UNIVERSITY OF TARTU
Institute of Computer Science
Software Engineering Curriculum

Harald Astok

Elasticsearch in Pipedrive

Master's Thesis (30 ECTS)

Supervisor: Dietmar Alfred Paul Kurt Pfahl (PhD)
Co-supervisor: Andreas Sepp (MSc)

Tartu 2020

Elasticsearch in Pipedrive

Abstract:

Having a proper and effective search functionality in SaaS application is becoming more and more important. For example, eBay gets 3.5 billion search queries every day and querying directly from relational databases can become very complicated and resource consuming with such volumes. One way to handle scalable searchable data is to use the open-source Elasticsearch search engine. This thesis provides a case study of the adaption of Elasticsearch in an Estonian based SaaS start-up Pipedrive and an overview of the author's participation in the process. It analyses previous solutions that were implementing the usage of Elasticsearch, what technologies they were using and what problems emerged. Then it describes in detail what the current architecture looks like and how Elasticsearch is configured for Pipedrive.

Keywords:

Elasticsearch, Apache Kafka, data streams, microservice, search, indices, agile workflow, project management, scalable software development, designing scalable architecture

CERCS:

P175 Informatics, system theory

Elasticsearch Pipedrive’is

Lühikokkuvõte:

Sobiva ja efektiivse otsingu funktsionaalsus tarkvara kui teenus rakendustes on tähtsuselt üha rohkem kasvamas. Näiteks eBay’s tehakse iga päev 3.5 miljardit otsingu päringut ning pärimine otse relatsioonilisest andmebaasist võib muutuda väga keeruliseks ja ressursi mahukaks selliste mahtude juures. Üks viis käsitleda skaleeritavat otsitavaid andmeid on kasutada vabavaralist Elastisearch otsingumootorit. Käesolev magistritöö on juhtumiuuring, mis käsitleb Elasticsearchi kasutamist Eesti põhisel tarkvara kui teenus idufirmal Pipedrive ning ülevaadet töö autori osalusest arendusprotsessis. Lisaks analüüsitakse eelnevaid lahendusi, mis kasutasid Elasticsearchi, mis tehnoloogiaid kasutati ning mis probleemid tekkisid. Seejärel kirjeldatakse detailset praegust kasutusel olevat arhitektuuri ning kuidas Pipedrive on Elasticsearchi enda jaoks seadistanud.

Võtmesõnad:

Elasticsearch, Apache Kafka, andmevoog, mikroteenused, otsing, indeksid, agiilne töökeskkond, projekti juhtimine, skaleeriva tarkvara arendus, skaleeriva arhitektuuri arendus

CERCS:

P175 Informaatika, süsteemiteooria

Table of Contents

List of acronyms

SaaS – Software as service

MEDLINE - Medical Literature Analysis and Retrieval System Online

WWW – World Wide Web

REST - Representational State Transfer

API – Application Programming Interface

CRM – Customer Relationship Manager

STOMP - Streaming Text Oriented Messaging Protocol

AMQP - Advanced Messaging Queuing Protocol

DRAM - Dynamic random-access memory

UI – User Interface

DoS - Denial-of-service

KPI – Key Performance Indicator

1 Introduction

Communication and searching for information are two of the most popular uses of the internet (Hamburger & Ben-Artzi, 2000). In bigger applications, users can make millions of text search requests per day, which makes querying directly from database slow and inefficient. Having a good search functionality is becoming increasingly important for applications, because user's search queries can be used for future suggestions thus getting useful and resourceful information. For example, in 2016 people made over 63 000 Google search requests per second every day (Sullivan, 2016), which makes around 2 trillion searches per day. Goal of this thesis is to analyse and describe how Pipedrive has implemented the usage of Elasticsearch in its search functionality.

1.1 The importance of search functionality

Searching for information from the web has become daily activity for most people. Having the option of retrieving information from the massive databases available to the public has opened up a new field of scientific studies of how to retrieve the most accurate data the user needs. Those findings in turn have been implemented in applications available to be used. A search engine is the practical application of information retrieval techniques to large-scale text collections. (Crof, Metzler, & Strohman, 2015) It is designed to retrieve documents or other types of results based on the input of the user. Meaning the first step for the search engine is always looking at the data the user has input for the engine. Web search is the most common use of a search engine, but they can also be found in most of the modern enterprise applications.

One of the earliest information retrieval systems that had a computer handling scientific literature, was MEDLINE, built in 1971. It was based on searching from indexed documents written on a magnetic tape and could be used by 25 simultaneous users (Dee, 2007). Until the rise of WWW in the 1990s, this was one of the most advanced information retrieval systems. Search engines like Yahoo and Google use crawlers that scan web pages, downloading and providing them with unique ids and storing the pages for further analyzation. (Brin & Page, 2012) Since most of the enterprise applications cannot use WWW search due to the fact that they need to perform the search from their internal documents, the need for customizable index-based search engine rose. One of the first open-sourced search library for such usage was Apache Lucene in 1999 (Białecki, Muir, & Ingersoll, 2012). Over the years, there have been multiple search engines that use Apache Lucene library, such as

Apache Nutch (Nagel, 2020), Swiftype (Uplavikar, Malin, & Jiang, 2020) and Elasticsearch (Gormley & Tong, 2015).

In case there is a need to add search functionality to an application that has static data or one with minimal changes, migrating data into index-based search service and returning necessary documents upon request would not require a very complex architecture. Index is an optimized collection of JSON documents. Each document is a collection of fields, the key-value pairs that contain one's data (Elastic, 2020). However, if the data is constantly changing and the amount of changes is in millions of documents per day, more complex architecture is required to support the search service. One solution is to use data streams with the search service.

1.2 Scalable software architecture

1.2.1 Monolith vs microservices

Building a monolithic application is a go-to option for the most start-up companies, as it offers a great number of advantages in the early start. When starting with a small team, having one codebase is simpler to develop and deploy, quicker to run tests for all parts of the application and it scales horizontally. However, over time small companies grow and so does the codebase. When monolith becomes too big, it is too complex to fully comprehend and make quick changes correctly. Also, when teams grow and everybody is working on the same project, it is hard to keep up the quality and the code style. Every update deploys the whole project and it is difficult to scale as some parts have conflicting resource requirements. In addition, bug in one part can bring down the entire application. Additionally, having monolithic architecture is a barrier for adopting new technologies. Using different languages or frameworks affects the entire application, making it difficult to maintain (Kharenko, 2015).

If a company has grown large enough to have independent teams working on it, it is a good time to consider starting to split monolith into smaller, interconnected services. The idea is that each service acts as a small application itself, doing one job and passing information along to the other services. Some services might need an exposed REST API to communicate with others, but most consume messages provided by the APIs of other services (Thönes, 2015).

When designing an architecture for new services, developers must keep in mind the established conventions of the teams in order to keep the constant style of the whole application. If the new microservice needs a database, the naming of the tables and columns should be consistent and easy to understand. In addition, it is good to split the database so instead of sharing one schema with other services, each service has its own. If the application has only one database, where multiple services need information from, that one service is going to be bombarded with requests to get some small piece of information. This might result in the duplication of information and cause confusion of where to get some data. In case there is a one database that many services require information from, one solution could be to build a service just to handle requests of that one database.

In general, the microservice architecture pattern should correspond to Y-axis scaling in the Scale Cube model of scalability, that is described in Figure 1. When X-axis scaling consists of running multiple copies of the same application, so, certain amount of load will be distributed and when there is N number of copies, each copy will get 1/N of the load. The Y-axis of the Scale Cube represents a separation of work responsibility by either the type of data, the type of work performed for a transaction, or a combination of both. Z-axis scaling is similar to X-axis scaling in a way that each server runs an identical copy of the application, but the difference is that each server is responsible for only a subset of data. (Abbot & Fisher, 2009)

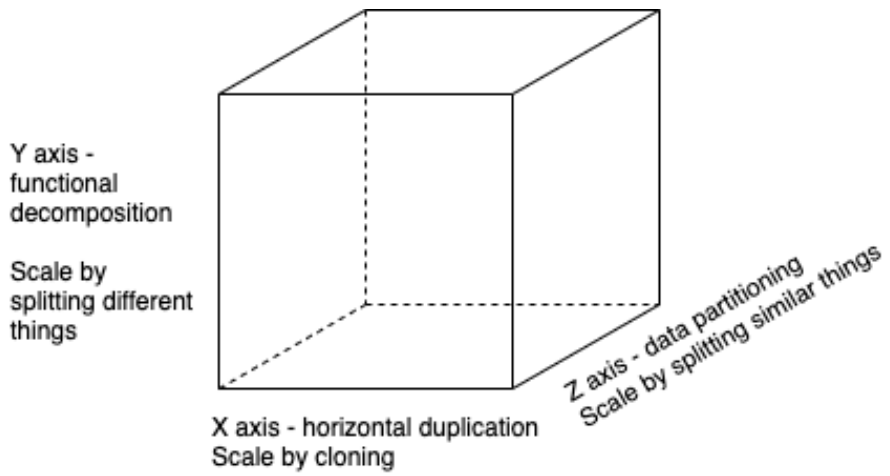

Figure 1 The Scale Cube

1.3 Thesis structure

In chapter 2, an overview of Pipedrive is given. It describes the internal structure of the company, how engineers are divided into teams, its core functionality and what is search in Pipedrive. Moreover, chapter 2 highlights what items are searchable, introduces visibility rules and item following.

In chapter 3, software used by Pipedrive that is related to search functionality is explained. Having a clear understanding of the technologies used helps reader to understand how some of the issues were solved.

Chapter 4 describes all of the previous versions of search architecture in Pipedrive. It starts from the earliest version and ends with the currently used implementation of Elasticsearch version 7. It discusses how data is mapped inside Elasticsearch and how certain types of information processed. Furthermore, what microservices are needed for the Pipedrive search to work and data getting to and from Elasticsearch is disclosed in detail.

In chapter 5, the author's contribution is brought out. It gives a clear overview of what his role has been in the search development process and what were the outcomes from him to Pipedrive.

Chapter 6 examines the metrics and the monitoring of the search services. It also analyses timeframe guidelines Pipedrive developers has set for themselves and what are the actual results.

Chapter 7 discusses potential future expansions. Two problems are described: the issue about handling API events is analysed and the future development plan for search user interface described.

2 Pipedrive overview

This chapter is going to give an overview of what Pipedrive is, its engineering processes and a broad overview of its search functionality.

2.1 Characterization of Pipedrive

Pipedrive is a sales-oriented CRM tool with over 90 000 customers, designed to help small sales teams manage intricate or lengthy sales processes. Their mission to help salespeople focus on actions that close deals. They realized that a sales management tool catered to specifically their needs did not exist yet. Founded in 2010, the company has quickly grown to over 600 employees with their offices in Estonia, United States, United Kingdom, Czech Republic, Latvia, Portugal and Ireland. Because of the competitive CRM market, Pipedrive has always considered themselves as a quick-moving agile company that uses an agile methodology.

Back in 2010, when the development started, all of the code was based in a single repository, which gave the few developers the company had an opportunity for a quick development process and an easier overview of the code. Since Pipedrive is growing rapidly both company and product wise, separating smaller parts from monolith was important to keep up the fast pace of the development process. Splitting monolith to microservices enables Pipedrive to keep up high quality standards of the product and allows developers to keep adding value to the product. The result of the constant scaling was that the number of teams grew and some issues started to arise. These problems needed to be solved quickly to keep up with the high speed of new feature releases. Things such as monitoring, automated deployment and knowledge sharing between teams have been proven to increase the efficiency and speed of the development cycle. Applying new methods and conventions which have been set by mostly learning from the experience of other Agile companies, has also influenced teams to do work in a similar way and use the same tools, so it is easier for engineers to switch between teams.

2.1.1 Engineering processes

Pipedrive engineering department used to be split into independent 6-8 person full stack teams with focuses on specific parts of the product. Microservice architecture was adopted to support this model as it gave teams the ability to grow and deliver faster over time. One of the cornerstones of autonomous teams being able to deliver quickly is the automated

release process. Anyone can start a deployment with their changes which are automatically tested and then deployed. Not all engineering teams worked on the product, some teams were created to support and develop the automation, test systems or support rest of the engineering teams. Such teams were for example DevOps tooling and infrastructure management teams.

Over time when the company grew and number of teams got to around 15 and some teams over 10 people, it became clear that there would be too much management and since each team had a small ownership over some part of the product, it was difficult for the engineers to switch teams. It was also challenging to find suitable people for lead roles inside teams. Furthermore, delivery times for teams started to decline as often they couldn't focus on developing new features because some older functionality needed fixing. Finally, knowledge sharing became a big issue, because only the team responsible for some part of the product knew about what was going on in that specific area and teams facing same problems started to invent different solutions to the same problems.

