DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Implementation of a Data Analytics System and Generation of Insights for a B2B SaaS Product

Simon Kreuzer

DEPARTMENT OF INFORMATICS

TECHNICAL UNIVERSITY OF MUNICH

Bachelor's Thesis in Informatics

Implementation of a Data Analytics System and Generation of Insights for a B2B SaaS Product

Implementierung eines Datenanalysesystems und Erstellung von Auswertungen für ein B2B SaaS Produkt

Author:	Simon Kreuzer
Supervisor:	Prof. Dr.-Ing. Pramod Bhatotia
Advisor:	Jonas Hess
Submission Date:	April 15, 2022

I confirm that this bachelor's thesis in informatics is my own work and I have documented all sources and material used.

[illegible]

Munich, April 13, 2022 Simon Kreuzer

Abstract

Every product needs to be as intuitive and simple as possible to use, whilst offering all required functionalities at the same time. For a SaaS product, this especially means to keep an eye on customers, learn from their behavior and improve the product accordingly. Nowadays, such products generate humongous amounts of data. In terms of business intelligence, processing these data sets requires the usage of a big data analytics infrastructure.

In this thesis we establish the data warehouse Google BigQuery together with visualization tools of Google DataStudio for the B2B SaaS product deskbird, and generate informative insights by using this setup. Based on the results of these analyses, it can be determined when to approach customers, talk about possible problems and how to solve them, but it also provides information about the importance of the product's features. Future decision making can then be led by gained knowledge to improve the product and convince customers that this solution is the best on the market.

Contents

Abstract iii

1 Introduction 1

2 Background 3
2.1 The B2B SaaS product deskbird . . . 3
2.1.1 Infrastructure of deskbird . . . 3
2.1.2 Workspace booking . . . 5
2.2 General background . . . 5
2.2.1 Business Intelligence (BI) . . . 5
2.2.2 Big data and analytics . . . 6
2.2.3 Online analytical processing (OLAP) . . . 6
2.2.4 Data warehouse and data lake . . . 7
2.2.5 SQL and NoSQL databases . . . 7
2.3 Firestore database . . . 8

3 Preparation 9
3.1 Objectives and prioritization . . . 9
3.2 Research on analytics setup and infrastructure . . . 10
3.2.1 Data warehouses . . . 11
3.2.2 Visualization . . . 12
3.3 Google BigQuery and DataStudio as solution for deskbird . . . 12
3.3.1 BigQuery . . . 13
3.3.2 DataStudio . . . 13
3.3.3 Google BI Engine . . . 13

4 Implementation 15
4.1 Analytics system setup . . . 15
4.1.1 Data import to Google BigQuery . . . 15
4.1.2 Views for data transforming . . . 17
4.1.3 Changes to the Firestore database . . . 18
4.2 Development and interpretation of analytical insights . . . 20
4.2.1 Historical overview of users and bookings . . . 20
4.2.2 Same desk booking . . . 25
4.2.3 Length of bookings . . . 28

5 Findings **31**
5.1 Customer approaching 31
5.2 Feature improvement 32

6 Conclusion and future work **35**

Appendix **37**
A.1 Firebase views 37
A.2 Views for JSON reprocessing 38
A.3 Full queries of the implementations 40

List of Figures **45**

List of Listings **47**

Bibliography **49**

1 Introduction

Every company wants its product to be as intuitive and simple as possible to use, whilst offering all required functionalities at the same time. In order to develop a successful product, a good and innovative idea is needed that both impresses and converts users into customers. However, requirements from customers are evolving and so are products from competitors. How can companies keep up with these changes and convince that their solution is the best?

Most important for improving a product is the direct communication with customers. Do they like or dislike the product? Why did they choose this solution? Do they have any suggestions for improvements? Simply talking will not be sufficient though. How would you know what the average customer does with the product? Do they know about all features you have implemented? You cannot tell which functionality is the most important by just asking a few people. Instead keeping an eye on all your customers can provide valuable information about the average user and reveal informative and unexpected results.

Humongous amounts of data are generated by using a product, especially when it is software as a service (SaaS). The type and format of such data sets can vary depending on the product. However, as today's technology offers great storage opportunities, most companies and products tend to store absolutely everything which results in a data lake [1].

This is the point where big data comes into play. Its goal is to reconcile all data sources, connect them and analyze the results to support decision making and improve the product. Each data source stores in a different manner. Some of them contain structured, others unstructured information. In order to bring all of them together and work with them, transforming through ETL tools (extracting, transforming, loading) and a big data warehouse are needed. These warehouses are made for huge data sets and can run complex queries on them in incredible short time which allows to convert the data lake into valuable information.

For the purpose of the underlying thesis, we make use of big data and cloud computing, two big trends of analytical tools [2], to support the business-to-business (B2B) software as a service (SaaS) product *deskbird* in decision making. Particularly, we set up a data analytics system for deskbird and develop a few analyses to get a first insight in general user behavior. As a base, we first clarify important terms and structures and how deskbird's products work in chapter 2. Subsequently, we start with research on different data warehouses and visualization tools in chapter 3. In order to achieve the best outcome, it is important to precisely inspect what is on the market and fits best to the requirements. As deskbird heavily relies on the Google Cloud infrastructure, Google

BigQuery allows us to start very fast and easily by providing seamless integrations for their other services. Using this data warehouse, we develop analytical SQL queries and visualize the results in Google DataStudio (see chapter 4). In chapter 5 we study the produced visual insights and discuss how they can support deskbird in understanding their customers' behavior and reacting to potential problems. Finally, we sum up the thesis' results in chapter 6 and give a short outlook to future work and further improvements.

2 Background

2.1 The B2B SaaS product deskbird

Companies become more hybrid and let their employees decide whether they come to the office or work from home. Especially due to the COVID-19 pandemic, there is a big change in how employees can and have to work. Such companies are targeted by *deskbird*[1] which offers them a multi-platform software solution to manage their workplaces.

The startup was founded in August 2019 and started with a solution to book desks and meeting rooms at coworking places. This feature especially targets small and hybrid companies or freelancers that do not have a fixed place to work at. Clients could look for coworking spaces nearby and directly claim a desk with their phone.

However, another product displaced the original idea of this startup: *Workspace Booking*. In July 2021 deskbird published a solution for companies to simplify their workplace management. Employees can book desks, meeting rooms and parking slots. This product especially benefits companies with a higher home office policy and employees that are not in office every day. The COVID-19 pandemic also contributed to the fact that more and more companies switch to hybrid working and want a simple opportunity to manage their workplaces. Accordingly, deskbird can report strong growth in the last months.

Furthermore, the startup came up with another product in February 2022: *Hybrid scheduling*. It allows employees to schedule their week and see who else is in the office. They can select one of a few statuses to define whether they are in office, at home, absent or on business travel.

Both workspace booking and hybrid scheduling are integrated in the same application, but customers can separately book each feature on its own or as a bundle. Employees can make their booking and scheduling with the mobile app on Android or iOS, but also using the browser. The startup is also working on integrations to make the usage of their products as comfortable as possible, accordingly you can already connect Slack and Microsoft Teams.

2.1.1 Infrastructure of deskbird

Every company has its own data infrastructure that evolves with the company itself. In order to find the best solution for the analytics setup we need to be aware of this

[1]`https://www.deskbird.com/`

infrastructure and its strong and weak points. Accordingly also deskbird has developed its individual setup, consisting of various clients and a backend (see Figure 2.1).

There are three client applications for the normal employee: Android, iOS and a web app. On all of these you can make bookings, schedule your next week and make changes to your profile. Both the Android and the iOS application are natively built on their platform, the web-app is currently best to use on a laptop or computer. Additionally, there is an admin portal for heads and workspace managers of the customer companies to change offices and its properties and also have an overview about the employees. Latter is less important for our analyses and this thesis as it only addresses a minor part of the users, but will be from relevance in later development.

All of these clients highly communicate with the backend. Almost nothing is actually stored on the client applications itself, but retrieved from the backend when needed. In conclusion, the backend is an important component and includes two different databases: The Firestore database [3] (see section 2.3) and a PostgreSQL database [4] which are both stored in the Google Cloud environment.

The Firestore database currently contains most of the backend data. As it is a NoSQL database like discussed in subsection 2.2.5, it is quite simple to extend during the process of development. New fields and attributes can be easily added without disturbing or crashing any other processes. In addition Firestore's real-time synchronization and offline support can be quite helpful for developing multi-platform applications [3]. For startups like deskbird that still evolve strongly the Firestore database is simple to use and an optimal solution.

However, for the reason that data gets more complex and at some point deskbird also needs to do more complex queries, the PostgreSQL database was introduced. It basically holds information about the desks and its occupation. Querying those needs on the Firestore database would have lasted way too long as it does not offer any analytical functions like SQL does. Instead data would have being extracted and analyzed using other programming languages and interfaces. We do not need that database in this thesis for our analyses, but we still have a short look at how we can integrate and use it in subsubsection 4.1.1.

Apart from that infrastructure of the actual product, there are also other components that can be monitored and analyzed. The homepage from deskbird would be one of

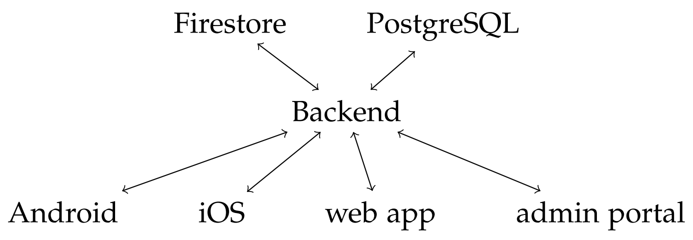


Figure 2.1: Infrastructure of deskbird

these, but also the information from advertisements can be used. In this thesis we focus on the main product of deskbird, further analyses will be done at some other point in the future.

2.1.2 Workspace booking

The feature *workspace booking* is currently the main product of deskbird and already provides us with a lot of data. For this reason we focus on this part of the application in our insight development in section 4.2 and have a short look at the functionality.

The main idea of this feature is that employees can book their workplace in the offices of their company. Thereby the workplace can currently be either an individual desk, a meeting room or even a parking slot. Users can claim the required resource as needed and also see who will sit around them.

A company is structured as can be seen in Figure 2.2. Each customer corresponds to one company that can have multiple offices, each office can have multiple floors with multiple areas, and finally, each area has multiple workplaces. This hierarchy also meets the structure of the Firestore and PostgreSQL database, while floors can be found in the `groups` and areas in the `zones` table.

2.2 General background

In this section we have a look at general background information about business intelligence, big data, databases and analytics. This helps to understand how and why data is structured as is at deskbird and how we can get the best and most efficient outcome of our implementations in chapter 4.

2.2.1 Business Intelligence (BI)

Decision making more and more relies on computer support. Those huge amounts of data being produced by applications, advertisements and communication cannot be processed manually any longer. The term *Business Intelligence* (BI) therefore describes the IT-based decision support, including several tools and organizational activities [5].

