.

Data for DimBroker comes from the HR extract file HR.csv. Those employees from the HR file that are brokers (as indicated by the EmployeeJobCode) will have data copied to the Broker table.

Task 16 SQL is shown in listing 4.8.

```
-- dimbroker
delete from master.dimbroker;
insert into master.dimbroker
  select
  row_number() over(order by employeeid) as sk,
  employeeid as brokerid,
  managerid,
  employeefirstname,
  employeelastname,
  employeemi,
  employeebranch,
  employeeoffice,
  employeephone,
  true as iscurrent,
  1 as batchid,
  (select min(datevalue) FROM master.dimdate) as effectivedate,
```

```
  '9999-12-31'::date as enddate
  from staging.hr
  where employeejobcode = 314;
```

Listing 4.8: Task 16 SQL

4.2.9 Task 17

Task 17 transform and load master.prospect.

Prospect table data is obtained from the Prospect file.

In this process, MarketingNameplate is set based on other fields. The tags are defined as 'High-Value', 'Expenses', 'Boomer', 'MoneyAlert', 'Spender' and 'Inherited' according to the documentation[2].

Because DuckDB doesn't support btrim function in Postgresql, so we need to define custom *btrim_equivalent* function.

```
def btrim_equivalent(input_string, trim_characters):
    start_index = len(input_string) - len(input_string.lstrip(
    trim_characters))
    end_index = len(input_string.rstrip(trim_characters)) - 1
    return input_string[start_index:end_index + 1]

con = duckdb.connect('tpcdi_'+str(scale_factor)+'.db')
con.create_function("btrim_equivalent", btrim_equivalent, ['VARCHAR', '
    VARCHAR'], 'VARCHAR')
```

Listing 4.9: btrim_equivalent

Part of Task 17 SQL is shown in listing 4.10.

```
-- prospect part 1
delete from master.prospect;
insert into master.prospect
  with date_record_id as (
    select
    dd.sk_dateid
```

[2]https://www.tpc.org/TPC_Documents_Current_Versions/pdf/TPC-DI_v1.1.0.pdf

```
    from master.dimdate dd
    inner join staging.batchdate bd
      on dd.datevalue = bd.batchdate
)

select
  p.agencyid
, dri.sk_dateid
, dri.sk_dateid
      ......
, nullif(btrim_equivalent(btrim_equivalent(btrim_equivalent(
 btrim_equivalent(btrim_equivalent(
  case
  when p.networth > 1000000 or p.income > 200000
  then 'HighValue'
  else ''
  end
  || '+' ||
  case
  when p.numberchildren > 3 or p.numbercreditcards > 5
  then 'Expenses'
  else ''
  end
  , '+')
  || '+' ||
  case
  when p.age > 45
  then 'Boomer'
  else ''
  end
  , '+')
  || '+' ||
  case
  when p.income < 50000 or p.creditrating < 600 or p.networth < 100000
  then 'MoneyAlert'
  else ''
  end
  , '+')
  || '+' ||
  case
```

```
    when p.numbercars > 3 or p.numbercreditcards > 7
    then 'Spender'
    else ''
    end
    , '+')
    || '+' ||
    case
    when p.age < 25 and p.networth > 1000000
    then 'Inherited'
    else ''
    end
    , '+'), '')
  from staging.prospect p
  cross join date_record_id dri;
```

Listing 4.10: Task 17 SQL

4.2.10 Task 18-20

Task 18 transform and load master.dimcustomer.

DimCustomer data is obtained from the data file CustomerMgmt.xml.

Customer has six ActionType.

- **NEW**. A new customer. A new customer is always created with 1 or more new accounts.
- **ADDACCT**. One or more new accounts for an existing customer. This does not require any change to DimCustomer.
- **UPDACCT**. Updates to the information in one or more existing accounts. No change to DimCustomer is required.
- **UPDCUST**. One or more updates to existing customer's information. Only the identifying data and updated property values are supplied in the Source Data.
- **CLOSEACCT**. Close one or more existing accounts. This does not require any change to DimCustomer.

- **INACT**. Make an existing customer and that customer's currently active accounts inactive. No specific account information is supplied in the Source Data.

Changes are made based on different ActionTypes.

Task 19 load master.dimessages with alert from master.dimcustomer.

A record will be inserted in the DImessages table if a customer's Tier is not one of the valid values (1,2,3). The MessageSource is "DimCustomer", the MessageType is "Alert" and the MessageText is "Invalid customer tier". The MessageData field is "C_ID = " followed by the natural key value of the record, then ", C_TIER = " and the C_TIER value.

A record will be reported in the DImessages table if a customer's DOB is invalid. A customer's DOB is invalid if DOB < Batch Date – 100 years or DOB > Batch Date (customer is over 100 years old or born in the future). The MessageSource is "DimCustomer", the MessageType is "Alert" and the MessageText is "DOB out of range". The MessageData field is "C_ID = " followed by the natural key value of the record, then ", C_DOB = " and the C_DOB value.

Task 19 SQL is shown in listing 4.11.

```
-- dimessages alert for dimcustomer
insert into master.dimessages
  select
    now()
  , 1
  , 'DimCustomer'
  , 'Invalid customer tier'
  , 'Alert'
  , 'C_ID = ' || customerid || ', C_TIER = ' || tier
  from master.dimcustomer
  where tier not between 1 and 3;

insert into master.dimessages
  select
    now()
  , 1
  , 'DimCustomer'
  , 'DOB out of range'
  , 'Alert'
```

```
  , 'C_ID = ' || customerid || ', C_DOB = ' || dob
  from master.dimcustomer
  where dob < (select * from staging.batchdate) - interval '100 years'
  or dob > (select * from staging.batchdate);
```

Listing 4.11: Task 19 SQL

Task 20 update master.prospect.

If a customer is 'ACTIVE' in master.dimcustomer, the attribute 'iscustomer' should be true.

Task 20 SQL is shown in listing 4.12.

```
-- update prospect
with current_active_customer as (
  select p.*
  from master.prospect p
  inner join master.dimcustomer c
  on upper(c.lastname) = upper(p.lastname)
  and upper(c.firstname) = upper(p.firstname)
  and upper(c.addressline1) = upper(p.addressline1)
  and upper(c.addressline2) = upper(p.addressline2)
  and upper(c.postalcode) = upper(p.postalcode)
  where c.status = 'ACTIVE'
  and c.iscurrent = true
)

update master.prospect
  set iscustomer = true
  where lastname in (select lastname from current_active_customer)
  and firstname in (select firstname from current_active_customer)
  and addressline1 in (select addressline1 from current_active_customer)
  and addressline2 in (select addressline2 from current_active_customer)
  and postalcode in (select postalcode from current_active_customer);
```

Listing 4.12: Task 20 SQL

4.2.11 Task 21

Task 21 transform and load master.dimaccount.

DimAccount data is obtained from the data file CustomerMgmt.xml.

Similar to Task 18. Task 21 also do different processing based on different ActionTypes: 'NEW', 'ADDACCT', 'UPDACCT', 'UPDCUST', 'CLOSEACCT' and 'INACT'.

There are no messages written to the DImessages table by this transformation.

4.2.12 Task 22

Task 22 transform and load master.dimsecurity.

DimSecurity data is obtained from the FINWIRE files. All FINWIREyyyyQq files are processed in ascending year and quarter order, and records of type SEC are used.

```
-- master.dimsecurity transform and load
truncate table master.dimsecurity;
insert into master.dimsecurity
  select
    row_number() over() as sk_securityid,
    symbol,
    issuetype as issue,
    s.st_name as status,
    f.name,
    exid as exchangeid,
    c.sk_companyid as sk_companyid,
    shout::numeric(12) as sharesoutstanding,
    CAST(SUBSTRING(left(firsttradedate, 8), 1, 4) || '-' || SUBSTRING(left(
   firsttradedate, 8), 5, 2) || '-' || SUBSTRING(left(firsttradedate, 8), 7,
    2) AS DATE),
    CAST(SUBSTRING(left(firsttradeexchg, 8), 1, 4) || '-' || SUBSTRING(left(
   firsttradeexchg, 8), 5, 2) || '-' || SUBSTRING(left(firsttradeexchg, 8),
   7, 2) AS DATE),
    dividend::numeric(10,2),
    case
      when lead( (select batchdate from staging.batchdate) ) over (
   partition by symbol order by pts asc ) is null
      then true
      else false
      end as iscurrent,
```

```
    1 as batchid,
    CAST(SUBSTRING(left(f.pts, 8), 1, 4) || '-' || SUBSTRING(left(f.pts, 8),
    5, 2) || '-' || SUBSTRING(left(f.pts, 8), 7, 2) AS DATE),
    '9999-12-31'::date as enddate
  from staging.finwire_sec f,
    staging.statustype s,
    master.dimcompany c
  where f.status = s.st_id
  and ((ltrim(f.conameorcik, '0') = c.companyid::varchar)
    or (f.conameorcik = c.name))
  and CAST(SUBSTRING(left(f.pts, 8), 1, 4) || '-' || SUBSTRING(left(f.pts,
   8), 5, 2) || '-' || SUBSTRING(left(f.pts, 8), 7, 2) AS DATE) >= c.
   effectivedate
  and CAST(SUBSTRING(left(f.pts, 8), 1, 4) || '-' || SUBSTRING(left(f.pts,
   8), 5, 2) || '-' || SUBSTRING(left(f.pts, 8), 7, 2) AS DATE) < c.enddate;
```

Listing 4.13: Task 20 SQL

4.2.13 Task 23-24

Task 23 transform and load master.dimtrade.

