Birkbeck University of London

School of Computer Science and Information Systems

Msc Project Report - Academic year 2020

Implementing a Business Intelligence application using open-source technologies.

Supervisor: Peter Wood.

Author: Sergio Munoz Paino.

Abstract

This project is interested in how small organizations can improve their data management processes, such as stream processing and data integration using open-source tools.

The implementation of an alert system that notifies the customer services department by email will illustrate potential uses of stream processing, whilst E.T.L scripts will be implemented to integrate various data sources into a dimensional data mart, with the aim of supporting data-mining and data visualization activities.

The guiding theme during the project will be to assess whether functional Business Intelligence applications could be developed using open sources tools in the context of small-medium organizations.

These organizations may not enjoy the same kind of I.T resources more prominent organizations do, but on the other hand, tend to have a more limited and consistent set of data sources that could provide meaningful insights and improvements on operational processes.

The evaluation of results and potential limitations of this application in the broader context of Business Intelligence will be discussed in the final part of this report.

Contents

Chapter 1

Project introduction.

1.1 Report structure.

The first section of the project focuses on the design elements, including a description of the architectural elements of the application and the dimensional design process followed.

The second section focuses on the issues and decisions made during the implementation of the stream processing and E.T.L scripts responsible for the integration of the data into the data mart.

In the last part of the report, decisions made to improve the user experience, and the performance of the application will be discussed.

1.2 Chapters outline.

Project specifications justifies the changes made concerning the project proposal, introduces the business scenario and user-requirements, to finally discuss the aims of the project.

Application design justifies the choice of Agile, introduces the architecture of the application, and discusses the implementation of the dimensional data mart, including fact and dimension tables, as well as the design of the staging area.

Application implementation describes the implementation of the Alert, justifies the use of RabbitMQ and described the E.T.L processes implemented, and the issues faced.

Testing and Evaluation focuses on the changes made to the application based on user feedback and performance-tuning.

Critical evaluation of the application discusses the achievements and limitations of the application, both on its own merits and considering the current landscape of data applications.

Chapter 2

Project specifications.

This section elaborates on the changes made concerning the project proposal and introduces the context of development for the application, and the compromises it imposed on the development phase.

2.1 Modifications from the initial proposal.

Whilst Business Intelligence remains the field of interest of this project, the focus on data architectures needed a serious reconsideration following the feedback received on the proposal.

Shortly after the initial implementation began, it became clear that specific knowledge in areas such as cloud or distributed computing was needed to produce an acceptable prototype, something that could not be attained in the available time.

Related to the above difficulties, combining Kafka with node.js led to, well, "Kafkaesque" situations.

Kafka's exploit Java's multithreading nature comfortably, but it does not lend itself particularly well to Node.js single-thread nature, being considered as a combination to avoid when dealing with stream processing [12] .

Attempts at combining Java and Node.js using specific packages (such as npm node-java), did not provide satisfactory results either, this set-up being excessively straining for a system that needed to run Zookeeper, Kafka server, KsqlServer, Node.js and PostgreSQL simultaneously.

Switching Node.js to Spring Boot seemed a valid option, but had to be quickly dismissed when it became clear it would introduce too much complexity in a development process that was already proving challenging enough.

Luckily, focusing on data integration and the implementation of a Data Mart proved to be a more complex area than initially thought, that still allowed to illustrate data stream processing and their subsequent integration alongside other data sources into the dimensional repository.

2.2 Motivation for this project

Although Big Data, Business Intelligence and Data Analytics are popular terms, they are rarely associated with small-medium enterprises (S.M.E.), which are lagging implementing Business Intelligence (B.I.) [20].

The paper "Big data for small and medium-sized enterprises: a knowledge management model"[38] provides guidance, based on several case studies, on how small organizations approach the implementation of B.I. Projects.

One of the advantages small organizations have compared to bigger ones is that they are usually closer to their clients, and can quickly identify which data sources are relevant for their information needs [20].

The main recommendation, therefore, is "start small but think big", focusing on exploiting existing data sources using open-source tools, aiming at keeping the costs down.

However, since data sources tend to be "siloed" across different department [40], data integration should be considered the initial area to focus on when implementing new B.I. Projects.

Data Warehouses (D.W.) unify different data sources into a single repository, and although they are developed for large scale organizations, their principles can be adapted to smaller organizations.

One of the advantages of D.W. is that they can be modelled to optimize analytic processes, being the Dimensional Modelling particularly "well-suited" [34] for the following reasons:

- It allows users to "mix-and-match" model components in an intuitive manner.
- It facilitates "slicing" the data, keeping one or more variables constant whilst modifying the others, something that will be familiar to Excel's Pivot Tables users.

Despite the advances achieved by tools such as IBM SPSS Modeler, modelling knowledge may still be necessary [34], particularly when interacting directly with business users that need to guide developers in identifying the relevant business processes and related dimensions.

What has been discussed in this section represented the starting point for this project, i.e. *"how could be a valid prototype be developed and what modelling process should be followed?"*

2.3 Development context

The kind of application this projects develops needs a set of data sources that unfortunately were not available at the time of writing this project.

Although data-sets are freely available on websites such as Kaggle, their use-case is focused on specific analytics purposes (i.e. sales of products), and do not lend themselves to the data integration scenarios we had in mind.

Because of this, the data sources discussed in the next chapter were created from scratch so that a relevant business scenario could be developed around them.

Generating random business data proved a quite tricky and complicated task since in a sense all the different data sources needed to ”fit” together, something that had a direct impact on the volumes of data that were generated.

As such this solution does not attain the complexity of a real business scenario in terms of variety and volumes of data but could be used as a proxy to illustrate the data management processes and design practices related to data integration.

2.3.1 Business case scenario.

The medium-sized organization represented in this project offers investment advice through an online portal, supported by a dedicated team of advisors, with customers being a mix of individuals, small organizations and large companies based in different countries around Europe.

Following the first interview with the stakeholders, the following requirements were gathered:

Functional Requirements

- The organization would like to understand their customers better, for example, the department would like to know if there are significant differences between countries, types of accounts or even the services provided by the investment managers, in terms of overall satisfaction.

- Reports are currently generated by pulling and merging several files into a single one; the application should simplify this process by allowing users to access the data they need. However, not every user can work with SQL; therefore, the application should allow both technical and non-technical users to gain insights from the data.

- The customer services would like to be more proactive and contact unsatisfied customers as soon as possible, something that currently happens at the end of the day when the surveys are pulled together.

  The application should notify the department as soon as surveys with a low degree of satisfaction are received.

Non-functional Requirements

- The organization has limited financial resources at present, meaning that they are not in a position to consider further investment in technology or specialized staff.

  They would like first to assess the benefits the application could bring and made the necessary financial decisions based on the results obtained.
- The application should minimize the impact of the existing architecture and represent a compromise between functionality and maintenance needs.

2.4 Project aims.

This project's primary goal is to illustrate how open-source tools can be applied to improve data management aimed at improved decision making in small-medium organizations where data tends to be siloed across departments.

The prototype implemented focuses on data stream management and the implementation of a series of E.T.L scripts written in JavaScript to integrate MongoDB, .csv files and relational data into the data mart created in PostgreSQL.

To illustrate one of the potential data exploitation uses, this project uses the open-source tool Metabase, which allows data-querying using SQL scripts or through a drag-n-drop interface, allowing technical and non-technical users to manipulate the data, offering as well as compelling data visualization capabilities.

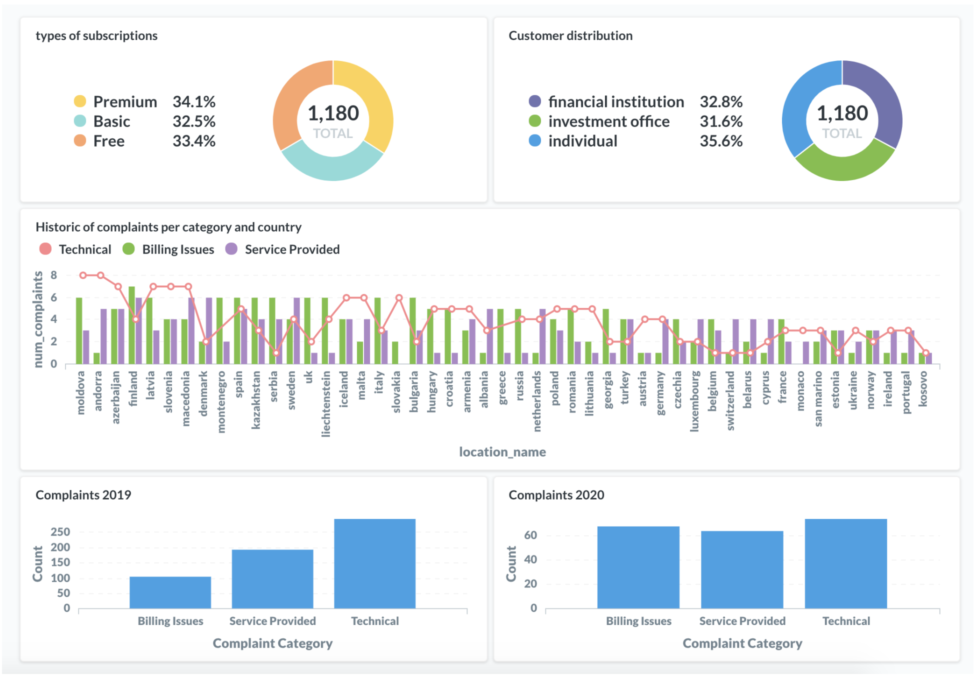


Figure 2.1: **Example of the dashboard created with Metabase** .

Chapter 3

Application design.

3.1 Development methodology

According to the consulting firm Gartner ”Between 70% to 80% of corporate business intelligence projects fail”, being the leading cause lack of understanding of user requirements, resulting in platforms with poor UX that users found unengaging or challenging to use.

Applying AGILE methodologies to Dimensional modelling reduces this risk by focusing on an incremental approach developing one start schema, testing it with the users, making the necessary adjustments and if everything went well, move to the next star schema [18].

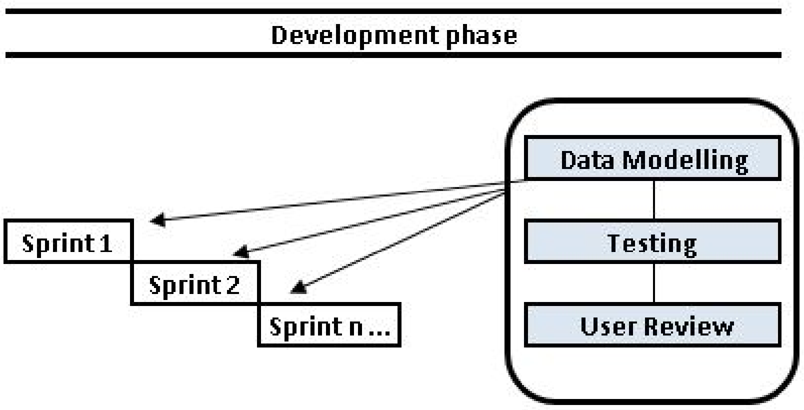


Figure 3.1: AGILE development.

Although AGILE methodologies have drawn its fair share of criticism, its suitable to ”handle unstable requirements throughout the development life-cycle” [25], and can provide quality software under time pressure when good quality assurance practices are in place.

3.2 Application Architecture overview.

The justification of the technologies used was developed in the project proposal, so as a quick reminder, the chosen technologies are well established, production tested, and concerning PostgreSQL offer good performance in both data exploration [15] and OLAP scenarios [30], especially when compared to other open-source databases such as MySQL.

The architecture presented in this project can be divided into two parts:

- The first relates to the data sources, where MongoDB is used to handle the web-application data, and PostgreSQL where most of the business entity data resides.
- The second relates to the application developed in this project, which re-uses the above elements and incorporates Node.js and RabbitMQ to perform the necessary data management processes.

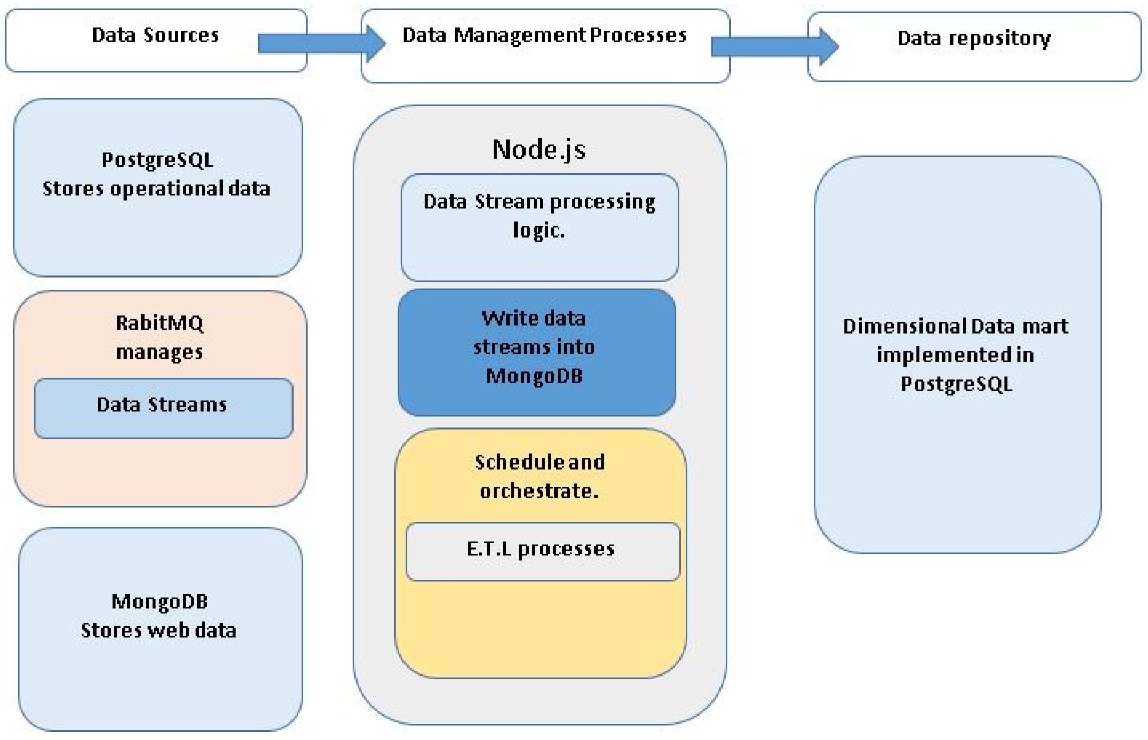

Data Sources
Data Management Processes
Data repository
PostgreSQL Stores operational data
RabitMQ manages
Data Streams
MongoDB Stores web data
Node.js
Data Stream processing logic.
Write data streams into MongoDB
Schedule and orchestrate.
E.T.L processes
Dimensional Data mart implemented in PostgreSQL

Figure 3.2: Application's architecture.

3.2.1 Application structure:

Organizing the code.

The central piece of the application is the back-end created with Node.js shown below, acting as the backbone between the data sources, the integration processes and the data mart, as well as implementing the stream processing logic.

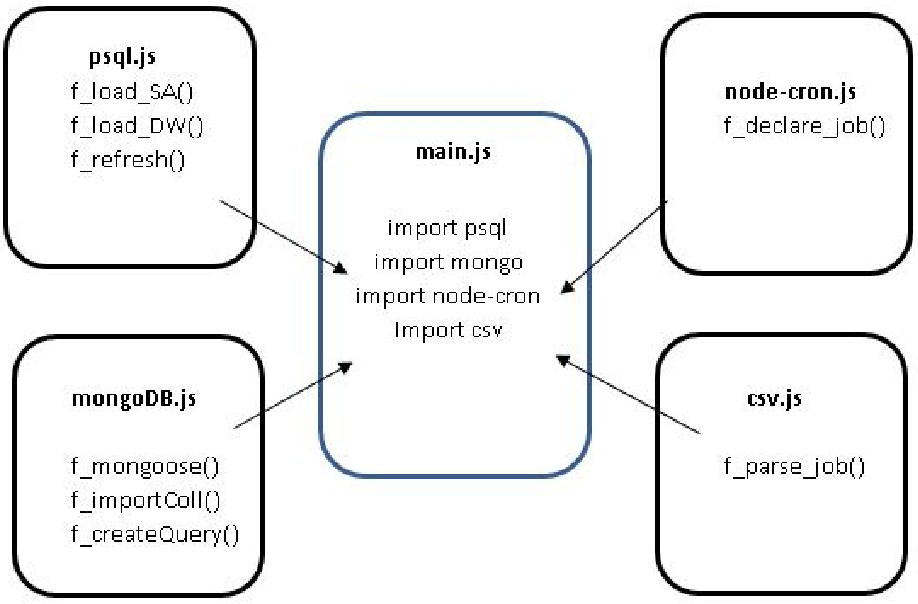


Figure 3.3: File structure.

Main.js relies on other files to be able to perform the necessary tasks, which raised the issue of tight dependencies in the code.

This was managed through the use of the ***".export"*** functionality, which, as explained in [37] enables loose coupling between the different elements that compose the application.

This allows for a more straightforward modification of the code when changes are needed, and it also improved adding new functionalities, such as different types of files to process, new databases amongst others, by simply modifying the *"import/require"* statement in main.js.

Simplifying architectures with Docker.

Since processing data streams proved harder than expected, RabbitMQ had to be added to solve the issues faced, something that will be covered in the Implementation chapter.

One of the user requirements was to minimize the impact of the application on the existing architecture, which leads to running RabbitMQ from Docker, rather than installing it locally.

The advantage of using Docker is the isolation it provides and how dependencies are managed. The operating system is only aware of Docker but does not need to manage or allocate resources to any of the underlying processes running inside the containers [24], allowing for faster and cleaner deployment.

3.3 Data sources overview.

For the project to be meaningful, a minimum variety of data sources needed to be included, as a way of illustrating how integrating data sources can provide insights that are not available when considering them in isolation.

