### *University of Pittsburgh*

### *School of Computing and Information*

**INFSCI 2711: Advanced Database Management System**

***Spring 2020***

**Final Project Report**

**A close up of a sign

Description automatically generated**

**Submitted By:**

|  |  |  |
| --- | --- | --- |
| **Name** | **Email Address** | **Student ID** |
| Aishwarya Jakka |  |  |
| Shruti Gupta | [shg104@pitt.edu](mailto:shg104@pitt.edu) |  |
| Piu Mallick | [pim16@pitt.edu](mailto:pim16@pitt.edu) | 4374215 |
| Reshma Sara Pothen | [rep83@pitt.edu](mailto:rep83@pitt.edu) |  |
| Soham Bhatnagar | [sob38@pitt.edu](mailto:sob38@pitt.edu) |  |
| Kenny Wu |  |  |
| Kwesi Randolph Aguillera | [kra40@pitt.edu](mailto:kra40@pitt.edu) |  |

**Online Retail Datawarehouse**

1. **Introduction**
2. **Sub-Teams**

Teams responsible for exploring various areas in the project:

* **Coordination**
  + Aishwarya Jakka
* **Initial Data Cleaning**
  + Kwesi Randolph Aguillera
* **Further Data Cleaning**
  + Respective Teams
* **SQL-DB**
  + Shruti Gupta
  + Aishwarya Jakka
* **MongoDB**
  + Piu Mallick
  + Reshma Sara Pothen
* **Neo4j**
  + Soham Bhatnagar
  + Kenny Wu
  + Kwesi Randolph Aguillera
  + Aishwarya Jakka
* **Front-end & Back-end integration**
  + Kenny Wu
  + Kwesi Randolph Aguillera
* **Documentation**
  + Piu Mallick
  + Reshma Sara Pothen
  + Shruti Gupta

1. **Dataset Description**

This [**Online Retail II**](https://archive.ics.uci.edu/ml/datasets/Online+Retail+II) data set (the original dataset is a **excel file**) contains all the transactions occurring for a **UK-based** and registered, non-store online retail between **01/12/2009 (1st Dec 2009)** and **09/12/2011 (9th Dec 2011)**. The company mainly sells **unique all-occasion giftware**. Many customers of the company are wholesalers.

* 1. **Attribute Information**

The dataset has the following attributes:

* **InvoiceNo**: Invoice number. Nominal. A 6-digit integral number uniquely assigned to each transaction. If this code starts with the letter 'c', it indicates a cancellation.
* **StockCode**: Product (item) code. Nominal. A 5-digit integral number uniquely assigned to each distinct product.
* **Description**: Product (item) name. Nominal.
* **Quantity**: The quantities of each product (item) per transaction. Numeric.
* **InvoiceDate**: Invoice date and time. Numeric. The day and time when a transaction was generated.
* **UnitPrice**: Unit price. Numeric. Product price per unit in sterling (Â£).
* **CustomerID**: Customer number. Nominal. A 5-digit integral number uniquely assigned to each customer.
* **Country**: Country name. Nominal. The name of the country where a customer resides.
  1. **Github Link**

The Github link for the project repository: [Click here](https://github.com/Aishwaryajakka/infsci2711_finalproject)

1. **Initial Data-Cleaning**

Following are the initial set of steps to clean the data:

* Removed records with blank CustomerIDs.
* Removed records weird StockCodes.
* Corrected Description where possible.
* Removed records where Description cannot be corrected.
* Removed zero Price items.
* Removed “Unspecified” country records.
* Changed Negative quantity records to positive quantity values.
* Separate Datetime into Date and Time columns.
* Calculate SubTotal for each record.

Following are the column stats for the year **2009-2010**:

* No. of records before cleaning: 525461
* No. of records after cleaning: 413084
* InvoiceNo Range: 489437 – 528618 / C489449 – C538168
* StockCode Range: 10002 – 90208 / 10123C – 90214Z & SP1002
* Quantity Range: 1 – 19,152
* Date Range: Dec 2009 – Dec 2010
* Time Range: 7:01 AM – 9:52 PM
* UnitPrice Range: $0.08 - $295.00
* CustomerID Range: 12346 – 18287
* No. of Countries: 36
* SubTotal Range: $0.06 - $15,818.40

Following are the column stats for the year **2010-2011**:

* No. of records before cleaning: 541910
* No. of records after cleaning: 300766
* InvoiceNo Range: 536365 – 564272 / C536383 – C564276
* StockCode Range: 10002 – 90208 / 10123C – 90214Z
* Quantity Range: 1 – 80,995
* Date Range: Dec 2010 – Dec 2001
* Time Range: 7:35 AM – 8:38 PM
* UnitPrice Range: $0.08 - $649.50
* CustomerID Range: 12347 - 18287

The **initial data cleaning** was done manually in **Microsoft Excel**.

Post initial data cleaning, some extra cleaning was done in order to meet various database needs, which will be discussed later in this report.

1. **Possible Aggregation Queries**

Following are the probable statistical and aggregation queries that the owner/manager of the store may want to view in order to get the growth report of their store:

1. What time of the day (which hour of the day) is the sale maximum per country?
2. What is the annual TotalSales per product?
3. What is the top product per year?
4. What is the top product per country?
5. Which item is sold below a certain threshold value? Or, what are the under-performed products based on the average sales last year?
6. Which customer spends the most (per country/overall)?
7. What is the best-selling month per country? (Given the year range, 2009-2011)
8. What is the best-selling product per month? (Given the year range, 2009-2011)
9. What is the change in TotalSales per country per year (Trend of Sales)?
10. What is the average spending of a customer per country? (TotalSales/Number of customers)
11. What is the frequently purchased item per customer?
12. **STAR Schema: Design**

Based on the queries (which we want to ask the database) stated above, we have designed the **DIMENSIONS** and **MEASURES**, which we popularly call as a **STAR schema**. Hence, first comes the aggregate queries, and based on the queries, we would like to build our aggregated database.

