Kivos Challenging Activity

System Engineer

An overview of the steps, highlighting decisions and challenges of the solution implementation

1) Understanding the data: Check the content of all the given CSV files and gather information about the data they contain, and how each file is connected to each other.

Combining common sense and basic information that was given about the files, the following conclusions are made:

- plan_digest_<pid>.csv file contains information about a specific plan. The columns:
  - Column 1: Plan ID (pid, e.g., 41c9377a45d4a37373cca0215766cca0), which is unique for every different plan
  - Column 2: Timestamp of when the plan was created (e.g., 20210507105638).
  - Column 3: Start date of the planning period (e.g., 20240311).
  - Column 4: End date of the planning period (e.g., 20240602).
  - Column 5: A descriptive name for the weeks the plan contains (e.g., Plan 202411-202422).
- plan_clusters_<pid>.csv file contains the mapping of stores to clusters for each existing planning in the data. All the unique clusters seem to be ‘Small’, ‘Medium’ and ‘Large’ stores. Every file contains only data for one planning (unique pid). The columns:
  - Column 1: Plan ID which is unique for every different plan.
  - Column 2: Store ID (e.g., 260, etc.), representing individual stores.
  - Column 3: Cluster ID (e.g., 26000020), which is unique for every different cluster.
  - Column 4: Cluster Label/Name (e.g., Medium stores), providing a descriptive name for the cluster.
- plan_sales_<pid>.csv file contains planned sales per week/item/store for each existing planning in the data. Every file contains only data for one planning (unique pid). The columns:
  - Column 1: Plan ID which is unique for every different plan.
  - Column 2: Date (e.g., 20240317), representing the specific week for the planned sales.

    - Column 3: Item ID (e.g., 1010150), identifying the specific item.
    - Column 4: Store ID, representing the store where the sales are planned.
    - Column 5: Planned Sales (e.g., 0.20408163265306), indicating the number of units expected to be sold.
- forecast.csv file has similar format to plan_sales (except the pid column) and sales. It contains the forecast of sales per item/store for a specific date. Those dates seem to keep a week frequency (they are 7 days apart from each other). The columns:
    - Column 1: Date, representing the specific week for which the sales forecast is made.
    - Column 2: Item ID, identifying the specific item.
    - Column 3: Store ID, representing the store for which the forecast is made.
    - Column 4: Forecasted Units (e.g., 2), indicating the number of units expected to be sold in the forecast.
- sales.csv file has similar format to plan_sales (except the pid column) and forecast. It contains the sales made last year per item/store for a specific date. Those dates seem to keep a week frequency (they are 7 days apart from each other). The columns:
    - Column 1: Date, representing the specific week when the sales occurred.
    - Column 2: Item ID, identifying the specific item that was sold.
    - Column 3: Store ID, indicating the store where the sales occurred.
    - Column 4: Actual Sales (e.g., 13), showing the number of units sold in that week.

Some important notes about the data are:

There are 2 plannings with different pids that correspond to the same planning period, and we know that in such cases we keep the one last generated (based on its timestamp).

Also, different plannings have different mappings of stores-clusters. One store that may appear as Large in a planning, can appear as Medium in another. So, we need to keep those relations.