In 2018, Pipedrive went over to tribes and missions instead of teams to mitigate problems mentioned above. As shown on Figure 2, tribes resemble large teams, consisting of around 20 people. All of them have different roles and each tribe is led by an engineering manager. Tribes then split into two – launchpad and missions. Launchpad's role is to support missions by doing research for oncoming missions, handle smaller tasks and bug fixes, improve overall product quality and fill any on-call duties. Missions, on the other hand, are a smaller team on people, usually involving engineers, product manager and designers. they concentrate on creating or redoing some bigger, more time-consuming features. Having a strict timeframe, mission members are expected to postpone other tribe duties and solely focus on their mission. After the end of the mission, people can go back to launchpad or if they wish, they can join a mission in another tribe. A common trait with the earlier system is that some tribes, the same way as teams once were, are built solely for support purposes. For example, Pipedrive has agile and personal coaching, infrastructure, devops tooling and quality assurance tribes. (Anikin, 2019)

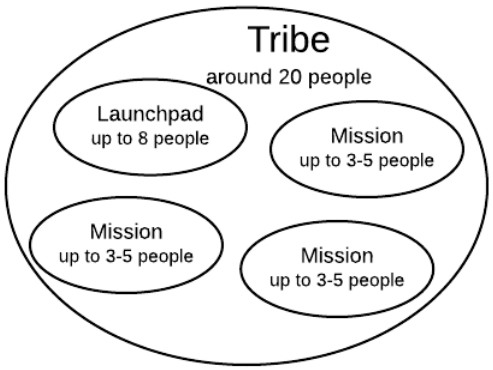

Figure 2 Tribe structure

2.1.2 Pipedrive's functionality

Pipedrive's core functionality centres around pipeline for managing deals and their associated contacts, dashboards for overseeing aggregated sales processes, shown in Figure 3. There, users can see their deals, different stages of the pipeline, what deals are at what stage in the pipeline and some additional information, such as contact person of a deal and its value. The main idea of the landing page is to keep the level of information minimal and only display the most important information. When opening a deal window, user can see and add more details, such as organisation the deal is linked with, custom details, notes, attached files, etc. Also, user can track who is following the deal and get the overall overview of the deal. Figure 4 displays the edited deal view, which gives user the access to all the details, including notes, scheduled activities, custom fields, attached files, etc. the owner of the deal has the option to change the visibility group of the deal. That is important as it determines the visibility rules which make the deal searchable for the other person or not.

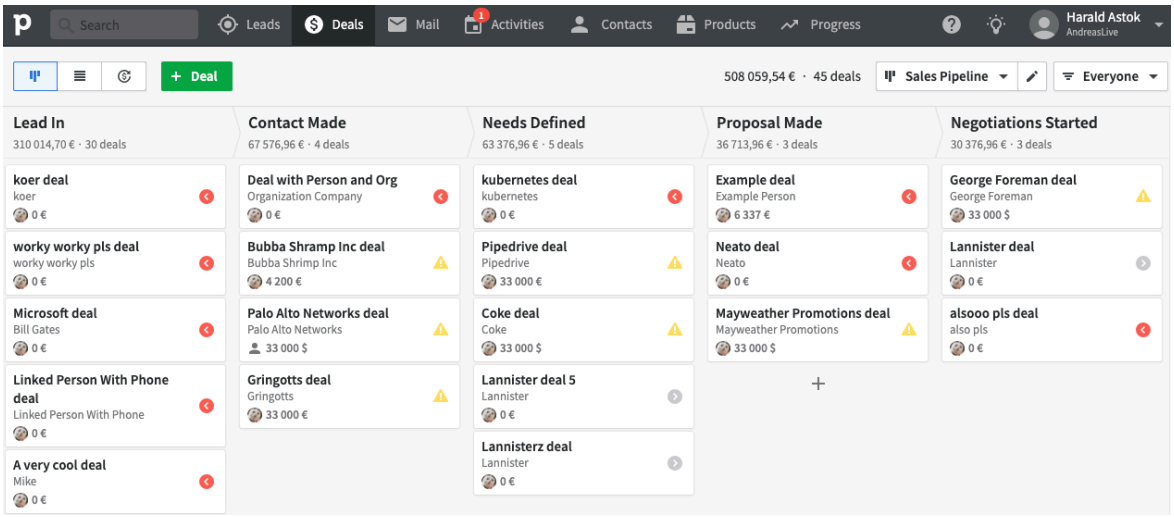

Figure 3 Pipeline

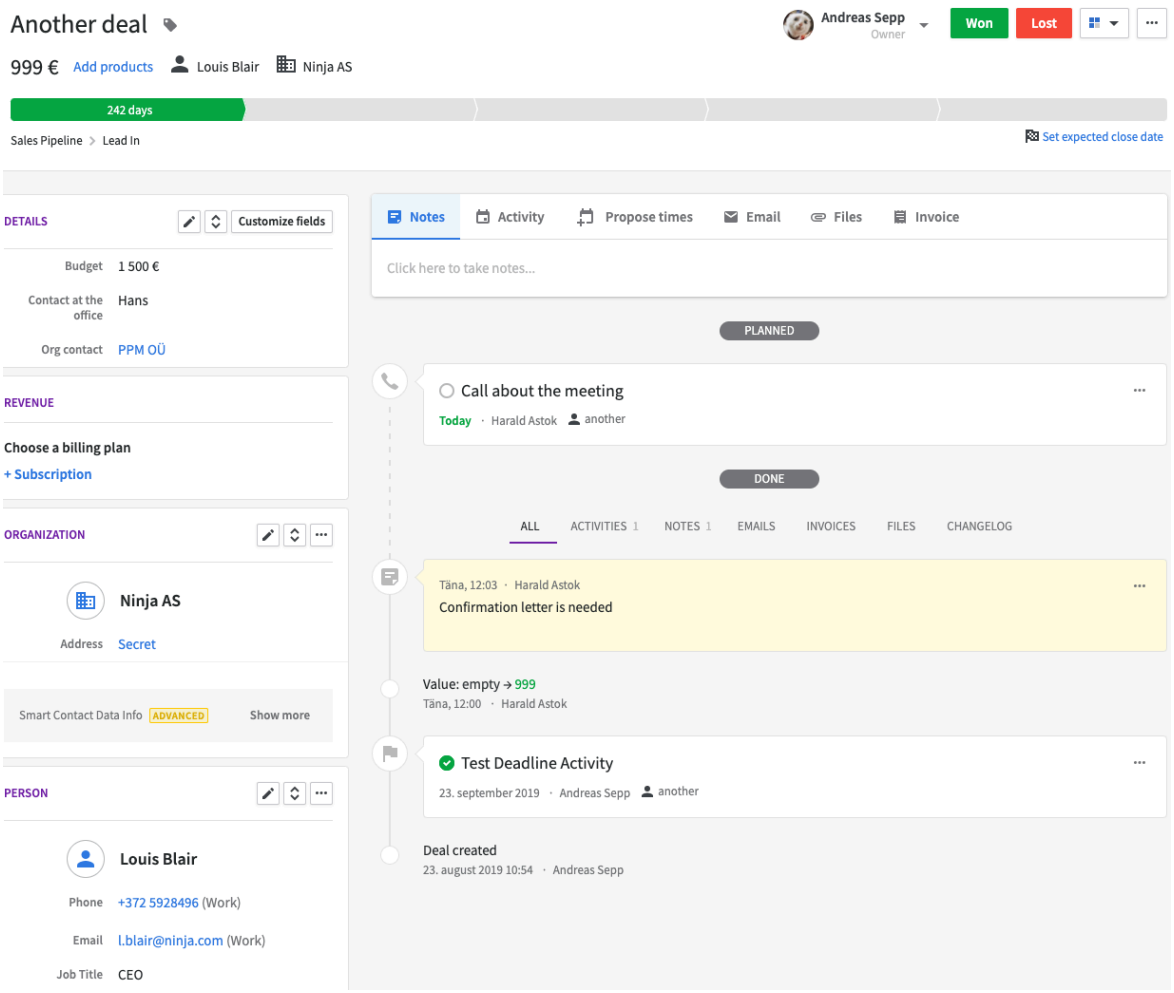

Figure 4 Detailed deal view

Users can also create person and organization contacts. Both can be linked with deals and persons can be linked with organizations. Organizations and persons are similar to each

other, as both have contact information, can be linked with other items and have custom fields.

In addition to persons and organizations, deals have also custom fields. Depending on what industry user chose when creating a Pipedrive account, some default fields are created, but all of them, default and create by user, can be viewed in one place under settings, as shown in Figure 5.

Field name	Type	Show in Add New dialog	Show in details view	Field API key
Secondary contact	Text	No	Yes	e0c042255cf1c4
Pank	Text	No	Yes	ce41efcad9921c
Acquisition	Numerical	Yes	Yes	25cedd7c91db74
Cash	Monetary	No	Yes	2c2a58ba6d02cb
Title	Text			title

Figure 5 Custom fields table

2.1.3 Searching data in Pipedrive

In Pipedrive, the term, used to describe overall search functionality, is called item search. “Item” in this context is used as a collective word when talking about deals, organizations, contacts, etc. and therefore “item search” is searching from said items. What values are searchable differs from item to item and is shown in Table 1.

Item	**Searchable values**
Deal	Title, notes, custom fields, file attachments names
Persons	Name, emails, phones, notes, custom fields, file attachments names
Organizations	Name, address, notes, custom fields, file attachments names
Products	Name, code, custom fields, file attachments names
Leads	Title, organization name, person name, email, phone
Mail attachment	File name

Table 1 Search values for items

When user searches from the UI and multiple results are found, the weight is calculated and matches are ordered from highest to lowest. Values in which the match has been found are then highlighted in the UI, as illustrated in Figure 6.

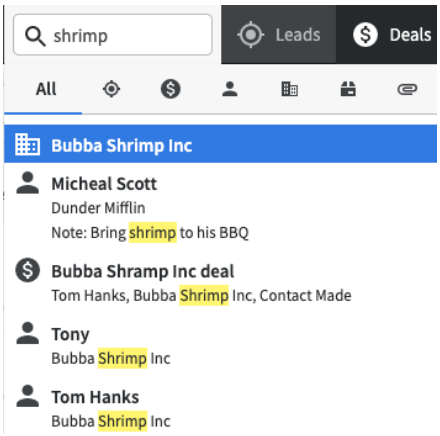

Figure 6 Searching from Pipedrive

2.1.4 Visibility rules in Pipedrive

Pipedrive has put a lot of effort and work into creating a product, where users have an option to choose which items are shared with whom and also an option to create custom groups. Visibility rules apply to items such as deals, contacts, products and they have 4 levels – owner only, owner's visibility groups, owner's groups and sub-groups, entire company. Descriptions of the levels are described in Table 2.

Level	**Description**
Owner only	Only the owner, admins and users in the parents groups can see and edit the details
Owner's visibility groups	Users in the same visibility group, admins and users in the parents groups can see and edit the details
Owner's groups and sub-groups	Users in the same visibility group, sub-groups, admins and users in the parent groups can see and edit details
Entire company	All users in the company can see and edit details

Table 2 Visibility levels

Admins can change the visibility for all items. They cannot directly control the visibility of a group other than moving that group around in the group tree hierarchy. As shown in Figure 7, if a user makes a group a parent of another group, that group's users sees all the items of child group users for example. Pipedrive users have an option to choose visibility groups to items, such as deals and contacts. Currently, a company that does not have advanced permissions feature enabled, there are two options for regular users – "Owner and followers" and "Entire company". If a user, who is in group 3, creates an item with the owner and followers' option, groups 1, 3, 5, 6 and 7 are able to search that new item. When entire company is chosen, every group can find the item. Visibility rules in search context are crucial to be implemented properly. Pipedrive is storing the visibility information in the Elasticsearch, so no further requests are necessary in order to determine what to display and what not.

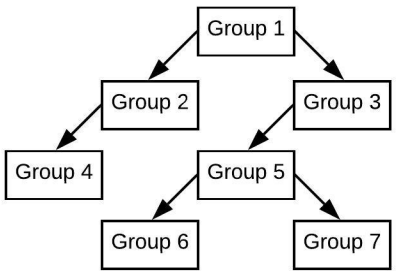

Figure 7 Visibility hierarchy tree

Company that has advanced permissions feature enabled, get 4 options for visibility groups – "Owner only", "Owner's visibility group", "Owner's group and sub-groups" and "Entire company". Owner's group and sub-groups have the same rules as owner and followers' option. When an item is created by a user in group 3 and owner only group is selected, group 1 and the user who created the item can see it. With the owner's visibility group option, groups 1 and 3 three get the option of seeing the created item.

2.1.5 Item following

Users also have the possibility to follow items or users. If a user A is a follower of user B, then user A sees all the items user B has, even if they are private or the user B is in another visibility group. Similarly with following items, user A can follow deal C and see all the details. However, if user A is in group 6 and the user B and/or deal C is in group 3 in the

aforementioned Figure 7, then user A must be added by somebody who already sees the deal C or user B. Someone in group 1 or group 3.

3 Software used by Pipedrive

This chapter's subchapters are going to talk about different frameworks that were or are used by Pipedrive. Understanding their architecture and work principle is necessary to know in order to understand how Pipedrive's search service works. Apache Kafka, RabbitMQ and Redis are used to implement Elasticsearch usage in Pipedrive search architecture.