**DIMENSIONS**: Customer, Stock, Time

**MEASURES**: Quantity, Sales

**STAR SCHEMA:**

Hence, the **tables** (with all the **attributes**) we would be considering are as follows:

* **FACT** (*CustomerID, Year, Month, Day, Hour, Minute, StockCode, Quantity, TotalSales*)
* **CUSTOMER\_DIM** (*CustomerID, Country*)
* **STOCK\_DIM** (*StockCode, Description, MaxUnitPrice*)
* **TIME\_DIM** (*Year, Month, Day, Hour, Minute*)

  However, the above table definitions may change depending on the databases we would be using.

1. **Databases**

We have chosen 3 databases, namely **SQL, MongoDB** and **Neo4j** for building our **Online Retail Datawarehouse**. We would be integrating each of the databases with front-end, which we will discuss later in the document.

* 1. **SQL DB**
     1. **Creating the ODB (Operational Database)**
* **Extra data-cleaning for SQL DB:**

Steps followed to clean the **Cleaned2009-2010.xlsx** and **Cleaned 2010-2011.xlsx.** (actual data files) from the data folder:

* Renamed column "Customer ID" to Customer\_ID
* Changed format of the InvoiceDate from MM/DD/YY to YYYY-MM-DD
* Changed format of the InvoiceTime from HH:MM AM/PM to HH:MM:SS (24 hour format)
* Changed format of the SubTotal to remove the "," formatting of numeric value.
* Saved the file as <filename>.xlsx
* Then saved file as <filename>.csv
* The above steps are repeated for the other file (2010-2011) as well.
  + 1. **Steps followed to import the data in phpMyAdmin:**
* Created SQL insert script
* Inserted the script in batches of 25000 as this was the limit for insertion
* The ODB creation process took a significant amount of time as import functionality didn’t work with the csv.
* Tried MySQL workbench to insert the script. All insertion done in one go
  + 1. **Creating the ADB (Analytical Database)**

**• Steps followed to create the ADB, relevant tables and furnish data**

* Created a new database ‘online\_retail\_adb’ .
* Once the Analytical Database was created, we were ready to create the table specified in the schema – specifically, 3 dimension tables – ‘customerdim’, ‘stockdim’ and ‘timedim’ along with the ‘fact’ table.
* For each of these tables, we used the insert scripts to move data from the ODB to the relevant ADB table.
* For the FACT table, we extracted the relevant fields from the ODB. Note that even though the ODB had ‘Sub Total’ as a field, it was renamed ‘TotalSales’ in the FACT table.
* For the customerdim table:

The customerdim table required the ‘Customer\_ID’ and ‘Country’ fields. This was done using a ‘groupby’ operator. To remove duplicates in CustomerID, operator was used. Finally, used insertion script to insert data into customerdim.

* For the stockdim table:

The stockdim table required the ‘StockCode’, ‘Description’ and ‘Unit\_Price’ fields.

There was different unit price for single StockCode, so we took the maximum unit price.

Insertion script was used to insert data into stockdim table

* For the timedim table:

The timedim table required the ‘Day’, ‘Month’, ‘Year’, ‘Hour’ and ‘Minute’ fields. Distinct rows were inserted from ODB

* **Challenge Faced:**

Since the database is too big, it took time to insert data.

* + 1. **Aggregation Queries**
* What time of the day is the sale maximum per country?

This query required three join, making the query very expensive. It took around 5 minutes to run

In order to improve the run time, we created a materialized view named ‘mview\_sum\_sale\_country’. Using the materialized view, run time decreased drastically.

* Annual total sales of per product:

This query required three join, making the query very expensive. It took around 7 minutes to run

In order to improve the run time, we created a materialized view named ‘mview\_totalsale\_product’. Using the materialized view, run time decreased drastically.

* Top product per year

This query required three join, making the query very expensive. It took around 5 minutes to run

In order to improve the run time, we created a materialized view named ‘mview\_totalsale\_product’. Using the materialized view, run time decreased drastically.

* Top product per country

This query required three join, making the query very expensive. It took around 5 minutes to run

In order to improve the run time, we created a materialized view named ‘mview\_totalsale\_product’.Using the materialized view, run time decreased drastically.

* Which item is sold below the threshold value (under-performed products)

This query required three join, making the query very expensive. It took around 7 minutes to run

In order to improve the run time, we created a materialized view named ‘mview\_quantity\_product\_year’. Using the materialized view, run time decreased drastically.

* Which customer spends the most per country

This query required three join, making the query very expensive. It took around 5 minutes to run

In order to improve the run time, we created a materialized view named ‘mview\_customer\_total\_country’. Using the materialized view, run time decreased drastically.

* Best-selling month (per country) [2009-2011]

This query required three join, making the query very expensive. It took around 5 minutes to run

In order to improve the run time, we created a materialized view named ‘mview\_sale\_month’. Using the materialized view, run time decreased drastically.

* Best-selling product per month [2009-2011]

This query required three join, making the query very expensive. It took around 5 minutes to run

In order to improve the run time, we created a materialized view named ‘mview\_quantity\_product’. Using the materialized view, run time decreased drastically.

* What is the change in total sales per country per year (trend of sales)? (

This query required three join, making the query very expensive. It took around 5 minutes to run

In order to improve the run time, we created a materialized view named ‘mview\_yearsale\_country’. Using the materialized view, run time decreased drastically.