DimTrade data is obtained from the Trade.txt and TradeHistory.txt files. The incoming files may be thought of as logically joined on the `T_ID` field. When a `T_ID` encountered does not match a TradeID from the DimTrade table, a new DimTrade record is inserted. When a `T_ID` is encountered that matches an existing TradeID in the DimTrade table, the DimTrade record is updated.

Because DuckDB is using tmp files to store temporary tables, too much tmp files occupies more space than space left on device, Task 23 is divided into two steps. First, join staging.trade and staging.tradehistory. Second, join and select values.

Task 23 SQL is shown in listing 4.14.

```
--step 1
insert into staging.trade_joined
select * from staging.trade t inner join staging.tradehistory th on t.t_id =
    th.th_t_id;

--step 2
```

```
delete from master.dimtrade;
insert into master.dimtrade (sk_createdateid, sk_createtimeid,
    sk_closedateid, sk_closetimeid, tradeid, cashflag,
        quantity, bidprice, executedby, tradeprice, fee, commission, tax,
    status, type, sk_securityid, sk_companyid,
        sk_accountid, sk_customerid, sk_brokerid, batchid)
SELECT
Case When (sth.th_st_id = 'SBMT' and (sth.t_tt_id = 'TMB' or sth.t_tt_id ='
    TMS')) or (sth.th_st_id = 'PNDG')
        then D.sk_dateid else NULL
end,
Case When (sth.th_st_id = 'SBMT' and (sth.t_tt_id = 'TMB' or sth.t_tt_id ='
    TMS')) or (sth.th_st_id = 'PNDG')
        then T.sk_timeid else NULL
end,
case when (sth.th_st_id = 'CMPT' or sth.th_st_id = 'CNCL') then D.sk_dateid
    else NULL
end,
case when (sth.th_st_id = 'CMPT' or sth.th_st_id = 'CNCL') then T.sk_timeid
    else NULL
end,
sth.t_id, sth.t_is_cash, sth.t_qty, sth.t_bid_price, sth.t_exec_name, sth.
    t_trade_price, sth.t_chrg,
sth.t_comm, sth.t_tax, st.st_name, tt.tt_name,
case when (CAST(sth.th_dts as DATE) >= ds.EffectiveDate) and (CAST(sth.
    th_dts as DATE) <= ds.EndDate)
    then ds.SK_SecurityID END,
case when (CAST(sth.th_dts as DATE) >= ds.EffectiveDate) and (CAST(sth.
    th_dts as DATE) <= ds.EndDate)
    then ds.SK_CompanyID END,
case when (CAST(sth.th_dts as DATE) >= ds.EffectiveDate) and (CAST(sth.
    th_dts as DATE) <= ds.EndDate)
    then da.SK_AccountID END,
case when (CAST(sth.th_dts as DATE) >= ds.EffectiveDate) and (CAST(sth.
    th_dts as DATE) <= ds.EndDate)
    then da.SK_CustomerID END,
case when (CAST(sth.th_dts as DATE) >= ds.EffectiveDate) and (CAST(sth.
    th_dts as DATE) <= ds.EndDate)
    then da.SK_BrokerID END,
1
```

```
from staging.trade_joined sth
join staging.statustype st on sth.t_st_id = st.st_id
join staging.tradetype tt on sth.t_tt_id = tt.tt_id
join master.dimsecurity ds on sth.t_s_symb = ds.symbol
join master.dimaccount da on sth.t_ca_id = da.accountid
JOIN master.dimdate D on CAST(sth.th_dts as DATE) = D.datevalue
join master.dimtime T on CAST(sth.th_dts AS TIME) = T.timevalue
```

Listing 4.14: Task 23 SQL

Task24 load master.dimessages with alert from master.dimtrade.

A record will be reported in the DImessages table if a trade's Commission is not null and exceeds TradePrice * Quantity. The MessageSource is "DimTrade", the MessageType is "Alert" and the MessageText is "Invalid trade commission". The MessageData field is `T_ID` =followed by the key value of the record, then `T_COMM` = and the `T_COMM` value.

Task 24 SQL is shown in listing 4.15.

```
-- dimessages alert for dimtrade
insert into master.dimessages
  select
    now()
  , 1
  , 'DimTrade'
  , 'Invalid trade commission'
  , 'Alert'
  , 'T_ID = ' || tradeid || ', T_COMM = ' || commission
  from master.dimtrade
  where commission is not null
  and commission > (tradeprice * quantity);

insert into master.dimessages
  select
    now()
  , 1
  , 'DimTrade'
  , 'Invalid trade fee'
  , 'Alert'
  , 'T_ID = ' || tradeid || ', T_CHRG = ' || fee
  from master.dimtrade
```

```
  where fee is not null
  and fee > (tradeprice * quantity);
```

Listing 4.15: Task 24 SQL

4.2.14 Task 25

Task25 transform and load master.financial.

Financial data is obtained from the FINWIRE files. All FINWIREyyyyQq files are processed in ascending year and quarter order, and records of type FIN are used. The surrogate key of the associated company must be obtained for the Company dimension reference.

Task 25 SQL is shown in listing 4.16.

```
-- financial
delete from master.financial;
insert into master.financial
  select
    c.sk_companyid as sk_companyid,
    year::numeric as fi_year,
    quarter::numeric as fi_qtr,
    CAST(SUBSTRING(qtrstartdate, 1, 4) || '-' || SUBSTRING(qtrstartdate, 5,
   2) || '-' || SUBSTRING(qtrstartdate, 7, 2) AS DATE) as fi_qtr_start_date,
    revenue::numeric as fi_revenue,
    earnings::numeric as fi_net_earn,
    eps::numeric as fi_basic_eps,
    dilutedeps::numeric as fi_dilut_eps,
    margin::numeric as fi_margin,
    inventory::numeric as fi_inventory,
    assets::numeric as fi_assets,
    liability::numeric as fi_liability,
    shout::numeric as fi_out_basic,
    dilutedshout::numeric as fi_out_dilut
  from staging.finwire_fin f,
    master.dimcompany c
  where ((f.conameorcik = c.companyid::varchar)
    or (f.conameorcik = c.name))
  and CAST(SUBSTRING(left(pts, 8), 1, 4) || '-' || SUBSTRING(left(pts, 8),
```

```
    5, 2) || '-' || SUBSTRING(left(pts, 8), 7, 2) AS DATE) >= c.effectivedate
24  and CAST(SUBSTRING(left(pts, 8), 1, 4) || '-' || SUBSTRING(left(pts, 8),
    5, 2) || '-' || SUBSTRING(left(pts, 8), 7, 2) AS DATE) < c.enddate;
```

Listing 4.16: Task 25 SQL

4.2.15 Task 26

Task26 transform and load master.factcashbalances.

FactCashBalances data is obtained from the data file CashTransaction.txt. The net effect of all cash transactions for a given account on a given day is totaled, and only a single record is generated per account that had changes per day.

Task 26 SQL is shown in listing 4.17.

```
1  -- factcashbalances
2  delete from master.factcashbalances;
3  insert into master.factcashbalances
4    with agg as (
5      select
6      a.sk_customerid as sk_customerid,
7      a.sk_accountid as sk_accountid,
8      d.sk_dateid as sk_dateid,
9      sum(ct_amt) as ct_amt_day
10     from staging.cashtransaction c,
11       master.dimaccount a,
12       master.dimdate d
13     where c.ct_ca_id = a.accountid
14     and ct_dts::date >= a.effectivedate
15     and ct_dts::date < a.enddate
16     and ct_dts::date = d.datevalue
17     group by
18       a.sk_customerid,
19       a.sk_accountid,
20       d.sk_dateid
21   )
22   , final_output as (
23     select
```

```
    sk_customerid,
    sk_accountid,
    sk_dateid,
    sum(ct_amt_day) over(partition by sk_accountid order by sk_dateid rows
   between unbounded preceding and current row) as cash,
    1 as batchid
    from agg
  )
  select * from final_output;
```

Listing 4.17: Task 26 SQL

4.2.16 Task 27

Task27 transform and load master.factholdings.

Data for FactHoldings comes from the HoldingHistory.txt file and the DimTrade table. The quantity and price values reflect the holdings for a particular security after the most recent trade. The customer can have a positive or negative position (Quantity) as a result of a trade.

