
Data Warehouse Schema Design and Implementation for Healthcare Data Analytics

Royal Thomas

Student ID: 16326926

A thesis submitted in part fulfilment of the degree of

BSc. (Hons.) in Computer Science

Supervisor: Prof. Tahar Kechadi



UCD School of Computer Science
University College Dublin

Table of Contents

1	Project Specification	5
2	Introduction	6
2.1	Aim	6
2.2	Report Structure	6
3	Background Research	7
3.1	Data Warehouse	7
3.2	Clinical Data Warehouse	7
3.3	Data Mining	8
3.4	Options for Implementation	9
3.5	Design Methodologies	12
3.6	Different Types of Architecture	16
4	Related Work and Ideas	18
4.1	Stride	18
4.2	Radbank	18
4.3	Google, UCC (Chicago), and UCSF Data Warehouse	18
4.4	SMEYEDAT	19
4.5	i2b2 Star Schema	19
5	Outline of Approach	20
6	Project Work Plan	21
6.1	Packages	21
6.2	Evaluation	22
7	Data and Context	23
8	Schema Design	26
8.1	Fact Constellation	26
8.2	EAV-Star Schema	26
8.3	Proposed Schema	28

8.4	Optimizing Schema for BigQuery	29
8.5	Architecture	32
9	Implementation	33
10	Performance Analysis and Evaluation	35
11	Conclusion and Future Work	38
11.1	Future Work	39
12	Appendix	42

Abstract

In recent years, there has been a surge in the digitization of hospital records. Raw, structured, and unstructured data is constantly collected through various sources such as information systems within the hospital which can provide temporal information such as physiological measurements(heart rate, blood pressure, etc.) along with progress notes on the patients. This data is then often stored into proprietary databases with various structures making it difficult to analyse as a single unit. Above it, healthcare datasets grow to be fairly large in size, this makes it a potential Big Data application to allow for efficient analysis allowing for early intervention. Understanding the data is crucial to leverage the resources and deliver high-quality care to patients efficiently.

In this project, the aim is to propose an efficient schema for the healthcare dataset that can be implemented within a data warehouse system followed by implementation. An extensible clinical data schema is proposed in this project which is implemented within a cloud data warehouse (BigQuery). Following this, the implementation was evaluated based on complexity of the SQL queries and the difference in throughput when compared to the implementation within an OLTP system. This work also highlights the differences between alternate approaches and considerations when choosing where to implement such a schema.

Modifications from Interim Report

Interim Report Sections:

1. Abstract
2. Project Specification
3. Introduction
4. Related Works and Ideas
5. Data Considerations
6. Outline of Approach
7. Project Work Plan
8. Summary and Conclusions

Additional Final Report Sections:

1. Data & Context
2. Schema Design
3. Implementation
4. Performance Analysis and Evaluation

Acknowledgements

The author would like to express my gratitude to my supervisor, Prof. Tahar Kechadi, for his support and guidance during the process of this project. I am also very grateful for the constructive feedback I have had from the moderators, Guenole Silvestre and Tony Veale that has helped me direct the project to the right track.

This project also gained a lot of benefit from the opinions and help from my colleagues in the School of Computer Science at UCD. I would like to wish everyone a wonderful future.

Chapter 1: Project Specification

The following are the core and advanced requirements as described by the section on the project page:

- Core

1. Conduct a literature review on data warehouse design and models.
2. Explore the healthcare dataset and derive Entity Relationship Diagram (ERD) of the dataset.
3. Propose an effective data warehouse schema in healthcare.
4. Propose a reference architecture of such a data warehouse.

- Advanced

1. Implement the proposed data warehouse schema using existing technologies such as Hive.
2. Transfer the healthcare dataset from SQL (on PostgreSQL) to the data warehouse.

- Dataset

1. MIMIC-III dataset, link: <https://mimic.physionet.org/>

Chapter 2: Introduction

A clinical data warehouse can be utilized to store and retrieve data for a plethora of reasons and it is currently well implemented in many industries. Within the healthcare industry, data warehouses can be used to store data related to patient health outcomes, administration, and management.

The domain of the project is the healthcare data analysis sector and the application of the project would be to figure out an effective database schema for building a data warehouse to aggregate electronic health records and other data from various systems. As opposed to many other industries, there is a difficulty with medical data due to the heterogeneous nature of the data to create a system that can unify data from various sources. [1]. This has severely impacted efforts to gain valuable information from the vast amount of data that is available today in the healthcare sector.

Without a proper data warehouse being implemented, the data from the various sources would then have to be queried on their own and the results combined leading to redundancies all through the process taking way more time than it should. Data analysts are not cheap and having a single source of information that can be queried in a fast and effective manner can help greatly improve patient outcomes through better understanding of the data.

2.1 Aim

This project aims to develop an efficient data warehouse based on the MIMIC-III dataset, which is a critical care dataset that has information related to patient care such as diagnoses and procedures. An extensible schema is proposed and implemented inside BigQuery, a cloud data warehousing tool. The goal is to be able to query the dataset with faster response times requiring simpler queries.

2.2 Report Structure

This report is structured in a top-down fashion, the next chapter describes the background research that was shown to be helpful during the project. This is followed by chapters describing the outline of approach and the work plan. This is followed by a brief introduction to the dataset followed by sections on how the data warehouse was designed and implemented. The project concludes with an evaluation of the implementation and with proposals of future work.

Chapter 3: Background Research

Due to the nature of the project being as to proposing an efficient schema, much background research is needed to ensure that state-of-the-art techniques and known practises are used when fulfilling the tasks of this project. For this reason, literature review was conducted to better understand what state-of-the-art means in the context of data warehousing today.

3.1 Data Warehouse

Bill Inmon defines DW as "a subject-oriented, integrated, nonvolatile, and time-variant collection of data" aimed at supporting users to make better and faster decisions [2]. This technology is now being used successfully in many industries;

1. Retail: Distribution, Pricing Policy, Customer Trend Analysis, etc.
2. Clinical: Research health outcomes, help with administration decisions, etc.
3. Public Sector: Store and analyse public data such as tax records, etc.
4. Banking: Lending decisions, Customer Analysis to tailor products for them, etc.

This is just a small list of examples, the use cases are endless. Data warehouses contain large amounts of information which are integrated from a plethora of different sources into a single store. This is kept as a separate entity as the operational database of the organisation.

Data warehouses are usually modeled based on multidimensional schemas such as Star or Snowflake to support OLAP queries, these are described in sections 3.4.1 and 3.4.2.

Kekri et al. [3] talked about how a data warehouse ensures consistency in terms of the data. It allows the combining of data from various legacy systems without having to necessarily set aside those systems and then combining it to form a coherent set. This has the potential to vastly improve data quality and allow end-users to access the data without having to worry about the legacy systems. Kekri et al. hence concludes that the principle of a data warehouse is to gather data from a wide range of sources, consolidate it into a single data repository.

3.2 Clinical Data Warehouse

Based on the huge success of information extraction from data warehouses in other areas, hospitals and other research groups began to invest in a data warehousing model suitable for clinical data.

Clinical Data Warehouse (CDW) are used in institutions for various purposes such as research, management, clinical practice and even administration[4]. Most of these areas such as management and administration are general purpose and these are called Enterprise Data Warehouses. We also have hospital-specific data warehouses that concentrate more on the research and patient outcome side of things [5] called Clinical Research Data Warehouses.

One issue with clinical data warehouses is that because clinical data is highly confidential, it is very common in the domain for data warehouses to be stand-alone and not share their data with other organisations. This is a huge problem for smaller institutions that do not produce a lot of data.

Table 3.1: Inmon - Differences Between Types of Warehouses

Domain	Healthcare	Other Domains
Transactions	Relatively Unique	Relatively Repetitive
Vocabulary	Not Common - Requires Normalization	Usually Normalized
Data Type	Mixed (Text, Written, Scans)	Numbers

Inmon describes the difference between data warehouses built for hospitals and other domains very well. He says that "The information needs of medicine and healthcare were fundamentally different than those of other institutions" [6]. An overview of these differences are shown in table 3.1 above. To begin with, Inmon argues that the transactions in healthcare are relatively unique as compared to those in other domains. Inmon used the example of airlines to depict how the information gathered in other domains is relatively repeated. For example, the process of booking and taking a flight is relatively similar among everyone: the user books a flight, checks in, sends their baggage, boards the flight and lands and so on. Similar actions are usually repeated and the importance with these types of information is the data within them such as, 'How much weight did the user send through?', 'How late were they to arrive for boarding' and so on. Transactions within healthcare will relatively be very unique, each department or subset organisation within a given hospital will have its own set of transactions that go through as entries. There are outpatient scans, doctor visits, radiology scans, etc. There isn't the same amount of repetition as it occurs in other industries.

The other important difference in healthcare is the type of data. In almost all cases, the result of a doctor-patient meeting is depicted in textual formats. For example, the doctors and nurses write written descriptions of the stay and outcome of the visits. After a surgery, for example, the results and the procedures conducted during it would also be conveyed in textual format. Another difference is that in order to build a well-functioning data warehouse, you would need a common vocabulary to describe the same idea. . While other domains already implement this very well, it is less so in the healthcare domain [6].

3.3 Data Mining

After you have a well-built data warehouse, the goal would be to attain very useful information from it. Data visualization, ad-hoc querying, reporting, analysis, and data mining are some of the examples of what can be done to the data.

While there are many ways to define what exactly what data mining is, Frawley et al. in 1991 have described data mining as the process of taking out potentially useful, new information from a given dataset. [7]. After an organisation has collected a huge corpus of data, various data mining algorithms are used to extract useful information from them

which in turns allows and enables the companies to make more sound decisions.

According to IBM in 2013, about 2.5 quintillion bytes of was being created daily [8], one quintillion is equal to one exabyte of data. It's a 1 with eighteen zeroes behind it. According to the latest predictions by the world economic forum [9], by about 2025, we are going to have 463 exabytes or 463 quintillion bytes being created daily. This is growth at a scale that is very hard to imagine and those with the ability to do something with this data will be doing a great disservice to the people of our world if we didn't.

For a data mining algorithm to be able to produce the best results, it needs to be fed data, and that too - a lot of data [10]. This is where the benefit of having a data warehouse to store the data comes into play. Users can query the data warehouse in order to produce datasets that can then be used to perform data mining and machine learning. With Big-Query, we are able to connect to the warehouse using a selection of languages and fetch the results into a format we prefer. For this project, there is an attached python notebook that demonstrates how to query the data warehouse to fetch data into a Pandas Dataframe which can then be used as, for example, training dataset for a classification model for EHR notes.

3.4 Options for Implementation

There are various options available when it comes to where such a data warehouse can be implemented. The choice is based on multiple factors such as who the end users are, the type of data being stored and the budget.

Traditional databases are usually optimized for OLTP (Online Transactional Processing), these store operational data of the organisation at the current time, these are very useful for very quick query processing, consistency and, data integrity. An example of such a database would be one that is responsible for dealing with user sessions on a website. It should be able to validate login information and allow the user to enter in with minimal delay. The data would be structured within these in a normalized manner to ensure data integrity. Hbase is an example of such a system built on top of Hadoop in order to facilitate OLTP workloads.

While on the other hand, OLAP (Online Analytical Processing) deals with historical data. The queries with these systems are usually very complex and take longer times to generate outputs. These are very useful when needing to perform a query on a large corpus of data. OLAP systems are primarily query answering systems used to query large datasets. They, usually, do not ensure fast response times as there is often a delay at the start when resources are initialized to perform the query. The data is stored within these systems in a more denormalized manner in the form of star and snowflake schemas. An example of such a system would be HIVE which is built on top of Hadoop.

3.4.1 HIVE

Apache Hive is a datawarehouse software built on top of hadoop that supports OLAP queries on data stored within HDFS. Hive supports HQL(HiveQL) which is an SQL-like language that can be used to query the data. Hive converts the query into batch MapReduce jobs.