Business intelligence supports in translating data into information with the main goal to support future decision making and combines data collection, data storage, and curriculum management. This extracted information can hold details about sales trends, user behavior and resource allocation. By using those insights, the product can be optimized and adjusted to the users' requirements, and future expansion is encouraged while decreasing the costs at the same time [6].

$$\begin{tikzcd}\text{company} \arrow[r] & \text{office} \arrow[r] & \text{floor} \arrow[r] & \text{area} \arrow[r] & \text{workplace}\end{tikzcd}$$

Figure 2.2: Structure of companies in deskbird

2.2.2 Big data and analytics

With the increased usage of computers and the internet the amount of data being stored increases rapidly. A huge load of different information is saved in order to improve the users' experience and make systems smarter. The necessity of more and more detailed information pushes the size of data. Such data is usually referred to as *big data*.

Big data is often defined by the so called V's. These were originally *volume*, *velocity* and *variety*, but later other definitions and more V's came up. The most common used 5 Vs are the following [2]:

- *Volume*: The amount of data is huge. Large companies store data sets that we cannot even imagine. Every second new information is being generated and stored, caused by e.g. videos, images, audio, messages, etc.
- *Velocity*: Data needs to be processed fast. The challenge to do this increases with the amount of data. Thinking about e.g. YouTube, that allows us to upload and watch videos in great quality and at any time in high speed.
- *Variety*: We want to process different types of data from various sources. This also produces a lot of data-types that traditional database management tools and applications cannot handle.
- *Veracity*: The stored data should always be useful, meaning that truthfulness and reliability is given. This is mandatory for exact and accurate analyses.
- *Value*: This measures the worth in information that we receive by processing the huge data sets.

There is still a lot of controversy and confusion about the V's, R. Patgiri and A. Ahmed came up with a whole V family, abbreviated as $V_3^{11} + C$ [7]. V_3 stands for *voluminosity*, *vacuum* and *vitality of volume*, V^{11} denotes all other V's and C describes *complexity*.

Apart from the challenge of enough storage space for the big data sets, another one is to process the mass of information and extract even more details as a base for future developments and decisions. This is where analytics comes in. Its goal is to make huge data sets useful and helpful. This can lead to interesting and surprising statistics, as well as insights of a product and its usage, as we see in chapter 5.

2.2.3 Online analytical processing (OLAP)

Online analytical processing (OLAP) [8] systems are the core for analytics and aim to support in decision making. Its goal is to extract and process information from a data warehouse with the main advantage that it is usable by non-expert users. According to Nigel Pendse, an OLAP tool should pass the FASMI (fast analysis of shared multidimensional information) test [9]. This means, they need to allow fast and interactive queries that can also be handled by every user, help in analysis tasks, allow sharing data while caring about security, and of course support large data sets.

2.2.4 Data warehouse and data lake

A data warehouse is a collection of different data from various heterogeneous sources with the ability to connect and analyze the information. It is primarily used for strategic decision making using OLAP techniques [10] (see subsection 2.2.3). The central and structured storage of different data sets eases the further processing for every user with access to the data warehouse.

Apart from that, data warehouses are mainly made for conventional and structured data sources from SQL databases. The introduction of big data, which also includes unstructured data from various sources to a large extend of volume, came with the concept of the *data lake* [11]. This means that there is absolutely everything of information and data stored that emerges from maintaining and usage of a product [1]. However, for further processing and analyses on the data, they have to be prepared first. The unstructured data has to be transformed to structured data in order to connect them and do complex analytical processing.

2.2.5 SQL and NoSQL databases

It is always important to store your data in a manner that fits best to your requirements. If you want to do a lot of calculations for example, you might prefer another type of database than you would for just storing a lot of unstructured data. The right choice will help you to be more efficient and achieve your goals.

Traditionally, databases have been based on the relational model, also known as *SQL* databases. Over time new alternatives were introduced and *NoSQL* (Not only SQL) databases gained more attention and are already used quite widely. Other than SQL databases, they are based on non-relational models, most of them store simple key-value pairs, meaning that data is saved as a key or attribute name with its value. The idea of doing so is that simplicity leads to increased speed [12].

When comparing SQL and NoSQL databases with each other, we can clearly see their preferred usage. NoSQL databases are made for distributed storage and dynamic data formats as often needed in today's applications. Accordingly, they are a good base for big data. In comparison, SQL databases are better for complex queries or for storing data in which no changes of the storage format or schema are expected. With SQL queries you can do operations like joins, aggregations, etc., so this is the preferred choice for OLAP systems (see subsection 2.2.3) [13].

When thinking about big data (see subsection 2.2.2) NoSQL seems quite promising and it also became the preferred database type in this field. That is why there are now a lot of different NoSQL databases such as Google BigTable [14], but also open-source systems as MongoDB [15] or CouchDB [16]. For this thesis Google's NoSQL database *Firestore* (see section 2.3) is of importance as deskbird uses it for storing the most of its data as described in subsection 2.1.1.

2.3 Firestore database

The Firestore database [3] is a NoSQL database (see subsection 2.2.5). Apart from being a high performant and scalable database, Firestore especially aims at ease of application development. It is cloud-hosted and offers native SDKs for Android, iOS, web apps and other programming languages. This makes it very easy to develop a distributed multi-platform application and connect all devices with each other.

Additionally, Firestore provides real-time updates to all connected devices and even offline support. So the applications can be used at every time, changes are pushed as soon as the device is back online. Together with that it supports seamless integrations with other Firebase and Google Cloud products, making it an comfortable choice when using Google BigQuery as data warehouse for analytics.

Firestore stores data in collections, sub-collections and documents. The collections are comparable to tables of a traditional SQL database and the documents to rows. A document does not follow a strict schema and can contain any possible data. Collections can also contain sub-collections if necessary which allows to create complex hierarchies of collections. All in all this structure is quite flexible and allows to change data schema during the development process without affecting any old documents.

3 Preparation

In order to start developing and implementing analyses, a few decisions have to be made first. The main goal is to support the company with its future development, so we have to think about different aspects. Firstly, we have to take a look at the requirements of the company. What kind of analyses shall be made and how can their results help? Based on this and the infrastructure of the company, we can choose the tools like data warehouses and visualization tools to work with.

3.1 Objectives and prioritization

The most important preparation for an informative and helpful outcome of an analytics environment is to collect relevant topics and think about how they can be supportive. The results can vary a lot among different companies and what you set the focus on. Subsequently, you can make further decisions regarding the general setup and the implementation.

In order to get information about the desires and requirements of deskbird and its employees, we did a brainstorming with all colleagues and collected anything that came in mind. Some put down random ideas, but most of them focused on what would be good to know for their particular work. Finally, we had a long list of diverse topics, including quite general and also very specific ones.

When looking at the list, we can easily figure out three categories: *User behavior*, *error tracking* and *performance*. While error tracking and performance monitoring emits valuable information for the developers about problems in their code, user behavior tells us a lot about the customers and how they use the product. Latter helps for example the design team to draft or edit layouts and click-flows of specific parts of the application that can improve the users' satisfaction. We declared user behavior as the most important category, so this thesis focuses on this.

The users' app usage can be tracked either on backend or client side, both giving different information though. Former can give us details about the general user behavior like the amount of active users, the number of bookings or the number of offices per company. Advantageous is that we only have one single data source in this case. On the other side the client tracking can provide us with more personal data like the number of clicks users need to make their booking, but this also needs us to implement it on all client platforms separately.

While both backend and client side tracking give us valuable information, we decided to prioritize former and get as much general company and user information from that as possible. Any further monitoring on the clients will be done at some other point in the future. For this reason, this thesis focuses on analyses on the backend data.

The following topics are the most prioritized ones of this process and are also be focused on in the implementation in chapter 4:

- *How many users are booking actively?* Do all of the registered customers use the product and make bookings? Could they have a special position in their company, so they do not need to book anything because of their own private room? If the rate of active users is quite low, you can talk to them and find out the reason. How many bookings are made per user? How does this change over time?
- *How many users book the same desk over and over again?* deskbird aims to create a hybrid workplace, so you do not have your own personal desk, but dynamically claim the place you need. In contrast to that, humans are known as creatures of habit. This rises the question whether users just randomly book desks or choose the same specific one they like most and are used to?
- *What is the usual booking length?* Do people like to book the full day or just one or two hours? This can also vary depending on the type of place that was booked (desk, meeting room, . . .) and the policy of the company.

3.2 Research on analytics setup and infrastructure

Every company and application has its own infrastructure with structured and unstructured data stored in one or multiple data sets in different locations. Accordingly, we need to have a look at deskbird and possible data sources first in order to setup a data analytics system that fits best. The infrastructure of deskbird is described in subsection 2.1.1.

Most of the professional BI-tools support various data sources and combine them when using the graphical user interface. However, they mostly cost a lot and cannot really do more than you could do by yourself with creating smart queries on your data sets. Especially for just a few sources at smaller companies like a startup and deskbird, a typical BI-tool is not mandatory. Instead, we gather our data by ourselves, query them and visualize the results.

In order to do that, we basically need three components for our setup:

- *Data warehouse*: We import our data from various sources to one big data warehouse. This has the advantage that we do not access the data sets directly, but work with a copy of it. Additionally, we have all the data in one place which makes connecting and querying the data a lot easier.
- *ETL tool*: Those tools are for extracting, transforming and loading data. In terms of data analytics and our setup, this means more or less importing the data from various sources to our warehouse. ETL tools are important because data is not

necessarily in a suitable format for the warehouse. We need to adjust and extract the relevant data [17].

- *Visualization tool*: Querying the data from the warehouse would theoretically give us all information we need. Though a good visualization helps to see important things faster and have a better comparison among the results.

3.2.1 Data warehouses

Data warehouses are the core for analyses on big data sets. They store information from various sources through ETL tools and make them accessible in multidimensional form with the main goal to generate analyses supporting the decision making of companies [18].

As they already exist for quite some time (official year of birth is considered as 1992 [18]), there have developed numerous and various data warehouses. This lets us choose one solution that fits best to our requirements.

One important consideration is the choice between a self-hosted and a hosted warehouse. While the hosted variant minimizes the effort for setting up, the self-hosted one lets us easily move to new servers if needed because of e.g. privacy requirements. This was also a big question for deskbird as a marriage with one solution is not ideal. Having the opportunity to change the provider seemed attractive when self-hosting the warehouse. That is why we have a look at both variants in the following.

Cloud based data warehouses

Cloud technology in general enables remote processing and data storing. This offers many advantages compared to local data processing and storing as the customers no longer need to have their own instance of the data and software [19]. Instead, it allows us to store the data and run queries directly in the cloud. Data warehouses with cloud architecture differ a lot from traditional architectures, each offering even has its own unique one [20].

Hosted data warehouses

Let us have a look at hosted warehouses such as *Google BigQuery* [21] or *Amazon Redshift* [22]. They usually are also cloud based with the advantage of a very simple setup. Mostly they have a lot of integrations that can be used which makes connecting data sources easy and replaces ETL-tools. In addition, you have a direct contact if you need help or something is not working as expected.

So this seems like a pretty solid solution if your knowledge about warehouses, data clusters and analytics is limited. On the other hand, you commit to one solution. If your requirements change over time or the provider does changes that do not fit to your requirements, you need to move your warehouse which can come with a lot of effort.

Open source and self-hosted data warehouses

Additionally, we have a look at self-hosted warehouses which mostly come with open source. This means we either host the warehouse on our own hardware or let it run on a cloud service. Both is related to a more complex setup as you need to have detailed knowledge about their infrastructure and data clustering.

