FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Martin Wirth

Balancing Keyword-Based Data and Queries in Distributed Storage Systems

Department of Distributed and Dependable Systems

Supervisor of the master thesis:	doc. RNDr. Pavel Parízek, Ph.D.
Study programme:	Master of Computer Science
Study branch:	Software and Data Engineering

Prague 2020

I would like to kindly thank my supervisors, Josef Bouška and Pavel Parízek, for their guidance and willingness to consult any of my questions related to technical or formal background of the thesis and to discuss any ideas along the way. I could not wish any better research freedom and work supervision.

A special acknowledgement also goes to the company of *Seznam.cz* for the opportunity to dedicate myself to exciting research in the area of distributed systems with the possibility of application to a real-world ad system, backed with support of various kinds.

A really great thank-you belongs to my family and friends, most especially to my parents for their endless patience and support without which I could not get this far in my career and life.

Finally, I would like to thank my colleagues in *Seznam.cz*, Petr Novák and Jakub Dvořák, for their assistance with any domain-specific issues and for the perfect working environment they helped to create.

Title: Balancing Keyword-Based Data and Queries in Distributed Storage Systems

Author: Martin Wirth

Department: Department of Distributed and Dependable Systems

Supervisor: doc. RNDr. Pavel Parízek, Ph.D., Department of Distributed and Dependable Systems

Abstract: Research in the area of load balancing in distributed systems has not yet come with an optimal load balancing technique. Existing approaches work primarily with replication and sharding. This thesis overviews existing knowledge in this area with focus on sharding, and provides an experiment comparing a state-of-the-art load balancing technique called *Weighed-Move* with a random baseline and an existing domain-specific balancing implementation. As a significant part of the project, we engineered a generic and scalable load balancer that may be used in any distributed system and deployed it into an existing ad system called *Sklik*. The major challenges appeared to be tackling various problems related to data consistency, performance and synchronization, together with solving compatibility issues with the rest of the still-evolving ad system. Our experiment shows that the domain-specific load balancing implementation produces data distribution that enables better performance, but *Weighed-Move* proved to have a great potential and its results are expected to be enhanced by further work on our implementation.

Keywords: distributed system, sharding, load balancing

Contents

1. Introduction

Year-by-year, computer systems are expected to process still larger data in still less amount of time. In many cases, the scope of data size has outgrown the capabilities of single-computer systems, leading to the necessity of using multiple computers to process the data. Furthermore, from several points of view, using multiple cheap computers is more economical compared to usage of one expensive super-computer.

Systems consisting of multiple cooperating computers, often termed as *nodes*, interconnected by a fast network are called *distributed systems*. From reasons stated above, this is a standard for today's medium- or large-sized systems. For instance, popular multimedia-hosting services, such as *YouTube*[1], or so-called *cloud* applications, such as *Google Drive*[2], or large e-shops like *Amazon*[3] – these are all prototype examples of distributed systems.

In short, distributed systems offer a potential of greatly scalable performance and/or storage space with low costs at the same time, when compared to single-node systems. This leads to their great popularity. Research in this area has been happening already for decades. However, we still have not discovered the best approaches to all aspects of these systems, not even by far.

This thesis aims to contribute to this research by conducting an experiment comparing state-of-the-art approaches to *load balancing* in these systems, i.e. (re)distributing of requests and data to system nodes.

In Section 1.1, distributed systems are described in a greater level of detail, defining related terms and issues. Section 1.2 then puts the thesis into the context of distributed systems and Section 1.3 lists some additional vocabulary for the rest of the text. The last section outlines the following chapters.

1.1 Distributed Systems

The main advantage of distributed systems, as broadly mentioned above, is their ability to *horizontally scale*, i.e. to change the number of nodes utilized in the system. A node, usually imagined as a physical computer, might be any unit with independent computational abilities, from a physical computer, over a virtual machine, to a *Docker*-like container.

Horizontal scaling brings its specific problems that need to be addressed and which are not present in a non-distributed environment. The core of the problems is that data is not located at a single place. We need to concern with *consistency* and *availability* (which are defined below), especially with respect to network *partitioning* (i.e. an event when the network gets split into several parts that can no longer communicate with each other).

Availability is defined not only by the fraction of time when the system must answer to requests but also by the speed of responding. We often demand a specified timeout threshold for *response time*, i.e. the amount of time from sending the request to receiving a response, also called as *latency*. Differing by the seriousness of consequences caused by breaking the limit, we may demand a so-called *soft limit* on the response time (delays are only unwanted) or a *hard limit* (delays may cause human death or great financial losses).

As for consistency, it specifies what is the expected state of data that should be enforced in the system. The requirements may be of various levels of strength. Two extreme scenarios are: first, that we need the data to be in the same state everywhere at all observable moments, referred to as *strong consistency*, or second, that we might get along with data being in the same state only at some time points, which is termed as *eventual consistency*.

With multiple nodes in the system, the data can be either distinctly partitioned and scattered over the nodes or replicated as a whole on them. The former approach is called *sharding*, while the latter is *replication*. These approaches can be combined.

Replication, obviously, does not effect the response time but changes the *throughput* of the system, i.e. the number of requests it is able to respond to. Sharding, on the other hand, primarily effects the response time because it manipulates the amount of data each node needs to handle and potentially process. For example, a node having only a fraction of the whole system data can process it in only a fraction of time. The response time then, naturally, effects the throughput.

As being effective and simple, replication is often the choice when attempting to horizontally scale a system. However, in cases when growing data size results in growing response time, sharding is necessary. A typical example of this case are distributed databases where processing of a request on a node depends on the amount of data stored on that node.

In general, having more nodes responding to requests requires definition of data and request distribution to nodes, which is called *load balancing*. In case of replication, we have complete data at all nodes and thus load balancing is only about deciding what requests are routed to which nodes. At sharding, on the contrary, we have no replication of data and thus there is only one way of routing requests: broadcasting. Therefore, load balancing is here only about defining what data chunks are distributed to which nodes. In a system combining sharding and replication, load balancing naturally involves both data and requests distribution to nodes.

This section is not an exhaustive description of all terms and aspects related to distributed systems. Only those needed for setting the context for this thesis topic have been covered. Some other are left for Section 1.3 or for more appropriate chapters later in the text.

1.2 This Thesis

In this thesis, we analyze and compare load balancing techniques in distributed systems where data is being stored, processed and retrieved and where the following is satisfied:

- We are fine with eventual consistency.
- Availability can be preferred over strict consistency.
- Network is reliable enough so that network partitioning can be ignored[1]. It is also assumed that the used network technologies guarantee message delivery to an addressed node exactly once and that messages can be delayed arbitrarily long but the message order is not changed.
- And there is a soft limit on response time.

Possibility of data growth together with limited response time demands sharding. Therefore we primarily focus on sharding leaving replication only as a possible improvement.

The challenging aspect of this topic is dealing with *hot spots* in data, i.e. parts of the data that are targeted by requests more often than the other parts. We also consider changes of these hot spots in time and changes in the data alone. Note that imbalance of load might be caused also by heterogeneity in performance of individual system nodes, not only by hot spot data. In this thesis, we focus on imbalance caused by hot spot data, as we assume that heterogeneity of system nodes is the lesser of the two problems in today's systems.

The environment for our experiments and our implementation is an existing real ad system *Sklik* of the company *Seznam.cz*, a leading company in the area of Internet services in the Czech Republic. Its main functionality is to search for ads related to a given query using keyword metadata of the ads. The returned ads should be relevant to the query and also bring profit to the advertising company.

For example, an ad might be a longer text advertising a new car *Equo* manufactured by an example company *Carmobile*[2], keywords related to this ad might be for instance *"Equo Carmobile"* and/or *"new car Carmobile"*. And an example query fitting these keywords (the second one, to be precise) may be *"new cars manufactured by Carmobile"*.

The scale of this system is tens of nodes handling dozens of gigabytes of ad-related data. In particular, hundreds of million active keywords and tens of

[1]This is practically true in systems that are not of a too large scale.
[2]These names are fictional and used only for the purposes of the example.

million active ads. It has to respond to several hundreds of new requests per second, each in a timeout of 500 milliseconds.

Ads relevant to a query must undergo an auction where the winning ad gets published [3]. The auction lasts a non-trivial time that depends on the number of input ads. As the system contains too many ads to be processable by a single node in the given timeout, sharding is necessary to limit the number of input ads competing in the auction on each node.

Data are being changed, added and deleted by advertisers. Also, since the advertising system is connected to the full-text search system, requests can create hot spot data and quite quickly change these hot spots as well (e.g. in case of big events in the real world).

As it is not a safety-critical system, eventual consistency typically suffices and in case of short network problems, consistency issues can be ignored.

All in all, *Sklik* is a fine representative for our investigated class of distributed systems.

1.3 Additional Terms

In this section we present a set of additional terms, which will help brevity of the following chapters.

Load balancing is considered to be ***load-aware*** when it somehow respects the current load of individual system nodes. ***Load-unaware*** load balancing means the opposite. Unless stated otherwise, we assume that the load imbalance is caused by hot spot data, not by the heterogeneity of performance of system nodes.

Data (re)distribution, ***data (re)balancing*** and ***(re)sharding*** are all used interchangeably and they are all understood as an event of changing data distribution to system nodes, no matter the size of the change.

Aggregate is a data structure containing all information about an object from a point of view of some business logic. In other words, it is a data unit of some processing. For example, an aggregate representing a customer of an e-shop might be a data structure containing the customer's name, shopping cart and other information needed to finish the customer's order. In processing of the order, we work with all these information at once.

Database, ***store*** and ***storage*** are all understood as interchangeable and represent some kind of structured storage of data.

Cache is any kind of secondary storage that replicates a part of some primary data source. The client usually tries to obtain data first from the cache and then, if there are some data missing, it queries the primary storage.

[3]This is not precise but close enough for the purposes of the introduction.

Components designate architectural units of a system. They can be of arbitrary size, e.g. a subsystem, a library or a class. Sometimes, especially in the terminology of the software architecture discipline, it is differentiated between modules (structural units) and components (run-time units, often instances of modules). In this thesis, we use only the term ”component”, meaning both modules and components.

Adjectives ***local*** and ***remote*** designate whether the particular specified activity is happening in the same network node or over network, respectively.

1.4 Thesis Outline

The rest of the text is organized into chapters with the following listed content. More details about individual chapters can be found at their beginnings.

Chapter 2 describes the ad system, which forms the context of our experiment. It is based on information from Section 1.2 and goes to much finer details.

The third chapter briefly lists the goals of this thesis project.

Chapter 4 is dedicated to an overview of related work in the area of interest of this thesis. Any conclusion of this information with respect to this thesis are left to Chapter 5, which also presents our originally intended goals and concepts of our work.

Our development, i.e. implementation, testing and documentation, together with related analysis, is the subject of Chapter 6.

There were two major experiments performed during our work. Chapter 7 presents an experiment that was conducted before we changed the concept and goals of this thesis, and which proved useful also in the final form of the project. The main experiment that has been promised in Section 1.2 is presented in Chapter 8.

The final chapter concludes all our results and work and lists possibilities for any future work.

2. Ad System

This chapter presents the ad system *Sklik* that serves for evaluation and implementation of our experiments with different load balancing techniques. As discussed in Section 1.2, it is a perfect representative for our research.

First, we introduce several domain-specific terms. Then we present how the ad searching works, briefly overview the system architecture, describe the load balancing implementation, and finally, we discuss the system's limitations.

Even though we present the system from the first-person point of view, it is only for purposes of text fluency. No components or algorithms mentioned in this chapter are results of our work on this thesis project, everything comes from the company *Seznam.cz*.

Also note that some aspects of the system are confidential and therefore we keep the description rather on a broad level, which is, however, fully sufficient for the reader's understanding of the system.

2.1 Domain-Specific Terms

Requests sent to the ad system have a form of a *query* enriched with meta-data, such as the identification of the website asking for ads. The query is organized into *words*.

The sender of the query is called *user* and the author of ads and related data is called *advertiser*.

Advertiser's data are organized in a structured, hierarchical manner, as illustrated in Figure 2.1. The top-level structures are *campaigns* that have more-or-less only a semantic meaning to the advertiser. For example, a campaign can contain ads for a specific topic (for example, mask sales). Campaigns are organized into *groups* that are intended to target a specific product or service of the advertiser (for example, a sale of home-made masks).

A group contains *keywords* and *ads*. Keywords are used to *match* a query, based on textual similarity. For example, a keyword *"home-made mask"* might match a query *"where to buy a home-made mask"*. If a keyword matches, one of the ads belonging to the keyword's group is selected as a candidate. More on this mechanism is presented in the following section.

To prevent confusion around the relation "to match", we understand it as symmetric in this thesis, i.e. if a query matches a keyword, then the keyword also matches the query.

Each keyword is linked with a *lexicon* which stores the actual text of the keyword. Keywords do not, in fact, contain the text but only point to a lexicon

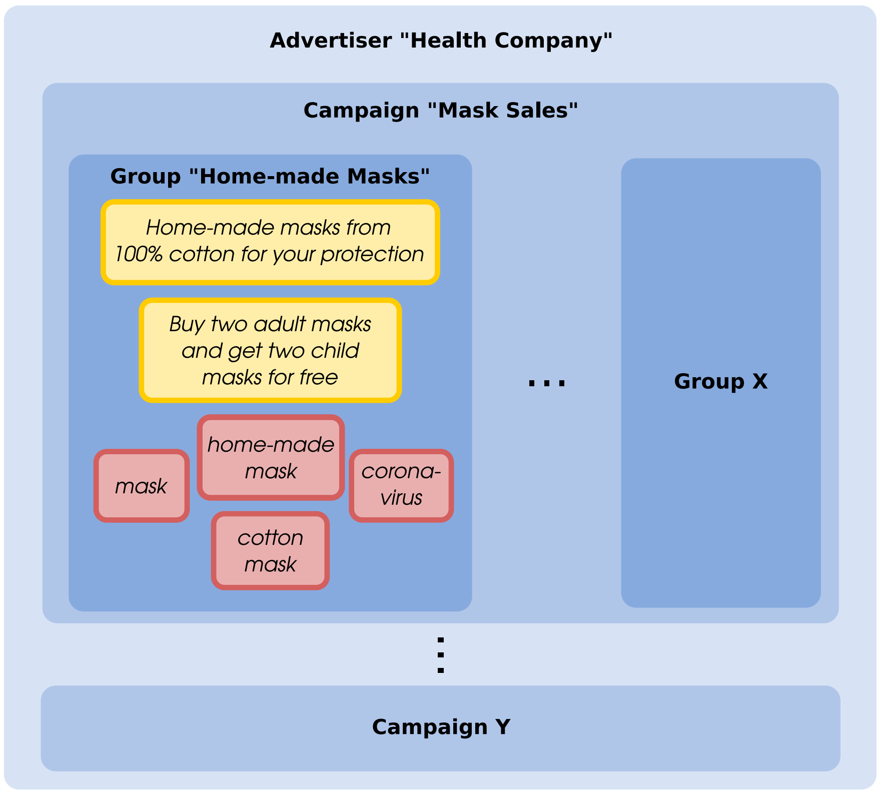

Advertiser "Health Company"
Campaign "Mask Sales"
Group "Home-made Masks"
Home-made masks from 100% cotton for your protection
Buy two adult masks and get two child masks for free
home-made mask
mask
corona-virus
cotton mask
...
Group X
⋮
Campaign Y

Figure 2.1: Advertiser Data Hierarchy
Blue rectangles denote advertisers, campaigns and groups. Yellow rectangles are ads and red rectangles represent keywords.

with this text[1]. This saves space since multiple keywords in the system typically share the same text.

To select the ads issued by the system for a given query, the matched keywords compete in an *auction*. Details are also left for the following section.

There are many more domain-specific terms in the system. Some are out of scope of this thesis and some are too early to introduce. Their definition is left for the sections below.

2.2 Ad Search

In this section, we present how ads are selected for a given query. The algorithm's pseudocode, described in detail by the following paragraphs, is summarized in Algorithm 1.

The basic idea is that ads matching the given query are searched using keywords related to the ads[2]. In other words the query is not matched onto the ad text – in fact, the ad text is completely irrelevant for the ad search – the query is

[1]For fans of the *Gang-of-Four* design patterns, it can be seen as an instance of the *Flyweight* pattern[4]. For fans of relational databases, it is normalization of data.

[2]Keywords are inserted into the system by the advertiser.

Algorithm 1 Ad Search

```
1: $candidates \leftarrow$ empty list
2: $queries \leftarrow$ Find all similar queries for the input query
3: for each $query \in queries$ do
4:     FindMatchingKeywords($query$, $candidates$)
5: $auctionData \leftarrow$ Compute auction data for keywords in $candidates$
6: $ads \leftarrow$ Execute the auction based on $candidates$ and $auctionData$
7: return $ads$

1: function FindMatchingKeywords($query$, $candidates$)
2:     $words \leftarrow$ Split $query$ into words
3:     for each $word \in words$ do
4:         $keywords \leftarrow$ All keywords beginning with $word$
5:         for each $keyword \in keywords$ do
6:             if $keyword$ matches $query$ then
7:                 Add $keyword$ to $candidates$
```

matched onto keywords. Therefore, keywords are the core entities in the process. Given a query, the algorithm finds relevant keywords and it is keywords, not ads, that then compete in the auction.

At the beginning of the process, the system obtains a query. It finds all similar queries for it, including usage of synonyms, words created by addition or removal of diacritic, etc. All these queries are split into words and participate in the search for relevant keywords. Keywords are searched separately for each of these queries and then all results are merged together.

To find relevant keywords for a given query, the words of the query are iterated and for each, all keywords beginning with that word are examined whether they match the query. The algorithm must respect different kinds of word matching, for example with or without declination. Details of this are confidential and not important for this text.

Once all relevant keywords are collected, the next step is preparation of data for the auction and the aution itself (both called together as the *auction routine* later in this text). The goals are (1) to reduce the number of relevant keywords to fit the final number of ads returned by the system, and (2) to select those ads that are most relevant (for example, based on the device type displaying the website) and at the same time most profitable to publish for the advertising company. Details of this process are, once again, confidential and not important for this text. There is only one aspect that is crucial: a part of the keyword's meta-data for the auction cannot be precomputed since it depends on the context and as such it is computed in real time.

This computation is quite time-expensive process and the overall time depends on the number of input keywords. If there are too many keywords matching the query on a single node, the node does not respond in the given timeout. This is the reason why we need to shard the data in the system – in order to decrease the number of input keywords participating in the auction routine on a single node.

During the auction routine, ads for the keywords are selected. As mentioned in Section 2.1, keywords and ads are organized independently in a group. Therefore, to find an ad for a keyword, one is selected from all the ads in the group the keyword belongs to.

The resulting ads are then returned as the response of the system.

2.3 Architecture

The core of the system's architecture is motivated by the need of sharding. It is illustrated by the component view in Figure 2.2.

An incoming request is routed to the *master server* which broadcasts the request to all the *slave servers*. Each slave server performs the search of relevant keywords, significant part of the auction routine and sends back the results. The auction routine is finished on the master server and the final result is returned as a response to the request.

The slave servers search for keywords and related data in their local storage, which is organized in separate key-value stores called *barrels*. The technology behind barrels differs but the main one is a combination of *Protocol Buffers* serialization [5] and proprietary indexing and structuring of the key-value pairs.

The barrels are constructed by components called *creators*. For a barrel shared by the slave servers, there is only one common creator. In case of barrels with sharded data, each slave server has its separate creators. The creators take the data from a backend database. Details around this mechanism are left for the next section.

This presented infrastructure is only a simplification sufficing for this text. In the real system there are many other components and, this presented core infrastructure is replicated as a whole in several instances. Incoming requests are then distributed among these instances. This is, however, out of our main interest – in this thesis, we are after sharding, as discussed in Section 1.2.

2.4 Load Balancing

As discussed in Section 1.1, load balancing in the configuration of the system core introduced in the previous section is only about data distribution and not query distribution, as we utilize only sharding and not replication[3].

The complete data is stored in a backend database. As there are other systems using the data in this database, the data are not stored in the ideal form for the

[3]Some form of replication is implemented, but above this system core, as described earlier.

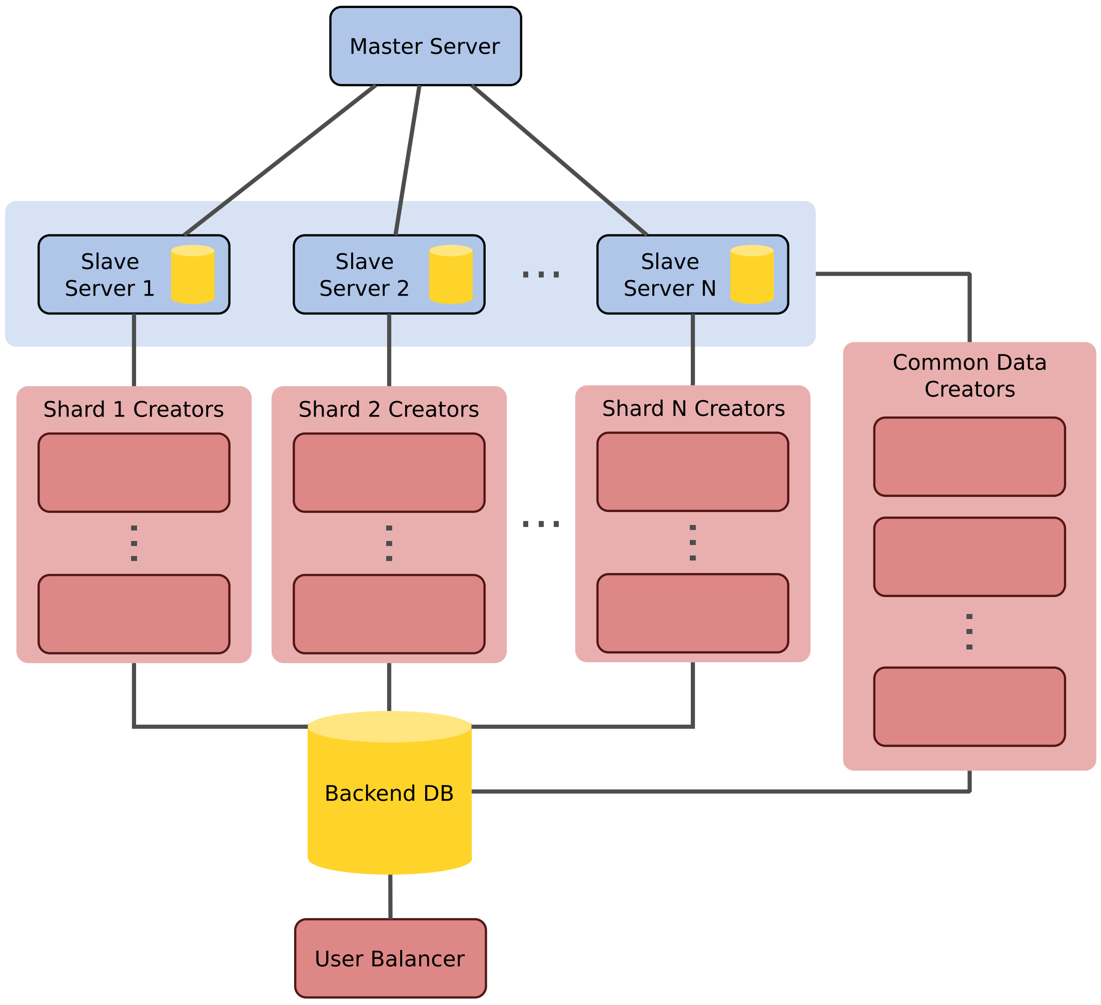


Figure 2.2: Component view on the original system's core
The blue part designates slave and master servers. The red part represents the sharder components and yellow parts denote storage components: the backend database and the barrels. Lines represent flow of data.

ad system. Therefore, the task is to determine what data should be distributed to what slave servers, transfer the data and do some its processing.

The component *User Balancer*, which is also illustrated in Figure 2.2, is responsible for assignment of data to the slave servers. It does not transfer the data, it only defines the assignment.

The algorithm of the balancer is weight-based. It assigns data items to the slave servers so that the sum of weight at individual slave servers is as uniform as possible. In other words, it solves a variation of the famous NP-complete problem called *Bin Packing* [6]. The weight is based only upon the numbers of keywords, ads and lexicons.

The selected granularity of data items is quite coarse: an item is the complete data of an advertiser, i.e. all data contained in all the advertiser's campaigns. This granularity was also selected by *Google* in their ad system in one of its legacy versions [7].

The balancer runs once a day, in the middle of night so that there are as least requests as possible that are effected by the inconsistent state before all data get to their right slave server.

Having an assignment of data to slave servers, the next is to perform the data transfer and the data processing. There are two steps for this. First, creator components, which have been introduced in the previous section, retrieve the data from the database, process them into a required form and construct final barrels for the slave servers. Note that the barrels are created whole, from scratch. A creator constructing a barrel with sharded data of a given slave server takes into account only the data assigned to that slave server by the balancer. Second, the slave servers download the barrels from the creators and replace their old barrels with the new ones.

While the balancer runs only once a day, creators construct the barrels in subsequent iterations. This way, any changes in the advertisers' data get to the slave servers even during the day. Furthermore, any new-coming advertisers that have not been assigned to a slave server by the balancer, are assigned to more-or-less random slave server and thus get to slave servers during the day as well.

2.5 Limitations

We are aware of three major limitations of the load balancing mechanism.

First, the granularity is too coarse. Some of the advertisers have large data and that means that getting a practically uniform distribution of data over slave servers might not be possible. Furthermore, if an advertiser data was larger than capacity of a slave server, the assigned server would get always overloaded.

Second, there is no handling of hot spots in the data. For example, if an advertiser has a lot of keywords with the same topic or there are multiple advertisers assigned to the same slave server who have a lot of keywords with the same topic, this slave server is a bottleneck for queries related to this topic. In other words, the load balancer targets only uniform distribution with respect to data space but not with respect to topics and their probability of being queried.

The third and last limitation is related to consistency and reaction time on data changes. It takes quite a non-trivial time until the new changes in the backend data or the at-night rebalanced state get correctly propagated to the slave servers. There are two reasons for this. First, the construction of barrels from scratch takes time. Second, different barrels gets constructed at different times leading to temporarily inconsistent barrels at slaves, which results in temporary ignorance of the inconsistent data (i.e. the new changes). Even though this is eventual consistency and in Section 1.1 we stated that we are satisfied with this, we might want to improve the change-reaction time.

3. Goals

The primary goal of this thesis project was to analyze and compare selected sharding techniques for distributed data-storage-and-retrieval systems characterized in Section 1.2. The focus was put on those techniques that are aware of non-uniform and changing distribution of requests on data. Since data access skew has been observed in quite a lot of representative Internet services [8], this topic is more than convenient to study.

We selected the ad system *Sklik* of the company *Seznam.cz* as a representative of such systems and thus a good placement for our experiments. The original load balancer of this system has several limitations, that were specified in Section 2.5. Our secondary goal was to target these limitations.

To summarize all our goals in a single list:

(A) We wanted to analyze the current state of research in the targeted area of sharding techniques, select representatives from them and conduct an experiment for their comparison. The intended experiment's context is the *Sklik* ad system, which has been presented in Chapter 2, more particularly, all techniques were implemented as a replacement of the *User Balancer* component.

(B) Utilizing the implementation created for purposes of the first goal, we also wanted to solve the limitations of the original ad system's load balancer that have been described in Section 2.5. Namely, we were after:

   (1) Data balancing with finer granularity than in the original system where a data item is as big as complete data of an advertiser. A fine granularity is necessary for getting close to an optimal, uniform data distribution.

   (2) Uniform distribution of ad data with similar topic over the slave servers in order to spread the load resulting from queries that retrieve ad data from that topic. Primarily in case of frequently occuring topics.

   (3) Reduction of time when the system happens to be in an inconsistent state caused by changes in the backend database.

4. Related Work

This chapter presents an overview of modern approaches and ideas in the area of distributed systems we characterized in Section 1.2. Note that we only summarize the related papers here, any conclusions that are based upon these papers and that affected our work are left for Chapter 5.

The first sections analyze possible architectures of such systems. Namely, so-called *RInK* and *LInK* architectures are presented in Section 4.1, while Section 4.2 discusses an architecture that respects locality of requests similarly to the principle of caches. The latter sections are dedicated to sharding algorithms. A broad decription of different aspects of *consistent hashing* can be found in Section 4.4, a load balancer from *Google* called *Slicer* is presented in Section 4.5 and Section 4.6 is dedicated to *Autoplacer*. The chapter ends with Section 4.7 briefly introducing *rendezvous hashing*.