Task 27 SQL is shown in listing 4.18.

```
-- factholdings
delete from master.factholdings;
insert into master.factholdings
  select
  h.hh_h_t_id as tradeid,
  t.tradeid as currenttradeid,
  t.sk_customerid as sk_customerid,
  t.sk_accountid as sk_accountid,
  t.sk_securityid as sk_securityid,
  t.sk_companyid as sk_companyid,
  t.sk_closedateid as sk_dateid,
  t.sk_closetimeid as sk_timeid,
  t.tradeprice as currentprice,
  h.hh_after_qty as currentholding,
  1 as batchid
  from staging.holdinghistory h,
    master.dimtrade t
```

```
18   where h.hh_t_id = t.tradeid;
```

Listing 4.18: Task 27 SQL

4.2.17 Task 28

Task28 transform and load master.factwatches.

Data for FactWatches comes from the WatchHistory.txt file. Surrogate keys must be obtained for the Customer, Security and Date dimension references. The date keys show the dates the watch was set and removed.

Task 28 SQL is shown in listing 4.19.

```
1  -- factwatches
2  delete from master.factwatches;
3  insert into master.factwatches
4    with watches as (
5      select w1.w_c_id,
6      TRIM(w1.w_s_symb) as w_s_symb,
7      w1.w_dts::date as dateplaced,
8      w2.w_dts::date as dateremoved
9      from staging.watchhistory w1,
10       staging.watchhistory w2
11     where w1.w_c_id = w2.w_c_id
12     and w1.w_s_symb = w2.w_s_symb
13     and w1.w_action = 'ACTV'
14     and w2.w_action = 'CNCL'
15   )
16
17   select
18     c.sk_customerid as sk_customerid,
19     s.sk_securityid as sk_securityid,
20     CAST(EXTRACT(YEAR FROM w.dateplaced) * 10000 + EXTRACT(MONTH FROM w.
     dateplaced) * 100 + EXTRACT(DAY FROM w.dateplaced) AS NUMERIC) as
     sk_dateid_dateplaced,
21     CAST(EXTRACT(YEAR FROM w.dateremoved) * 10000 + EXTRACT(MONTH FROM w.
     dateremoved) * 100 + EXTRACT(DAY FROM w.dateremoved) AS NUMERIC) as
     sk_dateid_dateremoved,
```

```
    1 as batchid
  from watches w,
    master.dimcustomer c,
    master.dimsecurity s,
    master.dimdate d1,
    master.dimdate d2
  where w.w_c_id = c.customerid
  and w.w_s_symb = s.symbol
  and w.dateplaced = d1.datevalue
  and w.dateremoved = d2.datevalue;
```

Listing 4.19: Task 28 SQL

4.2.18 Task 29-30

Task29 transform and load master.factmarkethistory.

FactMarketHistory data is primarily obtained from the file DailyMarket.txt.

Part of Task 29 SQL is shown in listing 4.20.

```
-- factmarkethistory
delete from master.factmarkethistory;
insert into master.factmarkethistory
  with market_dates_daily as (
    select
      dm.dm_s_symb
    , dm.dm_date
            ......
    from staging.dailymarket dm
    inner join master.dimdate dd
      on dm.dm_date = dd.datevalue
    order by
        dm.dm_s_symb
      , dm.dm_date desc
  )

  , high_low as (
    select
      dm_s_symb
```

```
20            ......
21    , max(dm_high) over(partition by dm_s_symb order by dm_date rows between
      363 preceding and current row) as fiftytwoweekhigh
22    , min(dm_low) over(partition by dm_s_symb order by dm_date rows between
     363 preceding and current row) as fiftytwoweeklow
23    from market_dates_daily
24  )
25
26  , high_date as (
27    select
28      hl.dm_s_symb
29            ......
30    , max(mdd.dm_date) as sk_fiftytwoweekhighdate
31    from high_low hl
32    inner join market_dates_daily mdd
33      on hl.dm_s_symb = mdd.dm_s_symb
34      and hl.fiftytwoweekhigh = mdd.dm_high
35      and mdd.dm_date <= hl.dm_date
36      and mdd.dm_date >= hl.dm_date - interval '52 weeks'
37    group by
38        hl.dm_s_symb
39      , hl.dm_date
40                ......
41  )
42
43  , low_date as (
44    select
45      hl.dm_s_symb
46    , hl.dm_date
47            ......
48    from high_date hl
49    inner join market_dates_daily mdd
50      on hl.dm_s_symb = mdd.dm_s_symb
51      and hl.fiftytwoweeklow = mdd.dm_low
52      and mdd.dm_date <= hl.dm_date
53      and mdd.dm_date >= hl.dm_date - interval '52 weeks'
54    group by
55        hl.dm_s_symb
56      , hl.dm_date
57      , hl.dm_close
```

```
                .....
)

, quarters as (
    select
      f.sk_companyid
                .....
)

, final_output as (
  select
    s.sk_securityid
  , s.sk_companyid
  , CAST(EXTRACT(YEAR FROM ld.dm_date) * 10000 + EXTRACT(MONTH FROM ld.
 dm_date) * 100 + EXTRACT(DAY FROM ld.dm_date) AS NUMERIC) as sk_dateid
  , case
    when q.eps_qtr_sum != 0 and q.eps_qtr_sum is not null
    then (ld.dm_close / q.eps_qtr_sum)::numeric(10, 2)
    else null
    end as peratio
  , case
    when ld.dm_close != 0
    then round((s.dividend / ld.dm_close) * 100, 2)
    else null
    end as yield
  , ld.fiftytwoweekhigh
  , CAST(EXTRACT(YEAR FROM ld.sk_fiftytwoweekhighdate) * 10000 + EXTRACT(
 MONTH FROM ld.sk_fiftytwoweekhighdate) * 100 + EXTRACT(DAY FROM ld.
 sk_fiftytwoweekhighdate) AS NUMERIC) as sk_fiftytwoweekhighdate
  , ld.fiftytwoweeklow
  , CAST(EXTRACT(YEAR FROM ld.sk_fiftytwoweeklowdate) * 10000 + EXTRACT(
 MONTH FROM ld.sk_fiftytwoweeklowdate) * 100 + EXTRACT(DAY FROM ld.
 sk_fiftytwoweeklowdate) AS NUMERIC) as sk_fiftytwoweeklowdate
  , ld.dm_close as closeprice
  , ld.dm_high as dayhigh
  , ld.dm_low as daylow
  , ld.dm_vol as volume
  , 1 as batchid
  from low_date ld
    inner join master.dimsecurity s
```

```
        on ld.dm_s_symb = s.symbol
        and ld.dm_date >= s.effectivedate
        and ld.dm_date < s.enddate
      inner join quarters q
        on s.sk_companyid = q.sk_companyid
        and q.fi_qtr_start_date <= ld.dm_date
        and q.next_qtr_start > ld.dm_date
  )
  select * from final_output;
```

Listing 4.20: Part of Task 29 SQL

Task30 load master.dimessages with alert from master.factmarkethistory. A record will be reported in the DImessages table if there are no earnings found for a company. The MessageSource is “FactMarketHistory”, the MessageType is “Alert” and the MessageText is “No earnings for company”. The MessageData field is `DM_S_SYMB =` followed by the `DM_S_SYMB` value of the record.

Task 30 SQL is shown in listing 4.21

```
-- dimessages alert for factmarkethistory
insert into master.dimessages
  select
    now()
  , 1
  , 'FactMarketHistory'
  , 'No earnings for company'
  , 'Alert'
  , 'DM_S_SYMB = ' || s.symbol
  from master.factmarkethistory fmh
  inner join master.dimsecurity s
    on fmh.sk_securityid = s.sk_securityid
  where fmh.peratio is null
  or fmh.peratio = 0;
```

Listing 4.21: Task 30 SQL

5 Results and Discussion

As mentioned in the previous chapter, the benchmark was performed on a local machine, with a total of four different scale factors (3, 5 ,7 and 9) and our scale factors increased linearly. In this chapter, we will be focusing on discussing the test result, the evolution of the loading times as the scale factor of the database increases.

5.1 Total Performance

We ran the scripts for 4 scale factors and show the results in Figure 5.1 and Figure 5.2 . Execution times were measured in seconds. The charts provided the execution time for 4 scale factors. The bar chart presents the information about the Load Test Time in seconds for four scale factors. As can be seen from the chart there is a significant difference between Execution Time for each scale factor. The time increases from 268.638778 seconds for SF 3 factor up to 2365.027075 seconds for SF 9. The line chart also shows that the execution time increased almost linearly as the data was increased from sf3 to sf7. But the growth trend suddenly increases from sf7 to sf9.