3.1 Apache Kafka

Apache Kafka is an open-source distributed streaming platform designed for managing event-driven architecture. At first, they might have one event and one consumer. In that case, there is only integration, but when company grows and there are three producers, each sending data to three consumers, there are now 9 integrations and things can get very complicated to build and maintain. As shown in Figure 8**,** Kafka can help by gathering all data streams into one place where consumers can source their data from. That place is referred to as the broker.

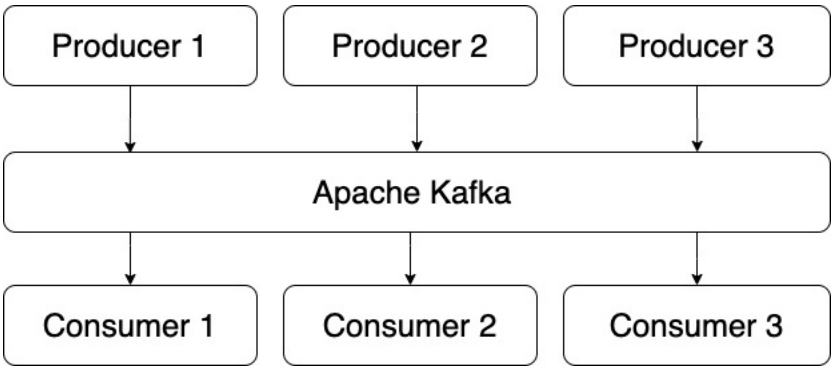

Figure 8 Kafka producers and consumers flow

Architecturally, Kafka divides data into topics and partitions (Sookocheff, 2015). Topic is a stream of data, identified by name and resembling sequenced storage of events. Topics are then split into partitions that are ordered and a message inside a partition is identified by an id or as it’s called, offset. Structure can be seen in Figure 9. Partitions are usually split to show high level concept of the consumer using Kafka, so that a consumer does not have to find information from several topics.

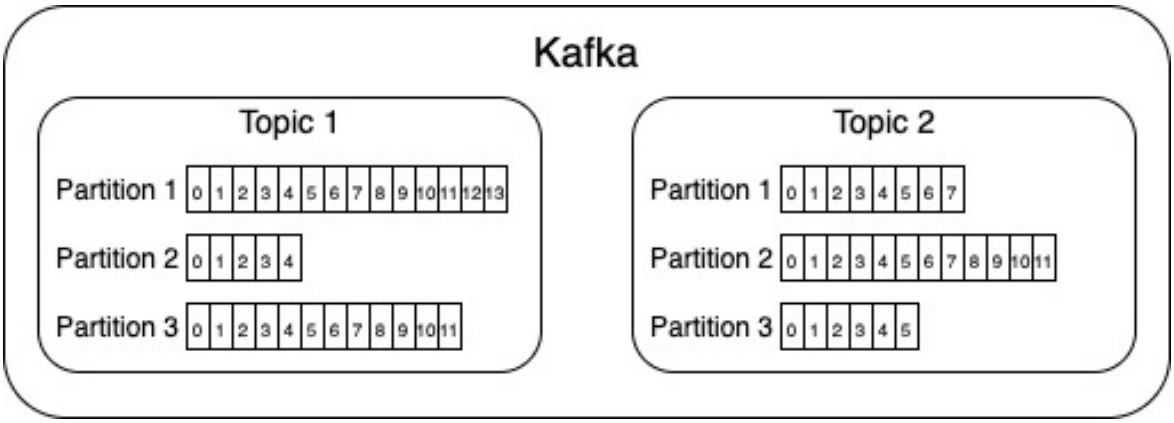

Figure 9 Kafka topic and partition architecture

Offsets in partitions are not interchangeable meaning that offset number 5 from second partition can not be used as offset number 5 in first partition (Kreps, Narkhede, & Jun, 2011). Also, data is kept only for a limited time defined by a user. Message ids are ever-increasing and once the data is written to Kafka, it is immutable. Oldest messages get deleted after they reach the time of deletion. Moreover, all of the messages sent to Kafka are guaranteed to be consumed at one point. It is also possible that some messages are consumed more than once. If a consumer is consuming messages and it should suddenly crash not delivering the next offset to the broker, then when consumer comes back up again, it continues from the last known offset consuming messages it has already consumed.

All topics are replicated and have replication factor of 3. This means a copy of the entire topic data is stored in two additional Kafka brokers and helps provide high availability as shown in Figure 10. Each topic has its own leader Kafka broker. Leader handles all its topics connections for messages production and consumption. In case of leader failure, one of the in-sync replicas takes over automatically (Apache Kafka, 2020).

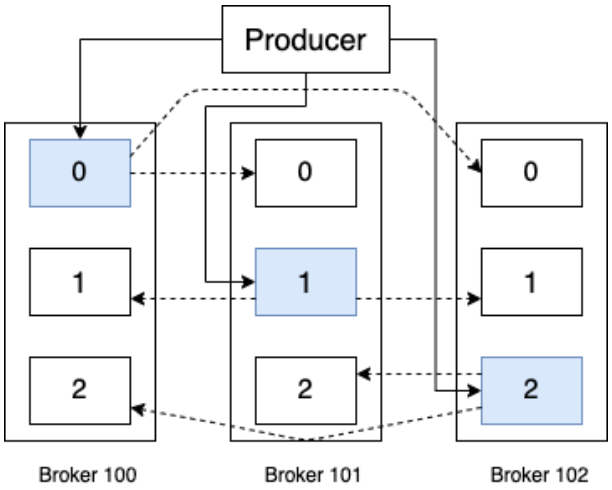

Figure 10 Replications in brokers

3.2 Kafka in Pipedrive

In Pipedrive, Kafka is used in variety of services. There are three main databases that send messages to Kafka, if a change is made to any one of these. In addition to search, Pipedrive uses Kafka for identity service data sync, which handles most of the request regarding users' settings and information from database; async data sync between different data centre databases; most async data exchange between services and provides reliable change events to databases via Debezium. Debezium is used to listen database changes via its binary log file and send those messages to Kafka.

3.3 RabbitMQ

RabbitMQ is an open-source message broker system written in Erlang (Jones, Luxenberg, McGrath, Trampert, & Weldon, 2011). It is used by distributed workers to queue messages for processing or to let applications share data via common protocol (Rostanski, Grochla, & Seman, 2014). The system supports many messaging protocols, STOMP and primarily AMQP amongst them. RabbitMQ producer and consumer flow is the same as Kafka's, shown in Figure 8. However, the two have a very different internal architecture. AMQP takes a modular approach, dividing the message brokering task between exchanges and message queues, as shown in Figure 11.

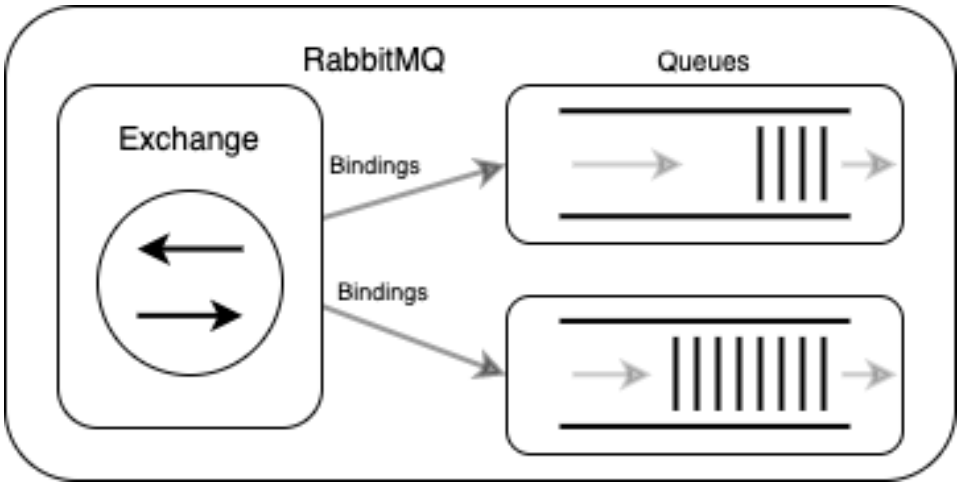

Figure 11 RabbitMQ exchange and queue architecture

Exchange acts as a router that receives messages coming from the application and routes those messages to queues, based on a set criterion. A queue stores messages and sends them to the message consumers (Dobbelaere & Esmail, 2017). Links joining exchanges with queues or other exchanges are called bindings. Inside bindings there are specific rules and criteria by which messages are routed and binding key. There are different exchange types the message can get to the queue. Each message sent comes with a routing key which determines the exchange type and to which queue the message is sent. (Johansson, 2019)

RabbitMQ is designed to be much closer to the classical messaging system, it handles messages in DRAM memory and the queue logic is optimized to work on short or near empty queues.

3.4 Elasticsearch

Elasticsearch is an open source search engine based on Apache Lucene. As shown in Figure 12 a running instance of Elasticsearch is called a node and one or more nodes can form a cluster. Data in Elasticsearch is stored in indices. When comparing it to SQL elements, an index is similar to a database table, which contains documents that are similar to table rows. When compared to the database terminology, a document corresponds to a table row and a document type represents a table (Sookocheff, 2015). Documents themselves are JSON objects. Inside those JSON files there are fields, which correspond to table rows, i.e., actual data that is searchable. Indices contain shards, which are basically subsets of documents of an index, so that an index can be divided into many shards. It is also possible to add replica

shards into different nodes to protect the node against data loss in case something happens to the node (Paro, 2015).

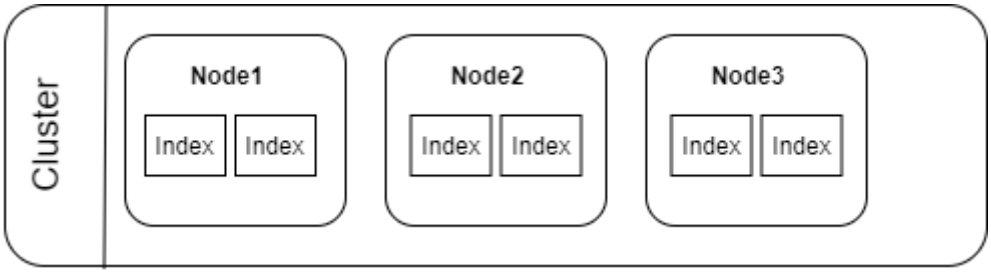

Figure 12 Elasticsearch cluster architecture

Some of the key benefits of Elasticsearch, according to Learning Elastic Stack 7.0 (Shukla & Sharath Kumar, 2019), are that Elasticsearch is very fast, easy to operate and scale, well documented and the user can easily use other elastic stack products, such as the data visualization tool Kibana with Elasticsearch. Also, it is possible to set up Elasticsearch in the Elastic cloud – a server for hosting clusters, using Metrics for monitoring and visualizing metrics or getting logs with Logs.

3.4.1 Why to use Elasticsearch

One of the main reasons Elasticsearch is favoured amongst many is its scalability. For example, if there is a research that at some point in the future is able to produce lots of results at the very beginning, when there is not much data yet, using relational database is justified since the data can be stored in a single database. However, when additional data is added, the database must be scaled vertically and more server power is needed. Horizontal scaling can be obtained by dividing the data among different servers, but that would be extremely complicated and not a good long-term solution. Elasticsearch divides shards between nodes automatically by constantly checking that the load is distributed evenly. So, when more data is expected to be added in the future, there is no need to worry about any scaling problems.

When updating (adding, changing or deleting) data from a database, the bigger the database, the longer it takes. The same is for maintenance. Elasticsearch scales up more efficiently performance wise than traditional databases. That also means that there is no service-wide downtime during maintenance.

In addition, when it comes to the architecture, databases must have schemas when created and when the work involves big data and there is a need to change the schema, it might take very long time to do it. Also, when a database contains lots of optional values, lots of disk

space is wasted for NULL values. Elasticsearch on the other hand does not impose schemas in documents and therefore contains only necessary data. (Kononenko, Baysal, Holmes, & Godfrey, 2014)

Another factor why people choose to use Elasticsearch is its performance. When designing a database for larger applications, data is usually divided between different databases for horizontal scaling. That usually improves 'create', 'delete' and 'update' requests, but makes 'select' requests complicated due to the fact that there is often a need to ask for information from multiple databases (Henzinger, Motwani, & Silverstein, 2002). When a client makes a request to Elasticsearch, the API request is sent towards any node that might have the needed data and response is sent to client. The whole flow can be seen in Figure 13.