Hive was initially developed by Facebook and introduced to the world through their paper [11], the Apache software foundation took it up and improved it and released it as an open source software under the name of Apache Hive.

A system utilizing Hadoop and Hive can be implemented locally and within the cloud. However, it requires provisioning and scaling machines and capacities. As data grows, you would need to scale your storage space and the compute power along with it. This is not a trivial task and requires further engineering skills. It is also very costly to scale up in this manner. You are going to have to either rent or buy these resources for an indefinite amount of time. Such a system would also have to be managed and administration of servers requires another set of engineering skills.

3.4.2 Amazon Redshift

Amazon Redshift is a relational, OLAP database built for the cloud that can run complex analytical workloads using SQL. This service however isn't serverless, you would need to setup collection of servers called clusters. This means that you still need to specify the storage size and scale up in an ad-hoc manner as the data grows.

Because you need to determine the size of your cluster apriori, you are billed irrespective of how you use the data. You are billed on an hourly basis and if the data warehouse is not constantly being queried, this could increase costs by a lot but if there is constant access and usage of the data, this could make it a lot cheaper.

Amazon redshift does come with an online Query Editor that allows you to run queries on the dataset directly from the AWS console. This allows users to not have to setup an external client. The results are instantly displayed on the screen. Users can then download the results as csv files for further analysis elsewhere.

3.4.3 Google BigQuery

Like Redshift, BigQuery(BQ) is a cloud based, serverless, managed, data warehouse system that is built on the Google Cloud Platform. The system scales on its own based on the size of the dataset to consistently ensure fast response times for queries.

The data is managed by the system itself instead of being stored inside Google Cloud Storage. The storage costs are similar to S3 and is very cheap.

BQ also has a web UI in their cloud console that can be used to run queries. Along with it, BigQuery supports libraries that can be used to query the data using many languages such as Python and Java. This allows us to for example directly connect to the data warehouse and fetch the data into Pandas dataframes to begin analysis instantly.

3.4.4 Comparison

After having gone through multiple benchmarks of the three online, multiple results as the one done by King, a game development company show that Hive is slower than BigQuery [12] for queries. With respect to the comparisons between BigQuery and Redshift in terms of performance, they both outperform each other based on the benchmark. While many

were found, none of them could actually conclude one as being better as there would always be another one saying otherwise. Hive also does not allow users to connect to the data warehouse using RESTful queries that the other two do support which will make it easier to integrate into other projects.

However, the speed within Redshift has an upper limit based on the configuration of the cluster. Bigger clusters with massive CPU storage results in faster queries while smaller ones slow down the queries substantially. With BigQuery, this is not an issue as the service scales compute power on the fly based on the data.

One of such benchmarks comparing the two, done by engineers at Panoply [13] show that in terms of loading data, BigQuery does perform much better, but this does not matter as much as query speeds when it comes to a data warehouse.

Both services support caching while they are different for each service. BigQuery caches results while RedShift caches data. This implies that RedShift has a gain when it comes to queries that query similar data but if a lot of the queries run on the data warehouse are repeats of each other, BigQuery comes out on top as it stores all the results for the past 24 hours. Queries within BigQuery that have been cached run for free and the data is returned instantly without any delay.

One major advantage that these two systems share is that they are columnar, as in, they store data columns rather than as by rows. This makes querying extremely fast and beneficial due to three things,

1. It only needs to scan the columns that we need to access and so it can ignore all the columns that are not included in the output
2. Columnar databases utilize compression to reduce the amount of data stored within each column, thus again leading to faster query speeds.
3. The final benefit is cost, because the amount of data accessed is used to calculate the running costs for these two systems, being able to choose which columns it needs to scan helps vastly reduce the cost.

The screenshot displays the Google BigQuery Online Query Editor interface. At the top, the 'Query editor' tab is active, showing a SQL query that filters for births in the US where the mother's age is under 20. Below the editor, the 'Processing location: US' is indicated, and a status message states 'This query will process 2.5 GB when run.' with a green checkmark. The 'Query results' section shows the query is complete, having elapsed 2.6 seconds and processed 2.5 GB. Below this, a table of results is displayed with columns for Row, Year, Births_Under20, and Births. The results show data for the years 1999 through 2004, with the number of births decreasing over time.

Row	Year	Births_Under20	Births
1	2004	3236	50670
2	2003	3332	49860
3	2002	3607	49182
4	2001	3915	47959
5	2000	4184	47353
6	1999	4296	46290

Figure 3.1: Google BigQuery Online Query Editor

	c	url
1	1042	http://www.abcxyz.com:80/jobbrowser/?format=json&state=running&user=9u45ok8
2	366	http://www.abcxyz.com:80/jobbrowser/?format=json&state=running&user=haiwb93
3	316	http://www.abcxyz.com:80/jobbrowser/?format=json&state=running&user=248nm5
4	268	http://www.abcxyz.com:80/jobbrowser/?format=json&state=running&user=20g578y
5	102	http://www.abcxyz.com:80/jobbrowser/?format=json&state=running&user=15llx5s
6	63	http://www.abcxyz.com:80/jobbrowser/?format=json&state=running&user=fmi7id4
7	62	http://www.abcxyz.com:80/jobbrowser/?format=json&state=running&user=xbqr86y
8	61	http://www.abcxyz.com:80/jobbrowser/?format=json&state=running&user=aj0hcai
9	54	http://www.abcxyz.com:80/pig/watch/0000038-140911233734310-oozie-oozi-W?format=python

Figure 3.2: Amazon Redshift Online Query Editor

3.5 Design Methodologies

The two most common approaches are the data-driven approach and the user requirement based approach [14]. The former style is called the "Bill Inmon Style" and the latter the "Ralph Kimball Style" [15].

The data-driven approach is based on building a multidimensional model according to the data we have available, while this is feasible in most instances - the issue with this approach is that it doesn't consider user requirements. One might only find out after building the data warehouse that it is not suitable for answering the questions that the end-user has [16].

The other common approach is a requirement driven approach in which the user's needs are considered first and then a multidimensional schema is built based on it. In this methodology, data mart creation occurs after the data warehouse has been built. The issue with this is that sometimes, the data might not integrate well into the DW.

Another alternate approach being researched at the moment is a hybrid approach in which both of these things are taken into account [17]. This approach allows users to design schemas that effectively support the user requirements while effectively considering the data available at hand. However, utilizing a hybrid approach leads to a more complex process with regards to designing the DW.

Christina et al. reviewed various Clinical DW design methodologies based on seven categories: "data integrity, sound temporal schema design, query expressiveness, heterogeneous data integration, knowledge/source evolution integration, traceability and guided automation" [18]. According to the study, about half of them used a hybrid approach. 26 out of

the 40 papers used dimensional modeling while 8 used the relational model. According to the study, none of the papers covers all of the categories listed above efficiently.

3.5.1 Star Schema

Star schema is the most common model [19] in data warehouse implementations. The star schema model, as shown in figure 3.1, consists of a central fact table surrounded by dimension tables directly linked to the fact table.

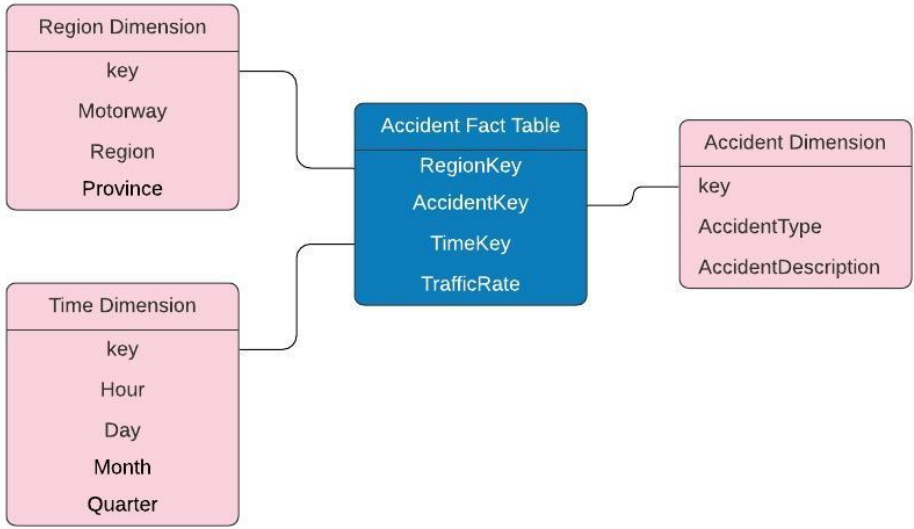


Figure 3.3: An example of a star schema

The fact table contains keys to the dimension tables and measures. In figure 3.3, 'TrafficRate' and 'Map' are measures, they contain the rate at which traffic flows as a number and the map contains spatial data. The dimension tables contain attributes that relate to that dimension.

3.5.2 Snowflake Schema

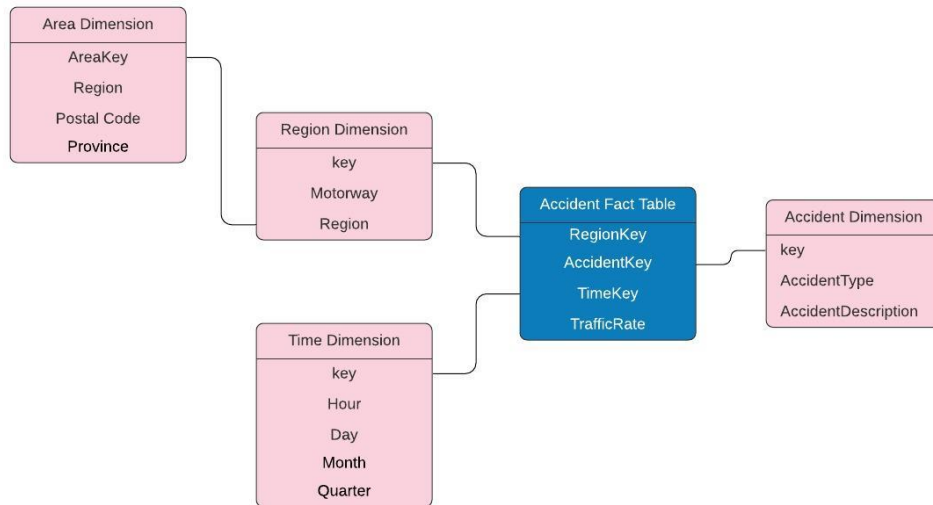


Figure 3.4: An example of a snowflake schema (github.com/jlroo)

Although very similar to the star Schema, the snowflake schema gets a bit more complex, instead of just a single fact table and dimension tables around it, the dimension tables are normalized and linked to other dimension tables. The purpose and the drive behind normalizing the dimension tables is to save space but that has downsides, while it now has much smaller space footprint - it results in more number of dimension tables resulting in a need for more join operations and thus more complex queries and reduced performance [20].

3.5.3 Fact Constellation Schema

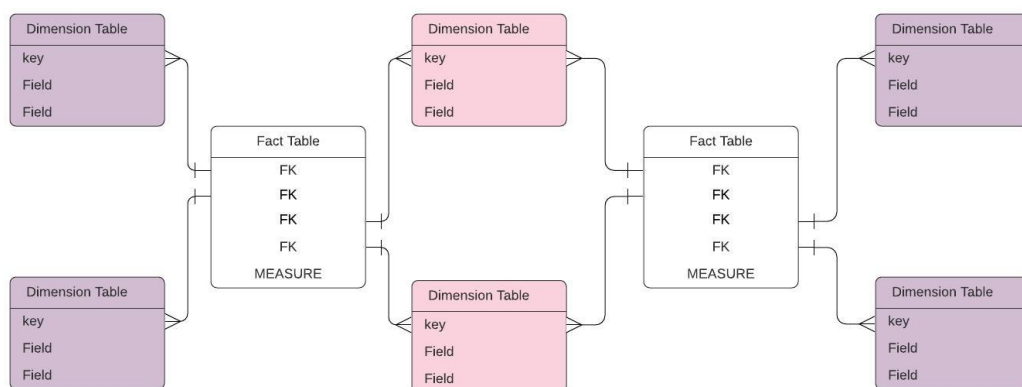


Figure 3.5: Fact Constellation Schema

A fact constellation schema is a schema in which a set of fact tables share common dimensional tables. It is also often referred to as a galaxy schema due to it resembling a group of star schemas connected to each other. A schema such as this can be used to decompose a complex fact table into smaller fact tables as this structure allows for higher flexibility in modeling but on the other hand querying ends up being more difficult due to the added aggregations required.

