POLITECNICO DI TORINO
Master degree course in Computer Engineering

Master Degree Thesis

The importance of data visualization tools in modern enterprises

Cost-effective solutions and empowering of an open source project

Supervisor
prof. Paolo Garza

Candidate
Giacomo Galliano
matricola: s292482

Internship Tutor
ContentWise SpA
ing. Riccardo Biondi

April 2023

Summary

In today's business world, data visualization tools have become a key component for many companies. This is the case also for *ContentWise*, a company that mainly works on recommendation engines and metadata enrichment. They have developed *Data Insights*, a plug-in service for the *UX Engine* that allows us to create and deliver custom dashboards to expose in a visually clear and effective way to the clients the incredibly high value generated by ContentWise's software.
The current implementation is based on Apache Superset 1.0, which provides most of the needed functionalities out of the box, but it also presents some important limitations.
The aim of this thesis project is to identify possible solutions to improve the current product. We started with a market analysis of the available tools in order to better understand the strengths and weaknesses of the main actors in the data visualization and exploration fields. After that, we came up with two possible solutions: the first is a completely custom web application built with the Next.js framework while the second leverages the 2.0 version of Apache Superset with all its improvements. In both cases, we integrated Cube as a semantic layer to increase flexibility and provide better performances.
The project ends with the evaluation of proposed solutions to identify the one that better fits ContentWise's needs.

Acknowledgements

I would like to thank Professor Paolo Garza for having guided me during this thesis project. Special thanks to ContentWise, Professor Paolo Cremonesi, and all the members of the R&D and CS teams. They all made me feel welcome since my first day in the company and allowed me to work on my thesis project in an active and stimulating environment. I'm very grateful for that. In particular, I would like to thank Riccardo Biondi for his willingness, guidance, and precious pieces of advice.
I would like to deeply thank my mother and my father for their constant support over all these years and for all the life lessons they taught me. A huge thanks to my brother, Lorenzo, who took on his shoulders most of the responsibilities and work at home to allow me to continue this path and for always being by my side. Many thanks to all the members of my big and united family for always encouraging me over these years.
Thanks to Dario, Francesco, Roberto, Paolo, and Stefano who shared with me all the joys and especially the struggles of the exam sessions and project deadlines. We had a great time together.
Thanks to my friends of a lifetime: Efrem, Tarik, Edoardo, Carlo, Francesco, and Cesare. For all the moments spent together and for those that will come. I am also grateful to all my hometown friends for the unforgettable memories together.

It has been a great and memorable journey, and now I'm eager to know what the future holds for me.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In the last few years, data visualization tools have become a key component for many companies. Regardless of the specific market field, the capability of transforming huge amounts of data and making them easily comprehensible represents a crucial element to provide benefits to organizations.

Today's market offers many solutions, each of which differs from the other in terms of offered functionalities, ease of use, and visualization capabilities. The majority of these tools are "paid" software, with license prices that can vary widely depending on the type of usage, but there are also alternatives in the open-source world that provide a wide set of standard and advanced functionalities for data visualization.

The aim of this thesis is to identify which could be the best solution in order to develop a data visualization tool that suits the company's needs, performing at the same time evaluations in terms of flexibility, time-to-market, and development and maintenance costs. We will compare a completely custom solution (a web application built with the *NextJS* framework) with a hybrid solution (leveraging the new release of *Apache Superset*) and in both cases, we will add *Cube* as a semantic layer between the data source and the visualization layer.

Chapter 2

Company and Project

ContentWise is a company that mainly works on recommendation engines and metadata enrichment, two factors that enforce each other thanks to Machine Learning algorithms. Over the last ten years, it has developed two products dedicated to IPTV providers, called *UX Engine* and *Metadata Foundry*, and one for catalogues enrichment in the fashion field, called *Catalogue Builder*.

Figure 2.1. ContentWise logo

This software generates incredibly high value, but nowadays the company still struggles to expose it in a clear and effective way to the customers. Often aggregates are made of hard-to-understand numbers or require analysis skills far from the users' needs.
To address this problem, *Data Insights* has been created. This is a plug-in service for the UX Engine that allows the company to create and deliver custom dashboards to visually show the customers the beneficial impact that ContentWise's software has on their business.

2.1 Data Insights

Data Insights is a solution that helps content and marketing people understand and improve the user experience of their digital services.
In today's market, the combination of content and user experience has a huge

impact on how a business performs. The main goal of those who work on catalogues is to provide users with the best library possible but to retain and increase the number of customers they cannot rely only on massive investments in original content and licensing. It is crucial for the user experience to be *personalized*.

A tailored UX comes with the need of defining and prioritizing metrics and KPIs that measure the impact of personalization but, other than that, it's necessary to provide a way to visualize them effectively.

2.1.1 Architecture overview

Data Insights is a cloud-native solution built using *Google Cloud Platform (GCP)* services. The application can be seen as the composition of four macro-layers:

1. *Ingestion Layer*, responsible for collecting data directly from the sources and loading it into Data Insights' data lake.

2. *Data Lake*, a centralized repository that allows you to store all your structured and unstructured data.

3. *Processing Layer*, responsible to process data to provide them in the right format to the visualization layer.

4. *Data Visualization Layer*, based on Apache Superset 1.0. Here is where customers' dashboards are created.

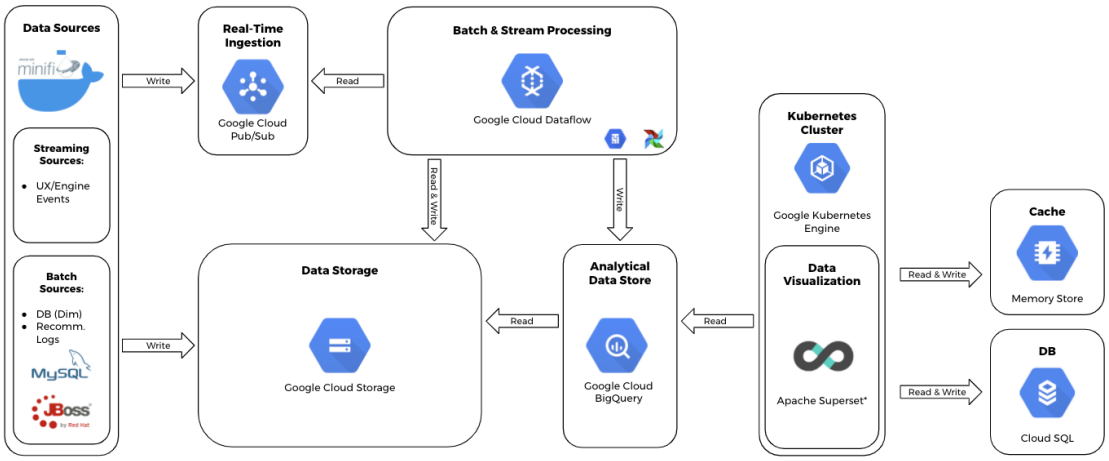

Figure 2.2. Data Insights architecture

Our focus will be mainly on the last layer and on part of the analytical one.

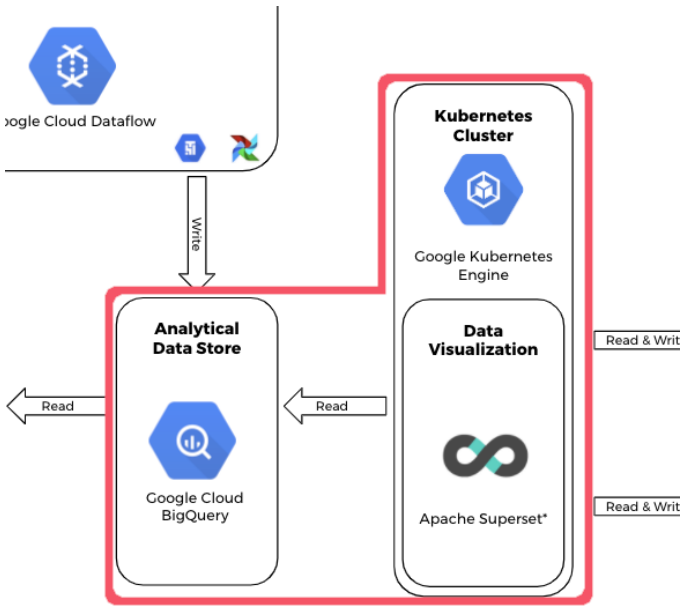

Figure 2.3. Data Insights architecture, our area of interest

2.1.2 Apache Superset overview

Apache Superset is the tool in use in the current version of *Data Insights*. It is an open-source enterprise-ready business intelligence web application. Its main goals are to help with:

- *Data visualization*: creating visual representations to represent data in a more understandable way.
- *Data exploration*: looking at the data from different perspectives.
- *Data Analysis*: extracting information from data from various measurements to define patterns and make business predictions.

We will see more details about how it works in the next chapters.

2.2 Project's goal and structure

The current implementation is based on Apache Superset 1.0, an open-source business intelligence web application, which provides most of the needed functionalities out of the box. It is a really good product, but it presents some important limitations:

- Dashboard stability: after refreshing a dashboard, colors in charts are incoherent to the previous representation.
- Dashboard spaces management: the way users can edit charts' dimensions and positions is quite limited
- Missing cross-filtering
- Filter impact visualization: when applying a specific filter, it is not clear which charts are affected by it.
- Overall charts customization: e.g. users can only select color palettes and not individual colors.

All these problems have been identified after a confrontation with the Customer Success department of the company which, among other things, is the one in charge of creating dashboards and presenting them to the customers. They presented us with the main drawbacks they faced while working with the tool and also with some observations made by the customers themselves.

The aim of this thesis is to analyze which are the major functionalities offered by the main actors in today's BI/Data Visualization market and try to understand which could be the best path to follow in order to develop a solution that meets the company's needs and possibly fixes all the above-mentioned issues.

We can divide the project into three phases:

- *Market analysis*: in this phase, we will look at a few of the most relevant BI/Data Visualization tools available in the market to understand which are their most interesting features and their weaknesses in order to improve our strategy and our business decisions.
- *Custom solution development*: during this step, we will develop a web application based on React, that should allow users to create different types of charts, perform queries, and visualize the information through customizable dashboards.
- *Hybrid solution development*: here we will adopt the last release of Apache Superset (which comes with many improvements over the version in use in the current solution) with the addition of a semantic layer between the data source and the visualization layer.

In both solutions, we will adopt Cube as a semantic layer in order to evaluate the benefits that may come with it.

2.2.1 MovieLens dataset overview

For this project, we used the data collected from MovieLens and made available by GroupLens Research, a research lab at the University of Minnesota.

MovieLens 25M Dataset

This dataset, ml-25m, describes a 5-star rating and free-text tagging activity from MovieLens, a movie recommendation service. It contains 25000095 ratings and 1093360 tag applications across 62423 movies. These data were created by 162541 users between January 09, 1995, and November 21, 2019. This dataset was generated on November 21, 2019. Users were selected at random for inclusion. All selected users had rated at least 20 movies. No demographic information is included. Each user is represented by an id, and

no other information is provided.
The dataset contains the following tables: *Ratings, Movies, Tags, Links, Genome-scores, Genome-tags.*
In this project, we will mostly focus on the first two tables.

- *Ratings Data File Structure*: All ratings are contained in the file ratings.csv. Each line of this file after the header row represents one rating of one movie by one user, and has the following format:

userId	movieId	rating	timestamp

Table 2.1. ml-25m, Ratings table structure

  Ratings are made on a 5-star scale, with half-star increments (0.5 stars - 5.0 stars), while timestamps represent seconds since midnight Coordinated Universal Time (UTC) of January 1, 1970.

- *Movies Data File Structure*: All tags are contained in the file tags.csv. Each line of this file after the header row represents one tag applied to one movie by one user, and has the following format:

movieId	title	genres

Table 2.2. ml-25m, Movies table structure

  Movie titles also include the year of release in parentheses. Genres are a pipe-separated list, and are selected from the following: Action, Adventure, Animation, Children's, Comedy, Crime, Documentary, Drama, Fantasy, Film-Noir, Horror, Musical, Mystery, Romance, Sci-Fi, Thriller, War, Western, (no genres listed).

MovieLens 1M Dataset

This second dataset has been used in the *Hybrid solution* to simulate a use case in which we had two different customers, each of them with the respective associated Google BigQuery project and dataset. This test was interesting to perform dashboard duplication tests (we'll talk in more detail when talking about the *Hybrid solution*).
It contains 1,000,209 anonymous ratings of approximately 3,900 movies made by 6,040 MovieLens users who joined MovieLens in 2000. It's made up of

three tables: *Ratings, Movies, Users*.
For continuity with the previous dataset, we placed our focus mainly on the first two tables.

- *Ratings Data File Structure*: All ratings are contained in the file "ratings.dat" and are in the following format:

userId	movieId	rating	timestamp

Table 2.3. ml-1m, Ratings table structure

  UserIDs range between 1 and 6040, MovieIDs range between 1 and 3952, Ratings are made on a 5-star scale (whole-star ratings only), Timestamp represents seconds since midnight Coordinated Universal Time (UTC) of January 1, 1970. Each user has at least 20 ratings.

- *Movies Data File Structure*: Movie information is in the file "movies.dat" and is in the following format:

movieId	title	genres

Table 2.4. ml-1m, Movies table structure

  Titles are identical to titles provided by the IMDB (including the year of release), Genres are pipe-separated and are selected from the following genres: Action, Adventure, Animation, Children's, Comedy, Crime, Documentary, Drama, Fantasy, Film-Noir, Horror, Musical, Mystery, Romance, Sci-Fi, Thriller, War, Western.

Chapter 3

Business Intelligence

Business Intelligence (or *BI*) can be considered a technology-supported process or a collection of activities and methods to gather, transform and present organized information from fragmented data from multiple sources lowering the needed time to extrapolate useful knowledge that can support an organization's decision-making processes.

3.1 Data Exploration

Data Exploration is a key part of BI. It involves the process of discovering patterns, trends, and relationships within data. This step is essential to get a better understanding of the data and to possibly gain insights to inform business decisions. Some examples of techniques used for this purpose are data mining, data profiling, and data visualization.

