
A TCP-DI benchmark implementation using Citus

Marwah Sulaiman

Nishant Sushmakar

Olha Baliassina

Sara Saad

Prof. Esteban Zimányi

2024



1	Introduction	1
2	TPC Benchmark™ DI	2
2.1	Overview	2
2.2	Core Functionalities and Advantages	2
2.3	Phases of TPC-DI Benchmark	3
2.3.1	Preparation Phase	3
2.3.2	Initialization Phase	4
2.3.3	Historical Load Phase	4
2.3.4	Incremental Update Phases	4
2.3.5	Automated Audit Phase	5
2.4	Data in TPC-DI	5
2.5	Source Data Model	5
2.5.1	Internal Data Sources	5
2.5.2	External Data Sources	6
2.5.3	Reference Data	6
2.5.4	Customer Management System (CMS)	7
2.5.5	Target Data Model	7
2.5.6	Scaling	7
3	Citus Overview	8
3.1	Distributing PostgreSQL: Architecture, Features, Performance	8
3.1.1	Problem: Bottlenecks in PostgreSQL Scaling	8
3.2	Solution: Citus as a PostgreSQL Extension	8
3.3	Architecture and Functionality	9
3.3.1	Cluster Setup	9

3.3.2	Query Planning	9
3.3.3	Distributed Transactions	9
3.3.4	Distributed Deadlock Detection and Recovery	9
3.3.5	Simplified Management	9
3.4	Performance and Scalability	10
4	Airflow	11
4.1	Apache Airflow Overview	11
4.2	Core Functionalities and Advantages	11
4.2.1	Python-Based Configuration	11
4.2.2	Task Modularity and Parallelism	11
4.2.3	Robust Scheduling	12
4.2.4	User-Friendly Web Interface	12
4.2.5	Extendibility and Operators	12
4.3	How Airflow Complements Citus	12
5	Set up and Configurations of TPC-DI, Citus and Airflow	14
5.1	System specifications	14
5.2	Downloading the TPC-DI Kit	15
5.3	Data Generation with DiGen	15
5.3.1	Install Java 1.8 (ARM64 version)	15
5.3.2	Configure Java Environment Variables	15
5.3.3	Set Up the TPC-DI PDGF Tool	16
5.3.4	Verify PDGF Setup	16
5.3.5	Generate Data Using PDGF	16
5.4	Citus Installation	16
5.4.1	Initializing the Cluster	16
5.4.2	Configuring Master and Worker Nodes	17
5.5	Creating the Databases	17
5.6	Airflow Installation	18
5.7	Airflow Setup for Citus	19
6	Executing the benchmark	21
6.1	Data Warehouse implementation	21
6.1.1	Schema Preparation	21
6.1.2	Temp Schema	21
6.1.3	Distribution	22

6.2	TPC-DI ETL Workflow	23
6.2.1	Overall DAG Structure	23
6.2.2	Schema and Configuration Tasks	23
6.2.3	Creating Schemas	23
6.2.4	Dependency Ordering for Initialization	25
6.3	Initial Validation Task	25
6.4	Reference and Batch Date Loads	25
6.5	Loading and Transforming Core Dimensions	26
6.5.1	load_dimBroker	26
6.5.2	Parsing FINWIRE Files (Python)	26
6.6	Final Validation	26
6.6.1	run_validation_query_HistLoad	26
6.7	Task Dependencies and Flow	27
6.8	Summary of the ETL Process	27
6.9	Implemented ETL Workflow	27
6.9.1	Initialize and Prepare	27
6.9.2	Load Reference Data	27
6.9.3	Transform and Load Dimensions	27
6.9.4	Transform and Update	28
6.9.5	Validation	28
6.10	Metrics	28
6.10.1	Completion Timestamp	28
6.10.2	Elapsed Time and Row Count	29
6.10.3	Throughput Calculation	29
6.10.4	Validation Query Usage	29
6.11	Scaling	30
7	Performance Evaluation	31
7.1	Scale 3	32
7.2	Scale 5	32
7.3	Scale 10	33
7.4	Scale 15	33
7.5	Retrieving Task Durations for DAG Runs	36
8	Conclusion, Limitations, and Future Directions	37
8.1	Conclusion	37
8.2	Constrains in the Research	37

8.3 Future work 37

9 Reference Lists 38

For the second part of our Data Warehousing project, our team was tasked with implementing the TPC Benchmark™ DI (TPC-DI), which models a typical Extract, Transform, Load (ETL) process. We utilized Apache Airflow as the data integration tool to manage the ETL workflows, which were designed to populate a data warehouse implemented on Citus, our selected Database Management System (DBMS).

Our implementation of TPC-DI required testing with various scale factors (SF), each corresponding to different sizes of the data warehouse. This variation allowed us to analyze the performance and scalability of the ETL process under different data volumes, ensuring a comprehensive evaluation of the system's capability.

The report provides an overview of the benchmarking process performed in accordance with the TPC Benchmark™ DI v.1.1.0 standard specification, which was used as an implementation guide.

This report's explanations are based on the project's GitHub repository, accessible via the link below:
<https://github.com/NishantSushmakar/tpcdi-citus>

2.1 Overview

TPC-DI, the Data Integration Benchmark developed by the Transaction Processing Performance Council (TPC), is an industry-standard benchmark designed to evaluate the performance of data integration tools. These tools, often referred to as Extract, Transform, and Load (ETL) tools, are used to move and integrate data between different systems. By modelling a real-world scenario of extracting data from various sources, transforming it, and then loading it into a data warehouse for analytical and reporting purposes, the TPC-DI benchmark offers a representative view of modern data integration requirements. It also tests various system components associated with DI environments.

The TPC-DI benchmark is designed to provide objective and relevant performance data for users in the industry. It helps users compare the performance of different DI tools by providing a standardized way to measure their throughput and efficiency. The benchmark provides a comprehensive evaluation of DI tools, considering factors like data volume, transformation complexity, data accuracy, and consistency.

2.2 Core Functionalities and Advantages

- TPC-DI mirrors a real-world data integration scenario. It models a retail brokerage environment, simulating the integration of data from a variety of sources, such as an OLTP system, a customer management system, and financial newswire services.
- By following a standardized schema, dataset, and queries, different systems can be measured under similar conditions to compare their performance.
- TPC-DI covers multiple metrics such as load time, throughput, data volume, and resource utilization, providing a well-rounded view of performance.
- The benchmark encompasses a comprehensive workload with a wide range of data integration tasks.
- The Scale Factor parameter (SF), used for the adjustment of the data volume, enables users to test DI tools under different load conditions and extrapolate the results to their specific requirements.

2.3 Phases of TPC-DI Benchmark

TPC-DI has a defined ordered set of stages (or phases) which need to be completed in order to perform the benchmark.

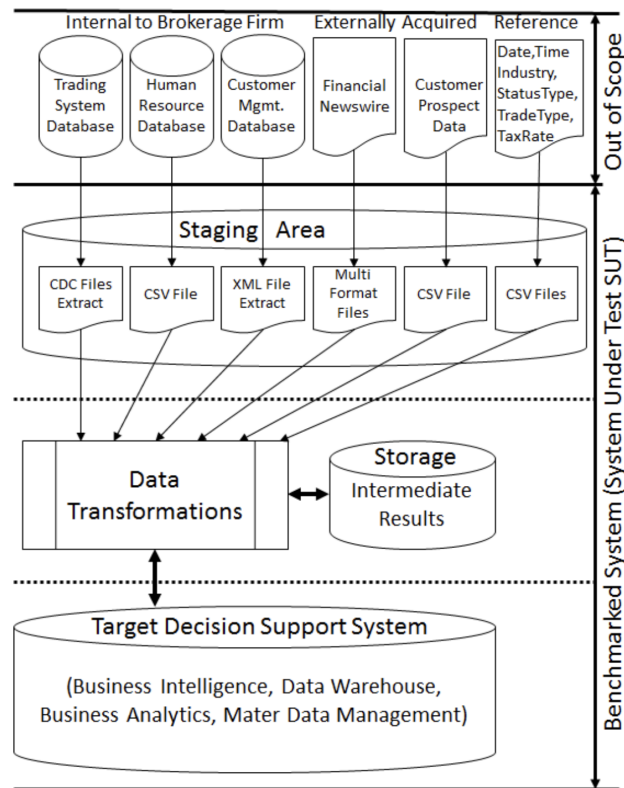


Figure 2.1: Benchmark System and Workflow

2.3.1 Preparation Phase

Data Generation

Data generation for the chosen scale factor is executed using the data generator tool DiGen, mentioned earlier. This process can be executed directly in the Staging Area or in an alternate location, with the generated data subsequently transferred to the Staging Area prior to the Historical Load. The time taken for data generation and copying to the Staging Area is excluded from the benchmark measurements.

Data Warehouse Creation

Setting up the Data Warehouse, including installing the product which will run the Data Warehouse (in our case - Citus), creating databases and tables and allocating disk space, is considered more of an administrative task rather than a data integration operation. As such, it is not included in the timed portion of the benchmark.

Data Integration Preparation

The data integration software (in our case - Apache Airflow) may require setup and configuration steps to execute the benchmark. These steps can vary depending on the implementation and are also not timed for the benchmark evaluation.