The *Apache Hadoop framework* [23] is the base of many data warehouses and allows you to process large data sets distributed across a computer cluster. It is highly scalable and fault-tolerant which makes this framework quite interesting for data analytics and helps a lot of companies to convert their low-value data into high-value information for e.g. business intelligence (see subsection 2.2.1). Although Hadoop is highly available and good in storing data, it also comes with weaknesses. It especially suffers under heavy concurrent load and is not optimized for analyzing data and making it immediately readable [24].

Another large scale data warehouse is *Greenplum* [25]. It mainly benefits from its massively parallel processing architecture. Their users interact with a coordinator node which optimizes the queries for parallel processing and lets worker segments execute its plan in parallel. In the end the coordinator collects the outcome and hands it over to the user.

One last popular open source data warehouse should not be forgotten: *Druid* [24]. It is designed for real-time exploratory analytics on large data sets and convinces with its great speed on aggregations, flexible filters and low latency data ingestion while querying billions of rows.

3.2.2 Visualization

By using a data warehouse we have all information in one place and can already execute queries to generate further insights. However, results would always be shown in boring tables and mostly don't offer a good comparison and overview of the content. This is where visualization tools come into play. They offer a bandwidth of tools like graphs and tables to colorize and draw the results of data extraction which strongly helps man to understand and compare information.

3.3 Google BigQuery and DataStudio as solution for deskbird

The decision for deskbird was simple and difficult at the same time. On the one hand, we want to avoid a marriage with one specific solution, on the other hand Google BigQuery is quite simple to maintain and fits perfectly to deskbird's infrastructure (see subsection 2.1.1). Having almost no effort for the warehouse's setup finally dropped the decision. There is even an extension by Google for the Firestore database (see section 2.3) that allows to import and stream all data to BigQuery in real time, so an ETL-tool is not necessary. In case we actually need to switch the warehouse for reasons like data privacy, this won't be a big deal, queries and data handling logics can simply be transferred.

There exist plenty visualization tools that support BigQuery as a core warehouse. Additionally, Google offers its own one, called *DataStudio* [26]. Again, keeping the data in the same environment, the Google Cloud, simplifies a lot and lets us start with DataStudio.

3.3.1 BigQuery

One of the most popular cloud data warehouses is BigQuery [21]. It can be accessed through different ways but the most important one for our implementations is the web interface. Here we get a good overview over existing tables and views, and can directly run self-written SQL queries. For this reason this is be the place where we develop our queries to extract information from our data lake.

3.3.2 DataStudio

There are many BI and visualization tools out there, most of them very powerful and complex. Google DataStudio [26] limits to simple visualizations, but this is sufficient for our requirements. It is a user-friendly visualization tool for representing complex data sets and is free for everyone to use.

DataStudio integrates BigQuery natively and lets us import our SQL queries and views as data sources. Additionally, we can add extra metrics and compare the output of BigQuery. Especially advantageous is that these metrics can be added by every user with access without any knowledge of SQL and databases. Subsequently, we can create a report, add the required data source and start visualizing with different tools.

3.3.3 Google BI Engine

Although BigQuery is already very fast in running complex queries, they still can last some time on very big data sets. For this reason, Google developed the *BI Engine* [27, BI Engine] to accelerate queries even further. It basically is a fast, in-memory analysis service which aims to analyze stored data in BigQuery and query in sub-seconds.

However, there are limits and also a reason why we cannot use the BI Engine in our implementations. It already supports the majority of functions, aggregations and SQL commands BigQuery offers. Though there are exceptions, so it especially does not support user defined and analytic functions. Latter is essential for the views in Listing A.1 where we extract the latest version of each entry, and this view is mandatory for further processing.

4 Implementation

In this chapter we want to implement our analytics system and develop some of the most relevant and interesting insights for deskbird. This includes preparation like the data import and restructure, as well as the development of SQL queries for analyses with subsequent visualization. Google BigQuery offers, apart from a legacy SQL dialect, its own *Google Standard SQL dialect*, which we use in this thesis as it supports the broadest range of functionality. Accordingly, all upcoming SQL code examples are written in this SQL dialect.

4.1 Analytics system setup

The Firestore database is a NoSQL database, which means it is quite unsuited for running complex analytical queries (see section 2.3). For this reason, we chose Google BigQuery as data warehouse like discussed in section 3.3. In this section we import our data to BigQuery and prepare it for further development of insights.

4.1.1 Data import to Google BigQuery

Google BigQuery is the data warehouse of our choice, but data is not stored there yet. In order to start with our queries and analyses, we need to mirror our data sets to BigQuery from the Firestore and the PostgreSQL database. Mirroring the data has also the advantage that we do not need to execute any queries on the original database, so no unintended changes can happen. Though we need to be aware that the data is always up to date.

Import Firestore database to BigQuery and setup stream extension

The collections from the Firestore database can be easily streamed to a dataset in BigQuery using an extension called *Stream Collections to BigQuery* [28]. This extension is provided by Firebase itself and lets us stream any changes from our Firestore database to BigQuery in real time and incrementally.

For each necessary collection in the Firestore database we need to create a separate stream extension. Every extension is configured with the project id `deskbird-bbe72`, the collection name to be streamed, the dataset id `firestore_mirror` and a table prefix which we always set to the exact same name as the collection to avoid confusion.

Looking at the BigQuery dataset we can find out that this extension only streams changes of the collection, but does not do an initial import of data. Therefore we have to do this manually. Luckily, the extension also offers a script[1] for importing all existing data from Firestore. We just have to open the CLI of the Google Cloud Shell and execute this command:

```
npx @firebaseextensions/fs-bq-import-collection
```

Subsequently, we are prompted to enter the same information as for the extension in order to do the import. When done, we can check the BigQuery table again and can now see the imported rows.

The created table from the import and stream is called `collection_raw_changelog`, where collection has to be replaced with the actual collection's name, and consists of metadata columns as well as a data column which holds the whole document from Firestore in JSON-format. This metadata is provided in five columns called `timestamp`, `event_id`, `document_name`, `operation` and `document_id`. For reproducing the edit history of documents the `operation` column is of relevance. It can contain the values `IMPORT` for the initial import with the mentioned script, `CREATE` for a new document, `UPDATE` for a change in the document and `DELETE` if a document was deleted.

The stream extension also sets up a view called `collection_raw_latest` for each collection table that shows only the latest entry of each document. An insight of how this view looks like can be found in Listing A.1.

Connection of PostgreSQL to BigQuery

The PostgreSQL database is hosted in the Google Cloud, which, again, makes things easier. As the PostgreSQL database contains only a small part of the whole data and also this part is not needed with a change history, we can completely avoid mirroring the data to BigQuery. Instead, we use the ability of BigQuery to do federated queries on other SQL databases in the Google Cloud. This allows us, after a one-time set up, to directly query data in PostgreSQL in real time, without copying or moving data, and use the results in BigQuery itself [27, Cloud SQL federated queries].

```sql
SELECT
  *
FROM
  EXTERNAL_QUERY('connection_id',
    '''SELECT * FROM table''')
```

Listing 4.1: Example of a BigQuery federated query

This feature is not used in the implemented analyses of this thesis. However, it will be needed in the future when extending, improving and continuing with further insights.

[1]A documentation of the import script can be found at `https://github.com/firebase/extensions/blob/master/Firestore-bigquery-export/guides/IMPORT_EXISTING_DOCUMENTS.md`

4.1.2 Views for data transforming

Each document from the Firestore database is now stored as a JSON-string and metadata in BigQuery (see subsubsection 4.1.1). Executing queries on a JSON-string is not possible though. For this reason, BigQuery offers JSON-methods to parse this string into separate columns so we can do complex queries on them [27, JSON functions]. Extracting the id from the raw bookings table then would look like this.

```sql
SELECT
  JSON_VALUE(DATA, '$.id') AS id
FROM
  `deskbird-bbe72.firestore_mirror.bookings_raw_latest`
```

Listing 4.2: Extraction of values in JSON-string in BigQuery

However, this method `JSON_VALUE(...)` always returns a string of the corresponding field. This makes comparing some types like boolean or numbers quite complicated and requires us to cast the types from the string to its actual type. The following types had to be casted for our use cases with `json` as the string representation we got from `JSON_VALUE(...)` before:

- *Boolean* can be parsed through

  ```sql
  SAFE_CAST(json AS BOOLEAN)
  ```

- *Numbers* can be parsed with

  ```sql
  SAFE_CAST(json AS NUMERIC)
  ```

- *Timestamps* in Firestore are stored in Unix timestamps in milliseconds. In order to transfer store those in BigQuery's `TIMESTAMP` format, we use this nested function call.

  ```sql
  TIMESTAMP_MILLIS(SAFE_CAST(json AS INT64))
  ```

- *Coordinates* in Firestore are stored as the datatype `geopoint`. The extension (see subsection 4.1.1) saves the coordinates in the JSON representation as `{"_latitude":...,"_longitude":...}`. BigQuery has the datatype `GEOGRAPHY` for coordinates. In order to parse to this type we need to execute the following functions.

```
ST_GEOGPOINT(
  SAFE_CAST(JSON_EXTRACT(json, '$._longitude') AS NUMERIC),
  SAFE_CAST(JSON_EXTRACT(json, '$._latitude') AS NUMERIC))
```

The most efficient way for creating the queries on those views would be to insert a view for each query with only the extracted fields you need for it. However, this also comes with a lot of effort. Instead, one view per table does its job and also allows everybody to query on it without any knowledge about the JSON representation and how to use it.

Those views are for extracting the JSON string only, though there are cases where we want to filter straight away. The cases are

- Only active users are from interest for our analyses. All inactive ones are ignored.
- Only bookings of the current used feature *workplace booking* contain valuable information, bookings on the old deleted *coworking booking* are ignored.
- In order to do a proper demonstration of the product there are dummy bookings added. These are marked as demo bookings and can also be ignored.

A list of all views can be found in section A.2.

4.1.3 Changes to the Firestore database

Transferring data from a NoSQL to an SQL database comes with challenges. NoSQL databases do not necessarily have a strict data format, so development over time leaves redundant data fields in old documents but also brings new ones without adding those fields to old data. This makes data inconsistent and also difficult to use in complex queries. During the development of the SQL queries, inconsistent data were detected and it was mandatory to do changes to the Firestore database before finalizing any analysis. Not all of the following changes have a direct impact to our implementations in section 4.2, but they are still relevant for future analyses.

Retrofit of partly missing field `companyId`

The field `companyId` in the documents of the bookings collection connects each booking with a company. Although this association could also be done by looking up the user that made the booking and its corresponding company, this field was added retroactively where missing. Through this change some queries became less complicated.

Rectification of companies' status

Caused by a lot of demos and trial accounts for customers, there exist companies in the database that are not actually used and should not be included in most of our analyses. There already is a field in the documents of the companies collection called `status`, though it is not used accurately. However, filtering on this status improves the informative value of our analyses. For this reason, all statuses of the companies were manually adjusted, some of the companies were even completely deleted.

Connection of the new and old version of an edited booking

When a booking is edited, this booking is left as is and gets the field `bookingStatus` with the value `editedByUser`. Additionally, a new booking is created which contains the edited information. Having a history of changes is helpful when we want to have a look at what users edit specifically. However, we need to know which was the old version of the booking, which cannot be easily monitored yet.