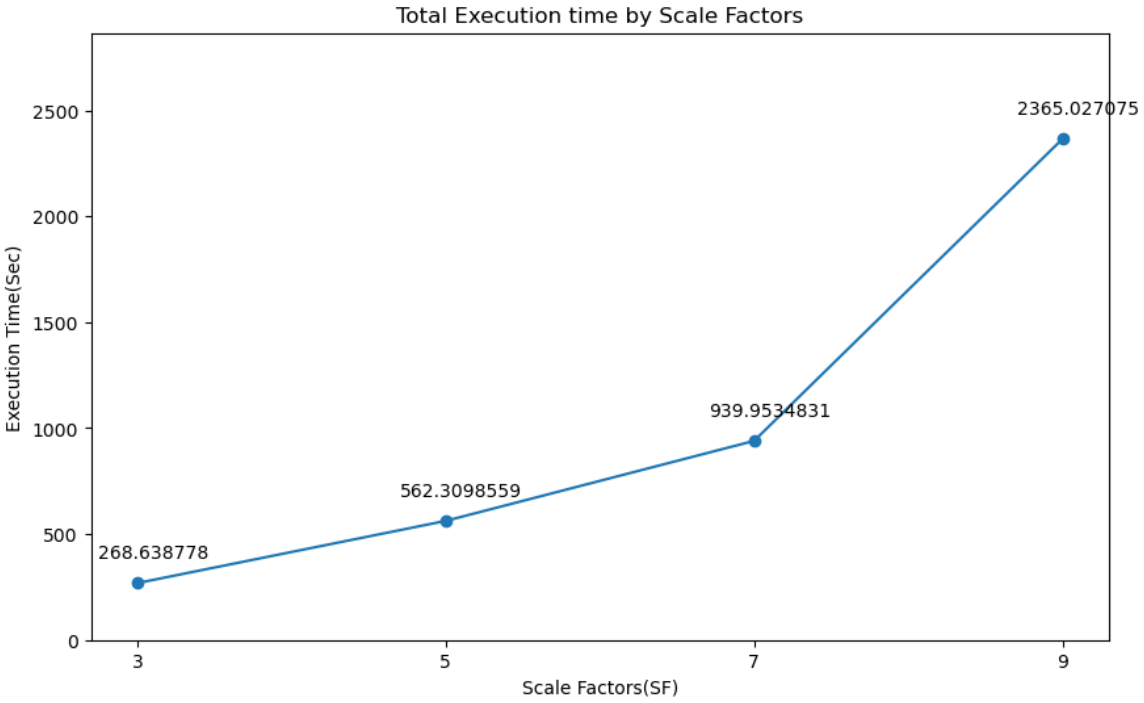


Figure 5.1: Line Chart of Total execution time

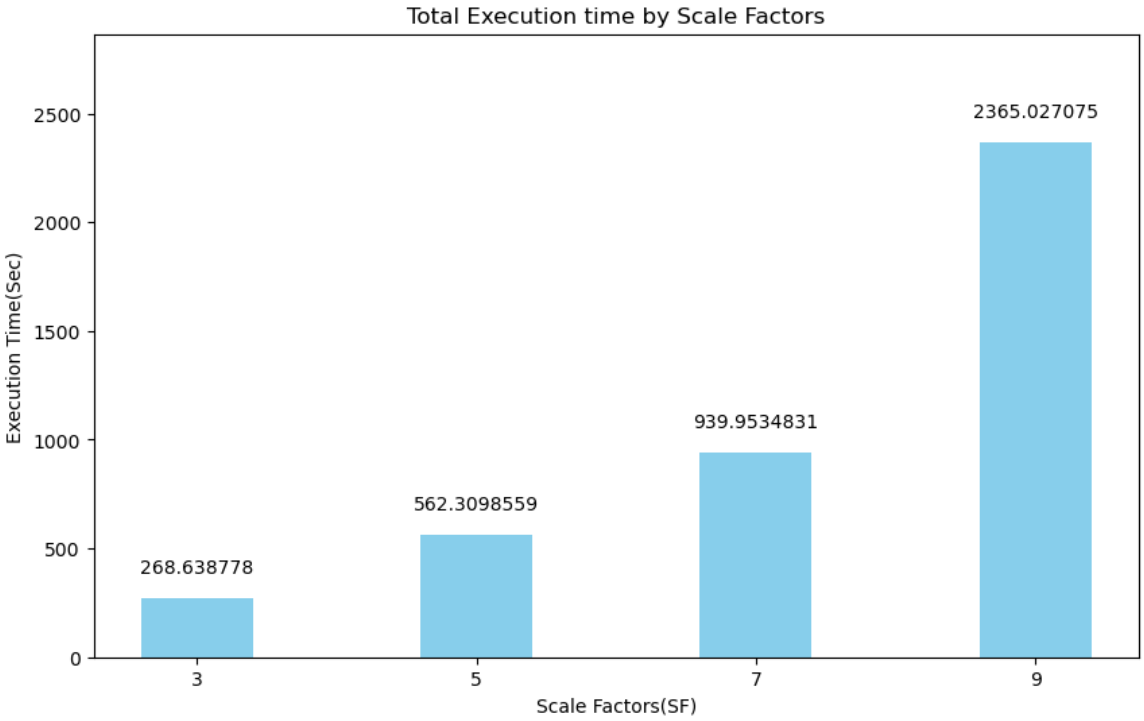


Figure 5.2: Bar Chart of Total execution time

5.2 Comparison of execution times for all tasks

From the Figure 5.3, we can observe the linear tendency of execution times for every tasks.

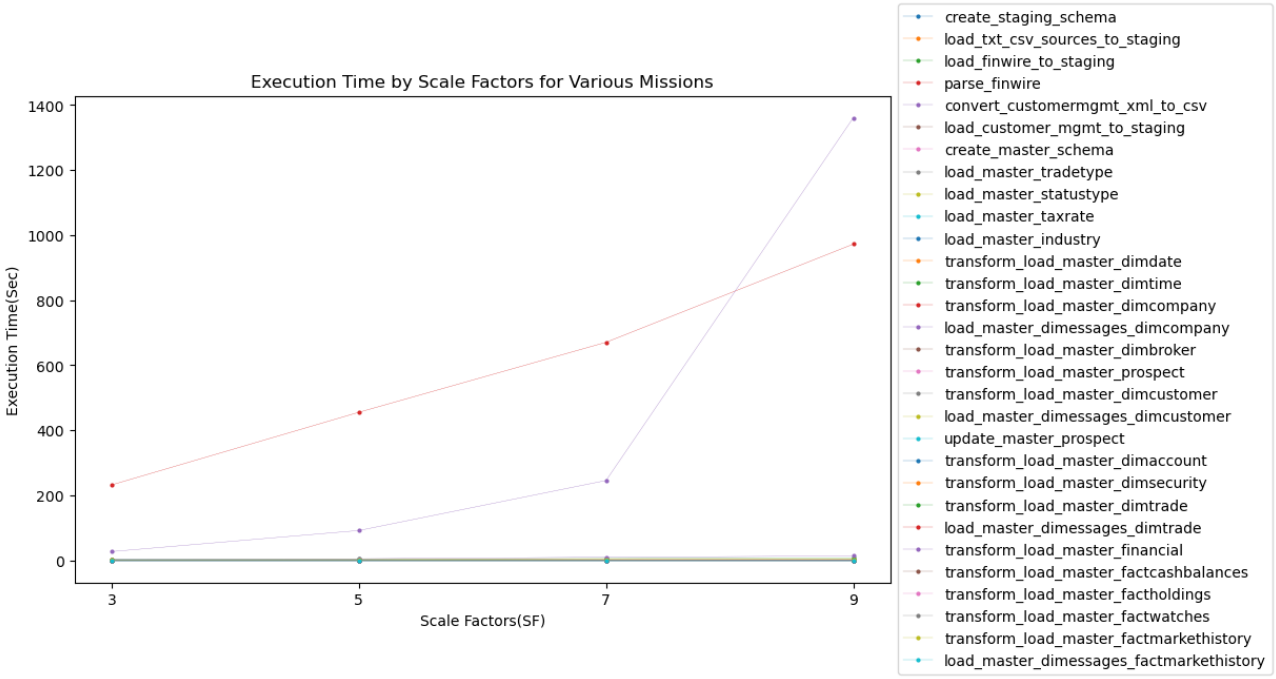


Figure 5.3: execution times for all tasks

From the figure above, we observed a significant disparity in execution times between various tasks.To show this big disparity, We also made the pie charts of Proportion of Execution Time of Various Tasks from SF3 to SF9. Then we found the execution time of `parse_finwire` and `convert_customermgmt_xml_to_csv` took up the two largest percentage of time.