* Average spending of a customer in that country (Total sales/ no of customer)

This query required three join, making the query very expensive. It took around 5 minutes to run

In order to improve the run time, we created a materialized view named ‘mview\_customer\_total\_country’. Using the materialized view, run time decreased drastically.

* Frequently purchased item per customer:

This query used same table, so materialized view was not required.

* + 1. Advantages and disadvantages of MySql:
* Advantages are that all the queries run in Mysql. It is possible to run every type of query by applying multiple joins
* Disadvantage is it takes a lot of time to run. With multiple joins, it becomes expensive

* 1. **MongoDB**
     1. **Creating the ODB (Operational Database)**
* **Extra data-cleaning for MongoDB:**

Steps followed to clean the **Cleaned2009-2010.xlsx** and **Cleaned 2010-2011.xlsx.** (actual data files) from the **data** folder:

* Changed the **file type** to **csv**.
* Loaded the **csv files** to **Jupyter Notebook** (**Python**) and converted them to dataframes.
* Concatenated two dataframes using the ‘**append’** function.
* The attribute ‘**Customer ID**’ is renames to ‘**CustomerID**’.
* The attributes ‘**InvoiveDate**’ and ‘**InvoiceTime**’ are then concatenated and renamed to ‘**InvoiceDateTime**’. Later, they are split to **Year** (**YYYY**), **Month** (**MM**), **Day** (**DD**), **Hour** (**HH**) and **Minute** (**MM**).
* The dataframe is then exported to the data folder as ‘**Online\_Retail\_DB.csv**’.
* **Steps followed to import the csv file to MongoDB:**
* Go to the **terminal** and login to **MongoDB** (by typing ‘**mongo**’ to stay in the same terminal or ‘**mongod**’ to switch to a new terminal).

**Pre-requisite**: Prior installation of MongoDB in the system and the mongodb service to be running.

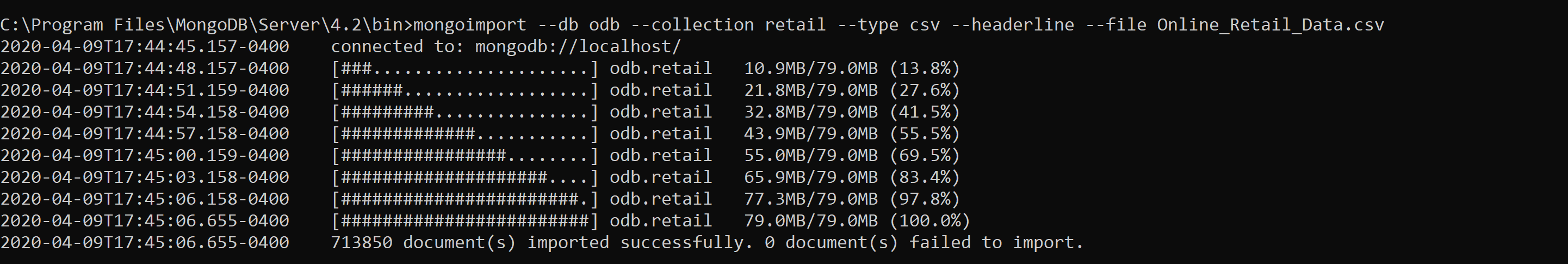
* Switch to or create a new database – ‘**odb**’ in MongoDB with the help of the following command in the terminal:

**use odb**

* In a separate terminal, navigate to the path where the cleaned data file is present. The **MongoDB import command** is executed outside the mongo shell - in a normal terminal, with the help of the following command:

**mongoimport --db odb --collection retail --type csv --headerline --file Online\_Retail\_Data.csv**

* At the end of this process, the relevant documents were successfully inserted. This is shown below.



* **Challenges Faced:**

The main challenge in ODB database was to split the InvoiceDate and InvoiceTime into individual Year, Month, Day, Hour, Minute (as the Date and Time were already split in the initial part of the data cleaning). Hence, we imported the csv file to Python to do the date and time splitting. Also, we appended two csv files to a single file. This could have also been achieved in the MongoDB as well. However, each import command in MongoDB (one for each csv file) was taking around 40 seconds. So, importing 2 csv files would have taken around 80 seconds. After combining the two files into one file, the import command took around 45 seconds, which means we saved around 35 seconds. Though, it is a matter of 35 seconds only. However, we are not sure if this strategy would work if any dataset is significantly larger than our dataset.

* + 1. **Creating the ADB (Analytical Database)**

**•** **Steps followed to create the ADB, relevant tables and furnish data**

* Created a new database ‘adb’ in MongoDB.

Specified ‘adb’ as the current database using the following command: **use adb**

* Once the Analytical Database was created, we were ready to create the table specified in the schema – specifically, 3 dimension tables (called collections in MongoDB) - customer\_dim, stock\_dim and time\_dim along with the FACT table.
* For each of these collections, we used the .getSiblingDB function to move data from the ODB to the relevant ADB table.
* For the FACT collection, we extracted the relevant fields from the ODB. Note that even though the ODB had ‘Sub Total’ as a field, it was renamed ‘TotalSales’ in the FACT table. The ‘forEach’ function is used to loop through the retail collection for all records.

db.getSiblingDB('odb')['retail'].find( { },

{ CustomerID: 1,

Day: 1,

Month: 1,

Year: 1,

Hour: 1,

Minute: 1,

StockCode: 1,

Quantity: 1,

SubTotal: 1

}

).forEach(function(rec){db.fact.insert(rec) } )

* For the customer\_dim collection:

The customer\_dim collection required the ‘CustomerID’ and ‘Country’ fields. This was done using a ‘$group’ operator. To remove duplicates in CustomerID, the ‘$match’ operator was used. Finally, as with the FACT table data insertion, ‘forEach’ was used to loop through the retail collection in the ODB and insert every record.

