FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Sharing Configuration Snippets with Community using REST and Web GUI

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Software & Information Engineering

by

Marvin Mall
Registration Number 1225943

to the Faculty of Informatics

at the TU Wien

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Franz Puntigam
Assistance: Univ.Ass. Dipl.-Ing. Markus Raab

Vienna, 22nd December, 2016

Marvin Mall	Franz Puntigam

Acknowledgements

I want to thank Markus Raab for his support during the writing of this thesis. His knowledge of Elektra and all work that is currently ongoing around the project was really helpful and essential to get the thesis done.

Furthermore I want to thank my parents for making this study as well as this thesis not only possible, but also very convenient and untroubled due to financial and moral support along the entire way.

Abstract

Configuration is what makes today's software as powerful as it is. It allows for distinct behavior of software systems without changing the actual program code itself. Obviously, there is not a single way to achieve configurability, which leads to the problem that many configuration paradigms, characterized by their systems, languages and formats, have developed over time. Elektra, a universal framework solving cross-platform related issues, strives to abstract from these differences and offers a uniform way to access application configurations. But even with features like configuration specification and validation, users often have a hard time figuring out what options they need to utilize in order to achieve their goals. This is the point where the developed REST service for sharing of configuration snippets steps in. It allows users to share commonly used configuration snippets and in return also to search, convert and download the snippets of others. Because the REST service stores data solely with the help of Elektra, we made a comparison, both on a theoretical as well as practical basis, between Elektra and a conventional database system. The results show that, Elektra, (1) lacks several essential features of a relational database system, (2) offers enough customizability to achieve most tasks decently, and (3) scales not nearly as good as a MySQL database.

Contents

CHAPTER 1

Introduction

1.1 Problem Statement

Nowadays, most applications allow configuration to alter their runtime behavior. But implementing a configuration system is not really done to ease development, rather to improve user experience and to cover more use cases at the same time. It is only the configuration possibilities that make most software as powerful as they are.

While various applications use different approaches to realize configuration, Elektra offers a uniform way of accessing and maintaining application configurations in a global, hierarchical key database. The Elektra framework does not only help to eliminate configuration duplication, it also assists in avoiding cross-platform related issues and offers convenient means to override configurations for individual users, directories or even processes.

Although Elektra aims for direct application integration (also called *elektrification*), it is also possible to mount configuration files of applications with independent configuration systems into the key database. This allows the user to manipulate said configuration files through the global key database in a uniform way and facilitates the usage of Elektra plugins (e.g. notification on configuration changes) while remaining independent from the Elektra library itself [Raa16].

One problem that Elektra does indeed solve for elektrified applications utilizing configuration specification, but not for mounted configuration files of applications with independent configuration systems, is the lookup of possible configuration parameters. While most applications already ship with a basic, most-commonly used default configuration, there are still often use case scenarios where other or more advanced configuration is favored or even required. Often it is not sufficient to look at the current configuration to find a way to attain the desired behavior.

Because individual applications follow different approaches when it comes to documentation distribution (especially for configuration possibilities), it can be quite cumbersome to configure applications to attain the desired behavior. Some developers ship a comprehensive manual with their application, others use their website as information source. The lack of a uniform strategy makes it often a lot harder for users to inform themselves as practically necessary, especially if only a small change is desired. This seems to be a bit ironic considering what configuration was originally meant to be for - making everyone's life easier.

1.2 Goal of this Thesis

The goal of this thesis is to develop a service that allows developers and administrators, as well as other users to search for configuration examples of arbitrary applications. The service shall serve as easy-to-use, central platform, giving the opportunity to avoid crawling through application documentations. It shall be powered by its users, which shall have the ability to share configuration snippets that are considered useful or commonly used. Through the power of Elektra, shared configuration snippets should then be downloadable in any desired and supported format.

As first step, the Elektra framework itself shall be targeted. Since Elektra is used to implement said service and to store configuration snippets, it shall be compared against another database system. This shall be MySQL, a commonly used, SQL based, relational DBMS (Database Management System). The intention is to learn and understand important differences, but also similarities between Elektra and aforesaid DBMS.

Following this, the developed REST service shall be presented and benchmarked against a solution based on MySQL, with the goal to show Elektras competitiveness as conventional database in real-world applications.

In particular, the questions to answer are:

RQ1 What are the differences in functionality between Elektra with its hierarchical key database and a conventional SQL based, relational DBMS?