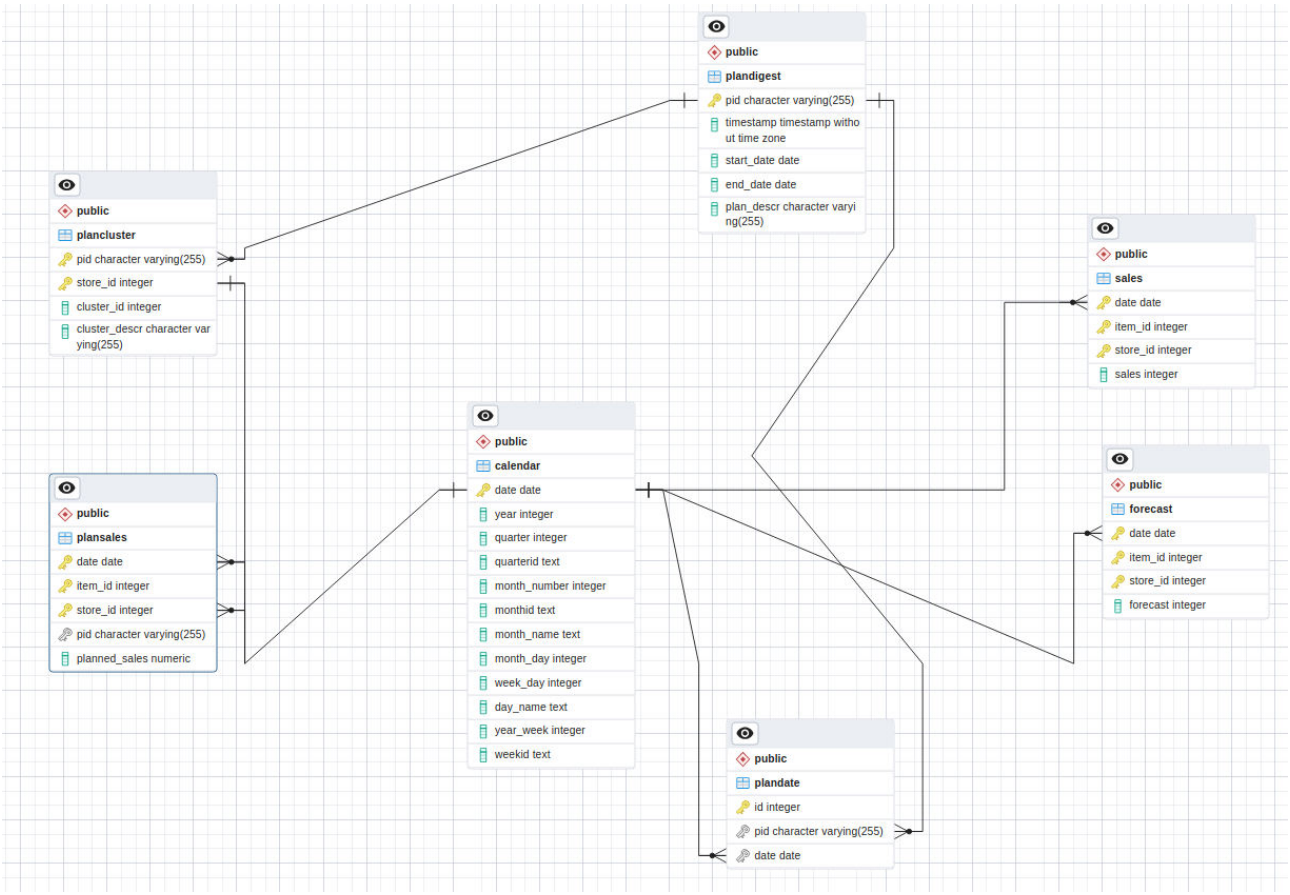
A notice about the types of the planned sales units, forecast units and actual sales is that the first one has a type of float number while the other two have integer. It is a little strange that they are float numbers. I assume that the set up is the following: you have a forecasting system, that gives the forecasted number of units to be sold for each date/store/item, meaning the expected demand. Then, this is used by some planning system which takes into account different constraints regarding the supply chain, and in the end gives us the planned sales for each date/store/item.

2) Designing an implementation: Based on the needs of the challenge and the data given, a solution needs to be designed to attack the problem.

So, we need to create a database consisting of different tables, with relations between them that represent the data well. For this, a first look in the questions that need to be answered is important, in order to have an insight of what relations should exist in the database so to be easy to connect

each table with one another. This process is needed to decide a database schema. Key decisions here are:

- DB selection: We need to decide what database to use in our solution, SQL or NOSQL? The answer here is obvious; our data are structured and we will need primary keys to create relations between our tables to perform our queries, so we'll go with SQL. I chose a postgres db to continue, because I have worked with it before.
- DB schema: Designing the db is one of the most crucial parts of the implementation. One error or mistake on the relations between the tables or the selection of tables to use, and our answers will not be reliable or even we'll not be able to perform some queries at all. The database schema I finally used can be seen in the ERD (generated with pgadmin) below:



A small description for each table:

- **calendar**: The date dimension of our database, it contains all the dates for the last years and their corresponding quarters, months, weeks, days etc. Primary key is the date.
- **Plandigest**: This table basically contains all the info we can find in accepted plan_digest.csv, but merged for all the plans. Also, no two plans in the table are over the same period, as the data has been already cleaned. It contains the pid (primary key) and the timestamp, start and end date, and plan description.
- **Plandate**: This table contains the pid of every plan and all the dates (date column is a foreign key from calendar table) between the start and end date of the plan (mentioned in the plandigest table – pid is foreign key here). The primary key here is just an incremental key.