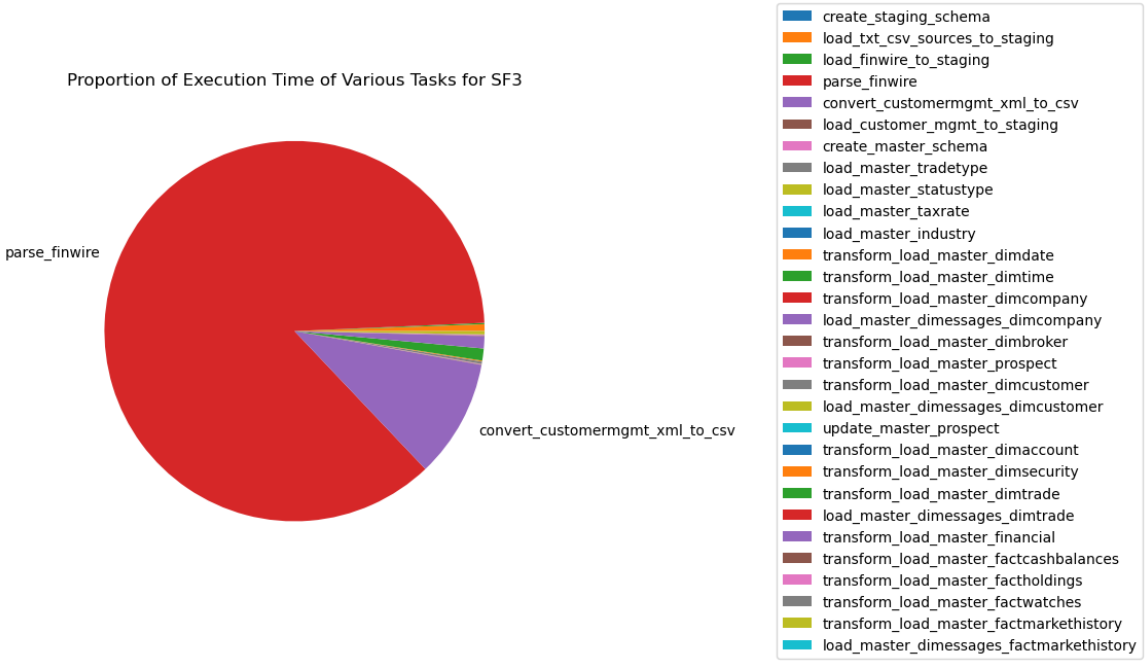


Figure 5.4: Proportion of Execution Time of Various Tasks for SF3

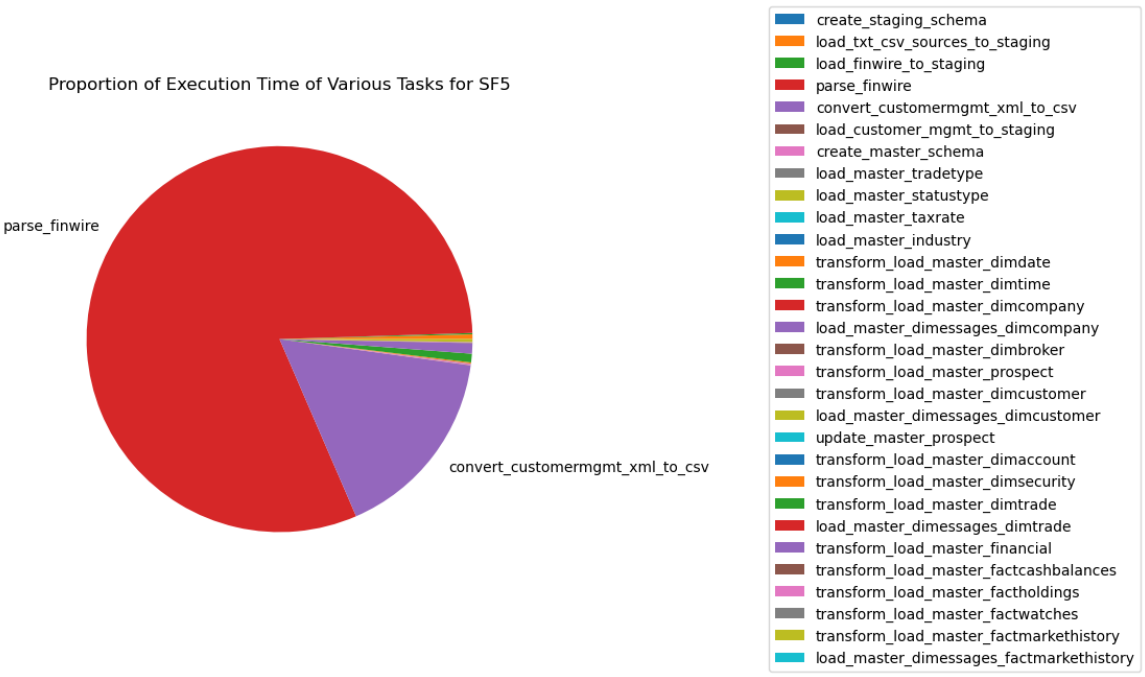


Figure 5.5: Proportion of Execution Time of Various Tasks for SF5

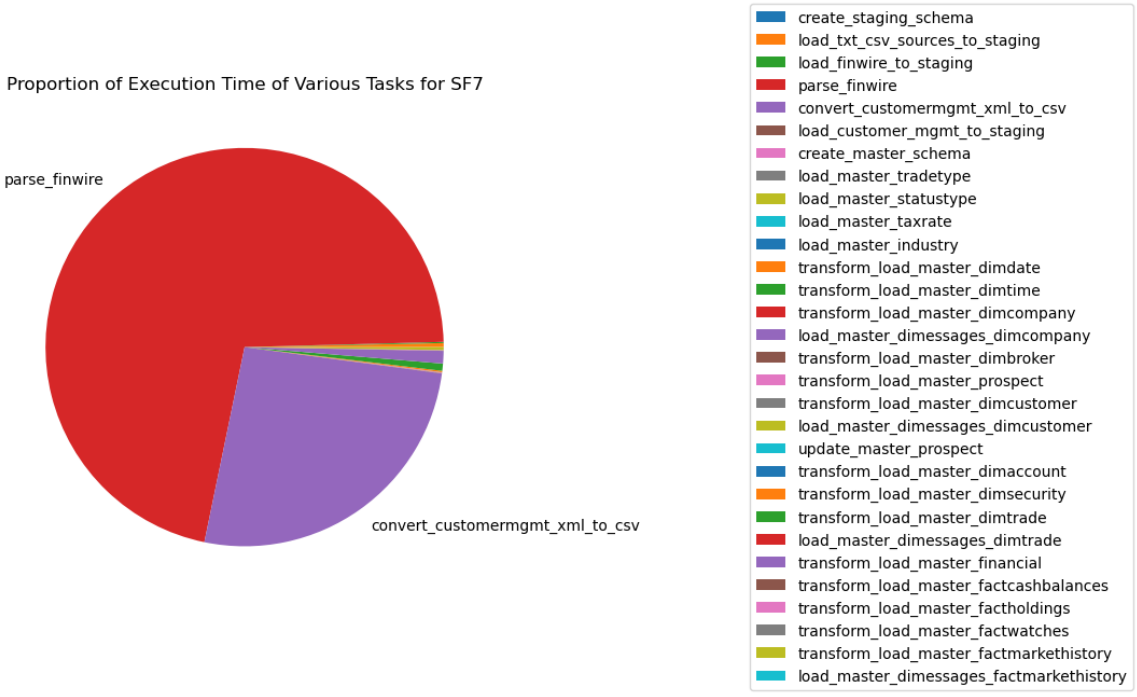


Figure 5.6: Proportion of Execution Time of Various Tasks for SF7

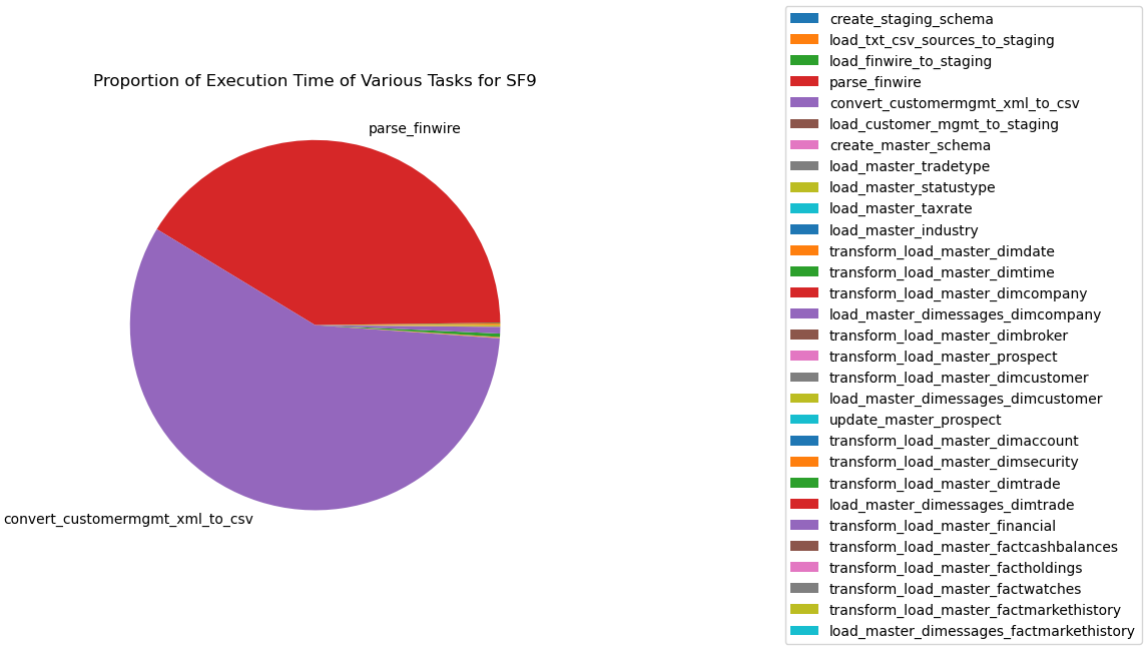


Figure 5.7: Proportion of Execution Time of Various Tasks for SF9

To clearly illustrate the trends of the remaining tasks, we made a strategic decision to exclude the two most time-consuming tasks from our data set. This approach allowed us to focus on the performance of the other tasks more effectively. Figure 5.8 presents the adjusted execution times, offering a clearer view of the underlying trends and patterns