2.3.2 Initialization Phase

The initialization phase ensures that the System Under Test (SUT) is prepared for a fresh Benchmark Run, with no residual data or artifacts from previous runs. During this phase, the Data Warehouse must be created and made ready for subsequent operations. Once all initialization steps are completed, the batch validation process, as described in Clause 7.4, is executed. Upon the successful execution of the batch validation query, a Phase Completion Record (PCR) is logged in the DImessages table. This phase is not included in the timed portion of the benchmark.

2.3.3 Historical Load Phase

During the Historical Load phase, the data warehouse tables are initially empty and are populated with data from the Batch1 directory in the staging area, following the transformations outlined in Clause 4.5 of the Specification document (e.g., converting formats, cleaning invalid records, deduplication). Certain source files may have unique ordering; for example, data unloaded from an OLTP table might be organized by primary key, while CDC extracts are usually ordered by the time of changes. Data arrives in various formats (text files, CSV, XML), requiring the DI system to handle multiple parsing and transformation rules. In addition, this phase also relies on a larger volume of data and additional sources than the Incremental Update phase.

After the Historical Load completes, a Validation Query runs to collect data used for correctness checks in the automated audit. Although this phase is timed and affects the benchmark metric, it can take as long as needed to fully process all data. Upon finishing the execution of the validation query, a Phase Completion Record (PCR) is written to the DImessages table to mark the phase's conclusion.

2.3.4 Incremental Update Phases

This phase mirrors the continuous stream of modifications that occur in real-world environments. It applies various changes received from source systems to the data warehouse so that the stored information remains accurate and up to date. Typical operations involve inserting new records, updating existing entries, and deleting those that are no longer valid. These adjustments are often tracked and managed through Change Data Capture (CDC) files, which log every data change within the source. By evaluating how the system handles these updates in real-time or near-real-time, the Incremental Update Phase measures the efficiency and responsiveness of the data integration pipeline.

2.3.5 Automated Audit Phase

Upon completing the data movements and transformations, the Automated Audit Phase validates the accuracy of the results. Pre-defined validation queries are run against the loaded data and compared with expected outputs to confirm consistency and completeness. Any discrepancies or errors discovered during this step are logged for deeper analysis, ensuring that the integrations and transformations satisfy the benchmark's criteria for data quality and reliability.

However, in this project we focused on executing only the Preparation, Initialization and Historical load phases of the benchmark.

2.4 Data in TPC-DI

The TPC-DI benchmark simulates a fictional retail brokerage environment that integrates diverse data types into a target data warehouse. The benchmark mirrors real-world scenarios where data from multiple systems and file formats is processed for analytics and decision support.

The data is generated using DiGen, a tool based on the Parallel Data Generation Framework (PDGF). PDGF supports all file formats required by TPC-DI, such as CSV, text, multi format, and XML.

2.5 Source Data Model

The TPC-DI source data model includes three primary categories of data:

- **Internal Data:** Originating from the brokerage's internal operational systems.
- **External Data:** Acquired from external providers, such as marketing agencies or financial newswires.
- **Reference Data:** Static information required during the initial (historical) load phase of the benchmark.

During the Historical Load phase, these data categories are represented as files that are extracted, transformed, and integrated into the target data warehouse. This process evaluates the data integration (DI) system's ability to handle diverse input formats and transformation rules under realistic conditions.

2.5.1 Internal Data Sources

Online Transaction Processing (OLTP) Database

The OLTP database is the central operational system for the fictional brokerage. It contains transactional records related to securities market trading. During the Historical Load phase, full table extracts are generated, with each table stored in a separate file, such as `Account.txt`, `Trade.txt`, `TradeHistory.txt`, `CashTransaction.txt`, `HoldingHistory.txt`, `DailyMarket.txt`, and `WatchItem.txt`. These extracts provide a complete snapshot of the operational data at a single point in time.

Human Resource (HR) System

The HR system contains employee information, including job descriptions, branch locations, and management hierarchies. For TPC-DI, this data is stored in a single file, typically named `HR.csv`. Like the OLTP extracts, it is a full table extract used during the Historical Load phase.

Customer Relationship Management (CRM) System

CRM systems are used to track customer interactions and account details. In TPC-DI, the CRM component provides customer-centric data that complements the OLTP and HR systems. Specific extracts and file formats may vary depending on the scenario.

2.5.2 External Data Sources

FINWIRE Service

The FINWIRE service represents financial data sourced from specialized newswire services. It includes historical information about publicly traded companies and securities. The files follow a quarterly naming structure, such as `FINWIRE2003Q1`, and include the following record types:

- **CMP (Company):** High-level company information.
- **SEC (Security):** Details about stocks and other securities.
- **FIN (Financial):** Financial metrics and statements.

By simulating data over multiple quarters, FINWIRE introduces realistic challenges to data integration tasks.

Customer Prospect Data

The PROSPECT file, named `Prospect.csv`, lists potential customers with demographic and contact details. This file is modeled as a daily extract, mimicking the routine acquisition of leads from external providers. Duplicates and inconsistencies are included, requiring the DI system to perform data cleansing and deduplication.

2.5.3 Reference Data

Reference data in TPC-DI consists of static information that does not change after the Historical Load phase. It includes the following files:

- `Date.txt` and `Time.txt`: Calendar and time details.
- `Industry.txt`: Industry classifications and segments.
- `StatusType.txt`: Definitions for account or trade statuses.
- `TaxRate.txt`: Tax rate information.
- `TradeType.txt`: Definitions of trade types.

These files emulate real-world reference tables that are updated infrequently, if at all.

2.5.4 Customer Management System (CMS)

The Customer Management System (CMS) delivers historical customer and account information in an XML file, typically named `CustomerMgmt.xml`. This file reflects actions such as opening or closing accounts and updating customer information. The XML format adds complexity to the DI system, which must parse and transform the data into a format suitable for the target data warehouse.

2.5.5 Target Data Model

The target data model represents a dimensional data warehouse, following the snowflake schema design. This structure is common in data warehousing and is designed for efficient querying and analysis.

- **Dimension Tables:** Describe key entities such as Date, Time, Customer, Account, and Security.
- **Fact Tables:** Track measurable events such as Trades, Holdings, Cash Balances, and Market History.
- **Reference Tables:** Store supplementary information used during transformations.

Figure 2.2 demonstrates the target data model.

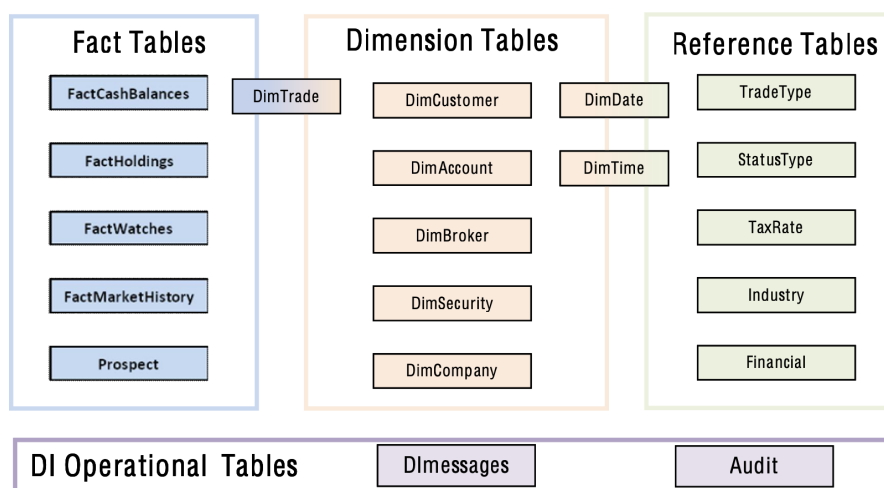


Figure 2.2: Pictorial overview of the Data Warehouse Tables

2.5.6 Scaling

TPC-DI uses continuous scaling based on the number of customers of the fictitious brokerage firm. The number of unique customers UCH that are present in the historical data set can be calculated as:

$$UCH(SF) = SF * 5000.$$

The source data for the fact tables and most dimension tables scale linearly with SF.

3.1 Distributing PostgreSQL: Architecture, Features, Performance

3.1.1 Problem: Bottlenecks in PostgreSQL Scaling

PostgreSQL itself is a very good relational database, but for certain workloads, it does have scalability bottlenecks. For example:

- **Software as a Service (SaaS):** When customers grow rapidly, large working sets can overload a single PostgreSQL instance. Traditional multi-tenancy approaches are unwieldy and not scalable.
- **Real-time Analytics:** Highly concurrent queries on constantly updating data pose scalability challenges for traditional OLAP stores and standalone PostgreSQL instances.
- **Key-Value Storage:** While JSON support in PostgreSQL makes it appealing for NoSQL use cases, update-heavy workloads suffer from high write amplification and scalability bottlenecks.
- **OLAP Workloads:** Complex query analysis of large datasets demands considerable processing power, which may not be provided by a single instance of PostgreSQL.

3.2 Solution: Citus as a PostgreSQL Extension

To address these challenges, Citus extends PostgreSQL for horizontal scalability by providing the following features:

- **Distributed Tables:** Data is transparently sharded across worker nodes, each running an instance of PostgreSQL, allowing horizontal scaling for large datasets.
- **Reference Tables:** These are replicated on every node, enabling fast JOINS and foreign key relationships with guaranteed consistency.
- **Columnar Storage:** In its newer versions, columnar storage efficiently executes analytical queries.