- Data Mining is a technique that mainly uses statistical models to elaborate and explore a huge amount of data in order to discover underlying schemas and hidden relationships between data. In this case, the term "mining" is more related to the act of "extracting" the relevant information. This technique is often used in classification, clustering, or association rules problems.
- Data Profiling is the process that identifies data quality issues, such as unknown or missing values, incorrect or duplicate data, values outside the normal range and so on.
- Data Visualization involves the use of visual representations to help users to interpret and understand the data. Other than that, it also provides

an intuitive way to present data to non-technical audiences. We'll dig into details in the next paragraph.

A deeper look into *Data Visualization*

As anticipated above, Data visualization is the graphical representation of data and information. It uses multiple visual elements, like charts, graphs, maps and more.
The main strength of these tools is that, if properly made, they can easily deliver the desired information to the observer, independently of his level of expertise. We live in a world that is surrounded by visual information, from advertisements, TV shows, and art. This can be considered one of the most universal and easily understandable languages to communicate with people from all around the world. By using colors, shapes and patterns is possible to instantly grab the attention of the spectator and deliver the desired message. Other than that, these visualization tools open the door to an interactive exploration of the data, increasing user engagement with the representation. To reach this goal, it is necessary to pay attention to the data we are working with and to the way the message is delivered. Charts and graphical representations must be built appropriately to be sure that the core information is not lost when the final user is interpreting the data.

Chapter 4

Semantic layer

Data is the key factor on which business decisions are based today. The increased adoption of cloud computing technologies led to an impressive volume and data distribution that added complexity to enterprises' data management; for example, they now need to operate with multiple data formats, definitions, and types.
To help solve these problems, semantic layers have been introduced.

4.1 Overview

Designing dimensions and defining metrics is not a new idea for companies. What has changed in today's market is that they no longer use only one business intelligence tool, but, depending on the needs of each team, different visualization tools might be adopted.
Around the end of the 20th century, BI tools were mainly built around SQL statements and cron jobs (a way of scheduling tasks to be executed in the future; generally used to schedule jobs that are periodically executed) and this made it very challenging to manage data pipelines. Other than that, storing huge amounts of data was expensive, and that increased the level of attention needed when extracting, transforming, and loading a portion of the dataset for consumption.
In the "traditional BI" scenario, there is an IT department in charge of formulating data analytics processes to achieve business objectives. Other departments store their data and then ask the IT team to generate performance reports, trends forecasts, and more. After 2010, as interactive dashboards became the primary way for data analyst teams to help business stakeholders consume their data, *self-service BI tools* emerged. These are tools that

opened also to non-technical users the possibility to filter, sort, and analyze data without the need for IT/BI technical teams.
As more companies invested in data, new low-code or no-code tools started to appear on the market. On one hand, many people were able to find more suitable products to their needs, but on the other, a new scenario occurred: inside the same company different teams might adopt different BI tools, each of them defining in their own way the metrics.
After quite some time struggling with this problem, in the data community the concept of *Headless BI* started to grow. These types of tools are agnostic from the presentation layer; hence the name *headless*.
A semantic layer is an example of *headless BI tool*; it can be seen as a business representation of the data that provides a unified view of data definitions from multiple data sources. This provides business users with an easy way to understand the data. The goal is to abstract metrics to a higher level, bringing them closer to experts in the specific business unit of the company. Looking at it in more detail, we can identify four main components (*Data Modeling, Access Layer, Caching and APIs*) that will be deeper analyzed in the next paragraphs. Typically, the semantic layer is a middleware between the data source and the data application.

For this project, we chose to use *Cube*, an open-source analytics framework that has been first released in 2019 and has a continuously growing community.

4.2 Cube

> "Cube is the Semantic Layer for building data apps. It helps data engineers and application developers access data from modern data stores, organize it into consistent definitions, and deliver it to every application."

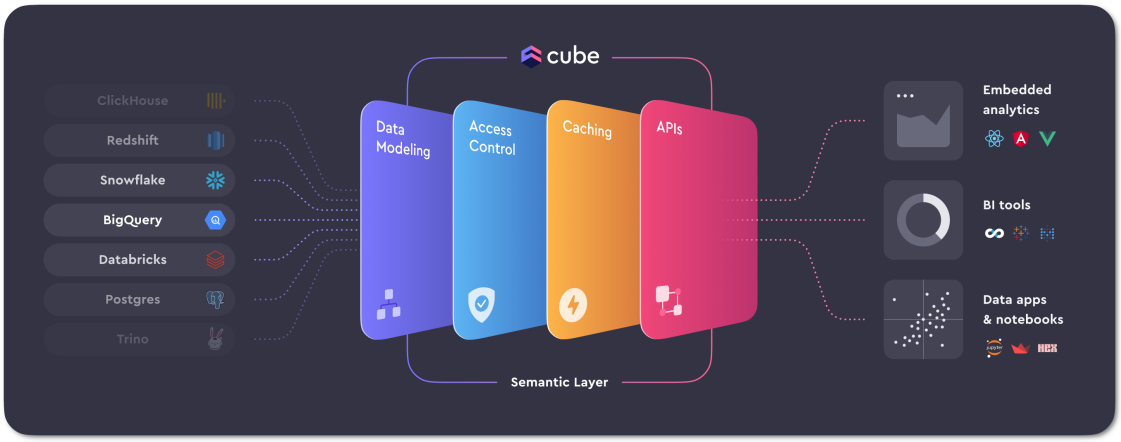

Figure 4.1. Cube overview

Usually, when building an application with analytics features is probable to face these three problems:

- *SQL code organization*: having to model many metrics and dimensions using pure SQL can become hard to maintain.
- *Performance*: "time-to-insight" metric has become more and more important. Using only SQL queries is not the fastest way to obtain results nowadays.
- *Access control*: all downstream data-consuming applications must have controlled and secured access to data.

Through its components, Cube is able to provide a solution to all the above-mentioned issues.
This framework is characterized by four main areas: *Data Modeling, Access Control, Caching, APIs.*

Data Modeling

Data Modeling in Cube is based on the concept of *Data Schema*, which is a way of modeling raw data into more useful business interpretations and pre-aggregating data in order to obtain optimized results. All this is obtained through JavaScript files.
In Cube, each table of data is represented as a *cube*, generally declared in

separate files, with one cube per file.
Within each of them, users can define:

- *Measures*, referred to as quantitative data. Each measure is an aggregation over a certain column in the database table. They are defined using different parameters (like the sql expression, the type, the format, the rolling window, and more...), and it is even possible to create *Calculated Measures* that are calculated using other measures.

```
cube(`JoinUnnest`, {
  sql: `SELECT * FROM movielens.join_unnest`,

  measures: {
    count: {
      type: `count`,
    },
    positiveRatings: {
      type: `count`,
      sql: `rating`,
      filters: [{
        sql: `${CUBE}.rating > 3.5`
      }],
    },
  },
  ...
});
```

```
cube(`JoinUnnest`, {
  sql: `SELECT * FROM movielens.join_unnest`,

  measures: {
    percentagePositiveRatings: {
      type: `number`,
      sql: `ROUND(
        ${CUBE.positiveRatings}/${CUBE.count}*100, 2
      )`,
      format: `percent`,
    },
  },
  ...
});
```

Figure 4.2. Cube Data Schema *Measures* and *Calculated Measures*

- *Dimensions*, referred to as categorical data. They represent the properties of a single data point in the cube. If they are time-based properties (type: ‘time‘) they are called *TimeDimensions*. Similarly to measures, they are characterized by multiple properties.

```
cube(`JoinUnnest`, {
  sql: `SELECT * FROM movielens.join_unnest`,

  dimensions: {
      movieTitle: {
        sql: `movie_title`,
        type: `string`
      },
      rating: {
        sql: `rating`,
        type: `number`
      },
      timestamp: {
        sql: `timestamp`,
        type: `time`
      },
      pn_rating: {
        type: `string`,
        case: {
          when: [
            {sql: `${rating} < 3.5`, label: `Negative`}
          ],
          else: {
            label: `Positive`
          }
        }
      }
    },
});
```

Figure 4.3. Cube Data Schema *Dimensions*

- *Joins*, to define relationships between cubes. The relationship between the two cubes can be described with one of these options: *belongsTo, hasMany, hasOne.*

```
cube('Authors', {
  joins: {
    Books: {
      relationship: `hasMany`,
      sql: `${CUBE}.id = ${Books.author_id}`,
    },
  },
});
```

Figure 4.4. Cube Data Schema *Join* example

- *Segments*, that can be seen as predefined filters. They come in handy when there is a complex filtering expression that can be reused for many

other queries.

```
cube(`Users`, {
  segments: {
    sf_ny_users: {
      sql: `${CUBE}.location = 'San Francisco' OR
${CUBE}.location like '%New York%'`,
    },
  },
});
```

Figure 4.5. Cube Data Schema *Segment* example

- *PreAggregations*, a way of caching expensive and frequently-used queries (they will be discussed in a more detailed way in the next paragraphs when talking about caching).

```
cube(`Ratings`, {
  sql: `SELECT * FROM movielens.movielens25m`,

  preAggregations: {
    main: {
      measures: [`${CUBE}.count`],
      dimensions: [`${CUBE}.rating`],
      timeDimension: `${CUBE}.timestamp`,
      granularity: `month`
    }
  },
});
```

Figure 4.6. Cube Data Schema *PreAggrgation* example

Access Control

In Cube, access control (or *authorization*) is based on the concept of *security context*, while *authentication* is handled outside.

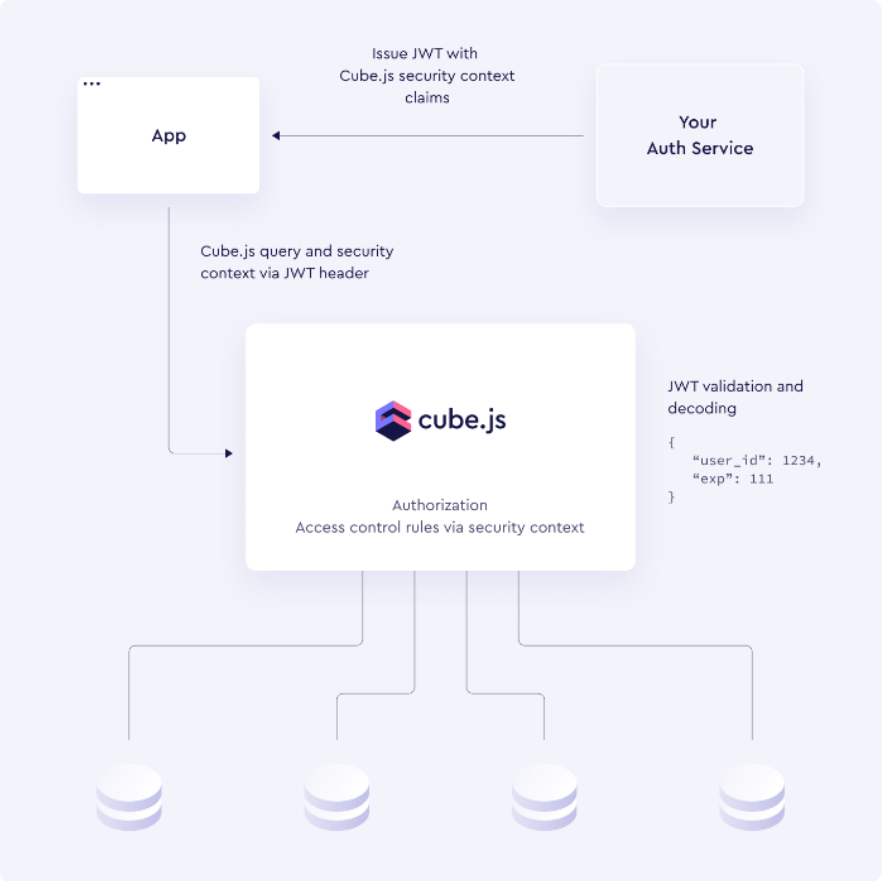

Figure 4.7. Cube *Access Control* use case example

In the above picture, we can see an example of a use case, in which a web server serves an HTML page containing the Cube client, which needs to communicate securely with the Cube API. To achieve this, the web server should generate a JWT (JSON Web Token) with an expiry timestamp. The server could include the token in the HTML it serves or provide the token to the frontend via an XHR request, which is then stored in local storage or a cookie. The JavaScript client is initialized using this token, which will be included in the following calls to the Cube API. The token is received by Cube and verified using any available JWKS(JSON Web Key Sets), if configured. Once decoded, the token claims are injected into the security context to evaluate access control rules.
Cube allows you to define granular access control rules for every cube in your data schema. It uses both the request and security context claims in the JWT token to generate a SQL query, which includes row-level constraints

from the access control rules.
The additional information about the user passed via JWTs is called *security context*. It can be seen as a verified set of claims about the current user that the Cube server should use to ensure that users can only have access to the data that they are authorized to access. To retrieve this property we have two options, via *queryRewrite* configuration option or via *COMPILE_CONTEXT* global variable. The main difference between the two is that *COMPILE_CONTEXT* is used by Cube at schema compilation time, which allows changing the underlying dataset completely, while the *queryRewrite* is only used at query execution time, which simply filters the dataset with a *WHERE* clause.

Caching

Cube provides a two-level caching system. The first level is called *in-memory cache* (active by default) while the second level is called *pre-aggregations* (requires an explicit configuration to be activated).

In-memory cache acts as a buffer for your database when there's a burst of requests hitting the same data from multiple concurrent users. It leverages an SQL statement as a key, that is used upon every incoming request to check the cache. If nothing is found inside of it, the query is executed in the database: the result set is returned and the cache is updated. If an existing value is present in the cache and the *refreshKey* value for the query hasn't changed, the cached value will be returned. Otherwise, an SQL query will be executed against either the pre-aggregations storage or the source database to populate the cache with the results and return them.
Refresh keys represent the "trick" used by Cube to prevent unnecessary, and potentially expensive, queries from hitting the database. By default, *refreshKey* values are:

- every 2 minutes, for BigQuery (our case), Athena, Snowflake, and Presto.
- every 10 seconds, for all the other databases.

but they can be modified in the configuration of each Cube.