- **Plancluster**: This table basically contains the info we can find in plan_cluster.csv, but merged for all the plans. We have composite keys here, the primary key is a combination of two columns, the pid (foreign key from plandigest as well) and the store_id. This is because that combination is unique in the merged plan_cluster data. Also, the table contains the cluster id and description.
- **Plansales**: This table basically contains the info we can find in plan_sales.csv, but merged for all plans. The primary key is a combination of three columns, the date (foreign key from calendar), the item_id and the store_id, because this is the unique combination in our data for this table. The table also has the pid as foreign key from plan_cluster and the planned_sales.
- **Forecast:** This table basically contains the info we can find in forecast.csv. Similary to plansales table, it has three columns as primary key, the date (foreign key from calendar), the item_id and the store_id. Also it contains a forecast column.
- **Sales**: This table basically contains the info we can find in sales.csv. Similary to plansales table, it has three columns as primary key, the date (foreign key from calendar), the item_id and the store_id. Also it contains a sales column.

Moving on to the database creation and the data loading process, SQL is the most optimal way to use for that task. So, first we create our database. We use a docker container to set up the database, so we can easily recreate it in any environment, and an SQL script that creates the tables.

But in order to make the whole process automated, a python app that consists of different scripts is used. The app also runs on a docker container, that communicates with the db container. The app contains a sort of a ‘pipeline’, that reads the csv files from the data directory, loads them into the database if they do not already exist, perform SQL queries that answer the questions of the challenge and present them in a mini dashboard. Key decisions here are:

- Using python: Being familiar with it, I chose python as basic language to use to create the whole app and automate the process. Also, there are some good libraries to use for running SQL scripts.
- SQLAlchemy: It is a robust and flexible library that simplifies database interaction in Python projects.
- Docker: Dockerizing the database and Python application offers improved consistency, portability, scalability, and resource efficiency while simplifying deployment and management.
- Streamlit: It is a quick and easy way to share data analysis findings with others. Its simplicity, interactivity, and integration capabilities make it a good choice for a mini dashboard where some results are presented.

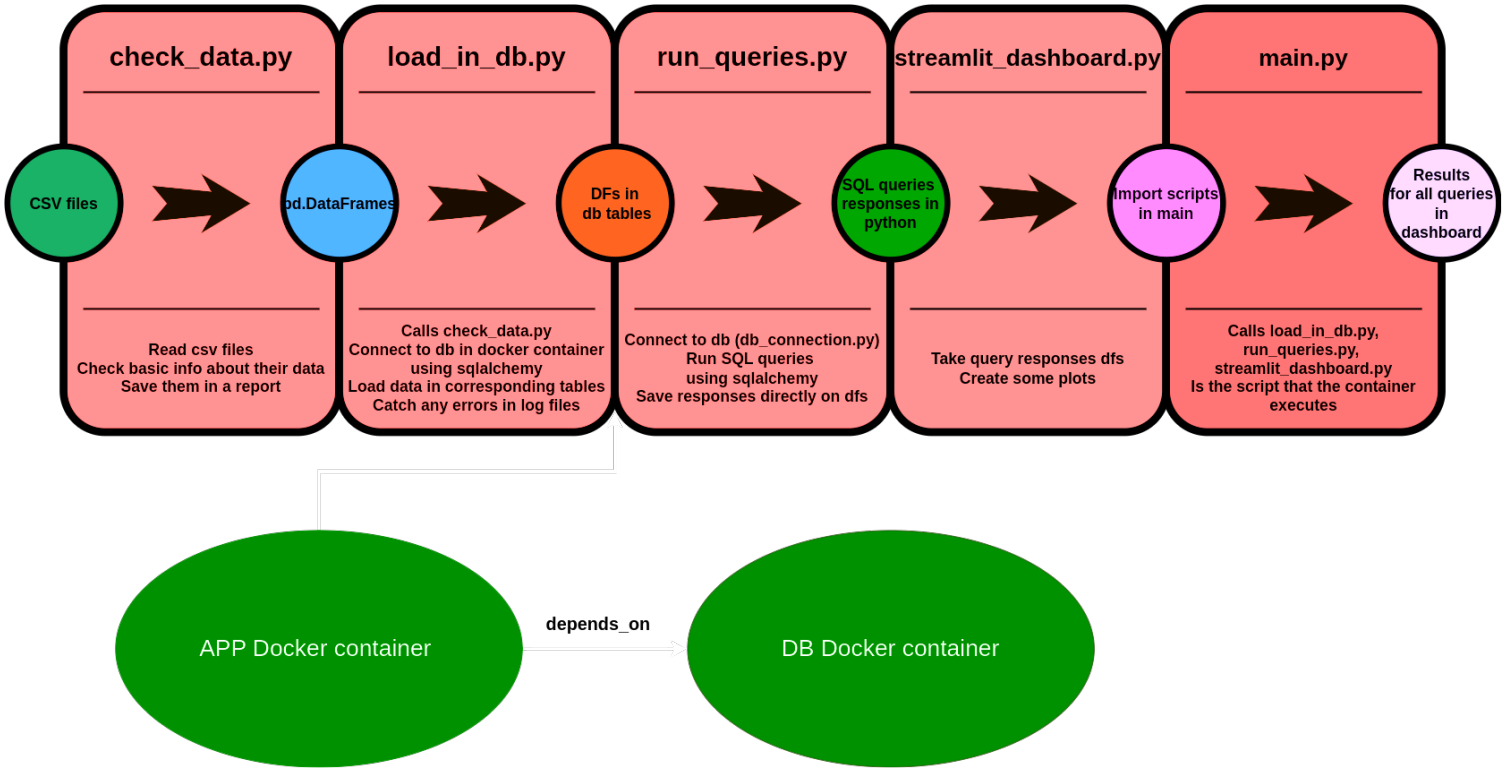
An important thing to mention in the above implementation is that there is a constraint in reading the initial csv data. The files need to fit in RAM, in order to be read using python. The most efficient way to handle this in a real scenario would be to first load the csv files in a staging database, and then use ETL methods to read them and load them into our database. But due to limited time this is not the approach I followed.