As such, the application works with data sources that are consolidated in business environments, such as relational data and excel/.csv files, and others like JSON that are increasingly relevant for businesses implementing part of their activities online.

3.3.1 JSON.

The JSON objects created represent satisfaction surveys submitted by users through the website application; there is a total of 1,180 objects contained in the data.json file.

The application assumes that there is no utility for this data once it has been exported from MongoDB and processed, meaning it can be safely deleted from the source database.

```
{
  "object": "survey",
  "content" : {
    "reviewer_id" : 5,
    "overall_satisfaction" : 1,
    "advisor_satisfaction" : 8,
    "easiness_interface" : 5,
    "usefulness_service" : 4,
    "interest_mov_app" : 7,
    "recommend_friend" : 6
   },
  "date_submitted" : "01-01-2019"

}
```

3.3.2 Relational Data.

The below E-R diagram was generated using Schema Spy (http://schemaspy.org/), representing part of the OLTP system the organization uses to support the day-to-day activities.

Besides the visualization capabilities, SchemaSpy also offers validation tools that confirmed that no anomalies, such as *”Columns whose name and type imply a relationship to another table’s primary key”* or *”Tables with incrementing column names, potentially indicating denormalization”* were detected.

The schema creation script can be found in Annex A,.

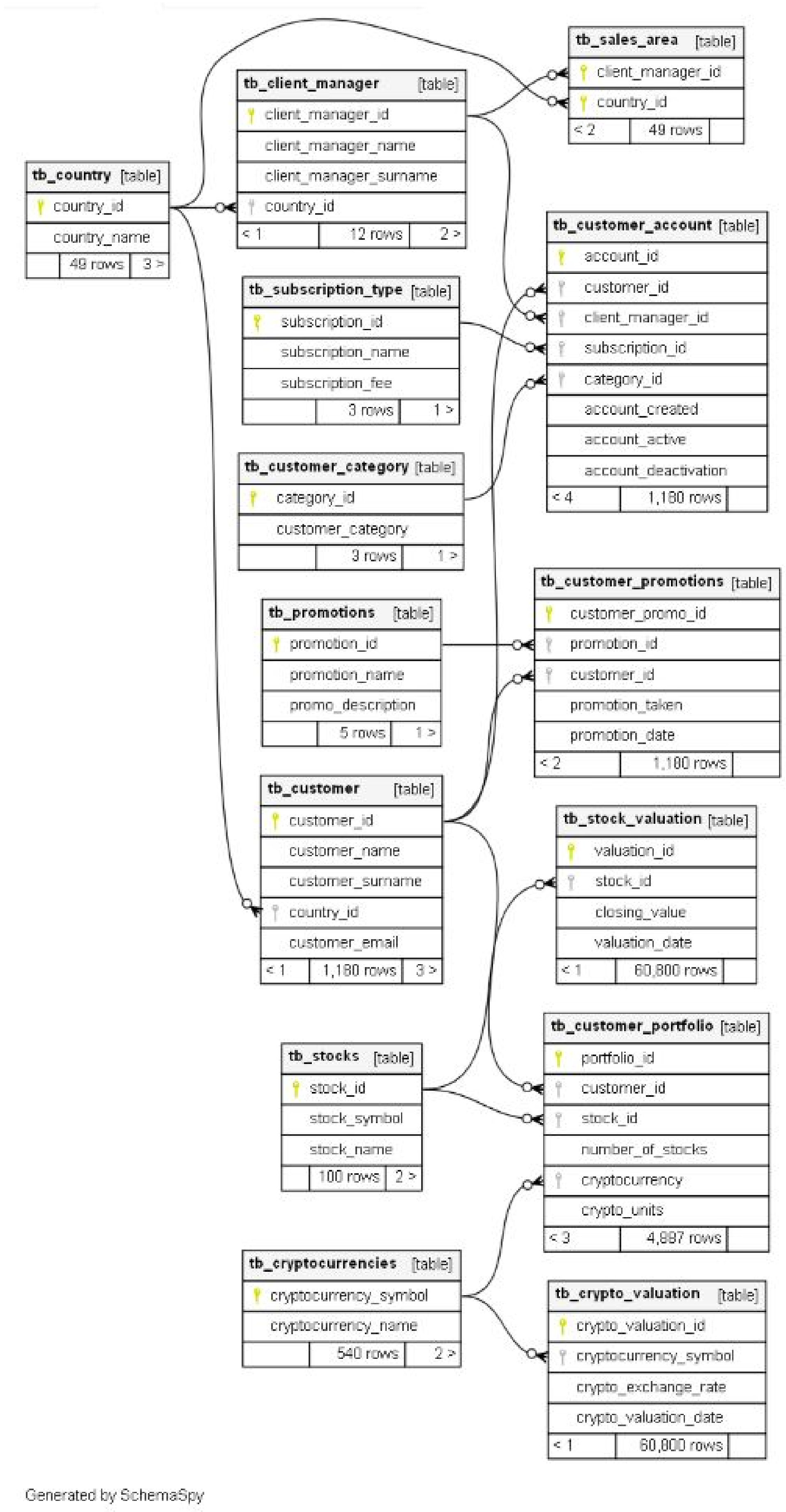


Figure 3.4: Database Design

The database is composed of 14 tables containing the more prominent business entities of the organization, such as customers, client managers, customer portfolios and promotions amongst others.

The schema was modelled to represent a third normal form (3NF) aimed at reducing data redundancy and guaranteeing referential integrity, for which the indications from [10] and several online tutorials were followed.

For example "country" in both *tb_customer, tb_client_manager* and *tb_sales_area* references *tb_country* through the use of a foreign key.

Talking about keys, although the schema contains many candidate keys, such as country, or customer category, since data had to be generated using Excel, the design favoured the use of surrogate keys represented by an incremental integer.

Because relationships between entities had to be generated randomly in Excel, using this type of Key eased the task of referencing entities via foreign Key, whilst avoiding formatting issues when generating the string chains in excel that would represent the INSERT INTO statements.

The main priority during the design phase was to create a schema complex enough to support a simple application on its own, for instance, an app to visualize investments and portfolio performances for customers across Europe, whilst still providing the necessary entities from which a dimensional model could be derived.

This is the reason why data was generated for all tables, despite not all of them being candidates for the dimensional model this project considers.

This exercise provided an excellent exercise in database design and its associated issues, since, during the process of populating the tables, various errors caused by Excel formatting issues, had to be corrected, of which the importance of using
to_date(date_source,'DD/MM/YYYY') when working with strings representing dates will never be forgotten.

3.3.3 CSV Files

This data source was chosen to represent one of the primary sources of information available in office "self-made" reports.

The .csv file represents an internal file generated by the customer services department to keep track of the different complaints and feedback submitted.

It contains 500 records, with their corresponding customer id, complaint categories, comments received and date submitted, ranging from 01-01-2019 to 31-08-2020.

The file is updated daily by a member of the customer services team and is currently used to provide some simple reports such as count of complaints per category using Excel filters and formulas.

```
customer_id, complaint_category, customer_comments, date_received
453, 1, I think the app is great and particularly fast, 2019-01-01
7, 2, I think the app isn't great nor incredibly reliable, 2019-01-01
391, 1, The service is not helpful nor particularly reliable, 2019-01-02
84, 2, The advice received isn't helpful nor particularly fast, 2019-01-02
14, 2, The service is not fast nor particularly useful, 2019-01-03
```

3.4 Data Mart implementation.

The data mart is supported by a ROLAP architecture implemented in PostgreSQL, allowing for an easier integration within the existing architecture "turning them into powerful OLAP tools with little effort" [28].

RDBMS also benefit from years of research in query optimization, detecting even star schemas "... by looking for a single large table with many smaller tables joined to it" (Joe Celko [6]), something particularly relevant for this project.

Although this architecture is not optimized for CUBE operations, for an average S.M.E, these limitations could be managed using tools such as Microsoft's SSAS, or at a smaller scale, excel's add-in Power Pivot [33], able to handle millions of rows.

In real-world applications, the data mart should be hosted in a separated server, but in this project because of hardware limitations, the most it could be done was to create a separated database.

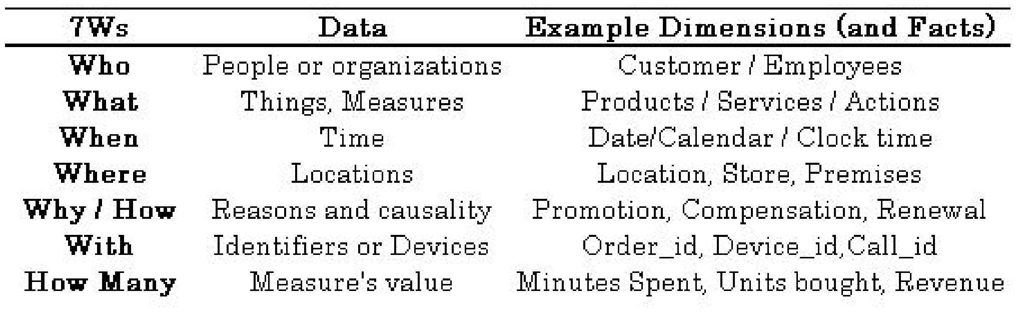
Since PostgreSQL does not natively support cross-database queries, the extension DBLINK will allow the application to show the steps needed to transfer data between foreign databases, something that will be discussed in more detail in the Implementation chapter.

3.4.1 Deriving dimensions and facts from requirements.

OLTP systems are designed to support the transaction processing within an organization, whereas data marts are designed to support a specific business process, as such their design should focus on identifying the structure of the business processes that need to be evaluated[7].

For the sake of simplicity, the user requirements were specified at the beginning of the report. However, as a way to show how dimensions and facts could have been identified in a real-world scenario, this section briefly discusses some natural language techniques, such as the "7W's" [18], that could help gather "..unambiguous, complete, verifiable, consistent and usable" [5] requirements and also help developers identify the relevant dimensions and facts.

Although Agile methodologies make use of user stories in the form of "As a user, I would like to [result] by [action] using [means]", when defining a dimensional model, the below "7ws" model shown below was considered a better tool to identify the relevant dimensions.



7Ws	Data	Example Dimensions (and Facts)
Who	People or organizations	Customer / Employees
What	Things, Measures	Products / Services / Actions
When	Time	Date/Calendar / Clock time
Where	Locations	Location, Store, Premises
Why / How	Reasons and causality	Promotion, Compensation, Renewal
With	Identifiers or Devices	Order_id, Device_id,Call_id
How Many	Measure's value	Minutes Spent, Units bought, Revenue

Figure 3.5: 7W's model as seen in Agile Data Warehouse Design.

From the above table and considering the user requirements described in section 2.3.1, the following dimension candidates were identified:

"Who" is represented by the customers and the investment managers.

"Where" is represented by the location.

"When" could be split into two potential dimensions,i.e. physical time (seconds, minutes...) and calendar date (day, week...). For this project, only the date dimension will be considered for analysis purposes.

"What" it is composed of the type of subscription (free, basic, premium) and the account category (individual, financial institutions..).

"How many" will represent the measures of interest, i.e. what is the minimum, maximum or average satisfaction?. How many complaints have been sent?. Was the promotion taken or not?.

3.4.2 Star Schema Design.

Kimball's approach to the construction of a D.W represents a more manageable approach, especially when compared to Immon's [16], for organizations that are starting to define their information needs, since these can be satisfied "one star schema at a time".

This is enabled by the used of conformed dimensions that could be re-used across several fact tables, in what is known as the Data Mart Bus Architecture [3].

As a guide during the star schema design phase, the steps described in [1] were followed to implement the star schemas.

The above provided the necessary guidance to develop the dimensional and fact tables, although, for the sake of simplifying the report, some of the steps will be combined into a single one.

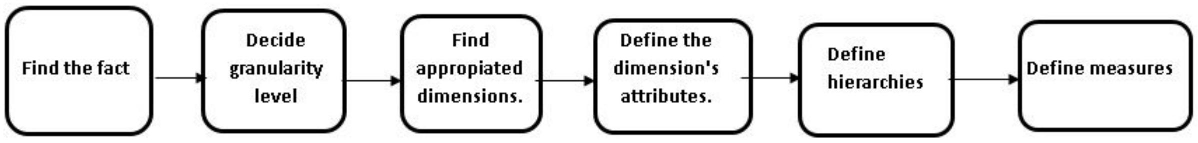


Figure 3.6: Design process from Diseño Multidimensional.

Finding the facts and measures

The facts, or quantitative measures, were easy to identify from the user requirements.

The Marketing and customer services department needs to ***"count"*** the amount of ***complaints*** and analyze how this count varies depending on the chosen dimensions.

The above can also be applied to the amount of ***promotions*** taken, i.e. ***"count"*** which ones were more popular.

Finally there is also an interest in finding the "degree" of ***satisfaction*** the customers have i.e. ***min,max,avg***.

As a side note, another fact table could have been generated from the comments section contained in the .csv file, which could have then been exploited by the use of word clouds, but because of the length of the comments, and the fact that creating this fact table would not have added anything new in terms of design and processes followed, it was decided not to include it in the final project.

Granularity level.

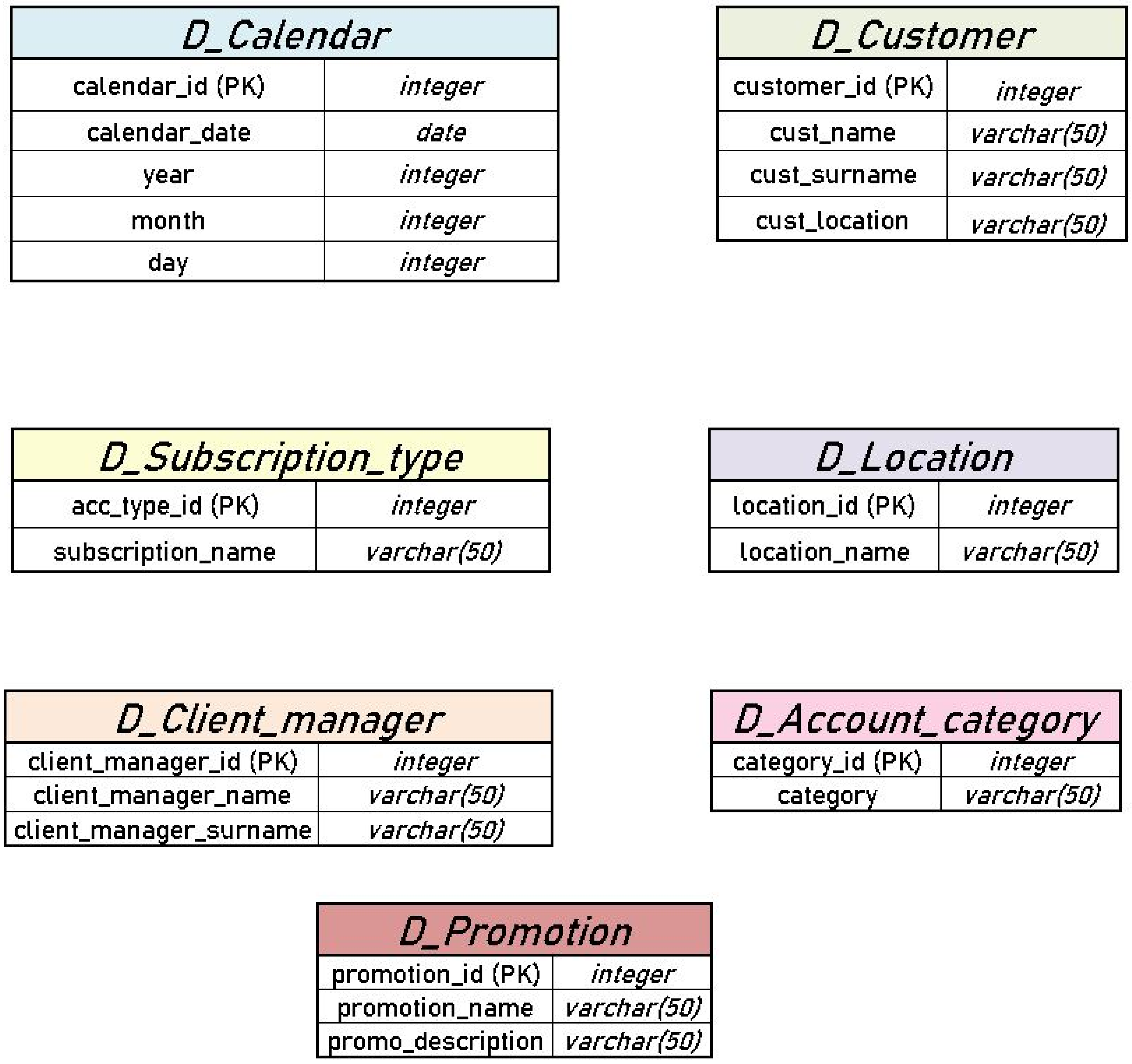
The level of granularity has consequences both in the database size and the measures that can be calculated. Choosing a low grain would increase the volumes of data to be stored; on the other hand, a larger grain may mean some measures cannot be calculated.

In this project, the level of granularity of the analysis was set at days (i.e. daily reviews).

As briefly discussed before, no hourly or time analysis was required at this stage.

Dimensions and attributes.