Pre-aggregations are designed to provide the right balance between time to insight and querying performance. The effects of this layer become more noticeable as the size of the underlying dataset grows. In fact, *pre-aggregation*

can be seen as a condensed version of the source data. By specifying particular attributes from the source data, Cube is able to *crunch* the data and reduce the size of the dataset by several orders of magnitude, ensuring at the same time that subsequent queries can be served by the same condensed dataset if any matching attributes are found. *Pre-aggregations* are defined within each cube's data schema, and cubes can have as many *pre-aggregations* as they require.
Also, in this case, a refresh strategy is available. By default, the *refreshKey* value is set to '1 hour', but it can be customized by setting the specific property in the pre-aggregation configuration.
The pre-aggregated data can be stored either alongside the source data in the same database, in an external database that is supported by Cube, or in Cube Store, a dedicated *pre-aggregation* storage layer.

APIs

Here is where the "*headlessness*" nature of this type of tool is highlighted. Headless BI tools must be able to provide their data to every *head* application, independently by them being used for data visualization, automation, embedded analytics, or dashboarding. This is obtained thanks to different types of APIs. In Cube we have three types of APIs:

- *SQL API*: allows querying Cube via Postgres-compatible SQL. It enables the use of BI applications on top of Cube. In this scenario, each cube is represented as a table. Measures, dimensions, and segments in this table are columns. The SQL API transforms SQL query fragments from cube tables into Cube's internal query format through a process called *Cube query rewrite*. This type of API is generally used together with dashboard tools, notebooks, and legacy applications, such as Tableau, that users formerly would have directly connected to a data warehouse.

- *REST API*: are used to communicate with Cube's backend. They allow users to get the data for a query, get the SQL Code generated by Cube to be executed in the database, get meta-information for cubes defined in data schema, trigger a scheduled refresh run to refresh pre-aggregations, submit pre-aggregation build jobs and retrieve their refresh status, check the ready state and liveness state of the deployment.

- *GraphQL API*: represents the last addition to Cube's APIs. It has grown in popularity due to its ability for clients to ask for exactly the data they

need and nothing more. After receiving the GraphQL query, Cube will translate it into a native SQL for the client's database and execute it.

4.2.1 Architecture

Cube is generally deployed as an additional microservice in existing application architectures. The backend microservice is connected to one or multiple databases, taking care of database queues, data schema, caching, security, and API gateway, while the Cube client loads aggregated data from the backend, processes it, and sends it to the chosen visualization library.

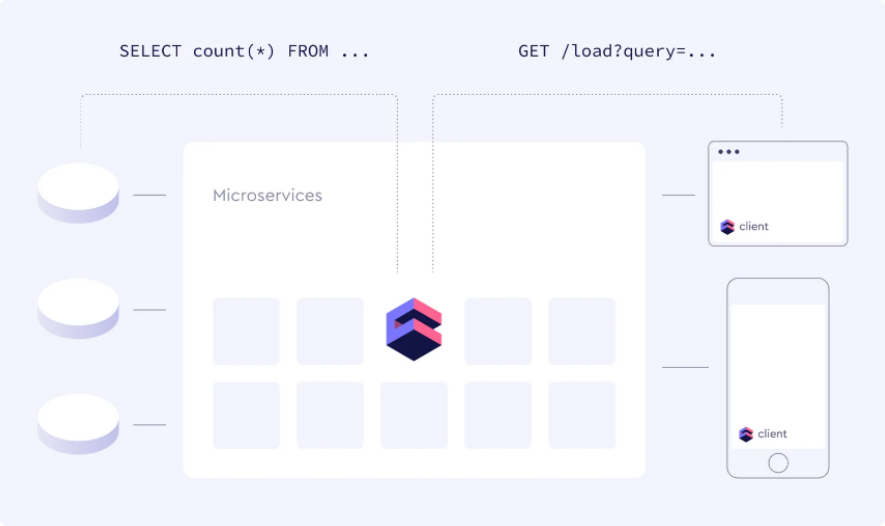

Figure 4.8. Cube architecture schema

Speaking about a production deployment, we generally have one or multiple API instances, a Refresh Worker, and a Cube Store cluster.

- *API instance*: processes incoming API requests and queries either Cube Store for pre-aggregated data or connected data sources for raw data. It is possible to horizontally scale API instances and use a load balancer to balance incoming requests between multiple API instances.

- *Refresh Worker*: updates pre-aggregations and the in-memory cache in background. It also keeps the refresh keys up-to-date for all defined schemas and pre-aggregations.

- *Cube Store*: is the purpose-built pre-aggregations storage for Cube. It uses a distributed query engine architecture. In every Cube Store cluster there are different components:
  - *Router nodes*: one or many of them to handle incoming connections, manage database metadata, build query plans, and orchestrate their execution.
  - *Worker nodes*: multiple nodes to ingest warmed-up data and execute queries in parallel.
  - *Blob storage*: local or cloud-based blob (Binary Large Object, a type of cloud storage for unstructured data) storage that keeps pre-aggregated data in columnar format. It can be run in single-instance mode but is generally better to run it as a cluster of multiple instances to increase concurrency.

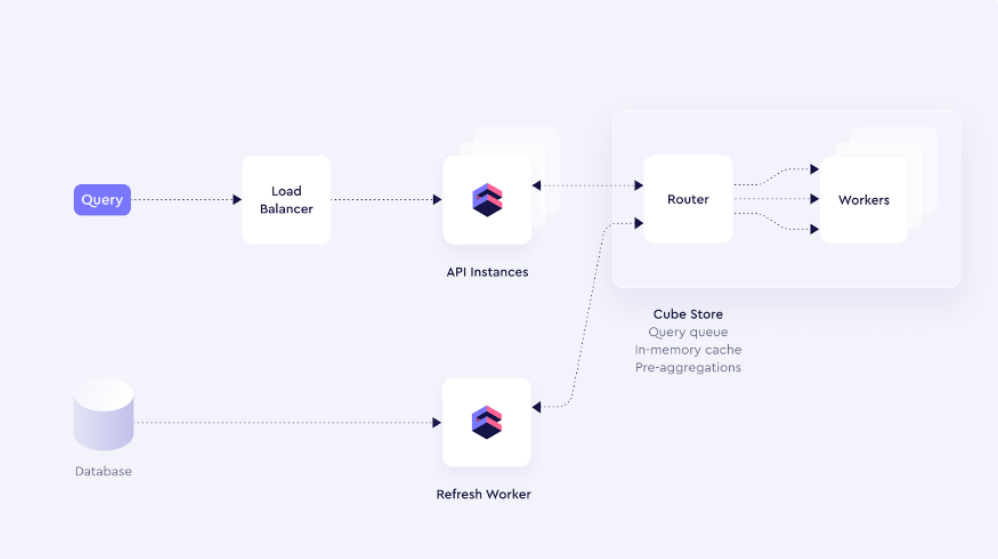

Figure 4.9. Cube production deployment architecture schema

Chapter 5

Web-based Applications

5.1 Multi-page vs Single-page Applications

Multi-page applications and single-page applications are two common architectural patterns used for web development.
Multi-page applications (MPAs) represent the traditional way in which web applications were made. Each user interaction (e.g. submitting a form) or action that changes what is displayed, causes the browser to request a new page from the server and reload the entire document. *Single-page Applications* (SPAs) consist of a single page that is rendered on the client-side using JavaScript. MPAs rely on server-side rendering and full-page reloads, while SPAs rely on client-side rendering and partial page updates. Both solutions have some advantages and disadvantages. MPAs are very good for Search Engine Optimization (SEO) performances, but they provide a worse user experience (UX), as each page reload causes a delay and possible white screens. Other than that, the consumption in terms of bandwidth and server resources is higher. SPAs allow for a better UX, as each page only transfers the necessary data and does not need to reload the entire page. They also perform better from the bandwidth consumption point of view. On the other hand, they have worse SEO performance, as each page comes with the same URL and search engines find it hard to index metadata.
The choice of a pattern over the other depends on the requirements and the aim of the web application. Multi-page applications are generally more suitable for static websites that may need to be SEO-friendly. Single-page applications are more appropriate for dynamic websites that need to provide an interactive user experience.

In our implementation, we decided that a *Single-page Application* was the best solution.

5.2 Development Tools

In this section, we will illustrate the tools that have been used to develop the custom web application.

5.2.1 React

React is a JavaScript library for creating user interfaces (UIs). It was released by Facebook in 2013 and soon became one of the most popular and most adopted tools for web development.

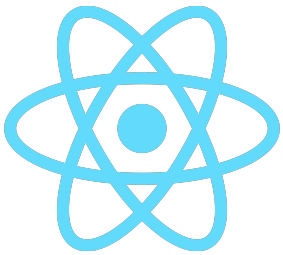

Figure 5.1. React logo

It is characterized by a declarative approach for rendering UI components, meaning that users are in charge of describing *what* they want to see in the UI and not *how* (imperative approach) things should be rendered. The only focus is on the *final state that the UI is in*. React is able to take charge of all the things that need to happen to ensure that the UI is properly represented. One of the problems that React aims to solve concerns DOM modification. The Document Object Model (DOM) is the data representation of the objects that comprise the structure and content of a document on the web. Manually performing operations on the DOM is really slow, but unfortunately, it is necessary in single-page applications in order to respond to users' actions and provide them with new content. Using React, the programmer will never directly modify the DOM but he will modify a *virtual DOM* instead. This operation is very fast and then React will take care of updating the actual DOM during a process called *reconciliation*. In this phase, React will figure out which are the changes that actually matter and then will make the least

amount of modification possible to the actual DOM.
React is characterized by three main concepts:

- *Components*: React uses a component-based architecture. Each visual element is broken down into smaller and reusable building blocks called components; they can have their own state and logic and can then be combined together to create complex User Interfaces.
- *Props*: they are custom arguments that can be accepted by components and can eventually change their behavior or aspect once they are rendered on the screen. *Props* can be passed from the parent component to child components, following what is called *one-way data flow*, which is a React characteristic for which data flows down the component tree.
- *State*: it can be seen as any information in the User Interface that can change over time, generally after being triggered by user interaction. To add this *state* logic to components, React provides a set of functions called *state hooks*.

The description of *how* the UI should look is done using *JSX*, which is a syntax extension that allows writing HTML-like code in JavaScript. This is very useful because it allows leveraging all the power and flexibility provided by JavaScript while still writing in a more familiar syntax. JSX is then compiled into JavaScript and rendered by React into the DOM. Other than that, is worth mentioning that in React JavaScript code and visuals often are in the same location, so there is no need of handling multiple files to define the behavioral and look aspects of a single component.
As specified at the beginning of this paragraph, React is a library, not a framework; so it does not provide an indication of the general behavior of the whole application.
Let us do now a short digression speaking about the MVC architecture. It is an architectural pattern that splits the application into three logical parts: the *Model*, the *View*, and the *Controller*. The Model is in charge of managing all the activities related to the data, so it handles database connections and it answers the requests made by the Controller. It never communicates directly with the View. The View part is in charge of providing a visual representation of the data by generating the UI for the final user. The View communicates directly only with the Controller. The Controller is the component that allows interconnecting the Model and the View by telling the Model what type of data it needs. After it receives data from it, it elaborates them to provide them in a suitable format for the View.

Going back to React we can say that it can be seen as the *V in the MVC architecture*. All its concerns are related to visual representation and keeping the visual elements up to date. For the M and C parts other technologies can be used, making React suitable not only for the development of new web applications but also for the refactoring of existing ones.

5.2.2 NextJS

Next.js is a React framework that allows developers to create fast and user-friendly web applications.

Figure 5.2. Next.js logo

When building an application you can identify different parts: *UI, Routing, Data Fetching, Rendering, Integrations, Infrastructure, Performance, Scalability, and Developer Experience*. React is a library that helps developers build UIs, but does not cover all the other parts that we have mentioned above. On the other hand, *Next.js* let you implement the UI using React and then incrementally provides features to solve the other application requirements. Here we will take a look only at the most important ones.

- Next.js comes with an *automatic code splitting and bundling* feature. When an application is moved from the development environment to the production one, the code needs to be compiled, bundled, minified, and split. Next.js is able to handle these transformations and help move the application into production. For *compiling* we refer to the process of taking code in one language (e.g. JSX) and outputting it into another language (e.g. JavaScript). The term *minifying* describes the mechanism of removing unnecessary code (e.g. comments) without changing code functionalities with the aim of reducing the size of the files and improving the application performances. With the *bundling* process Next.js is able to reduce the number of requests for files when a user visits a web page by resolving the dependencies and merging the files into optimized bundles for the browser. A web application can have many entry points (e.g.

multiple pages accessible via different URLs) and thanks to *code-splitting* the application's bundle is split into smaller chunks that are required by each entry point. By doing so is possible to improve the initial load time by only loading the code needed to run the page.

- In the client-server architecture we identify the client as the actor (e.g. a browser on a user's device) who sends requests to a server for the application code. After receiving the response, it turns it into a visual interface that the user can interact with. The server, on the other hand, is the one that stores the application code and receives requests from the client, elaborates a response, and then sends it back to the client. This introduction comes in handy when speaking about the *rendering* process, that is the conversion of the React code into the HTML code used for the UI. This mechanism can happen on the server or on the client, either at build time or runtime. Next.js offer three rendering methods: *Client-Side Rendering, Static Site Generation, and Server-Side Rendering* (with the last two that can be also referred to as *Pre-Rendering*). The first method represents what happens in a standard React application, in which the client receives an empty HTML along with the JavaScript instruction to construct the UI (the user will initially see a blank page). It is called that way because the initial rendering is performed on the client side. On the contrary, with the *Pre-Rendering* methods the HTML page is generated on the server side and then sent to the client. With *Server-Side Rendering* (SSR), the HTML page on the server is generated for each request (runtime). Together with the HTML page, the server will send the client also JSON data and JavaScript instructions; React will use these other files to make components interactive through the process of *hydration*. In *Static Site Generation* the HTML page is generated once at build time, then is stored in a Content Delivery Network (CDN) and re-used for each request. This rendering choice can be done on a page-by-page basis depending on the use case.

