Storing and Retrieving Data

Final Project Report

Group 11

Joris Bertens, M20180423, m20180423@novaims.unl.pt, contribution: ~30% of the project
Luis Riveros, M20180753, m20180723@novaims.unl.pt, contribution: ~30% of the project
Joana Lorenz, M20180412, m20180412@novaims.unl.pt, contribution: ~30% of the project
Yuri Ferreira, M20180427, m20180427@novaims.unl.pt, contribution: ~10-15% of the project

Project Overview

Table 1 gives an overview of the steps taken throughout the project. All operations were realized using Pentaho and the available integrations with MongoDB and MySQL as well as Saiku.

First, we conducted data cleaning on the airline, airport and lounge CSV files, further described in [3]. Next, a SQL relational database was created [1], for which the cleaned files of airlines, airports and lounges from [3] served as input. The raw data of the seats CSV was cleaned during the import into the embedded MongoDB model as described in [2]. At this point, we had two databases - the relational SQL database containing the airlines, airports and lounges rating information as well as the embedded MongoDB collection containing information on the seat ratings. Next, as described in [5], we created a Data Warehouse schema, which consisted of fact and dimension tables of the combined information from both database models. We then populated the data warehouse [6] by connecting to the existing databases and populating the dimension and fact tables from the MongoDB collection and the MySQL relational model. Last, based on the fact and dimension tables in the Data Warehouse, we created four data cubes [7] using the Pentaho Schema Workbench as well as Saiku Analytics, in order to answer the proposed questions.