Considering what has been discussed so far, the following dimensions and attributes were identified, represented in the below logical designs:



D_Calendar	
calendar_id (PK)	*integer*
calendar_date	*date*
year	*integer*
month	*integer*
day	*integer*

D_Customer	
customer_id (PK)	*integer*
cust_name	*varchar(50)*
cust_surname	*varchar(50)*
cust_location	*varchar(50)*

D_Subscription_type	
acc_type_id (PK)	*integer*
subscription_name	*varchar(50)*

D_Location	
location_id (PK)	*integer*
location_name	*varchar(50)*

D_Client_manager	
client_manager_id (PK)	*integer*
client_manager_name	*varchar(50)*
client_manager_surname	*varchar(50)*

D_Account_category	
category_id (PK)	*integer*
category	*varchar(50)*

D_Promotion	
promotion_id (PK)	*integer*
promotion_name	*varchar(50)*
promo_description	*varchar(50)*

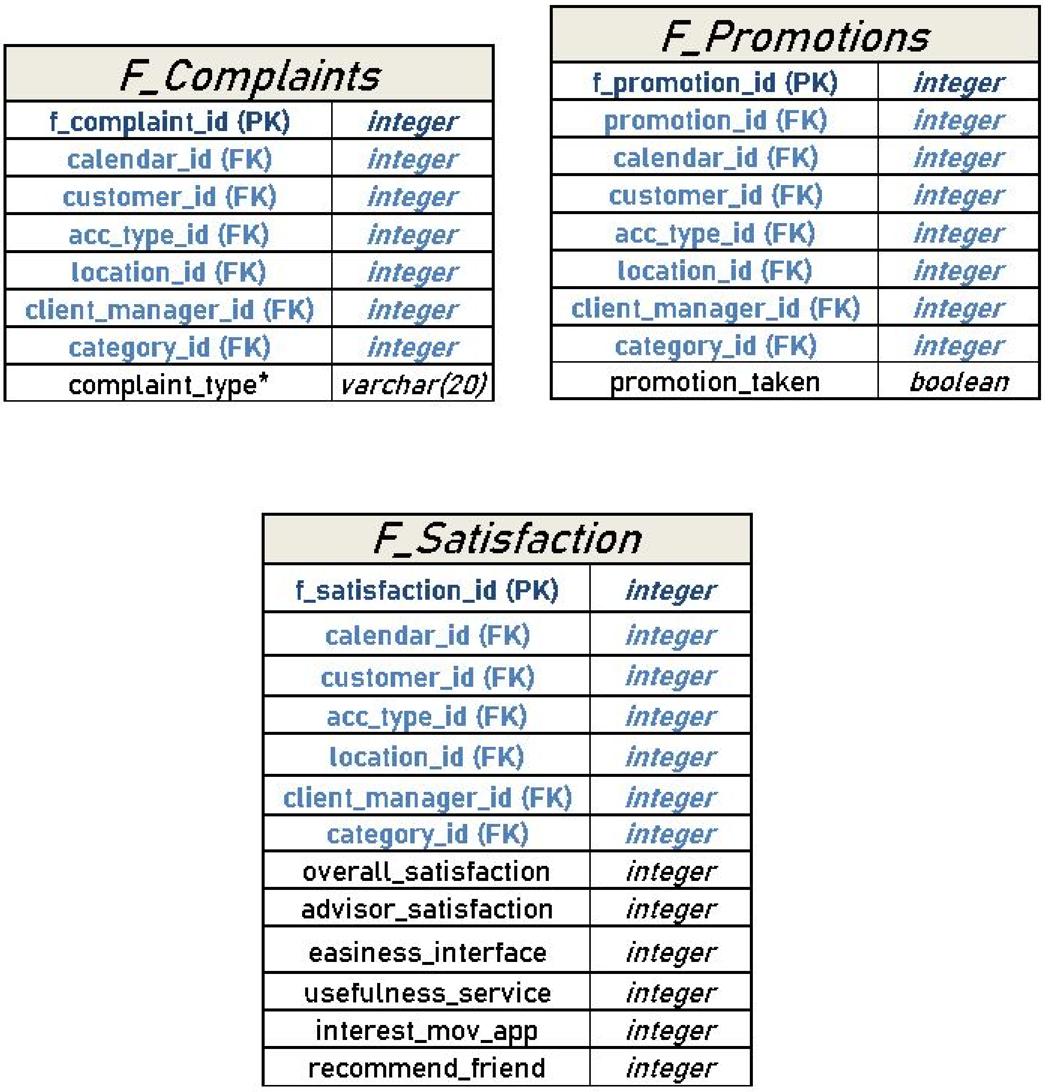
Figure 3.7: Dimensions Identified.

Define Hierarchies.

Because of the nature of the business, it was considered that hierarchies of the type Country > Region > City were meaningless, but Year > Month > Day could be useful for long-term analysis purposes.

3.4.3 Fact table logical design.

To simplify this section, and since the dimensions and attributes have already been described, this section will only show the logical design of the fact tables.



F_Complaints	
f_complaint_id (PK)	integer
calendar_id (FK)	integer
customer_id (FK)	integer
acc_type_id (FK)	integer
location_id (FK)	integer
client_manager_id (FK)	integer
category_id (FK)	integer
complaint_type*	varchar(20)

F_Promotions	
f_promotion_id (PK)	integer
promotion_id (FK)	integer
calendar_id (FK)	integer
customer_id (FK)	integer
acc_type_id (FK)	integer
location_id (FK)	integer
client_manager_id (FK)	integer
category_id (FK)	integer
promotion_taken	boolean

F_Satisfaction	
f_satisfaction_id (PK)	integer
calendar_id (FK)	integer
customer_id (FK)	integer
acc_type_id (FK)	integer
location_id (FK)	integer
client_manager_id (FK)	integer
category_id (FK)	integer
overall_satisfaction	integer
advisor_satisfaction	integer
easiness_interface	integer
usefulness_service	integer
interest_mov_app	integer
recommend_friend	integer

Figure 3.8: Fact tables.

Except for the dimension d_promotions, the rest are shared across all fact tables, something that allows the analyst to perform "drill-across" queries [36], for instance joining the results of queries against F_complaints and F_satisfaction to see if the amount of complaints received has any relationship with the satisfaction measures recorded.

3.4.4 Staging Area.

The staging area is an important part of the design of the application and will serve multiple purposes:

- Represent the initial loading point for MongoDB and .csv data.
- Simplify some of the E.T.L processes by performing the final data transformation stage before the data being loaded in the staging tables. For example modifying the strings

representing dates, to *"date"* data types that PostgreSQL recognizes using *to_date()* function.

- The fact that the staging tables mirror the attributes and data types of those in the data mart provides a first safety check before the data is loaded in the destination tables.

The staging area will also contain some "helper" tables to overcome the limitations of the extension DBLINK related to the lack of support for WHERE clauses,i.e. "dblink fetches the entire remote query result before returning any of it to the local system" [8], something that will be particularly useful when dealing with the incremental load of the data mart.

Chapter 4

Application implementation.

This section beings describing the process followed to implement the customer service alert system, comprising the implementation of the pseudo-real-time system, messages queues, event logic and data storage into MongoDB.

The second part will focus on the implementation of the diverse E.T.L processes implemented for each data source, populating the staging area tables and final data load into the data mart tables.

4.0.1 Stream Processing

Although the uses of data streams in B.I. are numerous, including fraud detection, real-time recommendation systems, amongst others, this section will focus on implementing alert functionality and writing streams to MongoDB as outlined below:

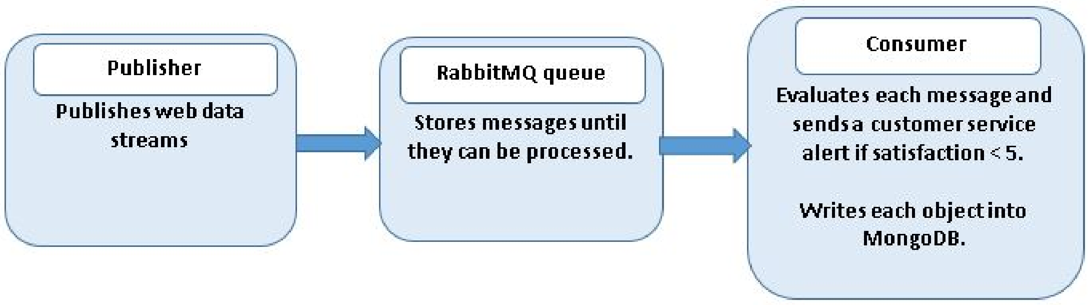

Publisher
Publishes web data streams
RabbitMQ queue
Stores messages until they can be processed.
Consumer
Evaluates each message and sends a customer service alert if satisfaction < 5.
Writes each object into MongoDB.

Figure 4.1: Stream processing diagram.

4.0.2 Potential issues with stream processing.

When processing data streams, situations such as servers crashing, sudden traffic spikes or asynchronous data processes should be considered to ensure data integrity.

A message queue can be a helpful tool when dealing with the above issues. Amongst their many uses [23], the relevant ones for this project were:

- **Traffic Spikes:** Although the volumes of data this application works with are limited, the fact that they are produced at random intervals can generate sudden data spikes that could lead to application crashes.
- **Asynchronous Messaging:** A message queue will allow for the implementation of an asynchronous pattern, something that will be needed considering the amount of write-to-database operations that need to be carried by the application.

4.0.3 RabbitMQ.

During the research phase carried as part of the proposal, RabbitMQ and Kafka were the main options considered. As explained in the introduction, the issues experienced with Kafka and following the trend of unexpected changes this year has brought upon us, RabbitMQ made it to the final implementation.

This said, both technologies are not equivalent and serve different purposes [22], something that will be discussed during the evaluation chapter.

4.0.4 Defining a publisher.

The first step to implement the alert functionality was to create a publisher following the indications from RabbitMQ documentation [41].

```
const amqp = require("amqplib");
const data =  require("./data.json");
async function publisher(){
    try {
    const connection = await amqp.connect("amqp://localhost:5672");
    const channel =  await connection.createChannel();
        // Create queue if it doesn't exists.
    const queue = await channel.assertQueue("web-data", {durable: true
       });
        for(const message in data) {
                setTimeout(() => {
                let object = data[message];
                channel.sendToQueue("web-data",
                Buffer.from(JSON.stringify(object)),
                {persistent: true, contentType:'application/json'});
                }, randomIntervalTime());
        }
    }
    catch (e) {
        console.log(`The following error has been found:\n${e}`);
    }}
```

The above code initializes the connection and creates a queue named "web-data". To prevent data losses, the options **{durable: true}** that prevents queues from being lost in case of a server shutdown, and **{persistent: true}** that preserves messages in the event of failure, were enabled.

Instead of working with HTTP requests, the file containing the JSON objects was used to generate the pseudo-random-data combining a for loop and the ***setTimeOut()*** function, that generates random intervals between 1 and 60 seconds, at which each object is sent to the queue.

It must be said that in more complex scenarios the publisher should not be sending messages directly to a queue, but to an exchange, where they can be filtered and sent to the appropriated queue based on header attributes or binding keys [21].

However, since this application only has one type of message "surveys", sending messages directly to the queue is an acceptable decision in the context of this project.

4.0.5 MongoDB shared connections and models.

One of the (many) stumbling blocks of the stream processing section was coordinating the processes of publishing data, receiving it and writing it to MongoDB.

Although connecting to MongoDB was very straightforward, sharing the connection across the application using native MongoDB drivers proved to be an excruciating process with many unsuccessful attempts.

The Object-Document-Mapper (O.D.M) Mongoose [27] solved this issue since it bounds any object it creates to a single connection, sharing it across the application thanks to the module.export functionality in Node.js.

```
// Any query or MongoDB operation carried out with Mongoose will be
   bound to the below connection to the web-data database.

mongoose.connect("mongodb://127.0.0.1:27017/web-data",
{ useNewUrlParser: true, useUnifiedTopology: true, useCreateIndex: true
   });
mongoose.connection.on("open", function(){
  console.log("Connection to MongoDB stablished.");
});
```

Unlike native MongoDB, Mongoose needs to define a (flexible) schema or "model" to which every objects needs to conform, before it can be written in the database.

The below code defines a model that will write data to the collection "survey" (in MongoDB it will be automatically pluralized and appear as "surveys"), contained in the "web-data" database specified above.

```
let schema = mongoose.model('survey', {
    reviewer_id: { type: Number, required: true},
    overall_satisfaction: { type: Number, required: true},
    advisor_satisfaction: { type: Number, required: true},
    easiness_interface: { type: Number, required: true},
    usefulness_service: { type: Number, required: true},
    interest_mob_app: { type: Number, required: true},
    recommend_friend: { type: Number, required: true},
    date_submitted: {type: String, required: true}}
);
module.exports = schema;
```

Now that the model is created, we can pass objects to the schema constructor, and store them in MongoDB using the ***save()*** function as seen below.

```
const model = new Schema({key: value, key: value...})
model.save().then(()=> console.log("Data has been written to MongoDB"))
```

4.0.6 Implementing the Event Logic within the subscriber.

When implementing the event logic, two options were considered, i.e. using MongoDB's functionality Change Stream or push the logic inside the consumer with JavaScript code.

Initially, MongoDB'S Change Streams seemed promising. The consumer needed to write the data to MongoDB, and the database would automatically perform the event-logic evaluation.

However, the idea was dismissed for the following reasons:

- Pushing the event logic to the database would add an extra overhead when debugging or refactoring was needed.
- Change streams can only be implemented through a replica, which means that another instance of MongoDB needed to be created [31], increasing the overall demands of the application.

The event logic was a simple if...then block, but not having previous experience with RabbitMQ; it was not clear how pushing the logic to the consumer would affect performance.

As feared, the initial attempts at implementing the logic failed until the following article [13] and several StackOverflow questions provided the solution, namely ***channel.prefetch(1).*** should be used to notify the engine not to send any more messages in the queue until the consumer had acknowledged the last one.

This implementation seemed to work fine, but during the testing phase, it was discovered that a small fraction of the data (3-4 objects on average) was lost. For the sake of completeness, the final code will be shown here, but the steps taken to solve this issue will be discussed in the Testing section.

```
const amqp = require("amqplib");
const Document = require("./model");
async function consumer(){
    try {
        const connection = await amqp.connect("amqp://localhost:5672");
        const channel =  await connection.createChannel();
        const queue = await channel.assertQueue("web-data");
        channel.prefetch(1);
        channel.consume("web-data", message => {
            let data = JSON.parse(message.content.toString());
            let survey = new Document({
        reviewer_id: data["content"]["reviewer_id"],
        overall_satisfaction: data["content"]["overall_satisfaction"],
        advisor_satisfaction: data["content"]["advisor_satisfaction"],
        easiness_interface: data["content"]["easiness_interface"],
        usefulness_service: data["content"]["usefulness_service"],
        interest_mob_app:  data["content"]["interest_mob_app"],
        recommend_friend: data["content"]["recommend_friend"],
        date_submitted: data["date_submitted"]
        });
        // Send an email to customer-services if the condition is met.
        if ( data["content"]["overall_satisfaction"] < 5){
        customerNotification(String(data["content"]["reviewer_id"]))
        };
        // Write to MongoDB function and acknowledge message only if
            successful.
        survey.save(function(error,object){
            if(!error){
            channel.ack(message);
              }
            })
        })
    }

    catch (e) {
        console.log(`The following error has been found: ${e}`);

    }
}
```

The function ***customerNotification()*** uses the email delivery service SendGrid, providing a registered email address needed to test this functionality with GMAIL. This function will alert the customer service department whenever the overall satisfaction of a particular customer is below 5.

SendGrid has a Node.js module that made the implementation of the email notification pretty straightforward once the API key had been registered in the sgMail variable.

The implementation of the notification service is as follows (for testing purposes our BBK address was used):

```
function customerNotification(customer_id){

    sgMail.send({
        to: "customer-services@investment-manager.com",
        from : "survey-manager@investment-manager.com",
        subject: ‘Please contact the customer number ${customer_id}.‘,
        text: ‘Please contact customer ${customer_id} and follow the
           guidelines discussed in the last meeting.‘

    })
}

module.exports = customerNotification;
```

4.1 Defining ETL processes

This section will focus on the data processes carried to transfer curated data into the data mart, describing the process for each data source, i.e. MongoDB, the .csv file and the relational data contained in the OLTP system.

4.1.1 Pitfalls with asynchronous operations.

The most significant issues faced during the implementation stage were related to the management of asynchronous processes within the application, something that caused numerous issues during the different stages of the E.T.L. process.

The issues caused by node.js non-blocking nature and the solution implemented to manage asynchronous processes will be briefly discussed in this section to avoid repetition throughout the different sections in this chapter.

The main problem was ”How can the application make sure that all the data is available in PostgreSQL, before the process of loading the dimensional and fact tables begins?”.

The initial implementation consisted of declaring two arrays, one for MongoDB and another for the .csv data. Then a function call to get the MongoDB or the .csv data was performed, and by looping through the results the arrays were filled, being subsequently passed as an argument to a function that sent the data to PostreSQL.

This worked fine until the volumes of data were increased during the testing phase, at which point the application began raising ”parse” exceptions.

Once it became clear that the application needed to manage asynchronous processes, the research led to the following potential solutions ”callbacks, promises or async/await”.

Callbacks were not a good candidate since the application could end up with a deeply nested set of callback functions, also known as "callback hell". Although "async/await" offered a less verbose implementation, "promises" were favoured since the process the application had to follow was clearer using a chain of *.then().then()* methods.

The above sequence of *.then()* methods represent the first "naive" attempt at implementing a promise chain. This quickly proved to be the wrong approach and shown that promise chains are not implemented by simply chaining several *.then()* together.

AS the M.D.N. documentation states, promises need to use the return value of the previous method, and if the object returned is a promise then " ..gets dynamically inserted into the chain." [11]

This was the critical aspect to consider when implementing the methods to process MongoDB, .csv data and also to link the different data processes in PostgreSQL using the P.G. client.

Having explained the issues faced with asynchronous code, the next sections will focus on the options considered to process each data source, and show the final implementation of the code.

4.1.2 Mongo.db data

To transfer the data from MongoDB two options were considered

- Perform a data dump and bulk load the data into PostgreSQL.
- Use JavaScript code to iterate through all the objects, store them in a variable, and pass it as part of a query in P.G. client.

Whilst the first option offered a better performance; the data was exported in binary format [26] meaning that further processing was needed before inserting the data in the relational tables.