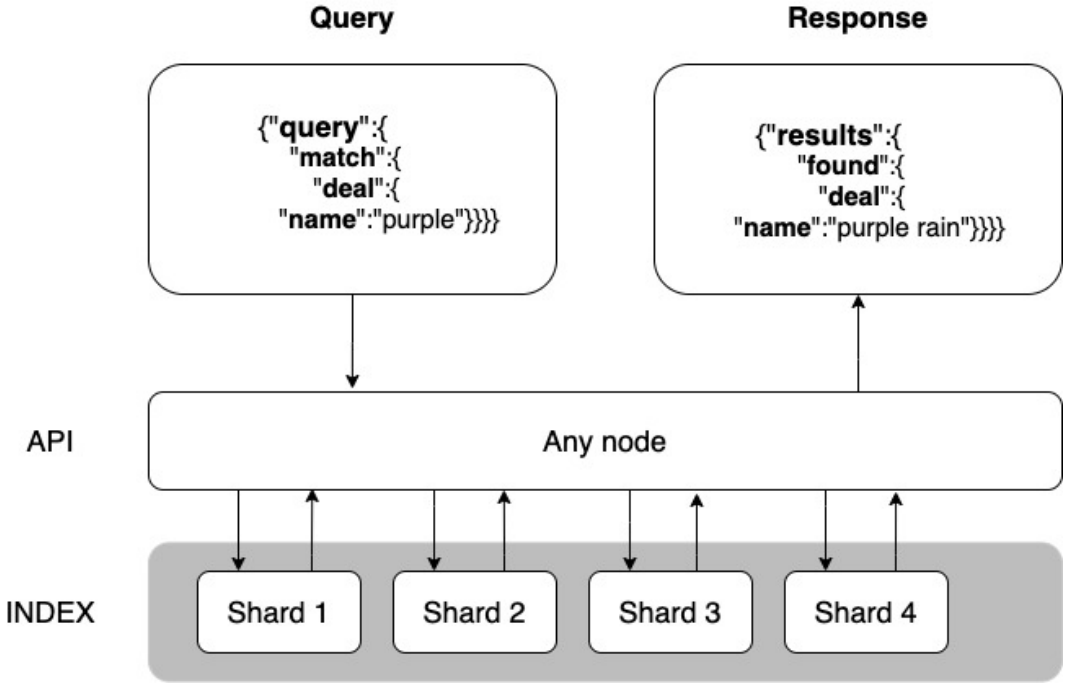

Figure 13 Query flow through Elasticsearch

3.5 Lucene

Lucene is an open-source Java library for efficient text-based search. Elasticsearch is a web server, which is based on JSON and built on top of that Java library. The difference between them should be recognised as it is relatively easy to mix them up or even think that Elasticsearch is an entirely separate product. Both are open-source and people can contribute if needed.

As illustrated in Figure 14, what Lucene does is that it takes the document, splits it into words and creates an index for each word. That index contains the word, frequency of the

word, what documents have that word and its position inside the documents. For the reason that each index has one word to match with the search term and it does not have to search through the text directly, the exact match can be found very quickly, giving it advantage over other search technologies. (Białecki, Muir, & Ingersoll, 2012)

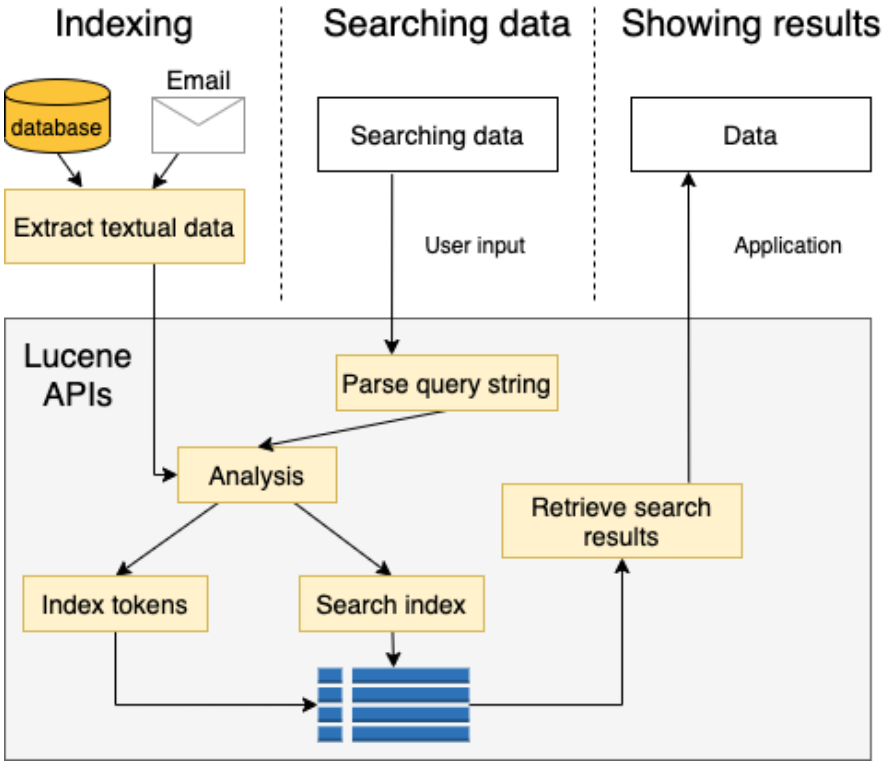

Figure 14 Lucene architecture

3.6 Redis

Redis (Remote Dictionary Server) is an open-source in-memory key-value store that can be used as a cache, database or message broker. It supports various kinds of data structures such as string, integers, hashes, lists, etc. (Redis, 2020) Within Pipedrive Redis is used for caching data for quick access.

4 Elasticsearch in Pipedrive search

With the product size of a Pipedrive, it has to have a proper and scalable search backend. Pipedrive has to serve 6,5 million search requests every day. This chapter gives an overview of Pipedrive's search microservice architecture and each microservices' responsibilities.

In Pipedrive, the service that handles everything related with UI is named “webapp” and the monolith is named “phpapp”, since it is written in php. In this chapter, all of the iterations of Pipedrive search architecture, which have used Elasticsearch, will be described. The author of this thesis has been part of the continuous development process since Elastic 5.

4.1 Elasticsearch 1.8

The first Elasticsearch based search backend used version 1.8 and was implemented around 2016 - 2017. As shown in Figure 15, Pipedrive did not use any data streams for the messages or a service in-between to handle any kind of queues. All of the messages that came in, were processed in the monolith and sent directly to Elasticsearch.

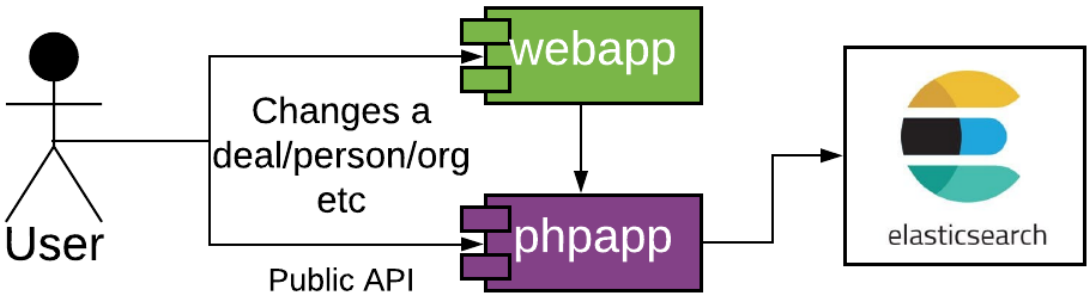

Figure 15 Elastic 1.8 architecture

With Elastic 1.8, every company had their own index inside Elasticsearch, which required Pipedrive to have over 15 servers with 3 clusters inside each server. The decision was made based on a database schema, where all the companies were divided into separate databases. Since it worked for the database, it was expected to work with Elasticsearch as well.

The whole system was extremely fragile in a way that if there was an error to writing into one index, it was probable that it would bring down the entire cluster. Companies were not divided into clusters by their size, so the size of the clusters was irregular and managing them became difficult very soon. Back then, there were a lot less requirements for search, so having one index for one company made sense. For example, visibility rules were

simpler, there were not as many features (multiple phone numbers and emails, leads, etc.) and not that many features were searchable.

4.2 RabbitMQ based Elasticsearch 2

When Pipedrive started using Elasticsearch, search was not a microservice, but a part of the monolith. Second iteration of the search architecture started using Elasticsearch version 2. As seen in Figure 16, when a user changed something in the UI or through public API, a message was sent to the monolith which created two events. One was sent to the database to make the necessary change there and the second message was sent to RabbitMQ to be processed and sent to Elasticsearch. It was suggested that RabbitMQ should get the messages directly from the database, but it was thought to be a security risk to have third party service hold a direct connection to Pipedrive's customer data.

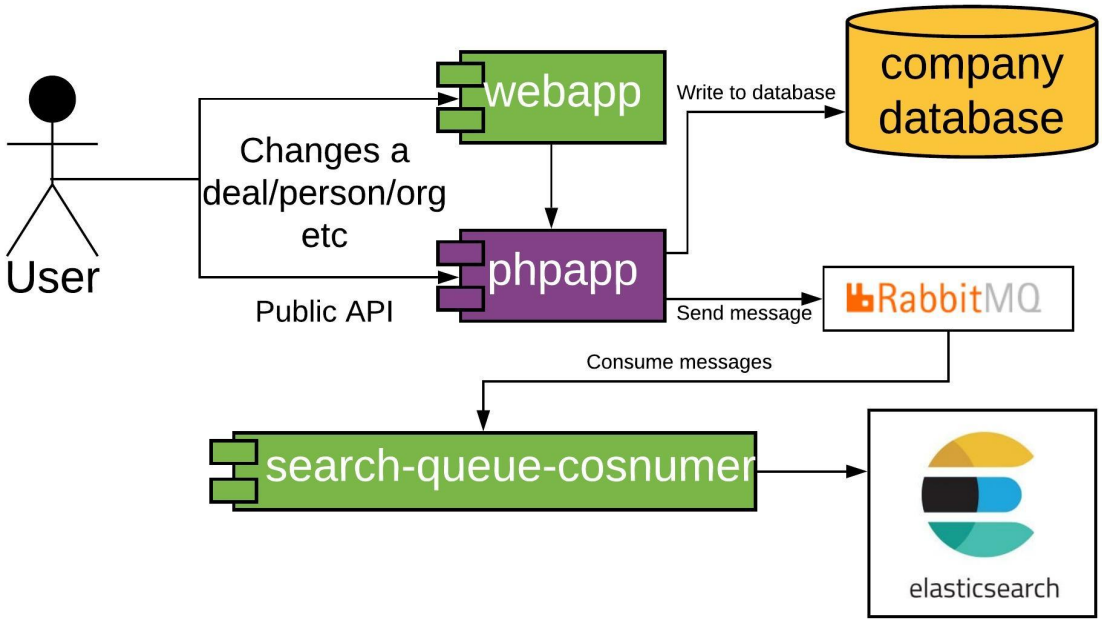

Figure 16 RabbitMQ based search

When user made a search request from webapp or from public API, all the documents matching the term were pulled from Elasticsearch and after they were post-processed in monolith.

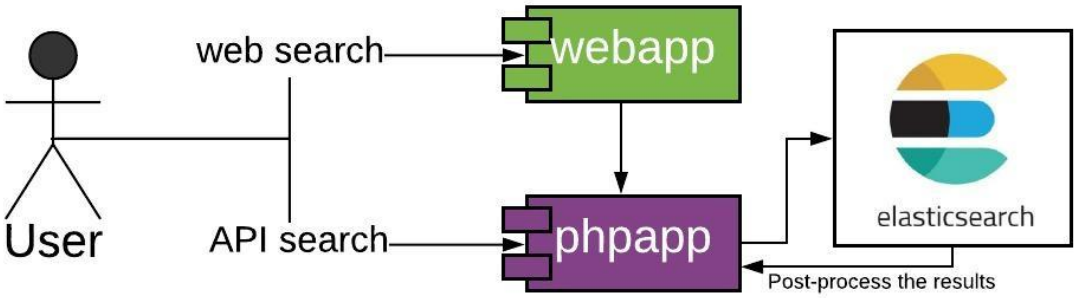

Figure 17 Retrieving data from Elasticsearch

That kind of flow was good enough at the time, but it was not scalable in the long term. Soon after going over to RabbitMQ and Elasticsearch based search backend, users started to complain about some items not being searchable. When investigating, it was discovered that some items never got to Elasticsearch and there were inconsistencies between database and Elasticsearch. As mentioned before, when a user did something that caused a change in the database, two messages were sent by the monolith to RabbitMQ and Elasticsearch in parallel. There is a fundamental flaw in it - if one operation should fail, there is going to be inconsistency between the two. The inconsistent part was always the flow to Elasticsearch, which showed that the problem was with RabbitMQ. Mechanism that sent RabbitMQ message was "fire and forget", meaning that monolith did not get a confirmation that the message sent actually got to the broker, actually If there was a connection error or some other reason why the message was not received by the RabbitMQ, monolith had no way of knowing it and assumed that everything was fine. What also happened was that monolith saved the new offset and consumption continued from the wrong offset, meaning that some data that was sent to RabbitMQ never made it to Elasticsearch. With a growing number of customers, complaints to the customer service about items being not searchable became so frequent that without further investigating, the first step for customer service was to rebuild index from Pipedrive side and see if that solves the problem. And mostly it did.