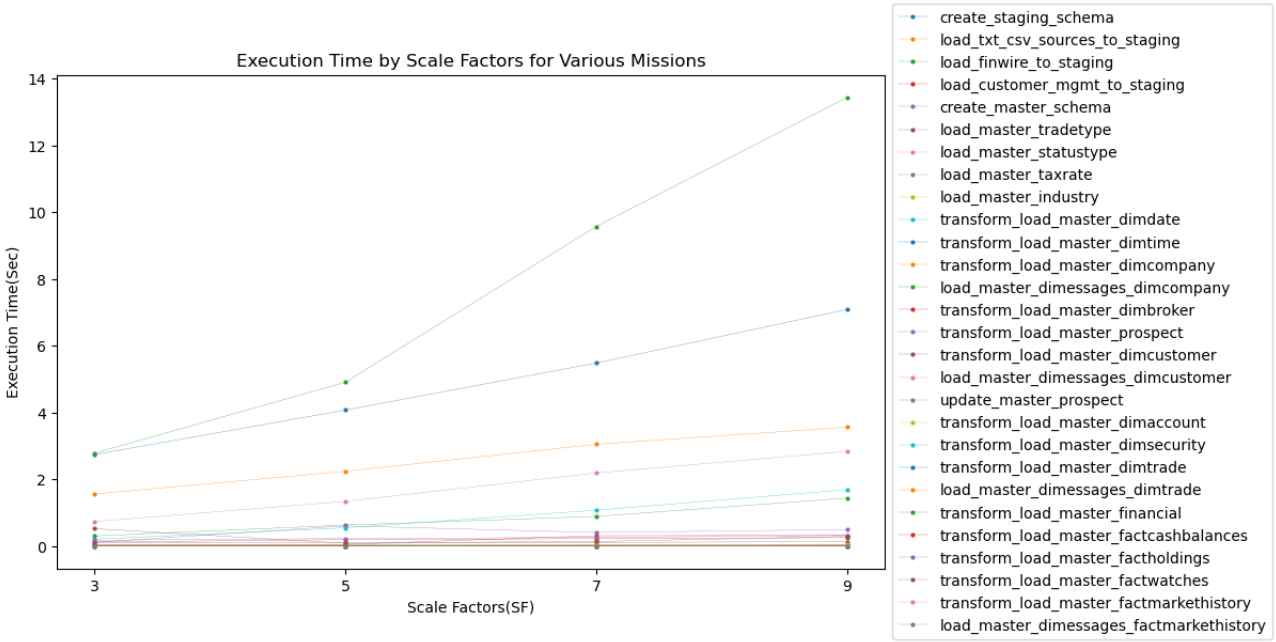


Figure 5.8: execution times for all tasks after removing two tasks

Then we also removed more time-consuming tasks which execution times were above 0.6s as the

Figure 5.9 shown. We can also know that most of the tasks are executed in less than 0.6 seconds.

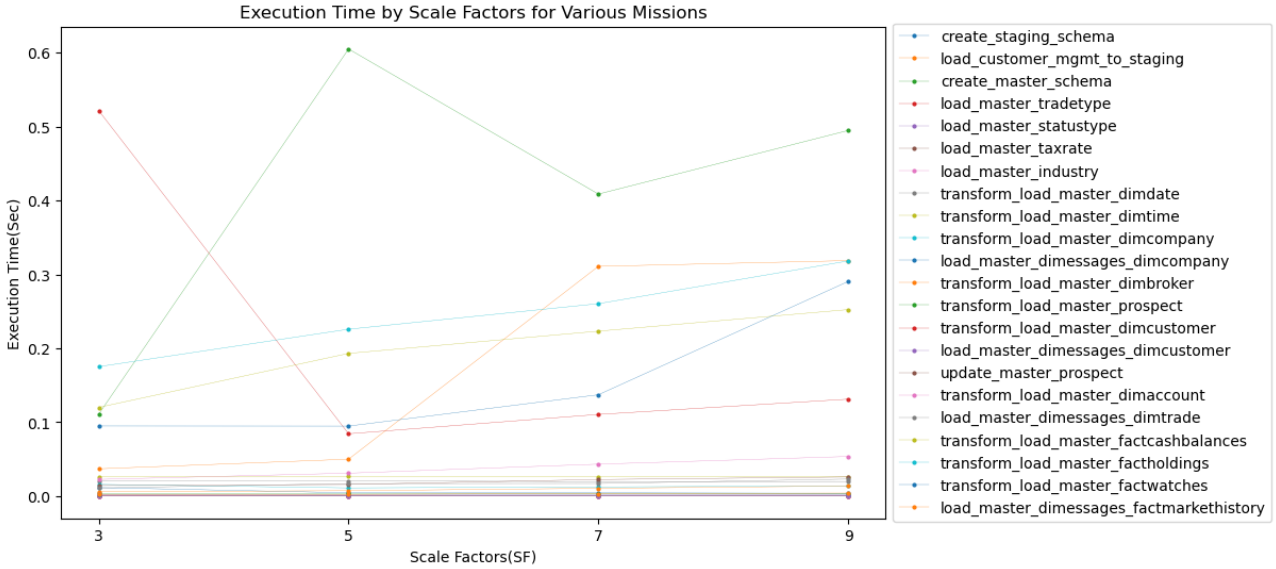
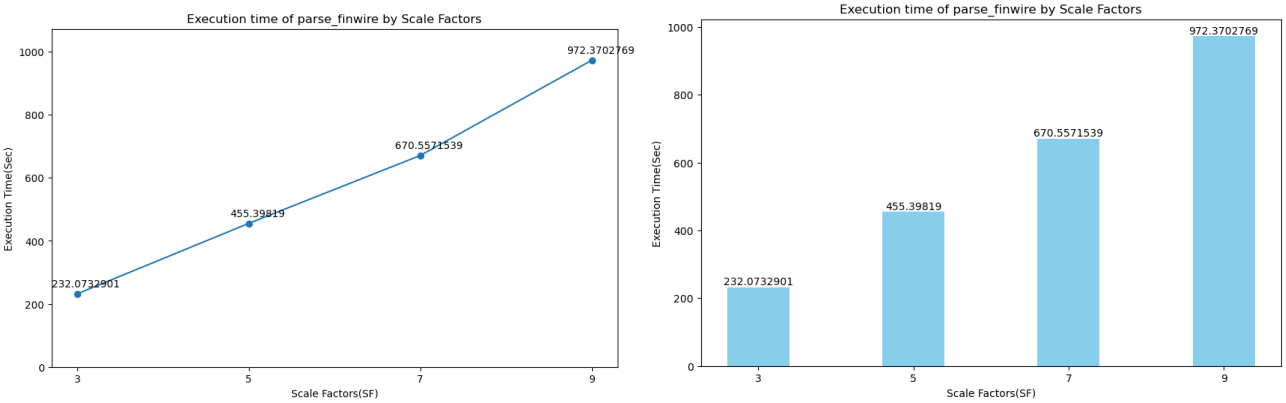


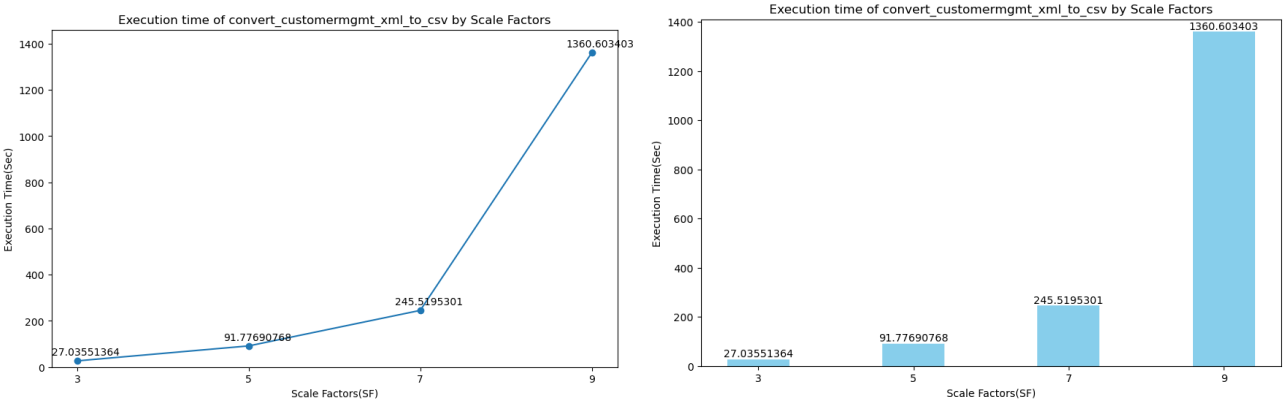
Figure 5.9: execution times for all tasks after removing more tasks

The two most time-consuming tasks were `parse_finwire` and `convert_customermgmt_xml_to_csv`. They are all in that hundreds seconds level compared to other tasks.