Also, some indexing could be used in the database. For example, we could index the plansales table that has the planned sales for every different plan on pid, if we had many real data for different plans.

3) **Implementation details:** To dive deeper in the solution we provided, let's see how exactly it works and what steps need to be taken on a high level (setup and execution instructions will be given in a README.md file) and check how the app works.

To begin with, we start the db container. We create the database by connecting to the db server in container (we can use pgadmin also, to have a better visual to our db). After creating the db, we can start the app container and the whole process runs.

In detail, the pipeline code can be seen here:



- The **check_data** script reads the csv files and saves basic info about the data in report files. Also, it checks for duplicates in plandigest files, and keeps only the last one generated for a unique planning (based on the timestamp), also keeping only the pids that correspond to that unique planning for other files as well (cluster, plan_sales).
- The **load_in_db** script uses check_data to load csv files, map all the data to the corresponding tables they'll be inserted into, and it tries to insert them, if they do not already exist, table by table. If there is an error in insertion, the error is written in the log directory, in a log file using the timestamp that the error happened on.
- There is the **db_connection** script that connects the app to the db, using the sqlalchemy library. It gets the credentials from a .env file.
- There is a **queries** script where the queries we use are defined in a dictionary.

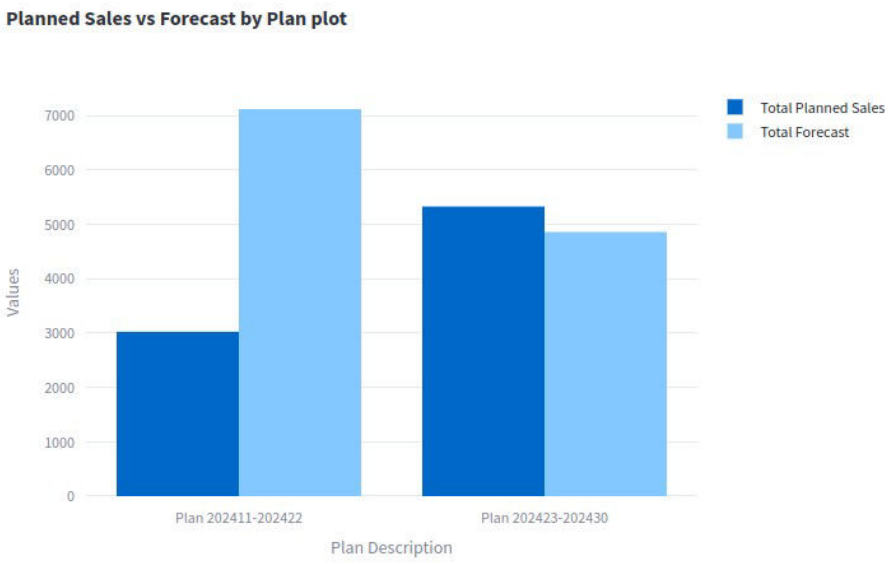
- The **run_queries** script takes the queries dict and tries to execute them, and log the possible errors.
- The **streamlit_dashboard** script defines the plots that will be shown in the dashboard.
- The main script is the one where everything is executed. It gets the data-db tables mappings, it calls the insertion function from load_in_db script (if data do not exist in db), then it takes the results of the queries from run_queries script and passes them in the streamlit_main from streamlit_dashboard script. The main is executed by docker, to run the dashboard on localhost:8501.
- The streamlit dashboard contains all the plots and output tables for each query executed for every question. The plots are interactive (we can see the exact value of each bar by hovering), and for the questions contain comparisons for plannings and clusters, there is a selection bar to choose which planning you want.