RQ2 Measured within the scope of the developed REST service, is Elektra a suitable replacement for a SQL based DBMS?

1. Which system does compete better when it comes to the lookup of entries by meta information (i.e. full text search)?
2. What impact does the effective database size have on the performance provided by both the REST service and the DBMS?

CHAPTER 2

Elektra as Database Management System

Storing the data of the developed REST service could be done anywhere, starting from plain text files to a large-scale, distributed database system. The magic-like conversion of configuration snippets does not happen in the storage, but in the REST service with the help of Elektra and its plugins. Yet it makes sense to use Elektra or better said its key database (KDB) as storage for the data as well.

This decision brings some risks with it as Elektra was never designed to act as large-scale data store and even less to be used as database management system for arbitrary data. But still it offers a convenient way to store structured data, which is the essential core of any database system out there. Of course most tasks require some additional effort which would normally be handled by the database management system, although this also certainly gives more freedom when it comes down to doing things.

Whether it is worth using Elektra as data store for a project or not is a question that cannot be answered very easily. The decision is very similar to SQL vs. NoSQL (of course with some additional constraints), which is also the reason why this comparison will be influenced by previous comparisons of said database designs. Similar comparisons have already been made by *Alexandru Boicea, Florin Radulescu and Laura Ioana Agapin* between MongoDB and Oracle [BRA12], by *Luke P. Isaac* between multiple SQL and NoSQL solutions [Iss14] and by *Matti Paksula* very generally between relational, SQL-based database systems and Redis as simple key-value store [Pak10].

But not only SQL vs. NoSQL is interesting, also the comparison between different SQL systems makes sense as their features vary often quite significantly. Therefore we will also have a look at SQLite and PostgreSQL [Gro16, Aut16] besides MySQL [AWT16]. Another resource next to the manuals of said DBMS is the thesis written by *Yang Xiaojie* about differences between MySQL and PostgreSQL [Xia11].

In case it is not clear yet, Elektra can be considered a NoSQL database as it does not store its data in a relational, table-like design. Yet it is no full database management system at all, due to very fundamental features being missing, as shown later. This shall not hamper us in our comparison though. We will simply try to come up with ideas where no straight forward implementation or concept is available yet.

2.1 About Elektra

According to [Raa16], Elektra basically consists of three main classes, namely *Key*, *KeySet* and *KDB* (*Key Database*). When talking about a *Key*, we always refer to a pair consisting of a unique name and a value (e.g. one line in an *ini* configuration file where the left side of the equation mark is the key name and the right side the value). A *KeySet* is a set of *Keys*, always sorted in alphabetical order by the key name. *KDB* is the managing class for access to the global key database, which delegates a lot of work to *backends*.

In its core, Elektra is a key-value store, utilizing a global hierarchy for its *Keys*. The term *global* stems from the fact that the key database can be accessed system-wide, although some keys might not be visible or accessible for all users and applications (i.e. keys in the *user* namespace are user-dependent).

Mentioned namespaces are not only useful for (implicit) access control, but also for overriding of configurations. It is possible to define a system standard configuration that can be overridden for every user, directory or even process (namespaces *user*, *dir* and *proc* respectively). But it is not necessary to override the entire configuration (for an application), Elektra allows to operate on every single key-value pair itself, which avoids unnecessary redundancy. Also other overriding and fallback strategies are possible through usage of the specification (*spec*) namespace. Applications need to make use of cascading keys to lookup configuration, if they want to utilize the overriding and fallback functionality.

The way Elektra stores and reads configuration depends on the used *backend*. Backends always consist of multiple plugins, but at least a *resolver* and a *storage plugin*. The resolvers task is to resolve filenames for storage files. Storage plugins are responsible for serializing a *KeySet* into a configuration format (as part of the `set` phase) as well as parsing the same back into a *KeySet* (during the `get` phase). Other plugins such as *filter plugins* can be added to backends as well. A more detailed description of the plugin concept can be found in [Raa10, Raa16].

Besides the core of Elektra, we also have *tools* that either help us accessing the key database (such as the equally-named command-line tool *kdb*) or utilize the library in another way. The developed REST service is one of these tools.

2.2 MySQL and Elektra

Whilst MySQL is a very typical RDBMS (Relational Database Management System) utilizing SQL as query language, Elektra is not really a DBMS at all, neither relational nor document-oriented like the NoSQL DBMS MongoDB. At its core, Elektra can be considered a data store (which is basically everything that allows for storage of data, see *Characteristics of the Database Approach* in [EN10]) with some additional features, of which a lot are made available through plugins and tools [Raa16]. This leads to a system that can be utilized as database, but is not a *database management system*.