- Next.js comes with a built-in file-system-based router built around the concept of pages. It automatically creates routes based on the file structure in the *pages* directory. Other *routing* features are the *dynamic routing*, which allows creating routes with parameters and query strings, *prefetching*, which preloads the code for the next page in the background, and *fallback pages*, which can be used to customize a loading indicator or an error page to show when a route is not found.

As we illustrated above, we can see why Next.js has become one of the top and most used React frameworks on the market. It provides many useful features that make it the ideal solution for developers who want to leverage the power of React but at the same time can enjoy a fast and easier development experience with less boilerplate code.

5.2.3 Firestore

Google Firestore is a cloud-hosted, *NoSQL document database* that allows developers to store and sync data for mobile, web, and server applications. It is part of the Firebase platform and Google Cloud services, and it offers several key features and benefits for app development.

Figure 5.3. Firestore logo

One of the main characteristics of *Google Firestore* is its flexible and hierarchical data model. Data is stored in documents, which are organized into collections. Documents can contain fields of various data types, including nested objects and subcollections. This allows developers to model their data according to their needs.
In addition to that, *Firestore* comes with expressive and efficient querying capabilities. Developers can use queries to retrieve individual or multiple documents that match their criteria, with support for filtering, sorting, pagination, and compound queries. They can also create *shallow queries* to retrieve data at the document level without needing to retrieve the entire collection or any nested subcollections. Queries are also indexed by default, which means that they are fast and scalable, regardless of the size of the data set.
A third characteristic of *Google Firestore* is its realtime and offline support. *Firestore* uses data synchronization to update data across multiple devices in realtime, with automatic conflict resolution. Adding realtime *listeners* to the app, it gets notified with a data snapshot whenever the data that the client apps are listening to changes, retrieving only the new changes and

avoiding retrieving the entire database. It also provides offline support for mobile and web apps, by caching data locally and allowing the client to write, read, listen to, and query the cached data. Once the connection is re-established Firestore synchronizes the local changes with its backend. This enables developers to build responsive and collaborative apps that work even in low-connectivity environments.

For this project we used the version 9 of the *Firebase JS SDK*, available as an *npm module*.

5.2.4 Ant Design Charts

Ant Design Charts is a React library of charts based on AntV that provides a set of high-quality and interactive data visualization solutions.

Figure 5.4. Ant Design Charts logo

Ant Design Charts aims to simplify the process of creating charts by providing a declarative and reactive API that can handle various scenarios with ease. Ant Design Charts also follows the design principles and specifications of Ant Design, a popular design system for enterprise-level applications.
It provides many different types of charts (*Line, Area, Column, Bar, Flowchart, Pie, Heatmap, Progress Plots, Relation Plots, and many more*) so that users can choose the one that best fits their needs to explore and understand data patterns and insights.
It allows users to deeply configure the behavior and the look of the charts; they have control over the plot container, the plot components (such as axis, legend, label, tooltip, annotations, etc.), the data mapping, the graphic style and the interactions that the final user can have with the chart.
In conclusion, Ant Design Charts is a powerful library of charts that can help users create beautiful and interactive data visualizations.

Chapter 6

Market Analysis

In the first phase of this project, we looked at some of the products already available on the market to figure out which are their most interesting features and their weaknesses in order to improve our strategy and our business decisions.

We took into consideration four products, that are *Qlik*, *PowerBI*, *Tableau*, and *Looker*.

6.1 Products analysis

6.1.1 Qlik

Qlik, more specifically Qlik Sense, is a data analytics solution that provides the ability to manage data and create visualizations using data from multiple sources.

Figure 6.1. Qlik sense dashboard overview

One of its main characteristics is the use of Artificial Intelligence (AI) to help users to use data more effectively in what they call "Augmented Analytics". Thanks to the Insight Advisor, the user is able to get suggestions about the best types of charts and visualizations, and recommendations about the associations between data sources; everything is dependent on the data that he's working with. Other than that, Qlik provides a natural language conversational analytics experience: this process is based on Natural Language Processing (NLP), to understand user intent, and Natural Language Generation (NLG), to provide insights. This feature can be used to explore data in a faster way: for example, let's suppose that the user is working with a Product Inventory data collection; on the dashboard, he has multiple charts showing different information in different forms and he wants to visualize only products from Italy that are less than 2500. Instead of manually creating the corresponding filters, with Qlik the user can simply ask the Insight Advisor *"Show me the Product Inventory for Italy under 2500"*. This will update the chart visualization with the corresponding results. This is a simple example, but more complex requests could be made:

- *"Show me sales from customers who bought in 2014 Q2"*: applies a time filter for a specific year and quarter, and automatically provides a rank analysis.
- *"Which customers did not buy any Car Boots in Caracas in 2014?"*:

combines an "exclude" category filter with both an "include" category filter and a date filter.

- *"What is the count of Customer that did not buy Bike Helmet?"*: combines aggregation with an exclusion filter.

- *"How much Sales Revenue comes from Company Name with Customer Age between 35 and 55?*: asks for a measure and a dimension, filtered by another measure.

All these just mentioned features are what make Qlik a very interesting product in today's market, but, as always, there are also some drawbacks that must be taken into consideration while analyzing a product. The most relevant can be summarized in:

- *Memory usage*: to provide faster access, the browser version of Qlik stores some data in memory but, sometimes, while working with huge datasets in analysis mode (chart construction phase) more than usual memory space is used, leading to a slowdown in the process and interfering with the proper functioning of the application.

- *Pricing*: licensing can be expensive; different licensing prices must be paid to give access to the dashboards to multiple users in edit or viewer mode.

6.1.2 PowerBI

PowerBI is one of the most used tools in the BI field. It allows users to create effective dashboards, reports, and interactive visualization.

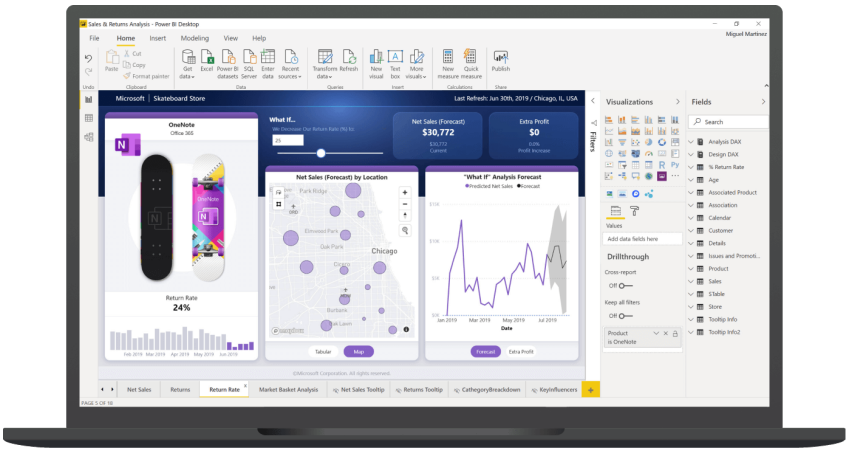

Figure 6.2. PowerBI dashboard overview

One of its main advantages is the ease of use also for non-technical users; thanks to its drag-and-drop feature, dashboards can be created without writing any code. Other than that, it provides the possibility to integrate with other Microsoft products. Also in this case, the tool can be connected to a wide range of data sources, making it easy to bring all the data into one place for analysis.
PowerBi comes with a Desktop and a Web version. A huge drawback, in particular for our scenario, is that these two types of products have different purposes and functionalities: the first one is in charge of handling almost all the chart and dashboard editing functionalities, while the Web version is more suitable for visualization and sharing. The desktop application provides users with advanced features for creating complex reports and dashboards. Users can create custom visuals, import data from multiple sources, and build complex calculations using DAX language. Once the report is created, it can be published to the Power BI service, which is hosted on the cloud. The web application provides users with a platform for sharing reports and dashboards with others. Users can share reports with specific individuals or groups, and the reports can be accessed from anywhere using a web browser or mobile device.

6.1.3 Tableau

Tableau is probably the reference product when speaking about data visualization.

Figure 6.3. Tableau dashboard overview

It is able to deliver great performance when handling huge amounts of data. One of the most useful features is the combination of drag-and-drop capabilities with real-time data exploration: by mixing these aspects, users can easily interact with the visual representation that they are constructing without the need of writing code. Given the huge amount of functionalities that it offers, the learning curve to master the product might be steep.
Also in this case, we have different products: Tableau Desktop provides everything that is needed to access, visualize, and analyze data. All the main editing functionalities for the creation of dashboards and workbooks are available here. Tableau Cloud is more suitable for the publication and visualization of dashboards created with Tableau Desktop.

6.1.4 Looker

Looker is probably the most interesting tool that we have analyzed in this phase.

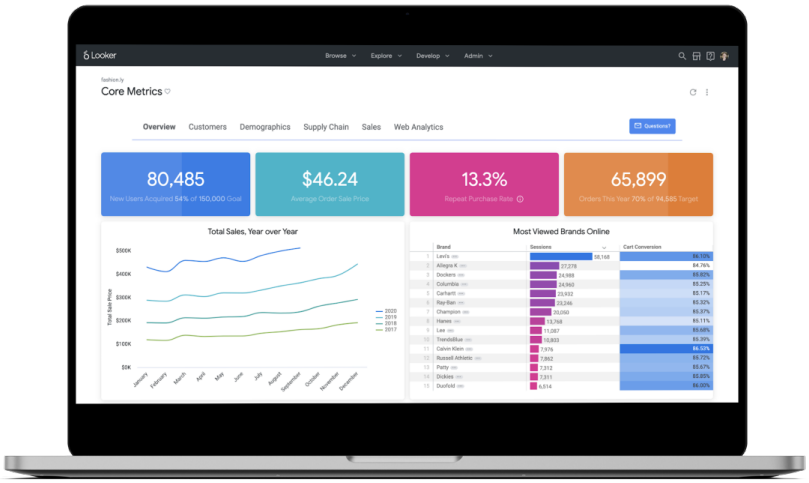

Figure 6.4. Looker dashboard overview

First of all, it is different from the competitors that we have tested in this project from an architectural point of view. It is built on a web architecture hosted in the cloud and it runs completely in the browser, hence, differently from PowerBI, Tableau and Qlik, there is no need of installing a desktop version.
A second noticeable difference is that Looker does not come with an analytics engine of its own but it operates directly on the data in the database, leveraging database features and working on the entire dataset.
Finally, the most interesting feature that makes Looker stand out from the others is the LookML language: it is a modeling language used to describe, with a higher abstraction level, dimensions, measures, aggregated calculations and many more. Thanks to LookML, queries are easy to write and reuse. As it is easy to guess, this is the core feature of Looker, but the fact that is a proprietary language can be seen as a huge downside if we think about a scenario in which a company wants to switch products after having defined all its metrics in Looker's LookML.

6.2 Takeaway

All the analyzed products provide interesting features that make them stand out from the competition.
The product that gets closer to our needs is Looker: we were looking for a solution that could be deployed on the cloud, so we decided to exclude PowerBI, Qlik, and Tableau for their limitations on the web application. Other than that, its semantic layer represents probably the most interesting feature we have observed.
Another important consideration is about licensing costs: all the above-analyzed products offer a free-trial license, which generally becomes quite expensive after its expiration. These expenses are not negligible and represent one of the reasons why we decided to explore possible solutions based on the use of open-source software, trying to understand what could be the possible drawbacks in terms of time-to-market and development effort.

Chapter 7

Custom Web Application

The first solution is a custom React web application that should allow the user to connect to the data source, perform queries, create different types of charts and compose dashboards to visualize the desired pieces of information. With this solution, we didn't address authentication and authorization. We know that these are key components in a real-world product of this type and, for this very reason, there are already plenty of implementations that are currently available. We decided to focus more on the building and the management of dashboards and charts.

7.1 Architecture

As previously mentioned, the architecture of this solution is made up of these components:

- **Frontend**: a graphic interface that allows users to interact with the application. It contains also the Cube Client needed to communicate with Cube Backend. It has been developed using React.
- **Backend**: built using the *Next.js* framework.
- **Firestore**: NoSQL document database used to persist application data regarding charts and dashboards.
- **Big Query**: data warehouse where the client's data (the MovieLens dataset in this case) are stored.
- **Cube**: semantic layer (middleware between the data source and the application) used to increase performances and data reliability.

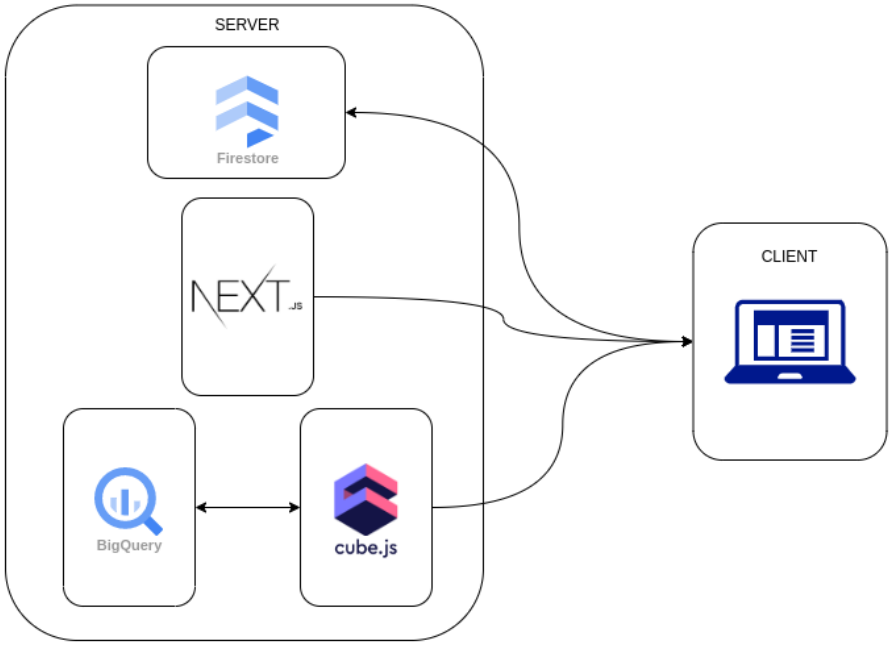

Figure 7.1. Custom web app architecture

In this solution, Cube has been deployed in a Docker container.

7.2 Implementation results

In this section, the custom web-based application will be presented in each of its parts.