4) **Approaching and answering the questions**

- Compare total planned sales against total sales forecast in each planning period
  - The approach I followed on this one was to find the sum of planned_sales per pid in the plansales table and sum of forecast in the forecast table. For the forecast table, we need to join (on pid, store_id, and item_id – we want to compare per planning period, store_id, and item_id) the forecast table with plansales to get the same dates of both, and compare them. We use join operation, because it makes sense to compare only same dates-stores-items combinations and not different ones of a table (eg left join). Finally, we get the total_planned_sales and total_forecast per pid.
  - Comparing planned sales to sales forecasts is a crucial process that helps organizations monitor their performance, make informed decisions, allocate resources effectively and adapt to dynamic market conditions.
  - Running the implemented query, we get the output shown in the table below:

	pid character varying (255)	plan_descr character varying (255)	total_planned_sales numeric	total_forecast bigint
1	41c9377a45d4a37373cca0215766cca0	Plan 202411-202422	3024.64	7121
2	a7760a00cc0276d8ad1aa3a5ac55ad60	Plan 202423-202430	5328.90	4860

  - We see the total_planned_sales and total_forecast per unique pid and its plan_descr. The total forecast for the first one is 7121 units while the total planned_sales are 3024.64, which is far less. This suggests that the sales forecast for this period is much more optimistic compared to the sales plan. Although, for the second plan, the total forecast is 4860 units while total planned_sales a little higher, at 5328.90, so in this case the sales forecast for this period fell short of the sales plan.
  - A plot to compare the values better:



- Perform the same comparison at cluster level (each cluster will have a subset of the total)
  - What we want here is the same comparison as above, not only per pid, but by cluster as well. (This,at least, is what I understand from how the question is stated in the assignement.) So, here we need the plancluster table as well, in order to get the clusters. First, we join the plansales with planclusters, to map each pid, store_id combination to a cluster, and we find total_planned_sales per cluster for each pid, for plansales table only. This is main subquery1.

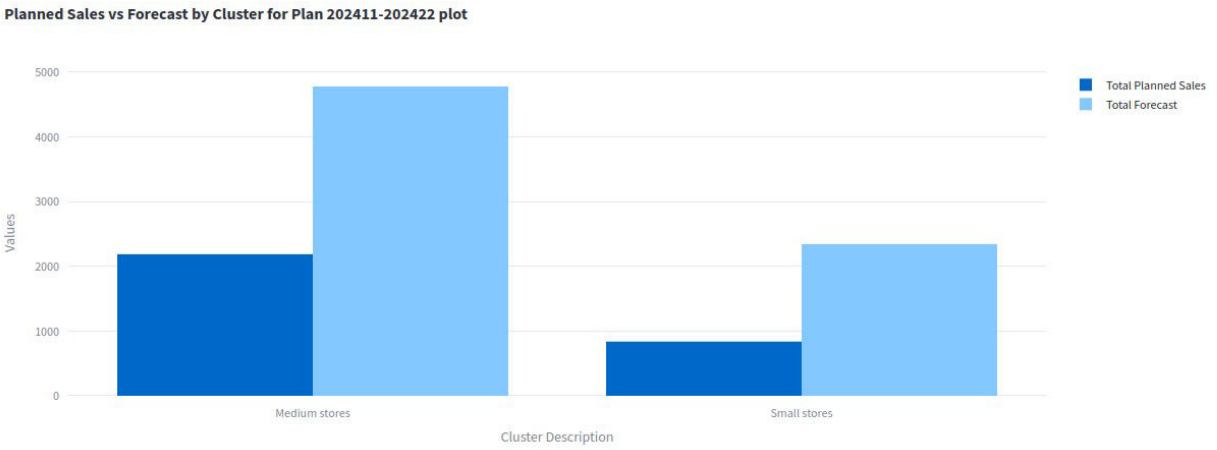
    For forecast table, first we need to join it with plansales (same way we did for the previous query), and then join the output of that join with the plancluster table, to get cluster info about forecast dates. This is main subquery2.