Consecutively we will try to give some insight about important aspects of a database management system, both from the perspective of MySQL as well as Elektra. The comparison will certainly not be able to address every aspect of such systems, which can easily be seen by looking at the scale of [EN10] or [AWT16]. It should rather be seen as attempt to compare Elektra against something it potentially could be.

2.2.1 Relations, Documents and Keys

MySQL

"In the formal relational model terminology, a row is called a *tuple*, a column header is called an *attribute*, and the table is called a *relation*. The data type describing the types of values that can appear in each column is represented by a *domain* of possible values.", as written in [EN10, p. 61]

Proper definition of this relations and their attributes is essential for the DBMS to be able to search through the data and lookup information. On the one hand it allows the DBMS to optimize but on the other hand it also enables applications utilizing the database to access the data properly (especially important for statically typed programming languages [1]). Column definitions can also contain access hooks [2], default values and more [EN10].

If someone would want to retrieve the email of a user, he would probably use an SQL query like `SELECT 'email' FROM 'users' WHERE 'name' LIKE = 'Jeff'`. In this example we are expecting the user *name* to be the unique identifier (also called primary key) of the table *users*. The outcome would be a result set (basically a table) consisting of one column and one row.

MongoDB

One of the bigger NoSQL groups are the document-oriented database systems. As described by [Str], they use *documents* to store their data instead of tuples. The structure of these documents can be uniform, but it does not necessarily have to be. It is also

[1] Typed languages use meta data to classify what kind of data is stored in a chunk of memory. Statically typed languages are normally referred to as having this information available at compile time already [Aah03].

[2] Like `ON UPDATE CURRENT_TIMESTAMP`, for example.

possible for every document to have a unique structure, i.e. different fields. The only required characteristic is an identifier field that is set on every document with a unique value (i.e. the primary key). Documents themselves are stored in *collections* (cf. relations). In the following example we show two documents represented by using JSON objects (which is the readable form of the stored BSON [3] objects):

```
{
    "name" : "Jeff",
    "email" : "jeff@example.com",
    "job": "Developer"
}
{
    "name": "Max",
    "telephone": "+1 1234 123456789",
    "university": "Technical University of Vienna"
}
```

In MongoDB, one of the most used NoSQL database systems for web development, documents are stored in *collections*. If one had to compare MongoDB to MySQL, they could say that collections are tables, with the important difference that their documents (rows) are not limited to columns defined by the collection (table) [Iss14].

MongoDB does also not use a query language like SQL, but rather a query interface. To find the document of Jeff, one would use a query like `db.users.find({"name": "Jeff"})`. They would then receive a JSON object like described above, which would give them immediate access to Jeff's email.

An obvious drawback is that one will have to check for the existence of properties (fields) in their application code as it is not required for every document to have the same properties set. This can also be seen as one of the bigger strengths of document-oriented databases though, as it allows for application development without having to adhere to a structure prescribed by the database.

Elektra

None of those structural restrictions is a thing when using Elektra. There are no relations, no collections and no documents. Elektra is a very simple key-value data store like Redis (with the difference that Redis does not persist its data and is not as flexible as Elektra) [Pak10].

On the one hand, this gives a lot of freedom when it comes to storing data, because values can be stored under almost any key in almost any format. On the other hand, this might not even be desired for some applications. For example, in some applications wrong data

[3] BSON, short for *Binary JSON*, is a binary-encoded serialization of JSON-like documents (see http://bsonspec.org/).

types can cause runtime issues (luckily this can be avoided by using *validation*) or if the data is expected to have a certain structure for each element of an array, the freedom is useless as a relational database system would offer the same possibilities with probably better performance.

The content we store below a key in Elektra is not limited. If no suitable data format supported by Elektra is available, it is still possible to store data as binary. This way arbitrary data can be stored, even videos or other things that are normally not kept in a database at all (that does not mean other DBMS cannot store binary data, most systems support BLOBs [4]; it is just highly discouraged to do so, at least in RDBMS).