Accordingly, a new field `previousBookingId` was introduced so that edited bookings can refer to its pre-edit version. For future edits this field is good to use, but it would be even better if we can add this field retroactively to old bookings. In order to do so, we need a list of edited bookings and find their previous versions.

Luckily, each booking has an exact timestamp of its creation and last change. So if we can find two bookings from the same user, one of them with the status `editedByUser` and a difference of less than five seconds between the creation time of the new booking and the last change time of the old booking with the mentioned status, this would be a good indicator that these two bookings are related to each other.

As we have the BigQuery database already set up we can easily query these information. Therefore we use an unfiltered view `bookings` which parses necessary fields from the JSON string. The output of this query then can be used in a script to add this field retroactively.

```sql
SELECT
  newBookings.id AS newBookingId,
  oldbookings.id AS oldBookingId
FROM
  bookings AS newBookings,
  bookings AS oldBookings
WHERE
  newBookings.userId = oldBookings.userId
  AND oldbookings.bookingStatus = "editedByUser"
  AND ABS(TIMESTAMP_DIFF(newBookings.creationTime, oldBookings.updateTime,
       SECOND)) <= 5
```

Listing 4.3: Connection of the previous and the current version of edited bookings

4.2 Development and interpretation of analytical insights

At this point we have all of our data sources connected to our data warehouse BigQuery and set up views to bring the data lake into a structured form. This means, that we do not have to worry about the original data structure anymore and can start with developing SQL queries and visualizing the results. In this section we have a look at the most important and interesting topics that help deskbird in decision making.

4.2.1 Historical overview of users and bookings

As deskbird's application is all about the users and their bookings, we first have a look at both in general. This gives us a great overview about all customers and how intensively they use the app. If necessary, deskbird then can approach them, talk about problems and issues and improve their product accordingly.

In particular we want to see a history for both per month and per day of each company with the following information:

- *Amount of users*: The amount of registered users at a specific date. This number can differ slightly from the number of contracted people as they have to create an account by themselves.
- *Amount of active users*: Not all registered users are booking actively. This can have various reasons such as users being on vacation or sickness. Active users are defined as those who have at least one booking in the given time range.
- *Amount of bookings*: The absolute amount of bookings that were made in the given time range.

Additionally, we can set this information into a relationship and create two further metrics: The *amount of bookings per user* and the *amount of bookings per active user*. This gives us a good insight in how actively companies and their employees use the application.

Implementation

In the first step we want to calculate the amount of newly registered users per company and month. Therefore we have a look at the creation timestamp of each user and assign them to the corresponding month. This can be done by extracting the `DATE` from the `TIMESTAMP` and truncate it to its month with `DATE_TRUNC(...)`. The amount of users can then be counted per company and month by grouping. Accordingly, the SQL query would look like this:

```sql
SELECT
  companyId,
  DATE_TRUNC(EXTRACT(DATE FROM creationTime), MONTH) AS month,
  COUNT(DISTINCT id) AS numberUsers
FROM
  ‘deskbird-bbe72.firestore_mirror.users‘
GROUP BY
  companyId,
  month
```

Listing 4.4: Amount of users per company and month

Secondly, we count the amount of active users and bookings per company and month. In order to achieve that information we proceed similar as for the amount of users. We truncate the creation timestamp to its month and the amount of active users can be counted by looking at the distinct `userIds` in the bookings table. The final SQL query then looks like this:

```
SELECT
  companyId,
  DATE_TRUNC(EXTRACT(DATE FROM creationTime), MONTH) AS month,
  COUNT(DISTINCT userId) AS numberActiveUsers,
  COUNT(DISTINCT id) AS numberBookings
FROM
  ‘deskbird-bbe72.firestore_mirror.bookings‘
GROUP BY
  companyId,
  month
```

Listing 4.5: Amount of active users and bookings per company and month

In the final result we want to see all companies with name and status, and statistics for each month they exist in the databases. Therefore we cross join the companies with an array of `DATES`, each representing a month, beginning from the creation month of the corresponding company. With this we can guarantee that no more and no less months since each company is existing in the databases are covered by our query.

```
SELECT
  companies.id,
  companies.name,
  companies.status,
  month
FROM
  ‘deskbird-bbe72.firestore_mirror.businessCompanies‘ companies,
  UNNEST(GENERATE_DATE_ARRAY(
    DATE_TRUNC(EXTRACT(DATE FROM companies.creationTime), MONTH),
    CURRENT_DATE(),
    INTERVAL 1 MONTH
  )) month
```

Listing 4.6: Cross join of companies with month array

Additionally, we want to make the number of users cumulative, thus we have an amount of the total registered users of each month instead of only the ones that newly registered. This can be done by building the sum over a window.

```
SUM(numberUsers) OVER (
  PARTITION BY
    companies.id
  ORDER BY
    month ASC)
```

Listing 4.7: Cumulative sum up of amount of users

Finally we can left join our results from before (see Listing 4.4 and Listing 4.5) on the `companyId` and `month`. We chose a left join here because we want to ensure that every company is listed, even if it does not have any bookings or users yet. In those cases this join would produce `null` values, but instead we want the number 0 for all the amounts. Replacing of the `null` values is possible with `IFNULL(value, 0)`. Additionally, we have to filter for companies that have the feature workspace booking enabled, all others cannot book any resources and need to be ignored. The full query can be found in Listing A.7.

In the beginning we also talked about a daily history. Mostly the monthly overview is the more solid one to keep an eye on companies and their activity. The reason for this is that a lot of users do not book every day or regularly and some even just book once in a few weeks. Anyways, a daily overview still holds a lot of information, especially on which days of the week users are more active. In order to achieve a daily overview, we can simply modify the query from before and replace every keyword `month` with `day`.

Subsequently, the query can now be added as data source in DataStudio (see subsection 3.3.2). Though two mentioned metrics are still missing at this point which we add now by modifying this data source. The formulas for bookings per user and booking per active users are the following:

```
SUM(numberBookings) / SUM(numberUsers)
SUM(numberBookings) / SUM(numberActiveUsers)
```

The `SUM` operator is important for a correct weighting, without DataStudio would sum up the result of each row which would lead to a different outcome. This is also the reason why we did not add this metric to the SQL query itself. Relative comparison of absolute metrics should therefore always be done in DataStudio.

Interpretation

In Figure 4.1 you can see the final visualization in DataStudio for the monthly history for a random customer of deskbird. The amount of users and active users is shown in the orange and yellow bar so you have a direct comparison, all other metrics are drawn as lines. We can clearly see how the amount of bookings increases with the amount of (active) users. Also the dip in December is visible where you would expect the users to book less because of the holidays.

The very low numbers in the first months from July to October 2021 are typical for almost every company and represent the testing and demo period where the customer tries out deskbird and compares it to competitors. As soon as they have decided to use the application and signed the contract their employees register and start booking.

Looking at the bookings and active users, they keep more or less the same from November 2021 to February 2022 while the total number of users increases. The drop in December can be justified by the holidays, January and February were still affected by the mandatory home office due to the COVID-19 pandemic. In Germany this is not the case anymore since mid of March 2022 and people are coming back to office. You can

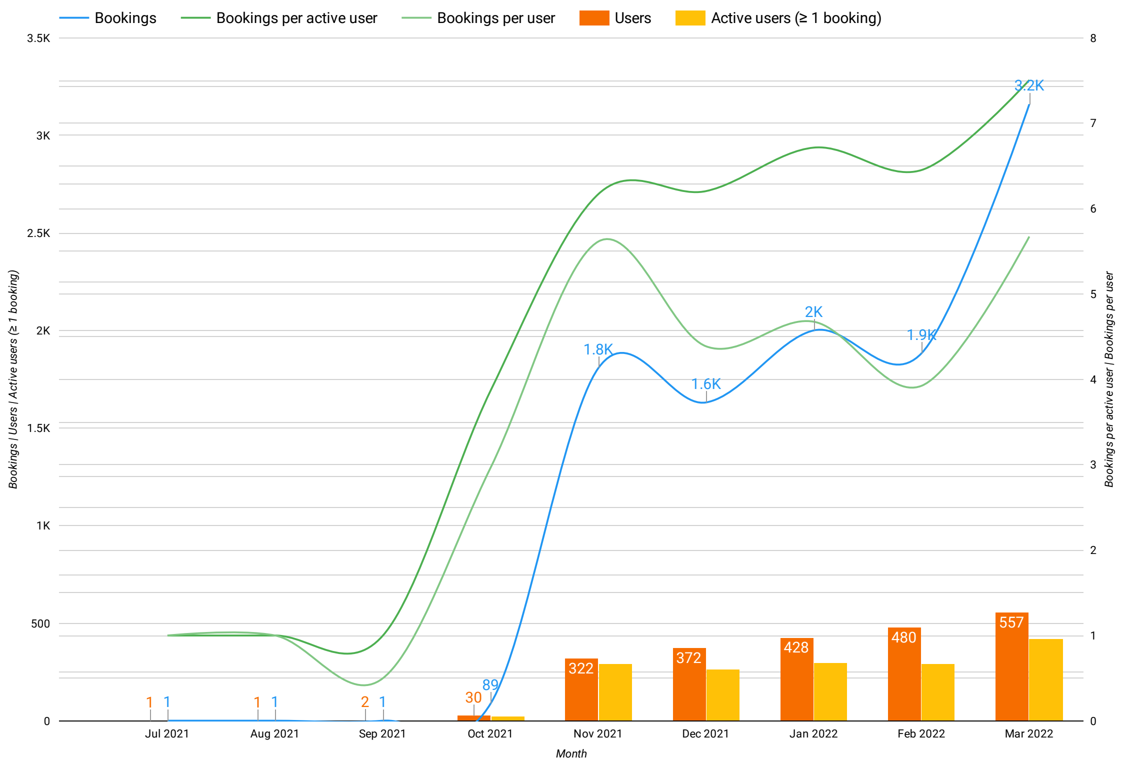


Figure 4.1: Monthly history of a random company

clearly see the big increase of bookings and active users in March and it is expected that this number grows even more in the upcoming months.

The two green lines represent the rate of bookings to the number of users. If those comparisons drop this is an good indicator to approach the customer and ask them whether they need support. In this example it more or less keeps the same and also starts increasing in March, so deskbird can be happy with this customer.

Let us now have a look at another company's statistic in Figure 4.2, but this time at the daily history of the month March 2022. In this case we do not see the total number of users as its daily change is not relevant for analyses. Though the rate of bookings to the total amount of users can still be seen in the light green line. We can clearly see the increase of bookings and active users since the drop of the mandatory home office in Germany on 20th of March which fits with our earlier observation on the monthly overview.

