TPC-DI

Data Warehouse

Ali Arous, Fabrício Ferreira, Ishaan Rachit Dwivedi, Ledia Isaj

December 2019

Contents

Abstract

Many data-driven organizations need to integrate data from multiple, distributed and heterogeneous resources for advanced data analysis. Data integration is the process of combining data from different sources into a single, unified view. Integration begins with the ingestion process and includes steps such as cleansing, ETL mapping, transformation, and loading. On the one hand, this process is complex, time-consuming, and uses most of the data warehouse project's implementation efforts, costs, and resources. On the other hand, it is crucial to implement a high performing, scalable and easy to maintain data integration system. There is a wide range of tools that can be used for the data integration process. Hence, it is necessary for an organization to compare and benchmark them when choosing a suitable one to meet its requirements. The TPC has released TPC-DI, an innovative benchmark for data integration. In this paper, we are going to follow the TPC-DI benchmark using Apache Airflow, Google Cloud Platform, and MySQL. [10] [13] [11] [12]

1 Introduction

A data warehouse is an integrated and time-varying collection of data primarily used in strategic decision making, utilizing online analytical processing (OLAP) techniques. It is essentially a database that stores integrated, often historical, and aggregated information extracted from multiple, heterogeneous, autonomous, and distributed information sources. [9] The links and relationships among the Extract-Transform-Load (ETL), data warehouse and data quality was denoted by Kimball and Caserta (2011): ETL systems extract data from the source data, enforce data quality and consistency standards, and conform data, which enable the separate sources to be used together and finally deliver data in a data warehouse with the presentation-ready format.

A more comprehensive acronym DI (data integration) replaced the ETL. The process of the ETL can be described by DI which extracts and combines data from source data with a variety of formats, transforms the data into a unified data model representation and populates it into a data repository. [14] Data integration is a vital step in building a data warehouse. It is essential to implement a high performing, scalable and easy to maintain data integration system. However, the process is complex, time-consuming, and uses most of the data warehouse project's implementation efforts, costs, and resources. Hence, it is essential for an organization to choose the best tool for this task among all the tools provided by different vendors.

TPC Benchmark^{TM} DI (TPC-DI) is designed as the first benchmark to evaluate Data Integration systems.[14] It is a performance test of tools that move and integrate data between various systems. The goal of the TPC-DI benchmark is to provide relevant, objective performance data to industry users. [7]

This report aims to represent an implementation of the TPC-DI benchmark using Apache Airflow to assist with the data integration phase and load the data warehouse in MySQL. Apache Airflow is widely used as an ETL tool, although it was built as a scheduling management tool.

Firstly, we are going to give an overview of the benchmark. Secondly, we are going to describe the source data that was generated and that will be used to feed the data warehouse. Thirdly, we are going to give an overview of the data warehouse that we aim to build. Moreover, a general description of the tool Apache Airflow is given, and the Google Cloud Platform. Furthermore, we illustrated how we implemented and executed this benchmark. The qualification test is presented as well as an important step to verify if the transformations are correct. In the end, the results of this benchmark are presented.

2 TPC-DI benchmark

TPC Benchmark^{TM} DI (TPC-DI) is a performance test of tools that move and integrate data between various systems. There are a variety of vendors that provide tools to utilize data integration. The purpose of the TPC-DI benchmark is to provide relevant, objective performance data to industry users. It provides data representing an extract from an Online Transaction Processing system, along with data from other sources that need to be transformed and loaded into a Data Warehouse. This benchmark emphasizes the characteristics of system components associated with DI environments:

- Manipulating and loading large volumes of data
- A variety of transformation types, such as error checking, surrogate key lookups, data type conversions, aggregation operations, data updates, etc.,
- Historical loading and incremental updates
- Consistency requirements ensuring that the integration process results in reliable and accurate data
- Multiple data sources with different formats,
- Multiple data tables with varied data types, attributes, and inter-table relationships.

The data model for the TPC-DI benchmark represents a retail brokerage. It contains data from OLTP systems combined with data from additional sources. The OLTP database represents a database with transactional information about securities market trading and the entities involved, i.e. customers, accounts, brokers, securities, trade details, account balances, market information,etc.

In many real-world systems, it is necessary to integrate data from different types of source systems, including different database vendors. In this benchmark, all source system data has been extracted to flat files in a staging area, before the integration process begins. Staging Area is also frequent in the real-world for allowing extracts to be performed on a different schedule from the rest of the data integration process, allowing backups of extracts that can be returned to in case of failures, and for potentially providing an audit trail.

The destination of the TPC-DI workload is a dimensional data warehouse. In a dimensional model, "dimension tables" describe the business entities of interest. In the TPC-DI benchmark, they are dates (DimDate table), times (DimTime), customers (DimCustomer), accounts (DimAccount), brokers (DimBroker), securities (DimSecurity), companies (DimCompany), and trades (DimTrade). Fact tables give measurement information describing what occurred, such as the price and volume of a transaction, or the status of something, such as the number of shares held on a certain date. In the TPC-DI benchmark, the fact tables describe holdings (FactHoldings), trades (DimTrade), cash balances (FactCashBalances), the market history (FactMarketHistory), and customer watches on securities (FactWatches). As we can see, certain tables can

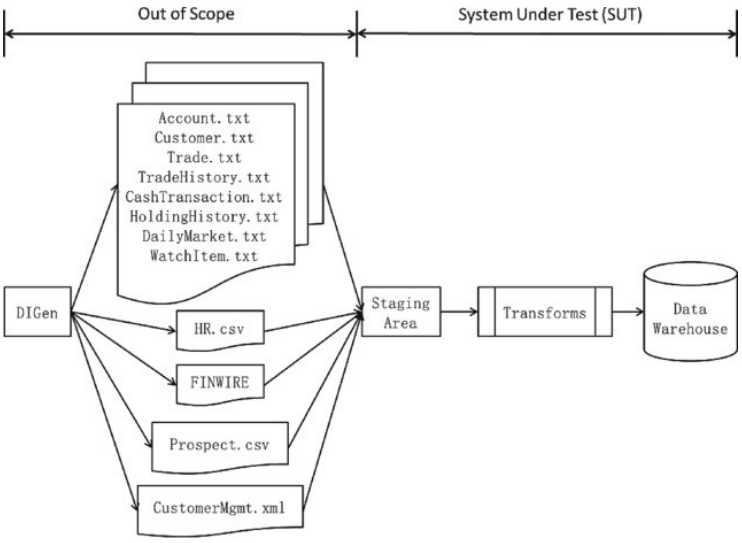

Figure 1: The data flow

serve more than one role: The DimTrade table is both a dimension table and a fact table. [7]

2.1 Phases of this benchmark

The benchmark consists of several phases which are executed in sequence.

1. **Data Generation**

   DIGen is the data generator program provided by the TPC for creating Source Data and audit information. The data may be either generated directly in the Staging Area or it may be generated in a different location and copied to the Staging Area before the Historical Load. Generating data and copying it to the Staging Area is not timed for the benchmark. The size of the dataset depends on the scaling factor you choose. We decided to use the scaling factor 5. Below you can see the code used to generate the source data.

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

2. **Data Warehouse Creation**

   Creating the data warehouse database and tables is not a data integration operation and is not timed for the benchmark.

3. **Data Integration Preparation** The data integration software may require additional preparation and configuration to perform the benchmark operations. This is not timed for the benchmark.

4. **Historical Load** The Historical Load includes different transformations. Destination tables are initially empty and being populated for the first time. Source files may have different ordering properties. Also, there are sources of data that are different from the Incremental Updates. The Historical Load naturally uses a larger set of data than an Incremental Update. After this phase is finished, the validation query will collect information that with be used for correctness in the automated audit phase. The time needed to load the data is recorded.

5. **Incremental Update phase** An Incremental Update includes different transformations comparing to the Historical Load. The input files show the changes in the table data since the last extract. However, the Prospect file is a full data set, so it is up to the DI application to determine what changes have occurred. Two Incremental Update phases are required in a TPC-DI benchmark run. By requiring more than one update, the benchmark ensures repeatability. The Validation Query collects certain information after each incremental update, that will be used to check for correctness in the automated audit phase. Each phase is required to complete in 30-60 minutes and is timed for use in the computation of the benchmark metric.