Because of this complexity, the second option was chosen, and the below promise-based method was developed:

```
function getMongoData(){
    return new Promise((resolve,reject)=>{
        const collection =  Document.find({})
        if(collection){
            resolve(collection)
        }else{
            reject(Error("No data was found in MongoDB"))
        }
    })
}
```

4.1.3 CSV Files.

For the .csv file a similar promise-based function was defined:

```
function getComplaints(){
    return new Promise((resolve,reject)=>{
        const csvData = [];
        fs.createReadStream('complaints.csv').pipe(csv())
        .on('data',(data)=> {
            csvData.push(
                [Number(data['customer_id']),
                Number(data['complaint_category']),
                data['customer_comments'].replace(/[\W_]+/g," "),
                String(data['date_received'].trim())])
            })
        .on('end',()=> {
            if(csvData){
                resolve(csvData)
            }else{
                reject(Error("No data was found in the .csv"))
            }
        });
    })
    }
```

Finding how to parse a .csv file was easy given the amount of tutorials available, the use of ***.createReadStream()*** and ***.pipe()*** were implemented to prevent potential memory issues when handling large files.

Using streams allows the application to process data as soon as it's available, instead of having to load the whole file in memory [19].

Dealing with semi-structured data in form of files is pretty much a "mine-field" since there is formatting guarantees, as such and although the data from the .csv file is pretty "clean" we have applied REGEX, ***.trim()*** and cast the values to the appropriated data type to make sure no formatting errors were generated by the data.

4.1.4 Data integration process in PostgreSQL.

The goal of this section is to consolidate the data sources into PostgreSQL, so the dimension and fact tables can be populated.

Since PG, PostgreSQL and PgAdmin will be mentioned quite often during this section, it needs to be clarified that PG refers to the application that allows communicating with PostgreSQL database via Node.js, whilst PgAdmin refers to the graphical interface on which some of the queries were tested and the results inspected.

The operations needed in this stage consist mainly of SQL statements, such as "SELECTS" and "INSERT INTO", because of this little code will be shown in this section.

In case the reader needs to check a particular step, the code is available in Appendix D.

As discussed before, not understanding how promises are chained caused several connection issues. In the initial implementation, the application generated a new client each time a query was performed, exhausting PostgreSQL resources which caused the database server to shut down.

Those issues were eliminated once the next query to be performed by the application was included as the return value of the function as shown below:

```
getMongoData() // returns a promise
.then((collection) =>{
     collection.forEach(doc => mongoArray.push(doc))
                return getComplaints() // returns a promise
                    })
.then((csvRows) =>{
    csvRows.forEach( row => csvData.push(row))
                return pg.connect() // returns a promise
                    })
.then( result=>{
    console.log("Connection established")
               return pg.query(...) // returns a promise
                    })
.then( result=>{ // perform next query...})
```

Unlike the previous data sources, pg supports promises out of the box, and as such, the only consideration needed was to make sure the return value of each .then() function was a promised-based function.

Consolidating Data sources.

Once the arrays containing the data for the .csv files and MongoDB had been populated and the connection to PostgreSQL established, the next step was to combine all data sources.

Following the advice given in the module Data Knowledge Management, a set of rows containing all the tuples needed had to be created first, so the necessary data could be fetched by performing simple "SELECT" statements. The steps to follow are then:

- Bring MongoDB and .csv data into their corresponding PostgreSQL tables.
- Combine all data sources to create a unique set of tuples.
- Select the necessary tuples to populate the staging dimension tables.
- Populate staging fact tables by referencing the appropriated FK from the dimension tables.

- If no errors had occurred, send the data to the final Data Mart tables.

A common table expression (CTE) was the first option considered, but since they only persist for the duration of the query, the application would have to recreate it as many times as INSERT statements were needed, something that was not an acceptable solution.

A temporary table seemed the best option after having found similar issues in StackOverflow, since unlike CTE, temporary tables are session persistent, being available for as long as the connection to the client is open.

Building the temporary table consisted in joining several tables from the OLTP system, such as tb_customer_account, tb_customer, tb_promotions... on their common attributes, and then perform two left joins for surveys and complaints on their common customer/reviewer id's.

```
CREATE TEMPORARY TABLE temp_table AS(
    SELECT  DISTINCT ca.customer_id, c.customer_name, c.
       customer_surname,co.country_id, co.country_name, cm.
       client_manager_id, cm.client_manager_name, cm.
       client_manager_surname, ca.subscription_id, st.subscription_name
       , ca.category_id, cc.customer_category, com.complaint_category,
       to_date(com.complaint_date,'YYYY/MM/DD') as complaint_date,
    s.overall_satisfaction,s.advisor_satisfaction, s.easiness_interface
       , s.usefulness_service, s.interest_mob_app,s.recommend_friend,
       to_date(s.survey_rec_date,'DD/MM/YYYY') AS survey_date
    FROM investment_manager.tb_customer_account ca
    JOIN investment_manager.tb_customer c
        ON ca.customer_id = c.customer_id
    JOIN investment_manager.tb_country co
        ON c.country_id = co.country_id
    JOIN investment_manager.tb_client_manager cm
        ON cm.client_manager_id = ca.client_manager_id
    JOIN investment_manager.tb_subscription_type st
        ON ca.subscription_id = st.subscription_id
    JOIN investment_manager.tb_customer_category cc
        ON ca.category_id = cc.category_id
    LEFT JOIN staging_area.sa_complaints com
        ON com.customer_id = ca.customer_id
    LEFT JOIN staging_area.sa_surveys s
        ON s.reviewer_id = ca.customer_id
    ORDER BY customer_id asc)
```

In the above step, SQL formula ***to_date()*** was used to modify the strings representing dates, in the .csv and mongo data, to actual date data types. The date format in the .csv file and the survey's objects were different, to avoid adding an extra step, and considering dates remain the same regardless of their style the original format was left.

Populating Dimension and Fact tables.

Once the temporary table had been created, the application only needed to populate the tables using several ”***INSERT INTO d_table(field1,field2..) SELECT DISTINCT col1,col2***”

Dimension tables need to be filled first then fact tables, to make sure the necessary foreign key references have been generated.

To reduce the number of lines of code needed in the main file, the queries related to the creation of the temporary table and all other tables contained in the staging area, were stored in a separated file ***”sqlQueries.js”*** and exported as an object to the main file, which allowed the application to perform the query needed as follows ***client.query(format(saQueries.insertCategory))***.

Populating the dimension tables was a straightforward process until the limitations of PG working with serial types were discovered.

Although the queries worked fine when being executed in PgAdmin, the fact that the column referring to the serial data-type field was not included in the ***INSERT INTO tb_name(col2,col3..)*** statement caused a parsing error.

PG's documentation did not offer any solution solve this issue, and since the queries worked without any problem when being executed through PgAdmin, a set of stored procedures was defined, meaning that d_calendar and all the fact tables would have to be populated calling a store procedure.

For illustration purposes we will show the stored procedure defined to populate the complaints fact table:

```
CREATE OR REPLACE FUNCTION staging_area.fill_f_complaints()
RETURNS VOID AS $$
BEGIN
INSERT INTO staging_area.SA_F_COMPLAINTS(calendar_id,customer_id,
   subscription_id,location_id,
client_manager_id,category_id,complaint_category)
SELECT DISTINCT CAL.CALENDAR_ID,CUSTOMER_ID,subscription_id,country_id,
client_manager_id,category_id,complaint_category
FROM temp_table t, staging_area.sa_d_calendar cal
WHERE CAL.CALENDAR_DATE = COMPLAINT_DATE
AND complaint_category IS NOT NULL;
END;
$$ LANGUAGE PLPGSQL;
COMMIT;
```

By doing so, when working with PG, the application will simply perform the following query ***client.query(staging_area.fill_f_complaints()))*** bypassing the previous parsing issues.

The same process was followed for the fact tables, creating three different functions for each fact table.

4.1.5 From Staging area to D.W. tables

Now that the staging tables have been populated the next step is to load them into the data mart. Since the transfer targes a foreign database, the queries will have to be modified to reflect this, and a new connection to the destination database needs to be created.

To connect to the new database, a new client is created, and then simply call ***DW.connect()*** after which point the application can perform the same kind of operations but this time using ***DW.query()*** instead of ***client.query()***

As discussed previously, PostgreSQL does not natively support queries against foreign databases, and as such, the extension DBLINK had to be created in the destination database.

Because of this the queries had to be modified as shown below:

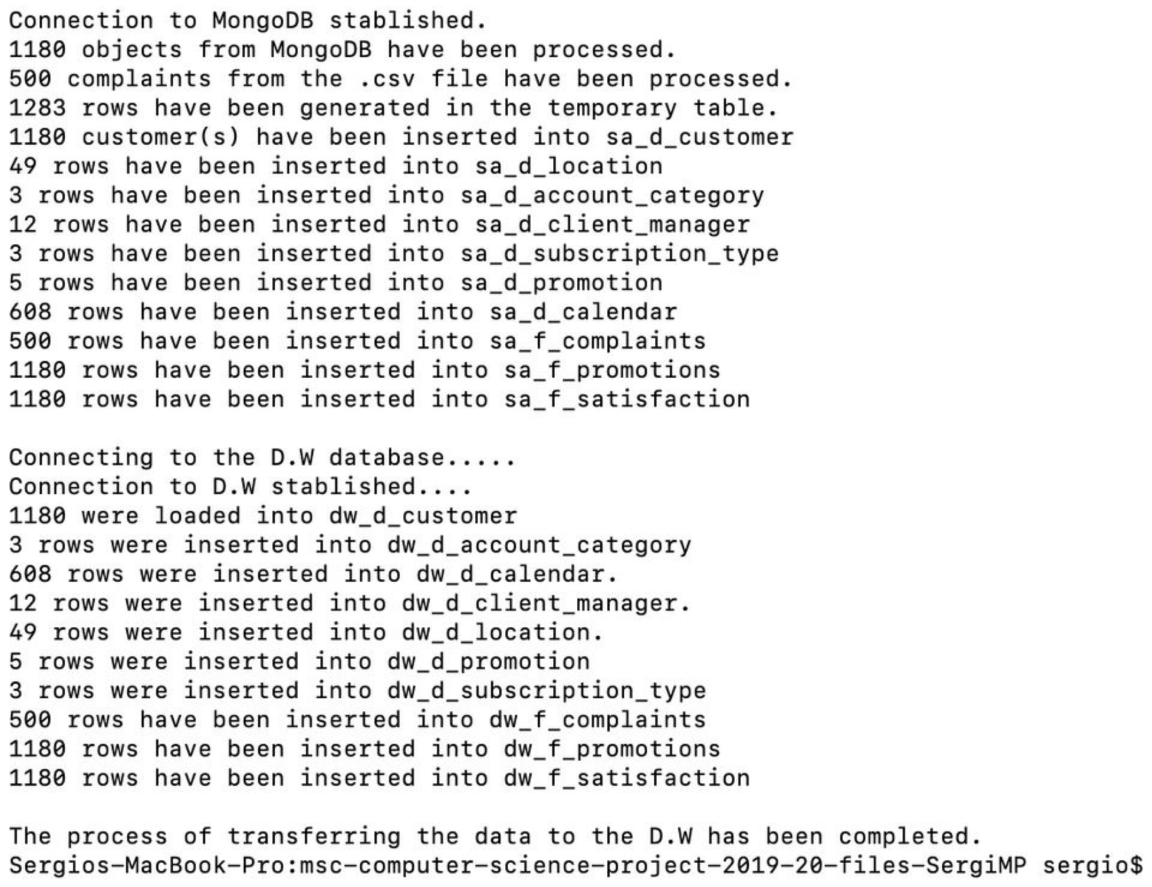
```
INSERT INTO DATA_MART.DW_D_account_category(category_id,
   account_category)
            SELECT *
            FROM DBLINK('host = localhost
                        user = postgres
                        password = postgres
                        dbname = investor',
    'SELECT * FROM staging_area.sa_d_account_category') as
    linktable(a int, b varchar(50))
```

Each query needs to reference all the necessary details from the source database, also specifying the fields and data types in the link table.

Despite being very verbose, the implementation of DBLINK is much more straightforward than the other foreign data wrappers, and since the only operations that need to be performed are INSERT operations, verbosity is an acceptable trade-off.

From this point, the application only needs to perform a series of INSERT into the data mart table to complete the initial load, at which point all the connections to the database will be closed.

The below image shows the different processes the application performs, from connection to databases, ETL processes to the final data-load into the data mart.



```
Connection to MongoDB stablished.
1180 objects from MongoDB have been processed.
500 complaints from the .csv file have been processed.
1283 rows have been generated in the temporary table.
1180 customer(s) have been inserted into sa_d_customer
49 rows have been inserted into sa_d_location
3 rows have been inserted into sa_d_account_category
12 rows have been inserted into sa_d_client_manager
3 rows have been inserted into sa_d_subscription_type
5 rows have been inserted into sa_d_promotion
608 rows have been inserted into sa_d_calendar
500 rows have been inserted into sa_f_complaints
1180 rows have been inserted into sa_f_promotions
1180 rows have been inserted into sa_f_satisfaction

Connecting to the D.W database.....
Connection to D.W stablished....
1180 were loaded into dw_d_customer
3 rows were inserted into dw_d_account_category
608 rows were inserted into dw_d_calendar.
12 rows were inserted into dw_d_client_manager.
49 rows were inserted into dw_d_location.
5 rows were inserted into dw_d_promotion
3 rows were inserted into dw_d_subscription_type
500 rows have been inserted into dw_f_complaints
1180 rows have been inserted into dw_f_promotions
1180 rows have been inserted into dw_f_satisfaction

The process of transferring the data to the D.W has been completed.
Sergios-MacBook-Pro:msc-computer-science-project-2019-20-files-SergiMP sergio$
```

Figure 4.2: Initial Data Transfer.

4.2 Incremental Load

Now that the initial data transfer has been completed, the goal is for the application to process only new data not present in the data mart.

Dealing with changes in data raises a common design consideration we have not discussed yet, i.e. ”Slowly Changing Dimensions”, what happens if, for instance, a customer changes his country of residence? or whether a customer category or promotion is renamed?. Those modified values would not be present in the data mart, and as such, those modifications should be reflected.

Initially, it was considered to include an extra column in some of the dimension tables to refer to the ”previous value column” as see in [39], but considering the amount of time available and the many issues faced when creating the data, data historicity was sacrificed and assumed not to be a crucial aspect of the prototype.

A possible solution to deal considering the above decision would have been defining an AFTER UPDATE trigger, but since this would be linked to the OLTP system logic, its implementation will not be discussed as part of this project.

The principal issue to solve was filtering out data given the previously discussed issues with

DBLINK and WHERE clauses.

Initial considerations.

It was assumed that after the initial loading of the tables, the application should perform the data transfer every day at 00:01 hours.

Therefore only complaints, surveys or accounts created in the previous day should be considered by the application.

Filtering MongoDB and CSV data

The application needed to implement a way to update the value for the ”previous day” date automatically, so complaints and surveys could be filtered.

Since JavaScript date functions return a UNIX time value, the process of converting this value to the required strings (both surveys and complaints having different date formats) was rather cumbersome.

Luckily the library ”Moment.js” was found, simplifying the process of working as shown below:

```
let yesterdaySurvey = moment().subtract(1,'days').format('DD-MM-YYYY');
let yesterdayComp = moment().subtract(1,'days').format('YYYY-MM-DD');
```

To extract only the necessary collections, it suffices to modify the initial function as shown below. For the initial data, an ”if” condition was added to filter out rows by date.

```
// MongoDB
const collection =  Document.find({date_submitted: yesterdaySurvey})
// CSV
if(String(data['date_received'].trim()) == yesterdayComp){//select row}
```

Where clauses and DBLINK

To solve the issues with DBLINK, and since no clear indications were found despite having spent a fair amount of time researching for similar problems, it was decided that a set of tables containing the last value processed for each dimensional table should be created.

```
CREATE TABLE staging_area.max_customer (
   max_customer INTEGER PRIMARY KEY
);
```

This was important because it helps the application distinguish between, for example, new customers and existing customers that have sent a new complaint or survey.

To achieve this distinction, the queries were modified to include the WHERE clause inside the DBLINK query, as follows:

```
INSERT INTO DATA_MART.dw_d_customer(customer_id,customer_name,
    customer_surname,country_name)
     SELECT *
     FROM DBLINK('host = localhost
                 user = postgres
                 password = postgres
                 dbname = investor',
                 'SELECT *
                 FROM staging_area.sa_d_customer
                 WHERE customer_id > (SELECT max_customer FROM
                    staging_area.max_customer')
                 as linktable(a int, b varchar(20), c varchar(20), d
                    varchar(50))`
```

This means that the results returned are already filtered by the inner query inside the ***DBLINK()***, rather than fetching all the data first and filtering it afterwards, as it would have been the case with a normal DBLINK() query.

To keep track of the last values processed, the code related to the initial load was modified, so after the data had been transferred, it inserted the last value (using the function ***MAX()***) for each one of the dimension tables.

This allowed the application during the Incremental Load stage, to have access to those values to filter the necessary data and send only new data to the data mart.

It needs to be noted that in the staging area the dimensional tables will select all existing data from the temp table, being the only one filtered by date "surveys" and "complaints".

This allows the fact tables to be populated with the appropriated keys in case existing customers send complaints, respond to surveys or take part in existing promotions, but by using the select clause as shown above it prevents the application from sending data that has already been stored in the data mart.

Incremental load process.

Considering what has been discussed so far, the process the application follows in the incremental load stage is as shown below:

- Delete the data contained in the staging tables from the previous day.
- Filter MongoDB and .csv data to include only the previous day data.
- Combine all data sources to create a unique set of tuples.
- Populate the dimension tables.
- Populate staging fact tables by referencing the appropriated F.K. from the dimension tables.
- Filter the data sent to the data mart, using the WHERE clause shown in the example above.
- Update the helper tables.

The main difference concerning the initial data-load was motivated by the insufficient "TRANSACTIONS" support P.G. offers.