db.getSiblingDB('odb')['retail'].aggregate(

{"$group" :

{ "\_id": {CustomerID:"$CustomerID",

Country:"$Country"} }

},

{"$match":

{"\_id" :

{ "$ne" : null }

}

},

{"$project":

{CustomerID : "$\_id.CustomerID",

Country: "$\_id.Country", "\_id" : 0}

}

).forEach( function (rec) { db.customer\_dim.insert(rec) } )

* For the stock\_dim collection:

The stock\_dim collection required the ‘StockCode’, ‘Description’ and ‘MaxUnitPrice’ fields. Initially, an index was created for ‘StockCode’ to allow for faster query execution. This was followed by an aggregation query on the retail collection. Note that Price was changed to MaxUnitPrice.

db.stock\_dim.createIndex( { "StockCode": 1 } )

db.getSiblingDB('odb')['retail'].aggregate (

[{

$project: {

StockCode: '$StockCode',

Description: '$Description',

Price: '$Price',

desc\_lec: {

$strLenCP: '$Description'

}

}

}, {

$group: {

\_id: '$StockCode',

Description: {

$first: '$Description'

},

MaxUnitPrice: {

$max: '$Price'

},

max\_desc\_len: {

$max: '$desc\_lec'

}

}

}, {

$project: {

StockCode: '$\_id',

Description: 1,

MaxUnitPrice: 1,

\_id: 0

}

}]

).forEach( function (rec) {

db.stock\_dim.insert(rec)

})

* For the time\_dim collection:

The time\_dim collection necessitated a two-step approach to insertion. Initially, the data for the ‘Day’, ‘Month’, ‘Year’, ‘Hour’ and ‘Minute’ fields were fetched, using the forEach function, from the retail collection and inserted into a temporary table ‘time\_dim\_temp’. Once done, the data was aggregated into ‘Day’, ‘Month’, ‘Year’, ‘Hour’ and ‘Minute’ into the final ‘time\_dim’ collection.

db.getSiblingDB('odb')['retail'].find( { },

{ Day: 1,

Month: 1,

Year: 1,

Hour: 1,

Minute: 1

}).forEach( function(rec){ db.time\_dim\_temp.insert(rec) } )

db.time\_dim\_temp.aggregate (

[{

$group: {

\_id: {

Year: "$Year",

Month: "$Month",

Day: "$Day",

Hour: "$Hour",

Minute: "$Minute"

}

}

}

]).forEach( function(rec){ db.time\_dim.insert(rec) } )

At the end of this process, all four collections were populated. The relevant statistics are displayed below.



* **Challenges Faced:**

An interesting challenge in MongoDB is that, unlike SQL, there is no direct option to insert files in a collection by selecting them from another. Instead, we had to read the data record by record and utilize the ‘forEach’ function to insert them. However, a major advantage of MongoDB is that, unlike relational databases, insertion of over 700,000 records took relatively minimal time. This presents major advantages in terms of processing time but also the volume the database can handle. Unlike in SQL, there was absolutely no need to spit the initial cleaned csv file. The FACT collection insertion took 5 minutes, ‘Customer\_dim’, 6 minutes and ‘Stock\_dim’, 4 minutes. ‘Time\_dim’ was split due to the creation of the ‘time\_dim\_temp’ collection. This took 4 minutes and 53 seconds. The creation of the final ‘time\_dim’ collection only took 13 seconds.

* + 1. **Aggregation Queries**
* **What time of the day is the sale maximum per country?**

We performed this query both with and without an index. Without and index, this was the longest running query – taking a whopping 16 minutes. With an index, it took a much more reasonable 45 seconds.

db.customer\_dim.createIndex( { "CustomerID": 1 } )

db.fact.aggregate(

[{

$lookup: {

from: 'customer\_dim',

localField: 'CustomerID',

foreignField: 'CustomerID',

as: 'customer\_rec'

}

}, {

$unwind: {

path: '$customer\_rec'

}

}, {

$project: {

Hour: 1,

Country: '$customer\_rec.Country',

TotalSales: 1,

\_id: 0

}

}, {

$group: {

\_id: {

Hour: '$Hour',

Country: '$Country'

},

TotalSalePerCountry: {

$sum: '$TotalSales'

}

}

}, {

$project: {

Hour: '$\_id.Hour',

Country: '$\_id.Country',

TotalSalePerCountry: 1,

rec: '$$ROOT',

\_id: 0

}

}, {

$group: {

\_id: '$Country',

MaxSalePerHourPerCountry: {

$max: '$TotalSalePerCountry'

},

rec: {

$first: '$rec'

}

}

}, {

$project: {

Country: '$\_id',

MaxSalePerHourPerCountry: 1,

Hour: '$rec.\_id.Hour'

}

}, {

$sort: {

Country: 1

}

}]

)

* **Annual Total Sales Per Product?**

This query required a grouping of ‘Year’, ‘Stock’ and sum of ‘TotalSales’ to produce ‘TotalAnnualSales’. Once the result set was obtained, ‘$sort’ was used to sort both ‘Stock’ and ‘Year’ in an ascending order. This took a very minimal 2 seconds to execute.

db.fact.aggregate (

[{

$group: {

\_id: {

"Year": "$Year",

"Stock": "$StockCode"

},

TotalAnnualSales: {

$sum: "$TotalSales"

}

}

}, {

$project: {

Year: "$\_id.Year",

Stock: "$\_id.Stock",

TotalAnnualSales: 1,

\_id: 0

}

}, {

$sort: {

Stock: 1,

Year: 1

}

}]

)