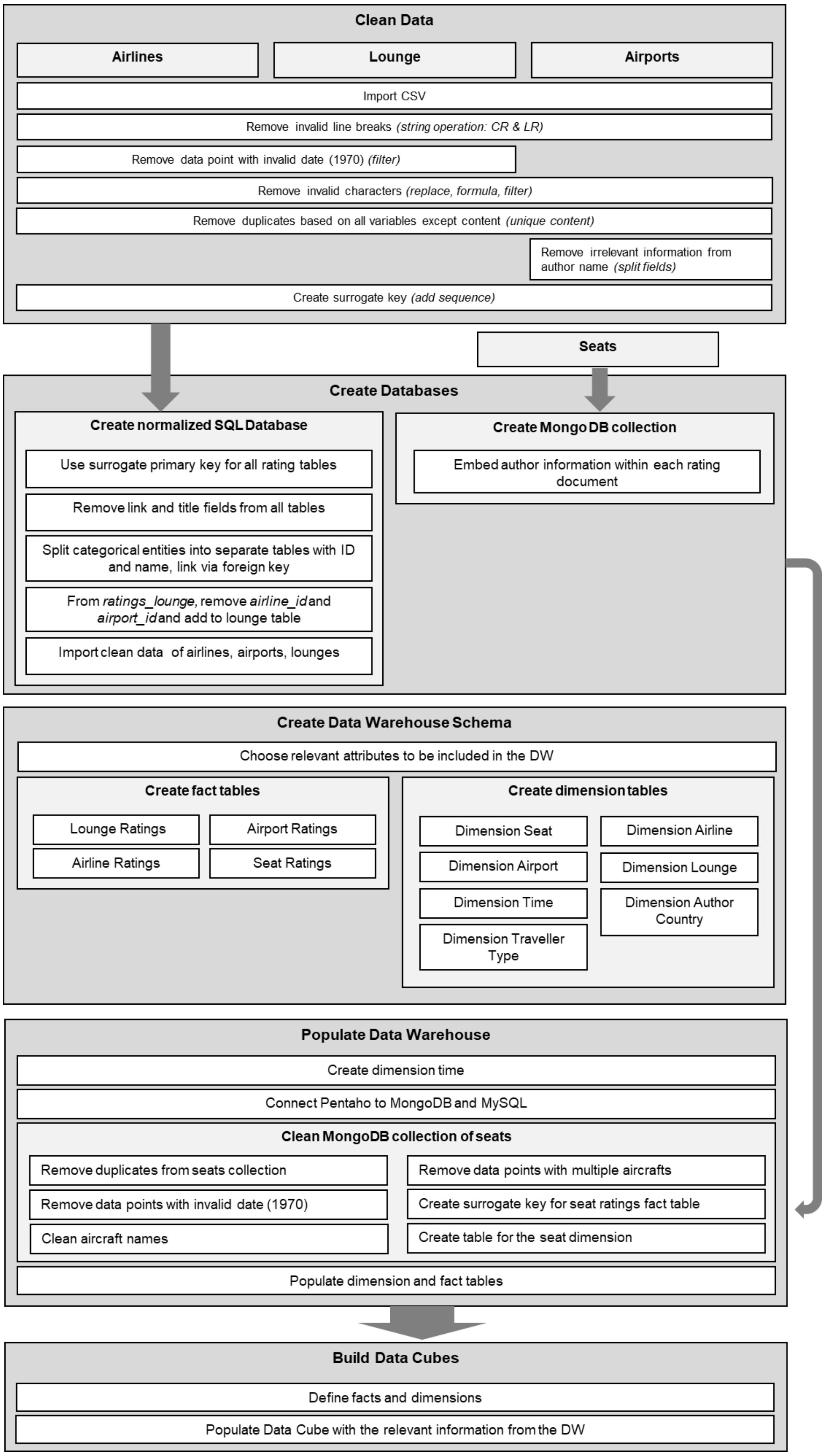

Clean Data
Airlines
Lounge
Airports
Import CSV
Remove invalid line breaks (string operation: CR & LR)
Remove data point with invalid date (1970) (filter)
Remove invalid characters (replace, formula, filter)
Remove duplicates based on all variables except content (unique content)
Remove irrelevant information from author name (split fields)
Create surrogate key (add sequence)
Seats
Create Databases
Create normalized SQL Database
Use surrogate primary key for all rating tables
Remove link and title fields from all tables
Split categorical entities into separate tables with ID and name, link via foreign key
From ratings_lounge, remove airline_id and airport_id and add to lounge table
Import clean data of airlines, airports, lounges
Create MongoDB collection
Embed author information within each rating document
Create Data Warehouse Schema
Choose relevant attributes to be included in the DW
Create fact tables
Lounge Ratings
Airport Ratings
Airline Ratings
Seat Ratings
Create dimension tables
Dimension Seat
Dimension Airline
Dimension Airport
Dimension Lounge
Dimension Time
Dimension Author Country
Dimension Traveller Type
Populate Data Warehouse
Create dimension time
Connect Pentaho to MongoDB and MySQL
Clean MongoDB collection of seats
Remove duplicates from seats collection
Remove data points with multiple aircrafts
Remove data points with invalid date (1970)
Create surrogate key for seat ratings fact table
Clean aircraft names
Create table for the seat dimension
Populate dimension and fact tables
Build Data Cubes
Define facts and dimensions
Populate Data Cube with the relevant information from the DW

Figure 1: Step-by-step Project Overview.

[1] Exercise #1

In MySQL, design a relational database (normalized) based on the files of airline, lounge, and airport. Describe any normalization steps that you might have performed.

In order to guarantee an organized database that allows for easy access and manipulation, while minimizing duplicates and maximizing efficiency, we applied normalization steps to the raw data as presented in *Table 1.*

Airline	**Airport**	**Lounge**
airline_name	airport_name	airline_name
link	link	link
title	title	title
author	author	author
author_country	author_country	author_country
date	date	date
content	content	content
aircraft	experience_airport	lounge_name
type_traveller	date_visit	airport
cabin_flown	type_traveller	lounge_type
route	overall_rating	date_visit
overall_rating	queuing_rating	type_traveller
seat_comfort_rating	terminal_cleanliness_rating	overall_rating
cabin_staff_rating	terminal_seating_rating	comfort_rating
food_beverages_rating	terminal_signs_rating	cleanliness_rating
inflight_entertainment_rating	food_beverages_rating	bar_beverages_rating
ground_service_rating	airport_shopping_rating	catering_rating
wifi_connectivity_rating	wifi_connectivy_rating	washrooms_rating
value_money_rating	airport_staff_rating	wifi_connectivity_rating
recommended	recommended	staff_service_rating
		recommended

Table 1: Overview of the Information in the Original CSV Files.

In order to comply with the **first normal form**, we needed a unique identifier for each data point in the tables. From the already existing columns it was almost impossible to create a composite primary key, since only a very large number of columns would (potentially) lead to a unique identification of a rating. Thus, we chose a surrogate primary key to ensure uniqueness, good performance through an integer representation as well as consistency of the primary key over time. Beside this, the field structure of the provided CSV files complied with the first normal form: In each CSV, one row was designated to a single rating, meaning that no repeating groups of data should be present. During data cleaning [3], we would verify this and clean potential errors, i.e. groups of data in specific columns.

In order to comply with the **second normal form,** all non-primary-key attributes must be fully dependent on the entire primary key. We removed the attributes *link* and *rating* from all tables, since they could easily be derived the other characteristics of a rating. E.g. *link* in the airline table is composed of the base URL '/airline-reviews/' and the *airline_name*. Thus, the fields were considered to be redundant and thus deleted. Beside this, all attributes within the tables depended on the surrogate primary key, meaning that the second normal form was respected.

For the **third normal form,** all attributes must be directly dependent on the primary key. First, we extracted the *airline_id* and *airport_id* from the *rating_lounges* table since they are directly dependent on the *lounge_id* and thus only need to be present in the *lounges* table, which is referenced as a foreign key to the *rating_lounges* table. Next, we decided to split categorical attributes into their own respective tables and assign them an integer ID that would then be used to reference them as foreign keys for the other tables. This was done in order to enable a simple adjustment of these variables when they are referenced to multiple tables and to facilitate the potential addition of further attributes for these separate entities, such as airlines and airports. For example, if someone was to add further information on all airports, such as their specific location, this would only be added to the *airports* table. Equally, if traveller types were to be renamed at a point in the future, this change would not need to be made in each rating table separately, but would only need to be done in the *type_travellers* table.