In this occasion, considering the tables in the data mart already contain data, having a single INSERT operation failing was a risk that could not be taken. Although P.G. offers support for transactions, their implementation is not particularly reliable, which lead to the decision of wrapping all the INSERT operations from the staging area to the data mart into a ***BEGIN...END*** block inside a stored procedure.

This guarantees that in case of server failure, power-cut or any potential issue, the transaction could be rolled back without affecting the tables in the data mart.

4.3 Scheduling Jobs with Cron.

To schedule the incremental load to be performed every day, the only thing needed was to wrap the whole process inside a function and then export it to the file where "node-cron" was initialized.

The structure of cron jobs is composed of five "*" symbols, representing minuted, hours, days, months and day of the week.

To run the application at one minute past midnight, every day we would write the following:

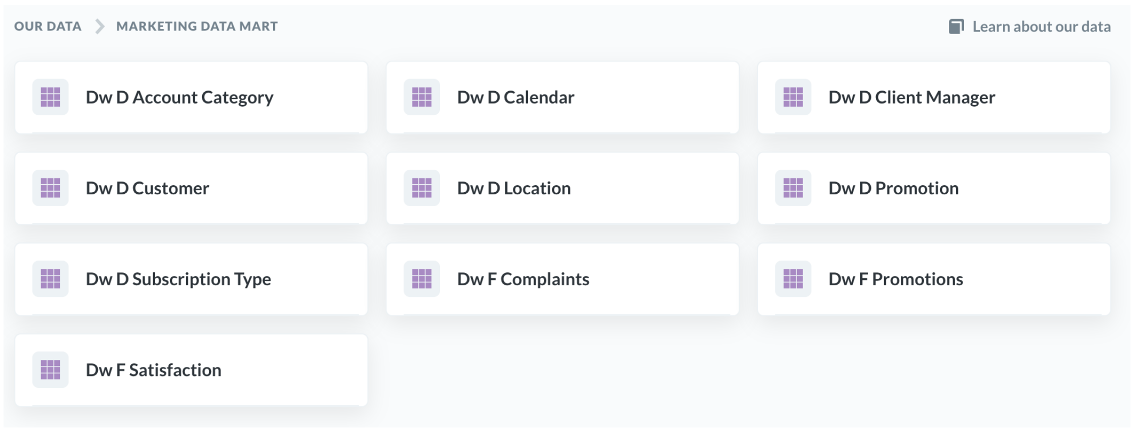
```
const cron = require('node-cron');
const incremental = require("./postgreSQLincremental");
cron.schedule( '01 00 * * *',()=>{
  incremental();
  })
```

It suffices to run this file and leave it open for the process to run at the indicated time.

4.4 Data Exploitation with Metabase

Now that the data has been transferred, Metabase will be used to illustrate one of the potential uses of the data repository.

This project will run Metabase locally, by simply downloading the .jar file from www.metabase.com, run it and connect to localhost:3000. After a simple configuration to indicate the database type, passwords etc. the below image confirms the set-up has been successful.


OUR DATA > MARKETING DATA MART
Learn about our data

Dw D Account Category
Dw D Calendar
Dw D Client Manager
Dw D Customer
Dw D Location
Dw D Promotion
Dw D Subscription Type
Dw F Complaints
Dw F Promotions
Dw F Satisfaction

Figure 4.3: Metabase main screen.

4.4.1 Defining queries

For users who do not feel confident using SQL, Metabase allows them to filter and query the data as if they were working with excel spreadsheets.

For instance, a user that clicks on the Dw F Complaints table could perform some simple queries, for example counting the number of complaints belonging to the category "Technical", using the graphical user interface shown below:

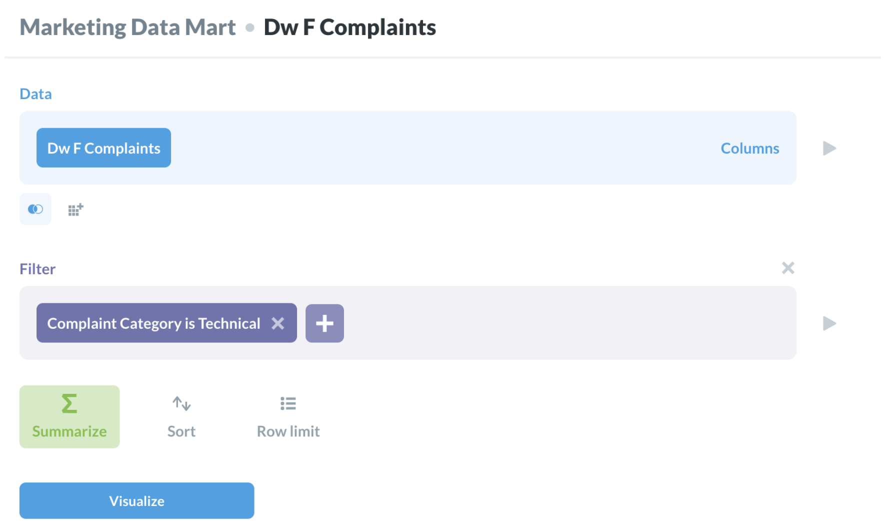
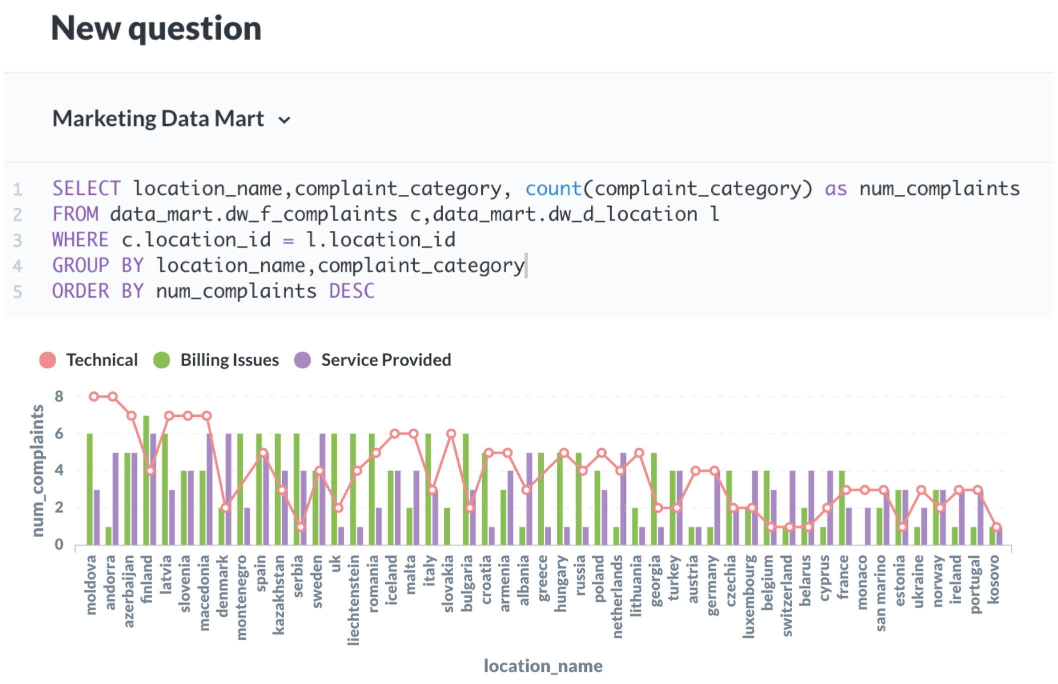

Marketing Data Mart • Dw F Complaints
Data
Dw F Complaints
Columns
Filter
Complaint Category is Technical
Summarize
Sort
Row limit
Visualize

Figure 4.4: Metabase query builder.

For more complex analysis joining several tables, for example, showing complaints by category and country name, SQL queries can also be performed, which results can then be visualized straight away and added to a dashboard if needed.



```sql
SELECT location_name,complaint_category, count(complaint_category) as num_complaints
FROM data_mart.dw_f_complaints c,data_mart.dw_d_location l
WHERE c.location_id = l.location_id
GROUP BY location_name,complaint_category
ORDER BY num_complaints DESC
```

Figure 4.5: SQL query and data visualization example.

The feedback provided by the users after having used the application and the modifications implemented will be discussed in the next section.

Chapter 5

Testing and Evaluation.

This section will cover the test carried to ensure the application was working correctly, i.e. that no data loss happened, and that the application would still operate under extreme cases.

As part of the evaluation, the modifications introduced to improve the user experience will be discussed and justified.

5.1 Data loss

As discussed during the implementation chapter in section 4.0.6, not all the data sent was being written into MongoDB.

The possible causes were either not all messages were being sent, or not all messages were being written.

The number of messages sent and received was verified by adding a ”counter” variable to both the publisher and the subscriber, that would be printed every time a message was sent or received.

Using ***console.log(counter)*** it was confirmed that all the objects were being sent by the publisher, the data loss happening when ***save()*** failed.

This issue was fixed by wrapping the acknowledgement of the message inside an ***IF*** statement. If the ***.save()*** operation failed, the message won't be acknowledged, and RabbitMQ will send it again to the consumer, guaranteeing that all messages would be written, as the successive test carried shown.

```
survey.save(function(error,object){  if(!error){
    channel.ack(message); } })
}) }
```

5.2 Modifying data volumes

Because a schema is enforced on both MongoDB objects and PostgreSQL, and no user interaction with the application exists, the testing phase focused on the volumes of data the application could handle to ensure both, that all the data generated was written into the database, and also that the application didn't suffer from time-outs or other kinds of exceptions.

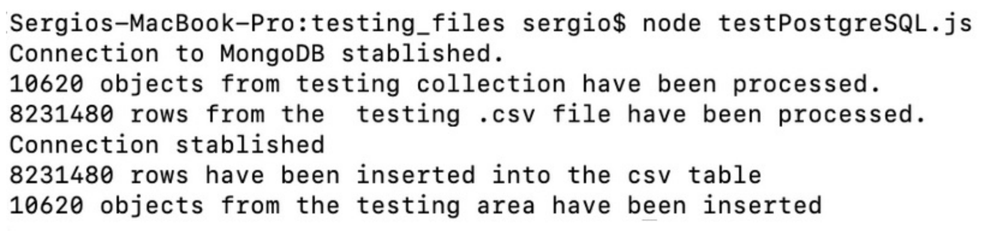
The testing was carried, creating a specific set of testing files and two separated databases.

The file data.json was increased to contain +10k objects, and the .csv file was replaced by the "calendar_summary.csv" belonging to the AirBNB Berlin data-set, containing 8,231,480 rows in a 100Mb .csv file.

A new database "testing-database" was created in MongoDB, and another testing database "test" was created in PostgreSQL, containing two tables for both MongoDB data and the .csv file.

The testing carried confirmed that consumer.js was able to write all the data into MongoDB, that the application was able able to handle .csv file without causing a crash regardless of their size, that the promise chain was kept, and that the data was stored in PostgreSQL without incurring in time-out errors.

As the below image confirms the application was able to handle the data, being the only unrelated issue encountered at this stage was the fact that using DELETE on the .csv table invariably crashed PostgreSQL's server until the function TRUNCATE was used.



```
Sergios-MacBook-Pro:testing_files sergio$ node testPostgreSQL.js
Connection to MongoDB stablished.
10620 objects from testing collection have been processed.
8231480 rows from the  testing .csv file have been processed.
Connection stablished
8231480 rows have been inserted into the csv table
10620 objects from the testing area have been inserted
```

Figure 5.1: Testing Results.

The other case that was necessary to consider was "what happens if, for a given day, there are no complaints or surveys to process?". In the initial implementation, two arrays were passed to a PG query, but when these were empty, the application crashed because of a parsing issue.

To solve this, an IF...ELSE block was added, and a dummy query was added in case the length of the array was zero, so the promise chain could be kept. The below snippet shows the modification carried to handle the "no surveys" event, a similar if...else block was created for the .csv data.

```
if(mongodata.length !== 0){
return client.query(format(saQueries.intertSurveys , mongodata ),[])
}else{
return client.query('SELECT * FROM STAGING_AREA.MAX_CATEGORY')
}.then(result=>{// next operation...}
```

5.3 User Feedback on Metabase.

Although Metabase is not considered a core element of the application, it provides an excellent way to assess the data integration process from an end-user perspective.

For evaluation purposes, the aim of the application was explained to work colleagues, who are used to work with excel, so they could query the data using Metabase as if they did for the fictitious marketing department this project considered.

The results were pretty unanimous, the application was appealing and the menus easy to use, but most users expected to have all the information they needed in a single table, as if it was an Excel spreadsheet, without having to perform any querying in SQL.

As briefly shown at the end of the previous chapter, METABASE offers a graphical user interface to query data in a similar way it would be done using Excel. However, this functionality is limited to a single table, and as such, the current data mart schema would not meet the requirements the users specified above.

A more specific set of requirements were collected using Agile methodologies through the use of "user-stories" by asking the users ***"If you worked for the marketing department, what would you like to see when opening the application?".***

The stories that were gathered followed the structure "As a user, I would like to ...". Which are shown below:

- What are the min, max and average satisfaction levels for each category?
- What are the three most valued advisors?
- What are the most popular promotions by type of customer?
- Do complaints have any relationship with the satisfaction levels?

Although these questions could quickly be answered by creating a specific dashboard in Metabase, the application should be able to satisfy these requirements without relying on specific external technology.

Views and Materialized views

The above requirements could be defined as the need for users to access a different representation of the underlying data., in this case, by using aggregation or rank functions.

A view or materialised view offer the best use-case for the above problem, since they remove the need to perform INSERT operations into another table, and the representation can be easily updated every time the view is queried, or in the case of a materialised view, every time it is refreshed.

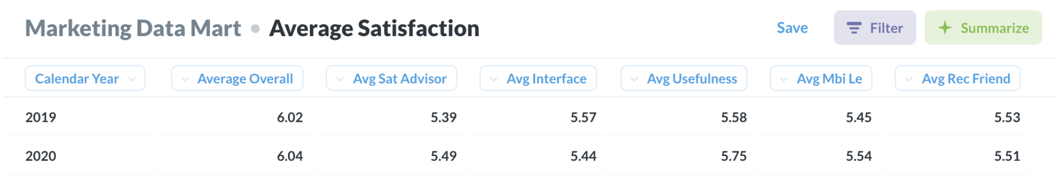
Materialised views seemed a better option since they are computed only when the view is refreshed, which compared to standard Views that are computed at query time, should offer better performance.

The query run-time of both types of views was evaluated using the EXPLAIN command, so an informed decision could be made.

During the first execution both types of views offered similar results around 40-60ms, but in subsequent calls, materialised views offered shorter run-times by an average of 10ms difference that should be more significant as data volumes grow.

Perhaps the main advantage materialised views have, is the fact that *they can be refreshed "without locking out concurrent selects on the materialised view"* [32], something that could, to a limited extent, allow the application to be updated more frequently than once a day, thanks also to the use of helper tables that limits the amounts of data the application needs to manage at a given moment.

The other advantage of using views is that in the case of Metabase, these can be queried as if it was another table, which allows developers to create tailored data-sets that meet user requirements, and from an application point of view, could enhance performance by decoupling the data source from the representation the user sees. If the data mart created as part of this project had to support a web-application targeted at users, it would be possible to generate a set of views from the application could extract the data, which would improve the performance of the application by sharing workloads across several tables.



Calendar Year	Average Overall	Avg Sat Advisor	Avg Interface	Avg Usefulness	Avg Mbi Le	Avg Rec Friend
2019	6.02	5.39	5.57	5.58	5.45	5.53
2020	6.04	5.49	5.44	5.75	5.54	5.51

Figure 5.2: Materialized view as Metabase Table.

5.4 Analysis of results

5.4.1 Functional requirements

Now that the application has been implemented, and going back to the set of requirements specified in Chapter 2, it can be said that the application has managed to integrate the existing data sources into a single repository in an automated manner. The testing carried shows that the application could work with larger volumes of data, although as we will discuss in the next section, this only represents one area of what the application should consider globally.

The implementation of materialised views allows for a straightforward way of tailoring the application to the end-user, and even if the volume and complexity of data the application had to deal with are modest in comparison to real-world scenarios, this project shows the potential this kind of applications could have in improving decision-making processes in organisations of the sort this project considered.

The implementation of the Alert system also shows potential uses of how real-time data could be used to improve organisational processes, generating different types of alerts whenever a particular event logic is met. However, considerable modifications should be undertaken if the application is to meet more challenging demands with larger volumes of data or more complex data processing.

5.4.2 Non-functional requirements

Through the use of open-source tools, a functional application has been developed at virtually zero cost, providing a prototype that the stakeholders could evaluate before embarking in more extensive and costly projects.

The application also shows that it is possible to implement custom ETL tools without having to rely on commercial tools such as Pentaho DI or Kettle, whenever the data sources and requirements are well defined, even when using languages such as JavaScript that are miles behind other languages such as Python in terms of data management capabilities.

The use of Docker minimised the impact of the addition of new elements into the existing architecture, and the technologies the application needs, either reuse existing components (MongoDB) or implements new ones in technologies that already exist in the organisation's systems (PostgreSQL), and provides an exciting way of building complex applications encapsulating and minimising the overall impact of the application on existing architectures.

One of the requirements that have not fully been met is the separation between the application and the databases.

As discussed during the implementation section, working with PG imposed severe limitations, particularly with its implementation of TRANSACTIONS or working with SERIAL data types, that left no choice but to implement some of the application logic through the use of stored procedures.

Whilst this is not a critical issue, it could hinder the debugging or maintenance tasks, splitting the code between the application and the database adds an extra overhead that ideally should have been avoided.

5.5 Critical evaluation of the application

Despite its achievements, the application developed should not be considered more than a prototype that could serve as a guide to develop more complex and resilient applications, something the next section will explain.

Technologies Used.

Working with Node.js was the only realistic option available at the time of the implementation, considering the amount of time it would have taken to become familiar with other technologies.

As such Node.js was chosen based on providing a versatile tool with a relatively friendly learning curve, that could provide for a fall-back plan in case changes had to be introduced in the project, which as it turns out, was the case.