* **Top Product Per Year**

This query required a grouping of ‘Year’, ‘Stock’ and sum of ‘TotalSales’ to produce ‘TotalAnnualSales’. The results were then sorted by ‘Year’. Execution time was once again, minimal at 3 seconds.

db.fact.aggregate (

[{

$group: {

\_id: {

"Year": "$Year",

"StockCode": "$StockCode"

},

TotalAnnualSales: {

$max: {

$sum: "$TotalSales"

}

}

}

}, {

$project: {

TotalAnnualSales: 1,

stock\_doc: "$$ROOT"

}

}, {

$group: {

\_id: "$\_id.Year",

TotalAnnualSales: {

$max: "$TotalAnnualSales"

},

stock\_doc: {

$first: "$stock\_doc"

}

}

}, {

$project: {

Year: "$\_id",

StockCode: "$stock\_doc.\_id.StockCode",

TotalAnnualSales: 1,

\_id: 0

}

}, {

$sort: {

Year: 1

}

}]

)

* **Top Product Per Country**

This query required a grouping of ‘Country’, ‘Stock’ and sum of ‘TotalSales’ to produce ‘TotalAnnualSales’. The results were then sorted by ‘Country’. Execution time was slightly longer at 52 seconds.

db.fact.aggregate (

[{

$lookup: {

from: 'customer\_dim',

localField: 'CustomerID',

foreignField: 'CustomerID',

as: 'customer\_rec'

}

}, {

$unwind: {

path: '$customer\_rec'

}

}, {

$group: {

\_id: {

StockCode: '$StockCode',

Country: '$customer\_rec.Country'

},

TotalSalePsPc: {

$sum: '$TotalSales'

},

all\_sales: {

$push: {

sale\_rec: '$$ROOT'

}

}

}

}, {

$project: {

StockCode: '$\_id.StockCode',

Country: '$\_id.Country',

TotalSalePsPc: 1,

rec: '$$ROOT',

\_id: 0

}

}, {

$group: {

\_id: '$Country',

TotalSalePerStock: {

$max: '$TotalSalePsPc'

},

rec: {

$first: '$rec'

}

}

}, {

$project: {

StockCode: '$rec.\_id.StockCode',

Country: '$\_id',

TotalSalePerStock: 1,

\_id: 0

}

}, {

$sort: {

TotalSalePerStock: -1

}

}], {

allowDiskUse: true

}

)

* **Query 5**

This query required a

* **Which customer spends the most (per country or overall)**

This is a two-part query. We first attempted to find out which customer spends most overall. This query equired a grouping of ‘CustomerID’ and ‘TotalSales’. An interesting aspect of this query is that initially, the execution, after running for over 16 minutes, produced a ‘TimeOut Error’. We remedied this by setting the ‘allowDiskUse’ flag to true. The execution took 8 seconds.

db.fact.aggregate (

[{

$project: {

CustomerID: '$CustomerID',

TotalSales: '$TotalSales',

rec: '$$ROOT',

\_id: 0

}

}, {

$group: {

\_id: '$CustomerID',

TotalSales: {

$sum: '$TotalSales'

},

all\_sales: {

$push: {

sale: '$rec'

}

}

}

}, {

$sort: {

TotalSales: -1

}

}, {

$project: {

CustomerID: '$\_id',

TotalSales: 1,

rec: '$$ROOT'

}

}, {

$group: {

\_id: null,

Max: {

$max: '$TotalSales'

},

rec: {

$first: '$rec'

}

}

}, {

$project: {

Sale: '$Max',

CustomerID: '$rec.\_id',

\_id: 0

}

}], {

allowDiskUse: true

}

)

For the second part, we grouped together ‘CustomerID’ and ‘TotalSales’ along with ‘Country’. Per the learnings of the first part, we left the ‘allowDiskUse’ flag at true. The execution took 7 seconds.

db.fact.aggregate (

[{

$lookup: {

from: 'customer\_dim',

localField: 'CustomerID',

foreignField: 'CustomerID',

as: 'customer\_rec'

}

}, {

$unwind: {

path: '$customer\_rec'

}

}, {

$group: {

\_id: {

CustomerID: '$CustomerID',

Country: '$customer\_rec.Country'

},

TotalSalePerCustomer: {

$sum: '$TotalSales'

},

all\_sales: {

$push: {

sale\_rec: '$$ROOT'

}

}

}

}, {

$sort: {

TotalSalePerCustomer: -1

}

}, {

$project: {

CustomerID: '$\_id.CustomerID',

Country: '$\_id.Country',

TotalSalePerCustomer: 1,

\_id: 0

}

}, {

$group: {

\_id: {

CustomerID: '$CustomerID',

Country: '$Country'

},

TotalSalePerCustomer: {

$max: '$TotalSalePerCustomer'

}

}

}, {

$sort: {

TotalSalePerCustomer: -1

}

}, {

$project: {

CustomerID: '$\_id.CustomerID',

Country: '$\_id.Country',

TotalSalePerCustomer: 1,

\_id: 0,

}

}], {

allowDiskUse: true

}

)

* **Best-selling month (per country) [2009-2011]**

This query utilizes the ‘$unwind’ operator that allows us to examine each record in turn. Using that, we were able to extract the ‘Country’ values and group by ‘Country’, ‘Month’ and sum of ‘TotalSales’. This data was then grouped separately to identify ‘MaxSalePerHourPerCountry’. Execution, due to sheer volume of calculation, was a bit longer at 50 seconds.

db.fact.aggregate (

[{

$lookup: {

from: 'customer\_dim',

localField: 'CustomerID',

foreignField: 'CustomerID',

as: 'customer\_rec'

}

}, {

$unwind: {

path: '$customer\_rec'