The resulting relational database schema can be seen in *Figure 2* below.

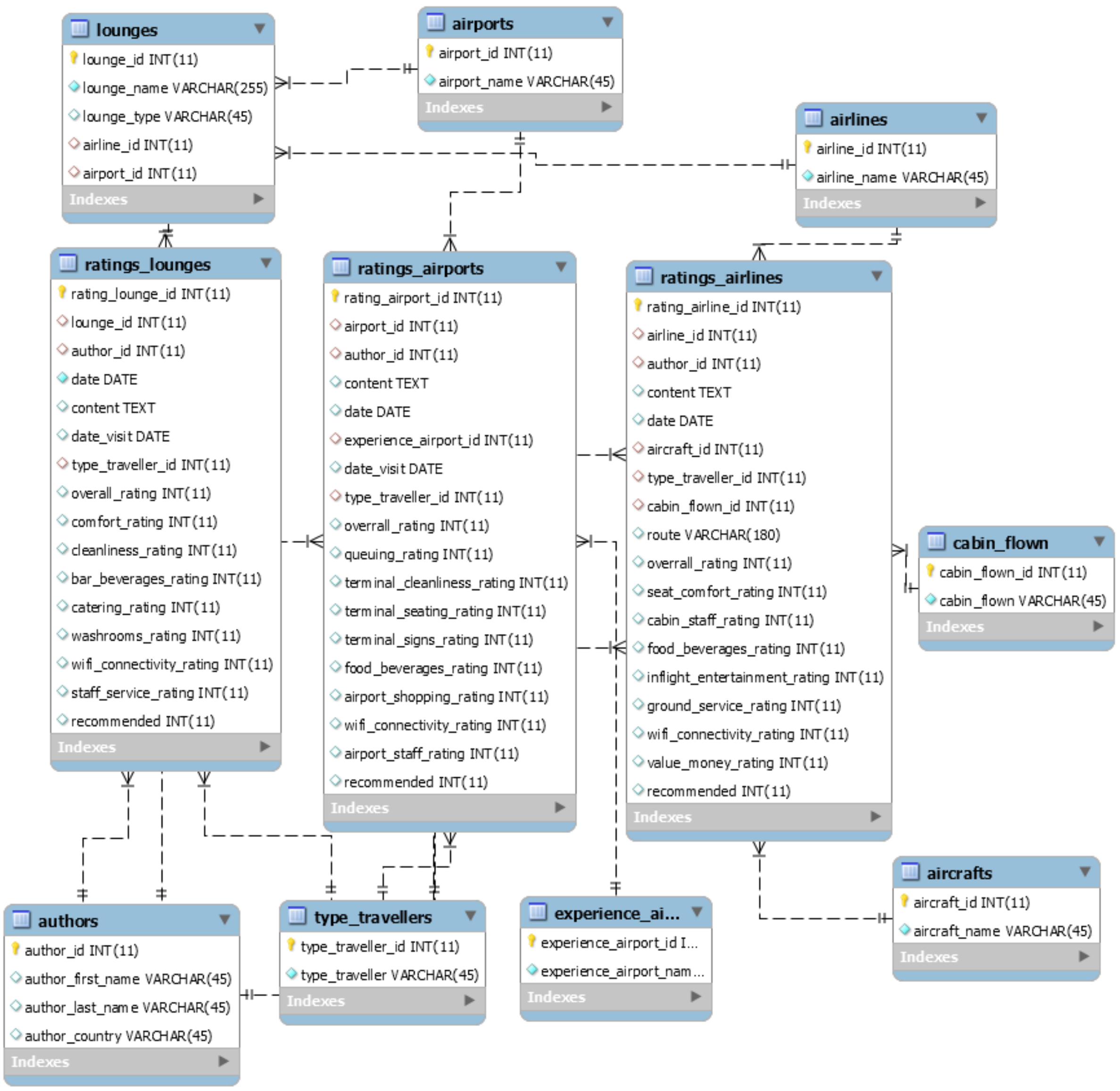


Figure 2: The Relational Database Model.