3.5.4 Denormalized Schema

Normalization is a technique used to reduce redundancy of data. This is crucial for OLTP systems as data that is not normalized will need to be updated in multiple locations if an update does occur. While Star Schema is a highly denormalized data structure, it still requires joins among the fact tables and the dimension tables way too often. Joins are costly for queries and slows down queries, this is trivial for tables with small number of rows but when data expands, the time required to join two such tables increases exponentially.

Within BigQuery, it is possible to denormalize the data further to form a form of data structure in which the table joins are already done. This leads to redundant data and increases storage space but storage is cheap and data integrity is not an issue because we are not updating the data within the warehouse and only analysing and performing queries on it. The benefit of having faster query times outweigh the disadvantages of redundant data.

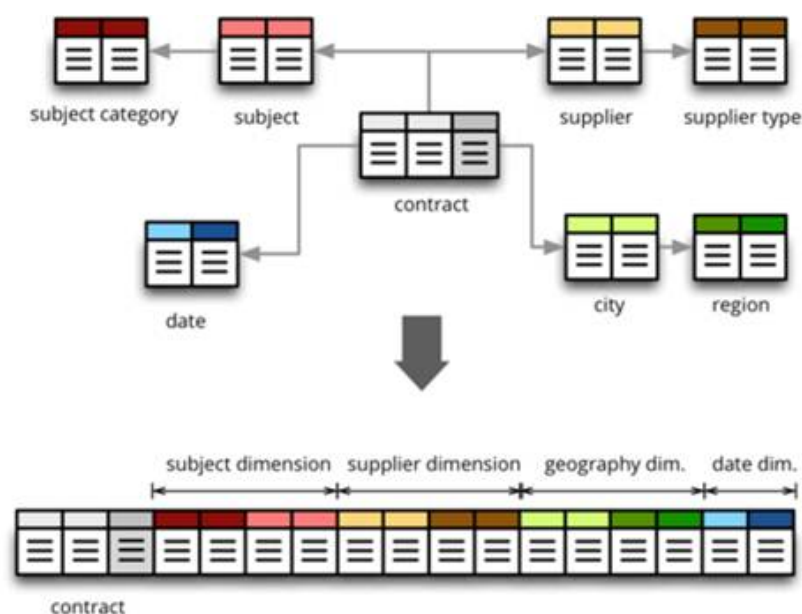


Figure 3.6: Example Denormalization (src: BigQuery Documentation)

In the example in Figure 3.3, if you perform a query on the snowflake schema before it is denormalized, you would have to perform a number of joins in order to do many of the complex queries possible, this results in slower response times. Denormalizing it and joining it into one single table results in there no longer being a need to join these tables allowing the user to perform aggregation and slice the data much faster.

Such denormalizations are then stored using the STRUCT data type within bigquery. This can be also applied to repeated data such as one that traditionally requires a bridge table to be stored as an array. For example, if a user in the database has multiple phone numbers, we would traditionally use a bridge table to show this relation. However, in a denormalized schema, you can store these numbers as repeated fields in the same table removing the need to perform two joins, one to the grouping table and one to the dimension table containing the data of the number.

3.6 Different Types of Architecture

The general architecture of a data warehouse consists of extracting data from various sources which is then cleaned and transformed in the staging area and then the data is loaded into the data warehouse which then persistently stores the data.

There are different ways, however, in terms of how data is stored in the data warehouse. Ariyachandra and Watson, 2005 describe a few of these models and how they integrate everything together [21]. They talk about how these are the models to be used in order to get guidance when building a new model.

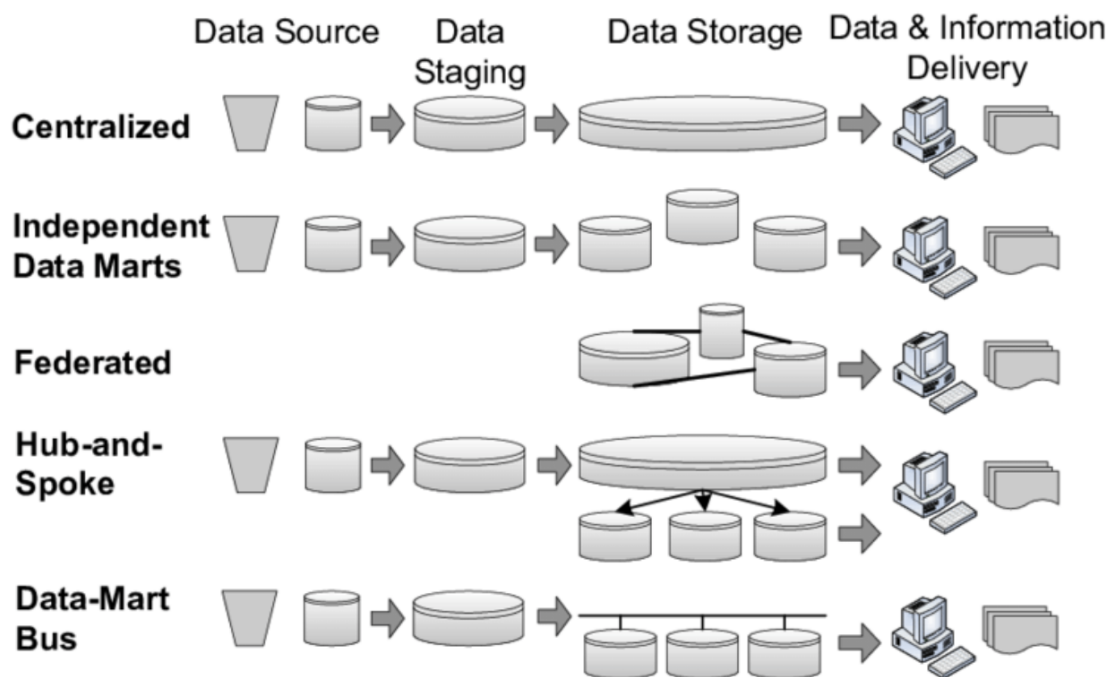


Figure 3.7: Basic Five Data Warehouse Architectures [22]

3.6.1 Independent Data Marts

As opposed to storing the entire data into a single storage section, data is divided and put into various subsections that are designed to suit different demands of individual units within a given organisation. For example, a business might have a different data mart for its sales section and another for its marketing section. According to Ariyachandra and Watson, these do not provide a "single version of the truth" because of how difficult is to analyse data across the data marts due to varying schemas and data definitions [21].

3.6.2 Centralized Data Warehouse

This is a data warehouse that is built based on the requirements of the entire organisation. It contains a single, large, data warehouse.

3.6.3 Federated Architecture

Federated data warehouses is an architecture used to combine multiple data sources in order to provide a single version of the truth unlike in independent data marts. This is done by leaving the existing decision-support systems in place and then combining the data using things like shared keys, global metadata and distributed queries [23].

3.6.4 Hub and Spoke Architecture

This is also oftentimes referred to as the 'Corporate Information Factory'. This is like the centralized data warehouse but it also has independent data marts into which data is fed from the centralized store. The centralised data warehouse in this instance acts as the hub and the data marts as spokes and that is where it gets the name from.

3.6.5 Data Mart Bus Architecture

This is based on a bottom-up approach. Multiple data marts are built and then connected using a shared bus [24]. This bus in this instance is the list of shared elements among the data warehouses.

3.6.6 Comparisons

Ariyachandra and Watson performed a study comparing the different architectures above by surveying various companies big and small on their design choices. According to the survey [21], the hub and spoke lead in terms of architecture usage followed by data mart bus.

According to another survey conducted by Forrester at a data warehousing conference to 213 practitioners, hub and spoke came on top followed by the centralized data warehouse [25].

However, describing this survey Agosta has claimed that "No data warehousing architecture is right or wrong in itself" [25]. He states that enterprises have succeeded with many different models and that the reason different institutions pick different architectures is that the architecture often mirrors the structure of the organisation. He states that centralized enterprises usually stick with centralized data warehouses and federated architecture works better in organisations with a "mixed pattern of governance".

Chapter 4: Related Work and Ideas

This section describes the common clinical data warehouses being used today and describes their key features. A lot more exist than are documented in this section but only a few provide satisfactory documentation online to study their systems.

4.1 Stride

Stride(Stanford Translational Research Integrated Database Environment) [26] is a research and development project made by Stanford University. Stride has three components. To begin with, it has a clinical data warehouse that contains data from over 1.3 million patients who were cared for at Standford University Medical center. It also has an application development framework for building research data management applications and a biospecimen data management system. Since all three of them are based on the same architecture, it supports linking data across the three components [26]. It also supports structured and unstructured data from various sources. The system utilizes HL7 messages, which are messages used to transfer electronic data between healthcare systems [27].

4.2 Radbank

Radbank [28] is another clinical data warehouse system built by Stanford. This data warehouse is used to integrate radiology and pathology data into a single location. The warehouse utilizes a relational database system. The schema was designed specifically to link radiology and pathology reports and then to support various queries over this domain. Like Stride, it utilises HL7 messages in order to keep the system up-to-date. Radbank stores the data as a relational model and queries over it using SQL. It does not denormalize the data as much as other systems.

4.3 Google, UCC (Chicago), and UCSF Data Warehouse

Google joined hands in the last few years with Stanford Medicine, UC San Fransisco and University of Chicago medicine to build a data warehouse that used deep learning methods to make predictions. Google acquires de-identified patient data in the form of EHR from the three institutions in order to train its model through HL7 [4]. The system utilises FHIR (Fast Healthcare Interoperability Resources) in order to store clinical data. FHIR is a data standard developed by HL7 in order to simplify the implementation of healthcare datasets.

It is a means of leveraging existing models that are easy to implement to accomodate exchanging data between healthcare applications.

This system uses the EHR to perform deep learning and hence according to the paper performs much better than the other models currently being used in things like predicting mortality (0.92–0.94 vs 0.91) [4]. Another benefit as stated by the paper is that it didn't use variables hand-picked by experts to perform these analytics but rather fed the entire data to the system which included thousands of variables and let the system decide what was important.

4.4 SMEYEDAT

SMEYEDAT (Smart Eye Database) is a clinical data warehouse concentrated on ophthalmology [29]. They have stored the data of 325,767 patients and stored it in a database built on an SQL database. They built the schema using a patient-centered data model and thus resulted in a "star-like" schema. As in, the patient was made the fact table with dimension tables such as diagnoses, procedures, etc. Why it is called 'star-like' and not a 'star' schema is not evident in the paper.

Rather than directly querying the database using SQL queries, they implemented Qlikview as a data visualisation and query tool [29]. This tool, in particular, was chosen due to it being successfully implemented in other fields of medicine [30].

4.5 i2b2 Star Schema

i2b2 (Informatics for Integrating Biology and the Bedside) is a clinical data warehouse that is modeled on the star schema structure. It has a central fact table surrounded by dimensions. It stores each observation as an entry as a fact. For example, a single diagnosis would be stored as a fact in this table. It uses the Entity-Attribute-Value (EAV) model to do this utilizing a concept table allowing it to expand the schema to support multiple forms of data.

Due to the nature of the EAV model, the queries end up being complicated as the result would then need to be pivoted to conform to the form a user would want it and thus they have an abstraction in the form of software that is used to produce the queries and so end users only need to worry about knowing how to use the software rather than knowing how to write queries.