6. **Automated Audit phase** In the end, the automated audit queries the data warehouse by performing extensive tests on the resulting data and creates a simple report of the results. [7]

2.2 Source data files

In this section, we are going to describe briefly the files created during the data generation phase. The number of rows in each file depends on the scale factor chosen. The files were generated into the Staging Area. DIGen produced the data into three directories. Batch1 contains all files used for the Historical Load. Batch2 and batch3 hold the files necessary for the incremental updates.

The source schema contains 18 different source tables, from various formats. Some of them are used for historical load, some for incremental phase, some for both. [13]

System/Reference	File	Format	Historical	Incremental
OLTP	Account.txt	CDC		Y
	Customer.txt	CDC		Y
	Trade.txt	CDC	Y	Y
	TradeHistory.txt	CDC	Y	
	CashTransaction.txt	CDC	Y	Y
	HoldingHistory.txt	CDC	Y	Y
	DailyMarket.txt	CDC	Y	Y
	WatchItem.txt	CDC	Y	Y
HR DB	HR.csv	CSV	Y	
Represent List	Prospect.csv	CSV	Y	Y
FINWIRE	FINWIRE	Multi	Y	
Customer Management System	CustomerMgmt.xml	XML	Y	
Reference	Date.txt	DEL	Y	
	Time.txt	DEL	Y	
	Industry.txt	DEL	Y	
	StatusType.txt	DEL	Y	
	TaxRate.txt	DEL	Y	
	TradeType.txt	DEL	Y	

Figure 2: Source Data

- *Account.txt* is a plain-text file with variable length fields separated by a vertical bar. It holds information about the account of the customer, for instance identifiers for the customer, managing broker, owning customer, name of the account, tax status, customer status type.
- *BatchDate.txt* holds the extraction date of the data files in the Staging Area.
- *CashTransaction.txt* is a plain-text file with variable length fields separated by a vertical bar. The file consists of information for the customer transactions, such as timestamp, amount of cash and name of the transaction. It has two extra fields when it is for an incremental update, a flag that denotes insert and the database sequence number.
- *Customer.txt* is a plain-text file used during the incremental update phase with variable length fields separated by a vertical bar. It has information about the customers, such as name, date of birth, tax, and status type,address, zip code, city, state or province, country,etc.
- *CustomerMgmt.xml* is an XML formatted file, ordered in time sequence, represents data extracted from a Customer Management System. It is used only in the Historical Load to populate the DimAccount and DimCustomer tables. It has information regarding the action needed to be taken for each

customer (new, add an account, update the account or customer, close, inactive), time of the action and information about the customers.

- *DailyMarket.txt* is a plain-text file with variable length fields separated by a vertical bar. The fields of this file involve the date of the last completed trading day, a security symbol, the closing, highest and lowest price for the security on this day, and the volume of the security. The database sequence number and a flag that denotes insert are available for the incremental update part.

- *Date.txt* is a plain-text file that will be used only during historical load and holds all the information regarding the dates.

- *FINWIRE* consists of a batch of files from a financial newswire created for each quarter. They have rich information about companies, their status, location, revenue, earnings, margin, inventory, assets, liabilities, etc. They provide data for the tables DimCompany, DimSecurity and Financial in the Historical Load phase.

- *HoldingHistory.txt* is a plain text file holding information about the trade and quantity of the security before and after the modifying trade.

- *HR.csv* is a CSV file containing details about the employees of the company and the employee reporting hierarchy.

- *The Industry.txt* comprehends the industry details and its sector.

- *Prospect.csv* is a CSV file that represents data that is obtained from an external data provider. The file contains names, contact information and demographic data on potential customers, some of whom are already customers of the brokerage. This file is a full extract each day, so that needs to be taken into account for the incremental updates.

- *StatusType.txt, TaxRate.txt, Time.txt and TradeHistory.txt* are used only during Historical Load, and they have information about the status, tax rate, detailed time, and history for the trades keeping the date when it was updated, respectively.

- *Trade.txt* is a plain-text file used during the historical load and incremental update phase. It demonstrates details about trades, such as the date and time, status type, security symbol, quantity of securities traded, the requested unit price, customer account, name of the person executing the trade, unit price, fee, commission earned, amount of tax, etc.

- *TradeType.txt* is a plain-text file used during historical load that holds information about the type of the trade, description, if it is a sale, and if it is a market order.

- *WatchHistory.txt* is a plain-text file holding details like customer identifier, the symbol of security to watch, date and timestamp of the action, activating or canceling the watch. The database sequence number and a flag that indicates the rows only need to be added are present during the incremental update step.
- *Audit Data* consists of a number of files used for auditing that are generated in all directories of the Staging Area. Each file contains information about a component of the generated data. The automated audit phase begins immediately upon completion of the last incremental update phase. [7]

2.3 Data Warehouse

In this section, we are going to describe the data warehouse. There are five fact tables, FactCashBalances, FactHoldings, FactWatches, FactMarketHistory, and Prospect. DimTrade can be either a fact table or a dimensional one, depending on how you use it. There 12 dimension tables and reference ones (dimensions in the strict sense of star schema), DimCustomer, DimAccount, DimBroker, DimSecurity, DimCompany, DimDate, DimTime, TradeType, StatusType, TaxRate, Industry, and Financial. Moreover, there are two operational tables, DImessages, and Audit.

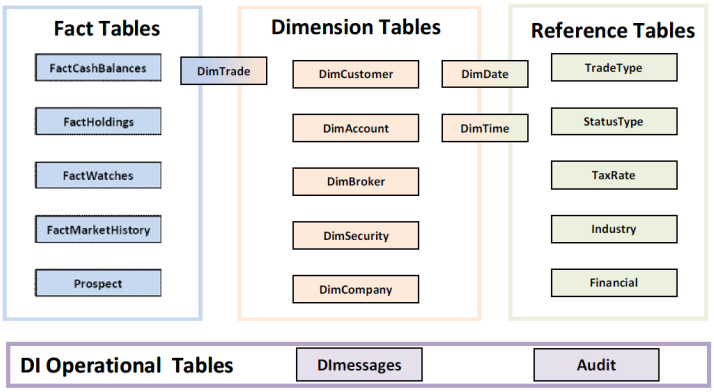

Figure 3: Overview of the Data Warehouse tables

In the figure below we can see the tables and main relationships between them.

2.3.1 DimBroker

This table is a subset of HR.csv file for the employee who are brokers and are identified with their EmployeeJobCode (34). It is structured as a history-

tracking dimension table but the provided data does not include any history of changes over time since in a real-world scenario, the rate of change would be too low for any consequential impact to the benchmarking results. As such, this table just provides a current snapshot of the HR data.

2.3.2 DimCompany

This table extracts data from the FINWIRE files with record type as 'CMP'. The records of this table may have entries where all fields except the PTS are duplicated and the changes are implemented in a history-tracking manner while the unchanged records are not recorded. This table requires the IsCurrent, EffectiveDate, and EndDate in the standard format with the EffectiveDate indicated by the PTS field. In case one of the companies SPRating is not one of the valid values for S&P long-term credit-ratings, a record needs to be inserted in the DImessages table.

2.3.3 DimCustomer

This table is obtained from the CustomerMgmt.xml data file. Customer data is stored as a sub-element of the relation Action element, where every Action will have a related Customer but may not require a modification to this table. XPath notation is used to identify specific data elements of the XML document. The transformations use the Prospect and TaxRate tables and the changes are implemented in a history-tracking manner. A record is entered into the DImessages table in case either any customer's Tier is not of the valid ones (1,2 or 3) or customer's DOB is invalid.

2.3.4 DimDate

This is a static table loaded from the Date.txt file without any transformations in the historical load and not modified again.

2.3.5 DimTime

This is a static table loaded from the Time.txt file without any transformations in the historical load and not modified again.