4.1 LInK Architecture

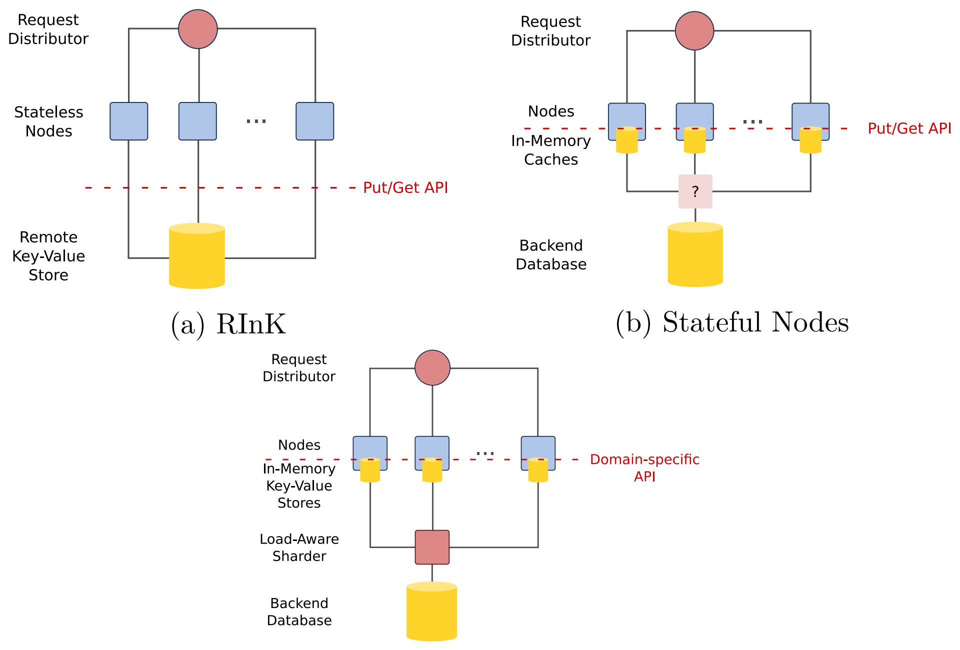
Three possible kinds of architectures of distributed data-storage-and-retrieval systems are discussed in a recent paper by Adya et al. [9]. Namely:

- *RInK* (Remote In-memory Key-value store) architecture, which is identified by stateless nodes accessing a remote domain-independent key-value store. More details are left for Section 4.1.1.
- An architecture of stateful nodes accessing their local domain-independent key-value stores, as described in Section 4.1.2.
- *LInK* (Linked In-memory Key-value store) architecture, which is characterized by stateful nodes with local domain-specific key-value stores populated by a load-aware data balancer. This architecture is discussed in detail in Section 4.1.3.

4.1.1 RInK

The RInK architecture (Figure 4.1a) involves stateless nodes querying a remote in-memory key-value store backed by a persistent storage.

The simplicity of the design provides robustness and relatively easy scaling. Being stateless, nodes can be easily added or removed. The RInK store, having trivial interface, can be easily scaled by replication and/or sharding. These are very attractive advantages of this architecture as the system is quite simple and



(a) RInK (b) Stateful Nodes

(c) LInK

Figure 4.1: Architectures of distributed data-storage-and-retrieval systems

can easily adapt to different load levels, which, in the end, has potential to results in lower maintenance costs.

Despite these tempting advantages of this architecture and its wide usage in the industry, the authors of the paper [9] argue that the design negatively effects the performance. The following list summarizes their arguments:

- First, to get the data, the stateless nodes need to query the remote storage, leading to a significant slow-down caused by the network overhead.
- The API of the today-used key-value stores is usually domain independent – they commonly provide only put and get operations with universal byte arrays for values and sometimes also for keys. This requires unnecessary data transformation. Furthermore, it is prone to reading more data than actually needed, called in the paper [9] as *overread*. For example, reading a whole person's contact book instead of a single contact, which is actually required, is an overread.
- Both the previous aspects then even generate another issue: each data-retrieval is associated with a costly process of marshalling and unmarshalling the data.

4.1.2 Stateful Nodes

Illustrated in Figure 4.1b, the architecture based on stateful nodes implements a cache storage directly into the nodes. Together they contain all the data of the system, in whatever distribution selected as appropriate for the system, all backed by a backend database.

This setup eliminates network overhead and a portion of (un)marshalling, leading to a measured 30-60% improvement of latency, approximately [9].

4.1.3 LInK

The LInK architecture (Figure 4.1c) utilizes in-memory key-value stores with a key-to-rich-object schema, placed locally to the system nodes. In addition, it uses a load-aware sharder that assigns whole key-value pairs, it does not assign only keys relying on the node to gather the value on its own.

The difference from the architecture of stateful nodes presented in the previous subsection is the emphasis on the values in the key-value pairs. Therefore, values are distributed along the keys during data balancing and they are accessed in a domain-specific manner, not just as byte arrays.

Another difference is the approach to data balancing. Its implementation in the previous architecture is left unspecified and thus there is often a load-unaware approach deployed – i.e. data are distributed without, or with only little, respect to the current system load. The LInK architecture, on the contrary, demands load-aware data balancing.

By implementing the transfer from the RInK architecture to the LInK architecture, the paper's authors [9] experienced a really significant system latency improvement of approximately 40-60%.

4.2 Cache-Aware Architecture

Another recently presented architecture [10] is based on locality of requests. It assumes that requests can be divided into some relatively disjunctive topics (the definition is system-specific) and the topic distribution is quite stable in time (i.e. the set of topics and their frequency of occurrence via requests do not change rapidly).

The idea is that all system nodes have a local in-memory cache storage and can access to a backend storage with the complete system's data. This backend storage may be local to the nodes on their disk, if it fits there, or, more-expectedly, remote. All is illustrated in Figure 4.2.

When a node receives a request, it primarily tries to use the locally cached data and in case of a miss, it retrieves them from the backend storage – similarly to the case of CPU cache and RAM.

With this setup and the assumption of request locality, the system can use a load balancer which distributes requests in a cache-aware manner. According to some algorithm, it decides which node has most of the data related to the

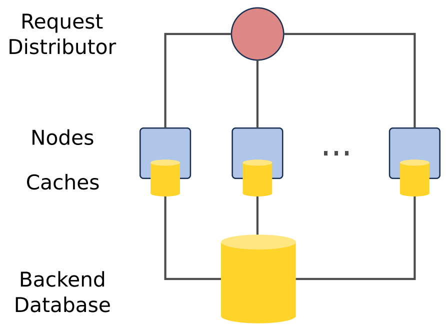


Figure 4.2: Cache-Aware Architecture

request's topic in its cache storage and forwards the request there. The goal of this approach is to minimize the amount of cache-storage misses.

To present real-world results of usage of this approach, in *Google*'s web search backend, cache misses were reduced by approximately 50% leading to a double-digit percentage improvement of the system's throughput [10].

At the same time, if the load balancer fails to identify the right node for the request or there is no good-enough node candidate for the request, sending the request to any node does not end up with a bad reply. It only pollutes the node's cache and impacts the system performance.

The paper's authors [10] call the load balancer's node selection algorithm as *Term-Affinitized Replica Selection* (*TARS*). Each node is assigned several static weights and a dynamic weight. The static weights are computed from the node's log of requests and capture what the node has in its cache (for each topic, there is a separate static weight). This computation is quite non-trivial, it is based on graph partitioning and solving small instances of NP-hard problems. The dynamic weight allows affecting the load balancing to get more uniform load distribution "by hand". The load balancer's decision of the right node for the given request is then based on all of these weights.

This architecture is not exactly appropriate for the distributed systems we are concerned with as it actually does no data sharding. In fact, the system nodes work with the complete system data, even though the performance is increased by caching. However, it was related to our original thesis concept, which is described in Chapter 5, and we found some of the ideas useful even afterwards. We wanted to inspire by this approach and use it for our custom balancing algorithm, which we wanted to compare in an experiment that is presented in Chapter 8. But, unfortunately, we did not have time for this, leaving it as future work.

4.3 Sharding In General

Sharding algorithms in distributed data-storage-and-retrieval systems are usually of two main kinds: (1) ad-hoc sharding specific to the particular system and (2) sharding based on consistent hashing [11].

Algorithms specific to a given system are typically based on the knowledge of the system's data semantics and structure. They are often of rather load-unaware nature – they do not take the current system load into account. This is the case of the data balancing present in the original implementation of *Sklik*.

On the other hand, consistent hashing, presented below, is a technique that is uninformed about details of the system data. It as an abstract model where the data items are represented only by keys with a defined ordering. In its original form, it is a load-unaware distribution of data as well, but modern approaches inspired by consistent hashing (e.g. *Google Slicer* [12] presented in Section 4.5) are load-aware.

There are, of course, sharding algorithms of different approaches, even though they are not so commonly used. Some of them are described at the end of this chapter.

However, what basically all the algorithms have in common is that they aim for uniform distribution of data with respect to its retrieval while minimizing the amount of data which is necessary to transfer when the system scales or when the data load distribution changes.

4.4 Consistent Hashing

Consistent hashing [11] is a popular sharding algorithm. In some form, it is used in many well-known systems, including key-value stores *Memcached* [13], *Cassandra* [14] and *Amazon DynamoDB* [15] or distributed lookup protocol *Chord* [16], which is utilized in distributed hash tables. *LinkedIn's Voldemort* [17] uses consistent hashing as well.

The reasons behind this popularity are its simplicity, effectivity for data without intense hot spots and small data transfer in case of system scaling. Furthermore, it can be implemented in a distributed manner without any central authority, which helps robustness of the system.

The model used by consistent hashing is a space of keys where the maximal possible key is followed by the minimal possible key as the successor, forming a ring. Data items and system nodes are represented via keys in that key-space. The model works only with these keys, it is not concerned with the data itself. An illustration can be found in Figure 4.3.

Keys identifying data items and also system nodes are selected randomly.

Complete randomness as defined in the theory of probabilistic algorithms is basically impossible to implement. Therefore, real systems use practically random keys, i.e. a random key is obtained via a hash function [1] or via any other approach that seems to generate a random key (with respect to the data item content).

The data distribution is defined as follows: all data items present on the ring clock-wise from the first previous system node are distributed to this node [2].

In the illustrated example, the range r marks an interval in the key space where all data item keys belong to the node $n1$ – i.e. in this case the key $k1$. Similarly, data item keys $k2$, $k3$, $k4$ and $k5$ belong to the node $n2$.

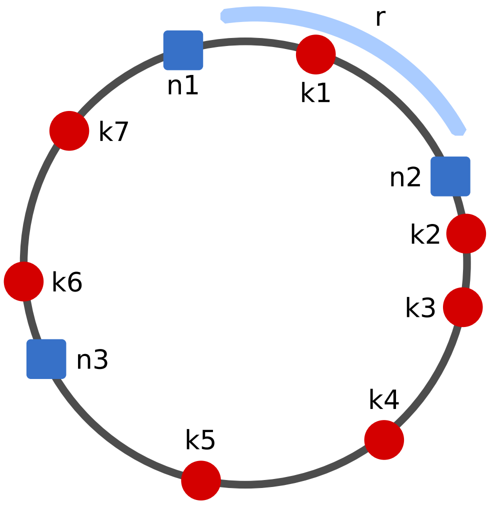


Figure 4.3: Consistent hashing
Data item keys (red circles) and node keys (blue squares) on the key space ring.

To get (probabilistically) uniform distribution of data items to nodes, it is proven [11] that using multiple keys for each single system node is necessary. More precisely, if N designates an upper bound on the number of system nodes, the best number of keys for each node is $c \cdot log(N)$ for a constant c. The greater constant, the more uniform data distribution. These multiple keys for a single node are sometimes [16] imagined as keys of *virtual nodes* that represent the same single real node. Others [15] – and us in this thesis as well – call them *tokens*.

As for one of the popularity reasons, the advantage of consistent hashing over an approach based on straight-forward assignment of data items to random nodes via a traditional hash function (i.e. a simple hash table from data items to nodes) is that when the system scales, relatively small amount of data must be transferred. When a node is added or removed, only the data items belonging to that node must be redistributed. In case of the straight-forward hashing, it could impact potentially all the data items.

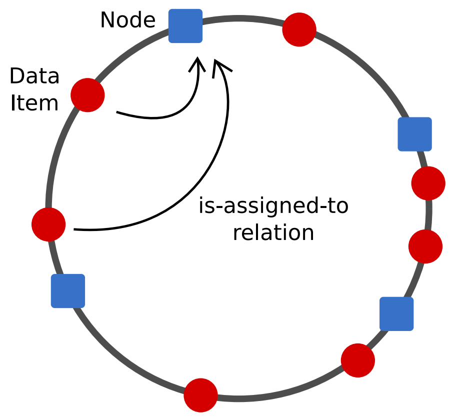
[1] For example, Chord uses the hash function *SHA-1* [16].

[2] Note that different sources use different definitions of what data item keys belong to what nodes. Some use clock-wise direction, some use anti-clock-wise direction, some use the ring-distance based approach. All are equivalent.

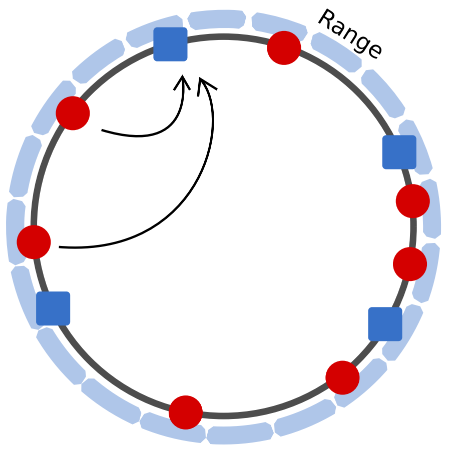
What is worth emphasizing is that consistent hashing does not respect real distribution of accesses to data items. It assumes that all data items are accessed uniformly. Therefore, intense hot spots in the data are a problem. More precisely, if hot data items are close to each other, the randomization related to key assignment will scatter them over the ring, with high probability. However, hot spots appearing on the ring are not solved. This is an issue of systems such as *Memcached* and *Chord* [9, 16].

4.4.1 Consistent Hashing Strategy

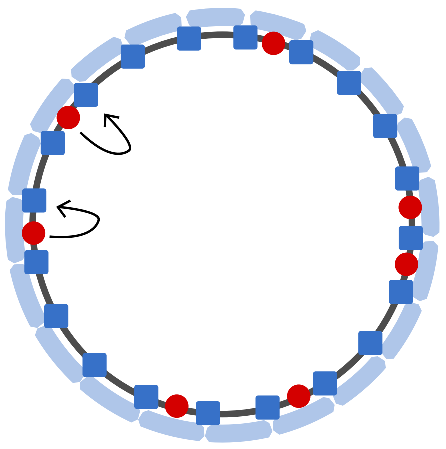
During evolution of *Amazon DynamoDB*, several strategies for key assignment were developed and analyzed [15].



(a) Strategy 1



(b) Strategy 2



(c) Strategy 3

Figure 4.4: Consistent hashing strategies in *Amazon DynamoDB*
Red circles denote data items, blue squares represent tokens and light blue arcs designate ranges

The first strategy, illustrated in Figure 4.4a, is the original strategy of consistent hashing. Each node is given several keys (i.e. tokens) in the ring, at random.

All data items in the range that precedes the token are assigned to the token's node.

This strategy has the following issue: when a new node is added to the system, all data items belonging to tokens neighboring the new node's tokens must be scanned whether they should be reassigned to the new node's tokens or not. This might be a consuming operation in case of a greater number of data items.

The second strategy is modeled in Figure 4.4b. It also uses multiple random tokens for a system node but the data items assignment to tokens is quite different. The key space is divided into a large number R of equally-long ranges. A range is basically an interval in the key-space ring and, in Figure 4.4, it is illustrated as a light blue arc. Data items in a range are assigned to the first token encountered on the ring clockwise from the end of the range. The number R is usually set to be much greater than the number of all tokens of all system nodes so that there is quite high granularity of the ring key-space which helps the uniform distribution of data items.

This second strategy solves the previous issue. When a node joins the system, only the ranges, not individual data items, of tokens neighboring the new node's tokens have to be iterated and potentially reassigned.

The third strategy, as presented in Figure 4.4c, is based on the second strategy. Again, the ring key space is divided into R ranges as before. But there are R tokens (notice the regular pattern in Figure 4.4c) and each node is randomly assigned R/N tokens, where N is the initial number of system nodes. Range assignment to tokens works the same as in the second strategy. The number of tokens stays the same, no matter how many nodes are in the system. When a node is removed from the system, its tokens are reassigned to the remaining nodes. And on the other hand, when a new node joins the system, some of the original nodes' tokens are assigned to it.

In our view, in the third strategy, the key space is bucketized into R buckets, data items are randomly assigned to buckets and system nodes are randomly assigned responsibility for these buckets. That is, the granularity is, in a way, artificially forced to buckets instead of data items. This may be desirable if the system has no control over the data item size or if the system wants to change the granularity often[3].

In the paper [15], the authors ran an experiment comparing these strategies. While all were given the same amount of storage space in the system nodes, they measured the data imbalance in the system. The third strategy won, followed by the first strategy. The second strategy was significantly worse. However, since information such as size of data items, number of nodes or other details were not published, we cannot really explain these results.

[3]We will refer back to this idea in Chapter 6 presenting our implementation.

There is also other research on this topic, Huang et al. [18] and Wang and Loguinov [19] inspect different strategies for data item assignment to system nodes as well.

All such similar strategies focus more-or-less primarily on how to get data items uniformly distributed among the nodes using a distributed form of consistent hashing. They sometimes even take the heterogeneous performance of the system nodes into account. However, none is aware of dynamic hot spots in data. This is what approaches presented in the next subsection aim to solve.

4.4.2 Load-Aware Consistent Hashing

To alleviate heavily loaded system nodes, *Cassandra*, which uses the original data item assignment strategy of consistent hashing, tracks load of the tokens and moves them along the ring to decrease the imbalance [14].

Rao et al. [20] propose several approaches for adapting consistent hashing to load-aware form:

- In the first approach, nodes are divided to heavy and light according to their load. In a distributed manner, one heavy node and one light one are always selected, and a token is transferred from the heavy node to the light node.
- The second approach is similar. One heavy node and several light ones are selected and the transfer goes from the heavy one to the lightest one.
- The third approach is centralized. A central authority gathers tokens that should be passed from heavy nodes to light nodes and transfers them.

Srinivasan and Varma [21] suggest an approach based on splitting and merging key space ranges assigned to tokens. Hot spot ranges are split, a new token is created for the new created range and the token is assigned to some of the neighbor nodes (i.e. the nodes of the neighbor tokens). On the other hand, cool ranges are merged together so that the key space does not get too fragmented, which would result in resource-consuming tracking of all tokens. To decide which ranges are hot and which are cool, Srinivasan and Varma use two threshold values of load.

4.5 Google Slicer

Google Slicer [12] is a state-of-the-art, universal, centralized, load-aware load-balancing middleware that has been successfully deployed to more than 20 *Google* systems.

On the first sight, it is quite similar to the presented dynamic approaches to consistent hashing. As in dynamic consistent hashing, data items are represented as keys (generated by a hash function) and there are also ranges of keys, called here as *slices*, which are assigned to system nodes. Other similarities will be mentioned along the way. Nevertheless, the *Slicer's* concept is different: neither nodes nor tokens are present as keys in the key space ring.

Assignment of slices to system nodes is done via an algorithm that is described in the subsection below. The other two subsections, 4.5.2 and 4.5.3, present the advantages of *Slicer* with focus on scalability.

4.5.1 Sharding Algorithm

The idea of the algorithm is moving slices node-to-node while splitting the hot slices (i.e. the ones with heavy load) and merging the cold ones. You probably find this familiar as it is similar to the approach of Srinivasan and Varma [21].

The *Slicer's* algorithm is called as *Weighted-Move*. In iterations, three steps are performed:

1. **Clean-up:** Data items (slices) assigned to nodes that are no longer present in the system are reassigned (no particular details are documented in the paper [12]).

2. **Merging cool slices:** Two adjacent slices in the ring are merged together if the following holds:

    - No more than 1% of data item keys are moved (the two slices do not have to be assigned to the same node).
    - There are more than 50 slices assigned to the system node that would loose its slice.
    - The resulting slice's load will be less than the mean slice load.
    - The target node's load will not get above the maximal node load.

3. **Moving slices:** Slices are being moved in order to reduce the load imbalance, which is defined as the ratio between the maximal node load and the mean node load, while not transferring too many data items at the same time.

    Note that only slices of the node with the maximal load can reduce the load imbalance. Therefore only those slices are considered during this step.

    They are (virtually) ordered descendingly according to their weight, which is defined as gain from the slice's move divided by the move's cost. The gain is reduction of the load imbalance. And the cost is defined as the number of data items moved.

    The slices are moved in this order until 9% of data items have been transferred.

As for the target node of the movement, it is either the coolest system node or there are other two strategies manipulating the replication factor of these slices[4].

4. **Splitting hot slices:** Slices with a high load value are split to smaller slices. With a little surprise on the first sight, split slices are not moved in this step. The goal is to get finer-grained load measurements, which opens new move opportunities for the next iterations.

   A slice is split if:

   - It is twice as hot as the mean slice load.
   - There are fewer than 150 slices assigned to the system node.

The selection of constants in the algorithm was driven by experience from experiments with the algorithm.

The beginning state of the sharding, even though it is more-or-less irrelevant, is assignment of equally split key-space to system nodes.

4.5.2 Advantages

Load-aware consistent hashing usually provides less control over hot spots because cooling down a hot spot results either in a random reassignment of hot spot's data items (as they are usually transferred to the neighbor nodes in the ring) or in a reassignment to a lightly loaded node – but, in such case, the decision was usually made in a distributed manner, having potentially not a full information of the system. On the contrary, *Slicer* transfers these data items to light system nodes (or performs a similar action in the two other strategies mentioned above) and does so with a full information of the system.

As discussed in the paper [12], *Google Slicer* balances significantly better with less data transfer than the previously used load-aware consistent hashing algorithm.

4.5.3 Scalability

To improve scalability, *Slicer* is divided into two components. First, *Assigner*, which runs the sharding algorithm, and second, *Distributor*, which handles actual distribution of data to system nodes. To scale the system, multiple *Distributor* instances can be used.

Distribution of data to system nodes happens via a pull model: the nodes ask the *Distributor* for their data.

[4]Besides sharding, slices are replicated. The replication factor differs slice to slice.

4.6 Autoplacer

Autoplacer [22] is a load-aware sharding technique also based on consistent hashing. Consistent hashing is utilized to generate default placement of items to system nodes and popular items are then being relocated. This happens through an algorithm that operates in iterations where, in each, placement of the top K hot spot items is optimized. This whole process is done in a decentralized fashion.

Restriction to K items in each round reduces the complexity for a solved instance of the NP-hard Bin Packing problem[5]. As these K items are selected new in each round, *Autoplacer* can optimize much larger amount of items in a greater time horizon.

Furthermore, the authors of the paper [22] wanted to prevent the ping-pong scenario, where several items are redistributed back and forth, by temporarily ignoring once relocated items until all items will have been considered for relocation. This guarantees that the algorithm does not concentrate on just a small subset of items. However, on the other hand, it is not suitable for use cases where load on items changes very often, but that is quite rare.

To prevent gains being less than costs, it always monitors the last iteration and when the obtained gain in load balance becomes less than a user-defined minimum, it does not perform the rest of the whole cycle.

To improve space-efficiency, instead of complete relocation tables tracking placement of all data items on the system nodes, the paper [22] uses a data structure called *Probabilistic Associative Arrays*, a combination of Bloom filters with a machine learning algorithm.

As for the comparison with other techniques, the paper's experiments [22] show that *Autoplacer* performs six-times better than a baseline load-unaware consistent hashing.

4.7 Rendezvous Hashing

Rendesvouz hashing [23] is a data sharding technique which is quite different from the previously presented approaches.

The core of the algorithm is a hash function that takes a data item and a system node on the input, for example it may use their IDs, and computes a hash on the output. To determine the node where a given data item should be placed, the algorithm computes hashes pairing the data item with each system node and takes the maximum of the resulting values. The node that was on the input for that maximal hash is where the data item should be placed. In other words, node providing the highest hash wins the data item.

[5]The paper authors [22] solve the Bin Packing problem using linear programming.

Rendesvouz hashing has similar positives as consistent hashing. It can be performed in a distributed manner and it requires little data item movement when a node is removed from or added to the system. However, in case of an incoming node, all original nodes must iterate through their data items and test whether a particular item should be moved or not – at consistent hashing, only the ring neighbors must do this.

Unfortunately, due to time restrictions, we have not investigated this technique in such detail as consistent hashing. From what we discovered, this technique is not as widely used but compared with consistent hashing, it shows some promising potential – as presented in a recent paper [23], which deployed rendezvous hashing into *Cassandra* and compared it with *Cassandra*'s implementation of consistent hashing.

5. Thesis Concept Evolution

With progress of our work on this thesis project, its concept was changing as we were getting aware of existing related research and the environment of *Sklik*. In this chapter, we describe the original concept of this thesis and reasoning behind the changes.

5.1 Original Concept

At the beginning of our work, we were attracted by advantages of architecture based upon a remote database queried by slave servers of the ad system, presented as RInK architecture earlier in Section 4.1.

The slave servers would query the database to obtain a list of matching keywords in the *FindMatchingKeywords* function in Algorithm 1 from Section 2.2. They could also load other data from such database.

The advantages seemed quite promising:

- Using a third-party database providing integrated scaling would outsource the problem of scaling to the database. That would decrease effort needed to further extend and maintain the ad system, which indirectly lowers the costs on development. Of course, the drawback would be inability to control details of the data retrieval, which might be an issue – but we didn't have any concrete problematic scenarios at that time for this.
- Furthermore, it would increase cohesion of the system since the data storage and data processing would be encapsulated in separate components. This would, again, lead to decreased development effort and costs.
- And finally, there would be a good chance on performance improvement as database people probably provide more performant solutions for data storage and retrieval than ad system developers, simply because they concentrate on that full-time.

Our vision was to use a remote NoSQL database as we identified that the use case perfectly fits the NoSQL data access characteristics, namely: (1) the keyword searching algorithm wants to access the data in an aggregate-oriented manner, i.e. data are stored in a single form, exactly as they are needed by the algorithm, and retrieved using a single indexed key, (2) we need to retrieve the data fast. In addition, we aspired after open-source solution, which is another commonly defined NoSQL database characteristic.

Therefore, our original concept of this thesis was to redesign the ad system architecture in this manner and compare it with the original implementation.

With this idea in mind, we spent quite an amount of time with deployment of a NoSQL database into the keyword searching mechanism (which has been described in Section 2.2) and with optimization of this solution. We present our results in Section 6.6 and Chapter 7.

However, this approach proved not to be ideal, as discussed in the next section, and not even possible with respect to the other parts of the ad system, as detailed in Section 5.3.

5.2 Change of Architecture

When we came across the paper of Adya et al. [9], we changed our mind about the originally intended architecture of the ad system, which is presented in the previous section. Not because the reasons for that architecture would be invalidated but we put performance first since the ad system is expected to handle heavy load.

Furthermore, our originally intended experiment, which should have compared the *RInK* architecture with the architecture of stateful nodes (as defined in Section 4.1), stopped being so interesting as the paper provided results of a similar experiment, even though they focused mainly on performance.

We decided to change our goals to implement a *LInK*-like architecture and, in our experiments, to focus on comparison of different approaches of data distribution to slave servers, playing with both sharding and also replication.

Even though our previous work proved not to be exactly useful for our later goals, there is one feature that we needed even then. The original ad system's barrels do not allow incremental changes of stored data, which is required for the *LInK's* dynamic redistribution of data – the original barrels can be only replaced as wholes. Our barrels, from the referred previous work, allow this.

We decided not to go as far as pure *LInK*. We use load-aware sharding and distribute keys together with values. However, we did not implement a fully domain-specific interface for the key-value stores. The reason is the current state of *NoSQL* databases – getting a really high-performant store with an interface that could be called domain-specific is at least challenging and out of scope of our thesis project.

One of the data distribution approaches we planned to experiment with was inspired by the *TARS* algorithm from the cache-aware architecture discussed in Section 4.2. That is, to use slave servers as only cache views on the backend database and to distribute queries so that they are processed by more-or-less still the same slave servers. We hoped that the system would have high throughput since a query would not have to be processed by more than one slave server, and, at the same time, this would be also fast because of the caching.