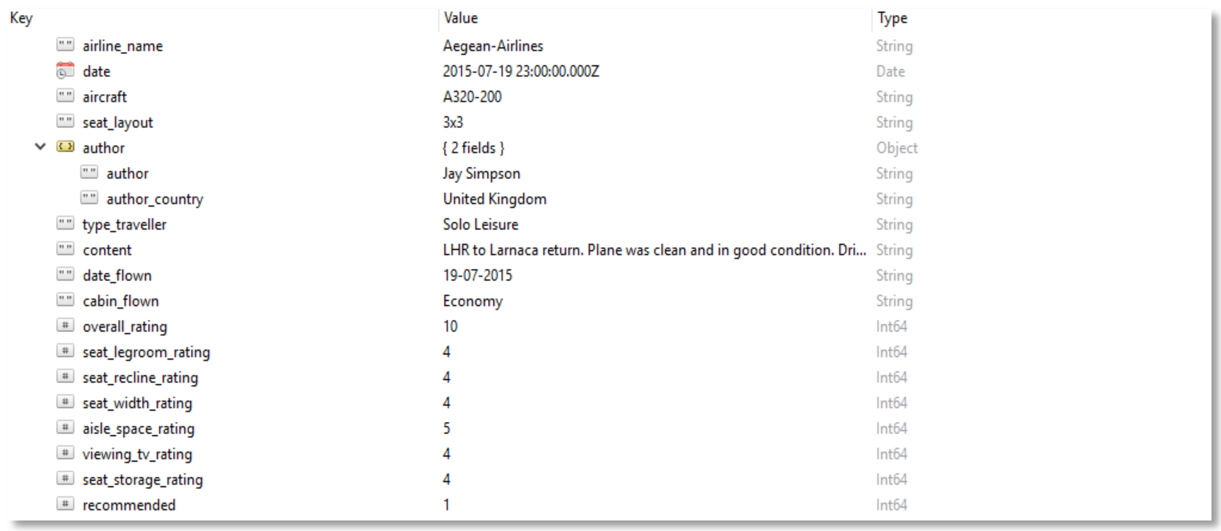
[2] Exercise 2:
In MongoDB create a collection with the information of seats.csv.

As the structure of the documents is a key decision in designing MongoDB data models, a choice had to be made between an embedded structure and a referenced structure. The process of deciding which of the two is more appropriate for the underlying data was done by following the MongoDB design guidelines. The first rule of thumb is that you should always prefer embedding unless there is a compelling reason not to. In this case it makes sense to use an embedded model in order to keep atomicity, to avoid the need to join documents and to be able to retrieve information in one single query.

An important reason to question the use of an embedded model is the storage limit that MongoDB has per document. This corresponds with rule of thumb number three which states that arrays should not grow out of bound and if they do, a referenced model is the preferred option. In our case, we decided to create one document for each rating, embedding the author country. Since each seat rating is done by one author for one airline and one aircraft type, the documents cannot grow out of bound, as it is highly unlikely that much more information will be added in the rating process. Thus, the data restricting of 16 MB's should not be exceeded and documents would remain in a simple structure.

Keeping in mind the second rule of thumb, which states that you don't embed in the case that you want to access an object on its own it, it was decided to only embed the author with the "author name" and "author country". This is because people will be mostly querying on "ratings" and "airlines", which is a compelling reason to not embed these two. The author, however, seems to be rather irrelevant to the business case and can thus be easily embedded, as it is much less requested information.

The information from the seat CSV was imported into MongoDB using Pentaho with the MongoDB connection. An example of the resulting document structure can be seen in *Figure 3.*



Key	Value	Type
airline_name	Aegean-Airlines	String
date	2015-07-19 23:00:00.000Z	Date
aircraft	A320-200	String
seat_layout	3x3	String
author	{ 2 fields }	Object
author	Jay Simpson	String
author_country	United Kingdom	String
type_traveller	Solo Leisure	String
content	LHR to Larnaca return. Plane was clean and in good condition. Dri...	String
date_flown	19-07-2015	String
cabin_flown	Economy	String
overall_rating	10	Int64
seat_legroom_rating	4	Int64
seat_recline_rating	4	Int64
seat_width_rating	4	Int64
aisle_space_rating	5	Int64
viewing_tv_rating	4	Int64
seat_storage_rating	4	Int64
recommended	1	Int64

Figure 3: the Embedded Model Structure.

[3] Exercise 3

Implement an ETL in order to:
a. Find the “duplicates” in airport file.
b. Find the rows that do not provide “valid” information in airline

In the first ETL step, the raw CSV files of *airline, airport* and *lounge* were cleaned before serving as an input to the relational database. This was done in order to ensure validity and the compliance with the normalization forms discussed in [1]. First, we removed invalid line breaks (“/n”) within the *content* field, which lead to the breaking of data points into the next row. We did this by removing the special character ‘carriage return & line feed’ from the field in Pentaho.

Next, duplicates were removed in all CSV files by checking for duplicates of all fields except the *content* field. This is due to the fact that rows exist where everything is the same, except the *content* field, which is only very slightly adjusted (e.g. a different character in a specific position). This could for example be ratings from people who were strongly dissatisfied with the service and wanted to voice their opinion through multiple equally bad ratings for a service, changing the *content* slightly so that the rating was not identified as a duplicate. Since this could skew the statistics, we considered ratings coming from the same author, on the same day for the same entity (airline, airport or lounge) with the same rating values the same, even if the content differed (slightly).