A difference to other database systems is that we have an absolute key for each value. If we would want to store a JSON object like in the MongoDB example in the key database, we would have to create several (absolute) keys. Whether we create a key for the root of the object or not is personal preference, but it does make sense if we want to filter for stored *users*. Creating the keys via the `kdb` tool would look like this:

```
kdb set /myapp/users/Jeff ""
kdb set /myapp/users/Jeff/email "jeff@example.com"
kdb set /myapp/users/Jeff/job "Developer"
```

Note that the name is implicitly stored as part of the key. If we would replace the name by a unique numeric ID as identifier, we could of course also store it as separate value as well. We could also store a complete JSON document (like in MongoDB) as value below a certain key (i.e. as string), but this would contradict the purpose of Elektra. The goal is to eliminate application specific formats, that is to offer a uniform way to access and represent (configuration) data, and to leave the actual storage task to plugins [Raa16].

2.2.2 Information Retrieval

Probably the most fundamental task of a database system is the retrieval of information (besides the insertion of new data, which we will discuss in section 2.2.3). Very often we don't want to retrieve all information, but only parts of it, i.e. limit the result by some (search) parameters.

In MySQL

We have tables with columns and rows, where we select rows that fulfill certain search criteria (e.g. `WHERE 'name' = 'Jeff'`). To execute such lookup operations, the DBMS offers an interface or better said a query language called SQL (hence the name MySQL). Retrieving rows based on their primary key may not be the best example, as this can be done within Elektra pretty easily as well. Therefore we should probably mention that all common (SQL based) DBMS can also lookup information based on any non-indexed column as filter criteria easily, even if not always very performant.

[4] Binary Large Object

To increase performance of lookups, we can add indexes to our database tables. Indexes are sorted copies of columns that can be used in queries with comparisons to filter and select rows quite efficiently. They are often stored as B-Tree, B^+-Tree or Hashmap. The latter does only deliver good performance for equality checks (i.e. `WHERE 'id' = 5`), while B-Trees also perform very well for in-equality comparisons (i.e. `WHERE 'id' > 5`). The biggest drawback of indexes, which is also the (only) reason why we try to avoid them, is that they increase the effort and time it takes to insert new and update existing rows. For small tables they may also increase the time to execute a lookup by introducing an additional step [5]. Besides that, they also consume additional disk space, as already mentioned before [EN10].

In Elektra

We can only lookup the keys below a certain point in the hierarchy, e.g. `/myapp/users`. Elektra or rather the key database itself does not offer a convenient way to look for keys that have a value matching certain search criteria, e.g. `value > 5`. Searching through a *column* like in MySQL is not possible either because of the missing structure that would be required to do so. Such tasks have to be done in the application using the key database or in a plugin, what makes lookups more expensive, regarding development time as well as performance itself.

It is also not easily possible to tell which keys below a path belong together (i.e. *build an entry*, assuming that we try to store a table of rows from MySQL in Elektra). We need to define a contract in order to know so, whereby the contract could be realized in many ways, of which some are:

1. Flag the key that represents the root of an entry with some meta value (e.g. `kdb setmeta /myapp/users/Jeff isentry true`).
2. Flag the key that represents the root of an entry with some special key value (e.g. `kdb set /myapp/users/Jeff "this-is-an-entry"`).
3. Use a well-defined layout for root keys (e.g. one level below the *table* root: if `/myapp/users` is the table root, `/myapp/users/Jeff` would be an entry root key).

If we can look for entry root keys, we can then tell that their child keys are attributes.

It would be possible to write plugins for filter tasks though, which could use the plugin configuration feature to allow for run-time defined behavior and therefore also for being reused among different applications. Unfortunately, at the time of this writing, such a plugin does not exist yet.

[5] DBMS read data in blocks or pages from the disk (e.g. 2, 4, 8, 16 or 32 KB at a time). If a whole table fits within one block, reading an index first would introduce an additional, unnecessary step [Pow06].

2.2.3 Data Insertion and Update

Another important database use case is data insertion and update. This tasks seem very straight forward, but there are some side-aspects that need to be considered in a comparison.

MySQL

Because of indexes the DBMS has to insert or update multiple occurrences of data during an insert or update operation. Inserting a row in a table itself is rather trivial as it is simply appended in byte-form to the file containing the table data. But for each index on the table, the corresponding column value has to be stored a second time along with the row offset in an index file. This takes time, as the index needs to be sorted (i.e. the correct place for the new index entry has to be determined first). In case of an update, the old index entry has to be deleted or moved, which increases the effort further [EN10].

Indexes should therefore be avoided for columns where either not many searches or where above-average many insert or update operations are performed (e.g. logging table).