Later on, as we were getting more closely familiar with *Sklik*, we discovered that we cannot be so liberal in selection of compared techniques for our experiment and we must primarily stick to sharding approaches. Details are left for the next section.

5.3 Need for Sharding

Being already mentioned in Sections 1.2 and 2.2, all keywords matching a query must undergo the auction routine outputting the best keywords with respect to relevancy of the keyword to the request and profit for the advertising company.

Time needed to perform the auction routine depends on the number of keywords on the input and it is the auction routine that is the most time-consuming process at slave servers. Since we want to finish in a given timeout, and preferably faster, we desire to minimize the time spent in the auction routine. That means that we are after minimizing the number of keywords matching a query present in a single slave server. In other words, we need to shard the data in order to have keywords distributed more-or-less uniformly over the slave servers so that the time spent in the auction routine is uniform across all slaves and thus minimal possible from the perspective of the whole system's user.

Therefore, we cannot use approaches that do not shard data in any way and we decided to concentrate on sharding algorithms and their comparison, which became the final goal of this thesis project.

5.4 Candidates for Experiment

We selected the following balancing techniques to be compared in this thesis' main experiment, which is described in Chapter 8:

1. The balancing technique implemented in the original system. Its description can be found in Section 2.4.
2. A random balancing of data to slave servers. Details are left to Section 6.5.
3. A balancing technique inspired by the *Weighted-Move* algorithm of *Google Slicer* [12], which has been presented in Section 4.5.

We originally wanted to compare more balancing techniques but that was out of scope of this thesis project with respect to the time constraints. We left this as future work.

More details are left for Chapter 8.

6. Design and Implementation

To briefly remind our goal, we needed to implement several different data balancing algorithms in the context of the *Sklik* ad system and conduct a comparison experiment. For this purpose, we decided to replace the ad system's data balancer with our custom implementation that would be efficient and, at the same time, modifiable to allow easy change of the data balancing algorithm.

We decided to go further and to develop a generic data balancer that is independent of the system context. This core part of our work is documented in Section 6.3.

To connect the generic data balancer with the system, a layer in-between must have been implemented. This layer consists of multiple components and is presented in Section 6.4.

The different implemented balancing algorithms are described in Section 6.5.

Section 6.6 is then dedicated to the work preceding our final decision about the thesis concept, which was discussed earlier in the previous chapter.

The ending sections of this chapter then puts the whole implementation into a single overall picture of the developed components and discuss aspects of the development such as testing and documentation.

But first, we put all into a single context in two opening sections of this chapter.

6.1 Definition of Terms

The data balancer thinks of the backend data simply as of a set of *data items* which each then further consists of *fragments*, no matter their semantics. If the data item in the real system cannot be further separated into smaller elements, a trivial fragment is the whole data item itself. The reason why we split data items further into fragments is that it fits the use-case of the *Sklik* ad system where we are balancing groups consisting of smaller keywords and ads. And since it fits one system, it is likely that it could prove useful in some others as well, even though majority of systems would not probably use this feature.

Each data item is identified by a *key* and each fragment by an *ID*. These identifications are used when a backend database communicates with the data balancer.

The complete data may potentially contain a very large number of data items, depending on the system. A data structure mapping each data item to a system node may take too much space and might not even fit into memory. To solve this, we use an abstraction called *slices*. A slice is a purely virtual entity used by the data balancer and has no intended analogy in the real data. It is used only

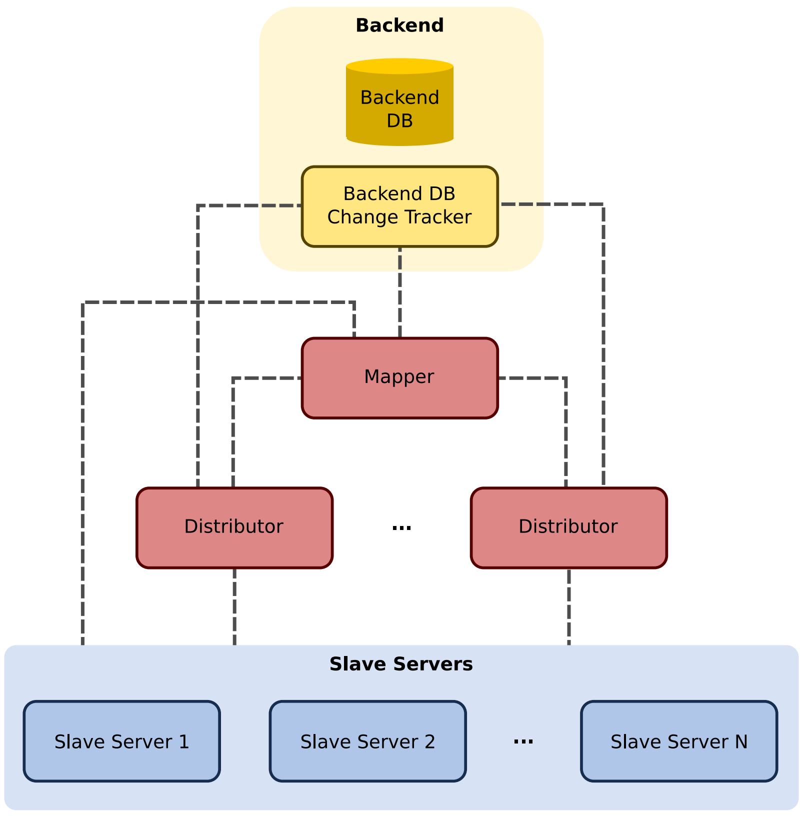


Figure 6.1: High-level overview of our implementation

to scale the granularity in order to control the space used by the data balancer's model. A slice represents an interval of keys. All data items with the key from that interval belong to the slice. You may notice the analogy to the idea of ranges from the third strategy to consistent hashing, which has been described in Section 4.4.1. Further details about slices will be discussed in Section 6.3.2.

Finally, in code of the generic data balancer, we call slave servers as *system nodes* because the implementation is independent of the *Sklik* ad system. In the following sections, however, we call them consistently as slave servers to keep the text simpler.

6.2 High-Level Overview

Our implementation is outlined in Figure 6.1. The data balancer, illustrated by the red-colored parts of the figure, is inspired by the two-tier architecture and other aspects of *Google Slicer* [12], which was described in Section 4.5.

In the *Mapper* component (further called also as *mapper*), we maintain a model of slice mapping to slave servers and run a balancing algorithm over this model to periodically recompute it (e.g. in order to keep the distribution balanced when the slave server load changes). The mapper has analogy in the *Assigner* component of *Google Slicer* [12]. Note that it is responsible for the model computation only, not for data transfer.

Changes of the mapper's model are sent to the *Distributor* components (further called also as *distributors*). These are responsible for the actual distribution of data to slave servers, including construction of the data in the form in which the slave servers use them. They replicate a part of the mapper's model and utilize it for propagation of data items and fragments to slave servers to which they are assigned by the mapper. Data delivery to slave servers is designed as a pull model, i.e. the distributor only prepares the data in some separated buckets and the slave servers ask for the data themselves.

The mapper and the distributors listen to data item and fragment changes from the backend database so that the model in the data balancer gets correspondingly recomputed and the data in slave servers get updated and potentially redistributed. The mapper also receives information about load from the individual slave servers, which is crucial for the data balancer to distribute data in a load-aware manner[1].

Tracking of data item and fragment changes in the backend database is considered to be outside of our implementation. A possible solution in case of SQL databases could make use of SQL triggers. For purposes of the experiment, we simulate this by a simple generator.

Because slave servers interact with the data balancer, we needed to put some code also into the slave server implementation.

And finally, as we needed to incrementally modify data in the slave servers' barrels and the original implementation of barrels does not allow their modification, we needed to replace the implementation with our own as well. We used *RocksDB* key-value stores [24] instead of barrels. More details about this replacement are left for the last section of this chapter.

6.3 Data Balancer

This section is dedicated to the generic core of our data balancer. Its whole design is driven by the desire to get (1) a generic solution that is independent of the concrete system where it is used, (2) a scalable solution that can adapt to different levels of utilization, and (3) a modifiable solution that allows easy changes of the balancing algorithm and other parts of the implementation.

Also, one of the most important requirements is to support incremental construction of data in the slave servers. That is, a change in the backend database

[1]As will be repeated later, the definition of what the load value represents is left as one parameterization of the generic data balancer core because this is specific to the system where the data balancer is deployed.

or a change in the data distribution should not force a complete rebuild of the data in the slave servers. Not allowing the data distribution change without a complete rebuild is one of the original ad system's drawback: a complete rebalancing during a time interval at night is required there. The initial state of the incremental changes is considered to be no data in the system. This presumption is no obstacle of deploying the data balancer into a system with already existing data. They can be recreated via incremental changes from the empty-system state.

Some aspects of our solution may seem as an overkill for the use-case in the *Sklik* ad system (e.g. the whole concept of slices is not necessary there) but the reasoning for such over-design is typically based on our quality requirements specified above.

The data balancer core described in this section cannot be used directly – in fact, it is a set of *C++* libraries, not an executable. It must be provided with strategies to fit the target system. For example, the first missing thing that comes to mind is that the core does not know what the data items and the fragments are – this must be defined when the data balancer is, as we say, *flavorized* for the concrete system.

To get everything into a single picture, the component view on the data balancer's architecture is presented in Figure 6.2. Individual parts and aspects of the data balancer, as well as of the figure, are subjects of the following subsections.

6.3.1 Centralized Design

Because the *Sklik* ad system is not large enough for the probability of network partitioning being too high, we decided to opt for a centralized design of the data balancer. This gives us better control over the whole process.

Even though distributed design is often preferred in distributed systems, due to its inherent robustness, and many load balancers are truly of a distributed nature, it is certainly not a must. For instance, *Google Slicer* [12], being a state-of-the art load balancer for very large distributed systems, uses a centralized design.

The central authority in our data balancer is the mapper component introduced earlier and presented in detail in Section 6.3.3.

6.3.2 Slice Model

In both the mapper and the distributor, we maintain an instance of a model that maps slices to slave servers. Because a slice is basically only a set or, more precisely, an interval of data items, this mapping actually defines mapping of every data item to which slave server it should be placed into. From the algorithmic

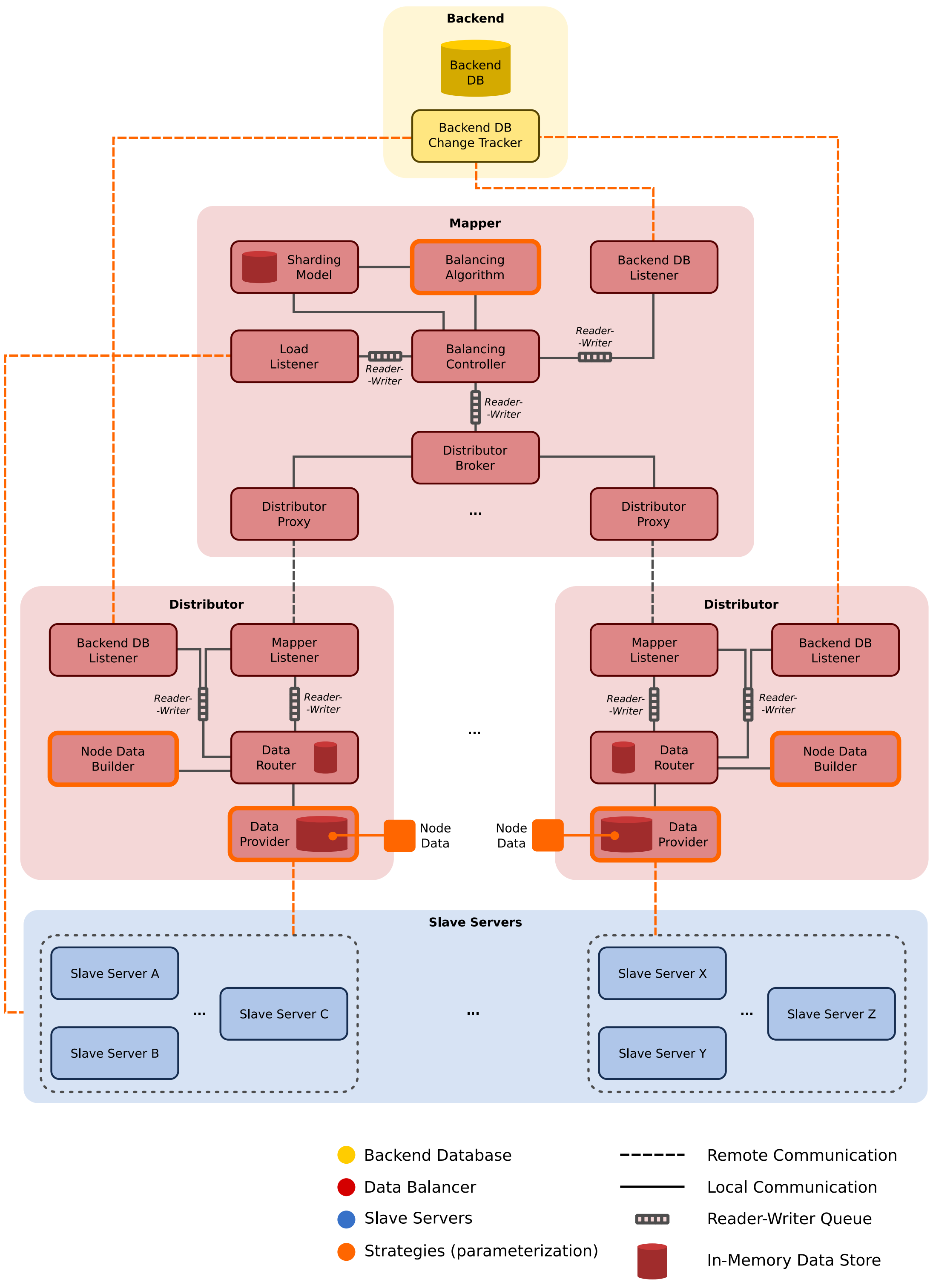


Figure 6.2: Data balancer component view

point of view, knowing this model makes the data distribution a trivial mechanical task (even though it is definitely not that simple in practice, but that is discussed in other sections).

By selecting the way slices are implemented, we may control how much memory space is taken by the model, which is crucial for systems storing a huge amount of data items. For example, using an in-memory map data structure to define what data items belong to a given slice does not save any memory space. On the other hand, keeping this map on the disk or in a database saves basically a maximum of memory space [2].

The selection of the strategy of how to define this map is one of the parameters of our generic data balancer core. In case of *Sklik*, we decided to use the in-memory map data structure because the number of data items is not that high there.

Similarly, as slices consist of data items, data items consist of fragments. Therefore, we might want to handle a similar map for fragments inside a given data item. We do not do this and leave this up to the strategies parameterizing the generic core[3]. To be more specific, the data balancer asks a strategy object to build all necessary data for a data item and it is up to the strategy object what fragments it decides to build.

6.3.3 Mapper

The *Mapper* component is responsible for maintaining the slice model, or, in other words, maintaining the definition of data distribution to slave servers. Importantly, note that the mapper is not responsible for actual physical data distribution – that is the task of the *Distributor* components.

In order to keep the model updated, there are three tasks the mapper must do. First, it must listen to data item changes from the backend database. Second, listen to load changes from slave servers. And third, perform the balancing algorithm repeatedly.

The slice model, together with other system modeling information, such as the number of slave servers, is held by the component *Sharding Model*. It contains all information about the system from the data balancer's point of view.

Another important component is *Balancing Algorithm*. It is called repeatedly to recompute the model according to its inner-implemented *balancing algorithm*. This algorithm can be changed, even online, when the data balancer is running. The particular algorithm implementation may be stateless but it may also hold

[2]Given that we use the slice-oriented implementation. Some other approaches might save up more space, of course.

[3]*Strategy*, or a *strategy object*, refers to the well-known *Gang-of-Four Strategy* design pattern [4].

Algorithm 2 Balancing Controller Operation

```
1: init model
2: updateModel(model)
3: modelChanges ← balancingAlgorithmInitialization(model)
4: commitChanges(model, modelChanges)
5: while not stopped do
6:     updateModel(model)
7:     modelChanges ← balancingAlgorithmIteration(model)
8:     commitChanges(model, modelChanges)

1: function updateModel(model)
2:     dataItemChanges ← readAll(R-W queue with backend DB changes)
3:     loadChanges ← readAll(R-W queue with load changes)
4:     model ← applyChanges(model, dataItemChanges, loadChanges)

1: function commitChanges(model, changes)
2:     model ← applyChanges(model, changes)
3:     sendChangesToDistributors(changes)
```

some inner data. However, this potential inner data is transparent to the mapper and ignored if the balancing algorithm is changed.

Propagating of model changes to the distributors is the responsibility of a component called *Distributor Broker*, which broadcasts it to *Distributor Proxies* that encapsulate the transportation to the distributors. Related consistency problems we needed to tackle are presented in Section 6.3.7.

As for threading point of view, remote calls (not necessarily *remote-procedure calls (RPC)* but any kind of remote communication) received from the backend database and from the slave servers are expected to be concurrent. Since all access the mapper's model, we needed to serialize them. We used *reader-writer (R-W) queues* for this purpose. These queues are read by a thread performs the balancing algorithm routine. On the other hand, out-coming remote calls to the distributors might be long-lasting. We decided to handle them by a separate thread exchanging information with the balancing algorithm thread via another reader-writer queue.

The whole mapper's operation is controlled by the *Balancing Controller* component and it is captured in Algorithm 2.

6.3.4 Distributor

The responsibility of the *Distributor* component is provision of data from the backend database to the slave servers according to the distribution defined by the slice model, which is being created by the mapper.

To fulfill this task, the distributor must (1) listen to slice model changes made by the mapper and update its slice model replica, (2) listen to backend database changes and propagate updated data items and fragments according to the slice model, and (3) construct the data in the form in which the slave servers need them so that it does not put another unnecessary load on the slave servers.

Listening to the backend data changes and the slice model changes is responsibility of the two appropriately named components *Backend DB Listener* and *Mapper Listener*, which are co-illustrated in the overview Figure 6.2. Similarly to the mapper, because these listeners receive changes concurrently and their further processing accesses the same distributor's data, we implemented serialization of these streams via reader-writer queues that are consumed by a single thread that runs the distributor's logic.

This logic is encapsulated in the main component of the distributor called *Data Router*. It manages the slice model replica, propagates the data item and fragment changes and utilizes another component *Node Data Builder* to create the final data for the slave servers.

It might seem that the object-oriented design might be better here as we broke the single-responsibility principle – the *Data Router* performs data propagation and slice model managing at the same time. The reason for this is that these two activities are very tightly coupled because the data consistency in the whole system is guaranteed by proper interleaving of these activities. Much more on this topic will be presented in Section 6.3.7.

The constructed data with attached propagation information are handed over to the component *Data Provider* that stores them in separated buckets called *Node Data* and provides these buckets to slave servers on their demand (via some remote communication). The slave servers are expected to poll for data repeatedly.

To allow control over the size of data transferred to the slave servers, which is appropriate in case of remote communication, we implemented a mechanism that limits the size of each *Node Data* bucket and when overflown, it creates a new *Node Data* where the additional data are stored. This might result in multiple *Node Data* buckets prepared for the same slave server. We store them in a queue, ordered as they are expected to be processed by the slave server, and they are provided to the slave server in this order.

Finally, because the slave servers' requests for the *Node Data* buckets are concurrent and the buckets are filled by the *Data Router* thread at the same time, a proper thread safety must have been implemented here. This topic is left for Section 6.3.9.

6.3.5 Distributor Scaling

As mentioned in the opening of Section 6.3, we wanted to implement a data balancer that is able to scale with the amount of processed data. The components that are responsible for data processing in our solution are the distributors[4]. Therefore, for our data balancer to be able to scale with the system, we made the number of distributors adaptable, even during run-time.

Each distributor is connected to the mapper via a dedicated *Distributor Proxy* and is responsible for a subset of the slave servers. This is illustrated by a dashed rectangle in the overview Figure 6.2. The slave servers in this subset do not know about any other distributor and the distributor does not know about any other slave servers.

A distributor responsible for the given subset of slave servers watches only data item and fragment changes that, according to the slice model, belong to these slave servers. Other data changes are ignored. As for the slice model in the distributor, it is replicated from the mapper's model changes as a whole, no restriction of the information to the slave server subset is made here.

The subset of slave servers handled by a distributor can be configured at run-time via an interface method of the distributor. The slave server has an address to its distributor stored in a configuration file – this might be an obstacle for potential dynamic scaling of the system but it can be easily solved by utilization of a configuration service.

6.3.6 Distributor Data Flow

This section discusses data flow in the distributor component as it is most important for understanding the data balancer and also for the following sections.

There are two input and one output data interfaces. All is illustrated in Figure 6.3. The inputs are (1) data item and fragment changes shown as red shapes in the top left corner, and (2) slice model changes whose illustration is in the top right corner and which consist of black ring key-space with blue slices and red data items that are placed around the ring according to their keys. The blue box with blue shapes in the bottom of the figure illustrates *Node Data* buckets introduced in the earlier sections. It is basically a set of data items and fragments prepared in their final form for the slave servers.

The input data, since their processing accesses the single distributor's slice model replica, are serialized via the illustrated reader-writer queues. Both queues are processed by the *Data Router* component, which updates the slice model with the incoming slice model changes and propagates the incoming data item and fragment changes to correct *Node Data* buckets. This is the most important part of the distributor for the two following sections, which are dedicated to data

[4]It is true that the mapper might want to take the individual data items into account as well, when balancing the data, but this is rather unexpected because of the implemented slice abstraction. Still, such mapper's balancing algorithms are supported but without any possibility of scaling.

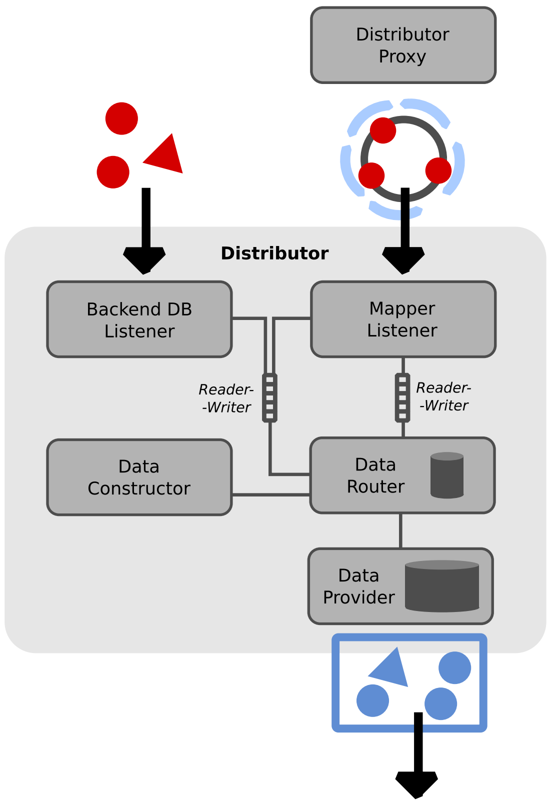


Figure 6.3: Distributor data-flow view

consistency.

For now, let us explain what data go further from the *Data Router* to the *Node Data* buckets. Naturally, we propagate all the changed data items and fragments as reported by the backend database. But additionally, we must also propagate data item changes that sometimes originate from the slice model changes – more precisely, when a slice is remapped from a slave server to another, data items in the slice need to be removed from the original slave server and created in the new slave server.

Note that slave servers do not send remapped data directly between each other. This would be an alternative design whose advantage might be more balanced network traffic in the system, but we rejected it because of the more complex logic related to data consistency – instead, we opted for the described centralized solution.

Before the data get to the *Node Data* buckets, they are processed by the *Node Data Builder* that creates exact data structures used in the slave servers. This helps to increase the performance of the slave servers as they do not have to do any preprocessing of the data and they can just focus on serving the requests. When the system performs some non-trivial offline precomputation of the slave server data, which is the case of *Sklik*, this might save a lot of computational resources and time[5].

The *Node Data* buckets are expected to be retrieved by the slave servers via some kind of remote communication (e.g. RPC).

As mentioned before, the buckets contain incremental changes to the existing slave server data. This is one of the advantages of our data balancer with respect to the original *Sklik* data balancer. We do not have to iteratively resend the complete slave server data but only the really necessary data is transferred, which saves network traffic and reduces the time needed by the slave servers to deploy new data.

6.3.7 Consistency Problems

Data consistency is one of the most important aspects of distributed systems – it ensures that the system behaves similarly-enough to a system running on a single computer. As mentioned in Section 1.1, there are multiple levels of consistency that may be guaranteed, which define what are the requirements of ”similarity” in the previous sentence. This section describes the reasoning behind the selection of our consistency model that is described in detail in the next section.

In Section 1.2, we stated that we target our work for systems that are satisfied with eventual consistency. However, some parts of a distributed system must usually satisfy a stronger consistency model in order to get the overall system's eventual consistency model.

For example, in our case, we cannot make the slice model replication only eventually consistent, because we may loose some of the distributed data in the overall context. Consider that the mapper issues such small slice model changes that they do not transform the slice model from a consistent state to another consistent state, just because the balancing algorithm usually does not do all modifications of the model in a single atomic operation but through a sequence of steps. If these changes would be delivered to the distributors without any further guarantees, the slice model replica in the distributor would be, at some moments, in an inconsistent state. And, obviously, the distributor might not deliver data correctly based on an inconsistent slice model.

Therefore the slice model changes must be applied to a distributor's slice model replica atomically, in subsequences that bring the model from a consistent state to another consistent state. It is similar to database transactions. We decided to call these subsequences as *quantums* and to pack all slice model changes issued by the balancing algorithm in its single iteration into a single quantum. The balancing algorithm is required to implement iterations so that they end

[5]This is not an advantage over the original implementation of the data balancing in *Sklik*, they do it similarly.

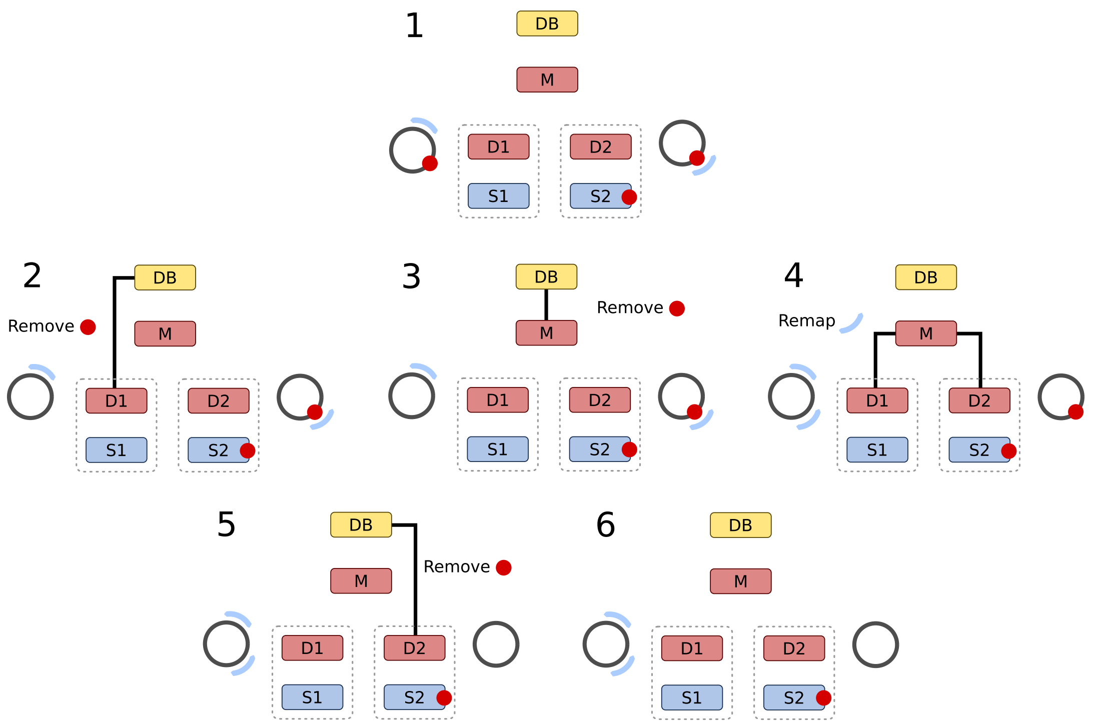


Figure 6.4: Data-loss scenario

with a consistent state of the slice model. A quantum is then delivered to the distributors and atomically applied to the distributor's slice model replica.