7.2.1 Landing page

On the landing page of the application, we can see a list containing all the dashboards that have been created. From the main table, we can get information like the name of the dashboard, the date on which has been created and how many days ago has been modified.

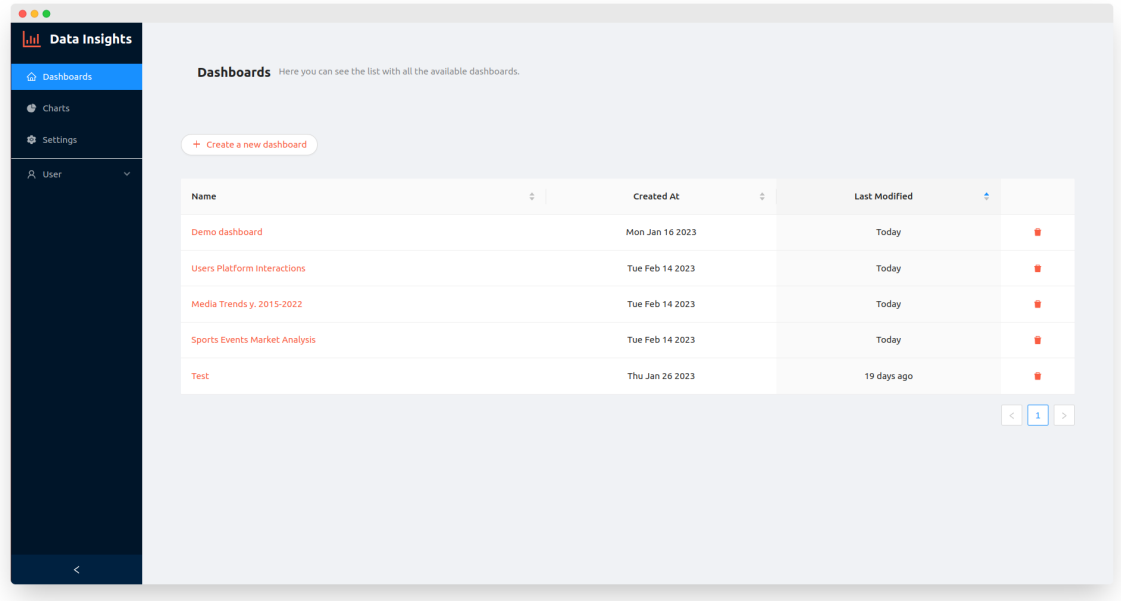

Figure 7.2. Custom Web App - Landing Page

From this page is possible to create a dashboard by clicking on the dedicated button. After that, the user will be redirected to the empty dashboard page. Other than that, is possible to visualize or delete an existing dashboard.

7.2.2 Dashboard Page

A dashboard page can work in two different modes: *visualization mode* or *edit mode*. By default, once the user opens a dashboard, he would enter in *visualization mode*. As the name suggests, each mode offers or limits the actions that can be performed by the user.

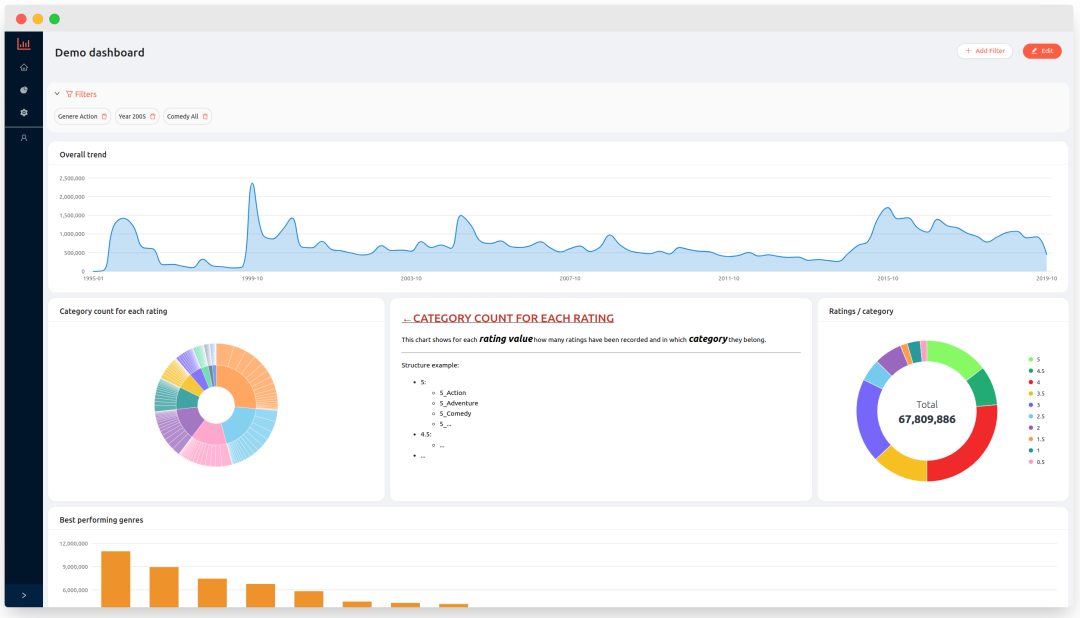

Figure 7.3. Custom Web App - Dashboard Page

The page can be split into three main sections:

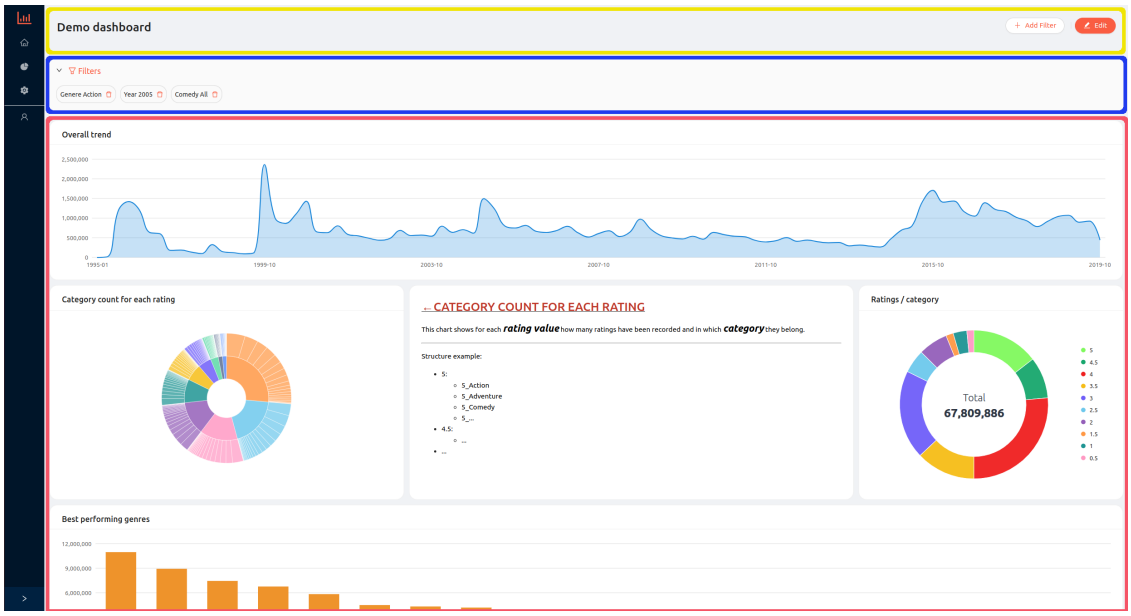

Figure 7.4. Custom Web App - Dashboard Page Sections. *Header* in yellow, *Filter Panel* in blue, *Dashboard* in red

- **Header**: it shows the dashboard's name and a few buttons that depend on the current visualization mode.
  - *Visualization mode*: we have the *Add Filter* button, which allows the user to create dashboard-level filters, and the *Edit* button to switch to the *Edit mode*.

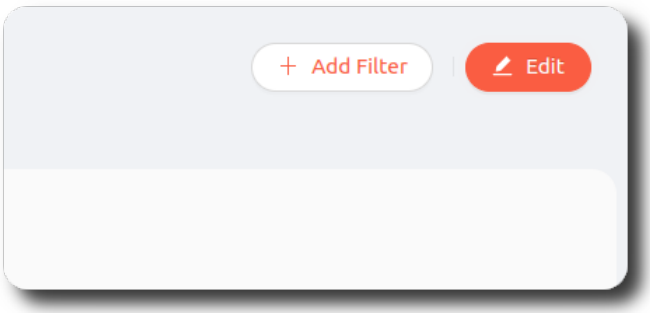

Figure 7.5. Custom Web App - Dashboard Page *Header buttons* in visualization mode

  - *Edit mode*: we have the *Add Textbox* button that allows creating

textbox descriptors, the *Add Chart* button that allows the user to choose between a list of the already created charts that are currently not assigned to any other dashboard, or to create a new chart from scratch. Finally, there is a *Save* button to persist all the changes that have been made and to switch back to *Visualization mode.*

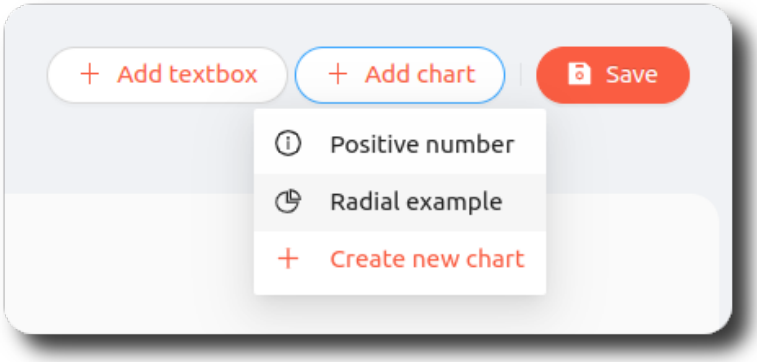

Figure 7.6. Custom Web App - Dashboard Page *Header buttons* in edit mode

- **Filters Panel**: is a collapsible panel that shows the filters that have already been created for the specific dashboard. The *active filters* are highlighted in green to be more recognizable. Hovering over the filter chip, a textual description of the filter will appear to provide details about what information the filter is trying to get. By pressing on the "bin icon" is possible to permanently delete an existing filter.

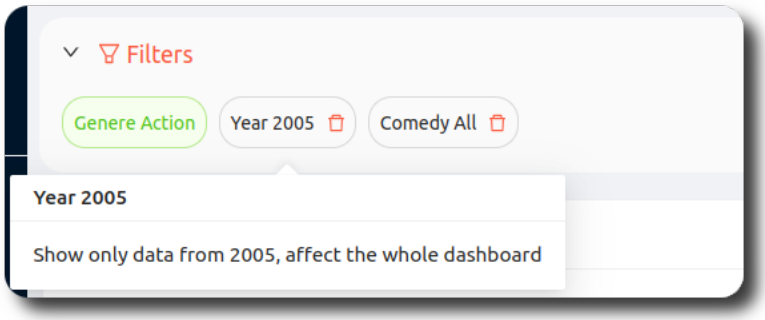

Figure 7.7. Custom Web App - Dashboard Page *Filters Panel*

- **Dashboard**: here is where the content is actually presented to the user. It consists of a *React-Grid-Layout* where items can be displayed, resized,

and moved around, providing huge flexibility when configuring the dashboard layout. In *visualization mode*, items are fixed, while in *edit mode* they can be modified in size and position or eventually be deleted.

Let's now take a closer look at the dashboard items.

Dashboard items

There are three types of possible items in our dashboard.

- **Chart**: here is where the results of the query are shown as charts of various types. The user can interact (e.g. select a slice, scroll the timestamp bar, etc.) with the charts according to the settings selected when it has been created.
  Each chart can possibly be affected by some filters. Each time this happens, a badge with a counter will appear on the top right corner of the card; this indicates the number of filters that are currently affecting the chart and, if the user hovers over it, a dropdown textbox with the information about the names of the filters will appear.
  If the query results are not available, an information message is delivered to the user.

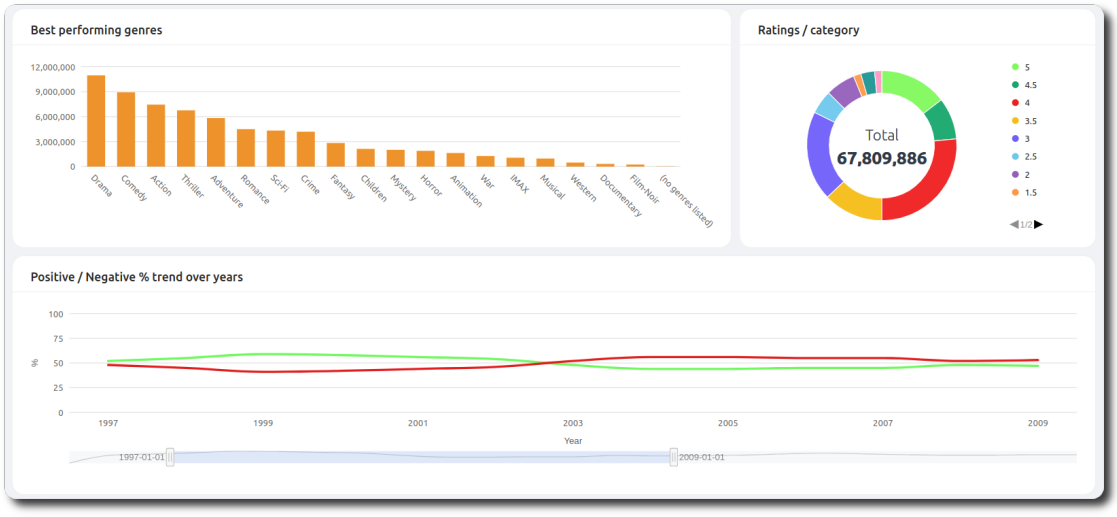

Figure 7.8. Custom Web App - Dashboard Item *Charts* example

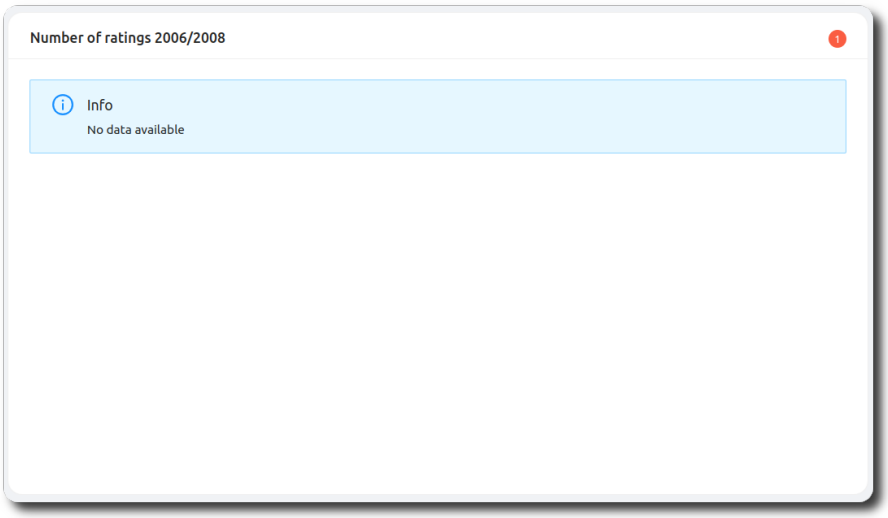

Figure 7.9. Custom Web App - Dashboard Item *Chart Info*

- **Textbox**: is the instrument used to add textual descriptions or comments that could be relevant when visualizing the dashboard. Each of these items is created using a *rich-text editor* that provides many useful editing options: from the basic ones, like font size, formatting, text alignment, and more, to some more advanced ones, like bullet and numerical lists, link embedding, code formatting, emoji support, and more. These text boxes come in handy when there is the need to facilitate chart interpretation, especially when the represented information is complex.