Chapter 5: Outline of Approach

After having done a review of over a hundred papers on the topic and a few dozen based solely on clinical data warehouses - it still seems like a very tough decision on having to settle on a specific model or architecture.

1. Considering that this project is based on data available on hand rather than questions that need to be answered, it seems more intuitive to design the system using a bottom-up or rather a data-centric approach and then figure out the questions that can be asked and answered based on the implementation.
2. Based on the research done in sections 3 and 4, it seems that the ideal schema would be a star schema due to how feasible it is to implement and how successful it already has been in the domain.
3. A bit more research is required in terms of choosing a specific architecture for the system but at the moment it seems as if the Centralized Data Warehouse seems to be the most suitable one for this project.
4. The next step involves designing the fact table and the dimension tables. As seen in other models in section 3, having an encounter in the center as a fact table and the other features built as dimensions around it seems to be the way forward.
5. Following this, as ELT (Extract, Transform and Load) is chosen over ETL (Extract, Load and Transform), the dataset will be exported from the PostgreSQL database and transferred onto the BigQuery database. The data is then transformed to conform to the schema proposed using SQL.

Chapter 6: Project Work Plan

In this section the project is broken down into individual work packages, these are described along with the approximate time intervals on the Gantt chart below.

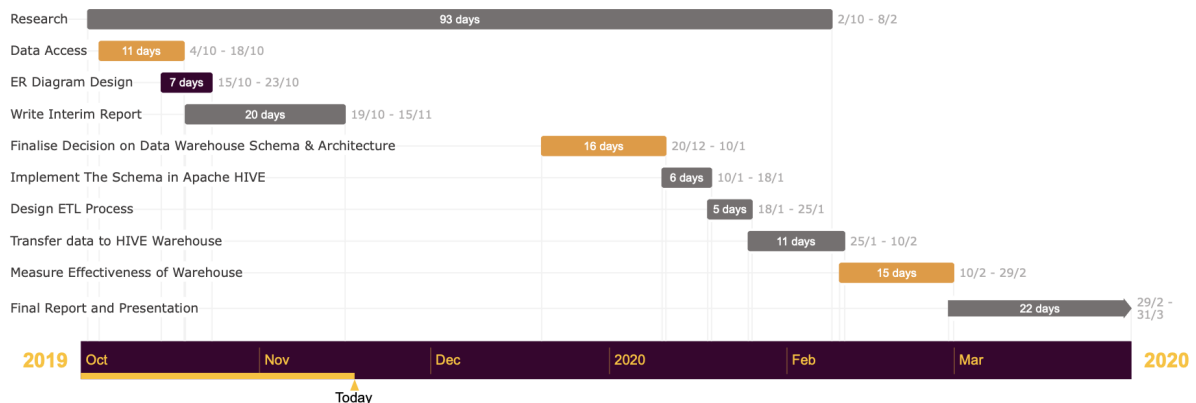


Figure 6.1: Project Workplan Gant Chart

6.1 Packages

1. Research

Considering how important research is to this project, this is not going to be one that is going to be done any time soon. There needs to be constant research done on all the choices that are to be made in order to ensure that they are done based on clear evidence and further research done in order to ensure that the decisions that have already been taken are still the sound choices.

2. Data Access

Attain the MIMIC-III database from PhysioNet and store it in a local device so that it can then be loaded onto the data warehouse once it has been implemented.

3. Derive ER Diagram

Design the ER diagram based on the attained healthcare dataset (MIMIC-III), this has been done and attached in Chapter 4.

4. Write Interim Report & Prepare the Presentation.

Complete the interim report summarising current progress and make a 10-minute presentation.

5. Finalise Decision on Data Warehouse Schema and Model

Perform more background research and finalize decisions.

6. Implement in BigQuery

Implement the decided model and architecture using a data warehouse system (BigQuery).

7. **ELT Process Design**

Figure out the best way to clean, transform and load the data onto the warehouse.

8. **Load Data into Warehouse -**

Load the MIMIC-III data into the warehouse.

9. **Measure Effectiveness**

Compare how good the result is with other implementations available on the market today in terms of its ability to support Big Data Analytics.

10. **Final Report and Presentation**

Write the final report and prepare the presentation

6.2 Evaluation

The data warehouse after being implemented within BigQuery. The data will be transformed and loaded and the warehouse will be compared the OLTP database to compare speed and query complexity. An ideal implementation would result in faster query responses for OLAP queries while also making queries simpler to write.

Chapter 7: Data and Context

For this project, the dataset used is the MIMIC-III dataset (Medical Information Mart for Intensive Care). This is a dataset built by researchers at MIT. It is a relational dataset containing de-identified and comprehensive clinical data from patients admitted to Beth Israel Deaconess Medical Center. The data in MIMIC-III has been de-identified according to Health Insurance Portability and Accountability Act (HIPAA) standards.

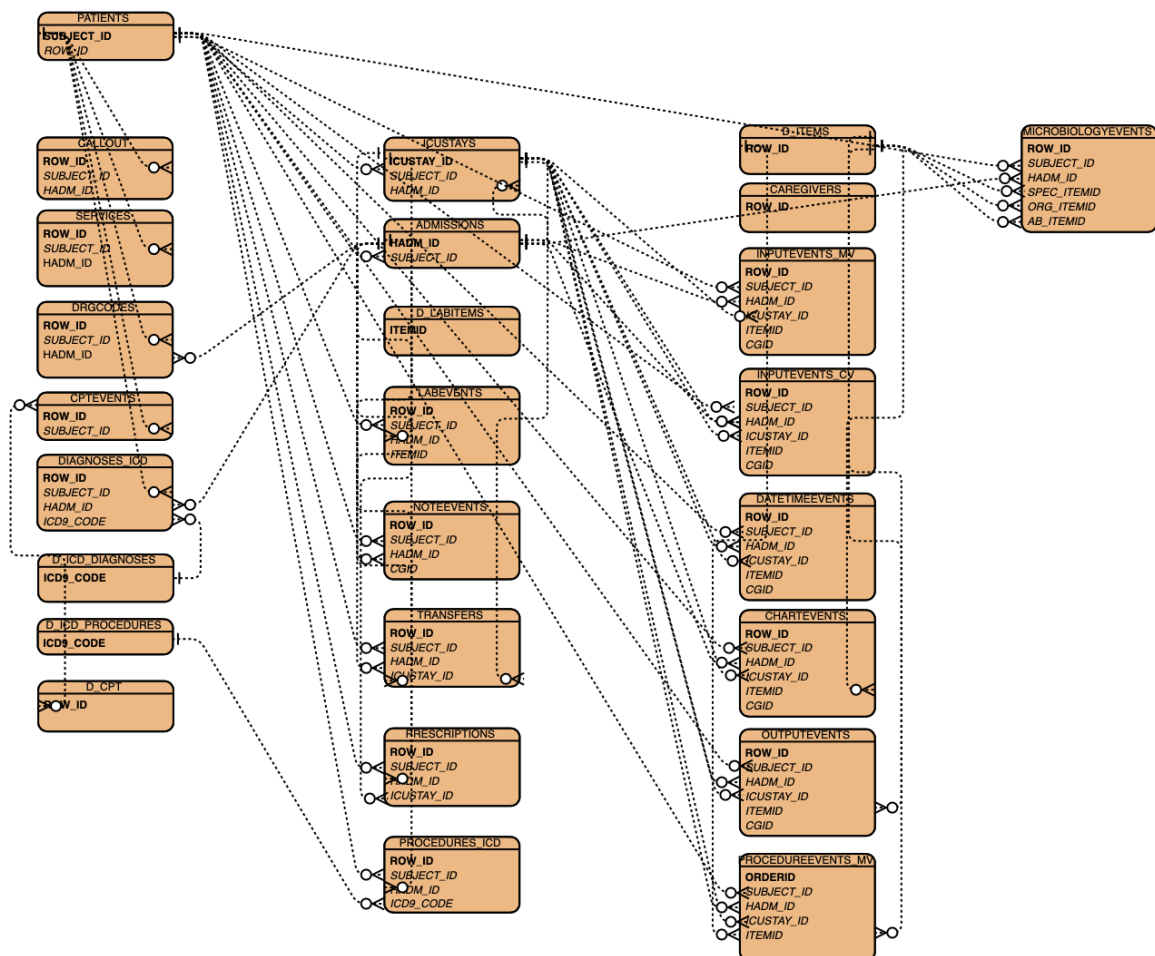


Figure 7.1: MIMIC-III Dataset ER Diagram

Within the database, patients are identified using the `subject_id` attribute present in many of the tables, each patient has one or more admissions (identified using the `hadm_id` in the admissions table) and each admission can have one or more icustays (identified using `icustay`).

A vast majority of the crucial data such as diagnoses, procedures and notes are grouped based on a single admission. Lab Events, however, are patient-centric and not necessarily admission centric even though they can occur within an admission.

This is a massive dataset that contains information of over 50 thousand hospital admissions to the critical care unit between the years 2001 and 2012. It contains data of 38,597 distinct

adult patients. This database is a relational database and contains 26 tables. [31]. An ER-Diagram for the dataset is shown in Figure 5.1. As evident, it is a very complex dataset with a lot of relations between the tables.

7.0.1 Accessing the Dataset

The dataset, although de-identified, still has detailed patient information that must be handled with care. Therefore, the dataset is not openly available on the web. Users must complete a required training course in order to access the data. Following the course, the dataset was available for download as csv files. The files were downloaded and stored locally.

7.0.2 Data Pre-Processing

The dataset is extracted from two systems, CareVue and Metavision and is combined into one. There is little left to be done with respect to pre-processing as the dataset is already cleaned and de-identification has been done. The only processing done was remove a few entries from the noteevents table where it was not linked to a specific admission. The only table in which entries could exist without being linked to a valid admission is the labevent table because it contains data for outpatients as well.

7.0.3 Data Loading

The dataset was then loaded into PostgreSQL using the script provided on the mimic repo. The data can be loaded in without using the script but script contains other optimizations that can speed up querying such as partitioning of the dataset along with indexing. The methodology found within the script is well tested over time and optimized so we know that the data is loaded in an efficient manner. Following this, the dataset was verified for data integrity using queries to check for entries within each table.

After having followed the steps above, a local OLTP database has been created that can represent a clinical data transactional database.