For a period of time during our implementation, we were thinking that the consistency model of the slice model replicas described above, which is just a little stronger than the eventual consistency model, would suffice to provide us with the overall eventual consistency model of the whole data balancer. I.e. that data in slave servers might appear in inconsistent state in the system but there would be some time points in which the data are completely consistent.

Later on, however, we realized that this is not true. This would work only in ideal scenarios. When we take nasty race conditions into account, we may still end up with data leaking out of the system. To provide one example for all, we illustrate a critical scenario in Figure 6.4, which is described in the following two paragraphs.

In the presented scenario, the picture number 1 represents the initial state of the system. There is a backend database, represented by the yellow component *DB*, which delivers changes to the mapper *M* and two distributors *D1* and *D2*, which are illustrated by the red-colored rectangles. Each distributor is responsible for one slave server. The slave servers *S1* and *S2* are identified by blue rectangles. A simplification of the slice model replica stored in each of the distributors is presented as a black ring key-space where the red circle on it identifies one concrete data item in the system and blue arcs around the ring designate slices that are mapped to the slave server handled by the particular distributor[6]. The red data item is placed in the slave server *S2*, which is illustrated as the red circle on the slave server *S2*.

In the following picture 2, the backend database *DB* issues a removal of the red data item. This change gets first to the distributor *D1*, which removes it from its model. Let's say it arrives to the mapper *M* and the distributor *D2* later – the whole scenario is about race conditions. The picture 3 shows arrival of the data item removal request to the mapper *M*. The mapper removes it from its model and, in the next picture 4, the balancing algorithm for whatever reasons issues a request for remapping of the slice, which was originally mapped to the slave server *S2*, to the slave server *S1*. Then, in picture 5, the data item removal request arrives to the distributor *D2*, which does remove it from its model but not from the slave server *S2* because the slice, where the data item belongs, is no longer under control of this distributor – it ignores any changes that are not the distributor's business (if it would not, other bad things would be happening in the system). The last picture 6 shows that even though the red data item should have been removed from the system, it stayed present in the slave server *S2*.

You could argue that the mapper could have let the distributors know, in the pictures 3 and 4, that the data item is being removed. But that would create problems originating from duplication of data item changes from the backend database. These changes would not arrive to the distributors exactly once. This would only result in different race condition problems.

The core of such race condition problems is the lack of synchronization between the backend database, the mapper and the distributors. Delivery of either backend data changes or slice model changes can be delayed for a long time during which the other changes may be processed and may break processing of the delayed changes, all resulting in an inconsistent state.

In the area of distributed systems, these problems of race conditions between dependent message processing can be solved by a well-known tool called *Lamport clock*, often called also as *logic clock* [25]. It serves for construction of an ordering of dependent events in a distributed environment so that causes precede consequences. Usually, it is implemented as a counter of messages where each message is assigned the counter's value as a timestamp identifying when the message originated. To create the described ordering, messages are being delayed until any preceding messages arrive. This ordering, called *causal ordering*, is not unique as non-dependent events can be arbitrarily interleaved.

Having causal ordering in the presented problematic scenario above, the slice remapping request from the picture 4 would never get processed in the distributor

[6]In the implementation, the distributor component is aware of all slices in the system, as we mentioned in Section 6.3.5. But for simplification, we illustrated here only slices assigned to the distributor's handled slave server.

D2 earlier than the data item removal request. Simply because these actions would become dependent by the knowledge of the mapper about the data item removal and this dependency information would get to the distributors together with the slice model change.

Details of our consistency model based on causal ordering, called *causal consistency model*, are left for the following section that is dedicated to the model's implementation in the context of our data balancer solution.

6.3.8 Consistency Model

To summarize the previous section into a single idea: because we build data for the slave servers incrementally, we are most concerned with ordering causes before consequences. This is allowed by the causal consistency model, which we had to implement.

We used a single Lamport clock [25] to identify the state of data in the backend database. Often, there are multiple Lamport clocks used in a distributed system (these are called *vector clocks*) because there are multiple sources of data changes – typically, each node can accept read and write operations. This is not our case since data changes come only from a single entity, the backend database.

As changes of the slice model, originating from the balancing algorithm, depend on the state of data items in the backend data (slices are a representation of data items), they must be labeled with our Lamport clock as well – to designate the state of the backend database on which the slice model changes are based.

Then, there are load changes reported by the slave server to the mapper. These serve for the balancing algorithm to have an idea about the system's load. However, these are only secondary information in the system – it is quite unimportant whether and when a particular load information arrives to the mapper – the goal is to have an estimate of the system load. No Lamport clock are therefore carried by the load changes.

Algorithm 3 presents the core of the mechanism that ensures causal ordering of changes processed by the *Data Router*. The idea is that the distributor never processes slice model changes (a quantum) if it has not seen all backend data changes that were seen by the mapper in the time when it issued the quantum (the quantum has the clock of the last data item change processed by the mapper). And on the other hand, the distributor never processes backend data changes if it has not received a quantum from the mapper that has seen these changes. This way, all changes are processed by the distributor in the same order as in the mapper, which can construct causal ordering quite easily – by putting the quantum in the ordering after the last data change processed by the mapper when this quantum originated.

This might seem as a little strict restriction. And that is quite right since

Algorithm 3 Data Router Consistency Algorithm

```
 1: $distributorClock \leftarrow$ **min**
 2: **while** not stopped **do**
 3:     $quantum \leftarrow$ WAITFORNEXTQUANTUM
 4:     $requiredClock \leftarrow quantum.clock$
 5:     **while** $distributorClock < requiredClock$ **do**
 6:         $dataChange \leftarrow$ WAITFORNEXTDATACHANGE
 7:         **if** $dataChange.clock > requiredClock$ **then**
 8:             PUTBACK($dataChange$)
 9:             **break**
10:         PROPAGATEDATACHANGE($dataChange$)
11:         $distributorClock \leftarrow dataChange.clock$
12:     PROCESSQUANTUM($quantum$)
```

it could be more liberal as the mapper could place the quantum into the order of backend data changes less precisely, at least in some cases. But note that all distributors are independent and do not have to be in the same state, so this is not as strict as strong consistency. And note that a slice model change may result in data item remapping from one slave server to another, in which case, in general, the quantum may have to be placed into the order quite precisely. Consider a situation when the slice remapping change must be put into the order when there are only data item changes belonging to that slice. So yes, it may be less strict but we decided to implement the causal ordering creation this way because it is straight-forward and not too strong at the same time.

Also note we enable processing of multiple subsequent quantums without any incoming backend data changes. Therefore the system can react on load changes properly, without the need of receiving a backend data change request.

With this implementation of causal consistency of distributors' change processing, the overall data balancing process is eventually consistent with data balancing that would be made by a data balancer with a mapper and all distributors being implemented in a single monolith node. That is, slave servers may be shortly in an inconsistent state but, after some time, consistency errors get fixed.

The rest of this section is dedicated to scenarios that we expect to be relevant in the case of real day-to-day use of our data balancer. For each, we describe an algorithm that ensures that the eventual consistency would not get broken:

- **Change of distributor's handled set of slave servers.** In this scenario, we want to get the same slice model in all distributors, that are related to the desired change. This makes them equivalent with respect to data change processing, i.e. they would propagate data changes identically if they would handle the same set of slave servers. In order to get the same slice model, we want to make all the related distributors to be in the same Lamport clock time[7]. This can be done by freezing the stream of backend data changes for a while. This will lead to distributors processing the rest of requests they currently have and they all stop waiting in the same state, having the same Lamport clock time. After this we can change the sets of handled slave servers and unfreeze the backend data changes stream again.

- **Scaling of distributors.** For this scenario of scaling the number of distributors, we implemented a special kind of *quantum*, which we call *reinitializing quantum*. This quantum contains slice model changes that bring a slice model from the state after its fresh initialization to a specified state. When the mapper receives a request of the distributor set change, it inserts a reinitializing quantum to be emitted just before the next ordinary quantum. This way, all slice models in distributors get rebuild to the state of the mapper's slice model, which gets any new distributors to the state when they are ready to serve incoming backend data changes and slice model changes.

  Equipped with reinitializing quantums, these are the steps needed to be performed in this scenario. We can do a similar trick as in the previous scenario. We freeze the stream of backend data changes, wait until the distributors get stabilized, change the sets of distributors and handled nodes and then we unfreeze the backend data change stream. The mapper automatically sends a reinitializing quantum when the distributor set is changed and thus any new distributors get initialized into the same Lamport clock time as the other distributors, just before they start processing any other incoming backend data and slice model changes (again).

  This way, the number of distributors can be changed without any loss of data or its consistency.

- **Scaling of slave servers.** This should be supported by the implemented balancing algorithm that should react on the changed slave server count stored in the *Sharding Model* component. The mapper has an interface method to control this count.

  The balancing algorithm is expected to remap slices from any removed slave servers to the other slave servers or to remap slices to any new slave servers from the existing slave servers.

- **Restart of a given slave server.** This scenario is currently unsupported by our implementation of the data balancer but it is quite easy to cover this feature. We only provide here an outline of this extension: The distributor contains the slice map model from which it can generate all data

[7]That is the *distributorClock* from Algorithm 3.

items belonging to the slices mapped to that slave server. Therefore it can construct the whole data of a slave server on a demand. Thus, if this was implemented, we could only freeze the backend data changes stream, wait until stabilization, ask the distributor to generate the complete data for the restarted slave server and unfreeze the stream again.

In the described scenarios, we use an action of freezing the backend data changes stream. Note that we did not implement any mechanism for this because it is specific to the concrete system where the data balancer is deployed and we did not implement any even for the *Sklik* ad system because tracking of changes in the backend database is out of scope of this thesis project.

6.3.9 Node Data Storage

Because of performance reasons, we release *Node Data* buckets to slave servers in bursts. Otherwise, as *Node Data* buckets are often being updated by the distributor with new data, we would have to, and we originally did, implement a quite strict locking mechanism, which proved to be a bottleneck when the data balancer was experiencing an intensive stream of new data from the backend database to the slave servers. We found out that even multiple levels of locking breaking down the mutual exclusion sections were not sufficient.

We ended up with a trick similar to the well-known utilization of two buffers being exchanged when one of them is ready (e.g. GPU memory buffers). One buffer is always read-only and one write-only. Each buffer is, in our case, a data structure of *Node Data* buckets.

This data structure is a simple key-value structure where a key is a slave server's identification and a value is a queue of *Node Data* buckets in the order as they should be retrieved and processed by that particular slave server.

The distributor fills the write-only data structure until it stores such data amount that is worth the time penalty of locking for the data structure exchange. At such point, we merge the data from the write-only structure into the read-only one. Note that we cannot do a pure structure exchange as the read-only structure may still contain data that has not been retrieved by the slave servers yet.

6.3.10 Balancing Algorithm Implementation

The data balancer allows change of the balancing algorithm by providing a different implementation of an abstract class called *Balancing Algorithm*. This way the balancing algorithm can be changed even during run-time.

This abstract class contains two methods that need to be overridden. The first method is expected to contain any initialization logic of the algorithm and is called when the mapper is initialized. The second method is called in iterations during normal operation of the mapper.

Each of the methods can manipulate the data distribution by basically three kinds of elementary operations: (1) mapping a given slice to a given system node, (2) splitting a slice in a given key into two smaller slices, (3) merging two adjacent slices mapped to the same system node.

Further details about these operations are properly documented in the implementation and not described in this text to keep the thesis scale in reasonable levels.

Any call of the balancing algorithm's method is required to produce a valid data distribution where each slice is mapped to a valid system node. Otherwise the data balancer does not guarantee consistency of data in the system.

6.3.11 Remote Communication

The mapper and the distributors are designed to run on separate nodes in the system network. Therefore, any communication between them and the rest of the system must be performed over network. The remote communication channels are illustrated as dashed lines in the overview Figures 6.1 and 6.2 from the opening of this chapter.

All remote communication endpoints were properly encapsulated in our implementation and therefore a change is easily possible. We practiced it even ourselves – more details are left to Section 6.4.3.

6.3.12 System Initialization

Being already mentioned in the introduction for Section 6.3, the initial state of the system is expected to have no data in it.

This is theoretically no problem if the data balancer is to be used in an entirely new system, and it is even not a problem for deployment of the data balancer into systems with already existing data because they can be filled by incremental changes pretending that all data just happened to be created when the data balancer started.

But it starts to be a problem with a need of frequent restarts of the system or similar requirements. In this cases, we would like to speed up the initialization of the data balancer.

For this purpose, as we needed this feature as well in the experiment presented in Chapter 8, we introduced an API of the data balancer that allows to burst-initialize the mapper component.

The user is expected to send lists of data items that should be created in the initialization, the balancing algorithm's initialization routine is run to balance this initial data and the resulting slice model is passed to the distributors. Then, the user needs to send the data items to the distributors and they distribute the data to the slave servers.

This speeds up the initialization quite significantly. The mapper can run the balancing algorithm only once and by that it can get a slice model that is more-or-less final for the phase of the system initialization. Without the burst-initialization the slice model would be created during many balancing algorithm's iterations as new data items were coming one-by-one via the API for the normal operation. Furthermore, when the slice model changes too rapidly, it tends to affect the whole data distribution very badly because data are transferred back and forth between the slave servers as the balancing algorithm realizes that it did not know about that many another data items and the previous slice model is totally wrong. Finally, getting some kind of final slice model fast allows the mapper to send it to the distributors fast and they can start with the data balancing early.

However, it still is not ideal. Best would be to put the data somehow in the slave servers and then to burst-initialize the mapper with this already existing data mapping. At this point, the data balancer might had to rebalance a lot of data since the initial data distribution selected by the user might not be good-enough but it would be capable of normal operation very fast (even though it might not provide the best performance at the beginning due to the possible heavy rebalancing). We did not implement this functionality because of the limited scope of this thesis project.

6.3.13 Congestion Control

Related particularly to the previous section's topic but also to other scenarios where the data balancer is exposed to an intensive data change requests from the backend database, there is a need for congestion control.

For example, when a distributor experiences many incoming requests for data item creations, it may run out of memory because it would be creating the data for the slave servers faster than they would be capable of retrieving this data via a remote communication channel.

We implemented quite a simple mechanism to prevent such situations: counters of active entities that prevent creation of more than a specified number by those entities. We count any entities in reader-writer queues and the *Node Data* buckets. Each entity kind has a dedicated counter.

If a thread wants to create a *Node Data* bucket over a given limit, it is sent to a passive sleep until the *Node Data* count drops under this limit.

Limit exhaustion at reader-writer queues just tells the caller that the operation should be tried again later.

6.3.14 Data Balancer Evaluation

In this section, we list positive aspects of our solution that are worth mentioning in order to potentially compare our data balancer with some other data balancers.

We succeeded in our goals mentioned in the introduction of Section 6.3. Particularly, our data balancer has the following properties:

- The data balancer is independent of the system where it is deployed.
- It constructs data for slave servers incrementally, listening to the newest backend data changes.
- The data for slave servers can be prepared exactly as they are needed there, saving resources of slave servers for request processing.
- It is able to scale with respect to data size and data complexity (by scaling the number of distributors), and also with respect to data granularity (by choosing a right implementation of slices).
- It allows scaling with the number of slave servers as well.
- It can handle congestion by incoming requests and by created data so that the components do not run out of memory.
- It is easily modifiable in several aspects including the balancing algorithm implementation, remote communication technology, etc.
- There is no persistent storage throughout the data balancer, everything is kept in memory. This allows deployment of the data balancer in some specific kinds of cloud environments with no disk available. At the same time, this does not prevent scaling.

But to be fair, we would also like to present potential limitations too:

- With such universality of the solution, deployment of the data balancer in a concrete system may still be a non-trivial task. It requires no hard algorithmic work but the amount of required code might be as high as hundreds or a few thousands lines of code, as we did experience ourselves in case of *Sklik* (but that is quite a complex system).
- There is only one environment where it has been deployed so far. Each different deployment increases the overall quality of the solution. Therefore, there might be some issues that have not been observed and fixed yet.
- We did not target fault-tolerancy in any way as it is out of scope of this thesis project.

6.3.15 Data Balancer Deployment

The data balancer is packaged as a set of *C++* libraries. It needs to be completed with several components and definitions in order to deploy the data balancer into a particular system – we call this completion as *flavor*.

The following list overviews what a flavor is expected to define and implement in order to create a fully-working data balancer:

- It defines data items, fragments and *Node Data* buckets. The data balancer core is independent of the particular system and therefore it uses abstractions of these entities.
- It provides the *Node Data Builder* implementation. The data are provided by the data balancer to the slave servers in the form in which they are needed. This is specific to the flavor.
- It implements proxies for communication with the slave servers and the backend database. We wanted to make the data balancer core independent of a particular technology used for remote communication with the rest of the system. The flavor is supposed to implement this communication and call specified interface points of the data balancer.
- It implements a component present in slave servers communicating with the data balancer. The way how data are used in slave servers and how load information is collected is flavor-specific.
- It implements tracking of backend database changes.

Details are present in the user documentation of the data balancer, which is attached to the thesis, as further explained in Section 6.9. A complex example of the data balancer's flavorization is described in the following section.

6.4 Data Balancer in Sklik

The data balancer presented in the previous sections requires a set of additional components and definitions, which were outlined in Section 6.3.15, in order to be deployed into a concrete system. This section is concerned with details of this deployment into the *Sklik* ad system.

The first five subsections address the five individual list items in Section 6.3.15, i.e. first we discuss the definitions of data items, fragments and *Node Data* buckets, then we present *Node Data* bucket building, RPC communication, a component used inside the slave servers and simulation of backend database change tracking. The last section discusses the selected scale of the data balancer.

A high level component view of the flavor, whose aspects are to be described, is illustrated in Figure 6.5.

6.4.1 Data Entities Definition

It is obvious that data items should not be too large because big objects cannot be balanced uniformly. Too small objects, on the other hand, may cause performance problems if their number is too high. This is solved by the slice abstraction of

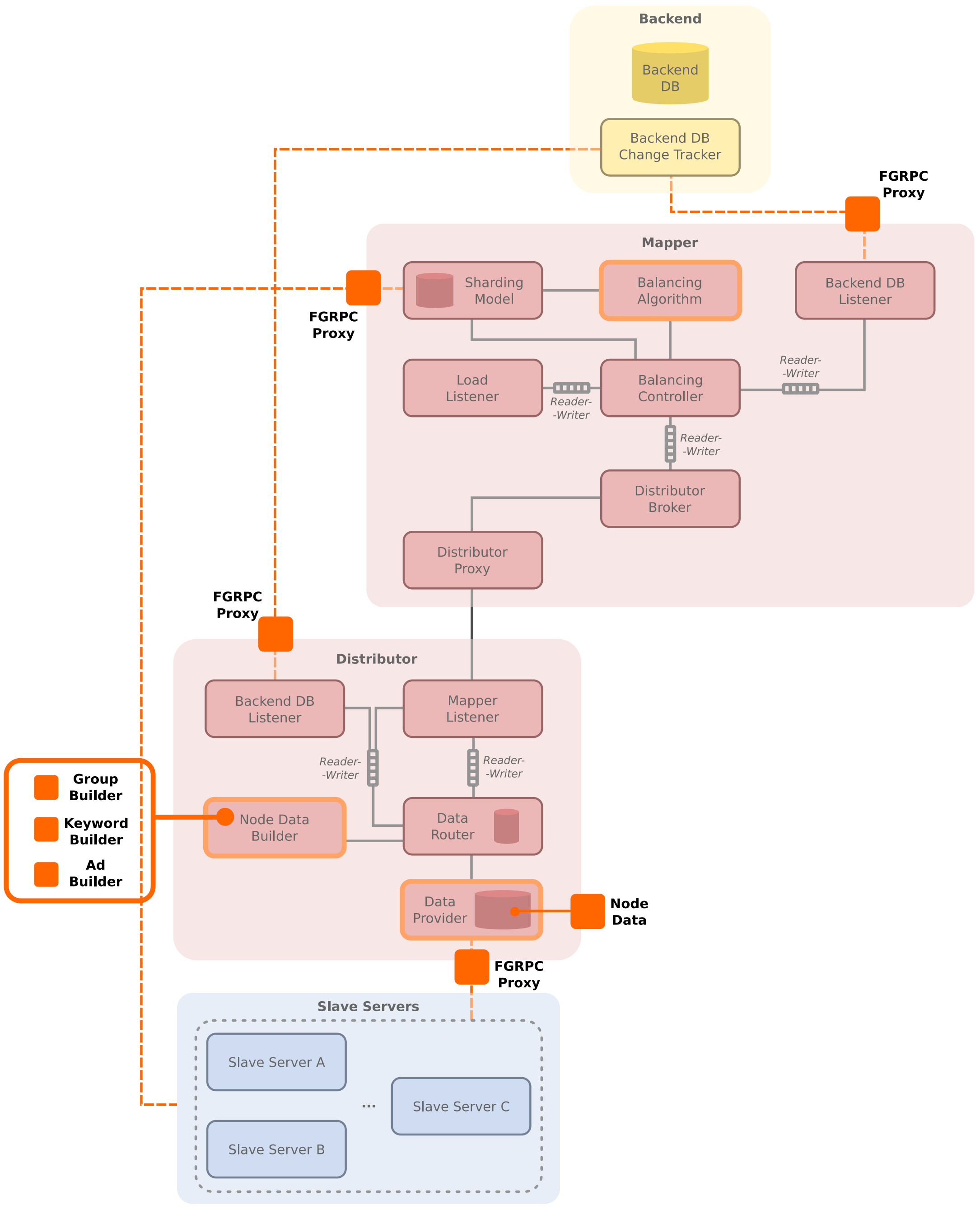


Figure 6.5: Sklik flavor of the data balancer

the data balancer described in Section 6.1, which enables artificial change of data items granularity. But still, there may be a performance penalty for too small data items[8]. The best approach is, without any surprise, to select the data item granularity not too big and not too small.

In the *Sklik* ad system, there are several data item granularity levels visible on the first sight. As presented in data hierarchy in Section 2.1, there are advertisers, campaigns, groups, keywords and ads.

Advertisers are the balanced entities in the original system and they proved to be too large with respect to future perspective. Ads and keywords are small but they are not independent, they are combined inside a group. This would force us to balance the groups to all slave servers that contain some of the group's keywords or ads, and thus duplicating the group's information in multiple slave servers. This and also other confidential implementation-specific details makes ads and keywords inappropriate for being the balanced data items.

Therefore, we found groups to be the optimal balanced entities. They are small and easy-enough to balance. Duplication of data shared by several groups, such as campaign and advertiser information, in all slave servers, where these groups are distributed (similarly as in the previous paragraph in case of duplicated group information) is not such a big problem. The fragments are then keywords and ads.

The *Node Data* buckets consist of quite a large number of lists. There are two lists for each kind of data entity: a list of created entities, which are already in the form as they are to be stored in the slave server, and a list of identifications of entities that should be removed from the slave server. When a *Node Data* bucket is retrieved by the slave server, all the server has to do is to add the created entities into its data structures and erase the entities that are requested to be removed.

6.4.2 Data Building

Our data balancer requires a *Node Data* bucket building strategy to be defined by the flavor. The *Sklik* flavor uses a strategy parameterized by a set of another building strategies, for instance a keyword building strategy, an ad building strategy, etc. This is illustrated in the bottom left part of Figure 6.5.

The intended implementation of these builders is based on retrieval and preprocessing of data from the backend database. However, due to our restrictions of access to the *Sklik* ad system's data, we needed to use a different implementation while still enabling the originally intended one with the backend database access.

[8]The performance penalty does not scale with the number of data items much. It may be something like overhead for retrieving data items of a slice from a database instead of just reading it from memory.

We retrieve the data from original system's barrels, which were described in Section 2.3. Because the data are already preprocessed there, it got us rid of this part of implementation but on the other hand it made the data retrieval more difficult as we could not use SQL or similar query languages.

6.4.3 Remote Communication

Remote communication into and out of our *Sklik* data balancer, which is illustrated by the orange dashed lines in Figure 6.5, is done via an RPC technology developed in *Seznam.cz* called *FGRPC* that is built upon *XML-RPC* [26] and *Protocol Buffers* [5].

Originally, we used the *gRPC* technology [27] because it is a widely spread and a modern approach for remote communication. However, after some time, we had to replace it with *FGRPC* due to compatibility reasons with the rest of the ad system.

6.4.4 Communication in Slave Server

The slave server needs to communicate with the data balancer (1) to retrieve *Node Data* buckets belonging to it and (2) to report load information.

Both these tasks are handled by a component called *Data Balancer Proxy* running in a separate process on the slave server. The reasons for a separate component and a separate process are that we did not want to slow down the slave server's request processing by any non-trivial communication with the data balancer, and it is also better from the design-quality point of view.

Data Balancer Proxy works in iterations repeated in given time intervals. Its operation is captured in Algorithm 4.

The load information is collected inside the slave server implementation and sent via a *Boost* [28] message queue to the process running the *Data Balancer Proxy*. A message contains an identification of a slice that is responsible for a data item whose keyword has been accessed during the slave server's operation. In other words, we track which slices are most used.

The information about what keywords in a slave server are present in what slices is sent via *Node Data* buckets: it is a part of the data structure representing a keyword. As slices change in the mapper's model, the information about slices stored in the slave server may get outdated. This is a little problem when the balancer gets a load information of a slice that no longer exists in the mapper. We solved this by spreading the load among the slices that cover the reported outdated slice, i.e. the slices that have a non-empty intersection with the reported slice.

Algorithm 4 Data Balancer Proxy Operation

```
1: while is time for next iteration do
2:     REPORTLOAD
3:     UPDATEDATA
1: function REPORTLOAD
2:     sliceLoadMessages ← READALLSLICELOADMESSAGES
3:     for each sliceLoadMessage ∈ sliceLoadMessages do
4:         AGGREGATELOAD(sliceLoadMessage)
5:     if is time for next load report then
6:         SENDAGGREGATEDLOADINFORMATION
1: function UPDATEDATA
2:     OPENROCKSDBS
3:     nodeData ← GETNODEDATA
4:     while nodeData not empty do
5:         APPLYNODEDATATOROCKSDBS(nodeData)
6:         nodeData ← GETNODEDATA
7:     CLOSEROCKSDBS
8:     COPYROCKSDBS
9:     SWITCHROCKSDBSINSLAVESERVER
```

The *Data Balancer Proxy* reads the load information from the message queue, aggregates it and in regular time intervals, which may be different from the *Data Balancer Proxy* intervals, it reports all collected load information to the data balancer.

The *Node Data* buckets, on the other hand, are retrieved from the data balancer via RPC calls in every interval of the *Data Balancer Proxy* and they are processed by applying all contained data creations and removals to *RocksDB* key-value stores. Remember that *RocksDB* stores are used as replacement of the original barrels introduced in Section 2.3. Then, copies of the *RocksDB* stores are created and the slave server is told to use these copies as new barrels for its operation[9]. The old barrels used by the slave server are thrown away.

Note that the *Node Data* buckets are retrieved from the data balancer in bursts which always end by provision of an empty *Node Data* bucket. This helps the performance of the whole data distribution, especially in the beginning when the system is intensively filled with initial data. The reason of the performance boost is that the *RocksDB* stores do not have to be opened, closed and switched into the slave server on every *Node Data* bucket retrieval, and thus the *Data Balancer Proxy* can query the data balancer for *Node Data* more frequently.

[9]This barrel-switching functionality is a part of the original *Sklik* ad system.

6.4.5 Backend Database Change Tracking

As already mentioned earlier, we consider tracking of changes in the backend database to be out of scope of this thesis project.

For the purpose of the experiment in Chapter 8, we used a simple ad-hoc generator that simulated the backend database change requests.

6.4.6 Scale

Being already explained in Section 6.4.2, we did not have to implement data construction in the form it would be implemented in the real use-case of the data balancer in the *Sklik* ad system, which saved a lot of computation resources that would be otherwise needed in the data balancer.

Therefore, for the purpose of the experiment, we could get along with only a single distributor that was even running on the same node as the mapper. This way, no RPC was needed for the communication between them since they both ran in the same process and could communicate via direct calls. All is illustrated in the overview Figure 6.5.

6.5 Implemented Balancing Algorithms

We implemented two balancing algorithms into our data balancer: (1) a random-based and (2) one inspired by the *Weighted-Move* algorithm of *Google Slicer* [12]. Both are compared in our main experiment described in Chapter 8.

As described in Section 6.3.10, a balancing algorithm is defined by two routines: (1) an initialization routine and (2) an iteration of normal operation.