4.2.1 Phone numbers

In addition to inconsistencies, the old search suffered from an array of problems that were not solvable with the current architecture. For example, phone numbers were treated as plain text and user could not find "*55 123 456*" if it was stored as "*+37255123456*". A contact could have multiple phones, but only the first phone was searchable. That was fixed by preprocessing phone numbers before they are sent to Elasticsearch document. When a user

enters a phone number, then an area code, spaces, parentheses, hyphens or underscores are removed and only clean phone number is saved. In addition to cleaned number, if a country code is detected, the international number is saved as well. For example, as shown in Figure 18, if a user enters “+1 (553) 43 90_98”, it is analysed and “553439098” is saved as a local phone and “+1 (553) 43 90_98” as a regular phone number in the Elasticsearch document.


```
{
    "_index": "person",
    "_type": "_doc",
    "_id": "7375052:10",
    "_score": 1.0,
    "_routing": "7375052",
    "_source": {
        "email": [
            "a.j@smith.com"
        ],
        "phone": [
            "+1 (553) 43 90_98"
        ],
        "phone_local": [
            "553439098"
        ],
        "company_id": 7375052,
        "visible_to": 3,
        "owner_id": 11118992,
        "custom_fields": [],
        "name": "Angelina Jolie",
        "id": 10,
        "add_time": "2020-05-06T13:54:47Z",
        "followed_by": [
            11118992
        ],
        "owner_group_id": 1
    }
}
```


Figure 18 Person document inside Elasticsearch

Having 2 fields for numbers allows Pipedrive to cover all of the use-cases related to searching phone numbers. Similar pre-processing is done for the search term. If user tries to find previously entered phone number and inserts “01 553_439 098”, then it is transformed to “01553439098” which matches with the international phone number and “553439098” with the local number. When “0” is the first number, it is treated as “+” prefix before the area code and therefore are equal to each other.

4.2.2 API events

API events did not have the full data to support custom fields searching, so Pipedrive had to do an additional database query for each RabbitMQ message received from that company's database to get all the necessary data about their custom fields. This was not scalable

and only enabled per-company when they requested it. The same thing occurred when customer wanted to search notes. Pipedrive did not apply any visibility rules in Elasticsearch and did post-filtering on Elastic results. If a user had a very limited visibility, in order to get 100 search results using a wide search term, service would potentially have to scroll through 10 000s of elastic results and after post-filtering 100 results could be returned for the user.

4.2.3 Pagination

One of the biggest problems emerged with pagination. Search supported searching for related items as well. If persons or organizations were found from Elasticsearch, additional query was done to the company's database to also get all the deals related to those persons and organizations, appended them to the end of the result. This caused the pagination to be flawed when user requested 100 items but actually could get 100+ items. Additionally, custom fields and notes data was stored in a separate Elasticsearch index. If 100 documents are requested from Elastic and they included a deal document and 3 notes documents for the same deal, 4 results would have to be merged into 1 and would only return 97 documents to the user, which caused further pagination inconsistencies.

4.2.4 Notes

Notes can be added to deals, organizations and persons and can be seen in the detailed view of the deal in Figure 4. As shown in Figure 19, notes document inside Elasticsearch contained note_id, parent_type and parent_id (lines 8-10). note_id was used to retrieve the content of the note from database, parent_type and parent_id were used to query from the correct Elasticsearch index and to build database query.

```
1.  {
2.     "_index":"note",
3.     "_type":"_doc",
4.     "_id":"62:2",
5.     "_score":1.0,
6.     "_routing":"62",
7.     "_source":{
8.        "note_id":2,
9.        "parent_type":"deal",
10.       "parent_id":17
11.    }
12. }
```

Figure 19 Notes document inside Elasticsearch

Since notes were stored in their own index in Elasticsearch cluster, it created a situation that when a search term had a hit with the notes, additional queries had to be done to their parent

item. When notes are stored in their parent item and there is a match with notes, all the additional information about the parent item that is needed to display results, is already in the document, where the note was matched. When comparing Figure 20, where the deal document inside Elasticsearch is shown and comparing the results shown in Figure 21 by the UI, the only additional information that is needed is the "*Smith*" organization, which is related to the deal and the stage of the pipeline the deal is in. Stage is not saved in the Elasticsearch because moving deals between stages is the most common activity by users and rebuilding a document every time is not optimal. Not only does that kind of mapping save space on the server, but it reduces the request time by not needing any additional queries from the parent item.


```
{
    "_index":"deal",
    "_type":"_doc",
    "_id":"7375052:13",
    "_score":1.0,
    "_routing":"7375052",
    "_source":{
       "id":13,
       "company_id":7375052,
       "title":"Caterer",
       "visible_to":3,
       "status":"open",
       "add_time":"2020-05-06T13:55:54Z",
       "owner_id":11118992,
       "custom_fields":[
          "White or black uniforms"
       ],
       "followed_by":[
          11118992
       ],
       "owner_group_id":1,
       "notes":[
          "Chef has to be picked up"
       ]
    }
}
```


Figure 20 Deal document inside Elasticsearch

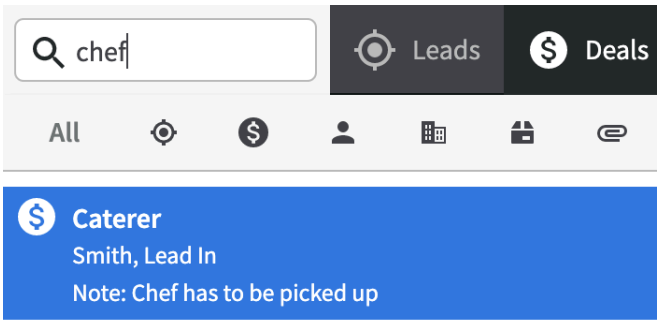

Figure 21 Search result when matching a note

All of those issues were addressed when a new architecture was planned.

4.2.5 Inconsistencies

One of the biggest problems Pipedrive had with search was the data inconsistency between database and Elasticsearch because of non-atomic write operations. Why it happened is described in more detail in Chapter 4.2. due to the inconsistencies caused by RabbitMQ not being reliable and not consuming all of the messages sent by the monolith, users search or autocomplete results varied because of different data sources and inconsistency. Therefore, the architecture was changed, and Kafka usage was implemented, as shown on Figure 30.

4.3 Kafka based Elasticsearch 5

In 2018, the decision to revamp all of search architecture was made. From this point, the thesis's author joined the search development team and worked on it nearly continuously until 2020. When going over to new systems the main goals were to upgrade Elasticsearch cluster version from 2 to 5, migrate all companies to use Kafka synchronization, apply new generation email domain search and make custom fields searchable for everybody. Main motivation behind it were unhappy customers who complained to customer support that they could not search something and customer service had to rebuild customers Elasticsearch index. Those kinds of complaints took up lots of engineers and customer support's time and it was decided that having proper and reliable search backend would have huge positive impact on the user experience and improve the Pipedrive as a product as well. That also meant bringing search logic out of the monolith and creating microservice architecture.

In Pipedrive, customers data is divided between company database – each company has their own database. To each of these company database servers Debezium was installed, which listens to the database's binary log file and produces Kafka events to ensure data

consistency. Whenever anything was changed in the source of the information – the database – it was propagated to Elasticsearch as well.

When users wanted to search from custom fields, they had to ask from customer support who then turned on the feature flag and manually rebuild their index along with custom fields values. With the RabbitMQ based Elasticsearch 2, it meant that processing every single RabbitMQ event from that company required an additional call to their database, which was not scalable.

Formatting phone numbers needed improvement because, previously, all the possible ways how users write phone numbers were not taken into account (with or without area codes, 0 instead of + in area codes, spaces in-between). Finally, current solution didn't have wildcard prefix support. Wildcard prefix gives the opportunity to find partial match, so for example "$ki*y$" matches with "kiy", "$kitty$" and "$kimchy$" (Elastic, 2020).

Previously, only text based custom fields were searchable, but the goal was to make numeric custom fields and optional fields searchable. Soon it became apparent that some of the fields were not going to be searchable, for example multiple choice fields, because of the way they were saved into database. In order for it to be possible, changes in custom fields database schema had to be implemented.

Finally, all visibility logic that is done in databases also needs to be replicated on Elasticsearch. The visibility logic is heavily relational - to verify the full visibility of a deal, a check to deals, such as deals_sharing, deals_follow, users_follow, permission_set_assignments, role_assignments tables needs to be made. Part of it can be provided before the Elasticsearch query from the database - user is admin, user followers, user group id, user parent group ids, user child group ids. In Elasticsearch documents, each item is stored with item id, item owner id, item owner group id, item follower ids. Cross-matching that with the fields from the initial database query, it can be determined if it is visible or not.

On Figure 22, the javascript code for generating part of the visibility query for Elasticsearch is shown. The business logic behind it is complicated as discussed in Item following chapter.

```
return or(

    // The item is globally visible
    is('visible_to', GLOBAL),

    // The user follows the item directly
    is('followed_by', userId),

    // The user follows the owner of the item
    any('owner_id', userFollowings),

    // PRIVATE- The item is made in the user's child groups or by the user directly
    and(is('visible_to', PRIVATE),
        or(is('owner_id', userId), any('owner_group_id', userRoleChildIds))),

    // SHARED - The item is made in the user's group or child group
    and(is('visible_to', SHARED),
        any('owner_group_id',
        [userRoleId].concat(userRoleChildIds))),

    // SHARED_BELOW - The item is made in the user's group, child group or parent grou
    and(
        is('visible_to', SHARED_BELOW),
        any('owner_group_id',
        [userRoleId].concat(userRoleParentIds, userRoleChildIds)),
    ),

    // files visibility is determined with post-processing
    is('_index', 'file'),
);
```

Figure 22 Creation of visibility query

4.4 Kafka based Elasticsearch 7

When Elasticsearch 5 development was finished and rollout from Elasticsearch 2 to Elasticsearch 5 to customers began, several issues became apparent. There were three main features that for various reasons failed to be implemented. Firstly, the numeric custom fields were not searchable. Secondly, notes were not searchable for non-admins and finally, sometimes weighing of search results buried the relevant matches due to many matches from custom fields, which were not searchable anymore. Elastic 7 based search combined all the features from both, older versions and the last 5% were ready to be migrated. As mentioned before, notes were stored in a separate index and it was decided that notes should be moved under the parent item, so that no additional queries would have to be done to get all necessary information. Furthermore, it was determined to only have maximum of 20 notes per deal/person/organization to be searchable. This is because due to the way Elasticsearch is built, any change to a document causes the whole document to be reindexed, which is a slow operation. Attaching too many things to a single document because it would be too slow. If all of the notes would be searchable, they would potentially start to use a significant amount

of memory because the size of the note is limited to 3MB (Pipedrive Developers' Corner, 2020). Having a limit of 20 notes was chosen because only 2% of companies have more than 20 notes.

In addition to missing features some additional issues needed fixing. Topics inside Kafka are partitioned based on company ids. Every topic has 32 partitions, meaning that one partition is shared between ~3.1% of all companies. If one company did a large amount of changes in their database, usually a bulk editing of all deals, persons, etc., would clog up the Kafka partition with their data. Kafka cannot consume the massive number of messages fast enough and the other 3.1% of companies on the same partition cannot search their newest data until it is resolved. Usually it auto recovers in ~30 minutes, but it still triggers several alerts and on-call engineer has to make sure if it is a false positive alert or something deeper. That issue was resolved by closely monitoring the total amount of unprocessed messages, then blacklisting those bulky companies until the queue is consumed and after then migrating their data to restore the data consistency in Elasticsearch.

Another challenging issue became apparent when search was being decoupled from the PHP monolith to a microservice. Pipedrive's public API had 5 search endpoints, which did similar things but had different response formats and provided fields that could no longer reasonably be provided after splitting the service from the monolith. Furthermore, over the years, users asked for new features from the public API and along with product requirements changing, all kinds of hacky changes were made to public API. For example, there was an endpoint */persons/find*, which was designed with the aim of searching for persons by name and returned an object shown on Figure 23. By popular demand, an argument search_by_email was added to search by email instead. When this argument was used, the endpoint response format also changed for an unknown legacy reason, as shown in Figure 24. Such problems really piled up and it was decided to revamp the public search API as well to be more future-proof and extendible.

```
{
    "id": 5,
    "name": "Example person",
    "email": "primary@email.com",
    "phone": "55123456",
    "org_id": 2,
    "org_name": "Example organization",
    "visible_to": "3",
    "picture": {
        "id": 1,
        "url": "https://...",
        "width": 512,
        "height": 512
    }
}
```

Figure 23 /persons/find response when search_by_email = false

```
{
    "id": 5,
    "name": "Example person",
    "email": "primary@email.com",
    "phone": "55123456",
    "org_id": 2,
    "org_name": "Example organization",
    "visible_to": "3",
    "additional_emails": [
        "secondary@email.com"
    ]
}
```

Figure 24 /persons/find response when search_by_email = true

Unfortunately, since search public endpoints are amongst the most used endpoints, it is not possible to just delete old them and tell users to use the new ones. Currently, the plan is set to attach deprecation warning to the response of old endpoints, which directs users to public documentation of Pipedrive API. After a few months those endpoints are removed from documentation, but they are still usable. Pipedrive keeps monitoring the usage over time and, hopefully, those endpoints can be deleted in the near future. Meaning that lots of code from monolith can also be deleted. Pipedrive web app has also used the general search endpoint, but since web search does not utilise a lot of the information fetched by that, it was concluded that web search should have its own endpoint, which only retrieves data that is shown in the search results.