    Finally, we join the tables from the two main subqueries, to get the total_planned_sales and total_forecast per pid and cluster_descr.

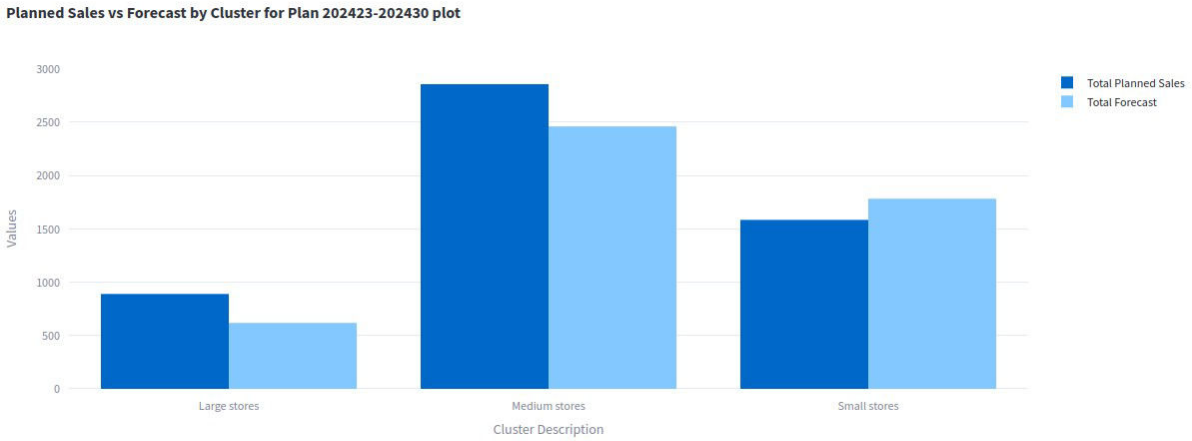
  - To shed more light in the above comparison between total planned_sales and total_forecast, we can use clusters, to see what was planned and forecasted per cluster of stores for each plan period. Running the implemented query, we get the output shown in the table below:

	pid character varying (255)	plan_descr character varying (255)	cluster_descr character varying (255)	total_planned_sales numeric	total_forecast bigint
1	41c9377a45d4a37373cca0215766cca0	Plan 202411-202422	Medium stores	2187.15	4779
2	41c9377a45d4a37373cca0215766cca0	Plan 202411-202422	Small stores	837.49	2342
3	a7760a00cc0276d8ad1aa3a5ac55ad60	Plan 202423-202430	Large stores	889.47	618
4	a7760a00cc0276d8ad1aa3a5ac55ad60	Plan 202423-202430	Medium stores	2855.61	2460
5	a7760a00cc0276d8ad1aa3a5ac55ad60	Plan 202423-202430	Small stores	1583.83	1782

  - We basically see the same total values as the previous queries, but seperated into the different store clusters. So, is we add all the total values for planned_sales and for forecast per pid, we can get the total numbers shown in the previous table. Indeed, we see that there is a big difference in plan sales and forecast for both clusters that appear in the first plan. A plot for the first plan to compare the values better:



- For the second plan though, the total planned_sales and forecasts are much more closer to each other for each one of the three clusters appeared in it, so the forecasts were more realistic in this case. A plot for the second plan to compare the values better:



- At this point, we create a view that we use at both question 4 and 5. What this view does, is to return a “table” containing the joined plansales and sales on same week id, store id, item id. We achieve that by joining each one of the plansales and sales with calendar table on date, and get the week id.
- Compare planned sales by cluster against last year in each planning period
  - Here the question is to compare the planned sales with last year’s sales, by cluster and for each planning period. The comparison we need to make looks like the previous question, but now we want the previous year’s sales and not the forecast.

    So, we have noticed that planned_sales, forecasts and sales, have dates that differ 7 days with each other, and all those days are Sundays. Also, a planning period seem to start on Monday and ends on Sunday (based on start and end dates of plandigest). We can see that very easily from joining the corresponding tables with calendar.