Citus's architecture leverages the strengths of PostgreSQL:

“So, in a nutshell, an extension is a collection of SQL objects—functions and tables that store metadata—and one shared library that implements those functions or modifies the behavior of

Postgres.”

This architecture ensures compatibility with PostgreSQL features and extensions, such as PostGIS, as well as its development cycle.

3.3 Architecture and Functionality

3.3.1 Cluster Setup

Standard PostgreSQL servers are set up as worker nodes, with one acting as the coordinator responsible for managing metadata and client connections.

3.3.2 Query Planning

A planner hook intercepts the queries, analyzes them, and determines the optimal execution strategy, including:

- **Direct Routing:** Queries filtered by the distribution column are routed to the appropriate shard for efficient CRUD operations.
- **Colocated JOINS:** Tables located on the same worker node allow local JOINS, efficiently supporting multi-tenant transactions.
- **Parallel Query Execution:** Queries without filters by the distribution column are parallelized across shards using the adaptive executor.

3.3.3 Distributed Transactions

- **Single-Node Transactions:** Execute on a worker node while maintaining PostgreSQL’s MVCC semantics; can scale out to multiple nodes.
- **Multi-Node Transactions:** Implemented using two-phase commit (2PC) with PostgreSQL’s transaction callbacks and a commit log to ensure atomicity.

3.3.4 Distributed Deadlock Detection and Recovery

A background worker periodically analyzes lock graphs for distributed deadlocks and resolves them by canceling transactions.

3.3.5 Simplified Management

Citus facilitates easy creation and management of distributed tables using SQL commands.

3.4 Performance and Scalability

Benchmarking demonstrates Citus's effectiveness for various workloads:

- **Multitenant:** HammerDB TPROC-C shows nearly linear scalability, making it suitable for SaaS applications.
- **Analytical:** For in-memory fitting data, parallelization leads to substantial performance gains for analytical queries.
- **Key-Value:** YCSB shows linear scalability, effectively handling NoSQL workloads.

Citus employs both distributed processing and in-memory techniques for optimal results.

4.1 Apache Airflow Overview

Apache Airflow is an open-source platform for creating, scheduling, and managing data workflows. It was originally developed by Airbnb to orchestrate a variety of data-related tasks on a daily basis. The main concept behind Airflow is the **Directed Acyclic Graph (DAG)**, which structures tasks so that each step naturally leads into the next without forming loops.

Modern data systems often require multiple stages — such as extracting data from a source, cleaning and transforming it, and then loading it into a target database or analytical system. Airflow brings these stages under a single, centralized framework. By representing the various steps of a pipeline as DAGs, it becomes possible to visualize, schedule, and manage complex data processes with a high level of transparency.

4.2 Core Functionalities and Advantages

4.2.1 Python-Based Configuration

Airflow workflows are defined in Python, which allows the inclusion of standard programming constructs like loops, conditionals, and imports. This code-centric approach simplifies version control, testing, and collaboration, making it easy to reuse or modify existing workflows when new requirements arise.

4.2.2 Task Modularity and Parallelism

Tasks within an Airflow workflow can run independently and, when resources permit, in parallel. This modular layout helps isolate errors and enables the reuse of task definitions across different pipelines. It also speeds up processing times by taking full advantage of available computing power.

4.2.3 Robust Scheduling

Airflow's built-in scheduler triggers tasks based on specified intervals or conditions (e.g., hourly, daily, or on demand). This design accommodates batch processing workloads, which are common in data engineering and integration scenarios. Specifying start times, end times, and intervals allows for flexible management of various routine tasks.

4.2.4 User-Friendly Web Interface

A web-based interface is available to monitor active workflows, inspect logs, retry failed tasks, and view historical runs. This interface is highly beneficial for troubleshooting, as it makes it straightforward to locate precisely where and when a workflow might have encountered issues.

4.2.5 Extendibility and Operators

Airflow includes a large set of “operators,” which are specialized snippets of code that run tasks for specific technologies, such as databases, cloud platforms, and data lakes. When an operator for a particular system is not pre-built, a custom operator can be developed in Python. This extensibility ensures Airflow can integrate with a wide range of external services and systems.

Additional Benefits. Beyond these primary features, Airflow supports parameterized workflows (leveraging Jinja templating), can easily incorporate version control for collaborative development, and allows pipelines to be tested much like any piece of software. These benefits reduce the risk of errors while maintaining a flexible, code-based framework for orchestrating data tasks.

Airflow helps automate complex data pipelines, removing the need for repeated manual steps such as loading data and running transformations. It also offers detailed logging and notifications, so failures or slowdowns can be spotted quickly. For systems that handle large amounts of data, Airflow can spread tasks across multiple machines to make sure everything runs in a reasonable amount of time. Its open-source nature and active community mean new features, tutorials, and plugins are frequently added.

4.3 How Airflow Complements Citus

When combined with **Citus**, Airflow can manage data pipelines that feed into a distributed environment. The following points show some ways Airflow and Citus work together:

- **Coordinated Ingestion:** Airflow controls the extraction and transformation stages and only loads data into Citus after these steps finish, preserving data consistency.
- **Parallel Loading:** Since Citus spreads tables across workers, Airflow can create tasks to load chunks of data in parallel, speeding up large data loads.

- **Incremental Updates:** Airflow can be set up to apply Change Data Capture (CDC) records or regular data refreshes. Citus scales horizontally to handle these updates efficiently.
- **Integrated Monitoring:** Airflow's logs and alerts show pipeline performance, while Citus offers a view of node-level operations. This combination simplifies troubleshooting.
- **Easy Growth:** As data volumes increase, new Citus nodes can be added. Airflow can adjust its tasks to make use of extra resources, ensuring the pipeline stays efficient.

Set up and Configurations of TPC-DI, Citus and Airflow

5.1 System specifications

In order to preserve consistency, all the results presented in the project report were obtained using a single system. Its parameters are presented in the tables below.

Specification	Details
Chip	Apple M3 Pro
Memory	18 GB
Disk specs	SSD 512 GB

Table 5.1: Hardware specifications

Tool	Version
TPC-DI	1.1.0
Citus	12.1.4
PostgreSQL	14.13
Apache Airflow	2.10.4
Python	3.10.5
Java	1.8
Visual Studio Code	1.96

Table 5.2: Software specifications

As part of the project, in addition to the tools that were mentioned in Task description, we used Python to write scripts for Apache Airflow, making use of popular libraries like NumPy, Pandas, and XmlToDict to perform data processing tasks. We also relied on Visual Studio Code as our code editor, which helped us write, compile, debug the scripts, as well as manage version control for our code efficiently. In order to use the PDGF data generation tool for TPC-DI, we required installing Java. And since Citus extends PostgreSQL to enable distributed databases, all its nodes (coordinator and worker ones) must run the same version of

PostgreSQL to maintain compatibility and avoid issues with distributed query execution, data consistency, and functionality.

5.2 Downloading the TPC-DI Kit

1. Navigate to the official website of TPC: TPC Benchmarks.
2. Choose the TPC-DI kit in the list of downloadable benchmarking tools.
3. Fill in the required form.
4. Once the form is sent, check the email box and find the link for downloading the .zip archive.
5. Download and unpack it in a convenient folder, as it will be used extensively in the following stages.

5.3 Data Generation with DiGen

The following step-by-step guide explains how to configure and use the PDGF tool for generating data as part of the TPC-DI benchmark.

5.3.1 Install Java 1.8 (ARM64 version)

1. Visit the following link: [Java 8 LTS for macOS](#).
2. Download the version of Java 1.8 that supports ARM64 architecture (this is the only compatible version for TPC-DI on macOS with an M3 chip).
3. Install the downloaded Java package by following the provided instructions.

5.3.2 Configure Java Environment Variables

1. Open the terminal and run the command: `ls -al` to check if the `.zshrc` file exists in the home directory.
 - If the `.zshrc` file is not present, create it by running: `touch .zshrc`.
 - If the `.zshrc` file already exists, proceed to the next step.
2. Open the `.zshrc` file using the command: `open .zshrc`.
3. Copy and paste the following lines into the `.zshrc` file:

```
export PATH="/opt/homebrew/bin:$PATH"
export JAVA_HOME=$(/usr/libexec/java_home -v 1.8)
export PATH=$JAVA_HOME/bin:$PATH
```

4. Save the `.zshrc` file by pressing `Command + S`.
5. Apply the changes by running the command: `source .zshrc`.

6. Verify the Java installation by running: `java -version`. The output should be the same as the following:

```
openjdk version "1.8.0_432"
OpenJDK Runtime Environment (Zulu 8.82.0.21-CA-macos-aarch64) (build 1.8.0_432-b06)
OpenJDK 64-Bit Server VM (Zulu 8.82.0.21-CA-macos-aarch64) (build 25.432-b06, mixed mode)
```

5.3.3 Set Up the TPC-DI PDGF Tool

1. Navigate to the TPC-DI directory location on the system.
2. Rename the PDGF folder to lowercase (pdgf) for consistency.
3. Navigate into the `tools` directory within the `pdgf` folder.