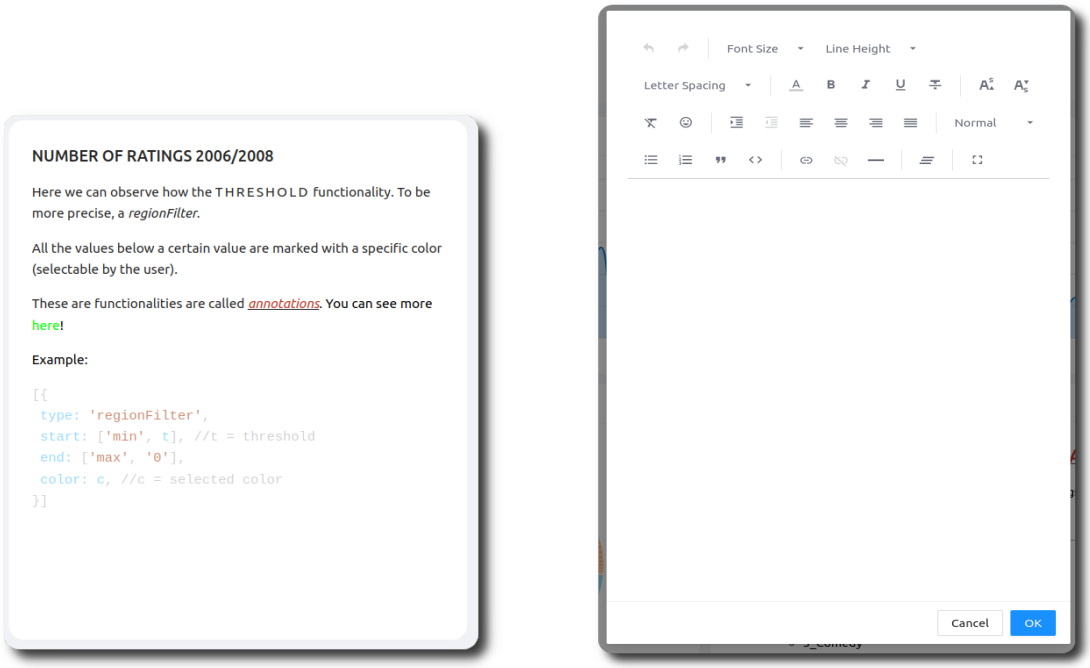

Figure 7.10. Custom Web App - Dashboard Item *Textbox* and *Textbox editor*

- **Filter**: dashboard-level filters represent a high-level interaction that the user can have with the dashboard, in both *visualization* and *edit mode*. Once clicked on the dedicated button, a pop-up menu will present all the options to create a filter.

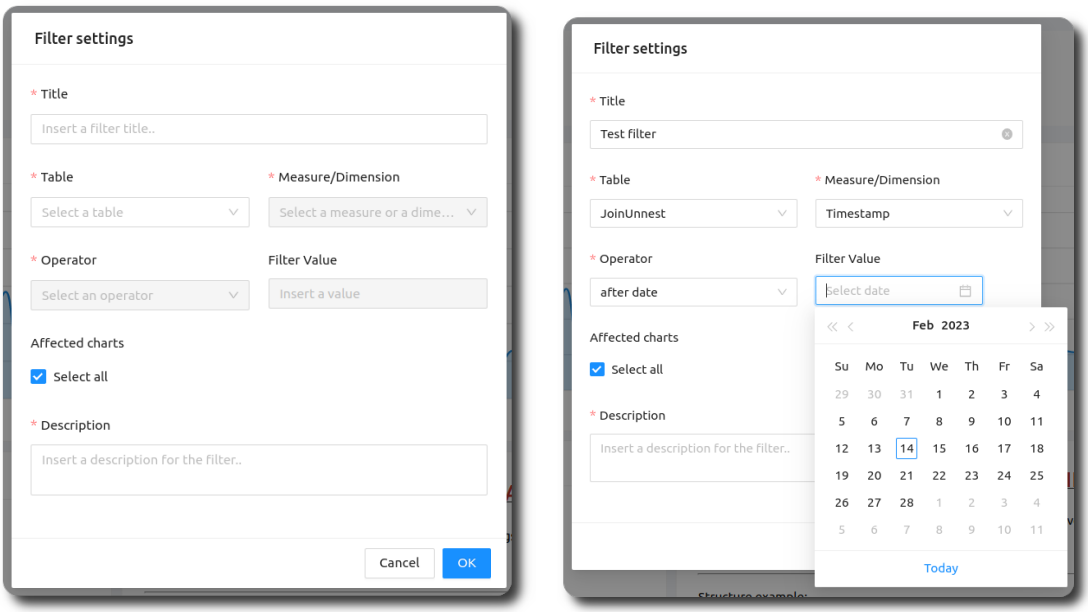

Figure 7.11. Custom Web App - Filter editor modal

The user must provide a series of parameters:

- Title: filter's title
- Table: here the user can choose only from the tables that are currently populating the dashboard's charts. This is done to prevent errors caused by filtering data fields that are not related to the current visualization.
- Measure/Dimension: allow to choose a measure or a dimension that belongs to the previously selected table.
- Operator: based on the type of the selected measure or dimension, different types of operators become available (numerical, text, date).

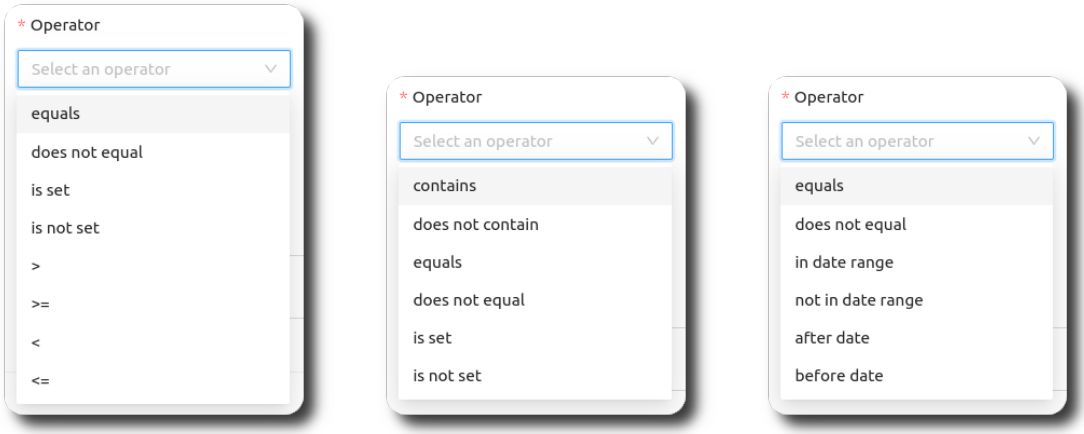

Figure 7.12. Custom Web App - Filter operator types: *numeric*, *text* and *date*

– Filter Value: here the user inserts the discriminant value for the filter.
– Affected Charts: allow to select the charts that are affected by the current filter. Two options are available: *select all* or *manual selection*.

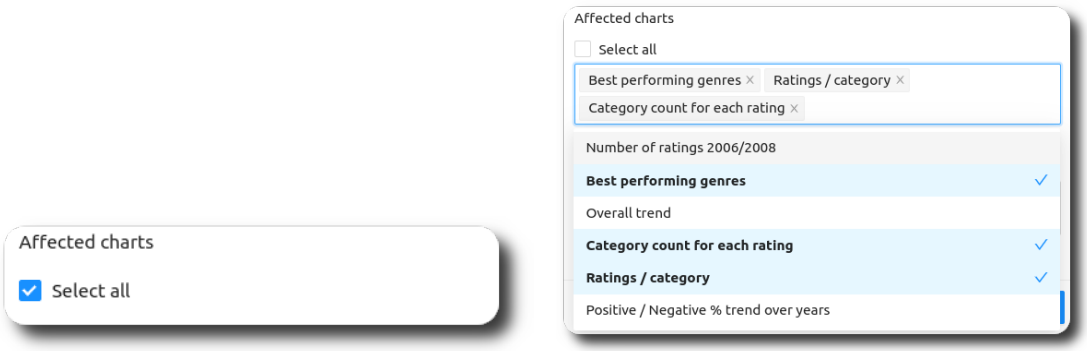

Figure 7.13. Custom Web App - Filter select affected charts

– Description: provide a textual description of the filter that will be displayed when the user hovers on the filter chip in the Filter Panel.

7.2.3 Charts Page

The charts page is where the user can decide to create a new chart or see all the ones that have already been created (in a real-world scenario, the content

of this page will be affected by the permission level of the user and, of course, different customers will only have visibility on their charts).
Here we can get the following information:

- Name: chart name.
- Chart Type: type of the chart; the possible values are: *Line, Area, Bar, Column, Pie, Radial Bar, Sunburst, Number*.
- Dashboard Name: if assigned, the user will see the name (and the link) of the corresponding dashboard, otherwise a *Not Assigned* message will be prompted.
- Last Modified: days since last modifications.

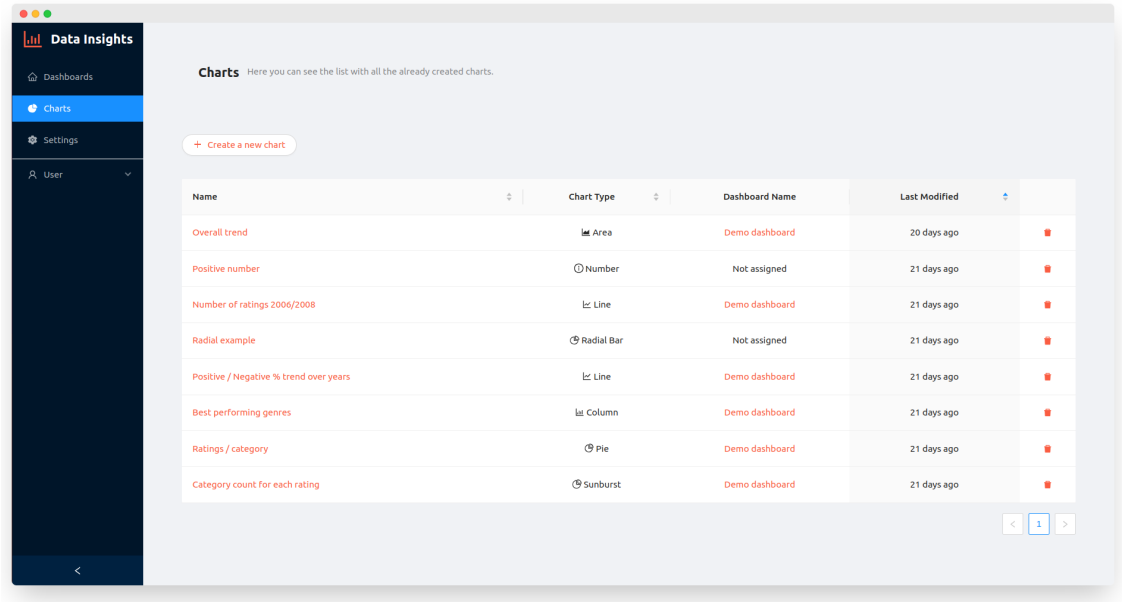

Figure 7.14. Custom Web App - Charts Page

7.2.4 Chart Builder Page

The Chart Builder Page is one of the most important of the entire project. Here users can generate the queries leveraging Cube's APIs and then visualize the result set in the more suitable chart representation.