2.3.6 Industry

This is a static table loaded from the Industry.txt file without any transformations in the historical load and not modified again. It consists of mappings for industries and is used in other tables like DimCompany.

2.3.7 StatusType

This is a static table loaded from the StatusType.txt file without any transformations in the historical load and not modified again. It consists of mappings for status type and is used in other tables like DimCompany.

2.3.8 TaxRate

This is a static table loaded from the TaxRate.txt file without any transformations in the historical load and not modified again. It consists of mappings for status type and is used in other tables like DimCompany.

2.3.9 TradeType

This is a static table loaded from the TradeType.txt file without any transformations in the historical load and not modified again. It consists of mappings for status type and is used in other tables like DimCompany.

2.3.10 DimAccount

This table is obtained from the CustomerMgmt.xml data file. Account data is stored as a contained element to the related Account and Customer elements. XPath notation is used to identify specific data elements of the XML document and all references are relative to the context of the associated Action data element. The transformations do not need any messages to be written to DImessages table.

2.3.11 DimSecurity

This table extracts data from the FINWIRE files with record type as 'SEC' with a surrogate key of the associated company obtained for the Company dimension reference. The changes are implemented in a history-tracking manner and requires the IsCurrent, EffectiveDate, and EndDate in the standard format. It has a dependency on the DimCompany table and requires any update to the company's DimCompany records must be completed before an update to that company's DimSecurity records.

2.3.12 Financial

This table extracts data from the FINWIRE files with record type as 'FIN' with a surrogate key of the associated company obtained for the Company dimension reference. It has a dependency on the DimCompany table and requires any update to the company's DimCompany records must be completed before an update to that company's DimSecurity records.

2.3.13 Prospect

This table is obtained from the Prospect file with AgencyID being a unique identifier assigned by the agency providing the data feed and cannot be duplicated within one data batch. While processing this file, a count of the new rows added to this table is kept which on completion, is written as a 'Status' message to the DImessages table.

2.3.14 DimTrade

This table is obtained from the Trade.txt and TradeHistory.txt files that are logically joined on T_ID field. This table is updated with addition of a new row or updating an existing row based on a match between T_ID and TradeID field from this table. A record is added to the DImessages table with an 'Alert' type if a trade's Fee is not null and exceeds TradePrice * Quantity.

2.3.15 FactCashBalances

This table is obtained from CashTransaction.txt data file. All the records for a given day are aggregated and added to this table as a totalled value in a single record. Thus, only a single record is generated for every account that had changes per day.

2.3.16 FactMarketHistory

This table is obtained from the DailyMarket.txt data file. It consists dependent fields that reference other tables and a few calculated fields. A record is is added to the DImessages table with an 'Alert' type if there are no earnings found for a company.

2.3.17 FactWatches

This table is uses the WatchHistory.txt data file. It requires surrogate jeys to be obtained for the Customer, Security and Date dimension references. Securities can be either added or deleted from a watch list but there is no concept of updating a security on a watch list.

2.3.18 FactHoldings

This table is obtained using HoldingHistory.txt data file and DimTrade table. The quantity and price values of any security represent the details of the latest trade. The customer may have positive or negative Quantity as a result of any trade.

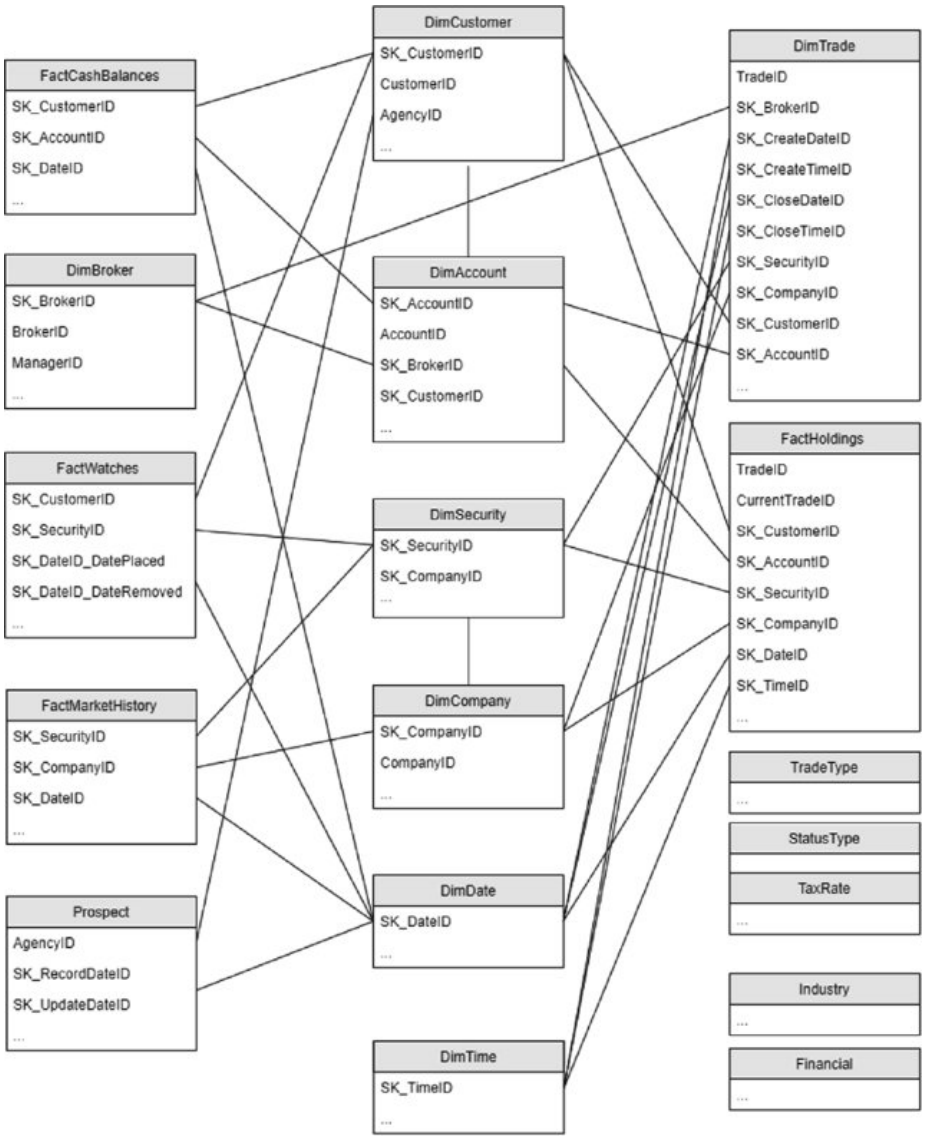

Figure 4: The main relationships among tables

3 Technologies Fundamentals

In this section, we are going to describe the technologies used during this benchmark.

3.1 Apache Airflow

Airflow was internally developed at Airbnb (an online platform for hotels and room bookings) to handle large amounts of data with reliability. It was later released as an open-source workflow management platform and is used for scheduling, monitoring, and organising complex workflows and data pipelines, in the form of DAGs (Directed Acyclic Graphs) of tasks as shown in Figure: 5. Airflow is based on Python but programs can be written in any language of the user's choice. The tasks can also be chained together and monitored via a rich user interface that includes dashboards and email notifications. The airflow scheduler executes your tasks on an array of workers while following the specified dependencies. Rich command line utilities make performing complex surgeries on DAGs a snap. The rich user interface makes it easy to visualize pipelines running in production, monitor progress, and troubleshoot issues when needed. [2]

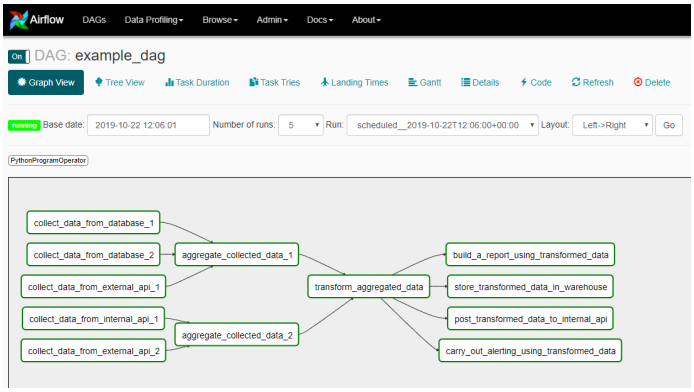

Figure 5: Example of DAGs in Airflow

The 'DAGs-as-code' paradigm was first created at Spotify, with the advent of Luigi. Luigi brings the power and goodness of software development best practices to the world of workflow management (e.g. Version Control Systems, peer-reviewed code, CI/CD). Apache Airflow is a workflow or DAG (Directed-Acyclic Graphs) orchestration system that allows users to author workflows in Python.[3] Tasks are DAGs in Airflow and consist of operators that define the task to be performed. The operators can be written in multiple languages

including Bash, Python, SQL, HTTP, among others. Airflow has had massive adoption since 2015 when it was made available as an open-source tool and is used in the tech stack of major tech companies such as Spotify, Lyft, Slack and 9GAG.