The random-based algorithm is quite simple and both the routines are implemented the same following way. The algorithm goes through all slices in the slice model and for each performs these steps:

1. If the slice contains more than one data item, it splits the slice into smaller slices, all containing exactly one data item. Each is mapped onto a random slave server.
2. Otherwise, if the slice has not been mapped yet, the algorithm randomly maps it.
3. Otherwise, if the slice contains no data item, the algorithm merges it with its neighbor.

Our variant of *Weighted-Move* algorithm is very close to how the algorithm is presented in the paper introducing *Google Slicer* [12]. We summarized the paper's description in Section 4.5. We use the same initialization routine and the same iteration routine. The only two differences we are aware of are: (1) that we do not assign random keys to the data items using a hash function but we assume that the data item IDs are already random-enough, and (2) that while the original *Weighted-Move* algorithm uses also strategies working with replication factors, we have the replication factor fixed to one and thus we utilize only the strategy of redistribution of data from a slave server to a slave server.

6.6 Keyword Search

This section is dedicated to the implementation related to the original concept of this thesis, which was introduced in Section 5.1. To remind it, we aimed for a system architecture where there would be a remotely-accessed NoSQL database that would provide data to the slave servers. As a part of the intended solution, we utilized a *RocksDB* key-value store to replace barrels of the original system in the ad search algorithm, which has been introduced in Section 2.2.

Although the direction of the thesis changed, the implementation done during this phase of our work proved useful even in our new direction. The original system barrels do not support incremental building, which is a core feature of our data balancer, while a *RocksDB* store does.

The following subsections are organized as follows. First, we explain our reasons for selection of the *RocksDB* database. Then, we briefly overview the evolution of this part of our system and we finish by explaining how it was deployed into the existing slave server implementation.

6.6.1 Reasons for RocksDB

Reasoning why to choose a remotely-accessed database and why we were attracted to NoSQL has been already explained in Section 5.1. In addition, because of the large and growing data amount in the system, we were looking for a scalable database that would be performant enough so that a single database can serve all the slave servers. Also, because of the data amount, we were looking for a database that offers disk storage of the stored data. Furthermore, an open-source solution was preferred.

The ad search algorithm needs only a key-based access to the database, i.e. no database's understanding of the values is required. This is the reason why we primarily focused on simple key-value stores as we believed that these could give us the better performance than databases that are concerned with structure of the values.

Due to the required disk storage, we put key-value stores such as *Redis* [29] aside. We concentrated only on databases that allow both memory caching and disk storage.

What we considered important during our research of existing key-value stores were benchmarks. We are aware of the fact that benchmark results in this area are quite specific to the concrete benchmark setup and database configuration, but still we see them as a good-enough lead. Unfortunately, we experienced that finding a single benchmark that would be absolutely relevant to our case is not easy.

Based on opinions and benchmarks we discovered [30, 31, 32], we thought the best choice might be a database based on *RocksDB* [24], which is a C++ library used as an engine for other databases. And a good standalone database, which would offer built-in scaling and which would be based on *RocksDB*, appeared to be *ArangoDB* [33], even though a connection to it from a C++ code might be a bottleneck.

6.6.2 Implementation Evolution

Because we aimed for fast retrieval of data from *RocksDB*, we were trying to optimize the keyword search implementation. This process of development lasted a long time during which we needed to solve several more-or-less common issues related to such task.

First of all, we learnt that aggregate-oriented retrieval of data from a NoSQL database is not ideal with respect to performance when there are big aggregates and the client program uses only a small part to decide whether to use the aggregate further or not. In our case, the decision whether a keyword matches a given query, which is a part of Algorithm 1 from Section 2.2, requires only a relatively small portion of whole data stored for the keyword. At the same time, a lot of keywords do not match the query so the rest of data read from the *RocksDB* does not get used at all. This reading of unused data, sometimes called as overread [9], hurts performance.

We had to break the keyword data apart into two separate *RocksDB* databases. The first one is called *Kw-Matching-Data RocksDB* and contains data needed for the decision whether the keyword matches the query – i.e. for the function *FindMatchingKeywords* in Algorithm 1. The other is called *Keyword RocksDB* and maintains the rest of the keyword data. This data split goes against the NoSQL's principle of aggregate-oriented access to data but one can think of this data to be actually two separate aggregates. This idea was already present in the original implementation as the keyword data were also split into multiple barrels, even though the split and its reasoning were a little different.

Another our experienced issue is that there are quite big performance differences between available modern serialization technologies.

Last but certainly not least, we confirmed that configuration of (*RocksDB*) database plays a big role in performance.

The performance measurements and further details related to these issues are described in Chapter 7.

The final implementation uses two *RocksDB* databases, as described above, together with *Protocol Buffers* [5] and *Flat Buffers* [34] as serialization technologies.

Kw-Matching-Data RocksDB is optimized for requests of values with a common prefix of keys – as Algorithm 1 uses this database to retrieve keywords with the same given first word. Because values from this *RocksDB* are very frequently requested, *Flat Buffers* is used as the value serialization technology due to its better performance compared to *Protocol Buffers*.

If the keyword gets to further processing, the rest of its data are retrieved from *Keyword RocksDB*, which is optimized for point look-ups, i.e. requests for a single key-value pair. Because deserialization of these values is not so frequent, *Protocol Buffers* are utilized for their (de)serialization. We could have deployed *Flat Buffers* here as well but we decided not to, as we wanted to keep the code simple and *Protocol Buffers* are much easier to use.

6.6.3 Deployment

The described implementation forms a replacement of a library that was responsible for a non-trivial part of the ad search algorithm presented in Algorithm 1 in Section 2.2. More precisely, we replaced the part referenced as the *FindMatchingKeywords* function that finds matching keywords for the given input query. Our new library uses the original code but it is greatly refactored so that is utilizes our new *RocksDB* databases and it does so efficiently.

For our library to be deployable into the implementation of the slave server, we must have developed an adapter of our *RocksDB* databases to fit the interface of the original barrels. After this, the *RocksDB* databases are transparent to the slave server.

Still, we must have altered the implementation of the slave server because we slightly changed the interface of our new library implementing the matching keyword search. This modification is, however, not too significant.

If you are interested in the exact overview of our libraries, see the next section where our complete implementation is put into a single overall picture.

6.7 Overview of Implemented Components

Now, when we have presented individual aspects of our implementation, we can overview, without any need of further explanation, the components that we de-

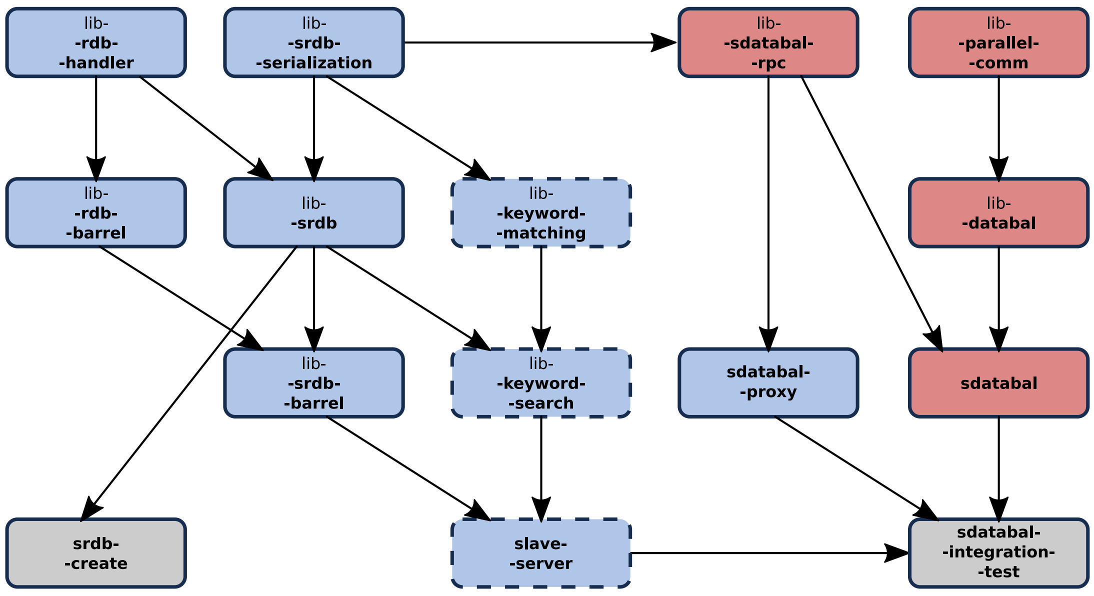


Figure 6.6: Overview of all developed components
Our completely new components are bordered with a solid line. Modified existing components have a dashed border. Arrows represent compile-time dependencies, always going in the direction to the dependent component. Blue color designates those components that are present in slave servers, red colors components that are designed to run outside of the slave servers as a part of the data balancer. Grey is the color of auxiliary components that are not supposed to run in production.

veloped[10]. They are illustrated in Figure 6.6, together with dependencies between them.

Our implementation can be divided into two parts, which are illustrated as blue and red colored parts in the referenced figure. The blue part contains components that run inside a search server. Majority of them relate to searching of matching keywords to a given query, which has been discussed in Section 6.6. The red part, on the other hand, consists of components that are run outside of the slave server as a part of the data balancer. These are described in Sections 6.3 and 6.4.

The following list offers a short description for each of the illustrated components. Details are omitted as they have been already discussed in the earlier sections.

- ***lib-rdb-handler***: Implements a proxy to a *RocksDB* database that eases the access to it and provides optimized configurations for different kinds of workloads.
- ***lib-rdb-barrel***: An adapter of a *RocksDB* database so that it can be used as a barrel inside the slave server implementation.

[10]Note that some existing components of the system that we modified are not presented because of the confidentiality of the information. These modifications, however, are only marginal.

- ***lib-srdb***: Definitions of all *RocksDB* databases that are used inside the slave server and whose data are distributed by our data balancer.
- ***lib-srdb-barrel***: Definitions of barrel adapters for *RocksDB* databases from *lib-srdb*.
- ***lib-srdb-serialization***: Encapsulates definitions of *Protocol Buffers* and *Flat Buffers* used to serialize values inside the *RocksDB* databases from *lib-srdb*.
- ***srdb-create***: Tools for transformation of original barrels to our *RocksDB* barrels.
- ***lib-keyword-search***: A library containing implementation of the search of matching keywords for a given query. We are not the primary authors of this library, we only non-trivially refactored and optimized its code.
- ***lib-keyword-matching***: An auxiliary library to *lib-keyword-search*, which determines whether a given keyword really matches a given query. Note that this code does not come from us, it is a part of the original implementation, but the library resulted from our refactoring.
- ***slave-server***: The implementation of the slave server. There are only little modifications from our side.
- ***sdatabal-proxy***: A proxy ensuring proper communication between the slave server and the data balancer. It consists of a small library used in the slave server implementation for load collection and a component designed to run as a stand-alone process, which is responsible for the actual communication.
- ***lib-databal***: Implementation of the generic core of the data balancer.
- ***sdatabal***: Data balancer consisting of the generic core implemented in *lib-databal* and providing a layer of parameterization above it to fit the *Sklik* ad system.
- ***lib-sdatabal-rpc***: Encapsulates definitions of *Protocol Buffers* used in *FGRPC* communication between the data balancer and the slave server.
- ***lib-parallel-comm***: Provides an implementation of a reader-writer queue needed by *lib-databal*.
- ***sdatabal-integration-test***: Integration tests checking correct communication and cooperation between *sdatabal*, *sdatabal-proxy* and *slave-server*.

Note that the implementation attached to the thesis contains only components that come entirely from us and do not contain confidential information. The overview of the attached implementation can be found in Appendix A.1.

6.8 Quality Assurance

The quality of the solution was being ensured by two ways: different kinds of testing and regular meetings with the supervisors of this thesis, *Josef Bouška*, who is a senior software engineer and a team leader in *Seznam.cz*, and *Pavel Parízek*, an associate professor at Charles University.

Technical aspects of the solution were consulted with *Josef Bouška*, who was reviewing whether the implementation properly fits into the *Sklik* ad system. Formal and research aspects of our work, on the other hand, were being reviewed by *Pavel Parízek*. With this setup, all major parts of our work were rechecked by more experienced engineers and/or researchers.

As for testing, the individual components in our solutions have been tested by automatic unit and single-component tests. The test coverage was not defined but, plainly said, we tested all non-trivial mechanisms of our implementation. Getters, setters and similar kind of code were not tested as this would consume a significant amount of time with almost no effect on the quality.

Tests for components *slave-server* and *lib-keyword-search*, which are listed in the previous section, do not come from us but from the original system implementation. Their scope is really non-trivial, and together with the fact they were written by somebody else, who had a different view on the implementation, further increases our trust in correctness of our work.

On top of previous, we also implemented integration tests checking that the data balancer and the slave server communicate and cooperate in the expected way.

And last but not least, our new implementation was manually reviewed that it returns the same results as the original implementation in the conducted comparison experiments, which are described in Chapters 7 and 8.

6.9 User Documentation

We decided to separate the user documentation of the solution from the thesis and to place it together with the implementation of the components. We wanted to keep the thesis in a reasonable scope and not to separate the user documentation from the implementation, which could lead to information mismatch over time.

The user documentation contains overview information and tutorials covering different expected use-cases of the components described in Section 6.7.

7. Keyword Search Experiment

As we stated in Section 5.1, our original thesis concept was different from what the thesis looks like now. We wanted to have a fast remote database that would be queried by the slave servers in the *FindMatchingKeywords* function of Algorithm 1 from Section 2.2. As a start, we tried to optimize this keyword search.

This chapter presents an experiment that was being conducted during our optimization efforts. It shows evolution of our keyword search implementations, together with their different performance measurements.

We begin with Section 7.1 describing individual implementations in detail. Then we continue with a presentation of the experiment: the measurement setups are described in Section 7.2 and the results are analyzed in Section 7.3. And we close the chapter with a conclusion in Section 7.4 where we summarize the results together with our gained experience.

7.1 Compared Implementations

Main reasoning and information related to choices that we made, and the final resulting implementation, have been already presented in Section 6.6. As a brief summary, we selected to use the *RocksDB* key-value store to be queried for data from the *FindMatchingKeywords* function of Algorithm 1.

In this section, we want to present different implementations of this function, i.e. different implementations of the *lib-keyword-search* library (see Section 6.7), that we created during its continuous optimization.

At the beginning, we saw potential in the aggregate-oriented access to data. When there is no need to compose the data to get what the algorithm works with but the data are prepared in that form, there should be a performance boost. And because the original access of Algorithm 1 to keyword data was not purely aggregate-oriented (it accessed several barrels in order to get a full keyword record), we decided to merge all these barrels into a single *RocksDB* key-value store, serializing the values via *Protocol Buffers*. This resulted in the first implementation for our experiment, called as ***RDB-Pr*** (*RocksDB Proto*).

When performance of this implementation was measured, we discovered that we got much worse latency of the keyword search than in case of its original implementation. We tried optimizing the *RocksDB's* configuration, focusing primarily on prefix-based requests, which are heavily used in Algorithm 1 to get all keywords with a given first word. We were driven primarily by the official documentation for *RocksDB* [24]. One of the most important steps, which is not so emphasized in the documentation, proved to be setting the *cache-index-and-filter-blocks* flag to false. Our work resulted in the second implementation being compared in the experiment, called ***RDB-Pr-Opt*** (*RocksDB Proto Optimized*). We experienced what makes database expert so well-paid: the database configuration plays a significant role in the application's performance.

Still, we did not get close to latencies of the original implementation. At this point, we realized that the aggregate-oriented data access is not ideal when a large part of the aggregate does not often get used because the whole aggregate proved not to fit the request. Therefore, we split our single *RocksDB* into two, stepping back to the original design of multiple barrels. The data layout inside these two *RocksDB* databases, named *Kw-Matching-Data RocksDB* and *Keyword RocksDB*, was already described in Section 6.6.2. Both used *Protocol Buffers* as the value-serialization technology. This implementation is referenced in the text below as ***RDB-Pr-Pr*** (*RocksDB Proto Proto*). At first, this did not bring us fruits but, on the contrary, in some cases, it even made the latency worse. Details are left for experiment analysis in Section 7.3.

Next, we did a couple of attempts, which are described in the few next paragraphs that provided us basically with the same performance.

Optimizing the previous implementation, we created the fourth version presented in this experiment: ***RDB-Pr-Pr-Opt*** (*RocksDB Proto Proto Optimized*). The optimization primarily consisted of (1) preparing repeatedly computed information only once in advance, (2) reusing variables and data structures instead of creating new ones or (3) pulling variables out of loops to class fields.

Further optimization was utilization of *Flat Buffers* instead of *Protocol Buffers* inside *Kw-Matching-Data RocksDB*, resulting in the implementation named as ***RDB-Pr-Fl*** (*RocksDB Proto Flat*).

Next, we tried to use custom serialization of values for *Kw-Matching-Data RocksDB*. It was based upon copying memory chunks taken by the value data structures, without any transformation, into byte arrays that are stored by the *RocksDB*. We call this implementation as ***RDB-Pr-NS*** (*RocksDB Proto No-Serialization-technology*).

At this point, we did another optimization of the code. In the most utilized part of code, we reused as many variables and data structures as we were able to and we tried to avoid maximum of variable copying by defining the variables as class fields and by passing maximum of parameters by reference. At the same time, we set the *cache-index-and-filter-blocks* flag to false also at *Keyword RocksDB*. This resulted in a tremendous performance boost. This implementation is named as ***RDB-Pr-NS-Opt*** (*RocksDB Proto No-Serialization-technology Optimized*).

The last implementation ***RDB-Pr-Fl-Opt*** (*RocksDB Proto Flat Optimized*) replaced our custom serialization back with *Flat Buffers*. This proved to be the solution with the best performance.

All our presented modifications of the *FindMatchingKeywords* function of Algorithm 1 were compared to its original implementation, which is later referenced as ***B*** (*Barrels*).

Detailed reasoning behind the performance behavior described above and details about the results of our experiment are presented in Section 7.3.

7.2 Measurement Setups

To get a clear idea about performance of our implementations, we used multiple measurement setups differing primarily in three aspects: 1) data size, 2) computing resources and 3) queries, i.e. the input of Algorithm 1 from Section 2.2. We wanted to know how a particular implementation behaves for randomly selected queries and for a repetitively used single query, what is the impact of available computational resources and how the response time scales with data size.

The following list is an overview of the individual setups. Note that the naming may seem cryptic at the first sight but it gets obvious after reading of the first setup. Below the list, one can find additional information that is common to all the setups.

- ***SQ1-1S-HR***:
  - *SQ1* stands for measurement iterations with still the same single query designated as *1* (the query is not published as it would reveal some content of the company's production data, which is confidential). This query matches two keywords from the data set of this setup, which is specified right below.
  - *1S* stands for data size of barrels of one selected slave server that is deployed in the production. In raw information, there are approximately 10 million keywords taking up 2.5 GB on disk.
  - *HR* stands for high computational resources: the virtual machine where the measurement was executed had 16 CPUs, 32 GB of RAM and 200 GB of SSD disk.
- ***SQ1-1S-LR***: Identical to *SQ1-1S-HR* but with low computational resources: 8 CPUs, 4 GB of RAM, 50 GB of SSD disk
- ***SQ1-5S-LR***:
  - *SQ1*: repetitive iterations with the query *1*, matching 6 keywords from the setup's data set
  - *5S*: 5 selected slave servers – i.e. approx. 30 million keywords, 6.7 GB on disk

  - *LR*: 8 CPUs, 4 GB of RAM, 50 GB of SSD disk

- ***SQ1-8S-HR***:

  - *SQ1*: repetitive iterations with the query *1*, matching 14 keywords from the setup's data set
  - *8S*: data from 8 selected slave servers – i.e. approx. 42 million keywords, 9.2 GB on disk
  - *HR*: 16 CPUs, 32 GB of RAM and 200 GB of SSD disk.

- ***SQ2-5S-LR***: Identical to *SQ1-5S-LR* but a different query *2* was used, matching 69 keywords in the setup's data set.

- ***SQ2-8S-HR***: Identical to *SQ1-8S-HR* but the query *2* was used, matching 136 keywords in the setup's data set.

- ***RQ1600-1S-HR***:

  - *RQ1600*: in each measurement iteration, a query is randomly selected from 1600 random queries from the production log
  - *1S*: data from one selected slave server – i.e. approx. 10 million keywords, 2.5 GB on disk
  - *HR*: 16 CPUs, 32 GB of RAM, 200 GB of SSD disk

- ***RQ1600-1S-LR***: Identical to *RQ1600-1S-HR* but with 8 CPUs, 4 GB of RAM and 50 GB of SSD disk

- ***RQ1600-8S-HR***: Identical to *RQ1600-1S-HR* but with data from 8 selected slave servers – i.e. approx. 42 million keywords, 9.2 GB on disk

- ***RQ1600-8S-LR***: Identical to *RQ1600-1S-LR* but with data from 8 selected slave servers – i.e. approx. 42 million keywords, 9.2 GB on disk

All virtual machines used in the experiment were running the *Debian 8* Linux distribution [35] and reported no significant load in a system's process overview (shown by the utility *top* [36]). The virtual machines were deployed in the company's cloud environment, no information about distribution of the virtual machines to physical servers is known to us.

All code was compiled using the *GCC* [37] compiler, version 4.9, with the optimization *O3* turned on.

The *RocksDB* databases were primarily opened for read-only access. We also conducted measurements with opening them for read-write access with no change in performance behavior.

All time measurements were done using the *steady-clock* timer from the *chrono* namespace of the *C++* standard library.

All measurements presented in this chapter were repeated 20 thousand times and all boxplots in Section 7.3 are always created based on the latter half of the measured values so that the presented results are not influenced by any kind of heating up of the system (e.g. stabilization of the CPU cache content).

7.3 Results and Discussion

This section presents our findings from the experiment in a form of analysis based upon the raw measured data.

We primarily focused on response times of the implementations listed in Section 7.1. This is the topic discussed in the first subsection below. The second subsection then discusses the aspect of the data space used by the individual implementations.

7.3.1 Response Time Comparison

The most important result of this experiment is that we managed to develop an implementation of the *lib-keyword-search* library that is, in terms of performance, more than comparable to its original implementation. This is important for the main work in our thesis project – we have a decent replacement for the original barrels that can be incrementally modified.

To support our claim, Figures 7.1 and 7.2 present boxplots of response times of our individual implementations for the measurement setups *SQ1-8S-HR* and *SQ1-1S-HR*, respectively. As you can see, the clearly best response times were measured for implementations *RDB-Pr-NS-Opt* and *RDB-Pr-Fl-Opt*. Furthermore, as visible in Figure 7.2, these implementations are comparable to the original implementation *B*. Figure 7.4, which provides a closer look inside Figure 7.2, shows that our implementations outperform the original one by approximately a factor of two.

As you probably noticed, the original implementation *B* is not present in Figure 7.1. Nor it is included in any measurements with setups using data from more than one slave server. The reason is that we had access to the data in the form of barrels, which were already split for individual slave servers, and that the benchmarked original implementation of the *lib-keyword-search* library cannot work with multiple barrels at once. As the barrels are constructed from scratch and there is no support for their merging, we decided, with respect to the scope of this thesis project, not to implement this non-trivial functionality ourselves. Our new implementation of the *lib-keyword-search* library works with a single *RocksDB* as well but we could have filled the database with data from multiple barrels.