Apart from that we now get a good insight in when exactly users make bookings and can compare the week days with each other. Although the week from 7th to 13th of March falls out of line – reason for that is the international women's day – we can see that most bookings are made at the start of the week. Just a few make their bookings already on Sunday, on Saturday there are usually no new bookings noticed. Additionally, we can see that to the end of the week, especially on Thursday, the number of bookings

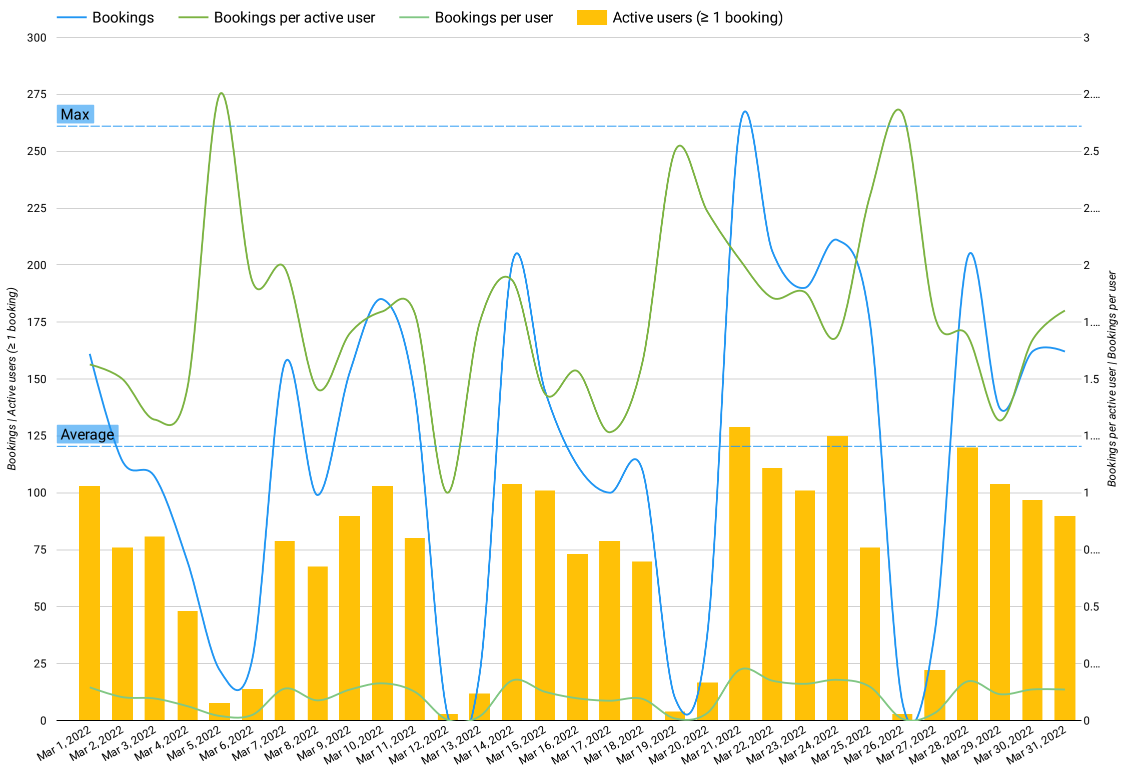


Figure 4.2: Daily history of a random company

also increases. A lot of people seem to like making their bookings for the upcoming week before they start into the weekend.

4.2.2 Same desk booking

The idea of deskbird is to not have an own desk, but sharing desks with other employees and booking one when needed. On the other hand humans are creatures of habit. That is why the question came up about how many users book the same desk over and over again and whether a feature for favoring desks is necessary and important.

This also can vary strongly across the different companies, so we want an overview about each company and the percentage of users that book the same resource for more than 50% / 60% / 70% / 80% / 90% of their time. Additionally we have to take care of the resource type as a normal desk will be booked differently than a meeting room or a parking slot.

In the following, an area is called `zone` and a resource like a desk `zoneItem`.

Implementation

Initially we calculate the rate between the number of bookings from one user at the same desk compared to the total number of bookings of this same user. The first number can be easily calculated by counting all bookings and grouping by `userId` and `zoneItemId`. If we want to count the total bookings of a single user though, we need to group by `userId` only. We could either do two separate queries on the same table, or make use of a window function. In the window we need to partition by `userId` and sum up the count we did before for every `zoneItemId`. That requires us to do the same count twice in the same query, but this is still more performant than querying the same table twice. Accordingly, we end up to calculate the rate for each `zoneItemId` with Listing 4.8.

```sql
SELECT
  bookings.userId AS userId,
  bookings.companyId AS companyId,
  bookings.zoneItemId,
  COUNT(DISTINCT bookings.id) / SUM(COUNT(DISTINCT bookings.id)) OVER (
      PARTITION BY userId) AS bookingRate
FROM
  'deskbird-bbe72.firestore_mirror.bookings' bookings
```

Listing 4.8: Booking rate per `zoneItem`

Additionally, we want to filter those tables by resource type. For our query we need to do counting and rate calculation directly in SQL and cannot outsource it to DataStudio as we did in subsection 4.2.1. Luckily, DataStudio can forward parameters to our query, so we can filter by joining the `zones` view (see Listing A.6) and look for the given types with the following part in the `WHERE` clause.

```sql
WHERE zones.type IN UNNEST(@type)
```

Subsequently, we now want to check those rates and count if it is higher than 50% / 60% / 70% / 80% / 90%. Because of the fact that only one rate per user can be higher than 50%, we simply can count those cases and group by company. Again, we do this by left joining our rates to the `companies` table, so we do not miss any company, and filter for those having workspace booking enabled. Thereby is `usersPerZoneItem` our result table from the previous rate calculation in Listing 4.8.

```sql
SELECT
  companies.id AS companyId,
  companies.name AS companyName,
  companies.status AS companyStatus,
  COUNTIF(usersPerZoneItem.bookingRate > 0.5) AS
      numberOfUsersBookingSameDeskOver50,
  COUNTIF(usersPerZoneItem.bookingRate > 0.6) AS
      numberOfUsersBookingSameDeskOver60,
  COUNTIF(usersPerZoneItem.bookingRate > 0.7) AS
      numberOfUsersBookingSameDeskOver70,
  COUNTIF(usersPerZoneItem.bookingRate > 0.8) AS
      numberOfUsersBookingSameDeskOver80,
  COUNTIF(usersPerZoneItem.bookingRate > 0.9) AS
      numberOfUsersBookingSameDeskOver90
FROM
  `deskbird-bbe72.firestore_mirror.businessCompanies` companies
LEFT JOIN
  usersPerZoneItem
ON
  companies.id = usersPerZoneItem.companyId
GROUP BY
  companyId,
  companyName,
  companyStatus
```

Listing 4.9: Booking rate counting per company

Now we have the number of users that book one desk more than 50% / 60% / 70% / 80% / 90% of their time per company. This number is not very meaningful without comparing it to the total number of users at the company. For this reason we count the number of users by grouping the `users` table and also left joining it to the companies. The full query can be found in Listing A.8.

Again, we now set up a data source in DataStudio and add extra fields to convert the absolute into relative numbers. The `SUM` here helps us to get a weighted total percentage.

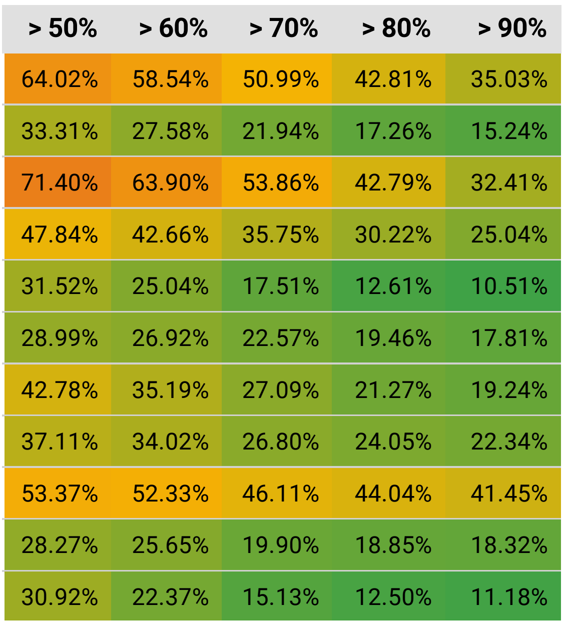
```
SUM(numberOfUsersBookingSameDeskOver50) / SUM(numberUsers)
SUM(numberOfUsersBookingSameDeskOver60) / SUM(numberUsers)
SUM(numberOfUsersBookingSameDeskOver70) / SUM(numberUsers)
SUM(numberOfUsersBookingSameDeskOver80) / SUM(numberUsers)
SUM(numberOfUsersBookingSameDeskOver90) / SUM(numberUsers)
```

Interpretation

The results of the implemented query can be best shown in a simple table as you can see in Figure 4.3. In this example we filtered for only active companies and desks as resource type. Each row presents the rate of users that book the same desk over and over again for more than 50% / 60% / 70% / 80% / 90% of their bookings. The companies' names and any further information are cut off for privacy reasons.

In general those rates are quite high, 71% of employees of one company have made more than half of their bookings at the same desk. This speaks very much for the fact that man is a creature of habit as we expected earlier. Another company notes 41% of their employees booking the same desk for more than 90% of their time.

However, there are big differences among the companies, so the first column of this table reaches from about 2% to 85%. This percentage strongly depends on policies of the companies. Some of them have most of their employees in office everyday while others



> 50%	> 60%	> 70%	> 80%	> 90%
64.02%	58.54%	50.99%	42.81%	35.03%
33.31%	27.58%	21.94%	17.26%	15.24%
71.40%	63.90%	53.86%	42.79%	32.41%
47.84%	42.66%	35.75%	30.22%	25.04%
31.52%	25.04%	17.51%	12.61%	10.51%
28.99%	26.92%	22.57%	19.46%	17.81%
42.78%	35.19%	27.09%	21.27%	19.24%
37.11%	34.02%	26.80%	24.05%	22.34%
53.37%	52.33%	46.11%	44.04%	41.45%
28.27%	25.65%	19.90%	18.85%	18.32%
30.92%	22.37%	15.13%	12.50%	11.18%

Figure 4.3: Same desk booking rate

mainly work from home and only make bookings every two weeks.

Overall we have about 48% of the employees that book half of their time at the same desk, 24% are even booking the same desk for over 90% of their time. These rates tell us that people like to sit on the same desk. Probably they have their favorite employees and friends sitting next to them. Depending on the requirements deskbird could now either encourage users to book more different desks or implement a simple-to-use feature to make book the favorite desk even faster. Maybe both at the same time would also be an option.

4.2.3 Length of bookings

How much time do users spend at one desk or meeting room? Do they use deskbird for very short bookings or are they in office the whole day? In this analysis we want to have a look at the length of bookings per company and offices, and find out whether there are differences across them.

In order to achieve this we want the amount of bookings for each company and office and count how many of those are in one of the following time ranges in hours: 0:00 – 1:59, 2:00 – 3:59, 4:00 – 5:59, 6:00 – 7:59, $\geq$ 8:00, full day. Thereby a full day booking is given if the user turns the corresponding switch in the application on. This claims the resource for the whole time the specific office is open on this day and sets a special flag in the backend database. All other time ranges are excluding full day bookings.