Although it was built as a scheduling management tool, it is widely used for moving large volumes of data and ETL operations, at companies including Paypal. Airflow allows to develop complex transformation pipelines with multiple dependencies with ease. Even Google uses Airflow as the underlying scheduler for it Cloud Composing service which allows data analysts and application developers to create repeatable data workflows that automate and execute data tasks across heterogeneous systems as shown in Figure6. Cloud Composer is a managed Airflow service to simplify the creation and management of workflows in the Google Cloud environment, including built-in integration with BigQuery, Dataflow, Dataproc, Datastore, Cloud Storage, Pub/Sub, and Cloud ML engine. Google says customers can orchestrate their entire GCP pipeline through Cloud Composer. Key reasons for Google selecting Airflow was that Airflow has an "active and diverse developer community", its support for Python, the support of diverse platforms with Airflow operators, support for multi-cloud setups, and lastly the command line and Web interfaces.[4]

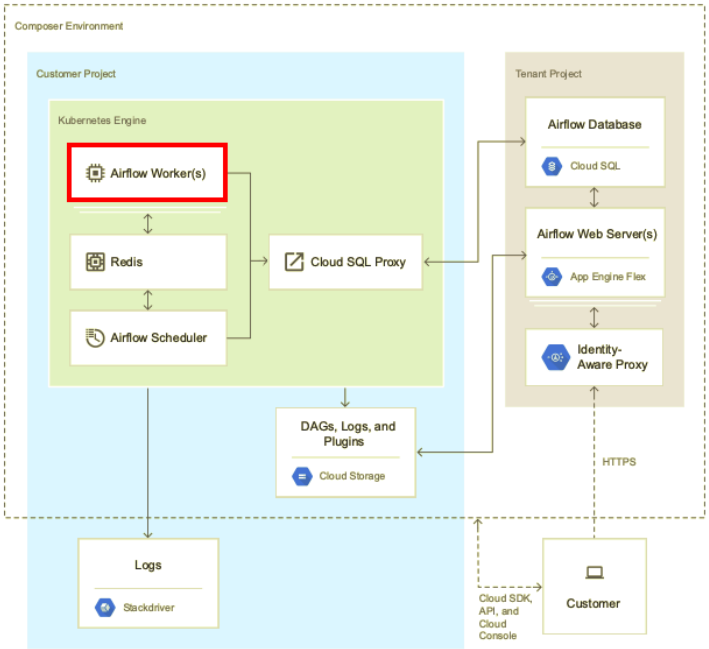

Figure 6: Google Cloud Composer Architecture

3.2 Google Cloud Platform

In order to achieve the desired level of data scalability during TPC-DI benchmark using MySQL, it was decided to rely on Google Cloud Platform.

Google Cloud Platform (GCP) is a set of physical and virtual resources that are contained in Google's data centers distributed around the globe in regions such as US, Western Europe, and East Asia. This kind of distribution leads to several benefits including redundancy in case of failure and reduced latency by locating resources closer to clients. GCP offers a wide range of services falling into these categories: computing and hosting, storage, databases, networking, big data, machine learning. [8, 1]

In this project, the service used is Google Compute Engine which enables you to create and run virtual machines on Google's infrastructure. Virtual machines boot quickly, have persistent disk storage and have consistently high performance. Compute Engine offers predefined virtual machine configurations that can vary according to one's needs and workload. Moreover, network storage can be attached to instances as persistent disk in HDD or SDD formats. This storage can be up to 64 TB in size. [5]

4 Implementation of Historical Load

The benchmark was implemented in Python using pandas library and others when necessary. We used the Airflow API also in python to make the communication. For some of the transformations we opted for using MySQL triggers since it is expected to be faster than fetching records in Python and then uploading to the database. In order to save query time for those triggers, we also created indexes on the attributes consulted by the triggers.

The Historical Load was implemented as a series of tasks of one DAG in order to build the correct dependency tree with the phases. Since one table might depend on another, we needed to make sure that one task can only run if the tasks of all tables it is dependent have finished. Figure7 illustrates this dependency. All the transformations were implemented in Python therefore there is no diagram with the transformation. However, we briefly describe in this section how the transformations were implemented and more importation can be found within the code provided.

4.1 Phase 1

The first phase comprises of all the tables that have no dependencies. It consists of a few dimension tables and the majority of reference (static) and operational tables.

4.1.1 DimBroker

This table was created by reading the *HR.csv* file and placing the information in a pandas data frame. We filtered only the records with *EmployeeJobCode = 314*, set the *EffectiveDate* as the minimum date found in the *"Date.txt"* file. The attributes *EffectiveDate*, *EndDate* and *IsCurrent* were set according to the history tracking clauses. Then the data frame was inserted into MySQL and an index with *BrokerID* was created.

DimCompany This table was created by reading all the *FINWIRE* files considering only the records of type *CMP*. After all rows were read, we then updated the *EndDate* and *IsCurrent* of "old" records of natural key value, according to the history tracking clauses. Then the data frame was inserted into MySQL. We also kept track of invalid *SPrating* values and added them to the DImessages table.

4.1.2 DimCustomer

This table was loaded using the library *lxml* to parse the customer XML file. For each action we checked its type and performed the correct operation accordingly.

- **NEW:** extracted all the information and saved as a new entry to a pandas data frame;