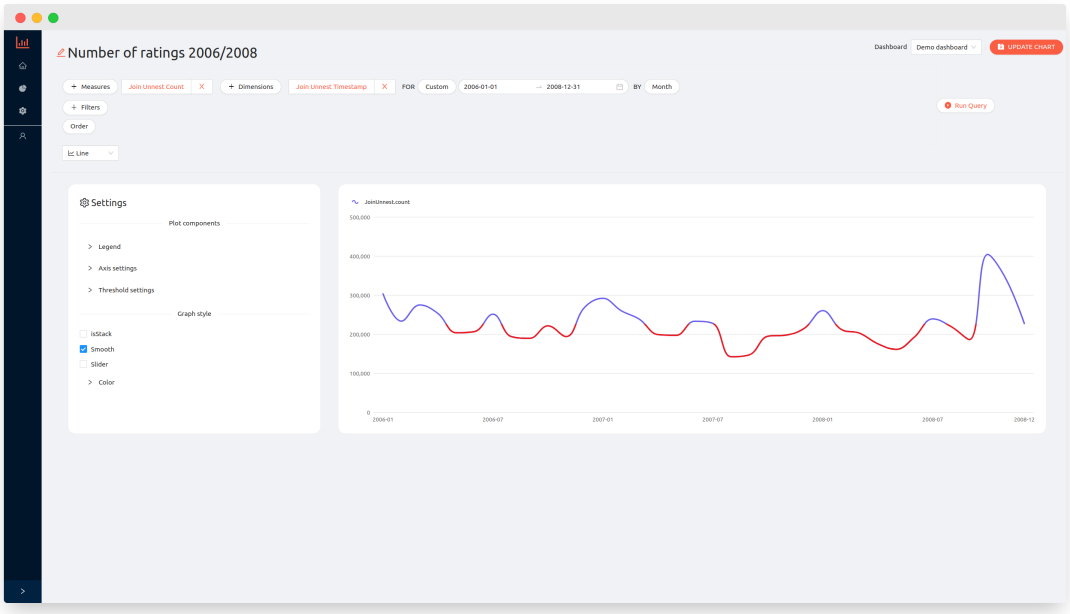

Figure 7.15. Custom Web App - Chart Builder Page

The page can be split into two main sections, the *Query Builder* and the *Chart Renderer*.

Query Builder

In the *Query Builder* section the user can compose queries by choosing the desired values for:

- Measures
- Dimensions
- Time Dimensions
- Filters
- Order

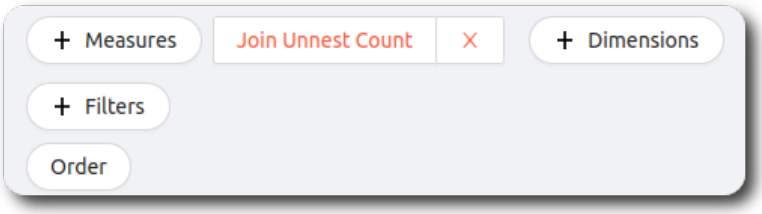

Figure 7.16. Query builder - part one

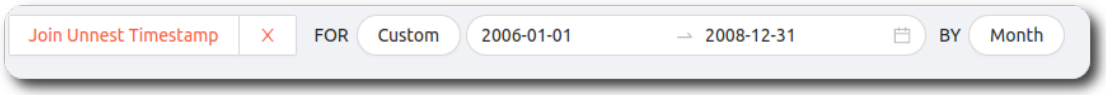

Figure 7.17. Query builder - part two

Each of the elements listed above can be selected via dropdown menus that show the available values from the respective Cubes (tables). The query itself is represented as a JavaScript object that describes an analytics query.

It is possible to directly query Cube backend via REST API, but in our case is much easier to use Cube JavaScript client together with the binding for the React framework.

- *Cube JavaScript Client*: provides methods to communicate with Cube API Gateway in order to retrieve and process data (it's possible to pivot the result set to display as chart or table, split in series or table columns, and many more)

- *Cube React Package*: provides convenient tools to work with Cube in React. We can use the *useCubeQuery* hook together with React functional components to execute Cube queries, and you also have some specific components to separate state management and API calls from rendering code or others to access directly Cube from anywhere in the application.

Chart Renderer

This section is where the chart is rendered. Users can effectively see if the chosen visualization type corresponds to the best way to deliver the desired information.

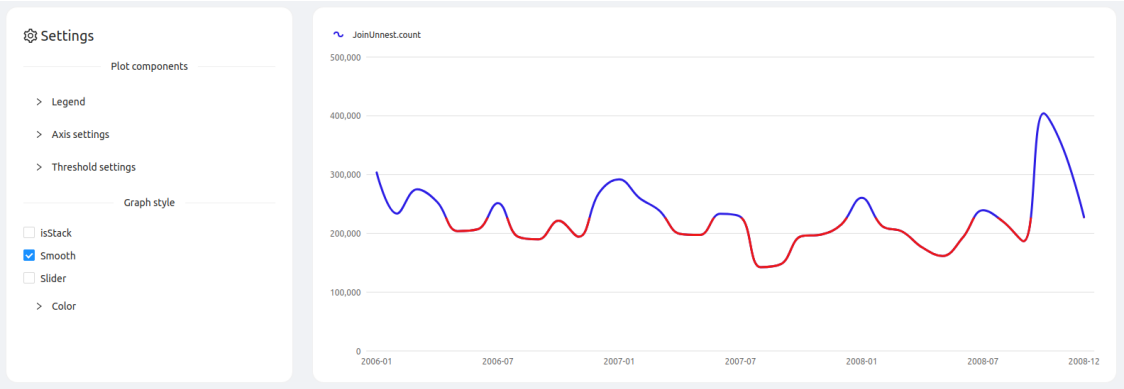

Figure 7.18. Chart Renderer

The ResultSet data returned by Cube backend is elaborated and presented in a chart visualization. Depending on which *interactions* have been enabled, users can interact with it by selecting or highlighting specific areas of the chart.
Thanks to an integration with the Ant Design Charts library, a chart-specific settings panel has been developed. This allows users to customize many aspects of the chart, in order to obtain the desired data representation.
Taking as an example the *Line chart* shown in the above picture, here are the settings that we have decided to make available for the user:

- *Plot Components*
  - *Legend*: allow the user to choose if he wants to see the legend and, if he does, which should be the *layout* (horizontal or vertical) and the *position* (top, bottom, left, right).
  - *Axis settings*: allow the user to choose which axis should be visible and, if it is, which should be its *title* and its *format* (Category or Time). If the axis represents time information, is also possible to provide a *label mask* (e.g. YYYY-MM) to obtain the date/time in the desired format. Other than that, is possible to limit to a *max value* the Y axis.
  - *Threshold settings*: allow the user to set a specific value or use the default one (mean) as a threshold. Values that are below that amount will be represented with a different color that can be set within this setting option.

- *Graph Style*:
  - *isStack*: in the case of a multi-line chart, when this option is enabled, values will be stacked one over the other instead of being independent.
  - *Smooth*: it changes the smoothness of the line that connects the points of the chart
  - *Slider*: used when there is a temporal dimension on the horizontal axis. If enabled, this option transforms it into a slider with an extendable window that can be moved over the whole axis to change interactively the time frame of the chart.
  - *Color*: allow to select the desired color for the line. In the case of a multi-line chart, multiple selectors will become available (one for each dimension value) allowing the user to color pick a specific color for each line.

In this project, we tried to implement eight different types of charts: *Area, Line, Bar, Column, Pie, Radial, Sunburst, and Number*.
This has been done to experience the required time and effort to add a new type of visualization from scratch. Each of them needs to be configured with a specific transformation function that maps the data from the format of the *ResultSet* (returned by Cube backend) into the chart-specific input data format.
Regarding the customization part, using a complex and complete library such as Ant Design Charts allow the developer to choose for each chart and its components the desired level of granularity for personalization.

Chapter 8

Hybrid solution

The aim of this second solution is to evaluate the required effort to integrate a semantic layer into an architecture similar to the one that is currently used by ContwntWise's solution. We will use the release 2.0 of Apache Superset and, in addition to that, Cube will be added as a semantic layer between BigQuery and Superset.
In this phase, we'll focus on the performance enhancement provided by Cube caching technology and we will also test the capability of handling requests from different users, each of them requiring a connection to its own data (e.g. scenario in which each client company has its own BigQuery project).

8.1 Architecture

Here, the architecture is very similar to the one adopted for the current Data Insights implementation in ContentWise (at least for the part related to this project - see figure 2.3 in paragraph 2.1.1). The only difference is the insertion of Cube between Google BigQuery and Apache Superset. By doing so, the visualization tool will be no longer connected directly to a BigQuery project, but to the Cube instance instead.

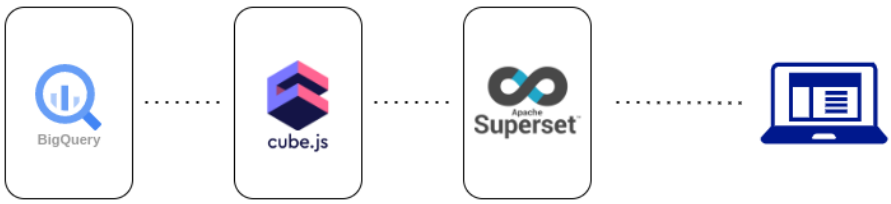

Figure 8.1. Hybrid solution architecture

In this scenario, both Cube and Apache Superset have been deployed on Docker containers.

8.2 Apache Superset

Let us go deeper into what Apache Superset is and which are its main characteristics.

Figure 8.2. Apache Superset dashboard overview

Superset is a cloud-native application, hence is very flexible. It can be used on a laptop, in a container as well as in distributed environments.

As written in paragraph 2.1.2, we can identify three main areas of the product:

- *Data Visualization*, that is the ability to create visual representations of huge amounts of data with the aim of providing useful, generally easy-to-understand, insights;
- *Data Exploration*, that indicates the process of looking at the data from a new perspective, even considering new metrics that could help users to better understand the data they are working with;
- *Data Analysis*, which consists in using the retrieved information to identify patterns, and make observations and predictions, in order to take informed business decisions.

Dashboards and slices

Dashboards can be considered the user interface through which Apache Superset allows users to communicate to the customers a "story" by combining different types of charts to form a narrative. They are made of different *slices*, which can have the form of data, graphs, numbers, or text.

Two types of dashboards can be identified, *Narrative Dashboard* or *Metrics Dashboard*. The first one is ideal for engaging and explanatory content, and generally, it uses text or content-based data rather than KPIs (Key Progression Indicators) or measurements to provide information. The second is more indicated for performance analysis, monitoring and canned analysis display (complex analysis conveyed in a visual format to the customers with the purpose to deliver meaningful insights) and it generally relies on KPIs, metrics and filters on measurements to show different facets of information. When building dashboards in Superset users can leverage different elements:

- *Tabs*: to group charts for a more structured narrative;
- *Header*: to add titles;
- *Text*: to add description text and even images (via HTML);
- *Divider*: to isolate different parts within the same dashboard;
- *Rows and Columns*: to configure charts layout within the dashboard.

Speaking about data visualization, is worth mentioning the huge amount of available chart types provided by default by Superset.

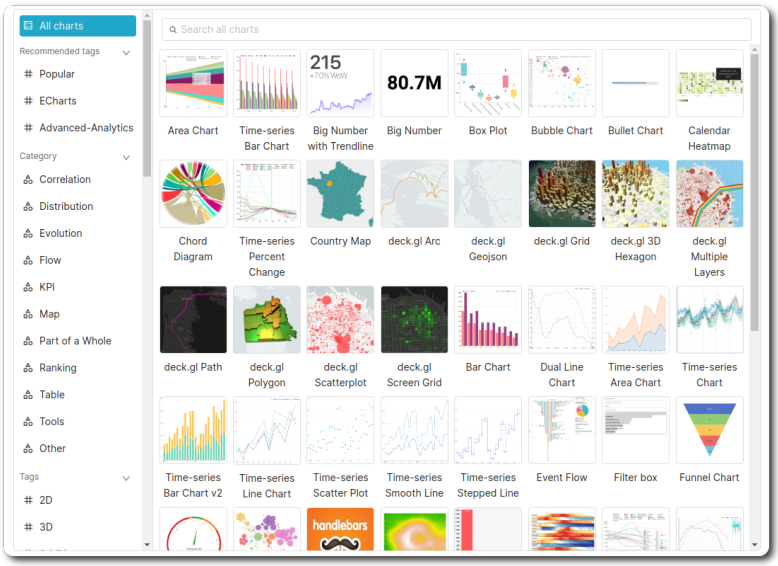

Figure 8.3. Apache Superset available charts

Currently, there are nearly 60 different charts available under 10 categories. This is a crucial strength of the product with respect to the others available on the market.

SQL Lab

SQL Lab is a powerful React-based SQL IDE inside Apache Superset, mainly used for data exploration. It allows users to write their personalized SQL query and eventually visualize the result set via Slices and it works with any database that comes with an SQLAlchemy Python connector. The main feature is called *SQL Editor*, and it is used to explore datasets by composing complex queries that can be saved and later re-used, or even shared with other users.

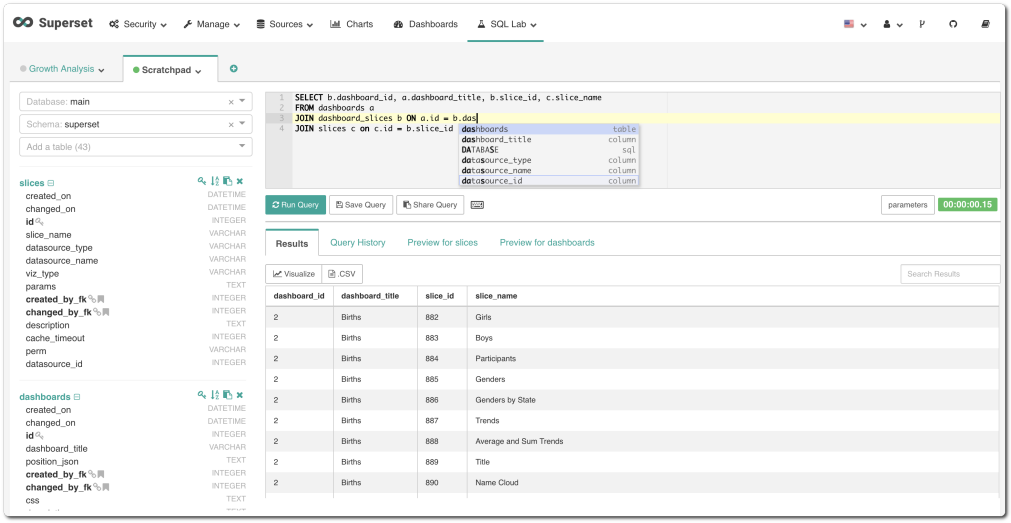

Figure 8.4. Apache Superset SQL Lab

Security

Apache Superset also comes with a Role-based Access Control (RBAC) framework that allows managing access via data access roles and row-level security. Three different roles are available:

- *Admin*: they have all the possible rights, including granting or revoking other users' rights or altering other people's slices and dashboards.
- *Alpha*: they have access to all data sources, but they cannot add or revoke access from other users. Other than that, they are limited to altering only objects that they own.
- *Gamma*: they can only consume data coming from data sources they have been given access to, hence they can only view dashboards or slices made from data sources that they have access to. They are mostly content consumers.