Implementation

For our implementation we basically need to calculate the duration of each booking by comparing its start and end time and count them, grouped by our predefined time ranges. We want to count those numbers per office and resource type so we have to join the areas, called `zones` in the databases, on `zoneId`. Then we can group by office and type and count the total number of bookings as well as the bookings in our time ranges. In order to do this we make use of the `COUNTIF(...)` function. For the full day bookings we just have to check the `isFullDay` flag, all others need a time comparison. The function `TIMESTAMP_DIFF(..., HOUR)` returns the full hours of the duration. Accordingly, a time range of 1:59 would result in 1 hour. To count all bookings in a time range of 0:00 – 1:59 hours this would look like this:

```
COUNTIF(bookings.isFullDay = FALSE AND TIMESTAMP_DIFF(bookings.endTime,
    bookings.startTime, HOUR) BETWEEN 0 AND 1)
```

For both `companyId` and `officeId` we also need their names for an understandable visualization. This can be reached by left joining our results from the counting to the companies and offices. Additionally, we filter for only those companies that have workspace booking enabled. The full query can be found in Listing A.9.

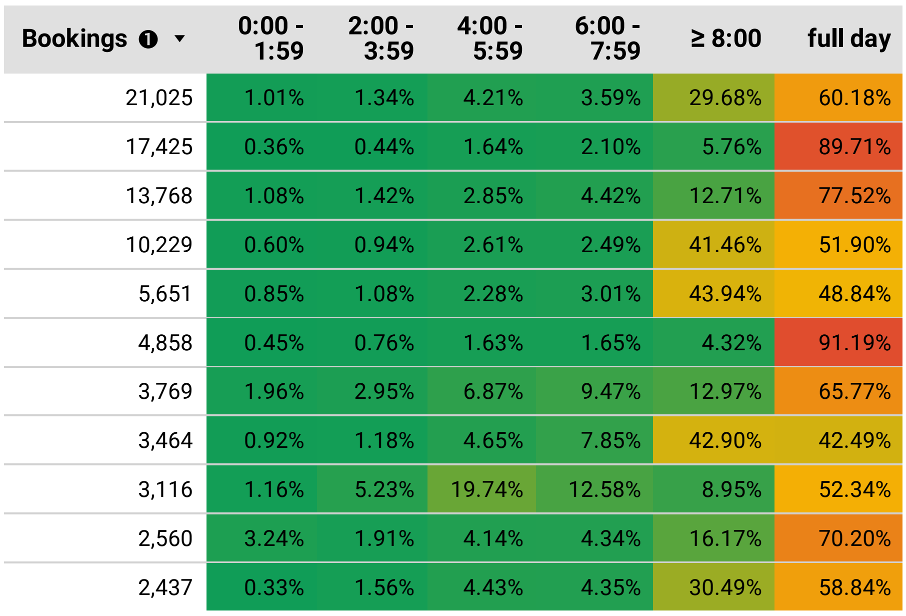
Similar to the two previous implementations we, again, want to calculate the percentage of our counting to the total amount of bookings. Accordingly, we add a few fields to the data source in DataStudio.

```
SUM(bookingTime0To2h) / SUM(numberBookings)
SUM(bookingTime2To4h) / SUM(numberBookings)
SUM(bookingTime4To6h) / SUM(numberBookings)
SUM(bookingTime6To8h) / SUM(numberBookings)
SUM(bookingTimeGreaterEquals8h) / SUM(numberBookings)
SUM(fullDayBookings) / SUM(numberBookings)
```

Interpretation

The results from the previously implemented SQL query can now be used for an overview per company, but also show us details of each single office if necessary. Some example rows of the company summaries can be found in Figure 4.4, filtered by active companies and desk bookings. Each row represents one company and its distribution of booking lengths.

One thing stands out immediately: Most of the bookings are full day ones. We can deduct from this that users work more or less the whole time of the opening times of their office, or at least claim their desk for this time range. The offices of the second row and company for example are open for ten hours each day. Assuming that most



Bookings ❶ ▾	0:00 - 1:59	2:00 - 3:59	4:00 - 5:59	6:00 - 7:59	≥ 8:00	full day
21,025	1.01%	1.34%	4.21%	3.59%	29.68%	60.18%
17,425	0.36%	0.44%	1.64%	2.10%	5.76%	89.71%
13,768	1.08%	1.42%	2.85%	4.42%	12.71%	77.52%
10,229	0.60%	0.94%	2.61%	2.49%	41.46%	51.90%
5,651	0.85%	1.08%	2.28%	3.01%	43.94%	48.84%
4,858	0.45%	0.76%	1.63%	1.65%	4.32%	91.19%
3,769	1.96%	2.95%	6.87%	9.47%	12.97%	65.77%
3,464	0.92%	1.18%	4.65%	7.85%	42.90%	42.49%
3,116	1.16%	5.23%	19.74%	12.58%	8.95%	52.34%
2,560	3.24%	1.91%	4.14%	4.34%	16.17%	70.20%
2,437	0.33%	1.56%	4.43%	4.35%	30.49%	58.84%

Figure 4.4: Booking length per company

of the people work eight hours a day with an additional hour of lunch break, it makes perfectly sense to just book the whole day.

Apart from that, the second highest rates can be found for eight and more hour duration. This also fits with a common eight hours job. Together with the full day booking this already sums up to about 90%. Just a few companies show other time ranges with higher rates than 10%. This also depends highly on the policy of a company: If many employees just work half-time, a higher percentage of 4:00 – 5:59 hours seems quite logical.

When changing the filter to only show meeting rooms, the distribution looks quite different. Full day is much rarer, a duration of up to four hours is most common. This also satisfies the expectation that meeting rooms are not used the whole day by the same person but just booked for a few hours.

We initially grouped the results by office and not only by company. Accordingly, we can compare the data across all offices of a single company. The analysis about this is very brief: Differences in one company are quite small, all offices have more or less the same percentages. One variation can still be found though. The distribution of full day and $\geq$ eight hours bookings is not consistent. However, there might be a simple reason for that. If an office is open for e.g. ten to eleven hours and an employee spends nine to ten hours in the office, some of them will just simply claim the full day while others are more precise and book just their actual time being in office.

Both of the visualizations drop questions, especially about the full day and very long bookings. Why are they distributed like this? Did some people just scan the QR code at their workplace and then booked the rest of the day because it is the presetting when doing so? Those and many other questions regarding this can probably be answered by comparing the creation and the actual start timestamp of each booking to clarify whether the booking was created at the time it already starts or some time earlier.

5 Findings

Our implementations from chapter 4 give us informative insights about user behavior. The visualizations with DataStudio allow us to easily compare this information and get a good overview. In addition, interactive filters and selections let us, and especially every non-expert user with access, even dive deeper into details. Apart from just having interesting information, what can we learn from these analyses? How can they support decision making for deskbird?

5.1 Customer approaching

The most important information we get from our implementations is the activity of the customers. We can exactly tell how many users are booking and how frequently they are doing so. Comparing this with expectations leads to a good feeling whether users like to use the product or not and also how this changes over time. If the activity of a company drops due to less bookings or less people using the app, this is an indicator to approach and talk with them. Usually they do not reach out to you on their owns and maybe they did not even notice that they use the product less than they used to. In a conversation they can tell you what is going wrong or if this is maybe just an exceptional situation and will fall back to normal behavior in some time. Either way, deskbird knows what concerns the customer, can learn from that and accordingly starts with adjustments on the product to improve the user experience.

While the example in Figure 4.1 looks promising and shows what deskbird expects from their users, other companies' activity decreases. If we examine the activity of another company in Figure 5.1, we can definitely see that its users book way less than they used to when they started with deskbird. The amount of users increases consistently while the number of bookings decreases at the same time. In addition, the active user rates are quite low, only a forth of the users made at least one booking in March 2022. Even in the best time of this company less than half of the people used deskbird in a whole month.

Reasons for that can be different. Maybe employees of this company just go to office once in a few weeks and mostly work from home, so the companies policy dictates this behavior and accordingly the amount of workplaces is not even big enough for more active users. Another reason could be that users do not need to book their desks because they are usually working at the same place and can expect it to be free. However, we do not definitely know the cause, so it is worth approaching this company and ask if everything is alright. If they are unsatisfied with the product, they can inform about that and trigger a discussion about possible solutions.

5.2 Feature improvement

Another thing we can draw from the visualizations is information about what feature could be especially important to some users. So this can be either developed if not yet existing or adjusted to fit the customers' requirements in the best possible way. If we, for example, have a look back at Figure 4.3, we can detect the strong bond of users to their favorite workplaces. In average, almost half of the users books the very same desk at least every second time. Other companies note even more than 70% of their employees spending half of their time at their favorite desk.

This implicates that a feature for saving the beloved places would probably be appreciated. At this time there is only a feature to favor an area, the specific desk cannot be saved yet, so this is definitely a thing to be implemented in the future. This favorite desk then could e.g. also be preselected for new bookings to even simplify things further.

An additional information about a possible feature improvement can be found in the distribution of booking lengths in subsection 4.2.3. Most people just make their bookings with the full day switch turned on, meaning they occupy the resource for the whole opening hours of the office. Although some offices are open for ten to eleven

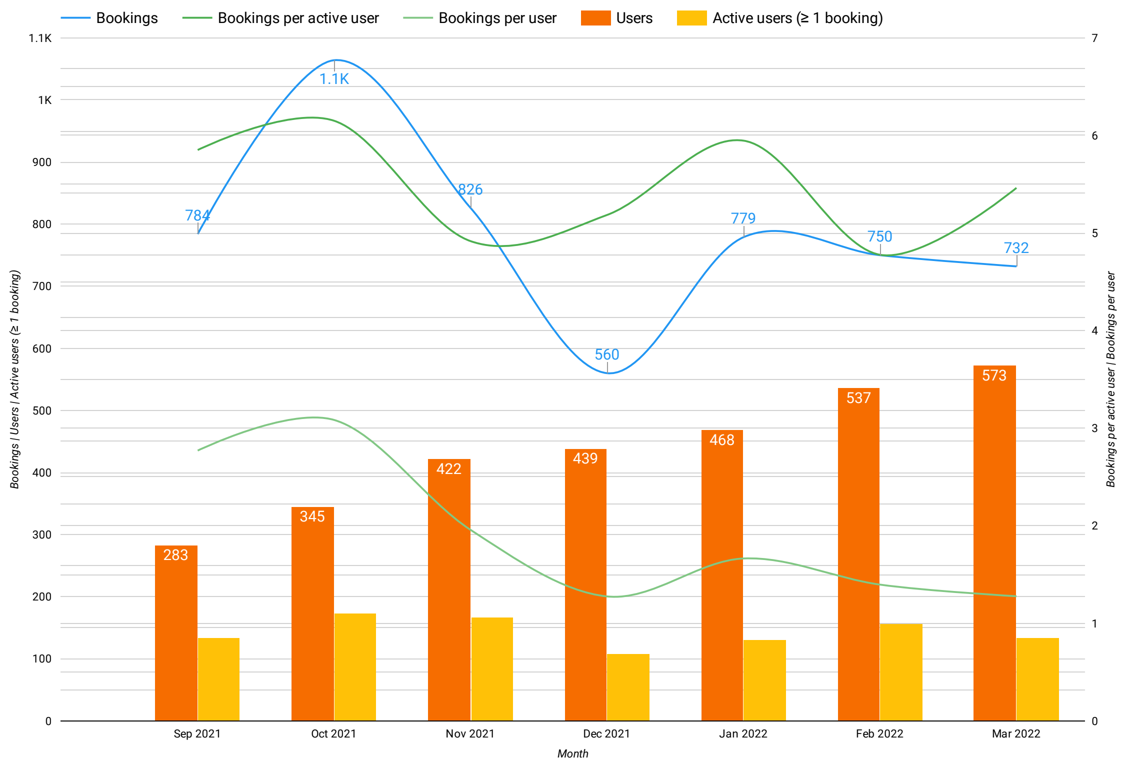


Figure 5.1: Monthly history of a random company with decreasing activity

hours, the rate of full day bookings is still high. Either those users spend the whole time in office or just turn this switch on for simplicity reasons. If someone is in office for about nine hours and the office closes after 10 hours opening time, it is probably unnecessary to free the resource for the last hour as no one will need it. Either way, the full day booking is an attractive feature. Currently this can be done by making a normal booking but turning the switch for full day on. Could it be useful to add an extra button where you can book the full day in just one click?