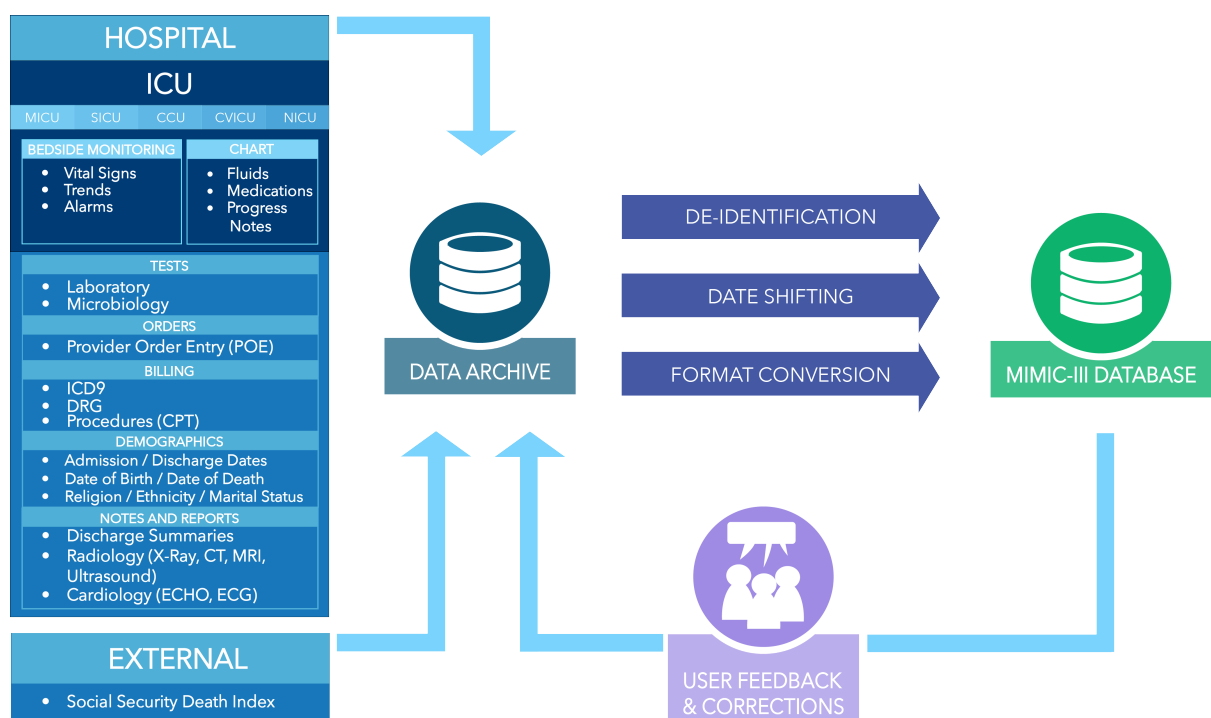


Figure 7.2: MIMIC-III Database Overview [32]

Chapter 8: Schema Design

This chapter introduces the methodologies used to design the schema implemented within the warehouse along with the alternative approaches that could have been used in order to implement a schema. Data warehouses use schemas to logically describe datasets. This chapter outlines benefits and disadvantages of the schemas that were considered for this data warehouse.

8.1 Fact Constellation

An initial implementation was based on a fact constellation schema, due to the nature of the data being divided into different processes such as diagnoses and procedures, we are able to create a fact table describing each process. Such an implementation would result in twelve fact tables (Diagnoses, Procedures, LabNotes, Lab Tests, etc.) along with a set of dimensions describing each and common dimensions such as patient and admission.

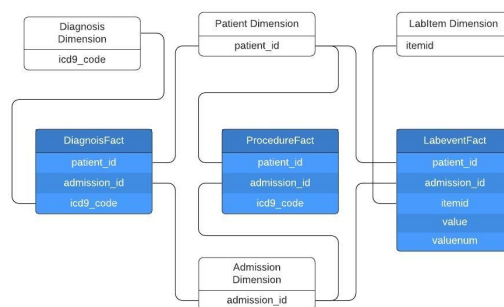


Figure 8.1: Example Implementation as a Fact Constellation (Not all tables are shown in this diagram)

This, however, leads to a lot of fact tables and the amount of data available in each is minimal. A normal query would usually be a combination of four or five fact tables. This would require the end-user to "drill accross" the tables, which basically means querying each fact table separately and aggregated to a common level of detail and then combining the results into one based on common dimensions. This is a complex procedure and results in unnecessarily large queries and a lot of joins.

8.2 EAV-Star Schema

The second model that was researched was the EAV structure (Entity-Attribute-Value). The EAV model is based on a key-value pair pattern commonly used in areas such as research and healthcare. The key advantage this sort of a model has over traditional ER schemas is that it is able to facilitate heterogenous data very well and expand to support the sparsity

of the data. It is a model which optimises for storage, rather than recovery of the data from the store. Before this dataset can be utilized for analysis, such as data mining and deep learning, the dataset needs to be pivoted into conventional relational table format [33]. This is a expensive, time consuming operation that often results in queries that are difficult to maintain. Since the dataset is large, and often is in the case of clinical data, flattening/pivoting at every query becomes a tedious task. Thus, this more than often results in a persistence strategy that involves transforming the data into a relational format and then loading it into a separate system often on different hardware.

Table 8.1: Example of Entries inside EAV Fact Table

concept(entity)	attribute	value
1	Age	23
1	Gender	Male
1	Heartrate	90
2	Age	64
2	Gender	Female
2	HeartRate	60

Table 8.2: Conventional Relational Table After Pivoting the EAV table

Entity	Age	Gender	Heart Rate
1	23	Male	90
2	64	Female	60

An example of entries within an EAV table is shown in table 8.1, as is evident, this format is not very useful to perform analysis on. This it needs to be transformed into a form shown in table 8.2. This data can then be used for other applications like machine learning to do analysis.

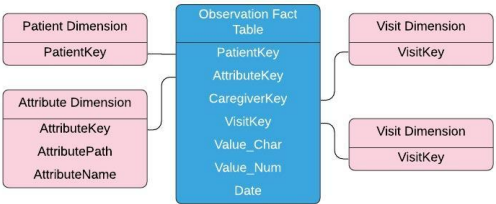


Figure 8.2: An EAV-Schema Implementation (Not all fields are shown)

Hence, an alternative model was required. It would not make sense to store data within the data warehouse only to extract the data onto another data warehouse, this is more suited for a transactional system that requires fast storing and heterogeneous, sparse data that is not yet known to the database system.

8.3 Proposed Schema

The proposed schema is based on the fact constellation or a galaxy schema, this is due to the nature of the data available. A fact constellation schema, as described above, is one which has multiple fact tables with common dimension tables. It is flexible and fit for our purpose. Most data within the database is derived from an admission or an episode of care, thus, a lot of the data can be grouped together into a single admission. However, the lab data is not centered around admissions but rather is patient centric and so has a different level of detail. While the rows within the primary fact table, describe a specific episode of care, a row inside lab event fact table describe a specific lab event per patient.

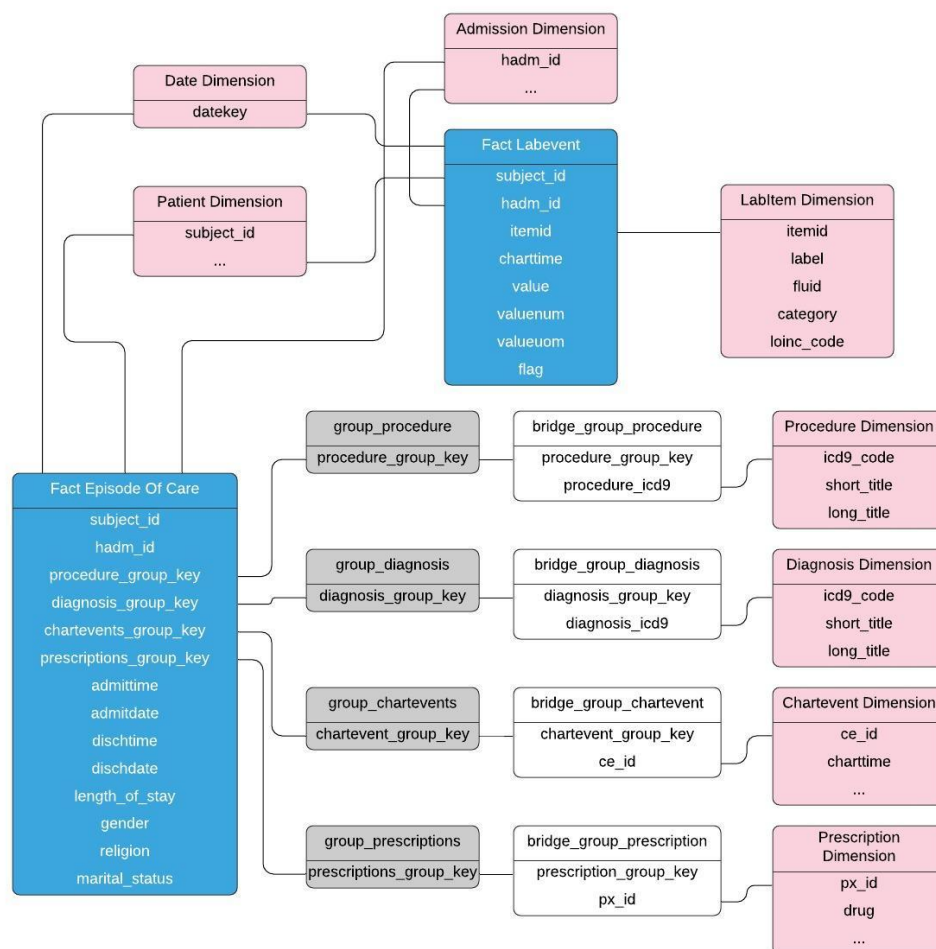


Figure 8.3: The Proposed Schema (Not all fields are shown)

The proposed schema is shown in Figure 8.3, it contains two fact tables and seventeen dimension tables. Time, Patient, Admission are three dimension tables that are common to both fact tables. This is what makes it a fact constellation. The tables shown in the figure do not show the entire model as it is too big to represent on paper, there are many more bridge tables and dimensions than shown. Each row in the 'Episode of Care' fact table represents a single admission and the details within that admission.

With each admission, there exists zero or more diagnoses, procedures, etc. These can be represented in two ways, either by continually expanding the table with more columns such that you have enough to store the max number of diagnoses, for example if the max number of diagnoses a patient in the database had in one admission was three, you could add three

columns to the table, (diagnosis_1, diagnosis_2 and diagnosis_3) but this is a terrible idea for clinical data as this would grow exponentially and would be very difficult to store. Hence, a bridge tables are used used to represent the collection of such items.

The group table is used to generate the bridge group keys. The bridge group table is used to represent the group key and all the diagnoses within said group. The dimension table (diagnosis, procedure, etc.) is used to describe a single event. This also saves space because the grouping is done irrespective of the patient, so if two patients have the same set of diagnoses for example, the same group key can be used to describe both patients.

8.4 Optimizing Schema for BigQuery

The table shown above in 8.3 can be optimized to better utilize the functionalities provided within BigQuery. BigQuery performs best when data is denormalized rather than having to do joins. Joins are a costly operation within the system. With a conventional datawarehouse system, the method of denormalization involves making an entry for each repeated dimension into the fact table. Denormalization within BigQuery is done taking advantage of nested and repeated fields. The denormalized data is then represented as an object (using a STRUCT) or an array of objects (by creating an ARRAY of STRUCTs).

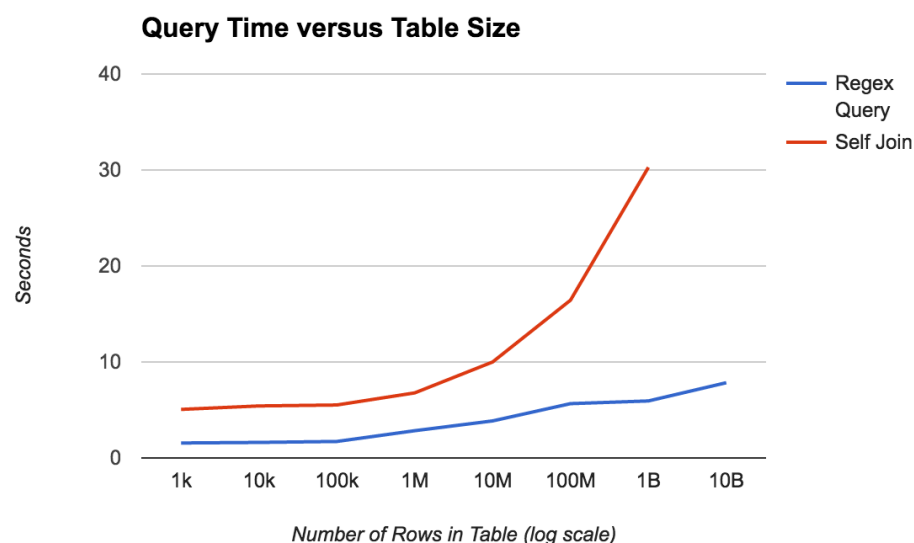


Figure 8.4: BigQuery Speed Comparison Between Self Join and Filtering [34]

Figure 8.4 shows how much beneficial it is to not have to join tables, as data grows, the time taken to join tables grows exponentially making queries exponentially slower as a result.