The final problem was that search backend itself is very intricate and has over 200+ use cases. Unit testing them was not good enough, as there are so many components with a “big black box” (Elasticsearch) in the middle. Changing something minor can very easily break some edge use case. In addition, search functional tests were only runnable in development

environment and they were not very stable. To solve it, integration tests were created, that, as shown in Figure 25, are ran every time search or search related microservice codebases are deployed. Tests try to cover as much of the use cases as possible. New tests cover the flows of user changing the data, it getting to Elasticsearch and when migrating the data, making sure that gets to Elasticsearch.

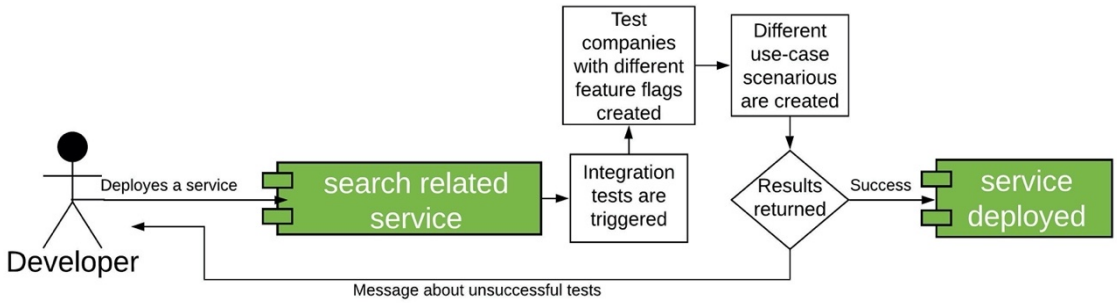

Figure 25 Search integration tests flow

4.5 Binary log for messages

Binary log file contains all the changes made in the database. Getting information from there gives user assurance, that whatever system or service listens to it, does not have the access to the full database. Using it with Debezium is a fool-proof method to get constant and reliable messages, it removes human error option from Pipedrive side as long as there are no major database schema changes. Having additional service in the flow creates a risk - if Debezium should go down, what would happen? So far, it has proved itself to be reliable and has not had any big outages over the last years it has been used in Pipedrive.

Database binary log is also used in other ways than getting messages to data streams. For example, web application can use it to store cache, monitor different aspects like load or make information available to other applications.

4.6 Scalable solutions for custom fields and notes

4.6.1 Custom fields

From Kafka based Elasticsearch 5, custom fields content is included in the Kafka events generated by Debezium straight from the database, so there is no need to do an extra query to the database for each message to fetch custom fields. This enabled Pipedrive to start providing searching from custom fields for everyone. However, there is still an existing problem where custom fields in the Kafka messages do not have a type attached and it is

impossible to know if a numeric value is actually an id reference to another object (where the id should not be searchable) or an actual numeric value input by the user that should be searchable.

There are plans to rework this architecture in the future to provide types of custom fields in the message itself, but until that, an alternative solution was implemented where Redis cache is used to store the types of each company's custom fields, so it could be used as a lookup for the type of the fields in Kafka messages to determine if the custom field is searchable or not. If the Redis cache happens to be down, as a fallback mechanism all custom fields are considered searchable because it is better to potentially have some seemingly irrelevant results in users search results than the items they are looking for to be missing from product point of view

4.6.2 Storing additional data with the parent item

Elasticsearch is not designed to support the relational data, there is no way to do a performant join between 2 indices. If relational data is needed, it has to be denormalized. Pipedrive did this for notes and custom fields by adding them as an array to the parent document (deals, persons, orgs, products). It was also applied to phones and emails to person documents. This is not too performant because adding each new entry has $O(n^2)$ complexity and every time any field in an elastic document is changed, the whole document needs to be reindexed. This caused to introduce upper limits to the number of items that can be in these arrays (20 for notes, 100 for custom fields, 25 for emails and phone numbers). This can cause problems where if user has a contact with a lot of notes/emails and can only find them by the first 20 or 25 entries.

4.6.3 New added use cases support

When rewriting most of the search backend, a lot of new use cases were added as well. For example, previously users could search emails only by the beginning of the email address and if the address ended with a non-country related domain, such as "$.works$", they were not searchable. Email addresses are now tokenized in a way that everything before "@" is split from the "." character and it is done for up to three times. Research showed that it is very unlikely to have an address with more than three periods in the first half of the email. For example, when user have an email "$john.doe.third@uni.works$", it creates following tokens: "$john$", "doe", "$third$", "uni", "$works$". When searching with "$@uni$" it is still

going to find it, because "@" is removed from the search term before the Elasticsearch query.

Furthermore, names of deals, persons, files, etc. get reversed tokens as well, so when user has 2 organizations "$Swedbank$" and "$Nationalbank$", both have tokens of their name in lowercase and reversed, "$knabdews$" and "$knablanoitan$". The aim of this it that when user wants to search for an item ending with "$bank$", he can find it. How it works is that when term "bank" goes to search backend, in addition to the regular term, also "$knab$" is queried from Elasticsearch. That way both "$Swedbank$" and "$Nationalbank$" are matched with query because of the reversed tokens.

Same was done with phone numbers. When user saves a phone number as "$+327\ (55)\ 123\ CALL\_NOW$", it gets stored in Elasticsearch as "$32755123callnow$", "$wonllac32155273$", "$+372\ (55)\ 123\ call\_now$" and is findable with "3725512", "$callnow$", "$CaLl\_nOw$" and "$+372\ [55]1\ 2\ 3call$".

4.6.4 Fixing pagination problem

Pagination was fixed in 3 parts. First step was for Elasticsearch to return fewer or equal number documents than number of search results. Because notes and custom fields were moved under their parent item, the amount of additional documents needed from Elasticsearch decreased. Second part was the post-processing of the results. With the visibility rules being moved to Elasticsearch documents, no additional elastic queries were necessary. Finally, having to search for related items from database had one of the biggest impacts on problems with pagination. That was solved when the search API separated Elasticsearch and related items results in the final results, making pagination to base only on Elasticsearch results.

4.7 Search microservices

Services listed below are the core services needed for the search architecture.

Database is not a service itself, but it does serve an important role in the system. When a user adds an item to the database, the change is saved in the database binary log file, which is listened by Debezium.

Debezium listens binary log files and produces Kafka messages. Kafka takes these messages and stores them in the right topic. Currently there are 13 topics that are of interest for the

search functionality. Topics are chosen by the tables from the database, which shows that without that kind of broker middleman and querying straight from the database, Pipedrive would have to constantly get data from 8 different database tables.

Kafka-search-queue-consumer (KSQC) is a service that listens to the 8 Kafka topics and sends them to Elasticsearch. Data coming into KSQC is processed according to their topic. Because each incoming items' data has a different format, it needs to be transformed according to the topic. One index can have multiple tables corresponding to it and KSQC gathers the necessary information from various database tables and combines it into a document which is going to be sent to Elasticsearch. For example, "person" index has 4 tables related to it: people, people_follow, people_sharing, data_extra. The flow of how 4 tables become one Elasticsearch document can be seen in Figure 26. Elasticsearch is a service where searchable documents are created based on information sent by KSQC and then retrieved by search service.

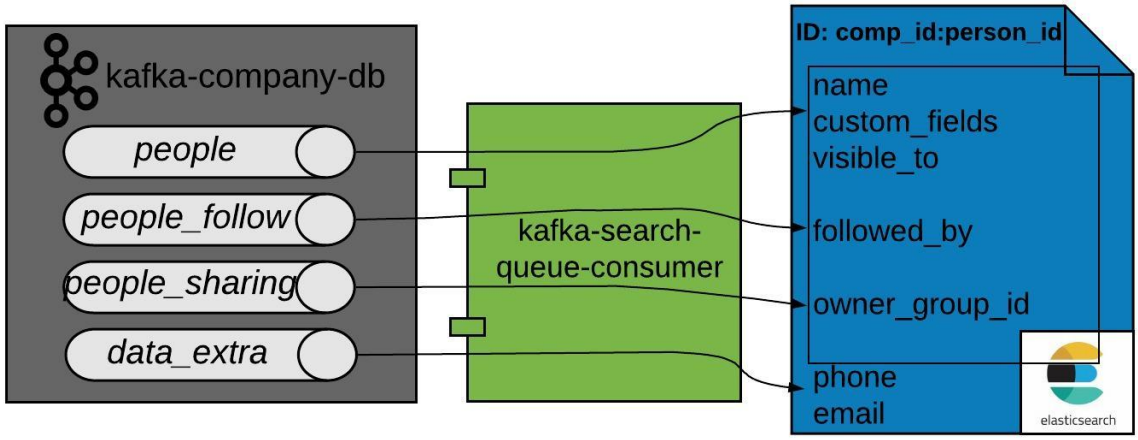

Figure 26 Document creation from Kafka messages

Search is the main service for the search functionality. All the calls from the webapp go through monolith to search service and public API calls go directly towards monolith and then to the search endpoints. Search service then takes the given parameters, such as search term and user information, to determine if visibility rules allow to return the found information. Currently, around 69% of the search requests come from webapp.

Phpapp (monolith) is the core service of Pipedrive. Even though for years the goal has been to reduce the usage of this service, it still holds some of the key components of Pipedrive and most of the public API requests are routed through it.

The following services are mainly used by developers in the development process, also for the monitoring purposes or by customer service to remigrate the Elasticsearch index.

In some cases, there is a need for a different way to get one company's data to Elasticsearch. For example, Kafka only has 30 days of data, anything older than that needs to be migrated separately or if a company might be moved from one datacentre to another and needs its data in the new datacentres Elasticsearch cluster. For those purposes elastic-migrator (EM) is created. EMs purpose is to remigrate company's data from database to Elasticsearch.

Search-mass-migrator (SMM) is used when there is a need for a large number of companies to be migrated. As illustrated in Figure 27, SMM takes a batch of company ids and starts requesting multiple instances of EM for the migration job. SMM has been used when Pipedrive was switching datacentres or if they are creating new Elasticsearch clusters.

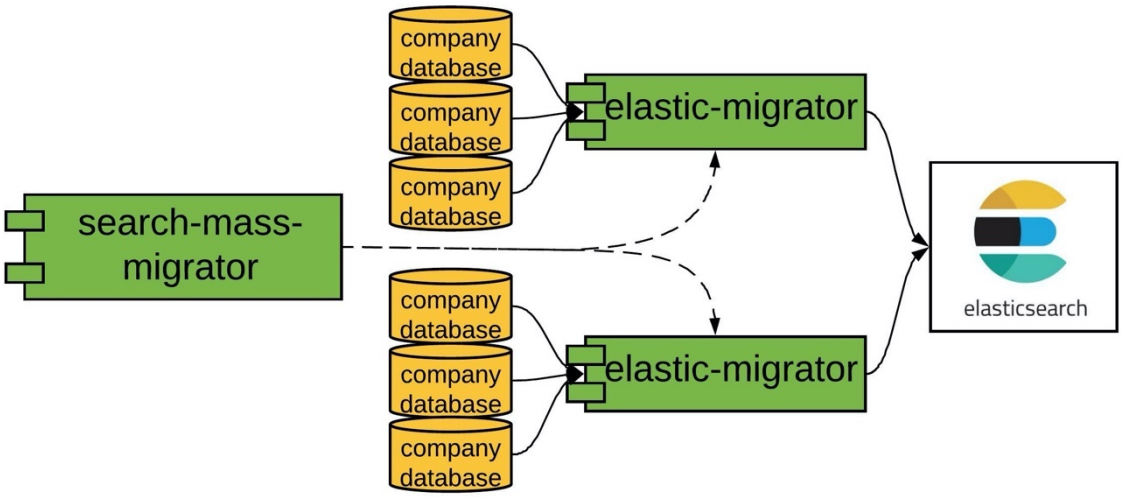

Figure 27 SMM and EM workflows

Search-consistency-checker (SCC) looks at the number of searchable items that the company has in its database, then how many documents it has in the Elasticsearch and, finally, compares the two results. It is mainly used in Pipedrive's backoffice system, where customer service can look at the data and remigrate the company if needed, as shown in Figure 28. As shown in Figure 29, the main functionality of the service is to return the number of each item in database and number of documents in Elasticsearch.