6 Conclusion and future work

In this thesis we have implemented a data analytics system and built insights for the B2B SaaS product deskbird. We chose Google BigQuery and DataStudio as it fits perfectly to the Google Cloud based infrastructure of deskbird and simplifies a lot of initial effort setting up the analytics environment. First insights about general user behavior such as a monthly history of users' activity or the bond to the very same workplace have already shown that our setup is working as expected and can give valuable information. This can be used to lead future decision making and improve deskbird's products in various ways like better customer approaching or feature improvements.

Nevertheless, monitoring users' behavior, developing analyses and improving the product will never find to an end. On the one hand, deskbird continues developing its applications, so the product will change over time. Accordingly, new analyses must also be added or existing ones be modified. On the other hand, you can carry on with the analytical insights and intensify detailed analyses to improve the understanding of more general reports. For this reason, deskbird continues in developing analyses and creating new insights to gain further knowledge of user behavior and support future growth.

Apart from that, we did not have a closer look at other data sources than Firestore yet. As deskbird is going to make more use of SQL databases for different reasons, better ways of streaming this data to BigQuery will be necessary. Additionally, client side tracking is going to be introduced which also forms an extra data source with completely different aspects of user behavior. For this purpose, streaming other data sources than Firestore to BigQuery will be subject to further research and development based on this thesis.

Appendix

A.1 Firebase views

Listing A.1: Firebase extension view for latest entry of each document

```sql
SELECT
  document_name,
  document_id,
  timestamp,
  event_id,
  operation,
  data
FROM
  (
    SELECT
      document_name,
      document_id,
      FIRST_VALUE(timestamp) OVER(
        PARTITION BY document_name
        ORDER BY
          timestamp DESC
      ) AS timestamp,
      FIRST_VALUE(event_id) OVER(
        PARTITION BY document_name
        ORDER BY
          timestamp DESC
      ) AS event_id,
      FIRST_VALUE(operation) OVER(
        PARTITION BY document_name
        ORDER BY
          timestamp DESC
      ) AS operation,
      FIRST_VALUE(data) OVER(
        PARTITION BY document_name
        ORDER BY
          timestamp DESC
      ) AS data,
      FIRST_VALUE(operation) OVER(
        PARTITION BY document_name
        ORDER BY
          timestamp DESC
      ) = "DELETE" AS is_deleted
    FROM
      ‘deskbird-bbe72.firestore_mirror.collectionName_raw_changelog‘
    ORDER BY
      document_name,
      timestamp DESC
  )
WHERE
  NOT is_deleted
GROUP BY
  document_name,
  document_id,
  timestamp,
  event_id,
  operation,
  data
```

A.2 Views for JSON reprocessing

Listing A.2: View for companies with JSON reprocessing

```
SELECT
  JSON_VALUE(DATA, '$.id') AS id,
  JSON_VALUE(DATA, '$.name') AS name,
  JSON_VALUE(DATA, '$.status') AS status,
  SAFE_CAST(JSON_VALUE(DATA, '$.allowsOfficePlanning') AS BOOLEAN) AS officePlanning,
  SAFE_CAST(JSON_VALUE(DATA, '$.allowsResourceBooking') AS BOOLEAN) AS resourceBooking,
  TIMESTAMP_MILLIS(SAFE_CAST(JSON_VALUE(DATA, '$.createdAt') AS INT64)) AS creationTime
FROM
  `deskbird-bbe72.firestore_mirror.businessCompanies_raw_latest`
```

Listing A.3: View for users with JSON reprocessing

```
SELECT
  JSON_VALUE(DATA, '$.id') AS id,
  JSON_VALUE(DATA, '$.firstName') AS firstName,
  JSON_VALUE(DATA, '$.lastName') AS lastName,
  JSON_VALUE(DATA, '$.email') AS email,
  JSON_VALUE(DATA, '$.companyId') AS companyId,
  TIMESTAMP_MILLIS(SAFE_CAST(JSON_VALUE(DATA, '$.createdAt') AS INT64)) AS creationTime
FROM
  `deskbird-bbe72.firestore_mirror.users_raw_latest`
WHERE
  JSON_VALUE(DATA, '$.status') = "active"
```

Listing A.4: View for bookings with JSON reprocessing

```
SELECT
  JSON_VALUE(DATA, '$.id') AS id,
  JSON_VALUE(DATA, '$.userId') AS userId,
  TIMESTAMP_MILLIS(SAFE_CAST(JSON_VALUE(DATA, '$.createdAt') AS INT64)) AS creationTime,
  TIMESTAMP_MILLIS(SAFE_CAST(JSON_VALUE(DATA, '$.updatedAt') AS INT64)) AS updateTime,
  TIMESTAMP_MILLIS(SAFE_CAST(JSON_VALUE(DATA, '$.bookingStartTime') AS INT64)) AS
      startTime,
  TIMESTAMP_MILLIS(SAFE_CAST(JSON_VALUE(DATA, '$.bookingEndTime') AS INT64)) AS endTime,
  JSON_VALUE(DATA, '$.companyId') AS companyId,
  JSON_VALUE(DATA, '$.workspaceId') AS officeId,
  JSON_VALUE(DATA, '$.resourceId') AS zoneId,
  JSON_VALUE(DATA, '$.zoneItemId') AS zoneItemId,
  JSON_VALUE(DATA, '$.bookingStatus') AS bookingStatus,
  SAFE_CAST(JSON_VALUE(DATA, '$.isDayPass') AS BOOLEAN) AS isFullDay
FROM
  `deskbird-bbe72.firestore_mirror.bookings_raw_latest`
WHERE
  SAFE_CAST(JSON_VALUE(DATA, '$.isInternalBooking') AS BOOLEAN) = TRUE
  AND SAFE_CAST(JSON_VALUE(DATA, '$.demoBooking') AS BOOLEAN) = FALSE
```

Listing A.5: View for workspaces with JSON reprocessing

```
SELECT
  JSON_VALUE(DATA, '$.id') AS id,
  JSON_VALUE(DATA, '$.name') AS name,
  JSON_VALUE(DATA, '$.businessCompanyId') AS companyId,
  JSON_VALUE(DATA, '$.address.city') AS city,
  JSON_VALUE(DATA, '$.address.country') AS country,
  ST_GEOGPOINT(
    SAFE_CAST(JSON_VALUE(DATA, '$.address.coordinate._longitude') AS NUMERIC),
    SAFE_CAST(JSON_VALUE(DATA, '$.address.coordinate._latitude') AS NUMERIC)
  ) AS coordinates
FROM
  `deskbird-bbe72.firestore_mirror.internalWorkspaces_raw_latest`
```

Listing A.6: View for zones with JSON reprocessing

```
SELECT
  JSON_VALUE(DATA, '$.id') AS id,
  JSON_VALUE(DATA, '$.name') AS name,
  SAFE_CAST(JSON_VALUE(DATA, '$.isActive') AS BOOLEAN) AS isActive,
  JSON_VALUE(DATA, '$.type') AS type,
  JSON_VALUE(DATA, '$.group.id') AS groupId,
  JSON_VALUE(DATA, '$.workspaceId') AS officeId
FROM
  `deskbird-bbe72.firestore_mirror.zones_raw_latest`
```

A.3 Full queries of the implementations

Listing A.7: Historical overview of users and bookings

```
WITH bookings AS (
  SELECT
    companyId,
    DATE_TRUNC(EXTRACT(DATE FROM creationTime), MONTH) AS month,
    COUNT(DISTINCT userId) AS numberActiveUsers,
    COUNT(DISTINCT id) AS numberBookings
  FROM
    ‘deskbird-bbe72.firestore_mirror.bookings‘
  GROUP BY
    companyId,
    month
), users AS (
  SELECT
    companyId,
    DATE_TRUNC(EXTRACT(DATE FROM creationTime), MONTH) AS month,
    COUNT(DISTINCT id) AS numberUsers
  FROM
    ‘deskbird-bbe72.firestore_mirror.users‘
  GROUP BY
    companyId,
    month
)
SELECT
  companies.id,
  companies.name,
  companies.status,
  month,
  IFNULL(SUM(numberUsers) OVER w, 0) AS numberUsers,
  IFNULL(numberActiveUsers, 0) AS numberActiveUsers,
  IFNULL(numberBookings, 0) AS numberBookings
FROM
  ‘deskbird-bbe72.firestore_mirror.businessCompanies‘ companies,
  UNNEST(GENERATE_DATE_ARRAY(
    DATE_TRUNC(EXTRACT(DATE FROM companies.creationTime), MONTH),
    CURRENT_DATE(),
    INTERVAL 1 MONTH
  )) month
LEFT JOIN
  bookings
ON
  bookings.companyId = companies.id
  AND bookings.month = month
LEFT JOIN
  users
ON
  users.companyId = companies.id
  AND users.month = month
```

```sql
WHERE
  companies.resourceBooking = TRUE
WINDOW w AS (
  PARTITION BY
    companies.id
  ORDER BY
    month ASC
)
```

Listing A.8: Same desk booking

```sql
WITH users AS (
  SELECT
    companyId,
    COUNT(DISTINCT id) AS numberUsers
  FROM
    `deskbird-bbe72.firestore_mirror.users`
  GROUP BY
    companyId
), usersPerZoneItem AS (
  SELECT
    bookings.userId AS userId,
    bookings.companyId AS companyId,
    bookings.zoneItemId,
    COUNT(DISTINCT bookings.id) / SUM(COUNT(DISTINCT bookings.id)) OVER (PARTITION BY
         userId) AS bookingRate
  FROM
    `deskbird-bbe72.firestore_mirror.zones` AS zones,
    `deskbird-bbe72.firestore_mirror.bookings` bookings
  WHERE
    bookings.zoneId = zones.id
    AND zones.type IN UNNEST(@string)
  GROUP BY
    userId,
    companyId,
    zoneItemId
)
SELECT
  companies.id AS companyId,
  companies.name AS companyName,
  companies.status AS companyStatus,
  IFNULL(users.numberUsers, 0) AS numberUsers,
  COUNTIF(usersPerZoneItem.bookingRate > 0.5) AS numberOfUsersBookingSameDeskOver50,
  COUNTIF(usersPerZoneItem.bookingRate > 0.6) AS numberOfUsersBookingSameDeskOver60,
  COUNTIF(usersPerZoneItem.bookingRate > 0.7) AS numberOfUsersBookingSameDeskOver70,
  COUNTIF(usersPerZoneItem.bookingRate > 0.8) AS numberOfUsersBookingSameDeskOver80,
  COUNTIF(usersPerZoneItem.bookingRate > 0.9) AS numberOfUsersBookingSameDeskOver90
FROM
  `deskbird-bbe72.firestore_mirror.businessCompanies` companies
LEFT JOIN
  users
```

```sql
ON
  companies.id = users.companyId
LEFT JOIN
  usersPerZoneItem
ON
  companies.id = usersPerZoneItem.companyId
WHERE
  companies.resourceBooking = TRUE
GROUP BY
  companyId,
  companyName,
  companyStatus,
  numberUsers
```