(a) Line chart (b) Bar chart

Figure 5.10: execution times for `parse_finwire`.

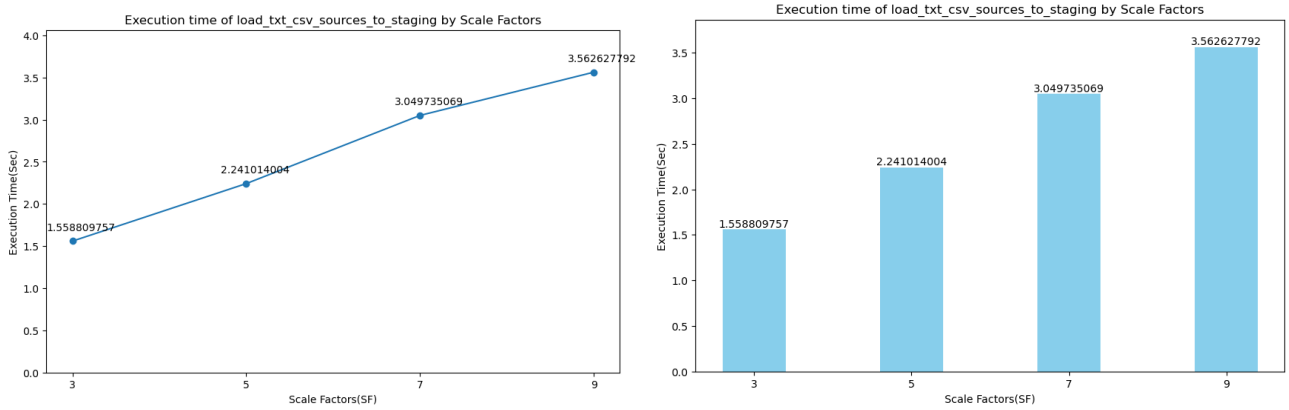


(a) Line chart (b) Bar chart

Figure 5.11: execution times for `convert_customermgmt_xml_to_csv`.

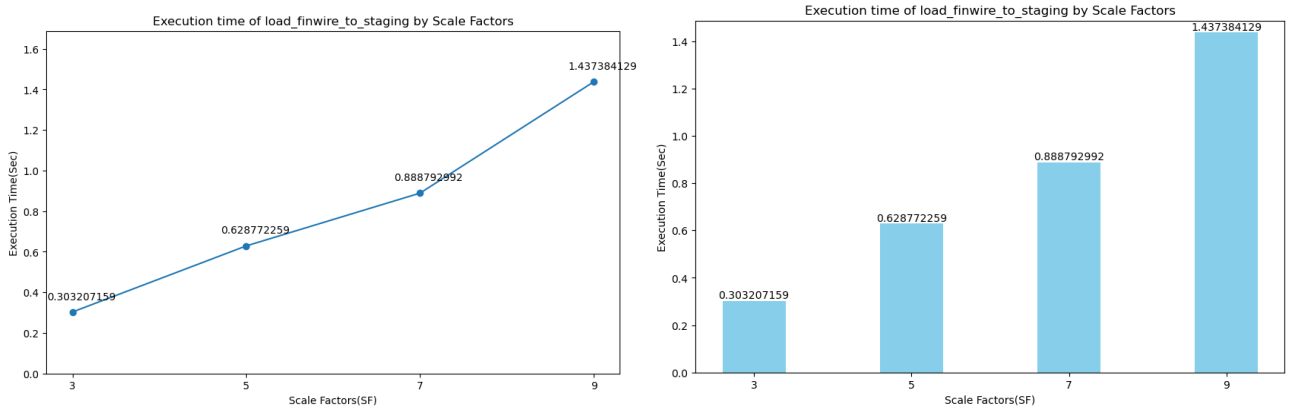
5.2.1 The tasks of linear growth

It could be seen that there were many tasks that grew linearly, such as `load_txt_csv_sources_to_staging`, `load_finwire_to_staging`, `transform_load_master_dimaccount` and `transform_load_master_dimtra` These tasks were mainly load tasks. The charts also showed that the loading time increased almost linearly as the data was increased. We thought the processes used for loading the data may have linear complexity (O(n)), meaning that their execution time increases proportionally with the size of the input data. It was same to the load performance we tested in TPC-DS project.



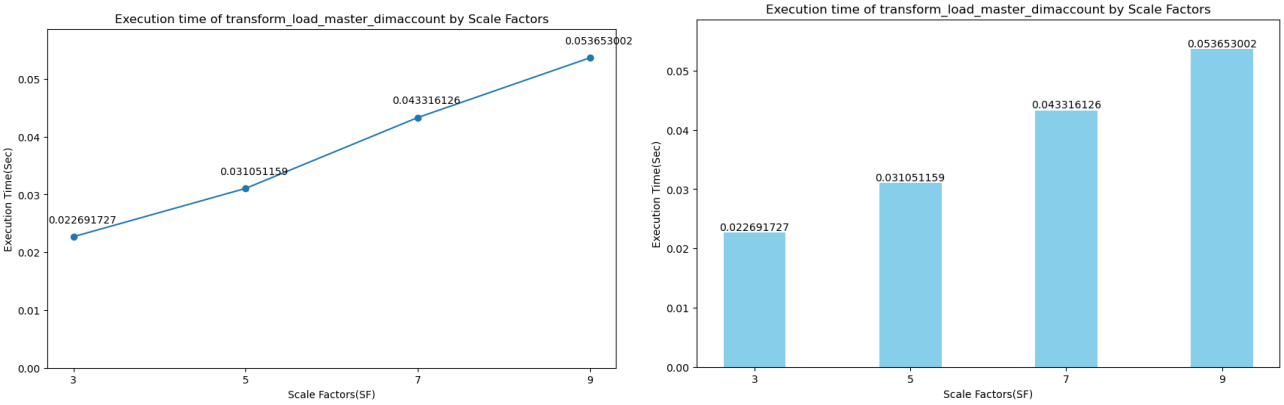
(a) Line chart (b) Bar chart

Figure 5.12: execution times for `load_txt_csv_sources_to_staging`.



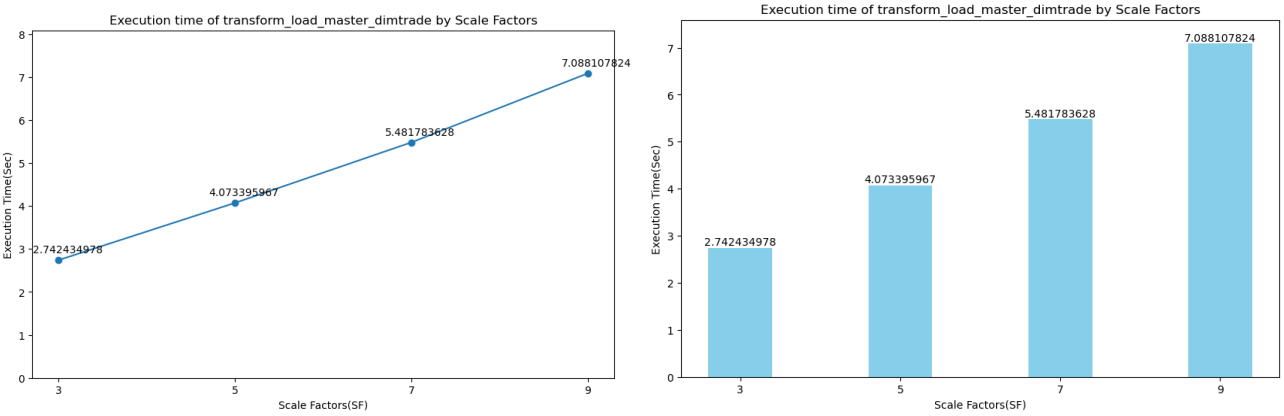
(a) Line chart (b) Bar chart

Figure 5.13: execution times for `load_finwire_to_staging`.



(a) Line chart (b) Bar chart

Figure 5.14: execution times for `transform_load_master_dimaccount`.