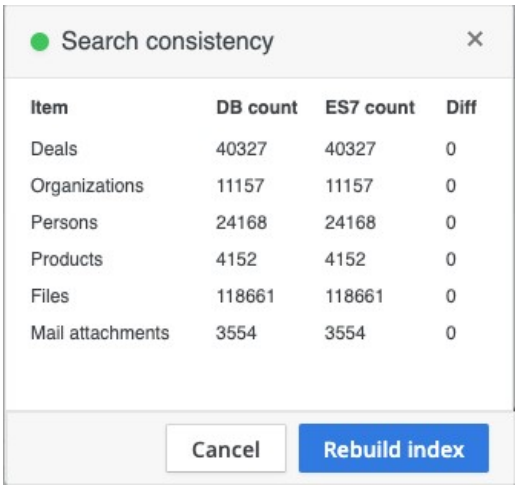

Figure 28 Consistency modal for backoffice


```
{
    "company_id": 62,
    "db_counts": {
        "deal": 2,
        "org": 4,
        "person": 4,
        "product": 1,
        "file": 3,
        "mail_attachment": 1,
        "note": 16
    },
    "es7_counts": {
        "org": 4,
        "person": 4,
        "file": 3,
        "deal": 2,
        "mail_attachment": 1,
        "product": 1
    },
    "es7_difference": {
        "deal": 0,
        "person": 0,
        "org": 0,
        "product": 0,
        "file": 0,
        "mail_attachment": 0
    },
    "in_progress": false
}
```


Figure 29 Search consistency check response

4.8 Searching with Kafka

As shown on Figure 30, the search architectural flow starts with webapp sending message to monolith and if a user did something to cause a change in the database, monolith produces

an event and then writes it to the database. All the changes made in database are saved in a binary log file which is being listened by Debezium. Debezium is a set of distributed services that capture row-level changes in your databases so that your applications can see and respond to those changes (Debezium FAQ, 2020). Those changes are then sent to Kafka where the topic is selected by the item type in which the change was made in.

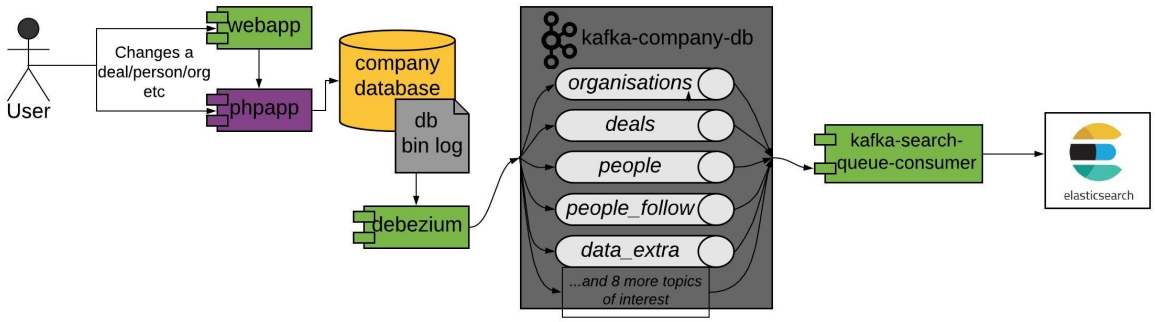

Figure 30 Data getting to Elasticsearch through Kafka

Actual searching is now built in a way that monolith is cut out from the flow. Queries from webapp and public API go directly to search service, which then send the request to Elasticsearch. Cutting out monolith was an important milestone for search service, so it would not have to rely on the monolith.

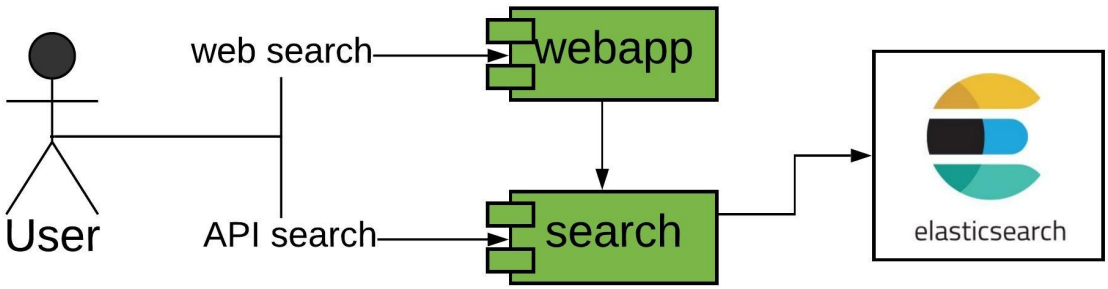

Figure 31 Searching flow in Pipedrive

4.9 Fixing inconsistencies with Kafka

The lack of reliability from previous data stream forced Pipedrive to look for a substitute and Kafka was chosen over RabbitMQ. Also, RabbitMQ is not actually a storage. For example, with Kafka, a consumer can get data which was produced weeks before consumer had been even created. However, with RabbitMQ this isn't possible. In addition, if the consumer is down for some time, RabbitMQ queue grows, as mentioned in Chapter 3.3, and it is not optimized for keeping a long queue. If the queue gets very long, RabbitMQ runs out

of DRAM memory to write messages and starts writing it to a disk without having a copy on DRAM. In such of case, there is a change that RabbitMQ can DoS itself. Kafka was specifically designed to be more resilient for keeping a longer queue.

4.10 Related items

Usually, deals have at least one organization or contact person linked to it. When a user searches a person, who is linked with certain deals, those deals also have to be retrieved and displayed, as shown in Figure 32. However, not all values for deals are saved in the Elasticsearch and some data has to be requested from database. That is done to reduce the load on Kafka. Because deals get moved around between different stages, their value and contact person or organization can change. Therefore, it is not optimal to rebuild an entire document for that. That is why some of that info is requested every time from database and Kafka can currently skip around 80% of the messages, because they are not relevant to Elasticsearch.

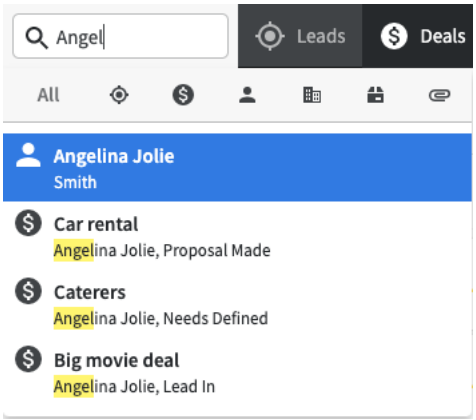

Figure 32 Search showing person and related deals

Searching those related items can be tricky if a user has set a complex multi-tier visibility ruleset. Without the proper implementation, there is a possibility of information leakage.

5 Author's contribution

Since December 2018, when planning for Kafka based Elasticsearch 5was initiated, the author of this thesis has taken part in every search services related mission that Pipedrive has had. That includes taking part in all of the planning stages, architecture and design discussions, implementation and validating. Search architecture is relatively small, so missions that started developing new microservices only needed three developers. Having such a small number of people working together gives an opportunity for each member to work on every aspect of the system, thus making them experts on the topic.

Search service was the first Pipedrive system that started using Kafka, which is why the whole process was highly experimental. It was set up previously as a proof of concept, but not a single service was actually using it. Being the first team who had an experience implementing the usage of Kafka in production, gave to all the members a very valuable knowledge and experience. Moreover, the information was also useful for all of the other teams within the company as it gave them an opportunity to also start using Kafka. The people who were already experienced, including the author of this thesis, were able to guide and help other teams through the process of implementing Kafka into their services. As a result, Kafka is used as a main data source for new events within Pipedrive, as described in Kafka in Pipedrive chapter.

Developing a search consistency checker, that is mainly used by customer service, became an area of interest for the author of this thesis. Data for modal shown in Figure 28 was previously sourced from the monolith. Since one of the goals from the beginning was to bring every part of search service out of the monolith, the author of the thesis implemented and set up consistency check into separate microservice. Additionally, the author created a cron job for the service, that runs 10 hours a day in both datacentres, during low load times and reports any companies that had inconsistencies between data in Elasticsearch and database.

One of the goals for Kafka based Elasticsearch 7 mission was to stop using monolith when searching. That gave developers an option to redesign the search API. The author of the thesis took upon himself to research and develop an endpoint that is used by the webapp.

Due to high volumes of search, monitoring and alerting systems had to be set up. Even though Pipedrive is already getting errors when something is not working, the author of this thesis took upon himself to create a dashboard of graphs, so developers can have a clear

overview of the service's health. Thanks to this new dashboard, a real time overview of search performance and quality can be accessed.

During the year and a half that the author spent on working with search service, he has learned a lot about the new technologies and got a lot of first-hand experience with them. Surrounded by an experienced team and commencing Kafka usage from scratch, meanwhile self-initiatively taking up challenging tasks upon himself to learn constantly, turned the author of this thesis an expert in search. As a result, he is now able to give onboarding sessions specifically about search for new developers joining Pipedrive and guide any teams who run into difficulties in that particular area.

6 Search metrics and monitoring

Having a constant overview of the services' health can be done manually, if the number of services is low enough. As there are hundreds of microservices in Pipedrive, doing it manually is not an option. Pipedrive uses Grafana as its monitoring application. Grafana is an open source visualization and analytics solution, where users can set up alerts and graphs to display their data (Grafana, 2020). In Pipedrive, engineers have set up dashboards for different services to have a clear overview of different metrics. They have also created an alerting system, as illustrated in Figure 33, to let on-call engineers know if some service is misbehaving and if the alerts go over the set threshold, automatic notification is sent.

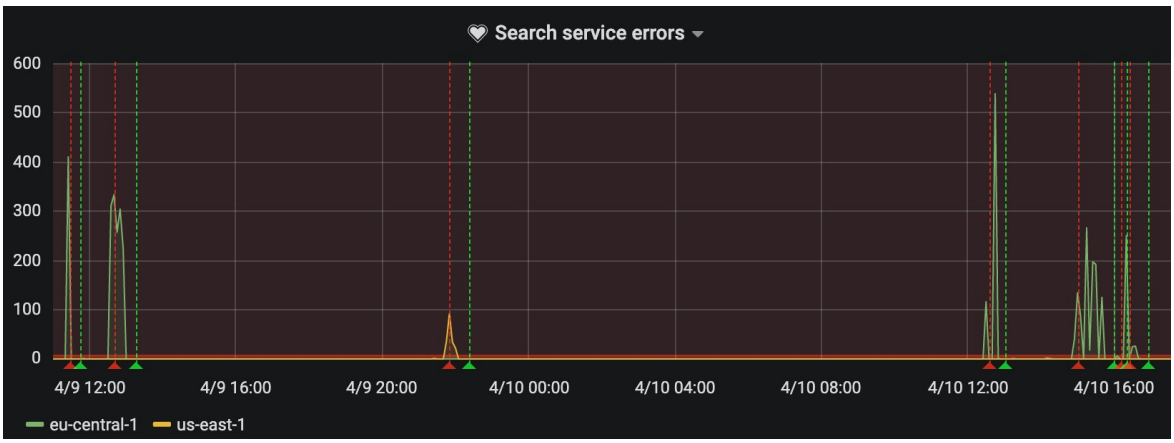

Figure 33 Live monitoring of search service

According to *Usability Engineering* (Nielsen, 1993), 1.0 second is about the limit for the user's flow of thought to stay uninterrupted, even though the user will notice the delay. Normally, no special feedback is necessary during delays of more than 0.1 but less than 1.0 second, but the user does lose the feeling of operating directly on the data (pp. 135). Therefore, Pipedrive has developed a *blueprint* for the response times for different scenarios as illustrated in Figure 34.

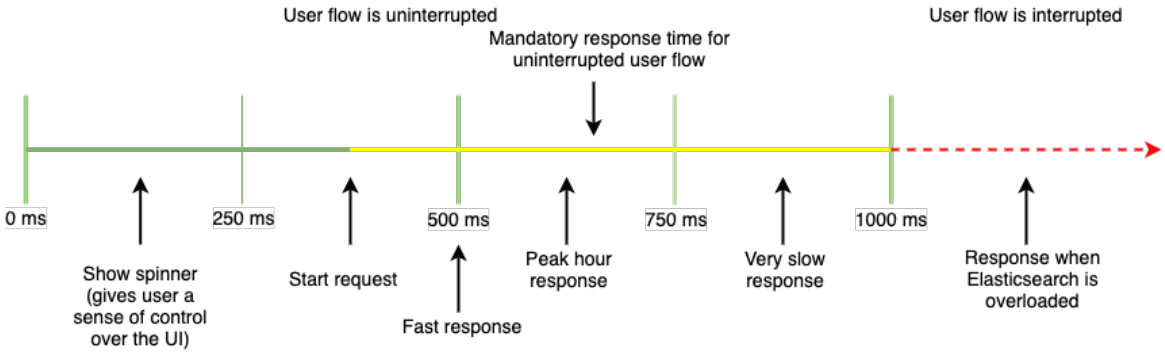

Figure 34 Response time flow

With a low usage, response time should be approximately 150ms. When usage is very high (peak hours are usually 14-15 UTC), response time has to be approximately 250-300ms and if the Elasticsearch is having a trouble processing the high load, response times are going to be 500ms to 4000ms. Looking at the graph, a question, such as why is it that the request is started at 350ms, might arise. The simple explanation for it is that the user has time to insert about 4-5 characters, which, consequently, results in a more precise search term. By having the delay at 350ms and assuming a normal distribution, about 50% of the requests are expected to be debounced. 350ms was chosen by trial and error after looking at the data.