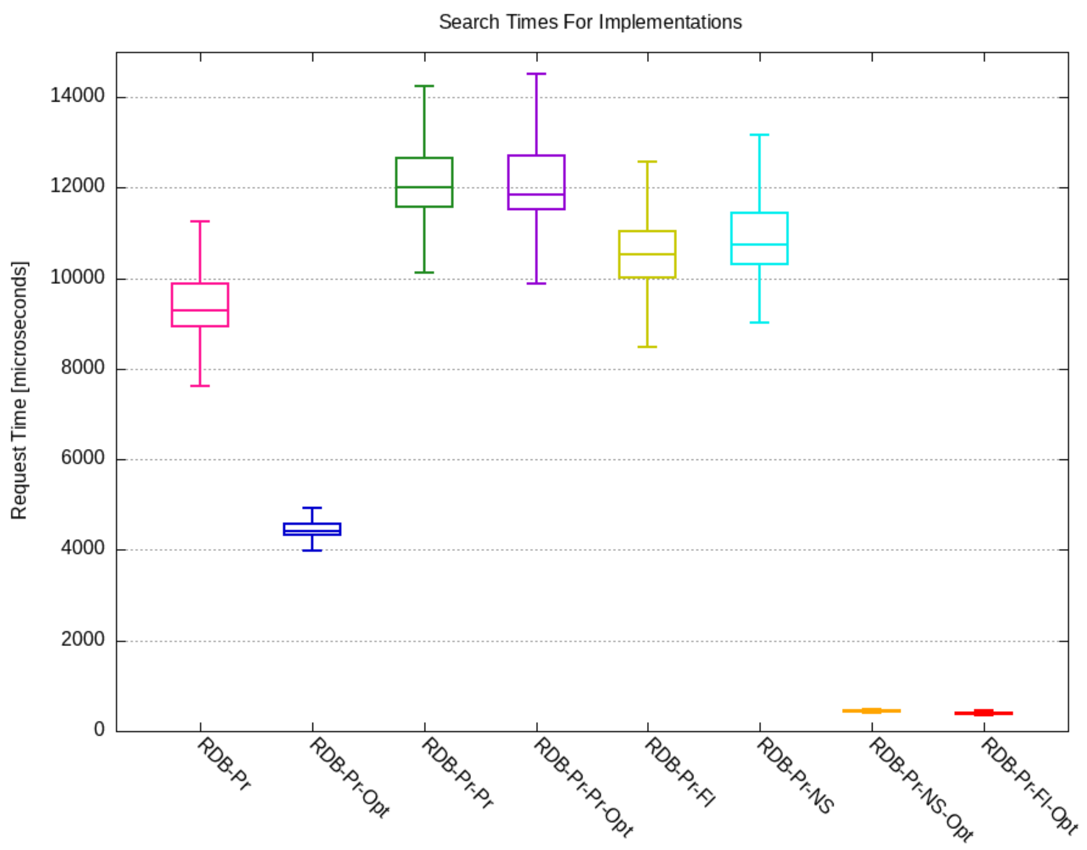


Figure 7.1: SQ1-8S-HR - Response times

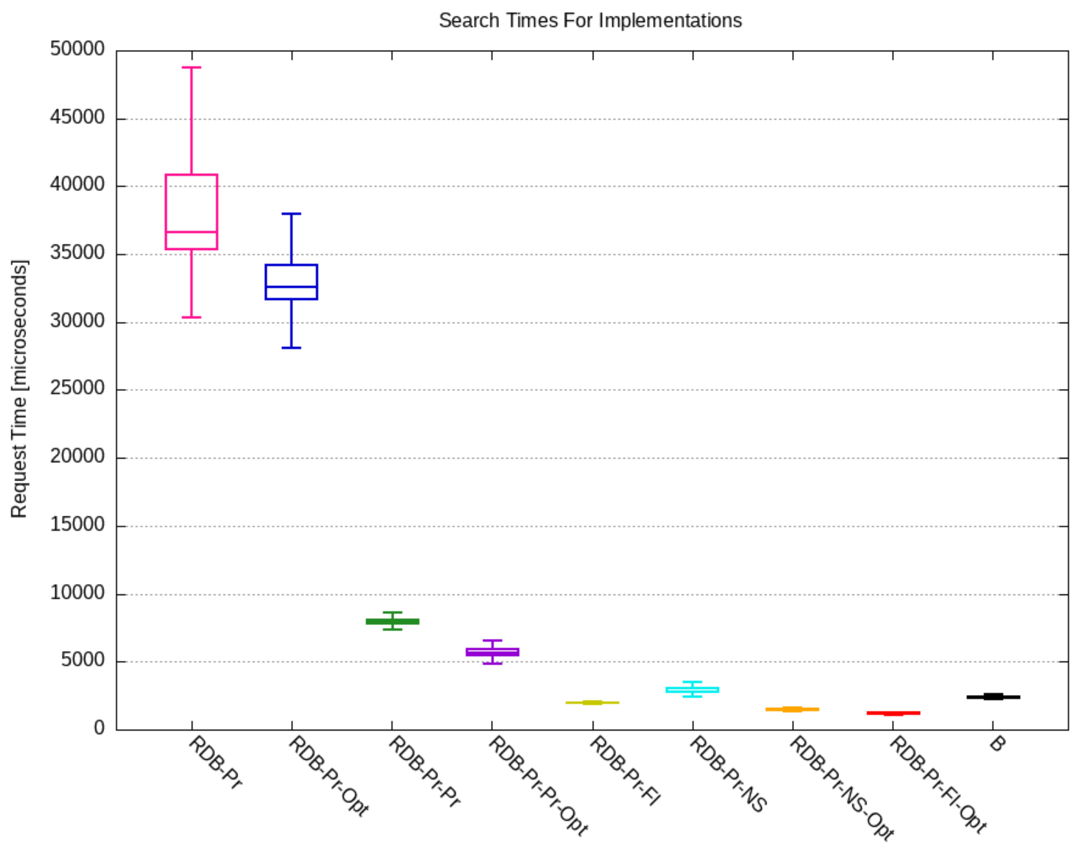


Figure 7.2: SQ1-1S-HR - Response times

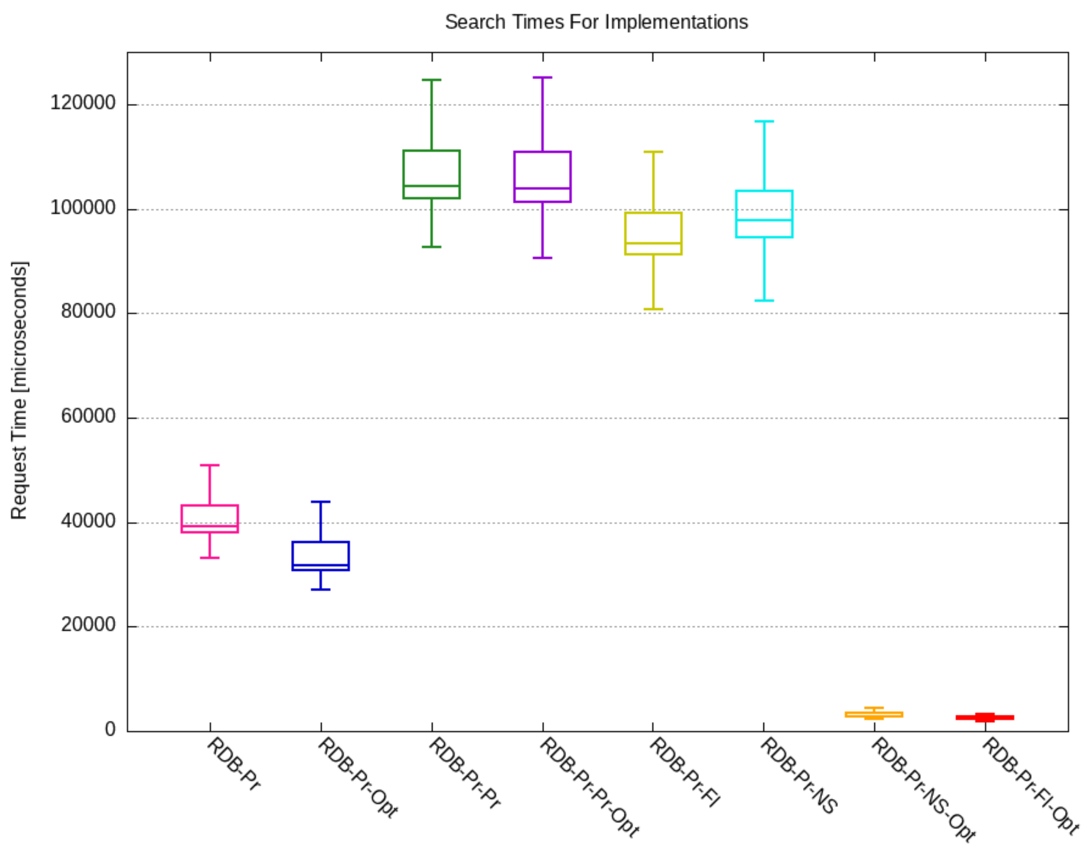


Figure 7.3: SQ2-8S-HR - Response times

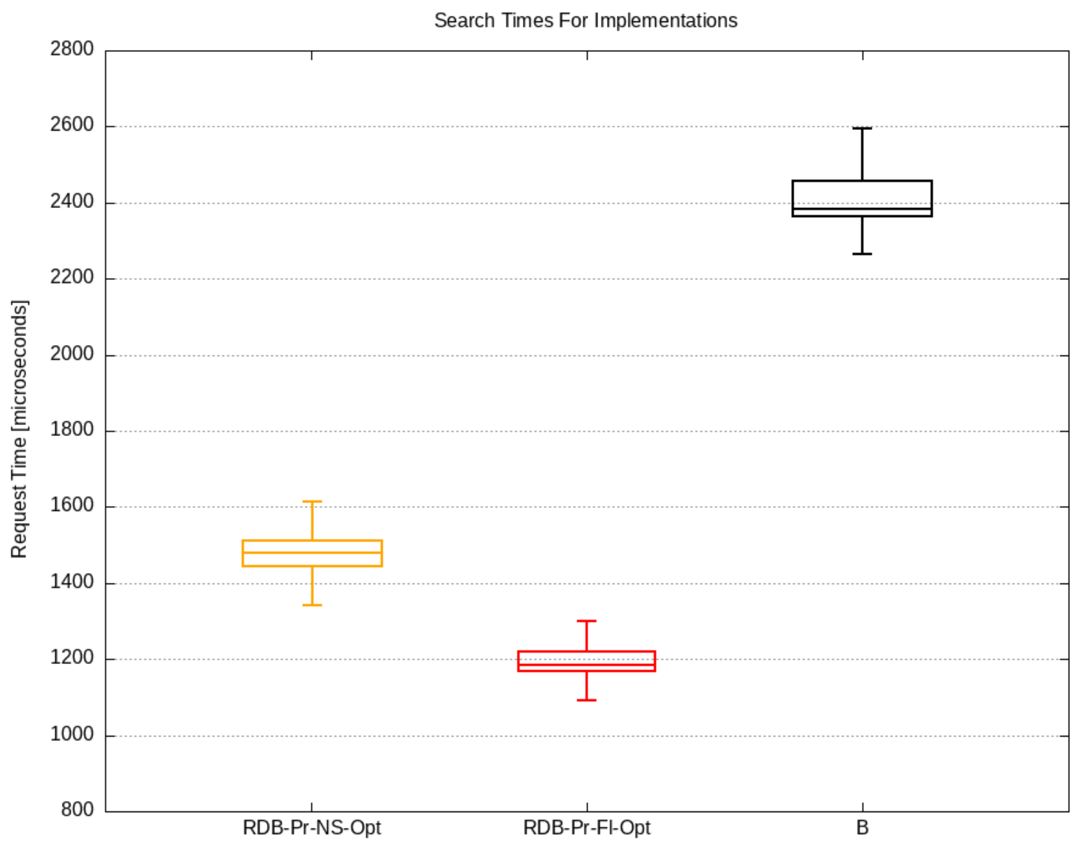


Figure 7.4: SQ1-1S-HR - Response times - focused

We also made other observations that are not so important for this thesis but are no less interesting:

- First of all, Figure 7.2 shows that overreads, i.e. reading more data than actually needed, may have a great impact on performance. While the implementation *RDB-Pr-Opt* uses a single *RocksDB* database for all the data, *RDB-Pr-Pr* utilizes two *RocksDB* databases that contain precisely those data that are needed in 1) deciding whether a keyword matches a query, and 2) completing the really matching keywords with the rest of the data.

  You may wonder why the implementations that make use of two *RocksDB* databases provide much worse results in Figure 7.1. We were also surprised by this observation but we created a hypothesis for this, which is supported by the presented data. We believe that it is caused by bad optimization of point look-ups in *Keyword RocksDB* (see Section 7.1). When a keyword proves to match a query, the rest of the keyword's data is retrieved from this *RocksDB* via a point look-up. If this is not optimized, the bad effects do not arise when there are only a few matching keywords. This is the case in Figure 7.2 as the measurement setup *SQ1-1S-HR* resulted in only 6 keyword matches. On the other hand, the measurement setup *SQ1-8S-HR* in Figure 7.1 had 14 matches and the setup *SQ2-8S-HR* in Figure 7.3 had 136 matches. The implementations *RDB-Pr-NS-Opt* and *RDB-Pr-Fl-Opt* were already optimized with point look-ups and do not follow this pattern anymore.

- Second, our experiment offers a comparison of the *Protocol Buffers* and the *Flat Buffers* serialization technologies. As can be seen in the earlier referenced figures, *RDB-Pr-Pr* provides significantly worse response times than *RDB-Pr-Fl*. These two implementations differ only in the used serialization technology and in the optimizations made in the implementation *RDB-Pr-Pr-Opt* that was developed in-between – these, however, have only a little impact as visible in the graphs. At the same time, from comparison of implementations *RDB-Pr-NS-Opt* and *RDB-Pr-Fl-Opt*, we can guess that the *Flat Buffers* implementation does something very close to direct byte array copying between memory and the serialization buffer.

- Third, as can be seen in differences between results for the pair of *RDB-Pr* and *RDB-Pr-Opt* and between results for the pair of *RDB-Pr-NS* and *RDB-Pr-NS-Opt*, configuration of the (*RocksDB*) database plays a great role in performance of a whole application.

- Last, there is another reason why *RDB-Pr-Fl-Opt* outperformed *B*. The structure of barrels in the original implementation is not too friendly to

CPU cache as the data access to one of the barrels does not behave according to the access-locality principle assumption, which is crucial for the best functioning of CPU caches. Details are rather confidential. Our implementation improved this sub-optimal data access, which surely had an impact on CPU cache miss frequency. However, we did no further experiments to support this hypothesis, due to already quite extensive scope of this thesis project. Note that this improved cache-friendliness is present in all our implementations.

Let us add that some of the measurement setups listed in Section 7.2 were designed to use bigger data than the memory size. Since the so-far presented measurements did not randomize the queries but used only a single query, the results from these measurements did not force the *RocksDB* database not to cache all requested data into memory, leaving the fact that the whole data cannot fit into memory irrelevant. Furthermore, note that using a single query probably took advantage of the improved cache friendliness of our implementations, which may have amplified the performance improvement of our implementations with respect to the original one.

This leads us to measurements based on setups with randomized queries, which we performed in order to get a detailed idea how the implementations would behave in a workload that is closer to the real use-case. In these experiments, we compared only our most successful implementation *RDB-Pr-Fl-Opt* and the original implementation *B*.

Note that this measurement was not conducted in an ideal way as: 1) each of the two implementations was benchmarked with a different random sequence of queries from the same super-set of 1600 production queries, 2) some of the queries did not match any keywords, in which case the requests returns quite quickly. This happened in no more than 20% of measurement iterations, by a rough guess from the measurement log, and it is the reason why the boxplots lean to zero. We are aware that the measurement could be improved but our goal was to get close to real use-case without any complex benchmarks – and we think we succeeded in this task.

The results are illustrated by Figure 7.5. We believe that we can confirm that the *RDB-Pr-Fl-Opt* implementation is, in terms of performance, at least comparable to the original barrels implementation.

As for the measurement setup *RQ1600-S8-LR* where all the data cannot fit into memory at once, the measured response time is only slightly worse than in case of the setup *RQ1600-S8-HR*. The difference is presented in Figures 7.6 and 7.7. However, the set of 1600 requests is randomly selected and we did not further investigate whether they truly cause collisions in the cached data of the *RocksDB* databases (with respect to the thesis project scope).

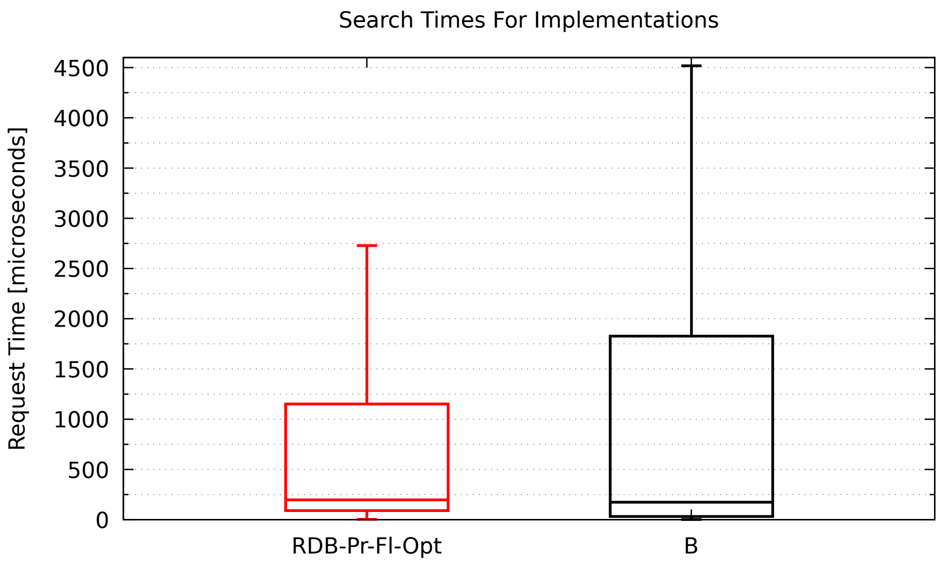


Figure 7.5: RQ1600-1S-HR - Response times

Let us note a couple of final observations. First, the presented measurements in this chapter used *RocksDB* databases that were opened for read-only access as no write access is performed by the *lib-keyword-search* library. We conducted also measurements where we opened the *RocksDB* databases for read-write access (with no writes actually performed as no are necessary in the implementation). The results seemed unchanged. Second, we also studied the evolution of response times in subsequent measurement iterations. We found nothing especially interesting: the response time stabilized quite fast and we observed only a single type of irregularity, which is best visible in jumps of the cyan-colored values of *RDB-Pr-NS* illustrated in Figure 7.8. We think that it is caused by data reorganization tasks running on background inside *RocksDB*. However, we did not analyze it further as it is out of scope of this thesis project.

Finally, note that, in order to keep the thesis scope at a reasonable level, we presented only a fraction of the data collected from all the measurement setups, which have been described in Section 7.2. We covered all the interesting observations. If you are interested also in the other data from this experiment, see Appendix A.2. Note that you may bump into a strange change of response times in setups that use data for one slave server and setups using bigger amount of data. We believe that this is caused by a change of the data set used in these two kind of setups – we needed to re-download the data so the barrels and *RocksDB* databases content may be different.

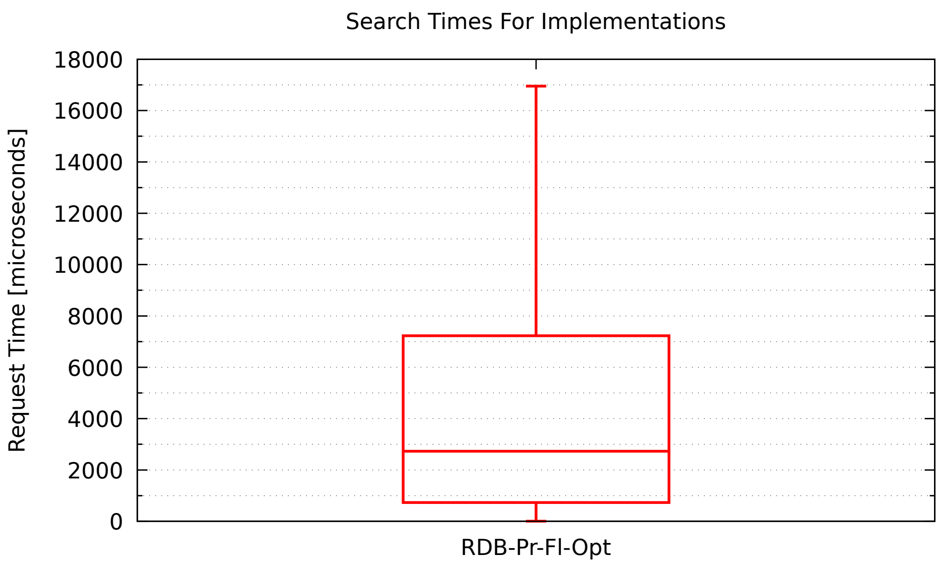


Figure 7.6: RQ1600-8S-HR - Response time

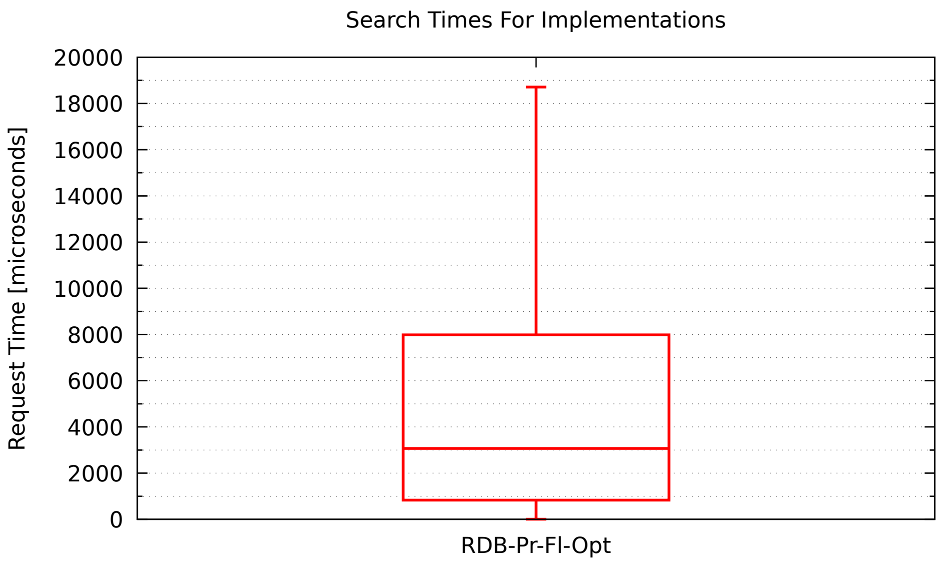


Figure 7.7: RQ1600-8S-LR - Response time

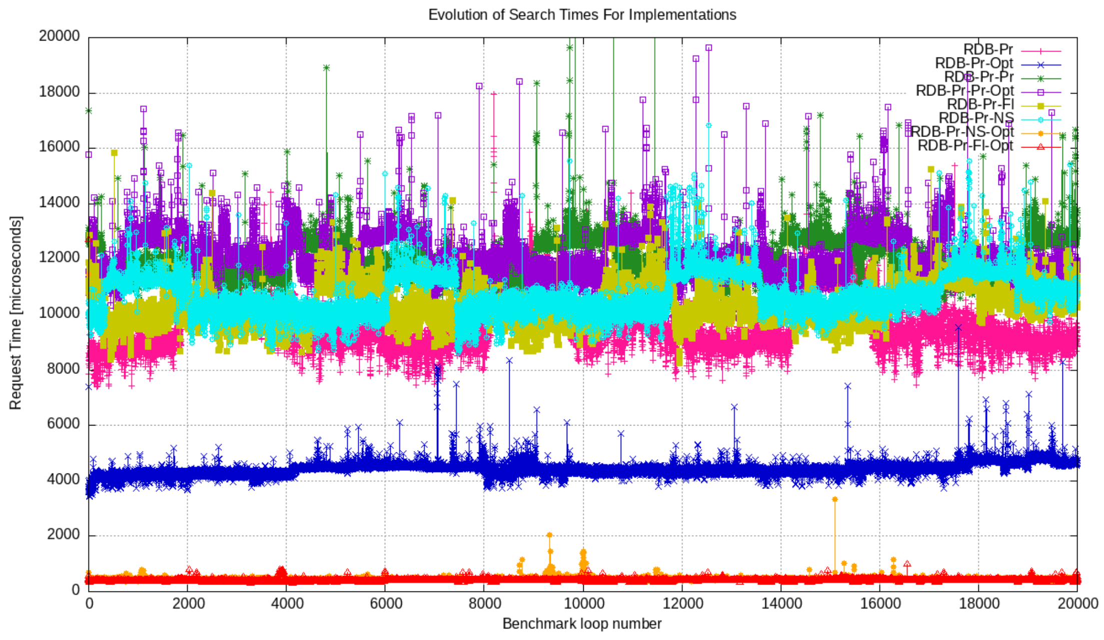


Figure 7.8: SQ1-8S-HR - Evolution of response time

7.3.2 Data Space Comparison

Our secondary interest in this experiment was in the space taken by the data used by our implementations compared to the original one, i.e. how much more or how much less data space is taken by the *RocksDB* databases in comparison with the original *barrels*. Results are illustrated in Figure 7.9. Red columns correspond to data size used in the measurement setup *SQ1-8S-HR*, i.e. size of data of eight slave servers. Blue columns show data size for the measurement setup *SQ1-1S-HR*, i.e. size of data of one slave server.

We discovered that keeping the keyword aggregates in a single *RocksDB* database saves data space. From the perspective of the whole system, it might free a significant amount of data (tens of gigabytes), in exchange for worse response times of the *lib-keyword-search* library. But since today offers cheap data storage and asks for fast processing, we believe that the data savings are not worth the higher response time.

Note that Figure 7.9 captures only data space taken by barrels that are related to this experiment. The slave servers use also other barrels, which are not captured here as they are irrelevant for this experiment.

7.4 Conclusion

In this experiment, we implemented and compared several possible replacements of the *lib-keyword-search* library, which has been introduced in Section 6.7. Our goal was to replace the original barrels with *RocksDB* databases that would allow

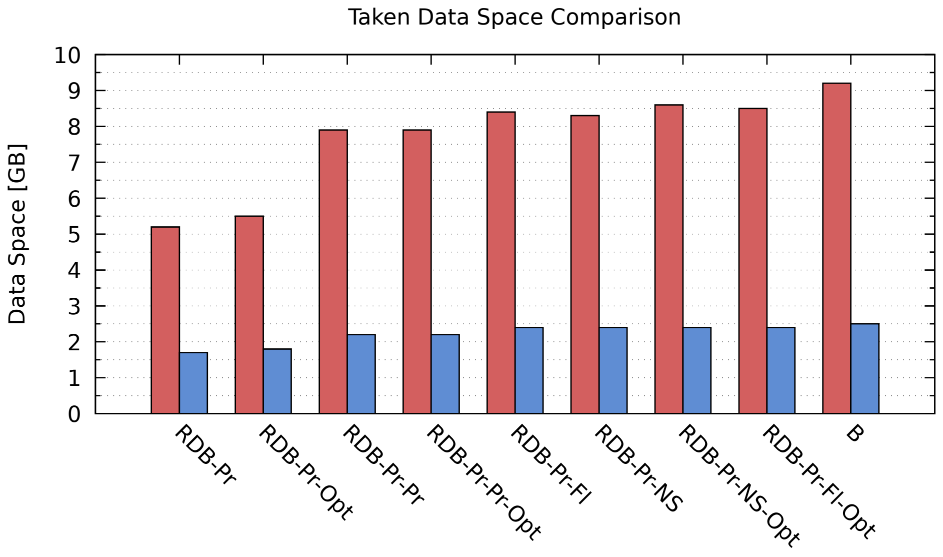


Figure 7.9: Comparison of data space taken by *RocksDB* databases and the original barrels

incremental modifications of the stored data [1]. At the same time, we were after a replacement that would be comparable to the original implementation with respect to performance.

As discussed in the previous sections, we succeeded with both of these goals. The best of our implementations, which has been referenced in the previous text as *RDB-Pr-Fl-Opt*, is comparable to the original implementation in all conducted measurements and, in some of them, it even significantly outperforms the original implementation. Functional requirements are satisfied as the incremental modification is one of the key features of the *RocksDB* key-value store.

RDB-Pr-Fl-Opt is used in our overall implementation of the data balancer. However, some little modifications were made after this experiment so the measurements of the current form of *RDB-Pr-Fl-Opt* might be a little different, even though we would not expect significant changes. Due to the limited scope of this thesis project, we did not benchmark this really final variant of *RDB-Pr-Fl-Opt*. As explained in Chapter 5 and in the introduction of this chapter, our thesis project concept changed after this experiment and further work in this area became out of scope of this thesis.

During the experiment, we experienced some interesting, more-or-less known facts related to database application optimization. For example, we learned (1) that a purely aggregate-oriented access to data may not be ideal and (2) that database configuration plays a really significant role with respect to the application performance.

[1]Note that, as discussed in section 5.1, our original goal was different – that influenced the final form of the experiment.

8. Balancing Experiment

This chapter presents the main experiment of this thesis project: a comparison of balancing techniques, which have been selected in Section 5.4, in the context of the *Sklik* ad system.

The idea of the experiment was to individually measure performance of the balancing techniques utilizing different kinds of workloads, collect metrics from these measurements and compare the results.

The chapter's sections are organized as follows:

- Section 8.1 lists the compared balancing techniques.
- Section 8.2 describes our selected workloads.
- The next Section 8.3 presents what metrics we were collecting.
- Section 8.4 is dedicated to the data scale chosen for the experiment and our reasoning behind it.
- In Section 8.5, we talk about the platform and the network environment used in the experiment.
- Our detailed findings and their analysis are subject of Section 8.6.
- Section 8.7 evaluates the experiment's quality and lists related future work.
- Finally, the overall conclusion is subject of Section 8.8.

8.1 Compared Balancing Techniques

As already mentioned earlier in Section 5.4, we decided to compare the following balancing techniques. Each is introduced by a code name used for references in the rest of this chapter.

1. ***Original***: The original system's load balancing implementation, which has been described in Section 2.4. Note that we did not change the implementation in any way, we only added the metrics collection code.
2. ***Random***: A random balancing algorithm implemented into our data balancer. Its description may be found in Section 6.5.
3. ***Weighted-Move***: A balancing algorithm inspired by the *Weighted-Move* algorithm of *Google Slicer* [12]. Detailed information about the algorithm presented in the paper [12] is located in Section 4.5, while further information about our implemented version is presented in Section 6.5. The balancing algorithm was, as in the previous case, implemented into our data balancer.

Note that the *Original* balancing technique differs quite significantly from its competitors since the load balancing implementation was completely changed and searching of matching keywords is slightly different as well (as was described in Section 6.6). On the other hand, the other two implementations differ really only in the balancing algorithm – they are both implemented inside our data balancer that offers easy modification of the balancing algorithm by providing a custom implementing of a single abstract class.

Finally, we would like to explicitly state the characteristics of the listed balancing techniques as it is important for this experiment:

1. *Original* does not change the data distribution in any way during a measurement. It is true that, in the production system, a rebalancing is done during the night, but the system is primarily utilized during the day. Therefore, we did not do any rebalancing in our experiment either – to simulate the day-time operation.
2. *Random* does no data redistribution during a measurement as well.
3. *Weighted-Move* is dynamic and reacts to the system's load by iterative rebalancing of the data.

8.2 Workload

A request to the system is basically formed by a single query for which relevant ads should be returned. We decided to represent this workload by a text file where each line represents a query sent to the system. The queries from the file were sent to the system in their exact order and repetition.

An alternative would be generation of queries using some parameterizable random distribution. We believe that the file-base approach is better as it is simple and provides very precise definition of the workload. A drawback might be a little laborious creation of the file but since it is only for purposes of the experiment, it is really not a big issue.

We designed the following workloads for the experiment. Performance of each balancing technique was measured under each of these workloads. Note that we only describe the characteristics of the workloads because the exact queries are confidential.

1. ***Single Query***: A single query repeated 100 times in a sequence. We selected a random representative of a common query, with respect to the number of keyword matches in the experiment data set, which is described in Section 8.4.
2. ***Production-like***: A sequence of 100 queries randomly selected from the production system's log so that it simulates a real-world workload.

8.3 Collected Metrics

In our measurements, we were tracking the following metrics:

1. ***Master latency***: Latency of the whole request in the master server (i.e. the duration since the master server receives the request until it issues the response).
2. ***Slave latency***: Latency of the request from the master server in a given slave server (i.e. the duration since the slave server receives the request until it issues the response). As we explained in Section 2.2, slave server latency directly depends on the number of matched keywords to a given query in the slave server. Therefore imbalanced latencies over individual slave servers for a given query directly means imbalance of data, which are needed to process that query, over the slave servers. In other words, the more balanced slave server latencies, the better is the data distribution for the particular workload.

We were primarily interested in the overall trend in the evolution of measured values in time, the statistical information such as the average and the variance values were not so important for us and thus they are not presented in this text.

All latency measurements were done using the *steady-clock* timer from the *chrono* namespace of the *C++* standard library.

8.4 Scale

Because of the limited time for our thesis project, we could not afford to conduct the experiment in our originally intended scale that would be comparable to the production system of *Sklik*.