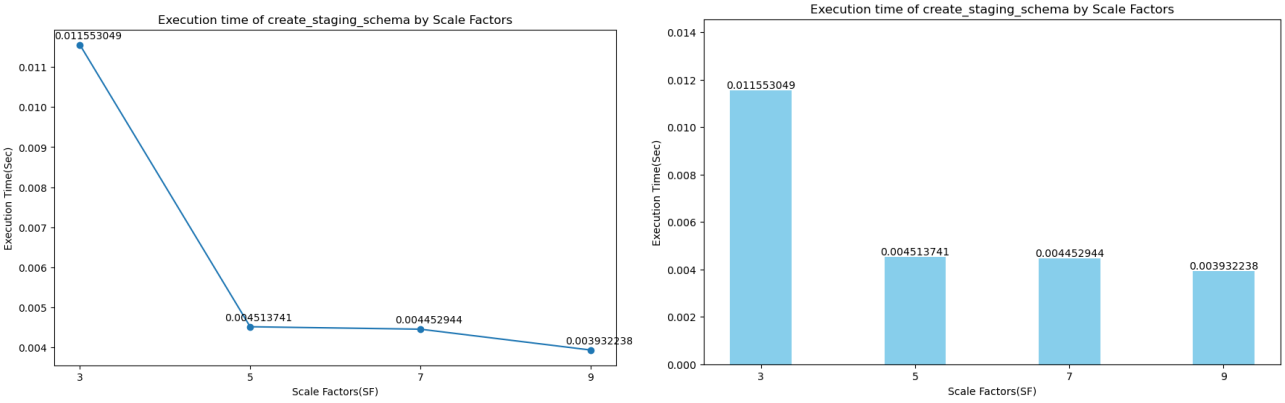


(a) Line chart (b) Bar chart

Figure 5.15: execution times for `transform_load_master_dimtrade`.

5.2.2 The tasks of decreasing tendency

But there are still some that do not show a linear growth trend. Instead, it was a decrease tendency such as create_staging_schema. We thought it was related to Caching: The system might be caching some of the operations required for creating the staging schema, making subsequent operations faster.

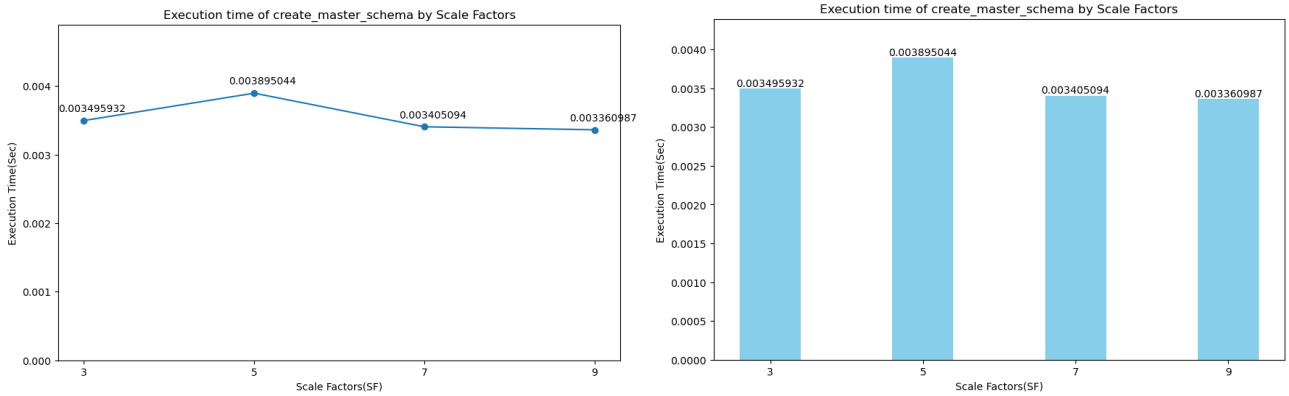


(a) Line chart (b) Bar chart

Figure 5.16: execution times for `create_staging_schema`.

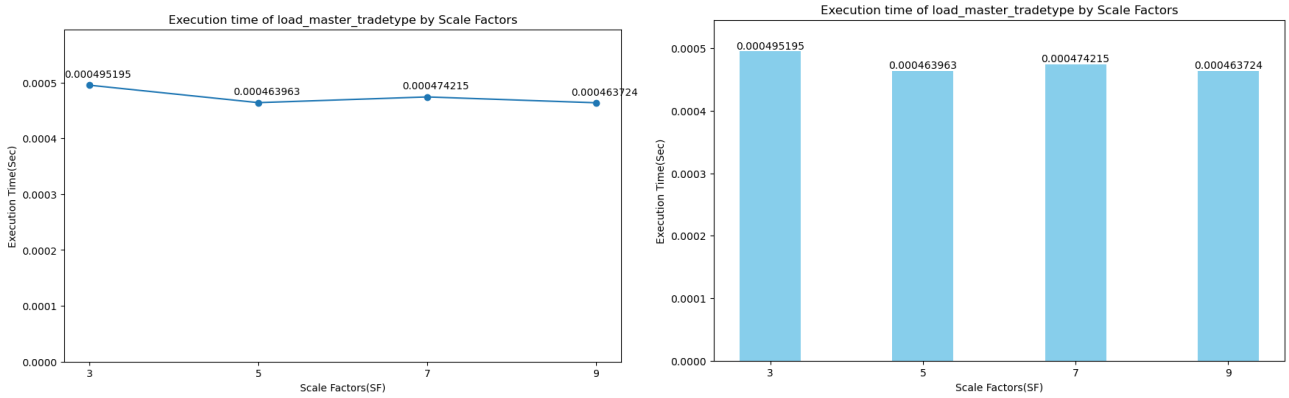
5.2.3 The tasks of stable trend

In addition, there were some tasks showed a stable up and down trend, such as create_master_schema and load_master_tradetype. We also thought it was related to caching effects: Repeated executions may benefit from caching, where data or execution plans are stored in memory, speeding up subsequent operations even the scale factors of data was increased.



(a) Line chart (b) Bar chart

Figure 5.17: execution times for `create_master_schema`.



(a) Line chart (b) Bar chart

Figure 5.18: execution times for `load_master_tradetype`.

5.3 Discussion

After conducting a benchmarking analysis across a broad spectrum of tasks with several scale factors, we got valuable information on how Duckdb handles escalating data volumes as well as modifications to node steps based on table data size. While the most majority of tasks showed non-linear runtime performance characteristics, there would be some tasks were notable exceptions.

6 Conclusion

6.1 Conclusion

The TPC-DI framework offers an extensive ETL benchmarking model for DuckDB. This detailed report provides a clear blueprint for replicating our benchmarking results with DuckDB. Through this project, we gained practical insights into Data Integration for systems, learning about the complexities of sourcing data from diverse formats and integrating it into decision support systems through various processing methods.
In this project, We have systematically learned the basic theories and concepts of TPC-DI benchmark and implemented it on DuckDB, which is an open-source and particularly fast database. We finished the data generation. We conducted experiments for four scale factors. It was evident that the only some execution time of load tasks grew linearly with the size of data.Most of the execution time of tasks did not follow the linear tendency. Then we presented and analyzed the results of our experiments in some charts with data visualization.
One significant challenge encountered was the TPC-DI documentation was poorly articulated.

6.1.1 Further Works

To fully complete our TPC-DI implementation, there are key steps yet to be executed:

- Implementing the two phases of Incremental Updates.
- Implementing the Automated Audit Phase

References

[TPC14] (TPC), Transaction Processing Performance Council (2014). “TPC Benchmark D1 Standard Specification”. In.

[Cod23] Code, Visual Studio (2023). “Visual Studio Code - Code Editing”. In: URL: https://code.visualstudio.com/docs.

[JZ20] Judith Awiti, Alejandro A. Vaisman and Zimányi (2020). “Design and implementation of ETL processes using BPMN and relational algebra”. In: *Data & Knowledge Engineering*.

[Koh+22] Kohn, André et al. (2022). “DuckDB-wasm: fast analytical processing for the web”. In: *Proceedings of the VLDB Endowment* 15.12, pp. 3574–3577.

[Mah21] Mahmud, Md Hedayet (2021). “What is benchmarking and how can it be used to measure our current performance against an ideal or best practice model?” In: URL: https://www.quora.com/What-is-benchmarking-and-how-can-it-be-used-to-measure-our-current-performance-against-an-ideal-or-best-practice-model.

[Pyt23] Python (2023). “General Python FAQ”. In: URL: https://https://docs.python.org/3/faq/general.html/.

[RM19] Raasveldt, Mark and Hannes Mühleisen (2019). “DuckDB: an Embeddable Analytical Database”. In: *International Conference on Management of Data (SIGMOD ’19)*. URL: https://dl.acm.org/doi/abs/10.1145/3299869.3320212.

[SG22] Saiid, Ahmad Abu and Rishika Gupta (2022). “TPC-DI-Benchmark: Data Integration with PostgreSQL & Airflow”. In: URL: https://github.com/risg99/tpc-di-benchmark.

[Sam21] Samy Kabangu, John Ebden (2021). “Benchmarking Databases”. In.