Elektra

In Elektra we have no indexes (yet), therefore data is only stored once (if no plugin is used which duplicates data). Storing a whole new entry (i.e. a number of new values representing a new row in a MySQL table) in the key database can be a lot more effort than in MySQL though, as we have to create a key-value pair for each *column* (field). As an example, look at this code:

MySQL:

```
INSERT INTO `users` (`name`, `email`, `job`)
   VALUES ('Jeff', 'Jeff@example.com', 'Developer')
```

Elektra:

```
KeySet * ks = ksNew(3,
   keyNew("/myapp/users/Jeff", KEY_END),
   keyNew("/myapp/users/Jeff/email", KEY_VALUE, "Jeff@example.com",
      KEY_END),
   keyNew("/myapp/users/Jeff/job", KEY_VALUE, "Developer", KEY_END),
   KS_END);
```

Of course we can optimize our application code, e.g. by using a struct holding the parent key and map for field names and values, which is then transformed by a function to a KeySet consumable by Elektra. This way we only have to construct a KeySet in one place of our application. Higher languages allow for even more powerful and convenient constructs.

To perform an update, we simply set a key-value pair again, which will possibly override existing data.

2.2.4 Multiple Users and Data Access

Database management systems allow for specifying different views on data for different users. This basically introduce access control, but it does not only serve the purpose of security (i.e. data protection), it also attempts to make the life of all users easier by only granting access to the part of the data that is actually relevant for the user or application [EN10].

MySQL

Access control is realized by granting and revoking privileges to special user accounts. This privileges can reach from very loose (like account level based access to a certain functionality or database) to very accurate (like read access on a specific part of a table). With the help of views [6] and stored routines [7], even more detailed access control is possible [EN10].

Elektra

There is not really a specifiable access control system implemented in Elektra itself (at least not at the time of this writing), but there are related features that can be utilized to create some sort of access control nevertheless. Elektra supports configuration overriding through namespaces. Although Elektra does not offer user accounts on its own, the operating system user accounts can be used to restrict access to different data. For example, it is possible to have data in the *system* namespace that is accessible by everyone and each user can have additional data in the *user* namespace that is only accessible by themselves. Granting another user access to the personal *user* data is not possible though (because Elektra is not yes able to express the permissions of a file systems ACL [8]) and all in all, there are way less options to restrict access than conventional DBMS offer [Raa16].

2.2.5 Disk Space

An interesting aspect of databases is the disk space being occupied. Although todays hard drives get cheaper every day, which makes this criteria less relevant for most applications, it can still be interesting for embedded systems with very limited storage space. Therefore we will have a small glimpse at the possibilities offered to reduce the required disk space.

MySQL

In MySQL, it is often not the raw data that takes the most disk space, but indexes on the data which are necessary to gain proper performance when looking up information

[6] Views are basically the result set of a stored query.

[7] Stored routines are also stored in the database system, but can do a lot more than only returning a result set. They may contain complex application logic that executes several queries in a row.

[8] *ACL* stands for Access Control List and is a list of permissions that can be attached to an object, e.g. a file.

(i.e. `SELECT` with `WHERE` in SQL). The rows of tables are normally stored unordered in byte-representation in a file with only the offset being known for each row. If we omit the indexes in favor of reducing the required disk space, we will have a drop in performance, although it does not really matter all that much in this scenario as the database would be very small anyway. In large-scale databases, it does not make sense to attempt avoiding indexes as the performance drop would be too tremendous. Here we normally trade disk space for performance.

Some storage engines (of MySQL) like *InnoDB* support *table compression*, which allows for smaller page sizes on the disk (than the usual sizes). Alternatively it is of course also possible to store compressed data right away (i.e. compress the data in the application layer instead of the storage layer, which makes the compression independent from the data store) [AWT16].

Elektra

Other than MySQL there is not *one* storage format being used in Elektra for everything. Through the plugin system it is possible to create any desired storage format. There are already many different storage plugins (for all common configuration formats) available, but the system gives us the ability to develop custom plugins tailored very heavily to a use case as well. We could for example create a storage plugin that can be used with exactly one data schema (which let us omit key names) and reduces the size of stored data through `LZ77` or `LZ78` [9] compression even further (although that makes only sense for larger files and makes them non-human-readable):

```
// Example structure in ini format
// (347 bytes (325 bytes without whitespace))
[ipv4]
jupiter = 127.0.1.1
jupiter.homenetwork = 192.168.0.1
jupiter.homenetwork/jupiter = 192.168.0.1
localhost = 127.0.0.1
mars.homenetwork = 192.168.0.2
mars.homenetwork/mars = 192.168.0.2
saturn.homenetwork = 192.168.0.3
saturn.homenetwork/saturn = 192.168.0.3
[ipv6]
ip6-localhost = ::1
ip6-localhost/ip6-loopback = ::1
ip6-localnet = fe00::0

// Special format without key names (format: hosts)
// (201 bytes, ~58% (62%) size of ini)
127.0.1.1 jupiter
192.168.0.1 jupiter.homenetwork jupiter
```