Furthermore, entries with invalid dates, such as entries from 1970, as well as data points with invalid values in the author names were considered invalid and removed, since this only concerned very few data points. Lastly, a surrogate key was created for each table by adding an integer sequence to each unique data point in the table. Through the mentioned adjustments, the files were shortened as shown in *Table 2.*

Csv Input	Initial Data	Data Clean
Airlines	41,396	41,303
Airport	19,997	17,725
Lounges	2,265	2,264
Seats	1,259	1,201

Table 2: Data Points in the Initial Data vs Clean Data.

Further data cleaning steps that were taken included the separation of the author name into first and last name. Even though this was not necessary, we considered it part of the normalization that lead to a cleaner data set. We also removed invalid characters from name fields (e.g. ‘-’), as well as adjusted the handling of special characters such as ‘é’. Also, we adjusted the airline names to the extent that it was safe to do so based on our domain knowledge. We did this by replacing the names of the aircraft producers ‘Airbus’ and ‘Boeing’ by ‘A’ and ‘B’ respectively. This served to reduce duplicates as now values such as “Airbus A330” and “A330” would be recognized as the same value. Data points with unspecified content (‘Please…’) were not removed, since we did not consider this variable to be crucial for the proposed business questions and these data points contained valid information in the other, more important fields.

[5] Exercise 5:
Design a data warehouse (DW). Justify your choice of schema.

To store the data accumulated from the two different sources (MySQL & MongoDB) it was asked to design a data warehouse (DW) and further consider the implementation of data marts. Data marts represent subsets of the data warehouse and are usually oriented at the questions of specific business lines or teams. Since in the case of the rating website, we did not consider there to be multiple audiences with different interests, we did not consider data marts to be necessary and thus only developed a data warehouse.

<u>Defining Facts and Dimensions</u>
The data warehouse is made of Fact tables and Dimension tables. Facts represent measurement data the user is interested in, usually these are in a numeric format. Dimensions are descriptive attributes that categorize and describe data warehouse measures in ways that support meaningful answers to questions.

In order to identify which attributes would be relevant for the data warehouse and their categorization into facts and dimensions, we analyzed the specific business questions provided with the project description. Some fields from the databases were not included in the Data Warehouse, as they did not seem to be relevant to the business questions and thus did not represent dimensions or facts. These fields were *content* and *route.*

As facts, we identified the different ratings, which are associated to a specific process or entity and thus represent four separate Fact tables (seats, airlines, airports, lounges). As dimensions we defined seven attributes that the users seemed to be interested to filter the data on, namely *the author country, traveller type, lounges, airlines, airports*, *seats* and *time*. Since the rating measures share attributes, such as the time or traveller type, dimensions would need to be shared between fact tables.

<u>Choice of the DW schema</u>
As a structure for the data warehouse we could consider a Star schema, Snowflake schema or a Galaxy (Constellation) schema. The choice of the appropriate schema was based on the structure of our data and the prerequisites we had for our data warehouse, based on the decisions on fact and dimension tables described above:

- Multiple fact tables
- Links between dimension tables
- Fact tables linking to the same dimension tables

When choosing the appropriate data warehouse schema for our data prerequisites, it was instantly clear that the Snowflake schema and the Star schema did not match the needs for this specific data warehouse, as we make use of multiple fact tables for the different rating information of *seats, airlines, airports* and *lounges*. The Star and Snowflake schema, however, are only appropriate when a single fact table is needed.

Thus, we aimed for a Galaxy, or Fact Constellation Schema. A Galaxy Schema can be defined as a collection of Star or Snowflake schemas - consisting of multiple fact and dimension tables. The Galaxy schema allows for fact tables to share dimensions, which was another prerequisite for our model. Advantages of using a Galaxy Schema include that it is highly flexible, there is no data redundancy and low memory/space is required. Disadvantages of using a Galaxy Schema include that is has a complicated design and is thus tough to implement and maintain. Also, it requires more complex queries, because of a higher number of joins used to connect the fact and dimension tables.