5.3.4 Verify PDGF Setup

1. Run the following command to check if the PDGF tool is configured correctly:

```
java -jar DIGen.jar -h
```

2. If the help section is displayed, the setup is correct.

5.3.5 Generate Data Using PDGF

1. Manually create a data directory inside the TPC-DI directory if it does not already exist.
2. While inside the `tools` directory, run the following command to generate data:

```
java -jar DIGen.jar -o ../data/ -sf 5
```

3. This command specifies an output directory (`../data/`) and sets the scale factor to 5.

Once these steps are completed, the data is generated and ready to be used for the following stages.

5.4 Citus Installation

Refer to the Citus installation guide for macOS: [Citus Documentation](#).

5.4.1 Initializing the Cluster

1. Install PostgreSQL using Homebrew:

```
brew install citus
```

2. Create directories for the master and worker nodes:

```
cd ~  
mkdir -p citus/master citus/worker1 citus/worker2
```

3. Initialize the PostgreSQL instances:

```
initdb -D citus/master  
initdb -D citus/worker1  
initdb -D citus/worker2
```

5.4.2 Configuring Master and Worker Nodes

To configure the master to find the workers, append the following lines:

```
echo "localhost 9701" >> citus/master/pg_worker_list.conf  
echo "localhost 9702" >> citus/master/pg_worker_list.conf
```

Configure PostgreSQL instances for Citus:

```
echo "shared_preload_libraries = 'citus'" >> citus/master/postgresql.conf  
echo "shared_preload_libraries = 'citus'" >> citus/worker1/postgresql.conf  
echo "shared_preload_libraries = 'citus'" >> citus/worker2/postgresql.conf
```

5.5 Creating the Databases

1. Start the databases:

```
pg_ctl -D citus/master -o "-p 9700" -l master_logfile start  
pg_ctl -D citus/worker1 -o "-p 9701" -l worker1_logfile start  
pg_ctl -D citus/worker2 -o "-p 9702" -l worker2_logfile start
```

2. Drop any existing databases:

```
dropdb -p 9700 db_name
```

3. Create new databases with WIN1252 encoding:

```
createdb -p 9700 Citus_M --encoding='WIN1252' --locale='C' --template=template0  
createdb -p 9701 Citus_M --encoding='WIN1252' --locale='C' --template=template0
```

```
createdb -p 9702 Citus_M --encoding='WIN1252' --locale='C' --template=template0
```

4. Load the Citus extension:

```
psql -p 9700 Citus_M -c "CREATE EXTENSION citus;"
psql -p 9701 Citus_M -c "CREATE EXTENSION citus;"
psql -p 9702 Citus_M -c "CREATE EXTENSION citus;"
```

by default, Citus uses UTF-8 encoding, which may lead to loading issues if the data is not UTF-8 encoded. As it was the case for implementation of TPC-DS, our data might include international characters like 'Ô' and 'É' that must be perserved, and therefore we used the WIN1252 encoding while creating the databases, and we set the client encoding in psql to the same one for running queries, to receive expected outputs and maintain data consistency.

5. Connect to the master node:

```
psql -p 9700 -d Citus_M
```

and add workers to the cluster:

```
SELECT citus_add_node('localhost', 9701);
SELECT citus_add_node('localhost', 9702);
```

6. With the query below check whether the port 9701 and 9702 are shown in the table:

```
SELECT * FROM master_get_active_worker_nodes()
```

The two ports should be visible

5.6 Airflow Installation

By default, Airflow uses the directory ~/airflow as its home.

1. Create a home directory for Airflow:

```
mkdir ~/airflow
```

2. Navigate to the Airflow directory:

```
cd ~/airflow
```

3. Create a Python virtual environment:

```
python -m venv ./
```

4. Activate the virtual environment:

```
source bin/activate
```

5. Set the Airflow home directory:

```
export AIRFLOW_HOME=~/.airflow
```

6. Define the Airflow version and Python version:

```
AIRFLOW_VERSION=2.10.4
PYTHON_VERSION="$(python -c 'import sys;
    print(f"{sys.version_info.major}.{sys.version_info.minor}")')"
```

```
CONSTRAINT_URL="https://raw.githubusercontent.com/apache
    /airflow/constraints-${AIRFLOW_VERSION}
    /constraints-${PYTHON_VERSION}.txt"
```

7. Install Airflow using the constraints file:

```
pip install "apache-airflow==${AIRFLOW_VERSION}"
    --constraint "${CONSTRAINT_URL}"
```

8. Run Airflow in standalone mode:

```
airflow standalone
```

9. Access the Airflow web interface by opening a browser and navigating to:

```
http://localhost:8080
```

Use the credentials displayed in the terminal to log in.

5.7 Airflow Setup for Citus

1. Install the necessary Airflow provider for PostgreSQL. In the Airflow directory, run:

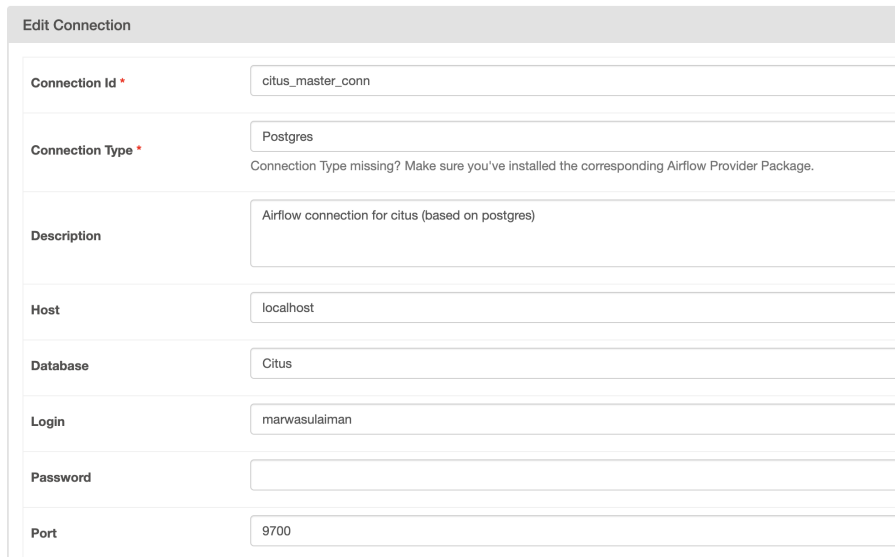

```
pip install apache-airflow-providers-postgres
```

2. Restart Airflow to apply the changes.
3. Open the Airflow UI:

```
airflow standalone
```

Then navigate to `http://localhost:8080/` in the browser.

4. Create a new connection in the Airflow UI:
 - (a) Go to Admin → Connections.
 - (b) Set the username to the username which was used to access the Citus database.
 - (c) Set the Database field to the Citus database name.



Edit Connection	
Connection Id *	citus_master_conn
Connection Type *	Postgres <small>Connection Type missing? Make sure you've installed the corresponding Airflow Provider Package.</small>
Description	Airflow connection for citus (based on postgres)
Host	localhost
Database	Citus
Login	marwasulaiman
Password	
Port	9700

Figure 5.1: Creating a new connection

5. Add the .py file containing the DAG to the dags directory.
6. Restart Airflow to ensure the new DAG appears in the UI.

6.1 Data Warehouse implementation

6.1.1 Schema Preparation

We created a `schema.sql` file containing the definitions for all required tables, including their columns, constraints, and indexes, as outlined in the TPC-DI benchmark specification (Figure 2.2). The schema creation process was automated using a DAG with a `create_schema` task executed via the `PostgresOperator`.

6.1.2 Temp Schema

A `temp_schema.sql` file was created to define "temporary" tables - the ones used for intermediate transformations:

- `hr:`
 - Contains employee data such as employee ID, manager ID, and contact details.
- `batchdate:`
 - Stores batch processing dates for ETL operations.
- `finwire_cmp:`
 - Holds company-related data from FINWIRE.
- `finwire_sec:`
 - Stores security-related data from FINWIRE.
- `finwire_fin:`
 - Contains financial data from FINWIRE, such as revenue, earnings, and assets.
- `prospect_temp:`
 - Stores temporary data related to customer prospects before final transformations.
- `customermgmt:`
 - Holds intermediate data for customer and account management updates.
 - Includes information such as customer actions, addresses, and contact details.
- `cashtransaction:`

- Temporarily records cash transaction details linked to accounts.
- `trade`:
 - A temporary table for trade operations, storing transactional data before transformation into final fact tables.
- `holdinghistory`:
 - Tracks intermediate changes in holdings for accounts, including before and after quantities.
- `dailymarket`:
 - Contains daily market data, such as high/low prices and volumes.
- `tradehistory`:
 - Temporarily records the history of trade operations, including timestamps and status.
- `watchhistory`:
 - Captures temporary data about watched securities, including activation and cancellation actions.

These tables enable the efficient processing of complex transformations without cluttering the final schema. For example, the `DimBroker` table was populated using data from `hr.csv`. The process involved the following steps:

1. A temporary table named `hr` was created, with a schema matching the structure of the `hr.csv` file.
2. Data was loaded from `hr.csv` into the `hr` temporary table.
3. The `DimBroker` table was populated using the data from the `hr` temporary table, with applying the required transformations in the insertion query.

The SQL script used to load the `DimBroker` table is located in the file `LoadBroker.sql`. The script was executed using the following Python code snippet:

```
with open('tpcdi-citus/LoadBroker.sql', 'r') as file:
    load_dimBroker_sql = file.read()

load_dimBroker = PostgresOperator(
    task_id="load_dimBroker",
    postgres_conn_id="citus_master_conn",
    sql=load_dimBroker_sql
)
```

6.1.3 Distribution

The data warehouse was created using a distributed Citus cluster with a single master and 2 worker nodes. For the designed schema we implemented categorizing tables into reference tables and distributed tables based on their size and usage patterns.