}

}, {

$project: {

Month: 1,

Country: '$customer\_rec.Country',

TotalSales: 1,

\_id: 0

}

}, {

$group: {

\_id: {

Month: '$Month',

Country: '$Country'

},

TotalSalePerCountry: {

$sum: '$TotalSales'

}

}

}, {

$project: {

Month: '$\_id.Month',

Country: '$\_id.Country',

TotalSalePerCountry: 1,

rec: '$$ROOT',

\_id: 0

}

}, {

$group: {

\_id: '$Country',

MaxSalePerHourPerCountry: {

$max: '$TotalSalePerCountry'

},

rec: {

$first: '$rec'

}

}

}, {

$project: {

Country: '$\_id',

MaxSalePerHourPerCountry: 1,

Month: '$rec.\_id.Month'

}

}, {

$sort: {

Country: 1

}

}], {

allowDiskUse: true

}

)

* **Best-selling product per month [2009-2011]**

This query required a grouping of ‘Year’, ‘Stock’ and sum of ‘TotalSales’ to produce ‘TotalAnnualSales’. The results were then sorted by ‘Year’. Execution time was once again, minimal at 3 seconds.

db.fact.aggregate(

[{

$project: {

Month: "$Month",

TotalSales: "$TotalSales",

StockCode: "$StockCode",

document: "$$ROOT"

}

}, {

$sort: {

TotalSales: -1

}

}, {

$group: {

\_id: "$Month",

max: {

$max: "$TotalSales"

},

product: {

$first: "$document"

}

}

}, {

$project: {

Month: "$\_id",

TotalSales: "$max",

StockCode: "$product.StockCode",

\_id: 0

}

}, {

$sort: {

Month: 1

}

}], {

allowDiskUse: true

}

)

* **Query 9**

This query required

* **Average spending of customers in a country**

This query required both ‘CustomerID’ and ‘Country’ from the ‘Customer\_dim’ collection. The data was grouped by ‘Country’ and the average of ‘TotalSales’ was calculated. The results were then sorted and displayed. This query took slightly longer to execute at 51 seconds.

db.fact.aggregate(

[{

$lookup: {

from: 'customer\_dim',

localField: 'CustomerID',

foreignField: 'CustomerID',

as: 'customer\_rec'

}

}, {

$unwind: {

path: '$customer\_rec'

}

}, {

$project: {

rec: '$$ROOT'

}

}, {

$group: {

\_id: '$rec.customer\_rec.Country',

AvgSpending: {

$avg: '$rec.TotalSales'

},

rec: {

$first: '$rec'

}

}

}, {

$project: {

Country: '$\_id',

AvgSpending: 1,

\_id: 0

}

}, {

$sort: {

AvgSpending: -1

}

}], {

allowDiskUse: true

}

)

* **Frequently purchased item per customer**

This query required a grouping of ‘CustomerID’ and ‘StockCode’ along with a ‘$sum’ variable in place to count the number of occurrences of an item. The results were then sorted by ‘count’. Execution time was once again, minimal at 3 seconds.

db.fact.aggregate(

[{

$group: {

\_id: {

CustomerID: '$CustomerID',

StockCode: '$StockCode'

},

count: {

$sum: 1

}

}

}, {

$project: {

CustomerID: '$\_id.CustomerID',

StockCode: '$\_id.StockCode',

count: 1

}

}, {

$sort: {

count: -1,

CustomerID: 1

}

}]

)

* + 1. **Advantages and Disadvantages of MongoDB**
  1. **Neo4j**
     1. **ODB Creation**
        1. **Creating CSVs**
        + We must change the column "Customer ID" to "CustomerID" in both Excel Files
        + We must save the files in a CSV format
        1. **Importing the Data**
        + Must store both the CSV files in the import folder of the Database
        + We will have to use the following queries to import the data and store it into nodes with the label ODB
          - :auto USING PERIODIC COMMIT 1000 LOAD CSV WITH HEADERS FROM '[file:///Cleaned\_2009\_2010.csv](file:///\\Cleaned_2009_2010.csv)' AS test CREATE (:ODB {Invoice: test.Invoice, StockCode: test.StockCode, Description: test.Description, Quantity: toInteger(test.Quantity), InvoiceDate: test.InvoiceDate, InvoiceTime: test.InvoiceTime, Price: toFloat(test.Price), CustomerID: test.CustomerID, Country:test.Country, SubTotal: toFloat(test.SubTotal)})
          - :auto USING PERIODIC COMMIT 1000 LOAD CSV WITH HEADERS FROM '[file:///Cleaned\_2010\_2011.csv](file:///\\Cleaned_2010_2011.csv)' AS test CREATE (:ODB {Invoice: test.Invoice, StockCode: test.StockCode, Description: test.Description, Quantity: toInteger(test.Quantity), InvoiceDate: test.InvoiceDate, InvoiceTime: test.InvoiceTime, Price: toFloat(test.Price), CustomerID: test.CustomerID, Country:test.Country, SubTotal: toFloat(test.SubTotal)})
        + Each entry in the CSV is converted into a node with the label ODB with each column as a property
        1. **Cleaning the ODB**
        + We will have to convert the values of InvoiceTime and InvoiceDate into a format which is understood by Neo4j
        + We use a combination of string manipulation and the Date() function for InvoiceDate using the following query
          - MATCH(m:ODB)

WITH [item in split(m.InvoiceDate, "/") | toInteger(item)] AS dateComponents, m AS m

SET m.InvoiceDate = date({day: dateComponents[1], month: dateComponents[0], year: dateComponents[2]})

* + - * We must use the Time() function for InvoiceTime using the following query
        + CALL apoc.periodic.iterate(

"MATCH(n:ODB) RETURN n",

"SET n.InvoiceTime = time(n.InvoiceTime)",

{batchSize:100000})