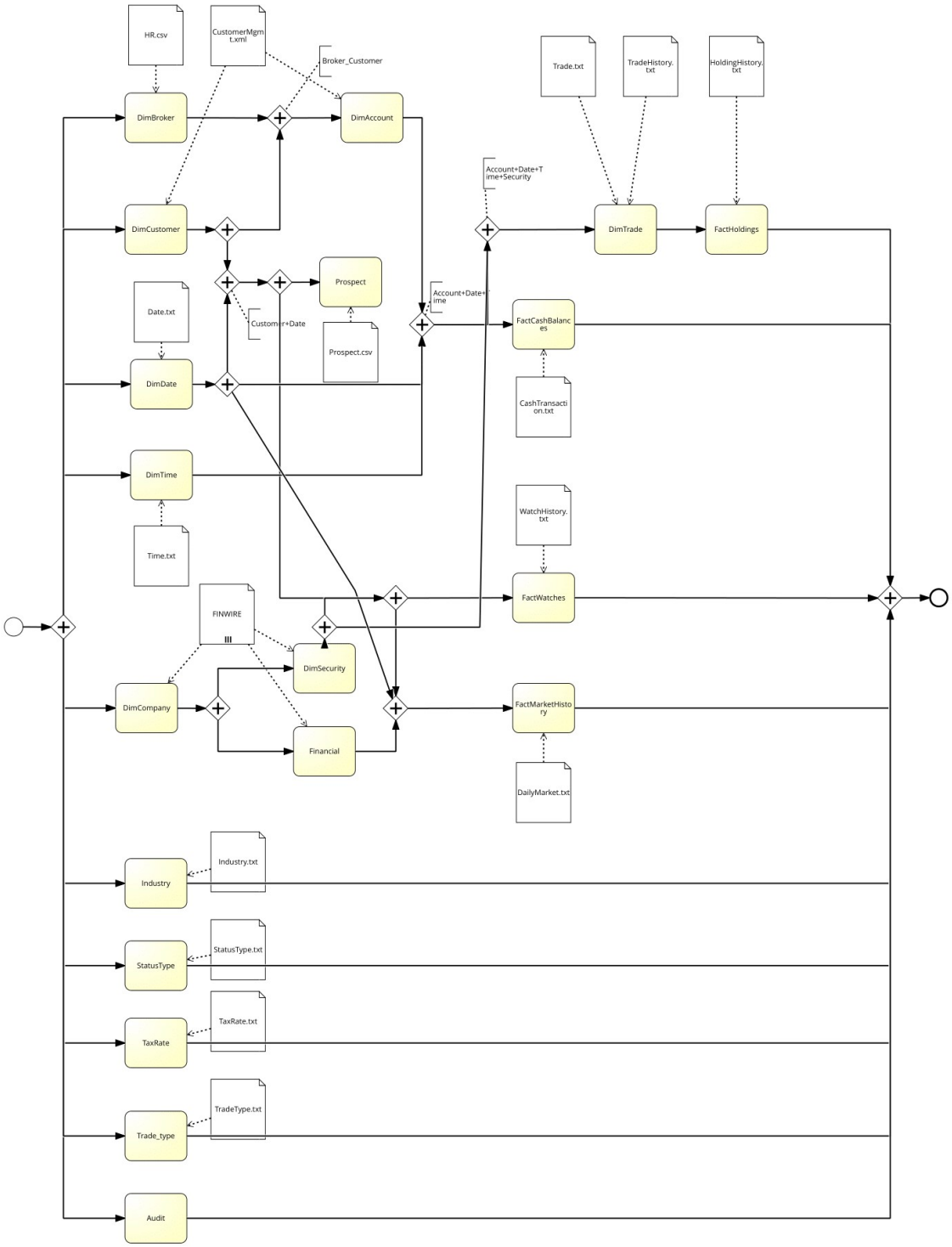

Figure 7: BPMN model of the Historical Load

- **UPDCUST:** extracted all the information and saved in a list of updates to be consumed later;
- **INACT:** extracted all the information and saved in the list of updates as an update of the status.

After all the file is processed, then we iterate over all the updates and create new entries to the data frame according to natural key and the new information. The "old" entries have their *EndDate* and *IsCurrent* updated, as stated by the history tracking clauses. Then the data frame is inserted in MySQL and an index of *CustomerID*, *EndDate* and *EffectiveDate* is created.

4.1.3 DimDate

This table was created by reading the date file and placing the information in a pandas data frame. The column *HolidayFlag* was converted from "True" and "False" to 1 and 0, respectively, since MySQL implements booleans as tiny integers. Then the data frame is inserted in MySQL and an index with *DateValue* is created.

4.1.4 DimTime

This table was created by reading the *Time.txt* file and placing the information in a pandas data frame. The columns *MarketHoursFlag* and *OfficeHoursFlag* were also converted from "True" and "False" to 1 and 0, respectively. Then the data frame is inserted in MySQL and an index with *TimeValue* is created.

4.1.5 Industry

This table was loaded by a simple read of the *Industry.txt* file, placing the information in a pandas data frame and then transferring it to MySQL. No indexes were created for this table.

4.1.6 StatusType

This table was loaded by a simple read of the *StatusType.txt* file, placing the information in a pandas data frame and then transferring it to MySQL. No indexes were created for this table.

4.1.7 TaxRate

This table was loaded by a simple read of the *TaxRate.txt* file, placing the information in a pandas data frame and then transferring it to MySQL. No indexes were created for this table.

4.1.8 TradeType

This table was loaded by a simple read of the *TradeType.txt* file, placing the information in a pandas data frame and then transferring it to MySQL. No indexes were created for this table.

4.2 Phase 2

This phase contains the tables that are only dependent on tables of phase 1. The Prospect fact table was also included here since it is a dependency of a table in phase 3.

4.2.1 DimAccount

This table was loaded by parsing the *CustomerMgmt.xml* file by using the library *lxml*. For each action we checked its type and performed the correct operation accordingly.

- **NEW and ADDACCT:** extracted the account information and saved as a new entry to a pandas data frame;
- **UPDACCT:** extracted the account information to be updated, added a new row to the data frame and updated the "old" *EndDate* and *IsCurrent* values, as stated by the history tracking clauses;
- **CLOSEACCT:** extracted the information as a new update to status, added a new row to the data frame with status INACTIVE and updated the "old" *EndDate* and *IsCurrent* values;
- **INACT:** according to the *AccountID* we iterated over all the accounts of that customer and updated their information like a CLOSEACCT action, appending a new INACTIVE row and updating the "old" history tracking information.

Then the data frame is inserted in MySQL, and the *SK_CustomerID* of each row was set using a trigger that fetches it based on *CustomerID* and their dates. An index with *AccountID*, *EndDate* and *EffectiveDate* was created.

4.2.2 DimSecurity

This table was created by reading all the *FINWIRE* files considering only the records of type *SEC*. After all rows were read, we then updated the *EndDate* and *IsCurrent* of "old" records of natural key value, according to the history tracking clauses. The value for *SK_CompanyID* was fetched on *DimCompany* by the company name or identifier. Then the data frame was inserted into MySQL and an index with *Symbol*, *EndDate* and *EffectiveDate* was created.

4.2.3 Financial

This table was created by reading all the *FINWIRE* files considering only the records of type *FIN*. Then the data frame was inserted into MySQL and the value of *SK_CompanyID* was set by a trigger that fetched on *DimCompany* by the company name or identifier, according to the value of *CoNameOrCIK*. Two extra columns were included on the table for the trigger but later they were dropped. No indexes were created for this table.

4.2.4 Prospect

This table was created by reading the *Prospect.csv* file into a pandas dataframe. The genres were converted according to the specification, if different than "M" and "F", and the *MarketingNameplate* was created. We also created an extra column, that was later dropped, on DimCustomer with the *ProspectKey*. We used this column to set the value *IsCustomer* in a trigger. We also used the trigger to set the *SK_RecordDateID* and *SK_UpdateDateID* columns, based on the column *Data* of the table. The columns Data and *ProspectKey* were later dropped from Prospect since they were only used for the trigger. A status message was inserted into DImessages with the number of prospects inserted. No indexes were created for this table.

4.3 Phase 3

This phase consists of the fact tables and one dimensional table that is dependent on a table from the previous phase.

4.3.1 DimTrade

This table was created by loading *Trade.txt* and *TradeHistory.txt* files into a pandas data frame. The values for *Status* and *Type* were extracted from *StatusType.txt* and *TradeType.txt*, respectively. We grouped the data frame in order to set the creation and close date and time of the entries, since some of them are updates. To set the columns *SK_SecurityID*, *SK_CompanyID*, *SK_AccountID*, *SK_CustomerID*, *SK_BrokerID*, *SK_CreateDateID*, *SK_CreateTimeID*, *SK_CloseDateID* and *SK_CloseTimeID* we used a trigger that fetches these values from DimSecurity, DimAccount, DimDate and DimTime. For the trigger to work, we included auxiliary columns to the table that were later dropped. We also inserted entries to DImessages in case a trade involved invalid commission or trade fee. No indexes were created for this table.

4.3.2 FactCashBalances

This table was created by loading *CashTransaction.txt* file into a pandas data frame. The values for *DayTotal* were extracted by the aggregate sum of the

grouped data frame by date and account. We grouped the data frame in order to set the creation and close date and time of the entries, since some of them are updates. To set the columns *SK_AccountID*, *SK_CustomerID*, *SK_DateID* and *Cash* we used a trigger that fetches these values from DimAccount, DimDate and previous records of FactCashBalances itself. For the trigger to work, we included auxiliary columns to the table that were later dropped, such as *CT_CA_ID*, *Date* and *DayTotal*. No indexes were created for this table.