Although new libraries are being implemented to improve Node.js performance in CPU intensive operations [29], its single-thread nature represents a severe performance limitation that justifies re-developing the application in a different language, for example, Python or Java.

Python would have allowed the application to use Kafka Streams and KSql, alongside other established technologies such as Apache Spark, increasing the potential uses of the application whilst staying relatively close to the original implementation in terms of syntax, especially when compared to JAVA.

Using Python, the Alert System could have been implemented using Kafka as stream processor instead of RabbitMQ, providing a more solid implementation that could have handled larger volumes of data and enable SQL queries on streams or more complex data transformations.

However, data stream processing need to be supported by the appropriated architecture to ensure a good performance, which often means creating, running and managing a dedicated cluster for the stream processor, usually in a cloud environment, which needs specific knowledge that may not easily be available within average S.M.E. organizations.

S.M.E. need to carefully consider the advantages and benefits stream processing can bring, compared against their increase in maintenance and overall cost, that in some cases may not be negligible.

Dimensional Design considerations.

Although this project discussed dimensional modelling, real-world scenarios require a much more extensive and complex process involving users, to define the appropriated facts and dimensions in a way that avoids data silos when dealing with multiple entities that may not share conformed dimensions.

The use of a Dimensional Matrix would be a useful tool to identify conformed and non-conformed dimensions, but this would only represent the first step of a complicated process.

Modelling a time dimension, for example, could be a tricky task to undertake. Our project did not include a physical time dimension (hours, minutes...) which would have been deemed essential in other types of businesses, something that had would require creating another time dimension, i.e. D_Time or D_clock_time, and add the primary keys as a foreign key into the D_calendar table.

The complexities do not stop there, what if for example, some facts need sales grouped only by months? Should another table be created to reflect this or is there a way to model this requirement more simply?, or considering the calendar dimension should Bank holidays" have been included as an attribute?, or Fiscal periods?.

When considering fact tables, should they be modelled as a periodic snapshot table? An accumulating one?.

This and many other questions highlight the fact that modelling business entities can be complicated. Although using a systematic approach such as the one described in [18] can help, it remains a complex and challenging task to implement without incurring in significant technical debt, something that was not possible to fully capture in this project.

Data Mart architectural considerations.

In terms of architecture, ROLAP represents a good starting point for the application. Most of the maintenance tasks the data mart would need load balance, configuring the server cache, partitioning tables, or indexing would be familiar for a database administrator.

This allows S.M.E. to reuse existing I.T. resources without adding a considerable overhead in the overall maintenance requirements.

Relational systems impose severe constraints in terms of frequency of updates and types of analysis that could be supported, something that was discussed in the project proposal.

If real-time analysis was deemed to be critical for the organization, this functionality should be diverted to MongoDB, since at best, what could be achieved with a ROLAP architecture is to increase the frequency of the updated.

As discussed in [9], the organization's information requirements do not remain static and evolve. As new requirements emerge involving more complex fact tables, and more massive amounts of data to store, a column database such as Big Table or Cassandra could represent a better solution both in terms of data storage and query performance.

A good explanation of how column databases compress data and how they make more efficient use of CPU cycles can be found in [17].

Cloud Environments and B.I.

The two-column databases mentioned before are usually found in cloud environments, which is becoming increasingly relevant in B.I. Contexts since they allow " cost-efficient increased scalability and flexibility" [14], although as discussed in this paper, serious considerations in terms of security or data availability are needed before a successful could migration could be implemented.

Besides well-known platforms like Azure, A.W.S. or Google Cloud, there is an increasing number of organizations catering at specific business areas, such as Customer Relationship Management of which the Salesforce platform represents the most popular choice.

These platforms benefit from the fact that business tends to have a well-defined set of business entities (clients, leads. accounts). This allows the application to define a series of functions that propagates data changes automatically as soon as new data is entered or modified, something that would not be possible in the application developed as part of this project, that is bounded to a batch-process (however frequent) to update the information.

The above example represents what is known as "Platform as a service" (PaaS), which allows organizations to run and manage applications without having to manage the infrastructure and maintenance needs they may have.

The fact that both the infrastructure and maintenance of the application is externalized explains the appeal this platform have for organizations that have well-defined information needs, but not necessarily the I.T. resources needed to manage them.

Using Cloud Platform, organizations can avoid large sunk-cost in terms of hardware investment, and simply treat them as a variable cost in the function of the resources used and services hired in the platform.

Open-source technologies and data governance

Open-source technologies represent a valid option that allows organizations to reduce costs and gain flexibility concerning third-party alternatives, although the risks associated with under-managing of the application are particularly relevant for non-commercial tools [4].

The above paper recommends that appropriated enterprise governance procedures are put in place to assess the total cost of ownership (TCO), so the results of investments in open-source tools could be scrutinized and discussed between the relevant stakeholders.

This last point links to an area that goes beyond architectures and software and enters into organizational matters.

Business Intelligence applications need to be supported by robust data governance structures, and organizations need to understand that business value is only generated when adequate data quality standards are met.

Data needs to be understood as a business asset, and the appropriated procedures, standards and controls should be implemented in the same manner organizations do to their inventories or finances, something that is often difficult to implement in organizations already find difficult to cope with standard business requirements.

What has been discussed so far highlights the fact that S.M.E. organizations need to perform a balancing act between the expected performance of the application and the resources available for their development and maintenance, including setting in place the appropriated data governance procedures.

An application may have a sound design, but it would not produce the expected results unless it uses the appropriated data, and the necessary resources are spent in its maintenance.

Data Analytics has, therefore, a cost in terms of the resources they need to offer the expected performance, which needs to be carefully considered by the organization's stakeholders who need to realize that successful B.I. Applications need to be understood within an organizational context that goes beyond purely I.T. matters.

5.5.1 Further Development

As briefly discussed in the previous section because of time constraints, essential areas of B.I. applications such as partitioning, physical organization of the tables, indexing or data compression have not been considered, even though they are critical areas to ensure optimal functioning of a data mart as the volumes of data increase.

Other than extending E.T.L. scripts to handle different data formats, or have included other types of NoSQL databases such as Redis, the main area to be explored would be Data Warehouse in the Cloud, using software as a service (SaaS) such as Snowflake [35].

Linked to the Cloud and particularly data stream processing, Cluster Computing is a relevant area worth exploring that can enable organizations to implement efficient ways to manage and react adequately to streams of data, in areas such as recommendation systems, fraud detection or industrial processes as seen in [2].

5.6 Lessons learnt

This project represents the first non-trivial application I have ever built, and as such many (often painful) lessons were learnt.

The first one I should mention is, not to blindly follow the hype.

The applications of data streams and Apache Kafka seemed so appealing that not enough attention was placed on developing the project beyond the Alert System. When the initial implementation failed, it took longer than expected to redirect the project to its final form, even if luckily, the technologies used have proven flexible enough to accommodate these changes.

Another one is that relational databases, still have enormous potential when developing applications for the kind of organizations this project considers, where the company I am currently working for could represent a good example. The data sources are well defined both in terms of their structure and contents, being the main issue the existence of data silos represented by departments where the information is kept.

As discussed in [20], integrating scattered data sources, for example, warranty returns, suppliers and sales, could help organizations predict the expected amount of returns using the appropriated statistical techniques, and plan the necessary action, something that is an immediate concern for many organizations and provides them with a good use-case of implementing the appropriated B.I. processes beyond fancy data visualizations.

With regards to the development stage, the main takeaway is "start small". I was so worried about not being able to build the prototype before the deadline, that the implementation was rushed, causing many application crashes.

The development phase only took off once I focused on successfully implementing a single feature, i.e. the database, and when completed, implement the second one, i.e. connecting P.G. admin to the database and perform some queries to assess the behaviour of the application.

Although many errors and crashed still happened, this sequential process allowed the application to grow gradually, having a solid basis to revert to in case the code needed to be refactored.

Not related to code as such, but having to create the data sources from scratch represented a tremendous effort, and in a couple of occasions it almost derailed the project, since Excel is an "excellent" tool to generate insidious formatting issues. The amounts of data the application handles is the area I am the least happy with, something that I tried to compensate by focusing more on the E.T.L. aspects of the application.

This project made me realize that there is a considerable component of software development in the data management field, and in turn, a good understanding of databases is needed to implement an application successfully, something I don't think I would have considered if it was not because of my studies at Birkbeck, including those modules like C.S. or F.O.C. that initially did not seem to have an immediate practical use.

Finally, now that this project is completed, and despite the shortcomings, a first project is expected to have, I can say I am proud to see an application I did not think I would have been able to create at the beginning of the previous academic year.

This has only been made possible thanks to the academic staff at Birkbeck, and the timely feedback provided by my supervisor for which I can only be grateful.

Appendix A

Database Schema Definition.

```
--  CREATE DATABASE investor;

-- Session configuration and start

SET TRANSACTION READ WRITE;

BEGIN WORK;

-- Begin schema creation.

CREATE SCHEMA investment_manager;

-- tb_country definition

CREATE TABLE investment_manager.tb_country (

    country_id INTEGER CONSTRAINT pk_country PRIMARY KEY,
    country_name varchar(20) NOT NULL

);

-- tb_client_manager definition

CREATE TABLE investment_manager.tb_client_manager (

    client_manager_id INTEGER CONSTRAINT pk_client_manager PRIMARY KEY,
    client_manager_name varchar(50),
    client_manager_surname varchar(50),
    country_id INTEGER NOT NULL,
    CONSTRAINT fk_manager_country FOREIGN KEY (country_id) REFERENCES
        investment_manager.tb_country (country_id)

);
```

```
-- tb_subscription_type definition

CREATE TABLE investment_manager.tb_subscription_type (

    subscription_id INTEGER CONSTRAINT pk_subscription PRIMARY KEY,
    subscription_name   varchar(10) NOT NULL,
    subscription_fee INTEGER NOT NULL CHECK(subscription_fee >= 0)

);

-- tb_customer_category definition

CREATE TABLE investment_manager.tb_customer_category (

    category_id INTEGER CONSTRAINT pk_category PRIMARY KEY,
    customer_category VARCHAR(50) NOT NULL

);

-- tb_customer definition

CREATE TABLE investment_manager.tb_customer (

    customer_id INTEGER CONSTRAINT pk_customer_id PRIMARY KEY,
    customer_name varchar(20) NOT NULL,
    customer_surname varchar(20) NOT NULL,
    country_id INTEGER NOT NULL,
    customer_email varchar(50) UNIQUE,
    CONSTRAINT fk_cust_country FOREIGN KEY (country_id) REFERENCES
       investment_manager.tb_country (country_id)
);

-- tb_sales_area definition

CREATE TABLE investment_manager.tb_sales_area (

    client_manager_id INTEGER,
    country_id INTEGER,
    CONSTRAINT pk_sales_area PRIMARY KEY (client_manager_id,country_id),
    CONSTRAINT fk_manager FOREIGN KEY (client_manager_id) REFERENCES
       investment_manager.tb_client_manager (client_manager_id),
    CONSTRAINT fk_territory FOREIGN KEY (country_id) REFERENCES
       investment_manager.tb_country (country_id)

);
```

```
-- tb_promotions definition

CREATE TABLE investment_manager.tb_promotions (

    promotion_id INTEGER CONSTRAINT pk_promotion PRIMARY KEY,
    promotion_name varchar(50) NOT NULL,
    promo_description varchar(100) NOT NULL

);

-- tb_customer_promotions definition

CREATE TABLE investment_manager.tb_customer_promotions (
    customer_promo_id SERIAL CONSTRAINT pk_promo_id PRIMARY KEY,
    promotion_id INTEGER NOT NULL,
    customer_id INTEGER NOT NULL,
    promotion_taken BOOLEAN DEFAULT FALSE,
    promotion_date DATE NOT NULL,
    CONSTRAINT fk_promotions_promoted FOREIGN KEY (promotion_id) REFERENCES
       investment_manager.tb_promotions,
    CONSTRAINT fk_customer_promotions FOREIGN KEY (customer_id) REFERENCES
       investment_manager.tb_customer

);

-- tb_customer_account definition

CREATE TABLE investment_manager.tb_customer_account (

    account_id INTEGER PRIMARY KEY,
    customer_id INTEGER NOT NULL,
    client_manager_id INTEGER NOT NULL,
    subscription_id INTEGER NOT NULL,
    category_id INTEGER NOT NULL,
    account_created DATE NOT NULL,
    account_active BOOLEAN NOT NULL DEFAULT TRUE,
    account_deactivation DATE DEFAULT NULL,
    CONSTRAINT fk_customer_acc_id FOREIGN KEY (customer_id) REFERENCES
       investment_manager.tb_customer,
    CONSTRAINT fk_customer_category FOREIGN KEY (category_id) REFERENCES
       investment_manager.tb_customer_category,
    CONSTRAINT fk_customer_acc_manager FOREIGN KEY (client_manager_id) REFERENCES
       investment_manager.tb_client_manager,
    CONSTRAINT fk_customer_acc_subscription FOREIGN KEY (subscription_id)
       REFERENCES investment_manager.tb_subscription_type,
    CONSTRAINT fk_customer_acc_type FOREIGN KEY (subscription_id) REFERENCES
       investment_manager.tb_subscription_type
);
-------------------------------------------------------------------
--
-- Data tb_cryptocurrencies

CREATE TABLE investment_manager.tb_cryptocurrencies (
    cryptocurrency_symbol varchar(10) PRIMARY KEY,
    cryptocurrency_name varchar(50) NOT NULL

);

--  tb_crypto_valuation

CREATE TABLE investment_manager.tb_crypto_valuation (
    crypto_valuation_id INTEGER PRIMARY KEY,
    cryptocurrency_symbol VARCHAR(10) NOT NULL,
    crypto_exchange_rate DECIMAL,
    crypto_valuation_date DATE,
    CONSTRAINT fk_crypto_symbol FOREIGN KEY (cryptocurrency_symbol) REFERENCES
        investment_manager.tb_cryptocurrencies

);

-- tb_stocks definition

CREATE TABLE investment_manager.tb_stocks (

    stock_id INTEGER CONSTRAINT pk_stock PRIMARY KEY,
    stock_symbol varchar(10) NOT NULL UNIQUE,
    stock_name varchar(100)
);

-- tb_customer_portfolio definition

-- Assumes customers have at least 1 stock and may or may not have cryptocurrencies.

CREATE TABLE investment_manager.tb_customer_portfolio (

    portfolio_id INTEGER PRIMARY KEY,
    customer_id INTEGER NOT NULL,
    stock_id INTEGER NOT NULL,
    number_of_stocks INTEGER NOT NULL CHECK (number_of_stocks >= 0),
    cryptocurrency VARCHAR(10),
    crypto_units INTEGER,
    CONSTRAINT fk_portfolio_customer FOREIGN KEY (customer_id) REFERENCES
        investment_manager.tb_customer,
    CONSTRAINT fk_portfolio_stocks FOREIGN KEY (stock_id) REFERENCES
        investment_manager.tb_stocks,
    CONSTRAINT fk_crypto_currency FOREIGN KEY (cryptocurrency) REFERENCES
        investment_manager.tb_cryptocurrencies
);
```

```sql
-- tb_stock_valuation definition

CREATE TABLE investment_manager.tb_stock_valuation (

  valuation_id INTEGER CONSTRAINT pk_valuation_id PRIMARY KEY,
  stock_id INTEGER NOT NULL,
  closing_value DECIMAL NOT NULL CHECK(closing_value >= 0),
  valuation_date DATE NOT NULL,
  CONSTRAINT fk_stock_valuation FOREIGN KEY (stock_id) REFERENCES
      investment_manager.tb_stocks

 );

COMMIT;

-- End of schema creation
```