* + 1. **ADB Creation**
       1. **Creating Tables**
       - We must create the tables from the ODB using apoc.periodic.iterate() function
       - This function allows us to select the information we need from the ODB in the inner query and create new nodes using the outer query in batches so that the system isn't overloaded.
         1. **FACT**

CALL apoc.periodic.iterate(

"MATCH(n:ODB) RETURN n",

"CREATE(:FACT{CustomerID: n.CustomerID, Year: n.InvoiceDate.year, Month: n.InvoiceDate.month, Day: n.InvoiceDate.day, Hour: n.InvoiceTime.hour, Minute: n.InvoiceTime.minute, StockCode: n.StockCode, Quantity: n.Quantity, TotalSales: n.SubTotal, TimeID:datetime({Year: n.InvoiceDate.year, Month: n.InvoiceDate.month, Day: n.InvoiceDate.day, Hour: n.InvoiceTime.hour, Minute: n.InvoiceTime.minute})})",

{batchSize:10000})

* + - * 1. **CUSTOMERDIM**

CALL apoc.periodic.iterate(

"MATCH(n:ODB)

RETURN n.CustomerID as CustomerID, n.Country as Country

UNION

MATCH(n:ODB)

RETURN n.CustomerID as CustomerID, n.Country as Country",

"CREATE(:CUSTOMERDIM{ CustomerID: CustomerID , Country: Country})",

{batchSize:10000})

* + - * 1. **TIMEDIM**

CALL apoc.periodic.iterate(

"MATCH(n:ODB)

RETURN n.InvoiceDate.year as Year, n.InvoiceDate.month as Month, n.InvoiceDate.day as Day,n.InvoiceTime.hour as Hour, n.InvoiceTime.minute as Minute

UNION

MATCH(n:ODB)

RETURN n.InvoiceDate.year as Year, n.InvoiceDate.month as Month, n.InvoiceDate.day as Day,n.InvoiceTime.hour as Hour, n.InvoiceTime.minute as Minute",

"CREATE(:TIMEDIM{ Year: Year , Month: Month, Day: Day, Hour: Hour, Minute: Minute, TimeID:datetime({Year: Year , Month: Month, Day: Day, Hour: Hour, Minute: Minute})})",

{batchSize:10000})

* + - * 1. **STOCKDIM**

CALL apoc.periodic.iterate(

"MATCH (n:ODB) RETURN n.StockCode as StockCode, max(n.Description) as Description, max(n.Price) as Price",

"CREATE(:STOCKDIM{StockCode: StockCode, Description: Description, Price: Price})",

{batchSize:10000})

* + - 1. **Creating the Relationships between the tables**
         1. **Creating Relationship between FACT and CUSTOMERDIM**

CALL apoc.periodic.iterate(

"MATCH(n:FACT),(m:CUSTOMERDIM)

WHERE n.CustomerID=m.CustomerID

RETURN n,m",

"CREATE (n)-[:IS\_CUSTOMER]->(m)",

{batchSize:100000})

* + - * 1. **Creating Relationship between FACT and STOCKDIM**

CALL apoc.periodic.iterate(

"MATCH(n:FACT),(m:STOCKDIM)

WHERE n.StockCode=m.StockCode

RETURN n,m",

"CREATE (n)-[:IS\_STOCK]->(m)",

{batchSize:100000})

* + - * 1. **Creating Relationship between FACT and TIMEDIM**

CALL apoc.periodic.iterate(

"MATCH(n:FACT),(m:TIMEDIM)

WHERE n.TimeID=m.TimeID

RETURN n,m",

"CREATE (n)-[:IS\_TIME]->(m)",

{batchSize:100000})

* + 1. **Aggregation Queries**
       1. **What time of the day (which hour of the day) is the sale maximum per country?**

CALL{

MATCH(c:CUSTOMERDIM)<-[r1:IS\_CUSTOMER]-(f:FACT)-[r2:IS\_STOCK]->(s:STOCKDIM)

WITH [SUM(f.TotalSales), time({Hour:f.Hour, Minute:f.Minute}), c.Country] AS Top\_Prod

RETURN MAX(Top\_Prod[0]) AS TotalSales, Top\_Prod[1] AS Time, Top\_Prod[2] AS Country

ORDER BY TotalSales DESC, Country

}

RETURN Country, collect(Time)[0] AS Time, ROUND(MAX(TotalSales)) AS TotalSales

ORDER BY Country

* + - 1. **What is the annual TotalSales per product?**

MATCH(s:STOCKDIM)<-[r1:IS\_STOCK]-(f:FACT)

RETURN s.Description AS Description, SUM(ROUND(f.TotalSales)) AS Annual\_Sales, f.Year AS Year

ORDER BY Description, Year

* + - 1. **What is the top product per year?**

CALL{

MATCH(s:STOCKDIM)<-[r1:IS\_STOCK]-(f:FACT)

WITH [s.Description, SUM(ROUND(f.TotalSales)), f.Year] AS INFO

RETURN INFO[0] AS Description, INFO[1] AS Annual\_Sales, INFO[2] AS Year

ORDER BY Annual\_Sales DESC, Year

}

RETURN MAX(Annual\_Sales), collect(Description)[0], Year

ORDER BY Year

* + - 1. **What is the top product per country?**

CALL{

MATCH(c:CUSTOMERDIM)<-[r1:IS\_CUSTOMER]-(f:FACT)-[r2:IS\_STOCK]->(s:STOCKDIM)

WITH [SUM(f.Quantity), s.Description, c.Country] AS Top\_Prod

RETURN MAX(Top\_Prod[0]) AS Quantity, Top\_Prod[1] AS Description, Top\_Prod[2] AS Country