4.3.3 FactMarketHistory

This table was created by loading *DailyMarket.txt* file into a pandas data frame. To compute the value of *PERatio*, we needed to sum the EPS of the previous 4 quarters and to do so we needed to first compute the year and number of this quarter, since it vary according to the date of the entry. We used auxiliary the columns *prev1_quarter*, *prev2_quarter*, *prev3_quarter*, *prev4_quarter*, *prev1_year*, *prev2_year* and *prev3_year* and *prev4_year* to do so with a trigger, which fetches the sum of EPS from Financial for these years and quarters. We also used other auxiliary columns and trigger to set the values for the columns *SK_SecurityID*, *SK_CompanyID*, *SK_DateID*, *SK_FiftyTwoWeekHighDate* and *SK_FiftyTwoWeekLowDate*, that were later dropped. The trigger also checks if the *PERatio* is NULL and insert an alert into DImessages registering that there were no earning for that company. No indexes were created for this table.

4.3.4 FactWatches

This table was created by loading *WatchHistory.txt* file into a pandas data frame. We first processed all the "ACTV" actions, that represent new entries and then the "CNCL" actions, that represent a remove action. For the "ACTV" we set the "Date" columns while "CNCL" we set the column "DateRemoved", that were further used by a trigger to set the values for *SK_DateID_DatePlaced* and *SK_DateID_DateRemoved*, respectively. The values for *SK_CustomerID* and *SK_SecurityID* were also set by the trigger using the auxiliary columns *CustomerID* and *Symbol*. All the auxiliary columns were then dropped and no indexes were created for this table.

4.4 Phase 4

4.4.1 FactHoldings

This table was created by loading *HoldingHistory.txt* file into a pandas data frame. We did not need to perform any extra transformation. We used trigger to set the values for the columns *SK_CustomerID*, *SK_AccountID*, *SK_SecurityID*, *SK_CompanyID*, *SK_DateID*, *SK_TimeID* and *CurrentPrice* that were obtained from DimTrade. No auxiliary columns were necessary and no indexes were created for this table.

4.5 Operational tables

4.5.1 DImessages

This table was populated with records indicating the timestamp when each batch (historical load and both incremental loads) started and finished, in order to calculate the throughput. During the load of the other tables, such as *DimCompany*, *DimCustomer*, *DimTrade*, *FactMarketHistory* and *Prospect*, records were also added to DImessages.

4.5.2 Audit

This table was populated with information of all audit files from the staging area. For each file generated by DIGen, one audit file is also generated with information about that file. This table will be further used for the qualification step.

5 Implementation of Incremental Updates

The Incremental Updates include different transformations than the Historical Load as the input files from the OLTP database are modeled as CDC extracts, which show the changes in the table data since the last extract. Files used as input to an Incremental Update are CDC extracts, and as such they contain additional "CDC_FLAG" and "CDC_DSN" columns at the beginning of each row. The CDC_FLAG is a single character I, U or D that tells whether the row has been inserted, updated or deleted since the last change.

Additionally, While destination tables in Historical Load are initially empty, they get loaded with data before being moved to the Incremental Load phase. This fact requires different processing for auxiliary columns, as well as for the load itself. For the auxiliary columns, which are additional columns added to the table for the sake of carrying some temporary data used during the load; these are directly added to the initial table schema in "CREATE TABLE" statement in the Historical Load setup step, and they are dropped subsequently by the script at the end of the Load, while in the Incremental Load, we need to alter the table in order to add any auxiliary columns required. This should happen before starting the script's logic. For the load, as in the Historical we are only filling the tables, we can directly bulk load the processed file from memory to the database on disk, while in the Incremental, sometimes we need to query the table in order to know whether this entry already existed from a previous load (which could be the Historical one, or a previous iteration of the Incremental). If so, we have to issue an update statement instead of the insertion.

5.1 Incremental Load

5.1.1 DimAccount

What changes in this table form the Historical Load, is that When *CDC_FLAG* is "I", a new DimAccount record is inserted. When CDC_FLAG is "U" an exiting record is updated in a history-tracking manner. Also when linking it to DimCustomer, we depend on IsCurrent = 1 instead of the Effective and End dates.

5.1.2 Prospect

This table is updated using the *Prospect* file. As the Prospect file is a full data set (not CDC), it would be the responsibility of the ETL tool to determine what changes have occured during the Incremental Load, this leads to one essential change in the script logic from the Historical Load. In out Historical Load implementation, we created a trigger before INSERT which helped in getting the SK_DateID from DimDate, a value subsequently assigned to SK_RecordDateID and SK_UpdateDateID. In addition the trigger assigned the field IsCustomer according to whether there existed an ACTIVE customer linked to the prospect

in the DimCustomer. However, in Incremental Load, we needed to keep this trigger, and add another one for updates. The updates are decided on from the scrip code. We first query the table for an existing record with the AgencyID. If we found one, we compare it with the row from the csv file to see if any of its fields is different. If they were the same, we discard the row from the file. Otherwise to update record in the table with its content. In case no record already existed for the row in hand, we added a new record to the table. In both Updating and Inserting Triggers we updated the SK_UpdateDateID, as the only case it should not be update we managed it in the script code.

5.1.3 FactCashBalances

This table is updated using the *CashTransaction.txt* data file. There are two differences from Historical Load, the first is matching SK_CustomerID, and SK_AccountID from DimAccount by using IsCurrent = 1 instead of Effective and End dates. While the second is the inability to compute the previous Cash depending on the file itself, but rather we need to query the table in order to get the last Cash before the update.

5.1.4 FactMarketHistory

This table is updated using the *DailyMarket.txt* data file. For the Incremental Load, it is not enough for the transformation to compute the highest and lowest values in the last 52 weeks (365 days) throughout the input file, as its values represent only the updates for some period of time which does not necessarily span a whole year. Thus, a more complex processing should be done by computing the highest and lowest value for the last 52 weeks throughout the input file, and also querying the FactMarketHistory table for its highest and lowest values in a year from the date of the given record. If the values concluded from the file are more extreme than the ones returned from the table, we consider them as the highest and lowest value for the record. Otherwise, we update them with the values read from the table.

5.1.5 FactHoldings

This table is updated using the *HoldingHistory.txt* data file and the *DimTrade* table. There is almost no change in the transformation from the Historical Load except for ignoring the CDC_FLAG,CDC_DSN fields from HoldingHistory.txt in Batch2 files as those fields have no effect on the course of the load.

6 Benchmark execution

6.1 Running benchmark on GCP

Since this benchmark focuses on measuring the performance on Big Data systems, it requires a robust infrastructure to run it in a viable time. For this reason, it was decided that instead of running locally it would be best to turn to Infrastructure as a Service technology and run it on the cloud. Therefore a virtual machine was rented on Compute Engine, a product from Google Cloud Platform (GCP). This service makes it possible to use robust machines to perform expensive operations that would not be possible to execute locally in a very quickly and straightforward way.

6.1.1 Creating the instance

Figure 8 illustrates the interface of the service, where you need to have an active project (in this case it is called "bdma", represented by the first red rectangle) and then it is possible to create new instances (rectangle number 2). The figure also pictures the list of all the instances created and their status (rectangle number 3), if it is running or stopped, for example.