When the search endpoints switched over from routing through the monolith, one of the biggest concerns was whether search microservice can handle the load and if the search query time goes up. One of the KPIs for the Kafka based Elasticsearch 7 search was that the average response time should stay under 1 second and it was achieved.

As shown in Figure 35, the mean response time for search requests is 620ms, which is within the peak hour usage response time frame. As shown on Figure 36, the average time for Elasticsearch query response is 72ms, which leaves 584ms for building a query, post-processing results and service to service latency. Figure 35 also shows that at the beginning of the year, the mean time was over 100ms slower. Before March, all of the requests were routed through monolith, but during March, Pipedrive gradually started to switch over to querying directly from search service, as shown in Figure 31. However, that 100ms is not noticeable and on the upside, users get much more features from search.

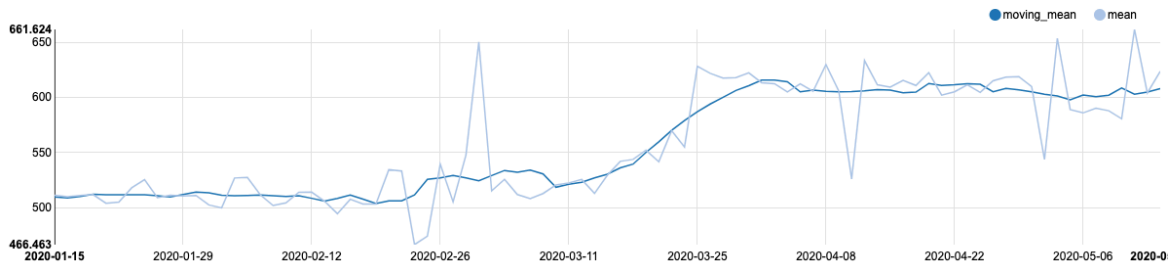

Figure 35 Mean search response times

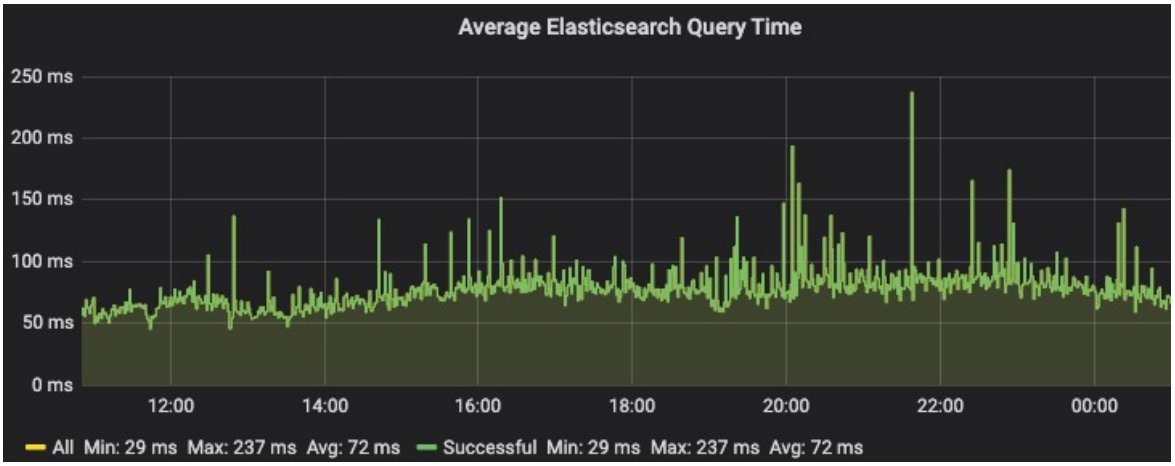

Figure 36 Graph of average Elasticsearch query time

Data graphs are used to monitor if changes made actually have positive impact on the user experience. When working on Kafka based Elasticsearch 7, score values of search results were changed. Score value of the results determines the order of which the results are being displayed in the UI. Lower score means that result user clicked on was ordered higher on the queue. As shown in Figure 37, the mean score has improved a lot and is currently 2.3. What it means is that Pipedrive can predict what result is most likely being searched.

Figure 37 Mean position of the result that was clicked on

7 Future work

Even though the current backend architecture has proved itself reliable by having consistency between Elasticsearch and database while also being scalable, there are some minor improvements that need to be addressed.

7.1 API events

To push user data changes to Elasticsearch, Pipedrive, currently relies on Kafka events generated from database changes by customers. This means that a single API event (e.g. a deal created) produces several messages at once (at least one in every table related to deals). This is a problem because Kafka does not ensure message ordering between different topics and each database table belongs in a topic of its own. This causes Pipedrive to have a lot of extra logic to ensure it supports database row changes related to one event coming in a random order.

To solve this problem, the plan is to create API events that are 100% reliable. Currently API events are unreliable because Pipedrive does not wait for them to be produced. What it means is that events are going to be produced only when user action is created. In case the producing of an event should fail, then the user's action must also fail.

7.2 Global search revamp

Pipedrive has also decided to revamp the global search. Search is divided into 3 different search areas: item search described in this thesis, quick help and blog. These are all separate and do not search for the same information. The main goal would be to combine these searches together into one search, so that the code is not duplicated and the global search would be able to provide the most useful results back to the user.

The research on this problem area has been started and the mission is planned launch in the 4th quarter of 2020.

8 Conclusions

This thesis presented a problem of how to build a reliable search backend using Elasticsearch. Systems that were described throughout this thesis are being used by 90 000 users, who make over 6 million requests every day. Meaning that a need for effective and scalable search functionality is essential in order to provide users with quick and valid information. It discusses all of the biggest issues that had surfaced from Pipedrive search implementation and what methodologies were used in order to solve them.

The aim of this thesis was to understand how to build a reliable search backend using Elasticsearch. The research gives an overview of what kind of solutions and technologies were used in the past and are being used now by Pipedrive to deliver accurate and prompt data. In addition, it describes multiple past versions of search architecture while bringing out all of its most affecting problems in details. Investigating previous versions gave an understanding on how to avoid and prevent some of the mistakes made by Pipedrive. Inconsistencies between Elasticsearch and the database was the biggest reason why Pipedrive even started considering using Kafka synchronization. Moreover, it discusses the solutions that were implemented in the following versions.

Reflecting on the work, Pipedrive has solved a reliable search architecture problem for itself by going over to Kafka-based Elasticsearch. Methodologies and architecture described here are very specific to Pipedrive use-cases and application but can be generalized for other applications. Even though this thesis cannot be used as a step-by-step guide for building a search service for any application, it can be used as a generic guide and gives valuable ideas on how to solve some issues. Furthermore, this thesis has brought out and also suggests that being experimental with implementing diverse applications into Elasticsearch might lead to huge success the same way it did with Kafka. Based on these conclusions, further research can be done by comparing the architecture described in this thesis to a solution using different applications.

References

Rostanski, M., Grochla, K., & Seman, A. (2014). Evaluation of highly available and fault-tolerant middleware clustered architectures using RabbitMQ. *FedCSIS* (lk 879 - 884). Academy of Business in Dabrowa Gornicza.

Dobbelaere, P., & Esmail, K. S. (1. September 2017. a.). Kafka versus RabbitMQ.

Abbot, M. L., & Fisher, M. T. (2009). *The Art of Scalability: Scalable Web Architecture, Processes, and Organizations for the Modern Enterprise.* Addison-Wesley.

Nielsen, J. (1993). *Usability Engineering.* Morgan Kaufmann.

Crof, W. B., Metzler, D., & Strohman, T. (2015). *Search Engines: Information Retrieval in Practice.* Pearson Education, Inc.

Dee, C. R. (2007). The Development of the Medical Literature Analysis and Retrieval System (MEDLARS). *Journal of the Medical Library Association*, 416-425.

Białecki, A., Muir, R., & Ingersoll, G. (2012). Apache Lucene 4. *Proceedings of the SIGIR 2012 Workshop on Open Source Information Retrieval*, 17-24.

Brin, S., & Page, L. (2012). Reprint of: The anatomy of a large-scale hypertextual web search engine. *Computer networks, 56*(18), 3825-3833.

Gormley, C., & Tong, Z. (2015). *Elasticsearch: The Definite Guide.* O'Reilly.

Paro, A. (2015). *ElasticSearch Cookbook, 2nd Edition.* Packt Publishing.

Kononenko, O., Baysal, O., Holmes, R., & Godfrey, M. W. (2014). *Mining Modern Repositories with Elasticsearch.* University of Waterloo, Waterloo, ON, Canada.

Henzinger, M., Motwani, R., & Silverstein, C. (2002). Challenges in Web Search Engines. *ACM SIGIR, 36*(2), 11-22.

Kreps, J., Narkhede, N., & Jun, R. (2011). Kafka: a Distributed Messaging System for Log Processing.

Jones, B., Luxenberg, S., McGrath, D., Trampert, P., & Weldon, J. (2011). RabbitMQ Performance and Scalability Analysis. *CS 4284 Systems and Networking Capstone*.

Hamburger, Y. A., & Ben-Artzi, E. (2000). The relationship between extraversion and neuroticism and the different uses of the Internet. *Computers in Human Behavior, 16*(4), 441-449.

Uplavikar, N., Malin, B. A., & Jiang, W. (2020). Lucene-P2: A Distributed Platform for Privacy-Preserving Text-based Search. *IEEE Transactions on Dependable and Secure Computing*.

Grafana. (2020, May 14). Retrieved from What is Grafana?: https://grafana.com/docs/grafana/latest/getting-started/what-is-grafana/

Thönes, J. (2015). Microservices. *IEEE Software, 32*(1), 116.

Sullivan, D. (2016, May 24). *Search Engine Land*. Retrieved from Google now handles at least 2 trillion searches per year: https://searchengineland.com/google-now-handles-2-999-trillion-searches-per-year-250247

Sookocheff, K. (2015, September 25). Retrieved from https://sookocheff.com/post/kafka/kafka-in-a-nutshell/

Shukla, P., & Sharath Kumar, M. N. (2019). *Learning Elastic Stack 7.0: Distributed search, analytics, and visualization using Elasticsearch, Logstash, Beats, and Kibana, 2nd Edition.* Pact Publishing.

Anikin, S. (2019, March 07). *Scaling Pipedrive Engineering — From Teams to Tribes.* Retrieved from Medium: https://medium.com/pipedrive-engineering/scaling-pipedrive-engineering-from-teams-to-tribes-8f14fd92df8c

Elastic. (2020, March 3). Retrieved from https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl-wildcard-query.html

Elastic. (2020, May 11). Retrieved from Glossary of terms: https://www.elastic.co/guide/en/elasticsearch/reference/current/glossary.html#index

Debezium FAQ. (2020, 03 25). Retrieved from Debezium: https://debezium.io/documentation/faq/#what_is_debezium

Johansson, L. (2019, September 24). *CloudAMQP*. Retrieved from Part 4: RabbitMQ Exchanges, routing keys and bindings: https://www.cloudamqp.com/blog/2015-09-03-part4-rabbitmq-for-beginners-exchanges-routing-keys-bindings.html

Kharenko, A. (2015, October 9). *Microservices Practitioner Articles*. Retrieved from Monolithic vs. Microservices Architecture: https://articles.microservices.com/monolithic-vs-microservices-architecture-5c4848858f59

Apache Kafka. (2020, March 21). Retrieved from Kafka 2.5 Documentation, Replication: https://kafka.apache.org/documentation/#replication

Nagel, S. (2020, April 30). *Apache Software Foundation*. Retrieved from NutchTutorial: https://cwiki.apache.org/confluence/display/nutch/NutchTutorial

Pipedrive Developers' Corner. (2020, May 12). Retrieved from API Reference: https://developers.pipedrive.com/docs/api/v1/#!/Notes

Redis. (2020, May 14). Retrieved from Introduction to Redis: https://redis.io/topics/introduction

I. License

Non-exclusive licence to reproduce thesis and make thesis public

I, _____Harald Astok_____,

(*author's name*)

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to

reproduce, for the purpose of preservation, including for adding to the DSpace digital archives until the expiry of the term of copyright,

___Elasticsearch in Pipedrive_____,

(*title of thesis*)

supervised by __Dietmar Alfred Paul Kurt Pfahl_____.

(*supervisor's name*)

2. I grant the University of Tartu a permit to make the work specified in p. 1 available to the public via the web environment of the University of Tartu, including via the DSpace digital archives, under the Creative Commons licence CC BY NC ND 3.0, which allows, by giving appropriate credit to the author, to reproduce, distribute the work and communicate it to the public, and prohibits the creation of derivative works and any commercial use of the work until the expiry of the term of copyright.

3. I am aware of the fact that the author retains the rights specified in p. 1 and 2.

4. I certify that granting the non-exclusive licence does not infringe other persons' intellectual property rights or rights arising from the personal data protection legislation.

Harald Astok

15/05/2020