Smaller, dimension tables such as `dimAccount`, `dimBroker`, `dimCompany`, and `dimCustomer` were des-

ignated as reference tables. These tables were replicated across all worker nodes using the `create_reference_table` function, ensuring efficient joins without requiring network communication between nodes:

```
SELECT create_reference_table('table_name');
```

Larger fact tables such as `factCashBalances`, `factHoldings`, `factMarketHistory`, `dimtrade`, `prospect` and `factWatches` were designated as distributed tables. The `create_distributed_table` function was used to shard these tables across worker nodes based on a selected distribution column:

```
SELECT create_distributed_table('table_name', 'distribution_column');
```

The script `Distribution.sql` contains the queries for setting all distributed and reference tables.

Validation: Initial and final validation queries ensured that the schema and data adhered to TPC-DI compliance requirements.

6.2 TPC-DI ETL Workflow

6.2.1 Overall DAG Structure

The DAG for the TPC-DI Historical Load is defined as follows:

```
with DAG(  
    "TPCDI_Hist_Load",  
    start_date=datetime(2024, 12, 20),  
    schedule_interval="@once",  
    catchup=False,  
) as dag:
```

```
...
```

- **DAG Name:** `TPCDI_Hist_Load`.
- **Schedule:** `@once` (runs a single time when triggered).
- **Start Date:** December 20, 2024.
- **Catchup:** Disabled, so historical runs are not attempted.

All tasks are defined within this DAG context block, and their dependencies specify the order in which they must run.

6.2.2 Schema and Configuration Tasks

6.2.3 Creating Schemas

create_schema

```
create_schema = PostgresOperator(  
    task_id="create_schema",
```

```

postgres_conn_id="citus_master_conn",
sql=createSchema
)

```

- **Purpose:** Executes SQL commands in `schema.sql`, creating the main TPC-DI schema (tables, constraints, indexes).
- **Connection:** Uses `citus_master_conn`, indicating the target is the Citus coordinator node.

create_temp_schema

```

create_temp_schema = PostgresOperator(
    task_id="create_temp_schema",
    postgres_conn_id="citus_master_conn",
    sql=createTempSchema
)

```

- **Purpose:** Executes SQL from `temp_schema.sql`, creating a temporary schema for intermediate tables.
- **Use:** Allows temporary data storage without cluttering the main schema.

set_dist_ref_schema

```

set_dist_ref_schema = PostgresOperator(
    task_id='set_dist_ref_schema',
    postgres_conn_id='citus_master_conn',
    sql = set_dist_ref_tables_sql
)

```

- **Purpose:** Runs `Distribution.sql`, setting distribution keys for distributed tables and configuring reference tables in Citus.
- **Effect:** Distributed tables are sharded across worker nodes, and reference tables are replicated to each node for efficient joins.

set_path

```

set_path = PostgresOperator(
    task_id="set_path",
    postgres_conn_id="citus_master_conn",
    sql=f"""
        TRUNCATE TABLE config;
        INSERT INTO config (key_name, value_text)
        VALUES ('base_path', '{PATH}');
    """
)

```

```
    """
)

```

- **Purpose:** Stores the base file path (PATH) in a configuration table for reference in SQL scripts.

6.2.4 Dependency Ordering for Initialization

```
create_schema >> create_temp_schema >> set_dist_ref_schema >> set_path >>
    run_validation_query_Init
```

These dependencies ensure proper initialization sequence, culminating in an initial validation query.

6.3 Initial Validation Task

run_validation_query_Init

```
run_validation_query_Init = PostgresOperator(
    task_id="run_validation_query_Init",
    postgres_conn_id="citus_master_conn",
    sql=validation_query
)

```

- **Purpose:** Executes SQL in `tpcdi_validation.sql` to check initial conditions (e.g., ensuring no left-over data and verifying schema readiness) and adds a timestamp in the messages table.

6.4 Reference and Batch Date Loads

- **Batch Date:**

```
TRUNCATE TABLE batchdate;
COPY batchdate FROM '{PATH}/data/Batch1/BatchDate.txt';
```

- **Date Dimension:**

```
TRUNCATE TABLE dimdate;
COPY dimdate FROM '{PATH}/data/Batch1/Date.txt' DELIMITER '|';
```

- **Tax Rate:**

```
TRUNCATE TABLE taxrate;
COPY taxrate FROM '{PATH}/data/Batch1/TaxRate.txt' DELIMITER '|';
```

- **Industry:**

```
TRUNCATE TABLE industry;
COPY industry FROM '{PATH}/data/Batch1/Industry.txt' DELIMITER '|';
```

6.5 Loading and Transforming Core Dimensions

For this part, we executed all the tasks needed for the transformations and loading of the whole final schema of dimension and fact tables, but for simplicity, we include here only two examples. All the tasks can be found in the HistLoad.py script here. Moreover, all the tasks are using the sql scripts (e.g. LoadAccount.sql, LoadCompany.sql, LoadFinancial.sql, ..) which include the actual transformations and loading of the tables, and they can be viewed here.

6.5.1 load_dimBroker

```
load_dimBroker = PostgresOperator(  
    task_id="load_dimBroker",  
    postgres_conn_id="citus_master_conn",  
    sql=load_dimBroker_sql  
)
```

- **Purpose:** Executes LoadBroker . sql, doing the needed transformations and loading broker information into the dimBroker table.

6.5.2 Parsing FINWIRE Files (Python)

```
Parse_Finwire = PythonOperator(  
    task_id='Parse_Finwire',  
    python_callable=ProcessFinwire  
)
```

- **Purpose:** Processes FINWIRE files to extract and format company, security, and financial data into CSV files for database loading.

6.6 Final Validation

6.6.1 run_validation_query_HistLoad

```
run_validation_query_HistLoad = PostgresOperator(  
    task_id="run_validation_query_HistLoad",  
    postgres_conn_id="citus_master_conn",  
    sql=validation_query  
)
```

- **Purpose:** Executes final validation queries to ensure data consistency, row counts, referential integrity, and TPC-DI compliance. It also inserts the timestamp in the messages table to indicate completion of the process.

6.7 Task Dependencies and Flow

The created DAG outlines dependencies so that tasks run in a logical order:

1. **Schema Creation / Setup → Initial Validation**
2. **Initial Validation → Load Reference Tables** (BatchDate, dimDate, taxRate, etc.)
3. **Load Reference Tables → Load dimension data** (brokers, etc.) → **Parse/Load FINWIRE**
4. **Parse/Load FINWIRE → Load Company, Financial, Prospect → Convert/Load Customer Management**
5. **Complete Customer Dimension → Update Prospect → Load Account, CashBalances, Security**
6. **Load TradeHistory, Trade, etc. → Transform Trade dimension/fact**
7. **Load Market History, WatchHistory → Transform Fact tables**
8. **Final Validation**

6.8 Summary of the ETL Process

- **Initialize and Prepare:** Creates schemas, configures distribution settings, and runs initial validation.
- **Load Reference Data:** Populates foundational dimension and lookup tables.
- **Build Dimensions:** Loads brokers, companies, and prospects, transforming raw data into structured dimensions.
- **Transform and Update:** Executes SQL transformations for accounts, securities, and trades.
- **Audit and Validation:** Logs ETL results and verifies TPC-DI compliance.

6.9 Implemented ETL Workflow

6.9.1 Initialize and Prepare

This stage involves setting up the database environment for the benchmark. The main and temporary schemas are created, and distribution and reference settings are configured in the Citus database. To ensure a clean startup, a preliminary validation query is executed to verify the initial conditions of the setup.

6.9.2 Load Reference Data

The reference data required for the benchmark is loaded from flat files. This includes:

- Batch date and date/time dimensions.
- Various lookup tables, such as industry classifications, trade types, and tax rates.

6.9.3 Transform and Load Dimensions

In this phase, data is ingested and transformed to populate the dimensional tables:

- **Brokers:** Broker information is loaded into the `dimBroker` table.
- **FINWIRE Data:** Company, security, and financial records from FINWIRE are parsed and initially loaded into staging tables. From there, the data is inserted into dimension and fact tables, such as `dimCompany` and potentially a financial-specific table.
- **Prospects and Customer Management Data:** Data from customer prospect files and XML records from the Customer Management System are converted into CSV or staging tables. These are then used to populate or update `dimCustomer` and other related tables.

6.9.4 Transform and Update

This stage involves executing SQL transformations to process and integrate data:

- Updates are applied to prospect records.
- Accounts, securities, and cash balances are loaded into the respective tables.
- Trade data and trade histories are imported from text files. These records undergo further transformations to ensure consistency and populate dimension and fact tables, such as `dimTrade`, `FactMarketHistory`, and `FactWatches`.

6.9.5 Validation

The final phase includes validating the processed data:

- ETL results are logged into DI-messages, to track the outcome of the operations.
- A final validation query is executed to ensure compliance with TPC-DI specifications. The query checks:
 - Data consistency.
 - Correct row counts in target tables.
 - Referential integrity across the database.