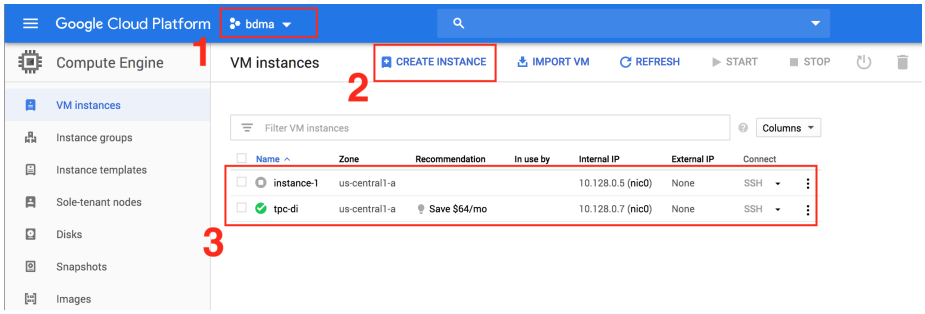

Figure 8: Interface of GCP's Compute Engine

When creating a new instance, there are different options to choose based on one's needs such as number of CPUs, amount of memory, disk size and operating system. Region can also be important depending on the context, for example, one could opt for including a HTTP server in an instance in Hong Kong to decrease the request latency from their users in Asia. In this case, the standards region (us-central1) were chosen since it is the cheapest. Regarding the specs, it was decided to opt for an instance with 30 GB of memory, 8 CPUs, Ubuntu 18 as operating system on a 100 GB disk. The reason is that a large memory was needed to process the files, a disk more than capable of generating and storing the data, a stable Linux distribution to easily run the experiment's script and

multiple CPUs for eventual parallelism. Figure 9 illustrates the creation of a new instance with all the specs of the one used to run this benchmark. The instance used was named "tpc-di", shown by Figure 8.

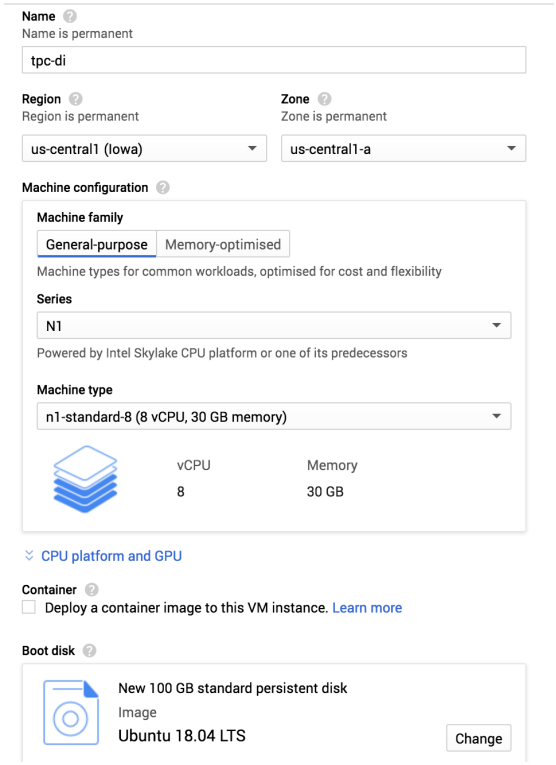

Figure 9: Options when creating a new instance on Compute Engine

6.1.2 Accessing instance and setting it up

There are many ways to connect to an instance from Compute Engine [6] and it is usually via SSH. For this work we used the Cloud SDK, which is a collections of tools to assist operations with all the products of GCP. To connect to the instance first one should start it on the Compute Engine web interface and then execute the command illustrated on Code 1 that connects via SSH to the instance named "*tpc-di*" of the project ID "*bdma*" on GCP.

Code 1: Connecting via SSH to instance "tpc-di" on GCP

```
gcloud beta compute --project "bdma" ssh --zone "us-central1-a" "tpc-di"
```

To run experiment it was necessary to prepare the instance by installing MySQL 8.0, Apache Airflow 1.10.6, JAVA SE, setting it up the connections between Airflow and the database, download the files from the project and generating the input files. The DIGen and the benchmark implementation are available on a GitHub repository to make it easier to setup the instance and consequently simpler to run the benchmark. This setup can be found on the file **setup.sh** provided in the source code that accompanies this report.

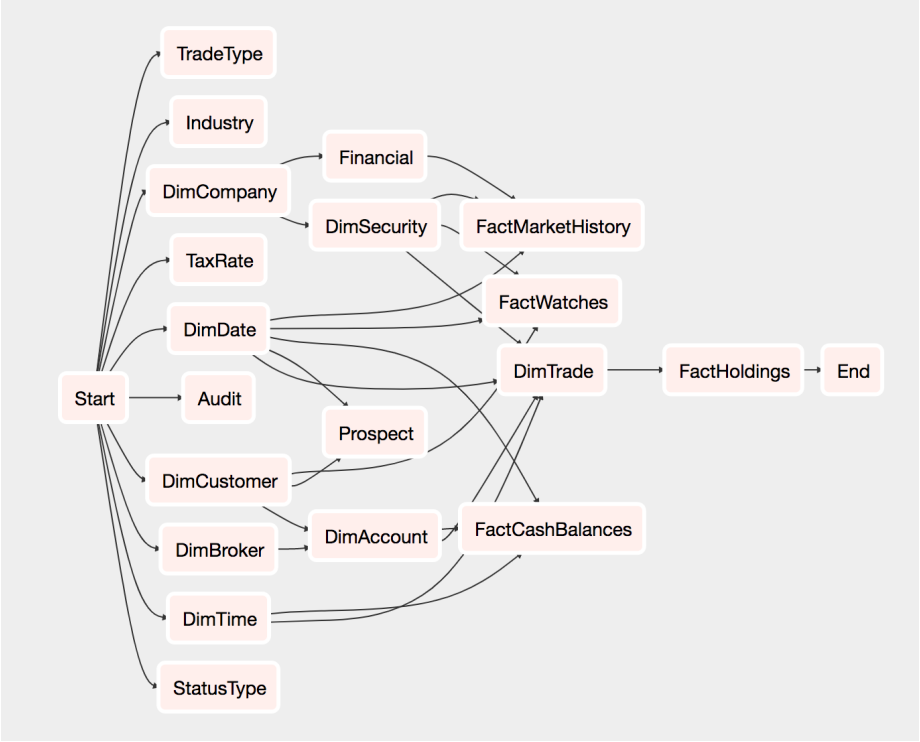

Figure 10: Pipeline graph generated by Airflow for Historical Load

6.1.3 Running the test

Now with a running instance that has been set up it is time to run the benchmark. To do so you can trigger the DAG using the command illustrated by

Code 2 or you can run each task individually. We chose the second one since Airflow does not execute sequential tasks instantaneously. The primary goal of the tool is to run scheduled DAGs that does not to be instantaneous. In order to make the correct measurements, and not rely only on the scheduler, we decided to run each task sequentially via a script, which automatise and make it easier to execute the pipeline (create database, record start time, execute all phases, record end time), on the file **historical_run.sh**. We used both the values on DImessages and the log of execution for the metrics, since we wanted to see the execution time of each table and possibly identify bottlenecks and opportunities of code improvement.

Code 2: Running historical load

```
airflow trigger_dag historical_load
```

Although we ran all the tasks sequentially, we still ran the DAG using the standard method for a smaller scale factor (3) in order to see the pipeline working automatically. The Figure10 shows the graph built by the tool with all the dependencies pictured as directed edges, while Figure11 illustrate the graph.

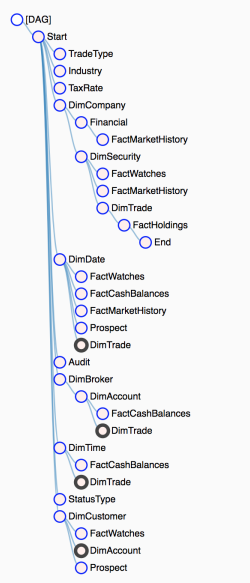

Figure 11: Pipeline tree generated by Airflow for Historical Load

The incremental DAGs and tasks were implemented the same way as during the Historical Load. Since some of the tables did not take part in the incre-

mental, some tasks could be performed in a earlier phase, which can save time. Although we created the scripts and ran the DAG, it did not finish in time and we could not attest the execution time. We believe that this stage was taking a long time due to the many look ups in Python.