[9] **LZ77** and **LZ78** (also known as **LZ1** and **LZ2**) are lossless data compression algorithms. These algorithms were crucial for the development of other compression algorithms such as **LZW** or **LZMA**.

```
127.0.0.1 localhost
192.168.0.2 mars.homenetwork mars
192.168.0.3 saturn.homenetwork saturn
::1 ip6-localhost ip6-loopback
fe00::0 ip6-localnet

// Additionally with LZ77 compression (window length 100)
// (144 bytes, ~70% size of uncompressed, ~41% (44%) size of ini)
127.0.1.1 jupiter
192.168.0` *%.homenetwork` >%` U 0.1 localhost` Q&2 mars` L
   (mars` s&3 saturn` p(` -!
::1 ip6-` q$` '"opback
fe00::0` 9%net
```

The same hosts file imported into a MySQL InnoDB table consisting of two columns (`CREATE TABLE 'hosts' ('ip' VARCHAR(40), 'aliases' VARCHAR(50))`) occupies 16.384 Bytes of space, which are exactly 16 KB and therefore the default (and in return also minimum) page size for a MySQL InnoDB database.

The somewhat larger hosts file from http://winhelp2002.mvps.org has a size of 392.493 Bytes (~383 KB) with all comments and unnecessary whitespace erased. The same file imported into the same MySQL table as described above occupies about 1.589.248 Bytes (~1552 KB) on the hard drive, without extra indexes being defined. So even with us not creating indexes by hand, MySQL does not seem to be very economical when it comes to storage space.

But it is not only the size occupied by the stored data, also the database system application itself occupies space. In case of Elektra, `libelektra-full` (which is a bundle of all other Elektra libraries excluding shared dependencies) has less than 1 MB. The average storage plugin has about 100 and 200 KB (excluding potential dependencies).
It probably does not make much sense to compare Elektra against MySQL here as the latter is not really meant to be used as embedded database. Instead we'll use SQLite (v3) as comparison (which is widely used in embedded environments), its library has about the same size as the Elektra library (< 1 MB). The only advantage of Elektra is that `libelektra-full` can be replaced by the smaller partial libraries if not all functionality of the bundle is used [Raa16]. SQLite, however, offers some other conveniences, as we will learn later on.

So in case a light-weight data store is required and no database supporting the SQL query language is needed, Elektra could be used as replacement.

2.2.6 Integrity

Another very important aspect of database management systems is guarantee on integrity. This is relevant in several scenarios, of which concurrent access and system failure are the most important.

MySQL

In [EN10] it is explained that relational database systems use transactions to ensure data integrity. Each write process on a database is encapsulated in a transaction, either explicitly by the user or implicitly by the system (e.g. single `INSERT` statement like in the example above). Transactions assure that no side-effects appear (e.g. overriding values because of concurrency) by executing operations following the principles of ACID [10]. They also ensure that in case of a system failure (crash of the database server), no data gets lost and the system state can be recovered without putting data integrity at risk (even in case of multiple, consecutive system failures).

To do so, the DBMS uses a very special logging technique that allows for recovery of the system state. Before data is actually written to the database, a log is created. This log also allows for recovery in case a transaction is aborted (explicitly by the user or application) and the previous database state has to be restored.

Elektra

Integrity: There is no database server running for the key database, every application accessing it runs an own instance of Elektra. Therefore crashes would be application specific and not related to the database itself, which forwards the responsibility for faulty recovery to the application using Elektra (wrong data is being rejected by Elektra of course, but recovery of not written (i.e. lost) data is not handled by Elektra).

Concurrency: It is the task of the resolver to ensure that concurrent calls to `kdbSet()` do not lead to overriding of configuration. It does so by locking single files, leading to serial access by the Elektra instances. Handling the blocking of threads is the responsibility of the implementing application. If concurrent calls do not access the same file, there is no need for locking though and the calls will be executed concurrently.