6.10 Metrics

The TPC-DI test uses several metrics to evaluate system performance:

- The elapsed time of Historical Load.
- The total number of Source Data rows in the Historical Load data
- The throughput of the Historical Load.

6.10.1 Completion Timestamp

In accordance with the TPC-DI specification, the completion timestamp (*CT*) for each phase is determined by searching the `DImessages` table for a record whose `MessageType` is 'PCR' and whose `BatchID` matches

that phase. For example, to find the completion timestamp of the Historical Load (where BatchID = 1), the following SQL query can be used:

```
SELECT messagedateandtime
FROM public.dimessages
WHERE messagetype = 'PCR' and batchid = 1
```

The value returned by this query is the official *completion timestamp* for the Historical Load. The initial timestamp, denoted as CT_0 , is collected at the beginning of the load, and the final timestamp, denoted as CT_1 , is the result of the query above.

6.10.2 Elapsed Time and Row Count

The total elapsed time E_H for the Historical Load phase is computed by subtracting the start time from the completion time:

$$E_H = CT_1 - CT_0,$$

where CT_0 is in seconds at the moment the load begins, and CT_1 is in seconds at the moment the load finishes.

The total number of rows R_H for the Historical Load is obtained from the *DIGen* tool. This row count reflects the volume of source data generated and loaded during the Historical Load (commonly referred to as Batch1).

6.10.3 Throughput Calculation

Once E_H (in seconds) and R_H (the total rows) are known, the Historical Load throughput T_H is calculated as:

$$T_H = \frac{R_H}{E_H}.$$

This metric indicates how many rows are processed per second during the Historical Load phase.

6.10.4 Validation Query Usage

In addition to measuring throughput, we executed the validation query twice, as it is required by the *TPC-DI* specification:

1. **After Creating the Schema:** Immediately following the creation of tables, constraints, and any initial load steps, the validation query verifies that the environment is set up correctly.
2. **After Completing All Transform and Load Steps:** The validation query is run again when the Historical Load finishes. This confirms that all data transformations, inserts, and updates conform to the required integrity and quality standards.

6.11 Scaling

For the purposes of this project and due to the limitations caused by running the benchmark on a single computer, we used the following scale factors:

- SF 3
- SF 5
- SF 10
- SF 15

Performance Evaluation

Table 7.1: Runtime Results for Different Scale Factors (in seconds)

Task ID	Scale 3	Scale 5	Scale 10	Scale 15
set_path	0.11591	0.11667	0.13121	0.11877
create_schema	0.33625	0.22866	0.33937	0.26775
set_dist_ref_schema	0.27673	0.25162	0.27808	0.27606
create_temp_schema	0.15625	0.11544	0.11920	0.11820
load_BatchDate	0.09964	0.10089	0.09916	0.09740
load_dimDate	0.20344	0.20652	0.21124	0.19995
load_taxRate	0.11673	0.12711	0.11430	0.11592
load_statusType	0.11502	0.11830	0.11611	0.11724
load_Industry	0.11455	0.11470	0.12461	0.11941
load_tradetype	0.11558	0.12699	0.11793	0.11427
load_dimTime	0.26267	0.27142	0.26850	0.27192
load_dimBroker	0.14459	0.16134	0.23194	0.32070
Parse_Finwire	0.78730	1.53938	3.32469	5.24251
load_dimCompany	0.14726	0.15520	0.19577	0.23997
load_Financial	33.06720	54.37581	106.77001	158.66231
load_prospect	0.35934	0.54169	1.10025	1.65632
cnvrt_customermgmt	19.53778	69.27367	339.96412	1982.17617
load_customermgmt	0.12816	0.17243	0.29064	0.37299
load_dimcustomer	0.47800	0.67071	1.23149	1.72997
load_dimessages_dimcustomer	0.12668	0.12813	0.13539	0.14370
update_prospect	0.18814	0.22237	0.27714	0.43399
load_dimaccount	0.30979	0.46770	0.83547	1.37463
load_CashBalances	2.84877	4.71596	9.42129	13.95457
load_dimSecurity	1.26879	3.31630	12.88459	28.87737
load_trade_history	0.68337	1.07400	1.98154	2.99055
load_trade	1.02871	1.67368	3.20017	4.69691
load_dimtrade	10.17437	16.22741	32.26138	52.59039
load_dimessages_dimtrade	0.17179	0.18284	0.22140	0.30595
load_dailymarket	1.38159	2.43797	5.65438	7.96378
load_fact_market_history	36.46949	80.75944	150.93474	234.79097
load_dimessages_factmarkethistory	0.15398	0.21094	0.16052	0.15837
load_watchhistory	0.83706	1.28344	2.19247	3.78351
load_factwatches	1.69641	2.71239	5.40773	8.52692
load_holdings	4.34170	7.30962	15.07885	24.54059
run_validation_query_Init	0.16082	0.17187	0.15963	0.15871
run_validation_query_HistLoad	0.87181	1.44786	1.73186	2.77079