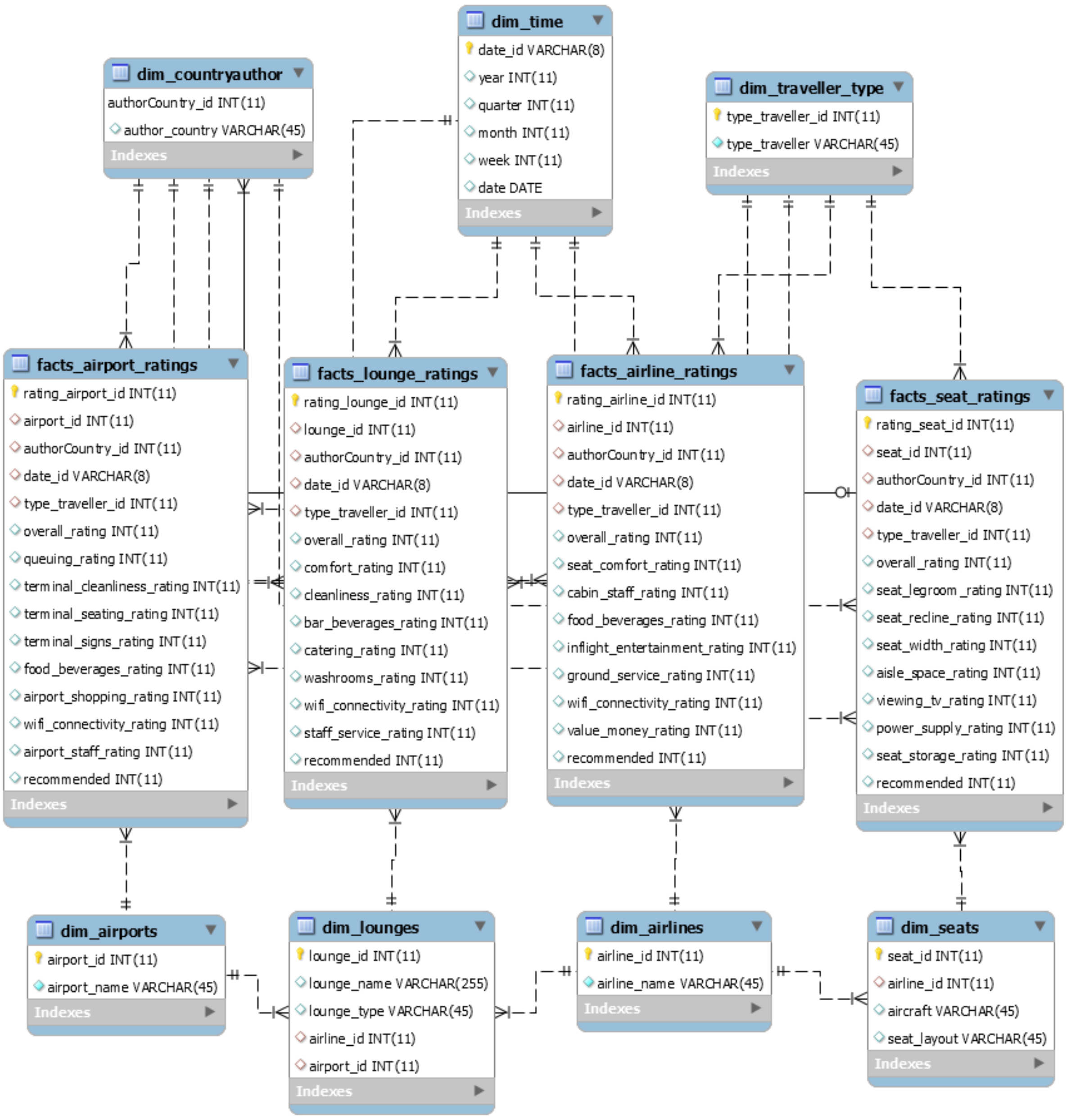


Figure 4: the Galaxy Schema Warehouse applied to flight rating datasets.

[6] Exercise 6:

Populate your DW with the data from your relational database (airline, lounge, airport) and the data from your MongoDB (seat).

In order to populate the data warehouse, we first created the dimension ‘time’ by creating date values starting from the first day of the first year that was present in the data (2000). We created a data point for each possible day with the breakdown of the day of the week, week, month, quarter and year of that specific date. Even though the proposed questions only aimed at month and year, we figured that the availability of time on even more levels would be helpful if data on another aggregation level was required at some point in time. The primary key of the time dimension table is the *date_id*, which represents the ”YYmmdd” string of the specific date. The key then served as a foreign key for all fact tables.

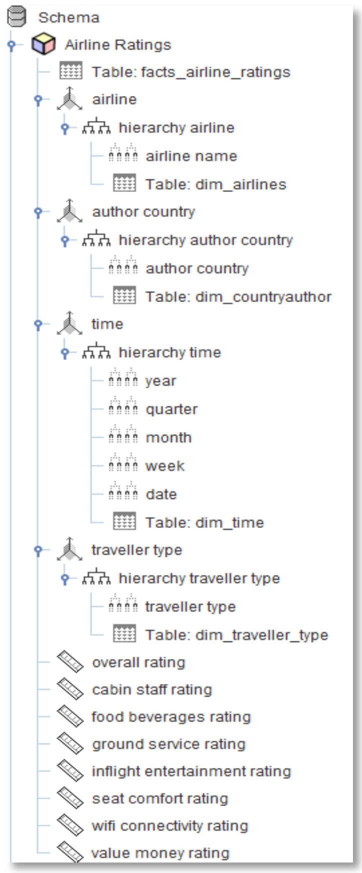
Next, through connections to both MongoDB and MySQL within Pentaho, we populated the chosen dimension and fact tables. Multiple tables, such as the dimension tables of airline or traveller type already existed in the same structure in the relational database and could thus easily be imported and only needed to be renamed.