SQLite

SQLite can be seen as a mix between Elektra and MySQL regarding this aspects, because it is also run by the application using it (so there is also no extra server necessary to access the database). But one advantage of SQLite over Elektra is that it takes lots of responsibilities like transaction management and faulty recovery from the application using it, while still providing a mature SQL interface and good performance (at least up to a certain scale of concurrency) [Aut16].

2.2.7 Stored Routines and Plugins

A common interest of database management systems is to centralize logic and to allow for extensions. This is often achieved by the concepts of *stored routines* [11] and *plugins*.

[10] ACID stands for Atomicity, Consistency, Isolation and Durability.

[11] *Stored routines* is the umbrella term for stored procedures and stored functions.

Stored routines on the one hand are tasks that can be implemented (once) within the database system, so that they can be used by different applications. Such routines are normally written in one of the languages provided by the database system (e.g. SQL). Supporting a feature like that makes sense for most database systems as it reduces code-duplication and therefore eases maintenance, but it can also eliminate extra communication time between server and clients.

Plugins on the other hand are additional software modules that extend the functionality of a system. They are written independently from the database system, often in compiled languages like C or C++. At runtime the plugins are then being loaded and invoked by the target system, which then alters or complements the behavior of the system [Raa10, AWT16, Gro16].
In general, plugins are considered the more powerful concept, although they often pursue entirely different goals.

MySQL

MySQL supports stored procedures, stored functions and plugins. The first two can be managed through the SQL interface, while the latter has to be processed by a compiler and can be loaded as well as unloaded by the database management system at start or run time. This requires that the plugin is stored on the same server as the DBMS though, it cannot be installed remotely through the SQL interface.

The power (features) of stored routines varies across different DBMS quite significantly, MySQL compared to PostgreSQL seems very weak regarding this aspect. In PostgreSQL we can use several different (procedural) languages to write stored procedures and functions, while in MySQL we only have a subset of SQL/PSM available. Although MySQLs SQL/PSM implementation enhances normal SQL by introducing procedural structures like loops and conditionals, it does still not provide as many conveniences as PostgreSQL due to a lack of choice between different procedural languages (of which the latter has quite a lot, according to their documentation [12]) [EN10, AWT16, SZT12, Gro16].

Nevertheless, procedures can be used to manipulate data during database operations or to simply replace them entirely (i.e. transfer query code and therefore responsibility to the database system). A common stored procedure task could be for example to archive all rows of a table that are older than a certain amount of days. Such a function could look like this:

```
DELIMITER //
CREATE PROCEDURE archive_logs(IN min_age_days INT)
BEGIN
    SET @delete_date = DATE(
        DATE_SUB(NOW(), INTERVAL min_age_days DAY)
    );
```

[12] https://www.postgresql.org/docs/current/static/external-pl.html

```
    START TRANSACTION;
    INSERT INTO `logs_old` (
        SELECT * FROM `logs` WHERE `created_at` <= delete_date
    );
    DELETE FROM `logs` WHERE `created_at` <= delete_date;
    COMMIT;
END //
DELIMITER ;
```

In MySQL we could even use *events* to execute such a stored procedure fully automated on a regular basis [AWT16].

Plugins on the other hand are written in C or C++. They can be used for basically any task as long as they adhere to the (generic) plugin API provided by the MySQL DBMS. Common types of plugins are storage engines, full-text parsers and daemons. But that does not mean they are limited to such use cases, they can also be used for other, almost arbitrary tasks as well. Their advantage is basically that they are not limited to the functionality provided by SQL/PSM, but that they can also invoke functionality of other software (e.g. use third-party libraries). The plugins are actually only limited in terms of how and when they are called by the DBMS (e.g. in what execution phase) [AWT16, GH10].

Very basic examples for MySQL plugins could be a white-list plugin that rejects all SQL statements not matching certain conditions or a plugin that rewrites SQL queries. The latter could be useful if certain (e.g. not so performant) queries are used in legacy software, but no changes within the software are possible [AWT16].

In theory it would also be possible to write a plugin that intercepts SQL queries and fetches data from a source outside the MySQL database if some special conditions are met (e.g. if a special table name was given). For example, such a plugin could:

1. intercept a query like `SELECT 'ip', 'aliases' FROM 'ndb_hosts'`
2. realize that the query fetches from a table with the prefix `ndb_` (for *non-database*)
3. and then use this information to retrieve the requested data from another source, e.g. the systems *hosts* file

Whether such a plugin, written for a MySQL server, makes sense, is definitely questionable because of the overhead being created by the required MySQL server. But looking at the **osquery** project [13], it seems that an SQL query interface for non-database sources is no new idea after all and also very useful concerning the unification and automation of tasks.