Appendix B

Staging Area / Data Mart Schema

Since the data mart tables mirror the ones in the staging area, only the staging area tables will be shown in the annex.

```sql
BEGIN WORK;
CREATE SCHEMA staging_area;


-- Table sa_surveys

CREATE TABLE staging_area.sa_surveys(

  reviewer_id INTEGER NOT NULL,
  overall_satisfaction INTEGER NOT NULL,
  advisor_satisfaction INTEGER NOT NULL,
  easiness_interface INTEGER NOT NULL,
  usefulness_service INTEGER NOT NULL,
  interest_mob_app INTEGER NOT NULL,
  recommend_friend INTEGER NOT NULL,
  survey_rec_date VARCHAR(20)
);

-- Table sa_complaints

CREATE TABLE staging_area.sa_complaints (

  customer_id INTEGER NOT NULL,
  complaint_category VARCHAR(30) NOT NULL,
  complaint_date VARCHAR(20)

);

-- Table sa_d_customer

CREATE TABLE staging_area.sa_d_customer (
    customer_id INTEGER CONSTRAINT pk_customer_id PRIMARY KEY,
    customer_name varchar(20) NOT NULL,
    customer_surname varchar(20) NOT NULL,
    country_name varchar(50) NOT NULL

);

-- Table sa_d_location

CREATE TABLE staging_area.sa_d_location (

  location_id INTEGER CONSTRAINT pk_location PRIMARY KEY,
  location_name VARCHAR(50) NOT NULL

);

-- Table sa_d_subscription_type

CREATE TABLE staging_area.sa_d_subscription_type (
  subscription_id INTEGER CONSTRAINT pk_subscription PRIMARY KEY,
  subscription_name VARCHAR(50) NOT NULL
);

-- Table sa_d_client_manager

CREATE TABLE staging_area.sa_d_client_manager (

  client_manager_id INTEGER CONSTRAINT pk_manager PRIMARY KEY,
  client_manager_name VARCHAR(50) NOT NULL,
  client_manager_surname VARCHAR(50) NOT NULL

);

-- Table sa_d_account_category

CREATE TABLE staging_area.sa_d_account_category (

  category_id INTEGER CONSTRAINT pk_category PRIMARY KEY,
  account_category VARCHAR(50) NOT NULL

);

-- Table sa_d_calendar

CREATE TABLE staging_area.sa_d_calendar (
 calendar_id SERIAL CONSTRAINT pk_calendar PRIMARY KEY,
 calendar_date DATE NOT NULL,
 calendar_day INTEGER NOT NULL,
 calendar_month INTEGER NOT NULL,
 calendar_year INTEGER NOT NULL
```

```
);

-- Table sa_d_promotion

CREATE TABLE staging_area.sa_d_promotion (

 promotion_id INTEGER CONSTRAINT pk_promotion PRIMARY KEY,
 promotion_name VARCHAR(50) NOT NULL,
 promotion_description VARCHAR(100) NOT NULL

);

-- Table sa_f_satisfaction

CREATE TABLE staging_area.sa_f_satisfaction(

 satisfaction_id SERIAL CONSTRAINT pk_satisfaction PRIMARY KEY,
 calendar_id INTEGER NOT NULL,
 customer_id INTEGER NOT NULL,
 subscription_id INTEGER NOT NULL,
 location_id INTEGER NOT NULL,
 client_manager_id INTEGER NOT NULL,
 category_id INTEGER NOT NULL,
 overall_satisfaction INTEGER,
 advisor_satisfaction INTEGER,
 easiness_interface INTEGER,
 usefulness_service INTEGER,
 interest_mob_app INTEGER,
 recommend_friend INTEGER,
 CONSTRAINT fk_calendar_sat FOREIGN KEY (calendar_id) REFERENCES
    staging_area.sa_d_calendar (calendar_id),
 CONSTRAINT fk_customer_sat FOREIGN KEY (customer_id) REFERENCES
    staging_area.sa_d_customer (customer_id),
 CONSTRAINT fk_acc_type_sat FOREIGN KEY (subscription_id) REFERENCES
    staging_area.sa_d_subscription_type (subscription_id),
 CONSTRAINT fk_location_sat FOREIGN KEY (location_id) REFERENCES
    staging_area.sa_d_location (location_id),
 CONSTRAINT fk_manager_sat FOREIGN KEY (client_manager_id) REFERENCES
    staging_area.sa_d_client_manager (client_manager_id),
 CONSTRAINT fk_category FOREIGN KEY (category_id) REFERENCES
    staging_area.sa_d_account_category (category_id)

);

-- Table sa_f_complaints

CREATE TABLE staging_area.sa_f_complaints (

  complaint_id SERIAL CONSTRAINT pk_complaint PRIMARY KEY,
  calendar_id INTEGER NOT NULL,
  customer_id INTEGER NOT NULL,
  subscription_id INTEGER NOT NULL,
  location_id INTEGER NOT NULL,
  client_manager_id INTEGER NOT NULL,
  category_id INTEGER NOT NULL,
  complaint_category VARCHAR(30),
  CONSTRAINT fk_calendar_comp FOREIGN KEY (calendar_id) REFERENCES
     staging_area.sa_d_calendar (calendar_id),
  CONSTRAINT fk_customer_comp FOREIGN KEY (customer_id) REFERENCES
     staging_area.sa_d_customer (customer_id),
  CONSTRAINT fk_acc_type_comp FOREIGN KEY (subscription_id) REFERENCES
     staging_area.sa_d_subscription_type (subscription_id),
  CONSTRAINT fk_location_comp FOREIGN KEY (location_id) REFERENCES
     staging_area.sa_d_location (location_id),
  CONSTRAINT fk_manager_comp FOREIGN KEY (client_manager_id) REFERENCES
     staging_area.sa_d_client_manager (client_manager_id),
  CONSTRAINT fk_category_comp FOREIGN KEY (category_id) REFERENCES
     staging_area.sa_d_account_category (category_id)
);
-- Table sa_f_promotions

CREATE TABLE staging_area.sa_f_promotions (

  f_promotion_id SERIAL CONSTRAINT pk_promo PRIMARY KEY,
  promotion_id INTEGER NOT NULL,
  calendar_id INTEGER NOT NULL,
  customer_id INTEGER NOT NULL,
  subscription_id INTEGER NOT NULL,
  location_id INTEGER NOT NULL,
  client_manager_id INTEGER NOT NULL,
  category_id INTEGER NOT NULL,
  promotion_taken BOOLEAN NOT NULL,
  CONSTRAINT fk_promo FOREIGN KEY (promotion_id) REFERENCES
     staging_area.sa_d_promotion (promotion_id),
  CONSTRAINT fk_calendar_prom FOREIGN KEY (calendar_id) REFERENCES
     staging_area.sa_d_calendar (calendar_id),
  CONSTRAINT fk_customer_prom FOREIGN KEY (customer_id) REFERENCES
     staging_area.sa_d_customer (customer_id),
  CONSTRAINT fk_acc_type_prom FOREIGN KEY (subscription_id) REFERENCES
     staging_area.sa_d_subscription_type (subscription_id),
  CONSTRAINT fk_location_prom FOREIGN KEY (location_id) REFERENCES
     staging_area.sa_d_location (location_id),
  CONSTRAINT fk_manager_prom FOREIGN KEY (client_manager_id) REFERENCES
     staging_area.sa_d_client_manager (client_manager_id),
  CONSTRAINT fk_category_prom FOREIGN KEY (category_id) REFERENCES
     staging_area.sa_d_account_category (category_id)
);
```

```sql
-- Creation of the helper tables

CREATE TABLE staging_area.max_date (
   max_date DATE PRIMARY KEY
);

CREATE TABLE staging_area.max_subscription_type (
   max_subscription INTEGER PRIMARY KEY
);

CREATE TABLE staging_area.max_client_manager (
   max_manager INTEGER PRIMARY KEY
);

CREATE TABLE staging_area.max_promotion (
   max_promotion INTEGER PRIMARY KEY
);

CREATE TABLE staging_area.max_customer (
   max_customer INTEGER PRIMARY KEY
);

CREATE TABLE staging_area.max_location (
   max_location INTEGER PRIMARY KEY
);

CREATE TABLE staging_area.max_category (
   max_category INTEGER PRIMARY KEY
);
COMMIT;
```

Appendix C

Stored procedures

```
-- Create taging_area.clean_staging_tables()

CREATE OR REPLACE FUNCTION staging_area.clean_staging_tables()
RETURNS VOID AS $$
BEGIN
DELETE FROM staging_area.sa_f_complaints;
DELETE FROM staging_area.sa_f_promotions;
DELETE FROM staging_area.sa_f_satisfaction;
DELETE FROM staging_area.sa_surveys;
DELETE FROM staging_area.sa_complaints;
DELETE FROM staging_area.sa_d_account_category;
DELETE FROM staging_area.sa_d_calendar;
DELETE FROM staging_area.sa_d_client_manager;
DELETE FROM staging_area.sa_d_customer;
DELETE FROM staging_area.sa_d_location;
DELETE FROM staging_area.sa_d_promotion;
DELETE FROM staging_area.sa_d_subscription_type;
END;
$$ LANGUAGE PLPGSQL;
COMMIT;

-- Create staging_area.sa_create_calendar()
EATE OR REPLACE FUNCTION staging_area.sa_create_calendar()
RETURNS VOID AS $$
BEGIN
INSERT INTO
    staging_area.sa_d_calendar(calendar_date,calendar_day,calendar_month,calendar_year)
    SELECT DISTINCT VALUATION_DATE,
    CAST(EXTRACT(DAY FROM VALUATION_DATE) AS INTEGER) calendar_day,
    CAST(EXTRACT(MONTH FROM VALUATION_DATE) AS INTEGER) calendar_month,
    CAST(EXTRACT(YEAR FROM VALUATION_DATE) AS INTEGER) calendar_year
    FROM INVESTMENT_MANAGER.TB_STOCK_VALUATION
    ORDER BY VALUATION_DATE ASC;
END;
```

```
$$ LANGUAGE PLPGSQL;
COMMIT;

-- Create staging_area.sa_create_incremental_calendar()

CREATE OR REPLACE FUNCTION staging_area.sa_create_incremental_calendar()
RETURNS VOID AS $$
BEGIN
INSERT INTO
    staging_area.sa_d_calendar(calendar_date,calendar_day,calendar_month,calendar_year)
        SELECT CURRENT_DATE - 1 AS VALUATION_DATE,
        CAST(EXTRACT(DAY FROM CURRENT_DATE - 1) AS INTEGER) calendar_day,
        CAST(EXTRACT(MONTH FROM CURRENT_DATE - 1) AS INTEGER) calendar_month,
        CAST(EXTRACT(YEAR FROM CURRENT_DATE - 1) AS INTEGER) calendar_year;

END;
$$ LANGUAGE PLPGSQL;
COMMIT;

-- Create staging_area.fill_f_complaints()

CREATE OR REPLACE FUNCTION staging_area.fill_f_complaints()
RETURNS VOID AS $$
BEGIN
INSERT INTO
    staging_area.SA_F_COMPLAINTS(calendar_id,customer_id,subscription_id,location_id,
client_manager_id,category_id,complaint_category)
SELECT DISTINCT CAL.CALENDAR_ID,CUSTOMER_ID,subscription_id,country_id,
client_manager_id,category_id,complaint_category
FROM temp_table t, staging_area.sa_d_calendar cal
WHERE CAL.CALENDAR_DATE = COMPLAINT_DATE
AND complaint_category IS NOT NULL;
END;
$$ LANGUAGE PLPGSQL;
COMMIT;

-- Create staging_area.incremental_f_complaints()

CREATE OR REPLACE FUNCTION staging_area.incremental_f_complaints()
RETURNS VOID AS $$
BEGIN
INSERT INTO
    staging_area.SA_F_COMPLAINTS(calendar_id,customer_id,subscription_id,location_id,
client_manager_id,category_id,complaint_category)
SELECT DISTINCT CAL.CALENDAR_ID,CUSTOMER_ID,subscription_id,country_id,
client_manager_id,category_id,complaint_category
FROM temp_table t, staging_area.sa_d_calendar cal
WHERE CAL.CALENDAR_DATE = complaint_date
AND COMPLAINT_DATE = (SELECT CURRENT_DATE - 1)
AND complaint_category IS NOT NULL;
END;
```

```
$$ LANGUAGE PLPGSQL;
COMMIT;

-- Create staging_area.fill_f_promotions()

CREATE OR REPLACE FUNCTION staging_area.fill_f_promotions()
RETURNS VOID AS $$
BEGIN
INSERT INTO staging_area.sa_f_promotions(promotion_id,calendar_id,customer_id,
subscription_id,location_id,
client_manager_id,category_id,promotion_taken)
SELECT DISTINCT p.promotion_id,cal.calendar_id,t.customer_id,t.subscription_id,
t.country_id,
t.client_manager_id,t.category_id,p.promotion_taken
FROM temp_table t
JOIN investment_manager.tb_customer_promotions p ON t.customer_id = p.customer_id
JOIN staging_area.sa_d_calendar cal ON cal.calendar_date = p.promotion_date
ORDER BY CALENDAR_ID ASC;
END;
$$ LANGUAGE PLPGSQL;
COMMIT;

-- Create staging_area.fill_f_satisfaction()

CREATE OR REPLACE FUNCTION staging_area.fill_f_satisfaction()
RETURNS VOID AS $$
BEGIN
INSERT INTO staging_area.sa_f_satisfaction(calendar_id,customer_id,
subscription_id,location_id, client_manager_id,category_id,overall_satisfaction,
advisor_satisfaction,easiness_interface,usefulness_service,interest_mob_app,
recommend_friend)
SELECT distinct cal.calendar_id,t.customer_id,t.subscription_id,t.country_id,
t.client_manager_id,
t.category_id,t.overall_satisfaction,t.advisor_satisfaction,t.easiness_interface,
t.usefulness_service,t.interest_mob_app,t.recommend_friend
FROM temp_table t, staging_area.sa_d_calendar cal
WHERE t.survey_date = cal.calendar_date
ORDER BY CALENDAR_ID ASC;
END;
$$ LANGUAGE PLPGSQL;
COMMIT;

--The below statement is used to reset the serial sequence.

SELECT SETVAL((SELECT pg_get_serial_sequence('staging_area.SA_D_CALENDAR',
    'calendar_id')), 1, false);
SELECT SETVAL((SELECT pg_get_serial_sequence('staging_area.SA_F_COMPLAINTS',
    'complaint_id')), 1, false);
SELECT SETVAL((SELECT pg_get_serial_sequence('staging_area.SA_F_PROMOTIONS',
    'f_complaint_id')), 1, false);
```

```
SELECT SETVAL((SELECT
   pg_get_serial_sequence('staging_area.SA_F_SATISFACTION','satisfaction_id')),1,
false);

-- Create data_mart.incremental_load_DW()
-- This function wraps all the INSERT statements in a transaction block.

CREATE OR REPLACE FUNCTION data_mart.incremental_load_DW()
RETURNS VOID AS $$
BEGIN;

INSERT INTO DATA_MART.DW_D_account_category(category_id,account_category)
          SELECT *
          FROM DBLINK(host = localhost user = postgres password = postgres dbname
             = investor,
          SELECT * FROM staging_area.sa_d_account_category WHERE category_id >
             (SELECT max_category FROM staging_area.max_category)) as
          linktable(a int, b varchar(50));

SELECT data_mart.filter_date();

INSERT INTO DATA_MART.dw_d_client_manager(client_manager_id,client_manager_name,
client_manager_surname)
          SELECT *
          FROM DBLINK(host = localhost
                    user = postgres
                    password = postgres
                    dbname = investor,
                    SELECT * FROM staging_area.sa_d_client_manager WHERE
                       client_manager_id > (SELECT max_manager FROM
                       staging_area.max_client_manager)) as linktable(a int,
                    b varchar(50), c varchar(50));

INSERT INTO DATA_MART.dw_d_customer(customer_id,customer_name,customer_surname,
country_name)
          SELECT *
          FROM DBLINK(host = localhost
                    user = postgres
                    password = postgres
                    dbname = investor,
                    SELECT * FROM staging_area.sa_d_customer WHERE customer_id >
                       (SELECT MAX_CUSTOMER FROM staging_area.max_customer)) as
                       linktable(a int,
                    b varchar(20), c varchar(20), d varchar(50));

INSERT INTO DATA_MART.DW_D_LOCATION(location_id,location_name)
          SELECT *
          FROM DBLINK(host = localhost
                    user = postgres
                    password = postgres
                    dbname = investor,
                    SELECT * FROM staging_area.sa_d_location WHERE location_id >
                        (SELECT max_location FROM staging_area.max_location )) as
                        linktable(a int,
                    b varchar(50));

INSERT INTO DATA_MART.dw_d_promotion(promotion_id,promotion_name,
promotion_description)
          SELECT *
          FROM DBLINK(host = localhost
                    user = postgres
                    password = postgres
                    dbname = investor,
                    SELECT * FROM staging_area.sa_d_promotion WHERE promotion_id
                        > (SELECT max_promotion FROM staging_area.max_promotion))
                        as linktable(a int,
                    b varchar(50), c varchar(100));

INSERT INTO
    DATA_MART.dw_d_subscription_type(subscription_id,subscription_name)
          SELECT *
          FROM DBLINK(host = localhost
                    user = postgres
                    password = postgres
                    dbname = investor,
                    SELECT * FROM staging_area.sa_d_subscription_type WHERE
                        subscription_id > (select max_subscription FROM
                        staging_area.max_subscription_type)) as linktable(a int, b
                        varchar(50));

INSERT INTO DATA_MART.dw_f_complaints(complaint_id,calendar_id,customer_id,
subscription_id,location_id,client_manager_id,category_id,complaint_category)
          SELECT *
          FROM DBLINK(host = localhost
                    user = postgres
                    password = postgres
                    dbname = investor,
                    SELECT * FROM staging_area.sa_f_complaints) as linktable(a
                        int,b int,c int, d int, e int, f int, g int, h
                        varchar(30));

INSERT INTO DATA_MART.dw_f_promotions(f_promotion_id,promotion_id,calendar_id,
customer_id,subscription_id,location_id,client_manager_id,category_id,
promotion_taken)
          SELECT *
          FROM DBLINK(host = localhost
                    user = postgres
                    password = postgres
                    dbname = investor,
                    SELECT * FROM staging_area.sa_f_promotions) as linktable(a
                        int,b int,c int, d int, e int, f int, g int, h int, i
                        boolean);
```

```
INSERT INTO DATA_MART.dw_f_satisfaction(satisfaction_id,calendar_id,customer_id,
subscription_id,location_id,client_manager_id,category_id,overall_satisfaction,
advisor_satisfaction,easiness_interface,usefulness_service,interest_mob_app,
recommend_friend)
           SELECT *
           FROM DBLINK(host = localhost
                      user = postgres
                      password = postgres
                      dbname = investor,
                      SELECT * FROM staging_area.sa_f_satisfaction) as linktable(a
                          int,b int, c int, d int, e int, f int, g int, h int, i
                          int, j int, k int, l int, m int);
END;
$$ LANGUAGE PLPGSQL;
COMMIT;
```

Appendix D

Application Code.

Since the code belonging to the incremental load stage and the testing phase is pretty much the same code with a few modifications, the code shown in this section belongs to the initial load of the data mart.