ORDER BY Quantity DESC, Country ASC

}

RETURN MAX(Quantity), collect(Description)[0], Country

ORDER BY Country

* + - 1. **Which item is sold below a certain threshold value? Or, what are the under-performed products based on the average sales last year?**
      2. **Which customer spends the most (per country/overall)?**
         1. **Overall**

MATCH(f:FACT)

WITH [SUM(f.TotalSales),f.CustomerID] AS Sales\_per\_Customer

RETURN MAX(Sales\_per\_Customer[0]) AS Total\_Sale, Sales\_per\_Customer[1] AS CustomerID

ORDER BY CustomerID DESC

LIMIT 1

* + - * 1. **Per Country**

CALL{

MATCH(f:FACT)-[r1:IS\_CUSTOMER]->(c:CUSTOMERDIM)

WITH [SUM(f.TotalSales), f.CustomerID, c.Country] AS Sales\_per\_Customer, c

RETURN MAX(Sales\_per\_Customer[0]) AS Total\_Sale, Sales\_per\_Customer[1] AS CustomerID, Sales\_per\_Customer[2] AS Country, c, Sales\_per\_Customer

ORDER BY Total\_Sale DESC, Country

}

RETURN MAX(Total\_Sale), collect(CustomerID)[0], Country

ORDER BY Country

* + - 1. **What is the best-selling month per country? (Given the year range, 2009-2011)**

CALL{

MATCH(c:CUSTOMERDIM)<-[r1:IS\_CUSTOMER]-(f:FACT)-[r2:IS\_STOCK]->(s:STOCKDIM)

WITH [SUM(f.Quantity), f.Month, c.Country] AS Top\_Prod

RETURN MAX(Top\_Prod[0]) AS Quantity, Top\_Prod[1] AS Month, Top\_Prod[2] AS Country

ORDER BY Quantity DESC, Country

}

RETURN Country, ["January", "February", "March", "April", "May", "June", "July", "August", "September", "October", "November", "December"][collect(Month)[0]-1] AS Month, MAX(Quantity) AS Quantity

ORDER BY Country

* + - 1. **What is the best-selling product per month? (Given the year range, 2009-2011)**

CALL{

MATCH(c:CUSTOMERDIM)<-[r1:IS\_CUSTOMER]-(f:FACT)-[r2:IS\_STOCK]->(s:STOCKDIM)

WITH [SUM(f.Quantity), s.Description, f.Month] AS INFO

RETURN MAX(INFO[0]) AS Quantity, INFO[1] AS Description, INFO[2] AS Months

ORDER BY Quantity DESC, Months

}

RETURN Months AS SrNo, ["January", "February", "March", "April", "May", "June", "July", "August", "September", "October", "November", "December"][Months-1] AS Month, collect(Description)[0] AS Description, MAX(Quantity) AS Quantity

ORDER BY SrNo

* + - 1. **What is the change in TotalSales per country per year (Trend of Sales)?**
      2. **What is the average spending of a customer per country? (TotalSales/Number of customers)**

MATCH(f:FACT)-[r1:IS\_CUSTOMER]->(c:CUSTOMERDIM)

WITH [SUM(f.TotalSales),f.CustomerID] AS Sales\_per\_Customer, c

RETURN AVG(Sales\_per\_Customer[0]) AS Average\_Spending\_Per\_Customer, c.Country AS Country

* + - 1. **What is the frequently purchased item per customer?**

CALL{

MATCH(f:FACT)-[r1:IS\_STOCK]-(s:STOCKDIM)

WITH [COUNT(f.StockCode), s.Description, f.CustomerID] AS INFO

RETURN MAX(INFO[0]) AS Frequency, INFO[1] AS Description, INFO[2] AS CustomerID

ORDER BY Frequency DESC, CustomerID

}

RETURN MAX(Frequency) AS Frequency, collect(Description)[0] AS Description, CustomerID

ORDER BY CustomerID

* + 1. **Justifications**
       1. **Deviation from the Star Schema by adding TimeID**
* We have added TimeID to the FACT table and the TIMEDIM table.
* This has been done as Neo4j does not handle the creation of a relation using multiple properties very will leading to a very inefficient query.
* Without the TimeID the query would run for 6+ hours if the dbms.memory.heap.max\_size configuration is changed from 1GB to 10GB in the config file.
* This was unacceptable and for the query to run in a timely manner we added TimeID
* The TimeID property is not used in anything except the creation of the relation between the FACT table and the TIMEDIM table.
  + - 1. **Unable to execute queries “Which item is sold below a certain threshold value? Or, what are the under-performed products based on the average sales last year?” and “What is the change in TotalSales per country per year (Trend of Sales)?”**
* Both the above queries require some conditional filtering after aggregation.
* In order to have multiple aggregations we need to use the CALL function in Neo4j.
* CALL allows us to perform more aggregations on the RETURN values of a query. However, this can only be done using a RETURN statement.
* Due to this we cannot use any conditional filters or conditions.
* This is one of the drawbacks of Neo4j we have found.
  + 1. **Advantages of Neo4j**
* By creating relationships, we are avoiding the need to define joins in the query which saves time
  + 1. **Disadvantages of Neo4j**
* There are no nested aggregations in a straightforward manner they must be done using the CALL function
* Cloning nodes is a very slow process and must be done using the APOC plugin when there are numerous entries
* Creating Relationships between nodes using multiple properties is a very slow and memory intensive process
* The CALL function does not allow any additional conditional filters
* There is no flexible way to create nested queries
* There is no Group By but there are aggregations which are not as flexible as they don't allow aggregation based on a category
* There is no way to create a View in Neo4j
  1. **Front-End**

1. **Comparison**
2. **Conclusion**