The reason is insufficient performance of the data balancer for the initial filling of the system with all beginning data. Note that this is not a big issue as this task is expected to be done very rarely during the system execution. Furthermore, the implementation is ready to be scaled to higher performance levels. But still, some work would be necessary to make it real. For instance, we would need more instances of the *Distributor* components but we have not implemented a remote communication proxies between the *Mapper* and the *Distributor* components.

Therefore, we would either need more time for the experiment initialization or more time to perform the work required to scale the data balancer up. However, with our time limitations, we decided to keep the experiment low-scaled.

The used data scale for the experiment was approximately a tenth of the production system's data. In raw numbers, we used 3.7 GB of data of three more-or-less randomly selected original system's production slave servers. There were around 250 thousand groups (i.e. data items) containing 8.5 million keywords.

As we used data from three original system's slave servers, it makes sense to use directly the original barrels for measurements of the *Original* technique. Therefore, we decided to use three slave servers throughout the whole experiment.

8.5 Platform and Network Setup

With the described restrictions of the experiment's scale, we did not need that many computational machines as we originally intended. We ran our experiment on five virtual machines inside the company's cloud environment. These are the parameters of the network and virtualization environment:

- The exact physical machine's parameters and details are confidential. We can share the CPU normal operation frequencies. The typical value was 2.2 GHz but there were a few exceptions of 1.8 GHz.
- The physical machines were geographically in the same cluster.
- The virtual machines were not migrated across physical machines throughout our measurements.
- We used a different set of virtual machines for measurements of each balancing technique. We were forced to this decision because of time constraints for the thesis project.

All the virtual machines used in the experiment were of the same following platform:

- 8 CPUs,
- 16 GB RAM,
- 100 SSD disk,
- the *Debian 9* Linux distribution [38].

The system components were deployed to the five virtual machines in the following manner:

- One machine was running the data balancer in the form illustrated in Figure 6.5, which is referenced from Section 6.4, i.e. it ran the mapper and one distributor in the same process. This machine was not used in measurements of the *Original* balancing technique because the data were already prepared on the slave servers.
- One machine was dedicated to the master server, which was introduced in Section 2.3.
- And the other three machines were utilized for the three slave servers.

All code was compiled using the *GCC* [37] compiler, version 6.3, with the optimization *O2* turned on.

8.6 Results and Discussion

Our hypothesis for this experiment was that advanced balancing techniques such as *Google Slicer's Weighted-Move* [12] would provide more balanced data distribution than the *Original* balancing implementation. Our primary argument was that while *Original* is quite simple, being developed as a draft balancing technique with expected future improvements, algorithms such as *Weighted-Move* have rich theoretical and experimental background.

Note that we present only the most important data from the experiment in this chapter. In case of interest in the complete and/or raw data, see Appendix A.3.

8.6.1 Original Technique Measurements

Measured slave server latencies in case of the *Original* balancing technique are presented in Figures 8.1 and 8.2. Figure 8.1 shows evolution of the slave server latencies in time in case of the *Single Query* workload. Figure 8.2 presents the same for the *Production-like* workload.

Note that we did not explicitly run the *Original* data balancing on the experiment's data. We used the data as they were distributed by the *Original* algorithm in the production system. We believe that running the balancing again, this time with no presence of the rest of the production system's data that were not used in our experiment, would not create a different data distribution. The reason is that the algorithm is based upon solving the *Bin Packing* problem [6] using a greedy approach. We do not provide any proof for this but it is quite intuitive that the algorithm would make the same decisions as in a greater data scope.

As you can see in Figure 8.1, data is not perfectly balanced for the query used in the *Single Query* workload. This may a bad luck for *Original*: no balancing algorithm can have data balanced optimally for all possible queries. As illustrated in Figure 8.2, in case of the *Production-like* workload, the latencies of individual slave servers are more balanced. But still, one can see that slave server 2, which is represented by the blue line in the figure, provides latencies out of the other two servers' latency mean. Thus, the data are not perfectly balanced from the perspective of the production queries either[1], which is the most important workload for the system.

Although imbalance of slave servers latencies is the most important metric in our experiment, the user of the system actually experiences the master server latency. Therefore, we present also this metric evolution in time in Figures 8.3 and 8.4, for both our designed workloads.

[1]Given that our representative production query set really represents the production workload. We have no reason not to believe this.

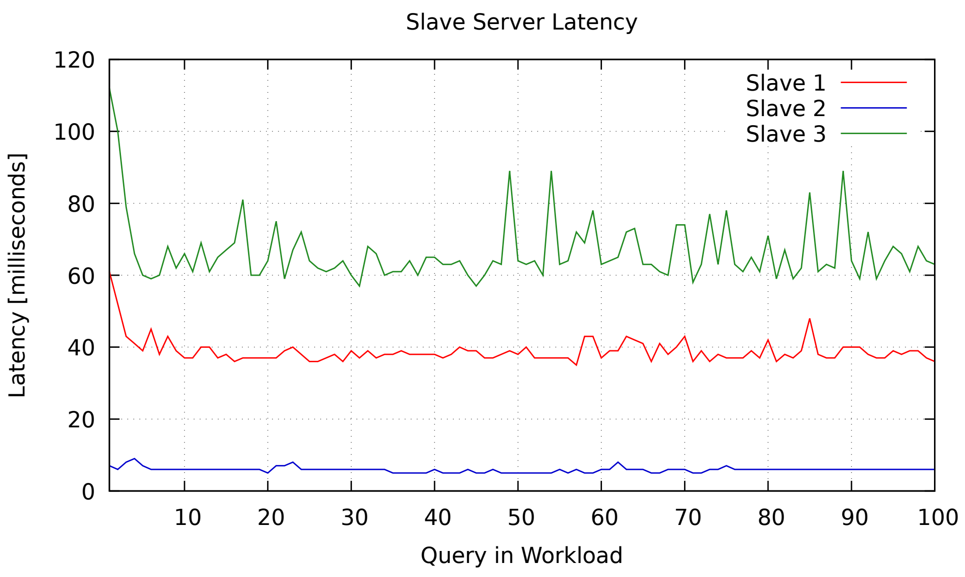


Figure 8.1: Slave server latencies in the *Single Query* workload on the *Original* balancing technique

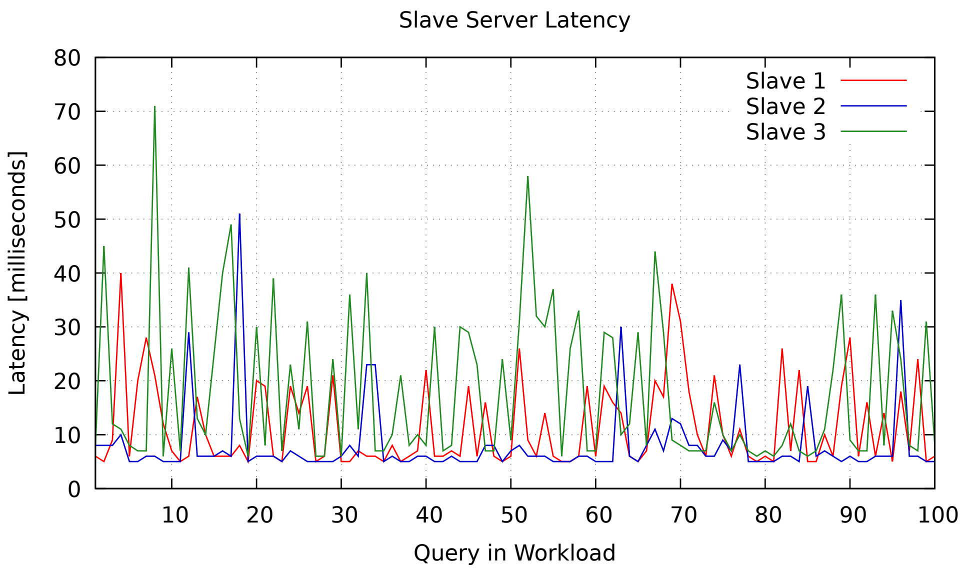


Figure 8.2: Slave server latencies in the *Production-like* workload on the *Original* balancing technique

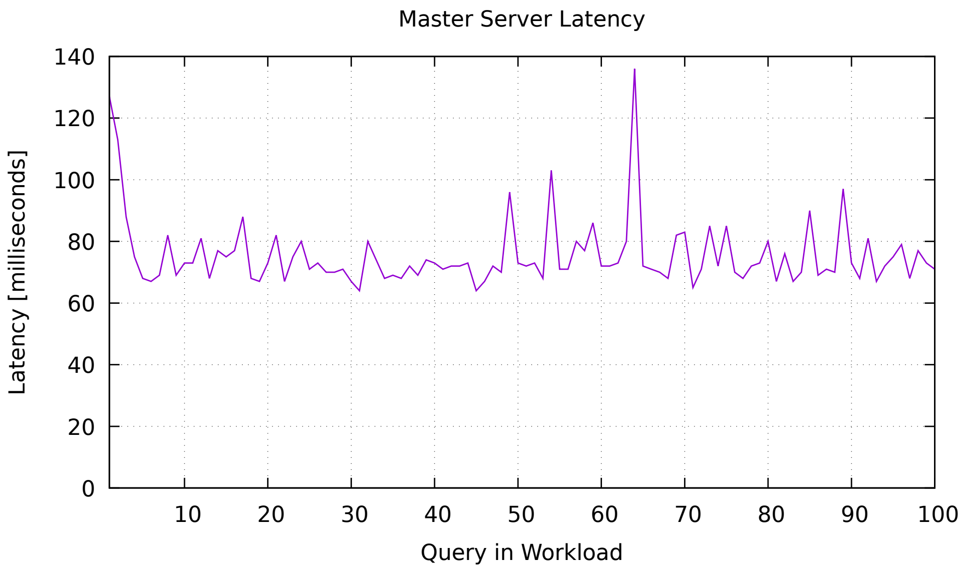


Figure 8.3: Master server latency in the *Single Query* workload on the *Original* balancing technique

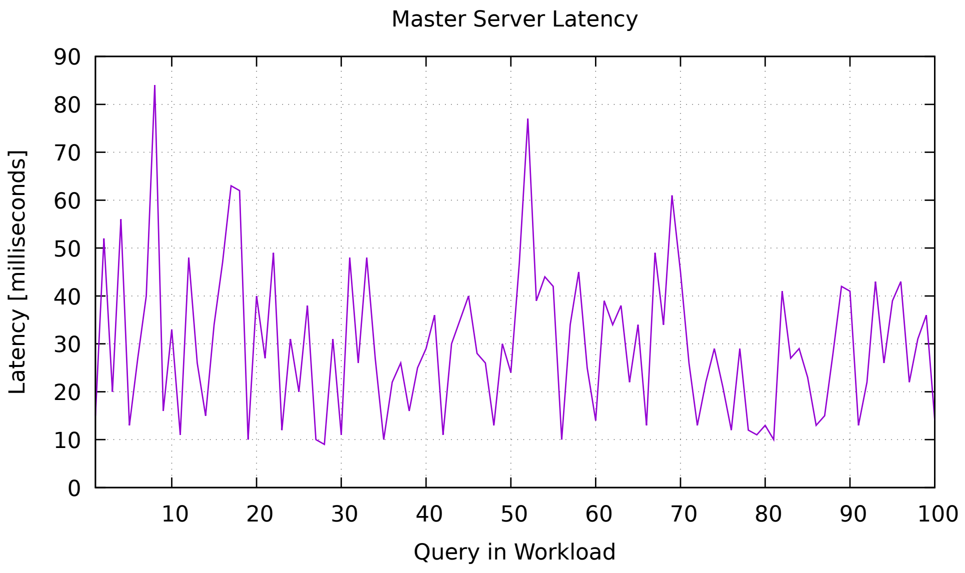


Figure 8.4: Master server latency in the *Production-like* workload on the *Original* balancing technique

8.6.2 Random Technique Measurements

The role of the *Random* balancing technique in our experiment was to be a baseline to know whether the other techniques are worth the efforts to invest time into implementing them.

The slave server latencies of *Random* under the *Single Query* workload is presented in Figure 8.5. The same for the *Production-like* workload can be found in Figure 8.6.

Unfortunately, there is great variance of the measured values so any conclusion about data balance are difficult to make. We did not have the time to investigate this issue so we cannot explain the effect.

Figure 8.5 shows that the data in slave servers are quite balanced but it is based only upon one single query. As visible from Figure 8.6, we cannot say the same thing in case of the *Production-like* workload – because of the great value variance.

On the other hand, an absolutely clear conclusion is that the performance is much worse compared to the *Original* balancing technique. It is hard to determine what caused this. As we already mentioned in Section 8.1, the implementation differs a lot. The system is quite complex and there may be several reasons for it. Unfortunately, due to the thesis project time constraints, we could not further investigate this issue.

We believe that both the performance problems, the great variance and the high latency values, could be resolved by further work.

The master server latency of the *Random* balancing technique under both our workloads is presented in Figures 8.7 and 8.8. These measured values are not that pessimistic as the slave server latencies, the variance is not that high because the master server latency is more-or-less a sum of the slave servers latencies whose variance is not synchronized. The average value seems not so terribly bad either, with respect to time limits for the ad system request processing.

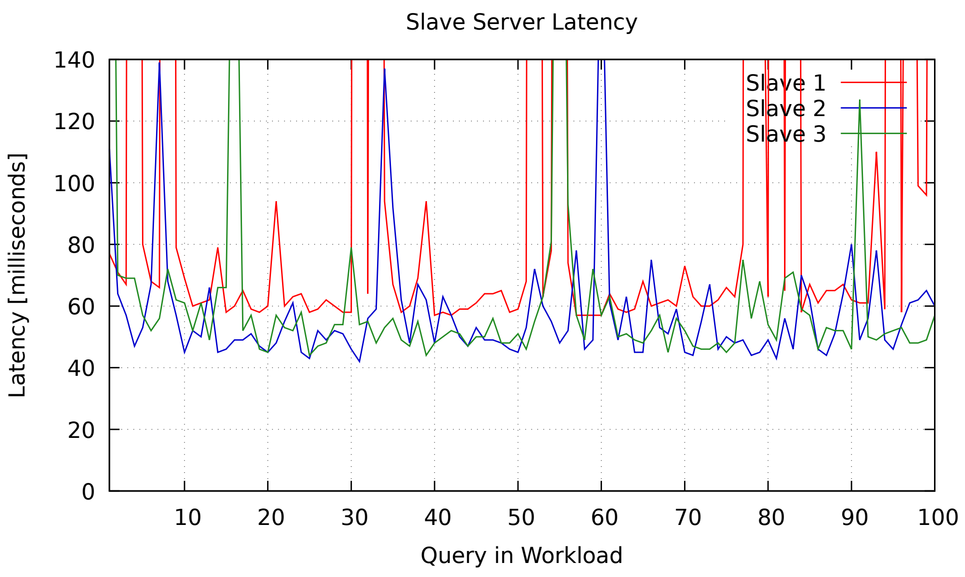


Figure 8.5: Slave server latencies in the *Single Query* workload on the *Random* balancing technique

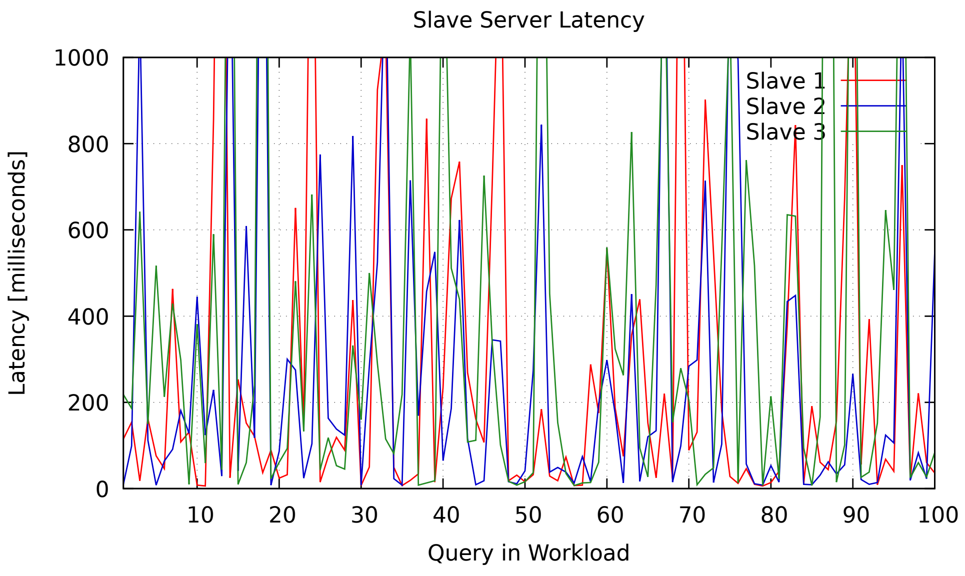


Figure 8.6: Slave server latencies in the *Production-like* workload on the *Random* balancing technique

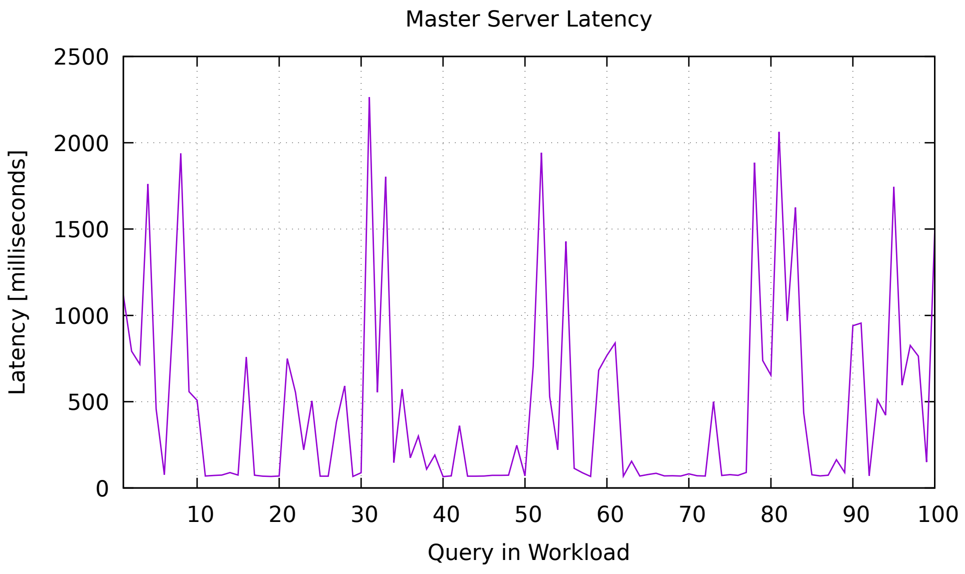


Figure 8.7: Master server latency in the *Single Query* workload on the *Random* balancing technique

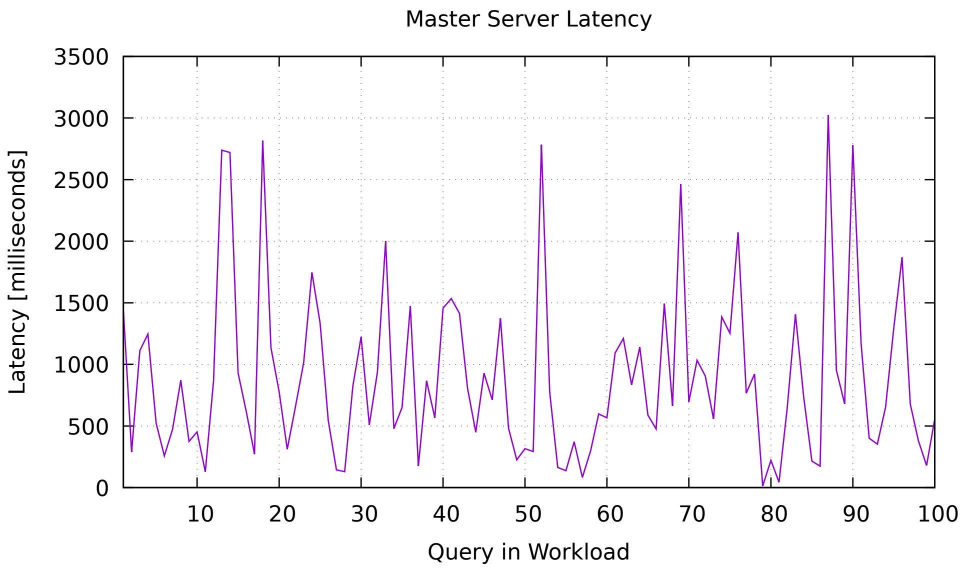


Figure 8.8: Master server latency in the *Production-like* workload on the *Random* balancing technique

8.6.3 Weighted-Move Technique Measurements

Results from measurements of the *Weighted-Move* balancing technique are, on the contrary to *Random*, quite positive.

As can be seen in Figure 8.9 for the *Single Query* workload, the slave latency is in this case higher than measured at the *Original* technique but the variance is not that great as at *Random*.

Furthermore, since the *Weighted-Move* balancing technique is load-aware, we were interested how the latency values evolve in time. We repeated the *Single Query* workload, which itself consists of 100 requests, 100 times and compared the measured results in the first and in the last of these workload iterations. In other words, we compare results from two intervals of requests: 1-100 and 9900-10000. The slave latencies in case of the first interval are presented in Figure 8.9 and the master server latency in Figure 8.11. The same for the second interval is captured in Figures 8.10 and 8.12.

We expected that individual slave server latencies would be getting closer to each other over time as the *Weighted-Move* algorithm would be rebalancing data to get a uniform load distribution. This did not happen. One can see that only the variance of the values got decreased. This may be caused also by heating up of the system such as CPU cache filling with appropriate data. However, we believe that this would have happened already during the first workload iteration so this effect should be caused really by the algorithm's data balancing.

The measurements for the *Production-like* workload were designed in a similar manner. We repeated the workload, which itself consists of 100 queries, 100 times and observed the measured latency behavior in different workload iterations.

Figure 8.13 shows slave latencies during the first workload iteration, Figure 8.14 shows them in the third iteration, Figure 8.15 in the fifth and Figure 8.16 in the last iteration number 100. We intentionally left the scale of the charts unchanged for easier comparison of the values. A closer look at the latencies in the fifth iteration is presented in Figure 8.17 and the master latency in this interval is presented in Figure 8.18. The master latency for the other intervals is not included in this thesis, to limit the text scope. In case of interest, see Appendix A.3.

As one can see, the *Weighted-Move* balancing technique decreased the data imbalance for the *Production-like* workload quite fast, within several hundreds of queries.

The slave server latencies are then comparable to those measured at the *Original* approach. The best comparison can be found in Figures 8.2 and 8.17.

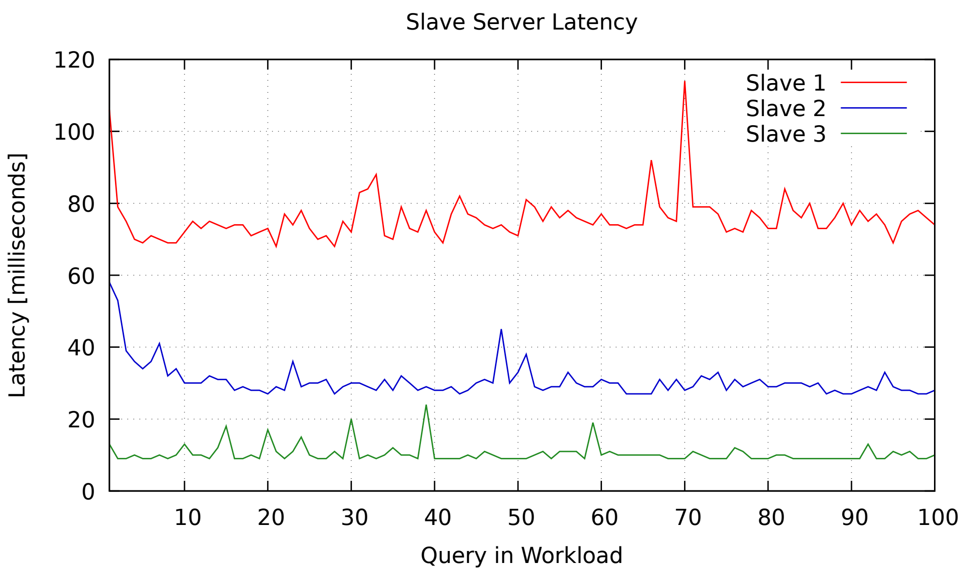


Figure 8.9: Slave server latencies in the *Single Query* workload on the *Weighted-Move* balancing technique, workload iteration 1

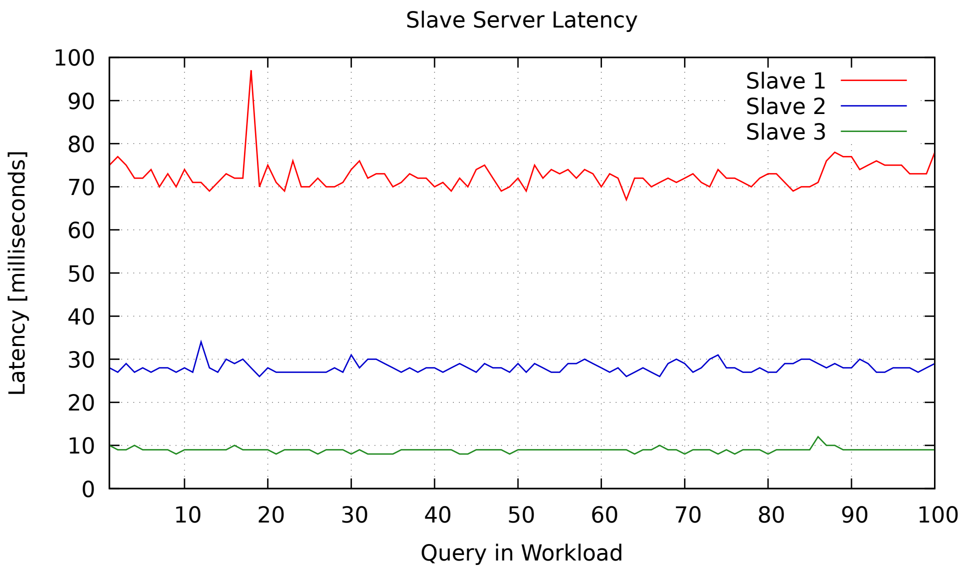


Figure 8.10: Slave server latencies in the *Single Query* workload on the *Weighted-Move* balancing technique, workload iteration 100

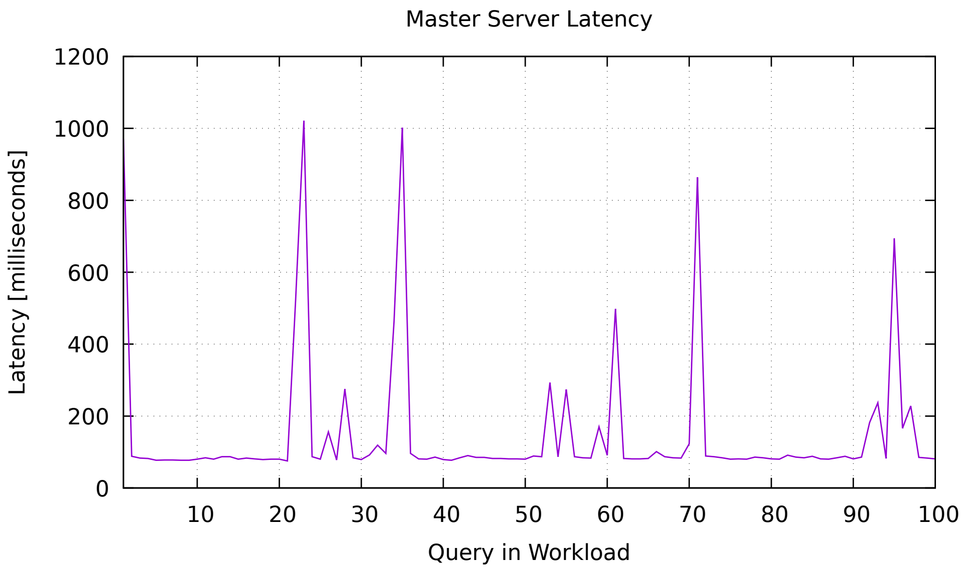


Figure 8.11: Master server latency in the *Single Query* workload on the *Weighted-Move* balancing technique, workload iteration 1