To see the complete set of files created for this project please visit https://github.com/Birkbeck/msc-computer-science-project-2019-20-files-SergiMP

```
// File model.js
let mongoose = require('mongoose');


mongoose.connect("mongodb://127.0.0.1:27017/survey-data",{ useNewUrlParser: true,
    useUnifiedTopology: true, useCreateIndex: true });

mongoose.connection.on("open", function(){
  console.log("Connection to MongoDB stablished.");
});

let Schema = mongoose.model('survey', {
    reviewer_id: { type: Number, required: true},
    overall_satisfaction: { type: Number, required: true},
    advisor_satisfaction: { type: Number, required: true},
    easiness_interface: { type: Number, required: true},
    usefulness_service: { type: Number, required: true},
    interest_mob_app: { type: Number, required: true},
    recommend_friend: { type: Number, required: true},
    date_submitted: {type: String, required: true}
  }
);

module.exports = Schema;

// File publisher.js

"use strict";
//docker run --name rabbitmq -p 5672:5672 rabbitmq
const amqp = require("amqplib");
const data = require("./data.json");
```

```
let randomIntervalTime = () => (Math.floor(Math.random() * 30) + 1) * 1000;

async function publisher(){

    try {
            const connection = await amqp.connect("amqp://localhost:5672");
            const channel = await connection.createChannel();
            // {durable: true} ensures messages are not wiped-out if server fails.
            const queue = await channel.assertQueue("web-data", {durable: true});
            for(const message in data) {
                    setTimeout(() => {
                        let object = data[message];
                        channel.sendToQueue("web-data",
                            Buffer.from(JSON.stringify(object)),{persistent: true,
                            contentType: 'application/json' });
                        console.log(`{ \n
                    reviewer_id: ${object["content"]["reviewer_id"]},\n
                    overall_satisfaction: ${object["content"]["overall_satisfaction"]},\n
                    advisor_satisfaction: ${object["content"]["advisor_satisfaction"]},\n
                    easiness_interface: ${object["content"]["easiness_interface"]}, \n
                    usefulness_service: ${object["content"]["usefulness_service"]}, \n
                    interest_mob_app: ${object["content"]["interest_mob_app"]},\n
                    recommend_friend: ${object["content"]["recommend_friend"]},\n
                    date_submitted:${object["date_submitted"]}
                }`);
                    }, randomIntervalTime());
            }
    }
    catch (e) {
        console.log(`The following error has been found:\n${e}`);
    }
}

\\ file consumer.js

"use strict";

const amqp = require("amqplib");
const Document = require("./model");
//const notification = require("./email");
let counter = 1;

async function consumer(){

    try {

        const connection = await amqp.connect("amqp://localhost:5672");
        const channel = await connection.createChannel();
        const queue = await channel.assertQueue("web-data");
        channel.prefetch(1);
        channel.consume("web-data", message => {

            let data = JSON.parse(message.content.toString());
            let survey = new Document({reviewer_id: data["content"]["reviewer_id"],
                                    overall_satisfaction:
                                        data["content"]["overall_satisfaction"],
                                    advisor_satisfaction:
                                        data["content"]["advisor_satisfaction"],
                                    easiness_interface:
                                        data["content"]["easiness_interface"],
                                    usefulness_service:
                                        data["content"]["usefulness_service"],
                                    interest_mob_app: data["content"]["interest_mob_app"],
                                    recommend_friend: data["content"]["recommend_friend"],
                                    date_submitted: data["date_submitted"]
                                   });

            //if ( data["content"]["overall_satisfaction"] < 5){
                notification(String(data["content"]["reviewer_id"]))};

             survey.save(function(error,object){
                if(!error){
                    console.log(`Writing message ${counter}`)
                    counter +=1;
                    channel.ack(message);
                }
            })
        })

    }
    catch (e) {
        console.log(`The following error has been found:\n${e}`);
    }
}

// file email.js

"use strict";

const sgMail = require("@sendgrid/mail");
// The API key is not included in the code.
const API_KEY = "supersecretAPI key";

sgMail.setApiKey(API_KEY);

function customerNotification(account){
// For testing purposes my bbk email address was used.
    sgMail.send({
        to: "xxxx@mail.bbk.ac.uk",
        from : "xxxx@mail.bbk.ac.uk",
        subject: `Please contact the account ${account}.`,
        text: `Please contact customer ${account} and follow the guidelines discussed in
            the last meeting.`

    })


}

module.exports = customerNotification;

\\ file csv_data.js

const fs = require("fs");
const csv = require("csv-parser");
```

```
function getComplaints(){
  return new Promise((resolve,reject)=>{
      const csvData = [];
      fs.createReadStream('complaints.csv').pipe(csv())
      .on('data',(data)=> {
          csvData.push(
              [Number(data['customer_id']),
              Number(data['complaint_category']),
              data['customer_comments'].replace(/[\W_]+/g," "),
              String(data['date_received'].trim())])
          })
      .on('end',()=> {
          if(csvData){
              resolve(csvData)
          }else{
              reject(Error("No data was found in the .csv"))
          }
      });
  })
  }

module.exports = getComplaints;

\\ file getMongoData.js

const Document = require("./model");
const { connection } = require("mongoose");

function getMongoData(){
  return new Promise((resolve,reject)=>{
      const collection = Document.find({})
      if(collection){
          resolve(collection)
      }else{
          reject(Error("No data was found in MongoDB"))
      }
  })
}

// file postgreSQL.js

const {Client} = require("pg");
let format = require("pg-format");
const { connection, mongo } = require("mongoose");
const getMongoData = require('./getMongoData');
const CSV = require('./csv_data');
const saQueries = require("./sqlQueries");
const dwQueries = require("./dwQueries");

const client = new Client({
    user: "postgres",
    password: "postgres",
    host: "localhost",
    port:5432,
    database: "investor",
    idleTimeoutMillis: 30000,
    connectionTimeoutMillis: 20000
})

const DW = new Client({
  user: "postgres",
  password: "postgres",
  host: "localhost",
  port:5432,
  database: "dwarehouse",
  idleTimeoutMillis: 30000,
  connectionTimeoutMillis: 20000
})

function initialDWload(){
let mongodata = [];
let csvData = [];
// make the field of the table where data will be inserted of type varchar, you will then
    apply the transformation to_date() in sql
getMongoData().then(collection => {
                      collection.forEach(doc=>{
                      mongodata.push([doc["reviewer_id"],
                                     doc["overall_satisfaction"],
                                     doc["advisor_satisfaction"],
                                     doc["easiness_interface"],
                                     doc["usefulness_service"],
                                     doc["interest_mob_app"],
                                     doc["recommend_friend"],
                                     doc["date_submitted"]]);
                        });
                        console.log(`${mongodata.length} objects from MongoDB have been
                            processed.`);
                        return CSV()
                        })
             .then(data => {
                      let complaintCode = {1: "Service Provided", 2: "Billing Issues", 3:
                          "Technical"}
                      data.forEach(row => {
                      csvData.push([row[0],complaintCode[row[1]],row[3]]);
                        })
                        console.log(`${csvData.length} complaints from the .csv file have
                            been processed.`)
                       return client.connect()
                        })
             .then(()=> client.query(format(saQueries.insertComplaints, csvData ),[]))
             .then(()=> {
               client.query(format(saQueries.intertSurveys, mongodata ),[])
               return client.query(format(saQueries.createTempTable))
             })
             .then((result)=>{
               console.log(`${result.rowCount} rows have been generated in the temporary
                   table.`);
               return client.query(format(saQueries.insertCustomers))
             })
             .then(result => {
```

```
        console.log(`${result.rowCount} customer(s) have been inserted into
            sa_d_customer`);
        return client.query(format(saQueries.insertLocation))
      })
      .then(result=>{
        console.log(`${result.rowCount} rows have been inserted into sa_d_location`);
        return client.query(format(saQueries.insertCategory))
      })
      .then((result)=>{
        console.log(`${result.rowCount} rows have been inserted into
            sa_d_account_category`);
        return client.query(format(saQueries.insertManager))
      })
      .then(result=>{
        console.log(`${result.rowCount} rows have been inserted into
            sa_d_client_manager`);
        return client.query(format(saQueries.insertSubscription))
      })
      .then(result=>{
        console.log(`${result.rowCount} rows have been inserted into
            sa_d_subscription_type`);
        return client.query(format(saQueries.insertPromotion))
      })
      .then(result=>{
        console.log(`${result.rowCount} rows have been inserted into
            sa_d_promotion`);
        return client.query((saQueries.insertCalendar))
      })
      .then(result=>{
        return client.query(('SELECT * FROM STAGING_AREA.SA_D_CALENDAR'))

      })
      .then(result=>{
        console.log(`${result.rowCount} rows have been inserted into sa_d_calendar`)
        return client.query(saQueries.insertFactComplaints)
      })
      .then(result=>{
        return client.query('SELECT * FROM STAGING_AREA.SA_F_COMPLAINTS')
      })
      .then(result=>{
        console.log(`${result.rowCount} rows have been inserted into
            sa_f_complaints`)
        return client.query(saQueries.insertFactPromotions)
      })
      .then(result=>{
        return client.query('SELECT * FROM staging_area.sa_f_promotions')
      })
      .then(result=>{
        console.log(`${result.rowCount} rows have been inserted into
            sa_f_promotions`)
        return client.query(saQueries.insertFactSatisfaction)
      })
      .then(result=>{
        return client.query('SELECT * FROM staging_area.sa_f_satisfaction')
      })
      .then(result=>{
        console.log(`${result.rowCount} rows have been inserted into
            sa_f_satisfaction`)
        return DW.connect();
      })
      .then(result=>{
        console.log("")
        console.log("Connecting to the D.W database.....")
        return DW.query(dwQueries.insertDWcustomer)
      })
      .then(result=>{
        console.log(`Connection to D.W stablished....`)
        console.log(`${result.rowCount} were loaded into dw_d_customer`)
        return DW.query(dwQueries.insertDWaccountCat)
      })
      .then(result=>{
        console.log(`${result.rowCount} rows were inserted into
            dw_d_account_category`)
        return DW.query(dwQueries.insertDWcalendar)
      })
      .then(result=>{
        console.log(`${result.rowCount} rows were inserted into dw_d_calendar.`)
        return DW.query(dwQueries.insertDWclientManager)
      })
      .then(result=>{
        console.log(`${result.rowCount} rows were inserted into
            dw_d_client_manager.`)
        return DW.query(dwQueries.insertDWlocation)
      })
      .then(result =>{
        console.log(`${result.rowCount} rows were inserted into dw_d_location.`)
        return DW.query(dwQueries.insertDWpromotion)
      })
      .then(result=>{
        console.log(`${result.rowCount} rows were inserted into dw_d_promotion`)
        return DW.query(dwQueries.insertDWsubscription)
      })
      .then(result=>{
        console.log(`${result.rowCount} rows were inserted into
            dw_d_subscription_type`)
        return DW.query(dwQueries.inserDWfcomplaints)
      })
      .then(result=>{
        console.log(`${result.rowCount} rows have been inserted into
            dw_f_complaints`)
        return DW.query(dwQueries.insertDWfpromotions)
      })
      .then(result=>{
        console.log(`${result.rowCount} rows have been inserted into
            dw_f_promotions`)
        return DW.query(dwQueries.insertDWfsatisfaction)
      })
      .then(result=>{
        console.log(`${result.rowCount} rows have been inserted into
            dw_f_satisfaction`)
        console.log("")
        console.log("The process of transferring the data to the D.W has been
            completed.")
        return client.query('INSERT INTO staging_area.max_date(max_date) select
            max(calendar_date) from staging_area.sa_d_calendar')
      })
```

```
            .then(result=>{
              return client.query('INSERT INTO staging_area.max_category(max_category)
                  select max(category_id) from staging_area.sa_d_account_category')
            })
            .then(result=>{
              return client.query('INSERT INTO
                  staging_area.max_client_manager(max_manager) select
                  max(client_manager_id) from staging_area.sa_d_client_manager')
            })
            .then(result=>{
              return client.query('INSERT INTO staging_area.max_customer(max_customer)
                  select max(customer_id) from staging_area.sa_d_customer')
            })
            .then(result=>{
              return client.query('INSERT INTO staging_area.max_location(max_location)
                  select max(location_id) from staging_area.sa_d_location')
            })
            .then(result=>{
              return client.query('INSERT INTO staging_area.max_promotion(max_promotion)
                  select max(promotion_id) from staging_area.sa_d_promotion')
            })
            .then(result=>{
              return client.query('INSERT INTO
                  staging_area.max_subscription_type(max_subscription) select
                  max(subscription_id) from staging_area.sa_d_subscription_type')
            })



.catch(e=>console.log(e))
.finally(()=>{
  connection.close();
  client.end();
  DW.end();
})

}

initialDWload();

module.exports = getMongoData;
```

Bibliography

[1] A. G. Alberto, L. R. Carles, R. M. Víctor, *Diseño multidimensional y explotación de datos*, U.O.C, **2020**.

[2] D. Alexander, C. Valentin, *Event Streams in Action*, Manning, **2019**.

[3] T. Ariyachandra, H. J. Watson, Which Data Warehouse Architecture is the Best?, **2008**, `https://www.exhibit.xavier.edu/management_information_systems_faculty/25]`.

[4] C. Arun, D. Mark, What Innovation Leaders Must Know About Open-Source Software, **2019**, `Gartner`.

[5] R. Bruckner, B. List, J. Schiefer, Developing requirements for data warehouse systems with use cases, **2001**.

[6] J. Celko, *Joe Celko's Analytics and OLAP in SQL (The Morgan Kaufmann Series in Data Management Systems)*, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, **2006**.

[7] T. Chenoweth, *A method for developing dimensional data marts. Vol. 46*, pp. 93 –98.

[8] dblink, `https://www.postgresql.org/docs/8.3/contrib-dblink.html`.

[9] W. Eckerson, Gauge Your Data Warehouse Maturity. **2004**.

[10] M. J. Hernandez, J. L. Viescas, *SQL Queries for Mere Mortals: A Hands-on Guide to Data Manipulation in SQL*, Addison-Wesley Longman Publishing Co., Inc., USA, **2000**.

[11] Javascript Promises. `https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise`.

[12] T. John, P. Misra, *Data Lake for Enterprises: Lambda Architecture for Building Enterprise Data Systems*, Packt Publishing, **2017**.

[13] L. Joor, About the prefetch count in RabbitMQ... **2019**, `https://medium.com/@joor.loohuis/about-the-prefetch-count-in-rabbitmq-5f2f5611063b`.

[14] A. Juan-Verdejo, B. Surajbali, H. Baars, H.-G. Kemper, Moving Business Intelligence to cloud environments. **2014**.

[15] M. Khushi, *Journal of cellular biochemistry* **2015**, *116*, 877 –883.

[16] Kimball vs. Inmon in Data Warehouse Architecture, **2017**, `https://www.zentut.com/data-warehouse/kimball-and-inmon-data-warehouse-architectures/`.

[17] M. Kleppmann, *Designing Data Intensive Applications*, O'Reilly Media., Inc., **2017**.

[18] C. Lawrence, S. Jim, *Agile Data Warehouse Design: Collaborative Dimensional Modeling, from Whiteboard to Star Schema*, DecisionOne Press, **2011**.

[19] lizparody23, Understanding Streams in Node.js, **2019**, https://nodesource.com/blog/understanding-streams-in-nodejs/.

[20] M. R. Llave, Business Intelligence and Analytics in Small and Medium-sized Enterprises: A Systematic Literature Review, **2017**.

[21] J. Lovisa, RabbitMQ Exchanges, routing keys and bindings, https://www.cloudamqp.com/blog/2015-09-03-part4-rabbitmq-for-beginners-exchanges-routing-keys-bindings.html.

[22] J. Lovisa, When to use RabbitMQ or Apache Kafka. **2019**, https://www.cloudamqp.com/blog/2019-12-12-when-to-use-rabbitmq-or-apache-kafka.html.

[23] W. Matt, Message Queues & You: 12 Reasons to Use Message Queuing, **2017**, https://stackify.com/message-queues-12-reasons/.

[24] I. Miell, A. H. Sayers, *Docker in Practice*, 1st, Manning Publications Co., USA, **2016**.

[25] H. Ming, J. Verner, Z. Liming, M. Babar, Software quality and agile methods. **2004**.

[26] MongoDB, Export MongoDB data with mongodump, https://docs.mongodb.com/database-tools/mongodump/.

[27] Mongoose Documentation: Connections, https://mongoosejs.com/docs/connections.html.

[28] K. Morfonios, S. Konakas, Y. Ioannidis, N. Kotsis, ROLAP Implementations of the Data Cube. **2007**.

[29] M. Patrou, K. B. Kent, M. Dawson, Scaling Parallelism Under CPU - Intensive Loads in Node.js, **2019**.

[30] PostgreSQL vs MySQL - 2ndQuadrant: PostgreSQL, **2020**, https://www.2ndquadrant.com/en/postgresql/postgresql-vs-mysql/.

[31] L. Raphael, C. Alyson, An Introduction to Change Streams, **2018**, https://www.mongodb.com/blog/post/an-introduction-to-change-streams.

[32] REFRESH MATERIALIZED VIEW- PostgreSQL documentation, https://www.postgresql.org/docs/10/sql-refreshmaterializedview.html.

[33] M. R. C. Santos, R. M. S. Laureano, C. E. R. Albino, How tax audit and tax advisory can benefit from big data analytics tools data analysis and processing in relational databases using SQL Server and Power Pivot Power View in Excel, **2018**.

[34] D. Schuff, K. Corral, R. D. St. Louis, G. Schymik, Enabling self-service BI: A methodology and a case study for a model management warehouse. **2018**.

[35] Snowflake Documentation, https://docs.snowflake.com/en/user-guide/intro-key-concepts.html.

[36] The Soul of the Data Warehouse, Part 2: Drilling Across, **2016**, https://www.kimballgroup.com/2003/04/the-soul-of-the-data-warehouse-part-two-drilling-across/.

[37] Using Clean Architecture for Microservice APIs in Node.js with MongoDB and Express, Youtube, **2019**, https://www.youtube.com/watch?v=CnailTcJV_U&list=PL3t_1qDeay5-4NzDWsvRPWRnGjvIaVUvx&index=101&t=1213s.

[38] S. Wang, H. Wang, Big data for small and medium-sized enterprises (SME): a knowledge management model, **2020**.

[39] S. Whiteley, Introduction to Slowly Changing Dimensions (SCD) Types, **2020**, https://adatis.co.uk/introduction-to-slowly-changing-dimensions-scd-types/.

[40] E. Wilder-James, Breaking Down Data Silos, **2017**, https://hbr.org/2016/12/breaking-down-data-silos.

[41] Work Queues in RabbitMQ, **2020**, https://www.rabbitmq.com/tutorials/tutorial-two-dotnet.html.