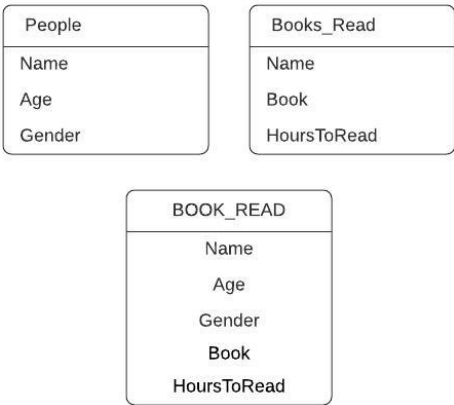


Figure 8.5: Traditional Flattening of a Table

Figure 8.5 shows how data is traditionally flattened, in this example, for each book a person has read, you would have a single entry within the BOOK_READ table. As in, if a person has read three books, you would have three rows in the 'BOOK_READ' table.

However, utilizing nested fields, we can store this information as an array rather than a new entry for each item. This results in the following structure. With this, you would only have one row per person and have all the books the person has read within the books_read field which is a repeated field. This can be queried using a simple UNNEST().

```
# Example query to fetch name and the number of books read.  
select Name, count(books_read) From BOOK_READ, UNNEST(books_read)
```



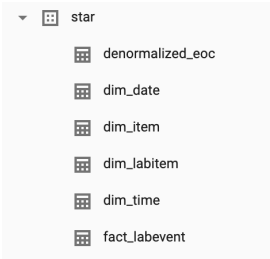
Figure 8.6: Denormalization Using Repeated Fields

In our schema, we utilize bridge tables to represented an array of diagnoses or procedures, this can be denormalized and stored within the fact table making it easier and faster to query.

In the end, after denormalization, we end up with six tables that consists of two fact tables, one representing the episode of care and one for the lab events and four dimensional tables (date, time, labitem (connected to labevent), item).

The episode of care fact table contains information pertaining to an admission (subject__id, hadm__id, admittime, admitdate, disctime, dischdate, dob, deathtime, deathdate, admission__type, admission__location, discharge__location, insurance, language, gender, religion, marital__status, ethnicity, edregtime, edouttime, first__diagnosis, hospital__expiry__flag, expire__flag and has__chartevents__data) along with the denormalized bridge tables (procedures, diagnoses, outputevents, noteevents, microbiologyevents, labevents, callouts, cptevents, transfers, services, drgcodes, datetimeevents, chartevents, procedureevents_mv, inputevents_cv, inputevents_mv and prescriptions).

The labevent fact table contains subject__id, hadm__id, itemid, charttime, value, valuenum, valueuom, labevent.



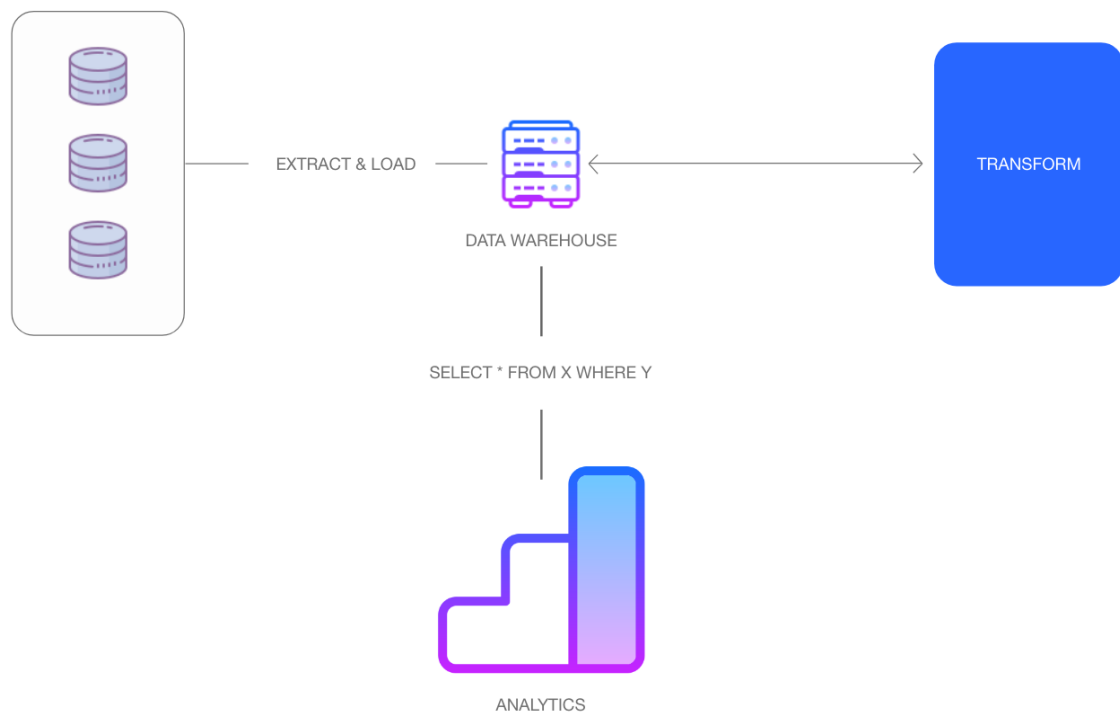
star
denormalized_eoc
dim_date
dim_item
dim_labitem
dim_time
fact_labevent

Figure 8.7: List of Tables

8.5 Architecture

The architecture for the data warehouse is based on an ELT (Extract, Load Transform) rather than the traditional ETL process. This is because BigQuery is scaleable and powerful and so transformations can be done within the data warehouse itself.

The process begins by extracting the data from various sources and loading it into BigQuery into a staging area, the data is then copied and transformed within BigQuery and stored inside the fact and dimension tables as specified in the schema. The data in the staging area does not get edited during the transformation process as it can be used to roll back in case an error occurs. This dataset is then queried and the results are extracted as CSV files on to other applications for analysis and other work.



Chapter 9: Implementation

The goal of this project is to design and implement an efficient data warehouse for storing healthcare data that allows end-users and analysts to query the data with ease while ensuring high throughput. To achieve this, a schema was designed as described in the previous chapter. Following this, the data warehouse was implemented inside BigQuery.

When implementing a data warehouse you need to choose between an ELT process and an ETL process. ELT was the chosen method chosen for this project.

Extract

The data is extracted from the databases and loaded directly into the data warehouse. Because the data was stored within a PostgreSQL database, the data within each table was copied onto a csv file using the following command,

```
Copy (Select * From foo) To 'example.csv' With CSV DELIMITER ',' HEADER;
```

Load

The files were then uploaded to BigQuery using Google Cloud Console and stored within the staging area.

Transform

The dataset was then transformed to conform with the schema using an SQL script that basically performs multiple selects and joins and aggregations on the datasets to form all the tables required to form the dimensional model. This took 1 minute and 43 seconds to finish.

This is based on taking row from the admissions table and then joining it with other relevant tables that conform to the same level of detail and producing the schema. For example, with the hadm_id, which is used to identify an admission, you can get all the diagnoses for that admission from the diagnoses_icd table by doing a left join on hadm_id, this was then grouped into an array of structs and stored.

After this all the fact tables and dimensions were ready within the DW for analysis.

9.0.1 Date and Time Dimensions

The dataset has time values, date values and values that have both time and date. Because the dataset contains values for these that are decades wide, it is not feasible to make just a

single datetime dimension as that would have 86400 entries for each day due to the number of seconds and then this would have to be expanded for all dates (86400 * 365 * X Years entries). In order to solve this issue, a date dimension was generated using SQL that stores dates from the year 2000 to the year 2999. Also, a time dimension was created to that can store each second within the day so in total there are 86,400 entries within it.

The date dimension has the measures (year, month, day, dayofweek, dayofyear, quarter, week, weekend, dayname, monthname) while the time dimension has (hour, minute, second, hour12 (hour in the 12-hour format), and AMPM (to combine with 12 hour format).

Query editor

+ COMPOSE NEW QUERY HIDE EDITOR FULL SCREEN

```
1 create table star.denormalized_eoc as
2 SELECT
3   admission.subject_id,
4   admission.hadm_id,
5   time(admission.admittime) AS admittime,
6   DATE(admission.admittime) AS admitdate,
7   time(admission.dischtime) AS dischtime,
8   DATE(admission.dischtime) AS dischdate,
9   DATE(dob) AS dob,
10  time(deathtime) AS deathtime,
11  DATE(patients.dod) AS deathdate,
12  admission_type,
13  admission_location;
```

Already Exists: Table mimic-275819.star.denormalized_eoc

Processing location: US

Run Save query Save view Schedule query More

Query results

Query complete (1 min 43 sec elapsed, 30.9 GB processed)

Job information **Results** JSON Execution details

This statement created a new table named mimic-275819.star.denormalized_eoc. Go to table

Figure 9.1: Result after performing the Transformation

Chapter 10: Performance Analysis and Evaluation

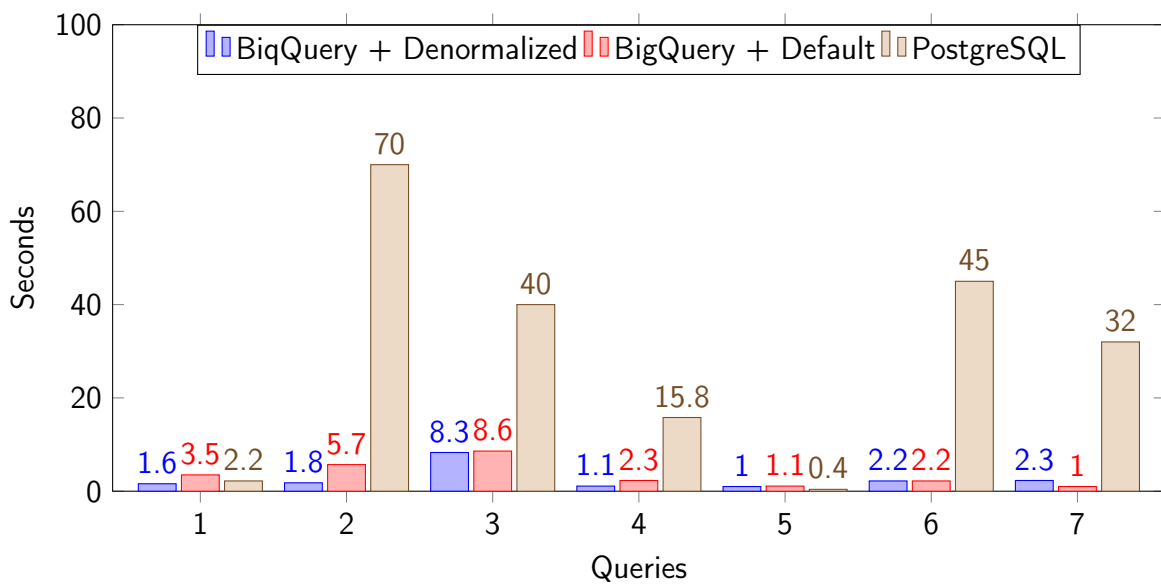
This chapter sets out how the project was evaluated for performance and ease-of-use. In order to do this, the data warehouse was set up following which a PostgreSQL server was set up on Google's cloud platform. This server was configured with 6GB of ram and 80GB SSD space. Data was then loaded into the PostgreSQL instance through Google Cloud Storage. Performance of the data warehouse is then compared by making the same query on three places,

1. The data within the live data warehouse within which the data is stored using the proposed schema.
2. The data within the staging area on bigquery which is stored as the original schema.
3. The PostgreSQL database that has the original schema.