Before importing the MongoDB data on seat ratings, we cleaned the data by removing duplicates and invalid data points, as this step had not been done for the population of the MongoDB collection, but we considered it relevant for a consistent data warehouse. Furthermore, we deleted rows where two aircrafts were entered. This is because it cannot be assumed for which aircraft the rating was given and thus the data points were considered invalid. Next, we assigned a surrogate key to the *facts_seat_ratings*, because such an integer key did not exist in the MongoDB collection. In order to allow for filtering on *seats*, we added the *airline_id* as well as *aircraft* and *seat layout* specification to the *dim_seats* table, as we assumed a unique combination of airline, aircraft and seat layout to define a seat (type). In the later cube, the user could also choose on which of these attributes from the *dim_seats* table they wanted to query. Each seat was then given a *seat_id* that served as a foreign key for the *fact_seat_ratings* table.

[7] Exercise 7:
Build data cubes to answer the given questions

A data cube can be described as a multidimensional extension of two-dimensional tables. A data cube is generally used to easily interpret data. They are useful when when you want to represent data, i.e. facts, in relation to multiple filters, i.e. dimensions. In a cube, every dimension represents certain characteristic of the data, in this case the dimension tables of the data warehouse. The data that is included inside a data cube makes it possible to analyze almost all the different variations of the included dimensions.

One of the advantages of using a multidimensional data cube is that it usually offers high performance, because it can be indexed directly into the structure of the data cube to gather subsets of data. This also means that the cube becomes sparser when you increase the dimensionality of the cube, which can be undesirable as the storage requirements will get higher. When executing the queries of the created cubes for this project we indeed experienced the cons of



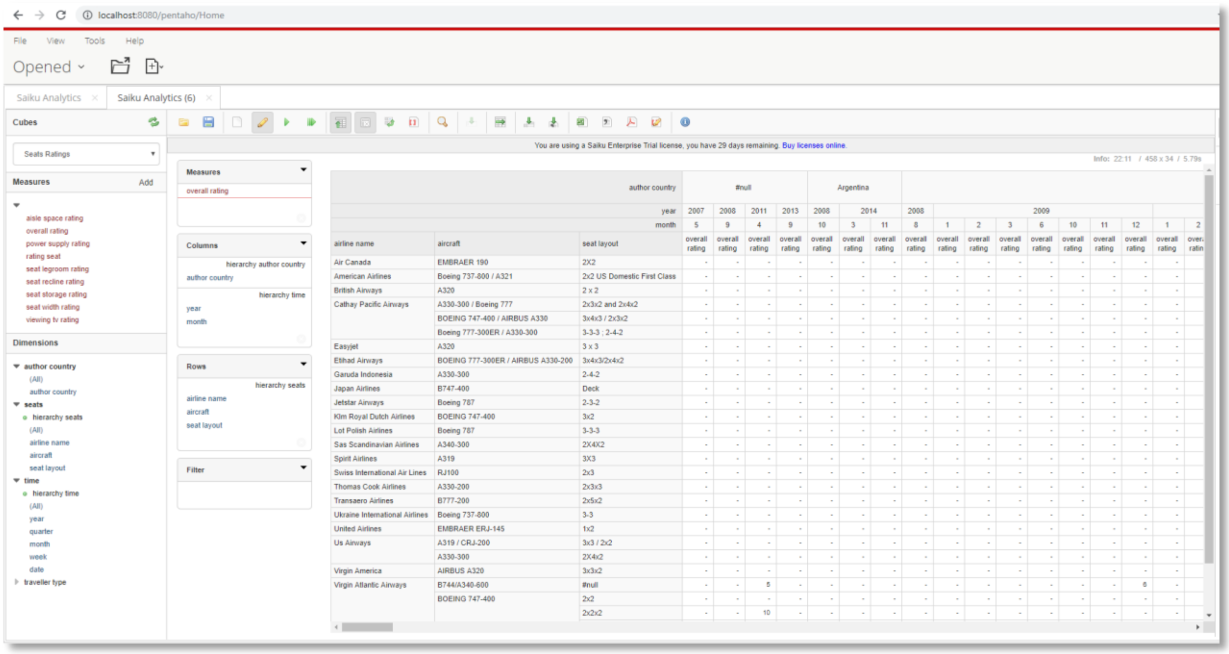
```
Schema
  Airline Ratings
    Table: facts_airline_ratings
    airline
      hierarchy airline
        airline name
        Table: dim_airlines
    author country
      hierarchy author country
        author country
        Table: dim_countryauthor
    time
      hierarchy time
        year
        quarter
        month
        week
        date
        Table: dim_time
    traveller type
      hierarchy traveller type
        traveller type
        Table: dim_traveller_type
    overall rating
    cabin staff rating
    food beverages rating
    ground service rating
    inflight entertainment rating
    seat comfort rating
    wifi connectivity rating
    value money rating
```

Figure 5: *Airline Ratings Data Cube Schema*

creating such a big sparse matrix.