    So, based on the fact that we describe plans on week’s granularity, we should map the plansales dates to sales (last year’s sales) on weekids. We can extract the week id of any date from calendar table. To do this, we join the plansales and sales table on same weeks

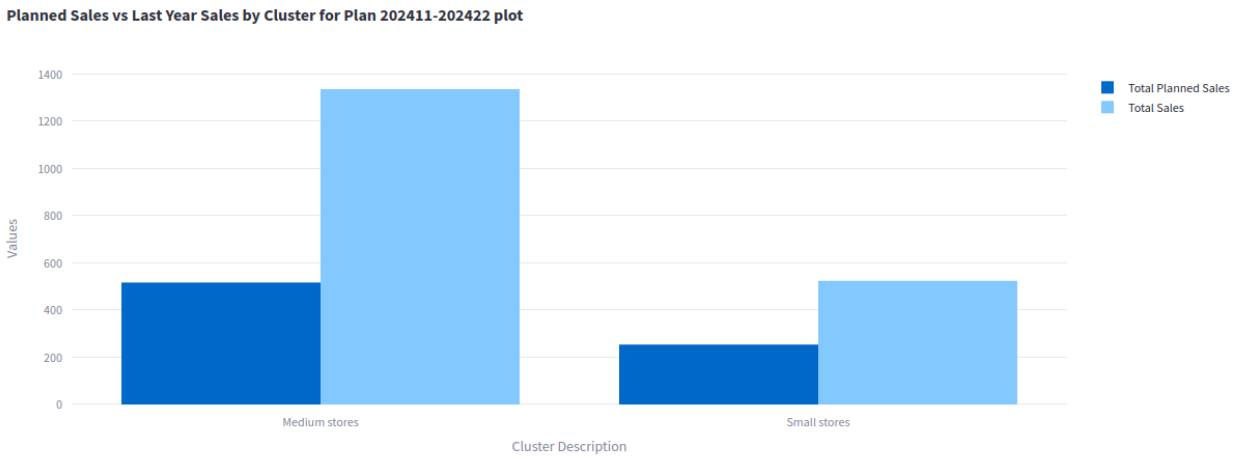
of year, when there is plan or sales information, using the view we created earlier. Then, we take the sum of planned_sales and sales, grouped by pid and cluster.

The sales.csv files has less rows than planned_sales and forecast, and it seems that some dates or dates, stores, items combinations are missing. So, it makes sense that after mapping the same combinations on plansales and sales, we see less total planned_sales than we did for the comparison of total planned_sales and total forecast.

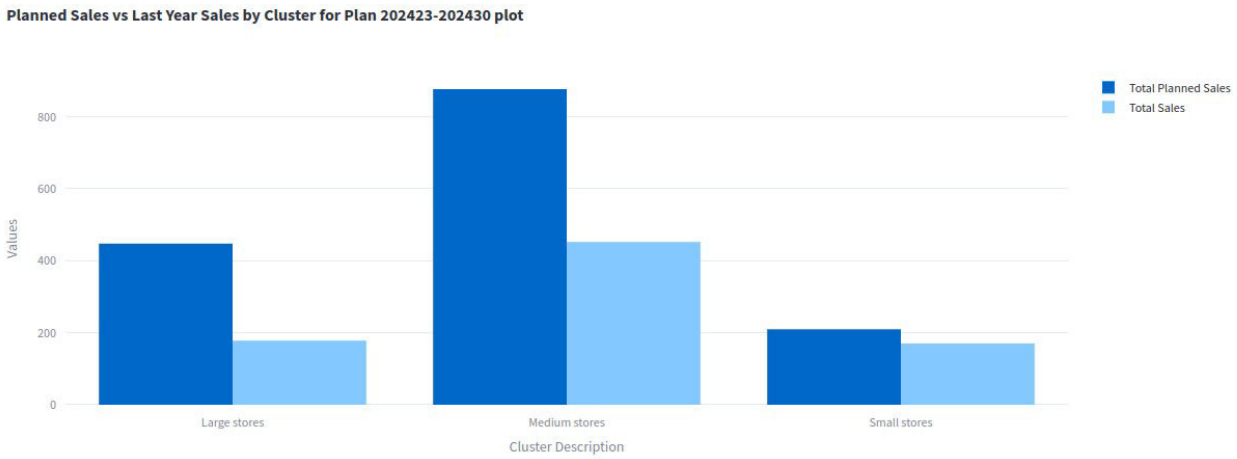
- Comparing planned sales against last year's sales can serve as a benchmark for performance improvement and can be helpful for trend analysis, examining year-on-year changes in sales performance that can reveal patterns.
- Running the implemented query, we get the output shown in the table below:

	pid character varying (255)	plan_descr character varying (255)	cluster_descr character varying (255)	total_planned_sales numeric	total_sales bigint
1	41c9377a45d4a37373cca0215766cca0	Plan 202411-202422	Medium stores	516.82	1336
2	41c9377a45d4a37373cca0215766cca0	Plan 202411-202422	Small stores	253.88	524
3	a7760a00cc0276d8ad1aa3a5ac55ad60	Plan 202423-202430	Large stores	447.39	178
4	a7760a00cc0276d8ad1aa3a5ac55ad60	Plan 202423-202430	Medium stores	876.34	452
5	a7760a00cc0276d8ad1aa3a5ac55ad60	Plan 202423-202430	Small stores	209.60	170

- As mentioned in the above, the total planned_sales in this query are less than the ones in the previous. Although, this makes sense because we had to map same weeks between current year and last year, and then calculate the total planned_sales for the remaining dates, to have a correct comparison between same things.
- So, for the first plan, for medium stores we have the biggest difference between the two total values, which indicates that planned_sales of this year for that cluster of stores are not expected to be as high as last year. This can happen due to a lot of different reasons that have to do with market conditions. For the small stores cluster, again the total_sales of last year are higher. A plot for the first plan to compare the values better:



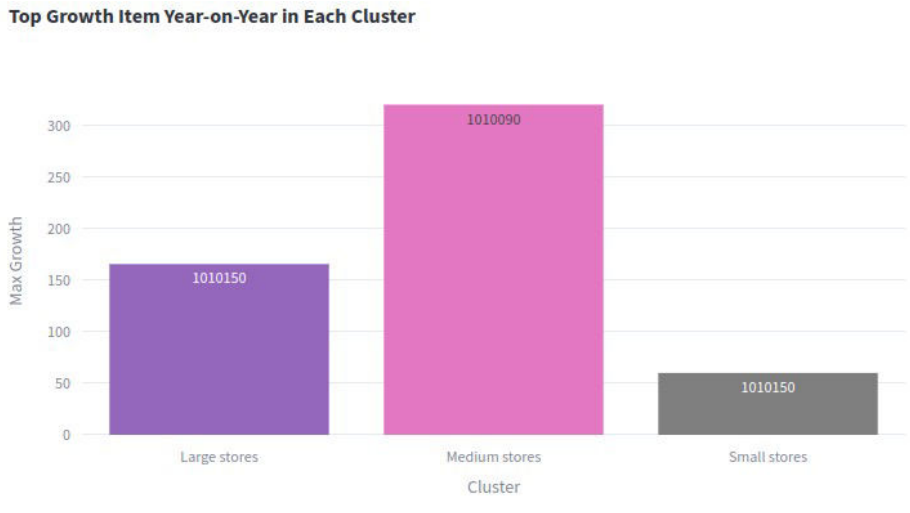
- Although, for the second plan we have compared between the two years, the total planned_sales for all the different clusters are higher than the actual sales of last year. A plot for the second plan to compare the values better:



- Highlight the item with the top growth (year-on-year) in each cluster
  - We create a view for the question 5 as well. The view extracts a table that has all the differences of items sales between plansales and last year sales per item id and per cluster. We need that view to keep the max difference (growth) and keep the item id as well.
  - Finally, in the next query we keep only the one max_growth for each cluster, as well as the item id that has that growth.
  - Running the implemented query, we get the output shown in the table below:

	cluster_descr character varying (255)	s_item_id integer	max_growth numeric
1	Large stores	1010150	165.74
2	Medium stores	1010090	320.42
3	Small stores	1010150	60.09

  - To calculate the top growth it makes sense that we use the last year's sales and planned sales of this year. So, we conclude to the item that has the max_growth over different clusters. The largest growth is noticed in the medium stores cluster, while the smallest in the small stores.
  - Growth is defined as total planned_sales - total sales
  - A plot to compare the values better:



* the plots are taken from the streamlit dashboard.

One thing I didn’t manage to complete in time was to run the sql script that creates the tables and relation of the database inside the container, using the docker-compose file, so the whole process would run together after the docker compose up command, and we wouldn’t need to do it “manually”.