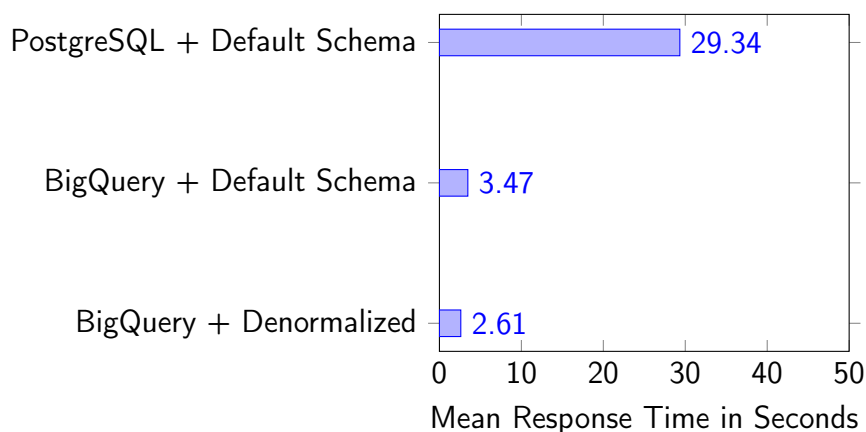
Queries (SEE APPENDIX FOR SQL)

The following queries were used to evaluate:

1. Common diagnoses and the number of patients affected.
2. Average number of chart events based on gender.
3. Chart events where month is May.
4. Most commonly used drugs on patients during admissions where they have been diagnosed with pneumonia
5. Most common diagnoses that patients who get prescribed 'Hydroxychloroquine Sulfate' have.
6. Simple 'select where' on chartevents (Biggest table with 27GB size)
7. Average value for each item tested inside lab events.



As is evident in the graph above, BQ consistently has very fast response times. However, there are scenarios in which the PostgreSQL database does better such as when the data queried is small in size or when the query is not very complex. This is the case with query 5 as it is joining two relatively small tables and making a query on it. Based on these queries and other queries done while testing, it is almost always the case that when PostgreSQL database takes over 4-5 seconds to generate the response, BigQuery tends to do better.



If we look at the mean response times, we can see that the denormalized schema is doing better than the default schema, using the proposed schema led to a 24.784% decrease in delay for responses.

Comparing the proposed schema + implementation with the PostgreSQL, we can see that the speed has improved by 11.24 times. This is an immense improvement, however, the selected set of queries might not be representational for all use cases. There might be some users who always use very simple queries and these might probably be faster on PostgreSQL. But for analytical queries or queries that span over a large number of rows, BigQuery is definitely going to perform better on the basis of how the two systems work itself.

The other benefit with denormalization is that queries become much simpler, you no longer need to worry about whether to use left join, right join or inner join. You also do not have to worry about what keys two tables are joined by. Having the tables pre-joined and stored allows you to replace the join with just a simple `unnest(table)` statement in the query. This has resulted in queries being much more easier to write. This is above the benefit of having a schema based on a star structure. The original schema had 26 tables and the way they

were connected was vague and difficult to decipher. But a star-schema is intuitive, it's easy to see what is being connected to what and you already know the level of detail for each element within the fact table as it all relates to a single episode of care. This makes analysis much more easier.

If you look at the queries inside appendix (At the end), you can see how much more intuitive and simpler queries are on the denormalized schema.

Chapter 11: Conclusion and Future Work

In summary, an extensible data warehousing schema was proposed to support efficient analysis over clinical dataset. The schema was optimised for being implemented within BigQuery utilizing features such as nesting and repeated fields to optimize for performance. The dataset was then uploaded and transformed within BigQuery to fit the schema. While the schema is optimized for dataset that was available, the schema is flexible and can be extended to support many more features or measurements within the clinical environment.

The implementation was evaluated by comparing the different scenarios, one on the source system, PostgreSQL, which is an OLTP system. The second being the dataset within BigQuery before it is transformed into the proposed schema and finally the dataset after it has been transformed into the proposed schema. This allows us to compare two things, how better BigQuery is over the OLTP system for the query along with the advantages/disadvantages of the proposed schema over the relational schema the data came in.

While the performance improvements seen here is relatively impressive, we do need to note that the dataset is fairly small compared to what an actual implementation would look like with terabytes of data. This is when data warehouses shine. When the data is relatively smaller, as in less than a terabyte or so in size, OLTP systems can perform a plethora of queries on it relatively quickly. For example, within this dataset, there exists a lot of relatively simple queries that will execute instantly on a traditional OLTP system that might take a second or two within the BigQuery implementation. This is because it is outside the scope of BigQuery, it is not meant to be a real-time database but one that can scale and analyse large datasets with ease.

The data warehouse supports querying through the online query editor, command line and through libraries in various languages (C, Go, Java, Node.js, PHP, Python, and Ruby). This allows end users to effectively extract data from the warehouse with ease onto their chosen platforms. An example scenario is shown inside the project repository within a Jupyter Notebook file in which the data warehouse is queried and the data is stored into a Pandas Dataframe allowing us to load it into machine learning/data mining or other use cases.

```
In [4]: os.environ['GOOGLE_APPLICATION_CREDENTIALS'] = '/Users/royalthomas/Documents/FYP/json/auth.json'

In [5]: client = bigquery.Client()

In [8]: # Select the common diagnoses and the number of patients that have been affected by it.
sql = """select short_title, icd9, count(distinct subject_id) as count_icd from star.denormalized
_eoc, unnest(diagnoses) group by short_title, icd9 order by count(icd9) desc"""
df = client.query(sql).to_dataframe()

In [10]: df.head()
Out[10]:
```

	short_title	icd9	count_icd
0	Hypertension NOS	4019	17613
1	CHF NOS	4280	9843
2	Atrial fibrillation	42731	10271
3	Cmnry athrscd native vssl	41401	10775
4	Acute kidney failure NOS	5849	7687

```
In [11]: df.size
Out[11]: 20952
```

11.1 Future Work

11.1.1 An Intelligent Query Tool

Rather than have end-users write SQL queries to fetch the data, it would be ideal for a subset of users to have a UI interface which can be used to query the data. Instead of writing queries, they should be able to drag and drop specific requirements which will then be converted into a specific query and the dataset will be shown within the software itself. This will remove the requirement of having to know and be well versed at SQL in order to query the dataset and will also allow administrators to limit the type of queries that can be run on the system, preventing issues like accidental updates or deletion of data.

11.1.2 Implementation Within Another Environment

As future work, I would like to benchmark the data warehouse on multiple platforms. There are a fair few options out there like Redshift, Hive and Snowflake that I would like to compare with BigQuery for performance, cost, scalability and storage capacity.

Bibliography

1. Dutta, R. *Health Care Data Warehouse System Architecture for Influenza (Flu) Diseases in Computer Science & Information Technology (CS & IT)* (Academy & Industry Research Collaboration Center (AIRCC), Mar. 2013).
2. Inmon, W. H. *Building the data warehouse* (John Wiley & Sons, 2005).
3. Kerkri, E. M. et al. *Journal of Medical Systems* **25**, 167–176. <https://doi.org/10.1023/a:1010728915998> (2001).
4. Rajkomar, A. et al. Scalable and accurate deep learning with electronic health records. *npj Digital Medicine* **1** (May 2018).
5. Shin, S.-Y., Kim, W. S. & Lee, J.-H. Characteristics Desired in Clinical Data Warehouse for Biomedical Research. *Healthcare Informatics Research* **20**, 109 (2014).
6. Inmon, B. <http://tdan.com/data-warehousing-in-a-healthcare-environment/4584>.
7. Frawley, W. J., Piatetsky-Shapiro, G. & Matheus, C. J. Knowledge Discovery in Databases: An Overview. *AI Magazine* **13**, 57. <https://www.aaai.org/ojs/index.php/aimagazine/article/view/1011> (Sept. 1992).
8. Jacobson, R. & Jacobson, R. 2.5 quintillion bytes of data created every day. How does CPG Retail manage it? - IBM Consumer Products Industry Blog 2013. <https://www.ibm.com/blogs/insights-on-business/consumer-products/2-5-quintillion-bytes-of-data-created-every-day-how-does-cpg-retail-manage-it/>.
9. Desjardins, J. How much data is generated each day? <https://www.weforum.org/agenda/2019/04/how-much-data-is-generated-each-day-cf4bddf29f/>.
10. Wu, X., Zhu, X., Wu, G.-Q. & Ding, W. Data mining with big data. *IEEE Transactions on Knowledge and Data Engineering* **26**, 97–107. <https://doi.org/10.1109/tkde.2013.109> (Jan. 2014).
11. Thusoo, A. et al. Hive - a petabyte scale data warehouse using Hadoop in *Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA* (eds Li, F. et al.) (IEEE Computer Society, 2010), 996–1005. <https://doi.org/10.1109/ICDE.2010.5447738>.
12. King, T. a. *Benchmarking Google BigQuery at Scale* Nov. 2018. <https://medium.com/techking/benchmarking-google-bigquery-at-scale-13e1e85f3bec>.
13. <https://blog.panoply.io/a-full-comparison-of-redshift-and-bigquery>.
14. Moscoso-Zea, O., Andres-Sampedro & Lujan-Mora, S. *Datawarehouse design for educational data mining in 2016 15th International Conference on Information Technology Based Higher Education and Training (ITHET)* (IEEE, Sept. 2016).
15. Lawyer, J. & Chowdhury, S. *Best practices in data warehousing to support business initiatives and needs in 37th Annual Hawaii International Conference on System Sciences, 2004. Proceedings of the* (Jan. 2004), 9 pp.-.
16. Abramson, I. *Data Warehouse: The Choice of Inmon versus Kimball*. IAS Inc (2004).
17. Tria, F. D., Lefons, E. & Tangorra, F. Hybrid methodology for data warehouse conceptual design by UML schemas. *Information and Software Technology* **54**, 360–379. <https://doi.org/10.1016/j.infsof.2011.11.004> (Apr. 2012).

-
18. Khnaisser, C., Lavoie, L., Diab, H. & Ethier, J.-F. in *Communications in Computer and Information Science* 76–87 (Springer International Publishing, 2015). https://doi.org/10.1007/978-3-319-23201-0_10.
 19. Sharma, A. & Sood, M. Exploring model driven architecture approach to design star schema for a data warehouse. *Proc. of Advances in Engineering and Technology Series* 7, 466–471 (2013).
 20. Butt, E. M. A., Quadri, S. & Zaman, E. M. *Star Schema Implementation for Automation of Examination Records in Proceedings of the International Conference on Frontiers in Education: Computer Science and Computer Engineering (FECS)* (2012), 1.
 21. Ariyachandra, T. & Watson, H. Key organizational factors in data warehouse architecture selection. *Decision Support Systems* 49, 200–212. <https://doi.org/10.1016/j.dss.2010.02.006> (May 2010).
 22. Dymek, D., Komnata, W. & Szwed, P. in *Beyond Databases, Architectures and Structures* 210–221 (Springer International Publishing, 2015). https://doi.org/10.1007/978-3-319-18422-7_19.
 23. Singh, S. Data warehouse and its methods. *Journal of Global Research in Computer Science* 2, 113–115 (2011).
 24. Kale, V. *Big data computing: a guide for business and technology managers* (Chapman and Hall/CRC, 2016).
 25. Agosta, L. *Hub-and-Spoke Architecture Favored* Oct. 2007. <https://www.information-management.com/news/hub-and-spoke-architecture-favored>.
 26. Lowe, H. J., Ferris, T. A., Hernandez, P. M. & Weber, S. C. *STRIDE—An integrated standards-based translational research informatics platform in AMIA Annual Symposium Proceedings* 2009 (2009), 391.
 27. Um, K. S., Kwak, Y. S., Cho, H. & Kim, I. K. Development of an HL7 interface engine, based on tree structure and streaming algorithm, for large-size messages which include image data. *Computer Methods and Programs in Biomedicine* 80, 126–140. <https://doi.org/10.1016/j.cmpb.2005.07.004> (Nov. 2005).
 28. Rubin, D. L. & Desser, T. S. A Data Warehouse for Integrating Radiologic and Pathologic Data. *Journal of the American College of Radiology* 5, 210–217. <https://doi.org/10.1016/j.jacr.2007.09.004> (Mar. 2008).
 29. Kortüm, K. U. *et al.* Using Electronic Health Records to Build an Ophthalmologic Data Warehouse and Visualize Patients' Data. *American Journal of Ophthalmology* 178, 84–93. <https://doi.org/10.1016/j.ajo.2017.03.026> (June 2017).
 30. Brenn, B. R., Choudhry, D. K. & Sacks, K. Outpatient outcomes and satisfaction in pediatric population: data from the postoperative phone call. *Pediatric Anesthesia* 26 (ed Lerman, J.) 158–163. <https://doi.org/10.1111/pan.12817> (Nov. 2015).
 31. Mandalapu, V., Ghaemmaghami, B., Mitchell, R. & Gong, J. Understanding the Relationship between Healthcare Processes and in-Hospital Weekend Mortality using MIMIC III. *Smart Health*, 100084 (Nov. 2019).
 32. Johnson, A. E. *et al.* <https://querybuilder-lcp.mit.edu>.
 33. Luo, G. & Frey, L. J. Efficient Execution Methods of Pivoting for Bulk Extraction of Entity-Attribute-Value-Modeled Data. *IEEE Journal of Biomedical and Health Informatics* 20, 644–654. <https://doi.org/10.1109/jbhi.2015.2392553> (Mar. 2016).
 34. <https://cloud.google.com/solutions/bigquery-data-warehouse>.