Listing A.9: Booking length

```sql
WITH bookingsPerOffice AS (
  SELECT
    bookings.officeId AS officeId,
    zones.type AS type,
    COUNT(DISTINCT bookings.id) AS numberBookings,
    COUNTIF(bookings.isFullDay = TRUE) AS fullDayBookings,
    COUNTIF(bookings.isFullDay = FALSE AND TIMESTAMP_DIFF(bookings.endTime, bookings.
        startTime, HOUR) BETWEEN 0 AND 1) AS bookingTime0To2h,
    COUNTIF(bookings.isFullDay = FALSE AND TIMESTAMP_DIFF(bookings.endTime, bookings.
        startTime, HOUR) BETWEEN 2 AND 3) AS bookingTime2To4h,
    COUNTIF(bookings.isFullDay = FALSE AND TIMESTAMP_DIFF(bookings.endTime, bookings.
        startTime, HOUR) BETWEEN 4 AND 5) AS bookingTime4To6h,
    COUNTIF(bookings.isFullDay = FALSE AND TIMESTAMP_DIFF(bookings.endTime, bookings.
        startTime, HOUR) BETWEEN 6 AND 7) AS bookingTime6To8h,
    COUNTIF(bookings.isFullDay = FALSE AND TIMESTAMP_DIFF(bookings.endTime, bookings.
        startTime, HOUR) >= 8) AS bookingTimeGreaterEquals8h
  FROM
    `deskbird-bbe72.firestore_mirror.bookings` bookings,
    `deskbird-bbe72.firestore_mirror.zones` zones
  WHERE
    bookings.zoneId = zones.id
  GROUP BY
    officeId,
    type
)
SELECT
  companies.id AS companyId,
  companies.name AS companyName,
  companies.status AS companyStatus,
  offices.id AS officeId,
  offices.name AS officeName,
  bookingsPerOffice.type AS type,
  bookingsPerOffice.numberBookings,
  bookingsPerOffice.fullDayBookings,
  bookingsPerOffice.bookingTime0To2h,
  bookingsPerOffice.bookingTime2To4h,
  bookingsPerOffice.bookingTime4To6h,
  bookingsPerOffice.bookingTime6To8h,
  bookingsPerOffice.bookingTimeGreaterEquals8h
FROM
  `deskbird-bbe72.firestore_mirror.businessCompanies` companies
LEFT JOIN
  `deskbird-bbe72.firestore_mirror.internalWorkspaces` offices
ON
  companies.id = offices.companyId
LEFT JOIN
  bookingsPerOffice
ON
  offices.id = bookingsPerOffice.officeId
WHERE
  companies.resourceBooking = TRUE
```

List of Figures

2.1 Infrastructure of deskbird 4
2.2 Structure of companies in deskbird 5

4.1 Monthly history of a random company 23
4.2 Daily history of a random company 24
4.3 Same desk booking rate 27
4.4 Booking length per company 29

5.1 Monthly history of a random company with decreasing activity 32

List of Listings

4.1 Example of a BigQuery federated query 16
4.2 Extraction of values in JSON-string in BigQuery 17
4.3 Connection of the previous and the current version of edited bookings . . 19
4.4 Amount of users per company and month 20
4.5 Amount of active users and bookings per company and month 21
4.6 Cross join of companies with month array 21
4.7 Cumulative sum up of amount of users 22
4.8 Booking rate per `zoneItem` 25
4.9 Booking rate counting per company 26

A.1 Firebase extension view for latest entry of each document 37
A.2 View for companies with JSON reprocessing 38
A.3 View for users with JSON reprocessing 38
A.4 View for bookings with JSON reprocessing 38
A.5 View for workspaces with JSON reprocessing 39
A.6 View for zones with JSON reprocessing 39
A.7 Historical overview of users and bookings 40
A.8 Same desk booking 41
A.9 Booking length 42

Bibliography

[1] E. Zagan and M. Danubianu, "Cloud data lake: The new trend of data storage," in *2021 3rd International Congress on Human-Computer Interaction, Optimization and Robotic Applications (HORA)*, Jun. 2021, pp. 1–4. DOI: `10.1109/HORA52670.2021.9461293`.

[2] R. K. Chawda and G. Thakur, "Big data and advanced analytics tools," in *2016 Symposium on Colossal Data Analysis and Networking (CDAN)*, Mar. 2016, pp. 1–8. DOI: `10.1109/CDAN.2016.7570890`.

[3] *Cloud firestore*, `https://firebase.google.com/docs/firestore`, Mar. 2022.

[4] *Cloud sql for postgresql documentation*, `https://cloud.google.com/sql/docs/postgres`, Mar. 2022.

[5] K. Gudfinnsson and M. Strand, "Challenges with bi adoption in smes," in *2017 8th International Conference on Information, Intelligence, Systems & Applications (IISA)*, Aug. 2017, pp. 1–6. DOI: `10.1109/IISA.2017.8316407`.

[6] B. Santos, F. Sério, S. Abrantes, F. Sá, J. Loureiro, C. Wanzeller, and P. Martins, "Open source business intelligence tools: Metabase and redash," in *Proceedings of the 11th International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management, IC3K 2019, Volume 1: KDIR, Vienna, Austria, September 17-19, 2019*, A. L. N. Fred and J. Filipe, Eds., ScitePress, 2019, pp. 467–474. DOI: `10.5220/0008351704670474`.

[7] R. Patgiri and A. Ahmed, "Big data: The v's of the game changer paradigm," Dec. 2016. DOI: `10.1109/HPCC-SmartCity-DSS.2016.0014`.

[8] A. Abelló and O. Romero, "Online analytical processing," in *Encyclopedia of Database Systems*, L. Liu and M. T. Özsu, Eds. New York, NY: Springer New York, 2018, pp. 2558–2563, ISBN: 978-1-4614-8265-9. DOI: `10.1007/978-1-4614-8265-9_252`.

[9] N. Pendse, R. Creeth, and B. (Firm), *The OLAP Report: Succeeding with On-line Analytical Processing*. Business Intelligence, 1995, ISBN: 9781898085218.

[10] B. Hüsemann, J. Lechtenbörger, and G. Vossen, "Conceptual data warehouse design," Jul. 2000.

[11] S. Huber and N. Litzel, *Was ist ein data warehouse?* `https://www.bigdata-insider.de/was-ist-ein-data-warehouse-a-606701/`, May 2017.

[12] Y. Li and S. Manoharan, "A performance comparison of sql and nosql databases," in *2013 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)*, Aug. 2013, pp. 15–19. DOI: `10.1109/PACRIM.2013.6625441`.

[13] N. Samuel, *Sql vs nosql databases: 5 critical differences*, `https://hevodata.com/learn/sql-vs-nosql-databases-5-critical-differences/`, Mar. 2021.

[14] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber, "Bigtable: A distributed storage system for structured data (awarded best paper!)" In *7th Symposium on Operating Systems Design and Implementation (OSDI '06), November 6-8, Seattle, WA, USA*, B. N. Bershad and J. C. Mogul, Eds., USENIX Association, 2006, pp. 205–218.

[15] K. Chodorow and M. Dirolf, *MongoDB - The Definitive Guide: Powerful and Scalable Data Storage*. O'Reilly, 2010, ISBN: 978-1-449-38156-1.

[16] J. C. Anderson, J. Lehnardt, and N. Slater, *CouchDB - The Definitive Guide: Time to Relax*. O'Reilly, 2010, ISBN: 978-0-596-15589-6.

[17] *Extract tranform load (etl)*, `https://databricks.com/glossary/extract-transform-load`, Mar. 2022.

[18] M. Golfarelli and S. Rizzi, "From star schemas to big data: 20+ years of data warehouse research," in *A Comprehensive Guide Through the Italian Database Research Over the Last 25 Years*, S. Flesca, S. Greco, E. Masciari, and D. Saccà, Eds. Cham: Springer International Publishing, 2018, pp. 93–107, ISBN: 978-3-319-61893-7. DOI: `10.1007/978-3-319-61893-7_6`.

[19] N. Shakhovska, N. Boyko, and P. Pukach, "The information model of cloud data warehouses," in *Advances in Intelligent Systems and Computing III - Selected Papers from the International Conference on Computer Science and Information Technologies, CSIT 2018, September 11-14, Lviv, Ukraine*, N. Shakhovska and M. O. Medykovskyy, Eds., ser. Advances in Intelligent Systems and Computing, vol. 871, Springer, 2018, pp. 182–191. DOI: `10.1007/978-3-030-01069-0\_13`.

[20] *Data warehouse architecture: Traditional vs. cloud*, `https://panoply.io/data-warehouse-guide/data-warehouse-architecture-traditional-vs-cloud/`, Mar. 2022.

[21] S. Fernandes and J. Bernardino, "What is bigquery?" In *Proceedings of the 19th International Database Engineering & Applications Symposium*, ser. IDEAS '15, Yokohama, Japan: Association for Computing Machinery, 2015, pp. 202–203, ISBN: 9781450334143. DOI: `10.1145/2790755.2790797`.

[22] I. Pandis, "The evolution of amazon redshift," *Proc. VLDB Endow.*, vol. 14, no. 12, pp. 3162–3163, 2021.

[23] G. s. Bhathal and A. S. Dhiman, "Big data solution: Improvised distributions framework of hadoop," in *2018 Second International Conference on Intelligent Computing and Control Systems (ICICCS)*, Jun. 2018, pp. 35–38. DOI: `10.1109/ICCONS.2018.8663142`.

[24] F. Yang, E. Tschetter, X. Léauté, N. Ray, G. Merlino, and D. Ganguli, "Druid: A real-time analytical data store," in *International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014*, C. E. Dyreson, F. Li, and M. T. Özsu, Eds., ACM, 2014, pp. 157–168. DOI: `10.1145/2588555.2595631`.

[25] Z. Lyu, H. H. Zhang, G. Xiong, G. Guo, H. Wang, J. Chen, A. Praveen, Y. Yang, X. Gao, A. Wang, W. Lin, A. Agrawal, J. Yang, H. Wu, X. Li, F. Guo, J. Wu, J. Zhang, and V. Raghavan, "Greenplum: A hybrid database for transactional and analytical workloads," in *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, G. Li, Z. Li, S. Idreos, and D. Srivastava, Eds., ACM, 2021, pp. 2530–2542. DOI: `10.1145/3448016.3457562`.

[26] G. Snipes, "Google data studio," *Journal of Librarianship and Scholarly Communication*, vol. 6, no. 1, 2018. DOI: `10.7710/2162-3309.2214`.

[27] *Bigquery documentation*, `https://cloud.google.com/bigquery/docs/`, Mar. 2022.

[28] Firebase, *Stream collections to bigquery*, `https://firebase.google.com/products/extensions/firebase-firestore-bigquery-export`, Mar. 2022.