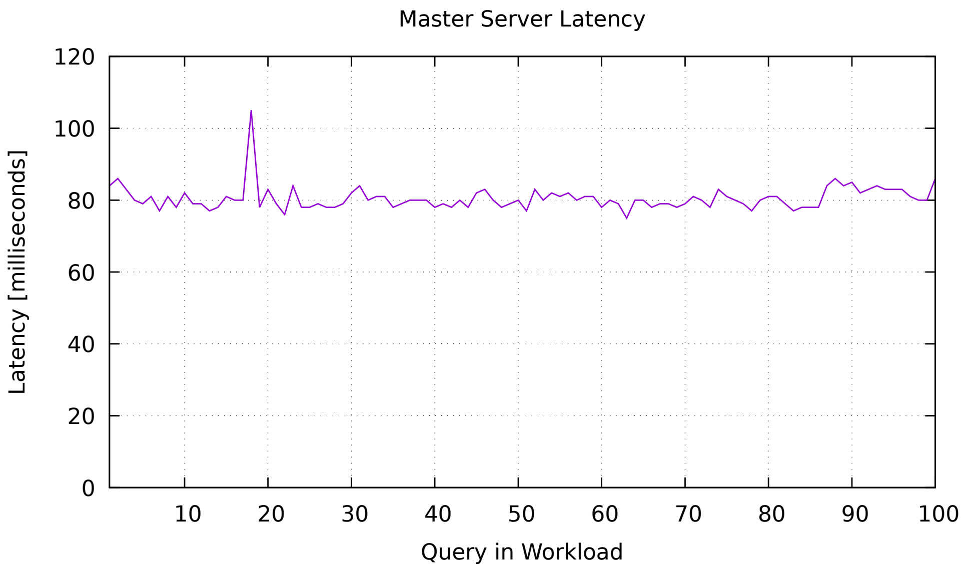


Figure 8.12: Master server latency in the *Single Query* workload on the *Weighted-Move* balancing technique, workload iteration 100

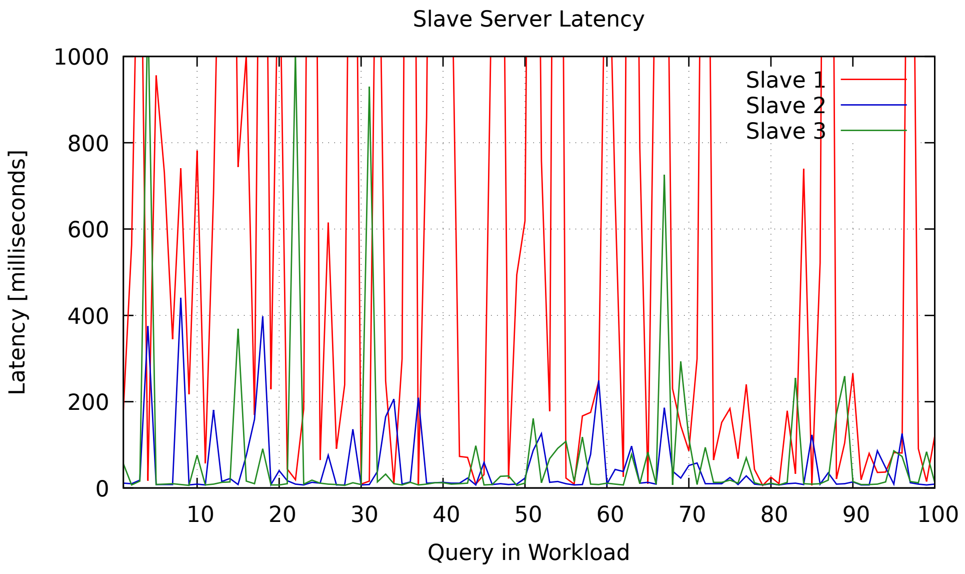


Figure 8.13: Slave server latencies in the *Production-like* workload on the *Weighted-Move* balancing technique, workload iteration 1

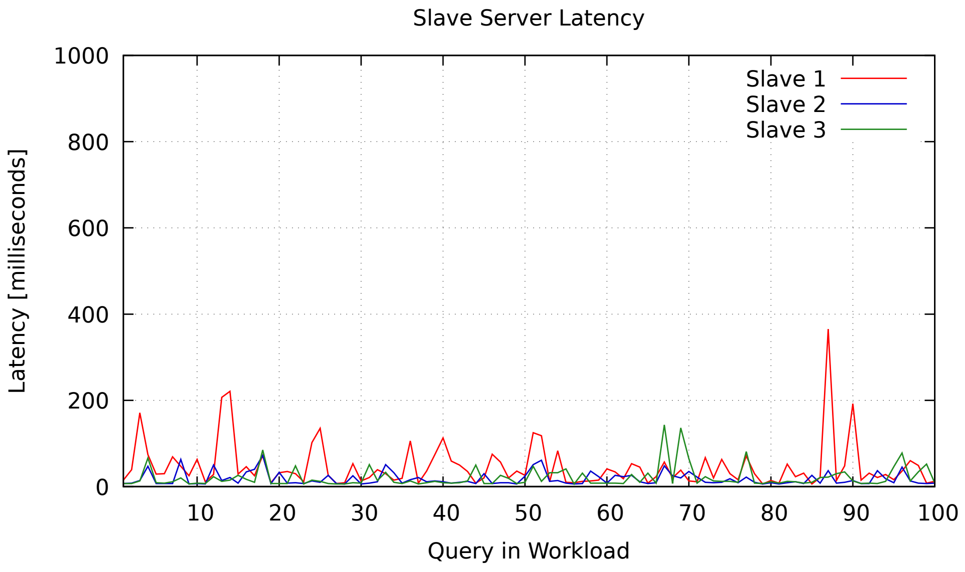


Figure 8.14: Slave server latencies in the *Production-like* workload on the *Weighted-Move* balancing technique, workload iteration 3

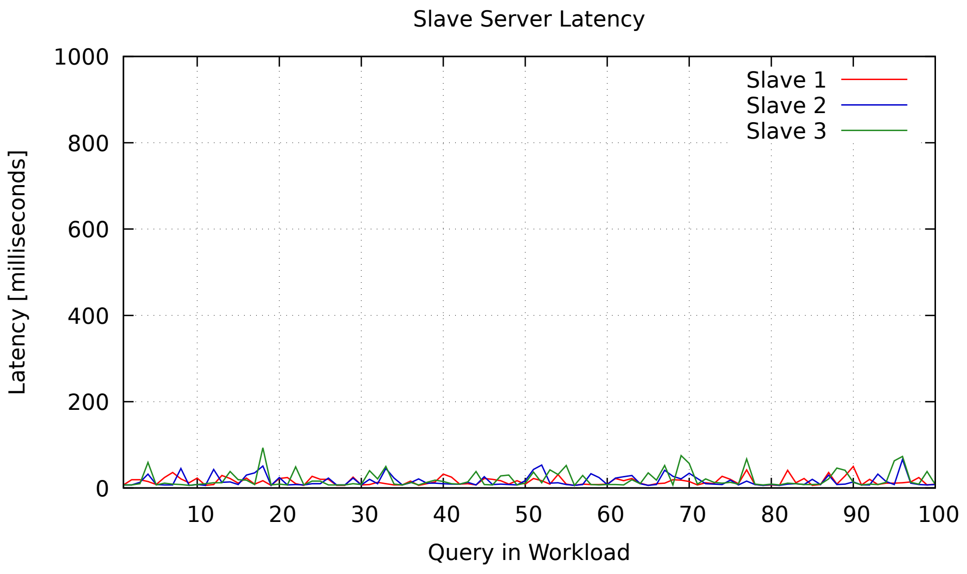


Figure 8.15: Slave server latencies in the *Production-like* workload on the *Weighted-Move* balancing technique, workload iteration 5

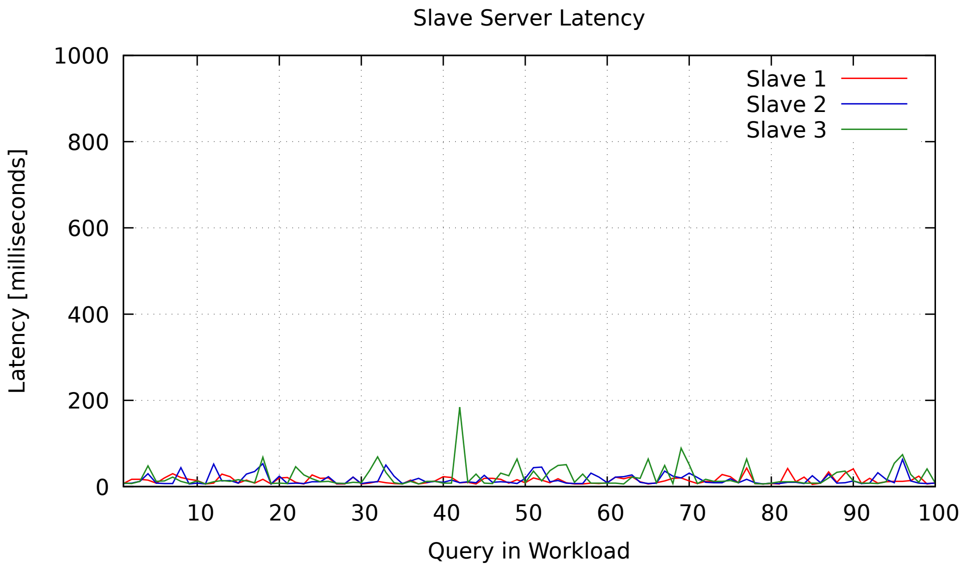


Figure 8.16: Slave server latencies in the *Production-like* workload on the *Weighted-Move* balancing technique, workload iteration 100

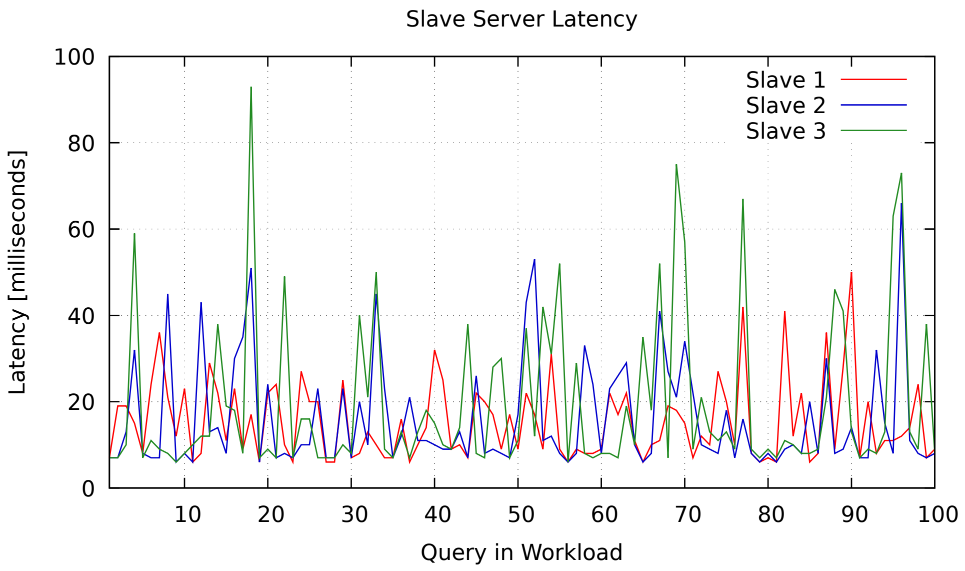


Figure 8.17: Slave server latencies in the *Production-like* workload on the *Weighted-Move* balancing technique, workload iteration 5, focused

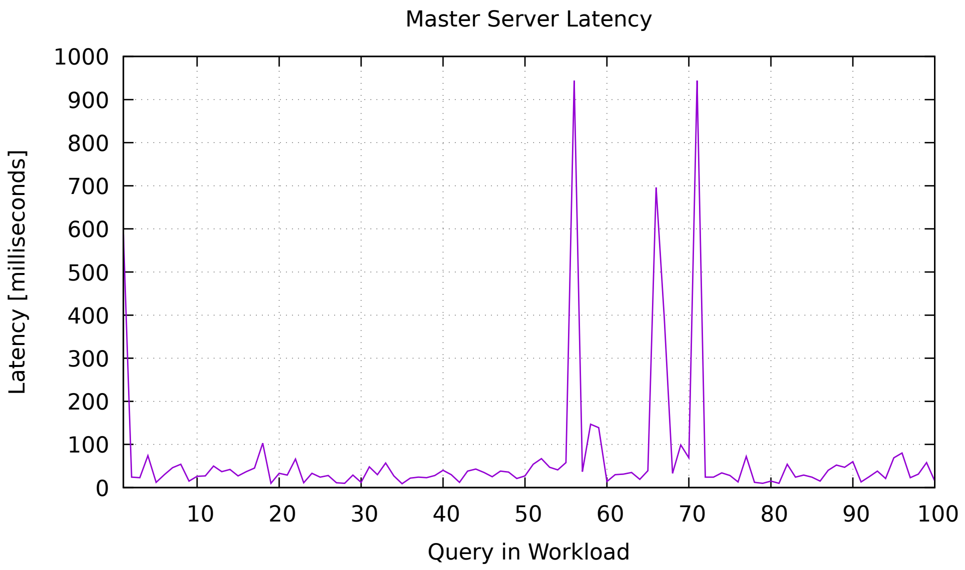


Figure 8.18: Master server latency in the *Production-like* workload on the *Weighted-Move* balancing technique, workload iteration 5

8.7 Future Work

There is surely further work needed in order to investigate what exactly caused the great variance in performance of the *Random* balancing technique.

Furthermore, we originally planned to conduct the experiment in scale comparable to the production system size so that the results would predict, as closely as possible, how the individual balancing techniques would behave if they would be really deployed into production.

Last but not least, we intended to compare more balancing techniques from chapter 4. We wanted to compare at least consistent hashing [11] (Section 4.4), ideally also some load-aware variant of consistent hashing (Section 4.4.2) and our idea for an advanced domain-specific balancing algorithm based on spreading data of the hottest topics (i.e. frequently occurring topics of queries).

All this was not done due to the time constraints for our thesis project and is subject of future work.

8.8 Conclusion

In this experiment, we compared three balancing techniques: (1) the *Original* balancing technique, which comes from the implementation of the original ad system, (2) the *Random* balancing technique, which is described in Section 6.5 and is based on random data distribution, and (3) the *Weighted-Move* algorithm of *Google Slicer* [12], which is described in Sections 4.5 and 6.5.

Measurements of the *Random* balancing technique showed much greater performance variance and also worse performance results than measured at the *Original* approach. We believe, however, that further work would resolve these issues.

As for our implementation of the *Weighted-Move* technique, it showed comparable performance results to the *Original* approach with quite fast reaction to data imbalance, which is a very useful feature.

In Section 8.7, we outlined ideas for future work that we expect to produce further interesting conclusions.

9. Conclusion

In this chapter, we would like to summarize our results from the whole thesis project (Section 9.1), list the greatest challenges of our work (Section 9.2) and overview suggested future work (Section 9.3).

9.1 Results

We accomplished goals of this thesis listed in Chapter 3. We analyzed the current state of the art in the area of sharding in distributed systems and conducted an experiment comparing the *Google Slicer's* balancing algorithm *Weighted-Move* [12] with a system-specific balancing technique and a base-line of a random balancing algorithm.

We discovered that our implementation of the *Weighted-Move* balancing technique is comparable to the original system's load balancing and reacts on data imbalance quite fast. There is certainly a promising potential.

Since measurements of our random balancing algorithm showed yet unexplained behavior, additional work on the data balancer implementation is appropriate and may result in further improvements of the *Weighted-Move* balancing technique over the original system's load balancing.

Not only for purposes of the experiment, we engineered a data balancer into the *Sklik* ad system.

We targeted all specified limitations of the current ad system, i.e. (1) the data balancer uses a finer granularity of distributed data, (2) we implemented a mechanism that reduces the time when the system data are in an inconsistent state when the backend data changes, and (3) in some extent, we provided a mechanism how to uniformly spread data with hot topics (we did not implement any concrete balancing algorithm that would do this exactly – but the implementation is perfectly ready for this).

We even went further and delivered a generic data balancer, which is deployable also to other systems that need data sharding, and which has interesting quality attributes. All advantages of our solution are listed in Section 6.3.14.

9.2 Project Challenges

In our humble opinion, we found this thesis project quite challenging. Its scope is quite non-trivial: it tries to capture the state-of-the-art approaches to sharding and to an architectural design of data balancers while providing a proof-of-concept implementation that was being designed and engineered as production-ready.

However, there is, naturally, some future work to the implementation that would be required in order to deploy the implementation into production.

The following list overviews some of the greatest challenges throughout the project:

- We began with zero domain-specific knowledge of the *Sklik* ad system. There was a lengthy process of learning and mistakes. The most significant events even influenced the concept of this thesis (which is presented in Chapter 5), and cost us a significant amount of time.
- As we desired to provide a state-of-the-art data balancer with attractive quality attributes, the implementation got really complex. We had to tackle many different kinds of problems related to distributed environment. To pick a few, issues related to consistency, synchronization, performance, congestion caused by memory exhaustion or remote communication. We put great importance to the system's architectural design.
- Even though our implementation was quite independent from the rest of the ad system, there were reasons why we had to keep pace with the ad system's development during our whole work. There were even technological changes or ports to newer versions of the operating system.
- We ran into many compatibility problems with the rest of the ad system. A lot of them originated from different features in different versions of libraries. The problems were partially caused by our lack of knowledge of the ad system but also by its continuous development.

There were, of course, many other problems but they are out of scope of this section.

9.3 Future work

In this last section, we would like to list a few ideas how to further investigate the topic of this thesis. Some of them are present in our plan beyond this thesis project and some of them are left for somebody else.

First of all, here is a list of further goals, that are of rather theoretical or experimental nature:

1. We originally planned to benchmark more balancing techniques than we eventually were able to fit into the time available for our thesis project. These are the algorithms that have not been covered in our experiment in Chapter 8 but are surely interesting to compare:
   (a) A basic variant of consistent hashing [11] covered in Section 4.4. Possibly also with different strategies presented in Section 4.4.1.

   (b) Some of the load-aware variants of consistent hashing, which are discussed in Section 4.4.2.
   (c) The *Autoplacer* algorithm [22] described in Section 4.6. However, its implementation might be a little challenging.
   (d) A technique based on rendezvous hashing [23], which is briefly introduced in Section 4.7.
   (e) And last, some kind of advanced domain-specific load-aware balancing algorithm. Comparison of advanced domain-specific and domain-independent balancing techniques would be especially interesting.

2. Generally, the rendezvous hashing balancing technique [23] and its possibilities should be investigated more thoroughly.
3. Another originally planned task that did not fit into our schedule was extension of the experiment from Chapter 7 by an alternative with the *Cap'n Proto* serialization [39] and/or an alternative that uses the *C++* standard library's *map* instead of the *RocksDB* database.

   Experimenting with *Cap'n Proto* is interesting because it would provide a comparison with *FlatBuffers* in a complex benchmark.

   The *RocksDB* replacement for a purely in-memory storage is attractive as well as it better fits today's cloud environments. We originally wanted a database technology supporting disk-storage but the arguments stopped being relevant when our goals changed, which is described in Chapter 5. The *C++ map* is interesting due to its simplicity but there are also other alternatives such as *Memcached* [13] for example.

Second and last, the following list overviews future work on our implementation:

1. The performance of the data balancer should be improved. It may be a problem for large data scales. The implementation is ready for scaling the distributors count, which is the bottleneck, but it requires creation of a proxy handling remote communication between the mapper and the distributors. Furthermore, it would be ideal to parallelize the distributor into more threads. That is also not a big problem, but the implementation is not prepared for it.
2. The initialization of the system is a bottleneck. The API of the data balancer could be improved as discussed in Section 6.3.12.
3. Fault-tolerancy was not our goal due to the thesis project scope but the data balancer robustness is not on a good level. For example, a crash of the mapper component would require a complete data redistribution. The implementation does not contain any obstacles for improvements in this area but it requires a non-trivial amount of work.

Bibliography

[1] YouTube. `https://www.youtube.com/`. [Online; accessed 15-April-2020].

[2] Google Drive. `https://www.google.com/drive/`. [Online; accessed 15-April-2020].

[3] Amazon. `https://www.amazon.com/`. [Online; accessed 15-April-2020].

[4] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., USA, 1995.

[5] Protocol Buffers. `https://developers.google.com/protocol-buffers`. [Online; accessed 27-March-2020].

[6] David S Johnson. Fast algorithms for bin packing. *Journal of Computer and System Sciences*, 8(3):272–314, 1974.

[7] Jeff Shute, Mircea Oancea, Stephan Ellner, Ben Handy, Eric Rollins, Bart Samwel, Radek Vingralek, Chad Whipkey, Xin Chen, Beat Jegerlehner, Kyle Littlefield, and Phoenix Tong. F1 - the fault-tolerant distributed RDBMS supporting Google's ad business. In *SIGMOD*, 2012. Talk given at SIGMOD 2012.

[8] Y Qi, X Shen, L Zhang, and X Li. Workload-aware data placement for cloud computing. Technical report, North Carolina State University. Dept. of Computer Science, 2017.

[9] Atul Adya, Robert Grandl, Daniel Myers, and Henry Qin. Fast key-value stores: An idea whose time has come and gone. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, HotOS '19, page 113–119, New York, NY, USA, 2019. Association for Computing Machinery.

[10] Aaron Archer, Kevin Aydin, Mohammad Hossein Bateni, Vahab Mirrokni, Aaron Schild, Ray Yang, and Richard Zhuang. Cache-aware load balancing of data center applications. *Proc. VLDB Endow.*, 12(6):709–723, February 2019.

[11] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. In *Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing*, STOC '97, page 654–663, New York, NY, USA, 1997. Association for Computing Machinery.

[12] Atul Adya, Daniel Myers, Jon Howell, Jeremy Elson, Colin Meek, Vishesh Khemani, Stefan Fulger, Pan Gu, Lakshminath Bhuvanagiri, Jason Hunter, Roberto Peon, Larry Kai, Alexander Shraer, Arif Merchant, and Kfir Lev-Ari. Slicer: Auto-sharding for datacenter applications. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 739–753, Savannah, GA, November 2016. USENIX Association.

[13] Memcached. `https://memcached.org/`. [Online; accessed 27-March-2020].

[14] Avinash Lakshman and Prashant Malik. Cassandra: A decentralized structured storage system. *SIGOPS Oper. Syst. Rev.*, 44(2):35–40, April 2010.

[15] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. In *Proceedings of Twenty-First ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, page 205–220, New York, NY, USA, 2007. Association for Computing Machinery.

[16] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking*, 11(1):17–32, 2003.

[17] Roshan Sumbaly, Jay Kreps, Lei Gao, Alex Feinberg, Chinmay Soman, and Sam Shah. Serving large-scale batch computed data with project Voldemort. In *FAST*, volume 12, pages 18–18, 2012.

[18] Xiangdong Huang, Jianmin Wang, Yu Zhong, Shaoxu Song, and Philip S Yu. Optimizing data partition for scaling out NoSQL cluster. *Concurrency and Computation: Practice and Experience*, 27(18):5793–5809, 2015.

[19] X. Wang and D. Loguinov. Load-balancing performance of consistent hashing: Asymptotic analysis of random node join. *IEEE/ACM Transactions on Networking*, 15(4):892–905, 2007.

[20] Ananth Rao, Karthik Lakshminarayanan, Sonesh Surana, Richard Karp, and Ion Stoica. Load balancing in structured P2P systems. In *Peer-to-Peer Systems II*, pages 68–79, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[21] Lakshminarayanan Srinivasan and Vasudeva Varma. Adaptive load-balancing for consistent hashing in heterogeneous clusters. In *2015 15th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, pages 1135–1138. IEEE, 2015.

[22] João Paiva, Pedro Ruivo, Paolo Romano, and Luís Rodrigues. Autoplacer: Scalable self-tuning data placement in distributed key-value stores. *ACM Trans. Auton. Adapt. Syst.*, 9(4), December 2014.

[23] Sally M Elghamrawy and Aboul Ella Hassanien. A partitioning framework for Cassandra NoSQL database using rendezvous hashing. *The Journal of Supercomputing*, 73(10):4444–4465, 2017.

[24] RocksDB. `https://rocksdb.org/`. [Online; accessed 10-May-2020].

[25] Leslie Lamport. Time, clocks, and the ordering of events in a distributed system. In *Concurrency: the Works of Leslie Lamport*, pages 179–196. 2019.

[26] Simon St Laurent, Joe Johnston, Edd Wilder-James, and Dave Winer. *Programming Web Services with XML-RPC: Creating Web Application Gateways.* " O'Reilly Media, Inc.", 2001.

[27] gRPC. `https://grpc.io/`. [Online; accessed 10-May-2020].

[28] Boost. `https://www.boost.org/`. [Online; accessed 02-Jun-2020].

[29] Redis. `https://redis.io/`. [Online; accessed 01-Jun-2020].

[30] NoSQL performance benchmark 2018 – MongoDB, PostgreSQL, OrientDB, Neo4j and ArangoDB. `https://www.arangodb.com/2018/02/nosql-performance-benchmark-2018-mongodb-postgresql-orientdb-neo4j-arangodb/`. [Online; accessed 01-Jun-2020].

[31] Benchmarking LevelDB vs. RocksDB vs. HyperLevelDB vs. LMDB performance for InfluxDB. `https://www.influxdata.com/blog/benchmarking-leveldb-vs-rocksdb-vs-hyperleveldb-vs-lmdb-performance-for-influxdb/`. [Online; accessed 01-Jun-2020].

[32] Research scientists put RocksDB on steroids. `https://research.yahoo.com/news/research-scientists-put-rocksdb-steroids/`. [Online; accessed 01-Jun-2020].

[33] ArangoDB. `https://www.arangodb.com/`. [Online; accessed 01-Jun-2020].

[34] Flat Buffers. `https://google.github.io/flatbuffers/`. [Online; accessed 03-Jun-2020].

[35] Debian Jessie. `https://www.debian.org/releases/jessie/`. [Online; accessed 10-Jun-2020].

[36] top. `https://linux.die.net/man/1/top`. [Online; accessed 10-Jun-2020].

[37] GCC. `https://gcc.gnu.org/`. [Online; accessed 09-Jul-2020].

[38] Debian Stretch. `https://www.debian.org/releases/stretch/`. [Online; accessed 09-Jul-2020].

[39] Cap'n Proto. `https://capnproto.org/`. [Online; accessed 12-Jul-2020].

List of Figures

A. Appendix

A.1 Attached Implementation

The attachment to this thesis contains a major part of our implementation. In this appendix, we briefly describe the attachment's structure. We do not introduce the listed components again, this was already covered by Section 6.7.

Note that, due to confidentiality reasons, we did not attach the complete source code we developed. There are only those components that come completely from our work and are not confidential. Neither we attached binaries and header files that are required for compilation and running of our attached implementation. We also removed certain small fragments in the attached code that contained confidential information.

All the attached components are located in the `implementation` directory. Each component is contained in a subdirectory with the component's name. All these subdirectories are structured in the same fashion, which is described below. Note that each component contains only a part of the following structure, which is relevant for that component.

- `README.md` and the `doc` directory contain user documentation.
- `CMakeLists.txt` serves for compilation of the component.
- The `debian` directory contains metadata for building the component into a *Debian* [35, 38] package.
- The `include` and the `src` directories contain *C++* header files and *C++* source files, respectively.
- The `proto` directory contains *Protocol Buffers* [5] definition files.
- The `conf` directory contains a configuration file for the given component.
- The `test` directory contains unit, integration, component and/or system tests.

A.2 Keyword Search Experiment Data

Complete data from the experiment presented in Chapter 7 are attached to this thesis in the directory `keyword_search_experiment`. The data are separated in subdirectories according to the measurement setups described in Section 7.2.

In each subdirectory, one can found *PDF* or *PNG* files with diagrams generated from the measured data.

The raw measured data are present in the text files that are named similarly to the implementation labels presented in Section 7.1. Each text file contains a header with computed average and variance, followed by individual measured values.

The scripts, which were used as the benchmark, were created ad-hoc and contain confidential information. Therefore, they are not attached to the thesis in any way.

A.3 Balancing Experiment Data

All data from the experiment that was subject of Chapter 8 are attached in the directory `balancing_experiment`.

There are three subdirectories, one for each balancing technique compared in the experiment. Each then further contains another level of directories, each for a different workload. The *Weighted-Move* balancing technique then contains even another directory level which determines the workload iteration (we repeated the workload 100 times).

The raw measured data are present in text files where the first column represents the system time when the particular latency was measured and the second column contains the latency value itself, in microseconds.

The PDF files present generated graphs from the text files.

Scripts used as the benchmark were created ad-hoc and contain confidential information. Therefore, they are not attached to the thesis in any way.