Chapter 12: Appendix

1 - QUERY : Get the most common diagnoses and the number of patients affected.

BIGQUERY + DENORMALIZED (1.6 sec elapsed, 18 MB processed)

```
SELECT
  short_title,
  icd9,
  COUNT(DISTINCT subject_id) AS count_icd
FROM
  star.denormalized_eoc,
  UNNEST(diagnoses)
GROUP BY
  short_title,
  icd9
ORDER BY
  COUNT(icd9) DESC
```

OLD SCHEMA + BIGQUERY (3.5 sec elapsed, 10.3 MB processed)

```
WITH
  common_diseases AS (
    SELECT
      short_title,
      diagnoses_icd.icd9_code,
      COUNT(diagnoses_icd.icd9_code) AS count_icd
    FROM
      icu.diagnoses_icd
    LEFT JOIN
      icu.d_icd_diagnoses
    ON
      diagnoses_icd.icd9_code = d_icd_diagnoses.icd9_code
    GROUP BY
      short_title,
      icd9_code
    ORDER BY
      COUNT(icd9_code) DESC)
SELECT
  short_title,
  common_diseases.icd9_code,
  count_icd,
  COUNT(DISTINCT admissions.subject_id) AS number_patients
FROM
  common_diseases
```

```

LEFT JOIN
    icu.diagnoses_icd
ON
    common_diseases.icd9_code = diagnoses_icd.icd9_code
INNER JOIN
    icu.admissions
ON
    diagnoses_icd.hadm_id = admissions.hadm_id
GROUP BY
    short_title,
    icd9_code,
    count_icd
ORDER BY
    count_icd DESC;

```

POSTGRESQL + DEFAULT SCHEMA (2.2 seconds)

```

WITH
    common_diseases AS (
        SELECT
            short_title,
            diagnoses_icd.icd9_code AS icd9,
            COUNT(diagnoses_icd.icd9_code) AS count_icd
        FROM
            mimiciii.diagnoses_icd
        LEFT JOIN
            mimiciii.d_icd_diagnoses
        ON
            d_icd_diagnoses.icd9_code = diagnoses_icd.icd9_code
        GROUP BY
            diagnoses_icd.icd9_code,
            short_title
        ORDER BY
            COUNT(diagnoses_icd.icd9_code) DESC)
SELECT
    short_title,
    icd9,
    COUNT(DISTINCT mimiciii.admissions.subject_id) AS number_patients
FROM
    common_diseases
LEFT JOIN
    mimiciii.diagnoses_icd
ON
    icd9 = diagnoses_icd.icd9_code
INNER JOIN
    mimiciii.admissions
ON
    diagnoses_icd.hadm_id = admissions.hadm_id
GROUP BY
    short_title,
    icd9,
    count_icd

```

```
ORDER BY
  count_icd DESC;
```

2 - QUERY : Get average number of chart events based on gender.

BQ + DENORMALIZED (1.8 sec elapsed, 2.5 GB processed)

```
SELECT
  gender,
  AVG(count_ce)
FROM (
  SELECT
    gender,
    COUNT(itemid) AS count_ce
  FROM
    star.denormalized_eoc,
    UNNEST(chartevents)
  GROUP BY
    subject_id,
    gender) a
GROUP BY
  gender
```

BIGQUERY + DEFAULT SCHEMA (5.7 sec elapsed, 4.9 GB processed)

```
SELECT
  gender,
  AVG(count_ce)
FROM (
  SELECT
    gender,
    COUNT(itemid) AS count_ce
  FROM
    icu.patients
  INNER JOIN
    icu.chartevents
  ON
    chartevents.subject_id = patients.subject_id
  GROUP BY
    patients.subject_id,
    gender) a
GROUP BY
  gender
```

POSTGRESQL + DEFAULT SCHEMA (70 Seconds)

```
SELECT
  gender,
  AVG(count_ce)
```

```

FROM (
  SELECT
    gender,
    COUNT(itemid) AS count_ce
  FROM
    mimiciii.patients
  INNER JOIN
    mimiciii.charterevents
  ON
    charterevents.subject_id = patients.subject_id
  GROUP BY
    patients.subject_id,
    gender) a
GROUP BY
  gender

```

3 - QUERY : Select chart events where month is may

BIGQUERY + DENORMALIZED (8.3 sec elapsed, 2.5 GB processed)

```

SELECT
  ce.charttime
FROM
  star.denormalized_eoc,
  UNNEST(charterevents) ce
LEFT JOIN
  star.dim_date
ON
  DATE(charttime) = dim_date.id
WHERE
  month = 5

```

BIGQUERY + DEFAULT SCHEMA (8.6 sec elapsed, 2.5 GB processed)

```

SELECT
  charttime
FROM
  icu.charterevents
WHERE
  EXTRACT(MONTH
  FROM
    charttime) = 5

```

POSTGRESQL (40 seconds)

```

SELECT
  charttime
FROM
  charterevents
WHERE

```

```
EXTRACT(MONTH
FROM
  charttime) = 5
```

4 - QUERY : Get the most commonly used drugs on patients during admissions where they have been diagnosed with pneumonia.

BIGQUERY + DENORMALIZED (1.1 sec elapsed, 90.4 MB processed)

```
SELECT
  prescription.drug,
  COUNT(DISTINCT hadm_id) num_times
FROM
  star.denormalized_eoc,
  UNNEST(prescriptions) prescription,
  UNNEST(diagnoses) diagnosis
WHERE
  LOWER(diagnosis.long_title) LIKE '%pneumonia%'
GROUP BY
  drug
ORDER BY
  num_times DESC;
```

BIGQUERY + DEFAULT SCHEMA (2.3 sec elapsed, 104.6 MB processed)

```
SELECT
  t3.drug,
  COUNT(DISTINCT t1.hadm_id) num_times
FROM
  icu.diagnoses_icd t1
LEFT JOIN
  icu.d_icd_diagnoses t2
ON
  t1.icd9_code = t2.icd9_code
INNER JOIN
  icu.prescriptions t3
ON
  t3.hadm_id = t1.hadm_id
WHERE
  LOWER(long_title) LIKE '%pneumonia%'
GROUP BY
  t3.drug
ORDER BY
  num_times DESC;
```

POSTGRESQL + DEFAULT SCHEMA (15.8 SECONDS)

```
SELECT
```

```

    t3.drug,
    COUNT(DISTINCT t1.hadm_id) num_times
FROM
    mimiciii.diagnoses_icd t1
LEFT JOIN
    mimiciii.d_icd_diagnoses t2
ON
    t1.icd9_code = t2.icd9_code
INNER JOIN
    mimiciii.prescriptions t3
ON
    t3.hadm_id = t1.hadm_id
WHERE
    LOWER(long_title) LIKE '%pneumonia%'
GROUP BY
    t3.drug
ORDER BY
    num_times DESC;

```

5 - QUERY : Get most common diagnoses that patients who get prescribed 'Hydroxychloroquine Sulfate' have.

BIGQUERY + DENORMALIZED (1.0 sec elapsed, 89.9 MB processed)

```

SELECT
    diagnosis.long_title,
    COUNT(diagnosis.long_title) count
FROM
    star.denormalized_eoc,
    UNNEST(diagnoses) diagnosis,
    UNNEST(prescriptions) prescription
WHERE
    prescription.drug = "Hydroxychloroquine Sulfate"
GROUP BY
    diagnosis.long_title
ORDER BY
    count DESC;

```

BIGQUERY + DEFAULT (1.1 sec elapsed, 104.6 MB processed)

```

SELECT
    t2.long_title,
    COUNT(t2.long_title) count
FROM
    icu.diagnoses_icd t1
LEFT JOIN
    icu.d_icd_diagnoses t2
ON
    t1.icd9_code = t2.icd9_code
LEFT JOIN
    icu.prescriptions t3

```

```

ON
  t1.hadm_id = t3.hadm_id
WHERE
  t3.drug = "Hydroxychloroquine Sulfate"
GROUP BY
  t2.long_title
ORDER BY
  count DESC;

```

POSTGRESQL + DEFAULT (0.4 sec)

```

SELECT
  t2.long_title,
  COUNT(t2.long_title) count
FROM
  mimiciiii.diagnoses_icd t1
LEFT JOIN
  mimiciiii.d_icd_diagnoses t2
ON
  t1.icd9_code = t2.icd9_code
LEFT JOIN
  mimiciiii.prescriptions t3
ON
  t1.hadm_id = t3.hadm_id
WHERE
  t3.drug = 'Hydroxychloroquine Sulfate'
GROUP BY
  t2.long_title
ORDER BY
  count DESC;

```

6 - QUERY : Simple 'select where' on chartevents.

BIGQUERY + DENORMALIZED (2.2 sec, 4.9gb)

```

SELECT
  ce.charttime
FROM
  star.denormalized_eoc,
  UNNEST(chartevents) ce
WHERE
  ce.caregiver.cgid = 15402

```

BIGQUERY + DEFAULT (2.2 sec elapsed, 4.9 GB processed)

```

SELECT
  charttime
FROM
  icu.chartevents
WHERE
  cgid = 15402

```

POSTGRESQL + DEFAULT (45 sec)

```
SELECT
  charttime
FROM
  mimiciii.chartevents
WHERE
  cgid = 15402
```

7 - QUERY : Get Average for each reading inside lab events.

BIGQUERY + DENORMALIZED (2.3 sec elapsed, 402.8 MB processed)

```
SELECT
  t1.itemid,
  label,
  fluid,
  AVG(valuenum)
FROM
  star.fact_labevent t1
LEFT JOIN
  star.dim_labitem t2
ON
  t1.itemid = t2.itemid
GROUP BY
  itemid,
  label,
  fluid
```

BIGQUERY + DEFAULT SCHEMA (1.0 sec elapsed, 402.8 MB processed)

```
SELECT
  t1.itemid,
  label,
  fluid,
  AVG(valuenum)
FROM
  icu.labevents t1
LEFT JOIN
  icu.d_labitems t2
ON
  t1.itemid = t2.itemid
GROUP BY
  t1.itemid,
  label,
  fluid
```

POSTGRESQL + DEFAULT (32 sec)

```
SELECT
```

```
    t1.itemid,  
    label,  
    fluid,  
    AVG(valuenum)  
FROM  
    mimiciii.labevents t1  
LEFT JOIN  
    mimiciii.d_labitems t2  
ON  
    t1.itemid = t2.itemid  
GROUP BY  
    t1.itemid,  
    label,  
    fluid
```