7.1 Scale 3



Figure 7.1: Gantt chart for SF 3

Total elapsed time = 00:02:37.482286 = 157.482286 seconds

Throughput = Rows / elapsed time = 4539962 / 157.482286 = 28828.4 rows/sec

7.2 Scale 5

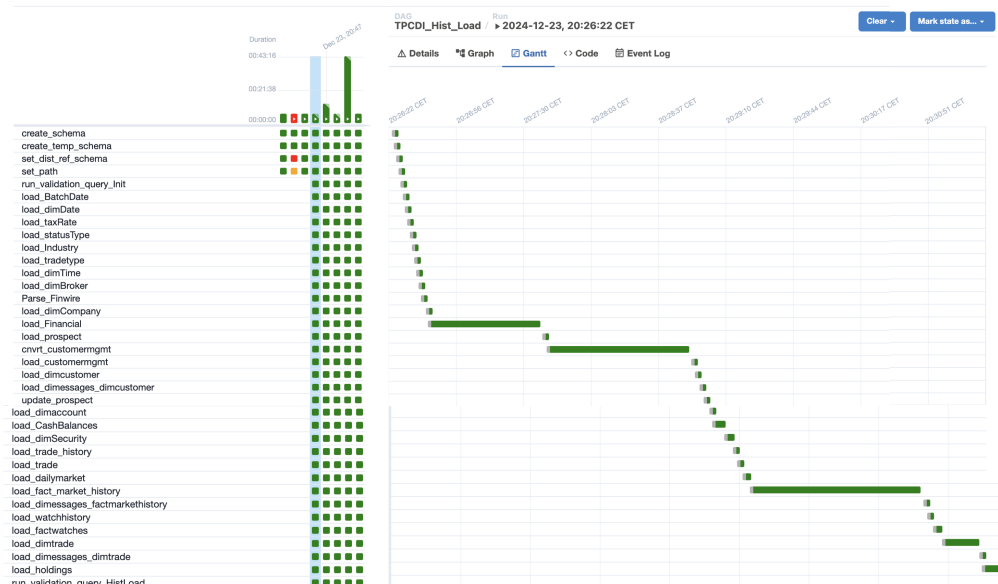


Figure 7.2: Gantt chart for SF 5

Total elapsed time = 00:04:53.10363 = 293.10363 seconds

Throughput = Rows / elapsed time = 7804509 / 293.10363 = 26627.13 rows/sec

7.3 Scale 10

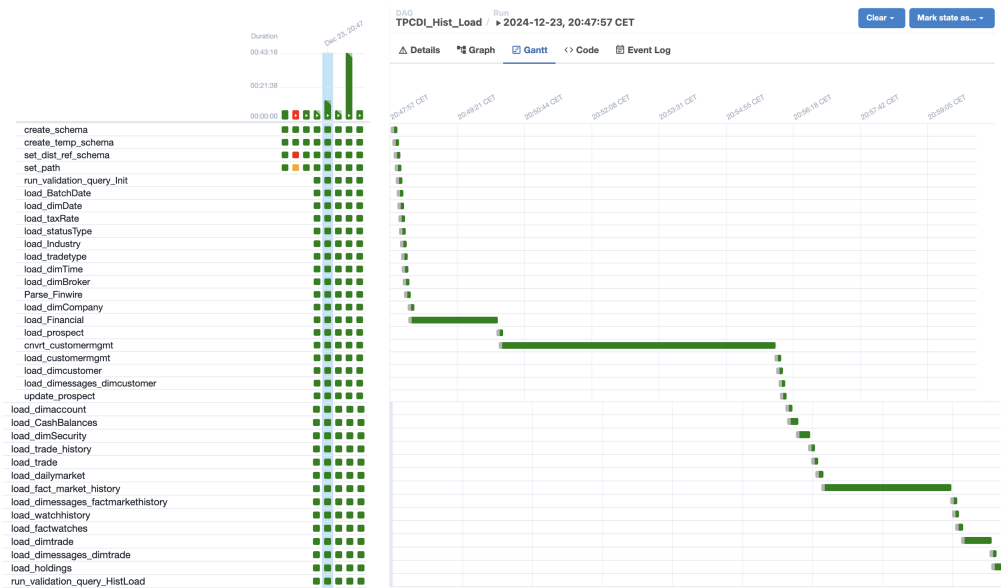


Figure 7.3: Gantt chart for SF 10

Total elapsed time = 00:12:20.939727 = 740.939727 seconds

Throughput = Rows / elapsed time = 15980433 / 740.939727 = 21567.8 rows/sec

7.4 Scale 15

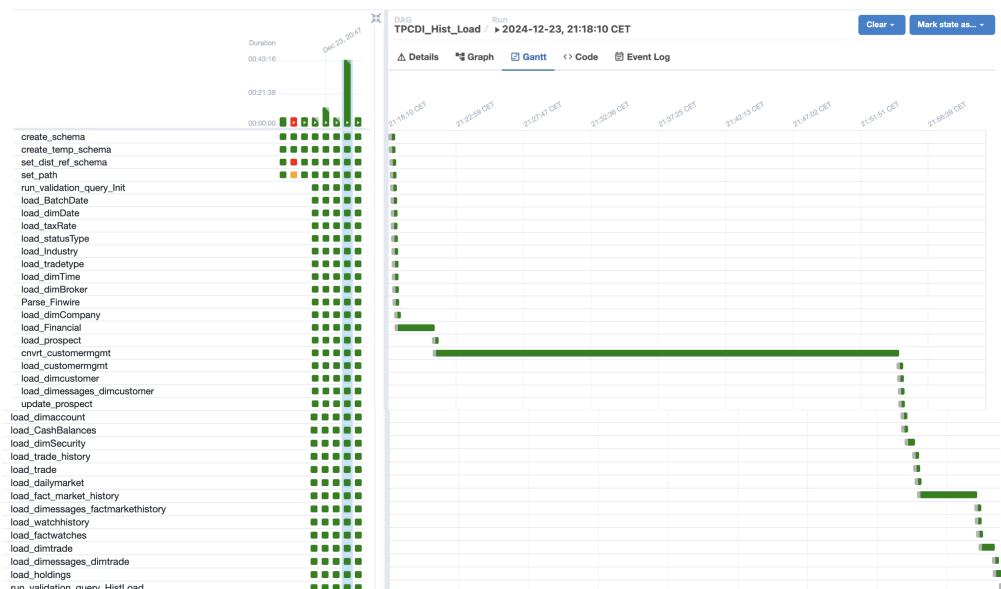


Figure 7.4: Gantt chart for SF 15

Total elapsed time = 00:43:05.002716 = 2,585.002716 seconds

Throughput = Rows / elapsed time = 24158202 / 2585.002716 = 9345.5 rows/sec

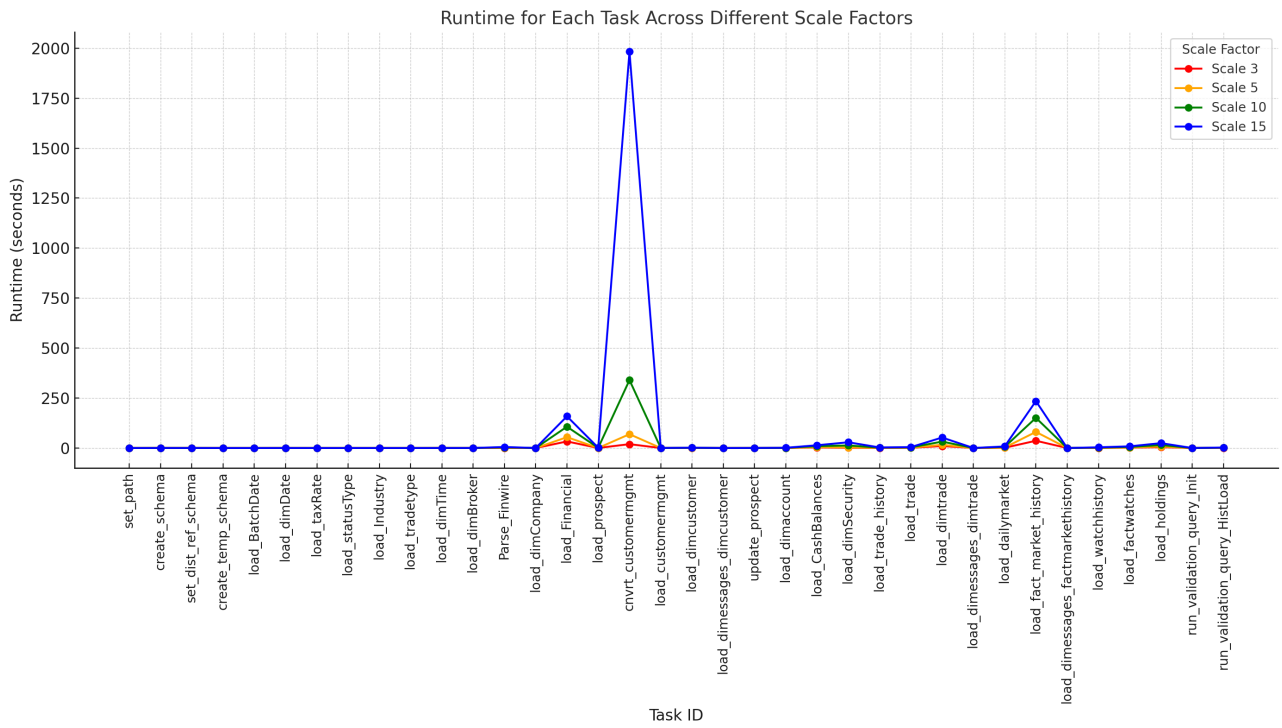


Figure 7.5: Runtime comparison for every DAG task and SF

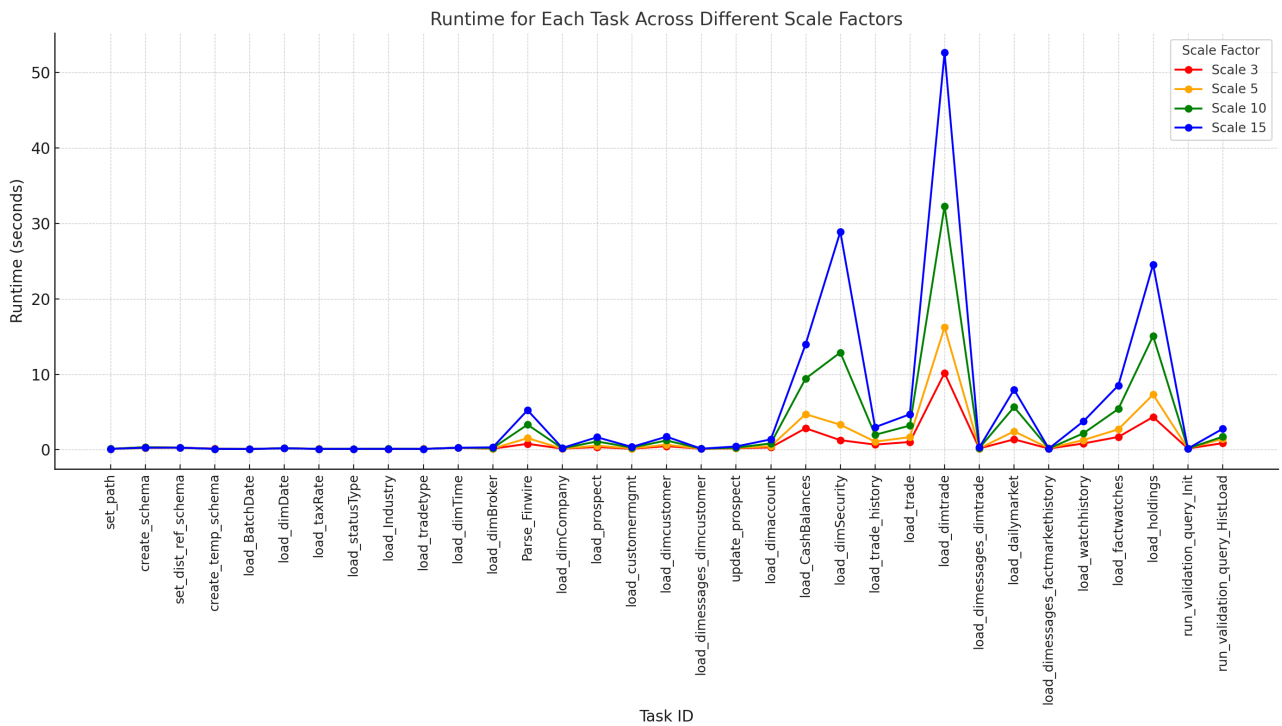


Figure 7.6: Runtime comparison for every DAG task and SF without 3 tasks with the highest runtime growth