There is also a specific role called *sql_lab* used specifically to grant access to the SQL Lab section of Superset.

Roles are composed of a set of permissions that can be divided into different categories:

- *Model Action*: Dashboards, Slices and Users are considered entities called *models*. Each of them has a fixed set of permissions like *can_edit,*

can_show, can_delete, can_list, can_add. If, for example, we want to allow a user to delete dashboards we could add the *can_delete* permission on the Dashboard entity of a specific role, and then assign that role to the specific user.

- *Views*: views are individual web pages, like the SQL Lab view or the Explore view. When a user is granted view permission, he will be able to see that view in its menu and be able to load that page.

- *Data source*: each data source has a specific permission. Unless a user is provided with the *all_datasource_access* permission, he will only be able to see Slices or explore data sources granted to him.

- *Database*: by providing access to a database to a user, he will be able to access all the data sources within that database and he will also be able to query that database in the SQL Lab (if the SQL Lab permission is granted to the user)

Row Level Security filters are filters assigned to a specific table, as well as a set of roles. For example, if we want members of the Human Resources team to only have access to rows where *department = "HR"* you should create a Row Level Security filter with the clause *department = "HR"* and then assign the clause to the HR role and the to the table it applies to.

Version 2.0 improvements

As of today, 2.0 is the last available release. Released in the summer of 2022, it brings many improvements in both the stability and functionality aspects. Taking into consideration the problems faced in the previous version (listed in paragraph 2.2), we are pleased to see that the problem with non-matching colors after a page refresh has been fixed: now the color assigned to a specific value does not change, avoiding misleading representation that could confuse who is looking at the chart. Together with that, the cross-filtering functionality has been improved and is now fully working: by selecting specific data from a chart (e.g. a "slice" of a pie chart representing movie genres) all the others charts within the same dashboards will be filtered by that value (e.g. genre equals 'Drama'). This represents a huge enhancement in the user experience while navigating a dashboard. Some new charts have been added (e.g. Horizontal Bar Chart), together with some customization options and

the *visualization switcher* has been improved, allowing users to change visualization type (e.g. from a line chart to an area chart) while retaining chosen metrics and dimensions.

8.3 Implementation and results

In this solution, we tried to integrate Cube, with the role of semantic layer, with version 2.0 of Apache Superset. They were both deployed as Docker containers.
Other than all the advantages highlighted in the previous chapters, adopting Cube in this type of scenario increases the overall flexibility. Ideally, once Superset is connected to the data source, this never changes its structure. Unfortunately, this is not what happens in most real-world cases: many upstream changes could occur, being them renamed or deleted columns, dropped tables, the migration of a data source, etc. In Superset terms, all these changes would translate into broken charts and dashboards that would need to be fixed manually. With the addition of Cube, the only thing to do is to update the data model without touching anything else.
Let us take as an example the "renamed or deleted column" scenario, in which a column is removed because it is redundant (it can be obtained by combining with a mathematical expression other two dimensions). One way to solve the problem could be to edit the dataset within Superset, removing the column or removing and then adding it back as a calculated column. This might work, but these types of changes would not be visible from the *SQL Editor* when performing data exploration and, if Superset is not the only tool used in the company, those changes would need to be replicated. Alternatively, we could simply replace the column with a calculation in Cube's data model. By doing so, Superset (and eventually other tools) would keep working as if that column was never removed or edited.
In case, for any reason, the company decides to move data from *Google BigQuery* to, for example, *Amazon Redshift*, we should only replace dataset-specific functions and update the *cube.js* configuration file in order to specify the right drivers for the connection. No other changes are needed to keep Superset dashboards and charts in a healthy state.

Having analyzed some possible use cases, we decided to simulate a scenario with two customers, respectively with their *BigQuery* projects, to see how the multi-user management works within Cube and, finally, we focused our

attention on *pre-aggregations* and their impact on performances.

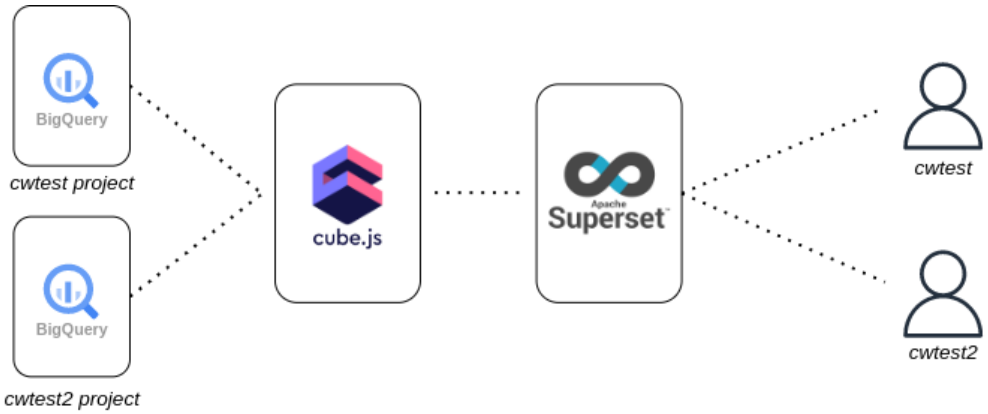

Figure 8.5. Hybrid solution scenario

The first customer, called *cwtest*, was using the Movielens 25M dataset, while the second, called *cwtest2*, was using the Movielens 1M dataset.
As explained before, in terms of access control or authorization, Cube allows the definition of granular access control rules for every cube in the data schema. The additional information about the user that is passed via JWTs is called *security context*. Therefore we configured the *cube.js* file to select the right database connection based on the customer information received; for each connection, we needed a *projectID*, a *key file*, and the *location*. Managing multiple users with Cube was not a complex task, mainly thanks to the fact that the configuration file is a JavaScript file, hence very flexible. Other than controlling access to the data, we experimented with the definition of customer-specific measures and dimensions: this feature comes in handy to answer each customer's needs, providing them with personalized metrics that are more meaningful for their specific business.

Moving on to performances, we decided to configure the *pre-aggregations* for the *Movies* and for the *Ratings* tables. One of the most representative examples of the effects of pre-aggregations is the one that involves a *time_dimension*. For example, in the *Ratings* table we decided to pre-aggregate on the *rating* dimension and on the *timestamp* time dimension, providing *month* as granularity value. As a test, we then used a query to obtain the trend for each rating over the whole time interval at our disposal; for each execution, we provided a different time granularity.
The obtained results can be summarized in the following table.

Query	Time Granularity	Query Completion Time(ms)
1	week	2006
2	week	2
3	month	1537
4	year	23
5	quarter	14

Table 8.1. Pre-aggregations example results

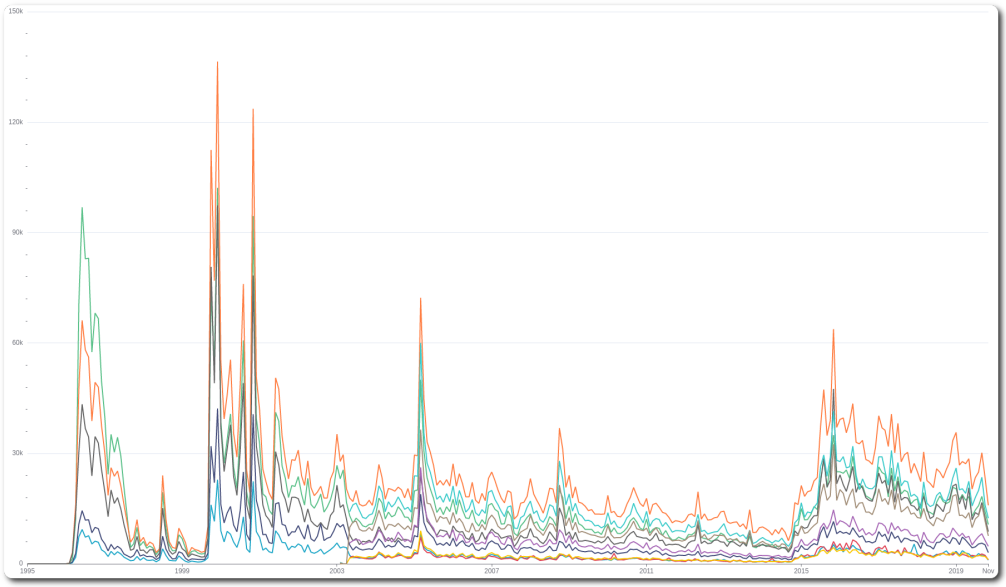

Figure 8.6. Pre-aggregations example chart

When a query is executed, the common Cube workflow is:

1. Check if the exact same query has been made within the time specified in the *refreshKey*. If present, pull the results from the *in-memory cache*.
2. If the query is not present in the *in-memory* cache, analyze the query against the defined set of pre-aggregation rules to choose the optimal one in order to create the pre-aggregation table.
   (a) If found a suitable pre-aggregation rule, check if an up-to-date copy of the pre-aggregation exists. If it does, execute a query against the pre-aggregated data.

3. If no suitable pre-aggregation rule is available, execute the query over the raw data.

Looking at the data from table 8.1, we can notice that the first query took more than two 2 seconds to be executed. On the second execution, the required time drastically decreases. This is because we executed the same query within the *refreshKey* time interval of the *in-memory* cache, so the result was ready-to-use. With the third query, we can see again that over 1.5 seconds are required; this is due to the fact that even if *month* is the selected granularity for the *pre-aggregations*, the first time the query needs to be executed over the raw data. The advantages of *pre-aggregations* are visible with queries 4 and 5. Here we can see that even if no queries with *year* or *quarter* granularity have been executed before, Cube is able to match the *pre-aggregation rule* with *month* granularity and provide an aggregation of results in few milliseconds.
Having made the above observations on the behavior of *pre-aggregations*, is clear that to increase the chances of user queries hitting the pre-aggregations rules we could add more dimensions and more measures to Cube's pre-aggregation definition. This will result in bigger pre-aggregations and, consequently, slightly slower query response time. This is the *trade-off* that users should keep in mind when defining their semantic layer to better suit their needs.

Chapter 9

Conclusions

This thesis presented two data visualization solutions suitable for exploring and analyzing huge datasets. The first one is a custom web application built using the Next.js framework while the second uses version 2.0 of Apache Superset, an open-source tool for data visualization and exploration. In both cases, we integrated Cube as a semantic layer to provide great flexibility in terms of metrics management and performance improvements.
Both solutions come with their advantages and disadvantages. The custom web application allows for more control over the design of the user interface and user experience, as well as the possibility to develop ad hoc functionalities and integrate the product with additional features (e.g. an alerting system for dashboard failures based on Cube's logs). Other than that, the chart library can be expanded and charts can be deeply customized according to the customers' needs. However, it requires more development and maintenance effort. On the other hand, Apache Superset provides a ready-made solution that supports out-of-the-box a wide range of data visualization types, as well as user authentication and role-based access control. Other than that, it has a huge and active community that keeps supporting the product and developing new functionalities; examples of that are the improvements introduced with the last release of the product, in which some key features, such as the *cross filtering*, have been added. But also in this case we have some limitations, especially related to the customization of the product and the development of new features that is bounded to Superset releases. Even if it is true that, being open-source, Superset source code could be modified and adapted to our company's needs in terms of user experience, we must keep in mind that every customization we perform should be then propagated to the future releases of the product. This could lead to a not negligible effort

in terms of maintenance of the solution.

In both cases, the integration of Cube as a semantic layer brings many advantages in terms of *flexibility* and *reliability*. By adopting this separation layer between the data and its visual representation we were able to improve the way we manage metrics and also increase the resilience of our system to future possible changes on the data side.

In conclusion, we observed that the second solution is the most suitable for our company's needs. Being the current version of *Data Insights* already based on Apache Superset, maintaining a similar environment would avoid the need of scheduling training time for the employees. Other than that, we were pleased to notice how some new interesting features, such as charts *drill-down* (starting from a chart, is the ability to navigate from a more general view of the data to a more specific one by interacting with the chart itself), are already in the development road-map of the next Apache Superset release.
Other than improving the current solution, the integration of Cube could open new scenarios for other ContentWise's products: for example, we could introduce a report section inside each customer's profile page of the *UX-Engine* in which, with a few static charts (maybe developed using the Ant Design Chart library) that directly query Cube, convey information about user's most valuable KPIs.

Bibliography

[1] MovieLens - F. Maxwell Harper and Joseph A. Konstan. 2015. The MovieLens Datasets: History and Context. ACM Transactions on Interactive Intelligent Systems (TiiS) 5, 4: 19:1–19:19. https://doi.org/10.1145/2827872

[2] Apache Superset documentation https://superset.apache.org/docs/

[3] Preset documentation https://docs.preset.io/docs/

[4] Cube documentation https://cube.dev/docs/

[5] Firestore documentation https://firebase.google.com/docs/firestore

[6] react-grid-layout https://github.com/react-grid-layout/react-grid-layout

[7] Qlik Sense https://www.qlik.com/us/products/qlik-sense

[8] Qlik AI https://www.qlik.com/us/products/qlik-sense/ai

[9] Semantic layer article on kyligence.io https://kyligence.io/blog/semantic-layer-the-bi-trend-you-dont-want-to-miss-in-2020/

[10] Introduction to the DOM https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction

[11] Semantic layer definition https://www.atscale.com/glossary/semantic-layer/

[12] The Rise of the Semantic Layer: Metrics On-The-Fly - https://airbyte.com/blog/the-rise-of-the-semantic-layer-metrics-on-the-fly

[13] The impact of Business Intelligence on the quality of decision making – a mediation model - Bernhard WiederMaria, Luise Ossimitz

[14] What is Data Profiling? https://www.talend.com/resources/what-is-data-profiling/#:~:text=Data%20profiling%20is%20the%20process,then%20leverage%20to%20their%20advantage

[15] MVC Architecture https://towardsdatascience.com/everything-you-need-to-know-about-mvc-architecture-3c827930b4c1

[16] Cos'è il data mining? https://aws.amazon.com/it/what-is/data-mining/