The four data cubes, of which each was based on a single fact table from the data warehouse, were created using the Schema Workbench. First a cube is created, which is set to the specific fact table you want to query on. After that the different dimensions are created, which all consist of labels that are hierarchically ordered. Labels can be seen as dimension attributes (i.e. airline name, author country etc.). These dimensions are all connected to the relevant dimension tables. Beside the specification of the dimensions, we added the measures. Each measure represents a column in the fact table and makes use of a specific aggregator. For this project we were interested in the different ratings per airline/airport/lounge/seat, which made us create a measure for every rating type, using the *average* as the aggregator.

After creating the cubes, the Pentaho Business Intelligence Tools was used to perform the queries on the data cubes. In Saiku Analytics it was easy to show the output of the data cube (see *Figure 5*) and the queries were performed in a process of dragging the right dimensions into the columns/rows. Within the tool, you also have the ability to select slices and subsets of data by filtering in the *Filter* option. The queries can be replicated by connecting to the relevant data cubes.



You are using a Saiku Enterprise Trial license, you have 29 days remaining. Buy licenses online.

Info: 22:11 / 458 x 34 / 5.79s

		author country	#null				Argentina												
		year	2007	2008	2011	2013	2008	2014		2008	2009								
		month	5	9	4	9	10	3	11	8	1	2	3	6	10	11	12	1	2
airline name	aircraft	seat layout	overall rating	overall rating	overall rating	overall rating	overall rating	overall rating	overall rating	overall rating	overall rating	overall rating	overall rating	overall rating	overall rating	overall rating	overall rating	overall rating	overa ratin
Air Canada	EMBRAER 190	2X2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	
American Airlines	Boeing 737-800 / A321	2x2 US Domestic First Class	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	
British Airways	A320	2 x 2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	
Cathay Pacific Airways	A330-300 / Boeing 777	2x3x2 and 2x4x2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	
	BOEING 747-400 / AIRBUS A330	3x4x3 / 2x3x2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	
	Boeing 777-300ER / A330-300	3-3-3 ; 2-4-2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	
Easyjet	A320	3 x 3	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	
Etihad Airways	BOEING 777-300ER / AIRBUS A330-200	3x4x3/2x4x2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	
Garuda Indonesia	A330-300	2-4-2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	
Japan Airlines	B747-400	Deck	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	
Jetstar Airways	Boeing 787	2-3-2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	
Klm Royal Dutch Airlines	BOEING 747-400	3x2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	
Lot Polish Airlines	Boeing 787	3-3-3	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	
Sas Scandinavian Airlines	A340-300	2X4X2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	
Spirit Airlines	A319	3X3	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	
Swiss International Air Lines	RJ100	2x3	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	
Thomas Cook Airlines	A330-200	2x3x3	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	
Transaero Airlines	B777-200	2x5x2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	
Ukraine International Airlines	Boeing 737-800	3-3	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	
United Airlines	EMBRAER ERJ-145	1x2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	
Us Airways	A319 / CRJ-200	3x3 / 2x2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	
	A330-300	2X4x2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	
Virgin America	AIRBUS A320	3x3x2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	
Virgin Atlantic Airways	B744/A340-600	#null	-	-	5	-	-	-	-	-	-	-	-	-	-	-	6	-	
	BOEING 747-400	2x2	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	
		2x2x2	-	-	10	-	-	-	-	-	-	-	-	-	-	-	-	-	

Figure 5: *Query: ratings per seat, per country of author, per traveller type, per month and year*

[8] Limitations

During this project, we found two main limitations. First, data cleaning was restricted, as in many cases, specific domain knowledge would have been required in order to determine the unique values. Examples of such cases are fields that are provided by the user as a string without any pre-defined values, such as the lounge name as well as the aircraft information. In both cases, users may input a slightly differing name and it is very difficult to determine whether this represents a different entity or just a different naming of the same entity or a typo. Even though we already cleaned the aircraft names to a certain extent, we could still not be sure whether certain values represented duplicates. Examples include ‘CRJ’ vs.

‘Canadaair RJ’ and ‘Aeroflot Business Class Lounge’ vs. ‘Aeroflot Business Class Lounge s’. With sufficient domain knowledge and the development of precise rules, the dataset could thus be cleaned to a much greater extent. This also means that at this moment, the data most probably still contains duplicates, which we are aware of but considered to be out-of-scope of this project.

The second challenge, which could be overcome through the consultation of websites and books, but turned out to be rather time consuming, was the integration of different programs that were needed for the specific tasks, into one workflow. For example, some time had to be invested in order to establish a connection between Saiku Analytics and the created Pentaho OLAP cube.