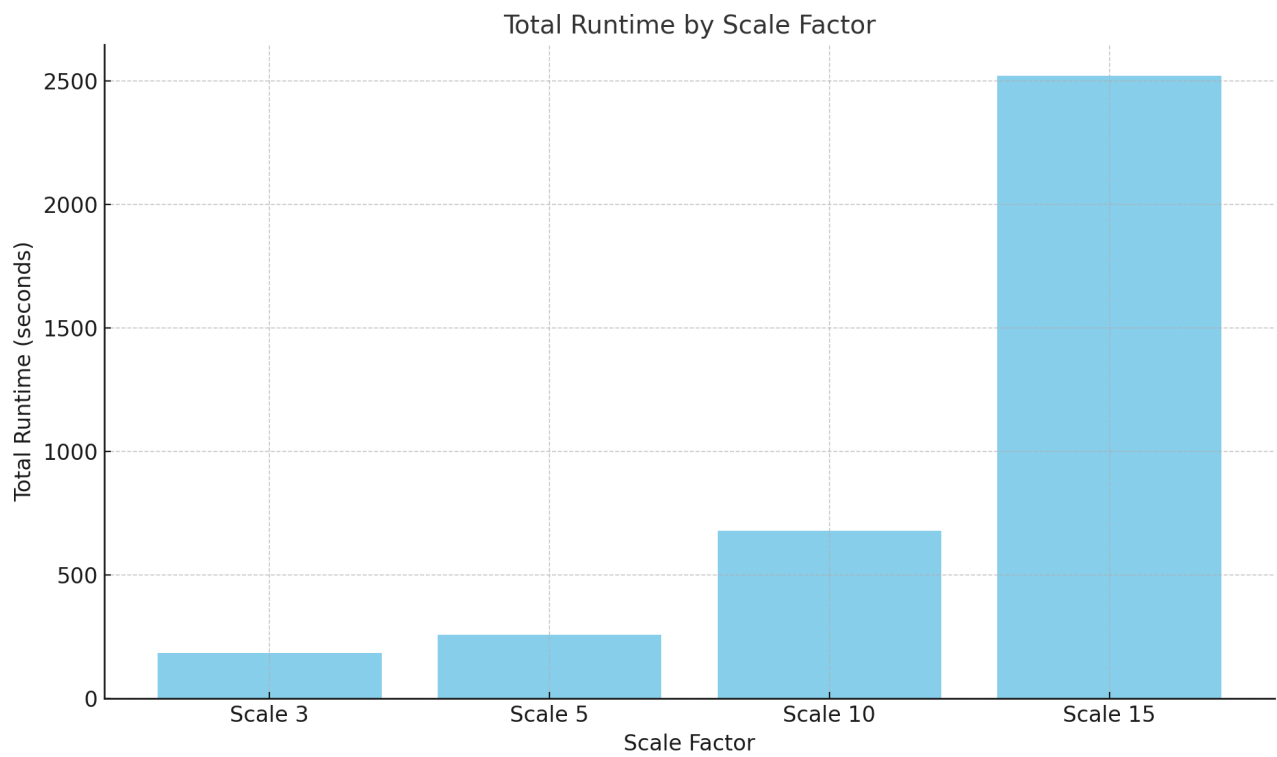


Figure 7.7: Total DAG runtime comparison

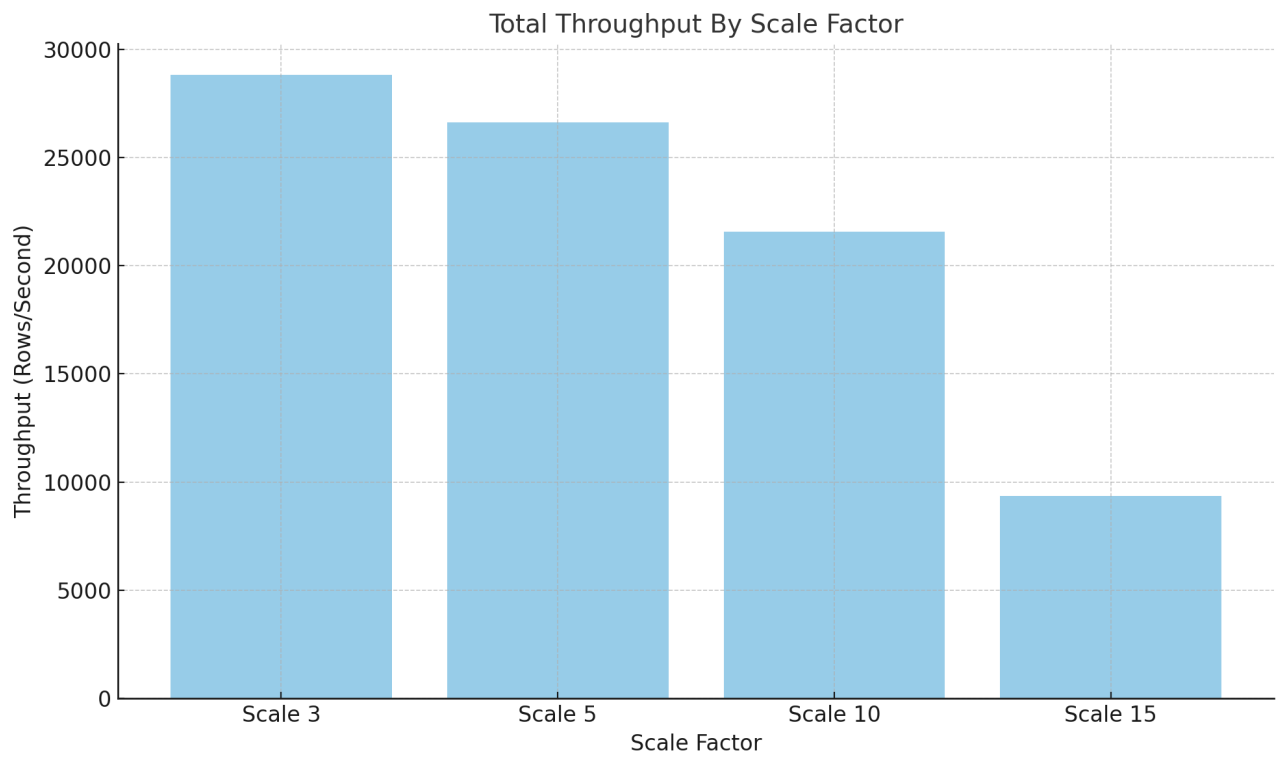


Figure 7.8: Total throughput comparison

From the results of TPC-DI benchmark, presented in the tables and graphs above, we can observe that some of the DAG tasks take noticeably longer to be completed than others. Among these outliers 3 tasks are the most distinguishable: `load_Financial`, `cnvrt_customermgmt`, and `load_fact_market_history`.

We can also notice that the throughput drops with an increasing scale.

7.5 Retrieving Task Durations for DAG Runs

Here is an instruction to replicate retrieving the task durations for every DAG:

1. Enable the experimental API:

```
enable_experimental_api = True
auth_backends = airflow.api.auth.backend.basic_auth
```

Edit the `airflow.cfg` file in the Airflow directory to include the lines above.

2. Retrieve the DAG run ID (replace with your airflow username and password):

```
curl -X GET "http://localhost:8080/api/v1/dags/TPCDI_Hist_Load/dagRuns" \
--user "admin:8r5pcYm8x4PvPq3N"
```

3. Get task details for the specific DAG run (replace with your airflow username and password):

```
curl -X GET "http://localhost:8080/api/
v1/dags/TPCDI_Hist_Load/dagRuns/
manual__2024-12-23T19:47:57.626130+00:00/taskInstances" \
--user "admin:8r5pcYm8x4PvPq3N" > scale10Tasks
```

Replace the DAG run ID with the one retrieved in the previous step. The example above retrieves task details for a scale 10 run and saves the output to a file named `scale10Tasks`.

Conclusion, Limitations, and Future Directions

8.1 Conclusion

The results presented pertain to the historical load analysis. It is evident from the data that an increase in scale corresponds to a decrease in throughput, highlighting the inverse relationship between these two metrics. The elapsed time for completing the historical load at each scale demonstrates an exponential trend, due to the increasing number of records to process with a higher scale factor. The use of the orchestration tool Airflow facilitated improved handling of the ETL process, while the adoption of Citus demonstrated how a distributed database can efficiently manage ETL operations.

8.2 Constrains in the Research

Limitation in Hardware: An important limitation in this project was the reliance on particular hardware configurations. Citus was supported only on macOS and linux due to compatibility issues which prevented some of the team members with Windows machines from reporting. While we reported load times and optimized query runtimes, our analysis did not have the complete evaluation of the benchmark's findings. Additional metrics from throughput, and data maintenance tests, would enhance the overall assessment of Citus's performance capabilities.

8.3 Future work

While we tested distribution on 2 nodes, distributing on more nodes and developing strategies to address possible distribution errors could improve TPC-DI compatibility with distributed databases, enhancing Citus's suitability for large-scale data warehousing.

1. Meikel Poess, Tilmann Rabl, Hans-Arno Jacobsen, and Brian Caufield. (2014). TPC-DI: The First Industry Benchmark for Data Integration. *Proceedings of the VLDB Endowment*, 7(13), 1367–1378. Retrieved from <https://doi.org/10.14778/2733004.2733009>.
2. Transaction Processing Performance Council (TPC). (2014). TPC Benchmark™ DI (Data Integration): Standard Specification Version 1.1.0. Retrieved from https://www.tpc.org/TPC_Documents_Current_Versions/pdf/TPC-DI_v1.1.0.pdf.
3. Rishika Gupta. (n.d.). TPC-DI-Benchmark: Data Integration with PostgreSQL & Airflow. GitHub Repository. Retrieved from <https://github.com/risg99/tpc-di-benchmark>
4. Citus Data. (2018). Distributed Execution of CTEs and Subqueries in Citus. Retrieved from <https://www.citusdata.com/blog/2018/03/09/distributed-execution-of-ctes-and-subqueries-in-citus/>
5. Citus Data. (v5.1). Installation Instructions for Single Machine on OSX. Retrieved from https://docs.citusdata.com/en/v5.1/installation/single_machine_osx.html
6. Apache Software Foundation. (2024). Airflow Documentation. Apache Airflow. Retrieved from <https://airflow.apache.org/docs/apache-airflow/stable/index.html>