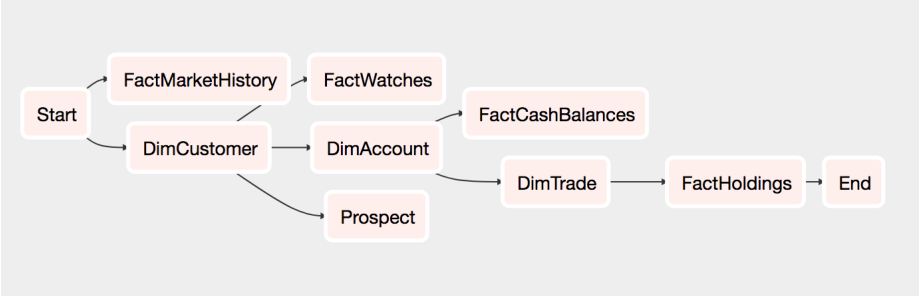

Figure 12: Pipeline graph generated by Airflow for Incremental Load

7 Qualification

The qualification test is an important step of every benchmark since it verifies if the transformations are correct, and this guarantees that the performance test is correct. Since results of this benchmark performed by different entities can be compared to each other, it is important to have a certain standardization in order to make fare comparisons. The qualification test makes sure that a minimum methodology compliance was achieved by analysing the state of the Date Warehouse after the transformations. The TPC-DI specification mentions a comparison tool downloadable from the TPC website, that would also bring information about how to use the tool to perform the qualification method. We, however, did not find any such tool or any information about it.

Nevertheless, We ran the visibility and the validation queries aiming to obtaining results that would qualify our implementation of the benchmark. For both scripts we had to adapt them for the MySQL dialect, using an online converter, after removing the comments. For the audit we still had problems since we did not run all the incremental tasks. We believe that it would be more useful to have different queries in separate files so we could spot the errors more easily and make the necessary corrections.

8 Results

The tables take varied amount of time for the historical load with the fastest ones such as the static tables Industry and StatusType take nearly 0s while the larger fact tables such as FactCashBalances and FactMarketHistory take from 5 to 7 hours to load. The load times of all tables have been shown in an ascending order on a log scale in Figure:13. The values very close to zero were truncated to 0 for lack for time granularity on this graph.

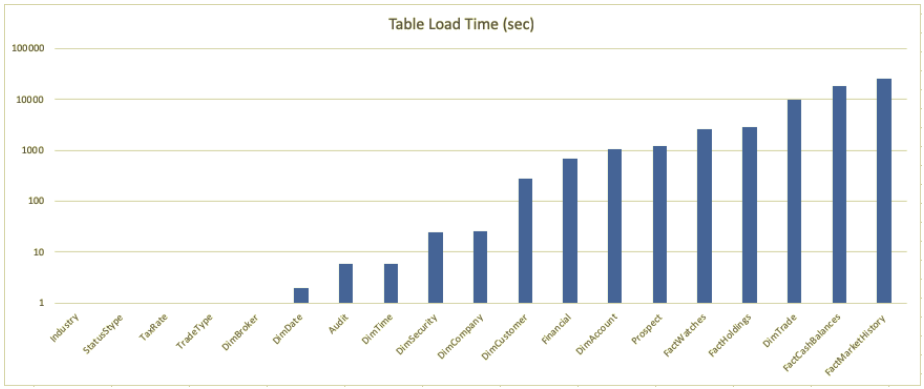

Figure 13: Table load time (in sec) on a logarithmic scale

The throughput of the above tables have been shown in Figure:14 on a log scale in the same order as the their load times. The throughput ranges from very small to around 600 rows/second, averaging out at 77 rows/second across all tables. The fastest throughput is observed for the smaller static tables such as DimDate and DimTime. The tables with the slowest throughput are FactWatches and Financial.

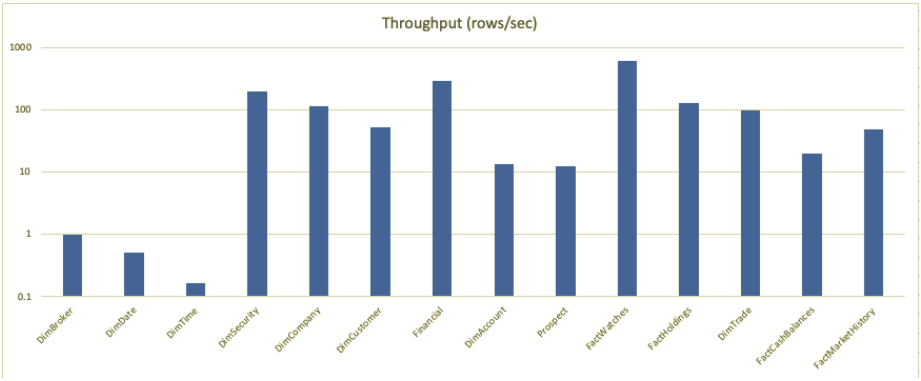

Figure 14: Throughput (in rows/sec) on a logarithmic scale

9 Conclusion

Data integration is an essential phase of building a data warehouse. For companies, it is crucial to implement high performing, scalable and easy to maintain data integration systems. Therefore, they must compare and benchmark different available tools, provided by different vendors, in order to determine the one that fits best their requirements.

TPC Benchmark^{TM} DI (TPC-DI) is a performance test of tools that move and integrate data between various systems. The goal of the TPC-DI benchmark is to provide relevant, objective performance data to industry users.

Apache Airflow is widely used for ETL even though it was built as a scheduling management tool. It is a platform to author, schedule, and monitor workflows or data pipelines. One of the key advantages of Apache Airflow compared to most of the tools on the market, is that it can backfill historical data. Moreover, compared to simply scripting and using crob jobs, we can monitor the success or failure status, retry in case of failure, benefit from scalability since there is no centralized scheduler, and deploy new changes constantly.

In the end, the results obtained did not seem very pleasing if we compare to other data integration softwares. Apache is a useful tool but by using Python to process the files and insert data into MySQL, there was a critical bottleneck. Although we used pandas parallel operations, that was not sufficient. In addition, MySQL has proven to perform inadequately through our implementation of TPC-DS. We therefore support the idea of choosing a better suitable software for ETL for such large data volume.

References

[1] *About the GCP services*. https://cloud.google.com/docs/overview/cloud-platform-services.

[2] *Apache Airflow Documentation*. https://airflow.apache.org/docs/1.10.5/.

[3] *Apache Airflow to Power Google's New Workflow Service*. https://medium.com/@r39132/apache-airflow-grows-up-c820ee8a8324.

[4] *Apache Airflow to Power Google's New Workflow Service*. https://www.datanami.com/2018/05/01/apach airflow-to-power-googles-new-workflow-service/.

[5] *Compute Engine*. https://cloud.google.com/compute/.

[6] *Connecting to instances — Compute Engine Documentation — Google Cloud*. URL: `https://cloud.google.com/compute/docs/instances/connecting-to-instance`.

[7] Transaction Processing Performance Council. *TPC Benchmark^{TM} DI Standard Specification Version 1.1.0*. 2014.

[8] *Google Cloud Platform Overview*. https://cloud.google.com/docs/overview/.

[9] Bodo Hüsemann, Jens Lechtenbörger, and Gottfried Vossen. *Conceptual data warehouse design*. Universität Münster. Angewandte Mathematik und Informatik, 2000.

[10] Meikel Poess et al. "TPC-DI: the first industry benchmark for data integration". In: *Proceedings of the VLDB Endowment* 7.13 (2014), pp. 1367–1378.

[11] Shaker H Ali El-Sappagh, Abdeltawab M Ahmed Hendawi, and Ali Hamed El Bastawissy. "A proposed model for data warehouse ETL processes". In: *Journal of King Saud University-Computer and Information Sciences* 23.2 (2011), pp. 91–104.

[12] *What is Data Integration?* https://www.talend.com/resources/what-is-data-integration/.

[13] Qishan Yang, Mouzhi Ge, and Markus Helfert. “Data Quality Problems in TPC-DI Based Data Integration Processes”. In: *International Conference on Enterprise Information Systems*. Springer. 2017, pp. 57–73.

[14] Qishan Yang, Mouzhi Ge, and Markus Helfert. “Guidelines of data quality issues for data integration in the context of the TPC-DI benchmark”. In: (2017).