Elektra

In Elektra we only have plugins. With plugins one can - very similar to MySQL - basically do everything, starting from very basic tasks like logging of configuration access to very

[13] https://osquery.io/

complex ones like remote notification of dedicated services on configuration changes. There are no restrictions defined by Elektra at all, as long as the plugin interface is sufficient.

There are also some plugins that support the execution of *scripts*, which can be seen as replacement for stored routines like we have them in MySQL. Scripts written in languages like Python or Lua can be altered without having to recompile anything; their execution requires some additional time and resources compared to native plugins though (because they are interpreted at runtime and therefore need a running interpreter).

When deciding to implement a plugin, a well-defined interface has to be used. The most important functions of the interface are `elektraPluginGet()` and `elektraPluginSet()`, which are called by the Elektra framework during configuration fetch and write respectively. The functions are handed a *Key* for potential error information and a *KeySet* to work with. What a plugin has to do with the *KeySet* depends on the type of plugin and the invoked function. A storage plugin for example may need to fill the *KeySet* in `kdbPluginGet()` and write the content of it to a file in `kdbPluginSet()`, while a filter plugin gets handed an already filled one to manipulate (normally also in both directions). Plugins can choose different placements, changing the execution point where they get called by the framework (e.g. `postgetstorage` for an invocation shortly after the *KeySet* was filled by a storage plugin) [Raa16, Raa10].

It is possible and good practice to reuse third party libraries within plugins because it eases maintenance and shifts some parts of the responsibility to other projects [Raa10]. Most of the storage plugins Elektra offers are written on basis of such a third party library implementing a parser and serializer for the configuration format, e.g. the *xmltool* plugin for *xml* files uses the libxml2 library.

An example for a filter plugin is given in section 5.

2.2.8 Resume

Most database management systems are already decades old, went through several iterations and have often multiple major versions behind them [AWT16, Gro16, Aut16]. A lot of effort has been put into optimizing their code base and most of them attempt to offer both convenient as well as performant solutions for all common tasks. The overall package makes it hard to compete against them, both in terms of performance and functionality.

Elektra is a project targeting the unification and centralization of configuration systems. It has never really attempted to be a competitive database (management) system overall, which obviously leads to many disadvantages in almost any imaginable aspect. Still it has its right to exist. Not only because its primary use case is very specific, but also because there are some use cases where the application of a key database makes more sense than deploying a full DBMS (e.g. embedded systems).

CHAPTER 3

Case Study

While the previous chapter consists mostly of theoretical information, we want now try and see what happens if Elektra is used as rather conventional database in a real-world application. For this case study, a REST service that can be used to share configuration snippets and that serves as back-end for a web front-end will be implemented. All crucial data is handled and stored by the back-end with the help of Elektras key database.

Although the project does also include the development of a web front-end, we will focus on the REST service as only this part of the application utilizes Elektra and interacts with the key database. Furthermore the service can also be consumed without a front-end.

3.1 Application Structure

In the following, a small introduction to the developed REST service will be given. The interface defined by the REST service will also be called API frequently [1].

From a technical point of view, the API is split into five parts, each recognizable by a unique prefix of the URI:

- **Root (`/`):** Provides API description and version information.
- **Authentication (`/auth`):** Responsible for authentication of users and requests.
- **User Management (`/user`):** Manages all data related to users (= accounts).
- **Snippet Management (`/database`):** Manages all data related to configuration snippets.
- **Snippet Conversion (`/conversion`):** Offers conversion possibilities for configuration snippets and discloses supported configuration formats.

[1] The service is an API (*Application Programming Interface*) adhering to the principles of REST.

Each of the above has its own controller and several resources to handle, on which users can operate by using the different HTTP methods (`OPTIONS`, `GET`, `POST`, `PUT` and `DELETE`). Further explanation for meaning and functionality of the HTTP methods can be found in [RR07].

The application design makes use of *controller classes* (publicly reachable classes that handle HTTP requests as explained above), *service classes* (private classes that access the key database, convert snippets or perform other logic like filtering) and *model classes* (encapsulate data that belongs together, e.g. multiple keys of a configuration snippet). If we assign the used classes to the three tiers *presentation tier*, *logic tier* and *data tier*, we can speak of using the *3-tier architecture*.

3.1.1 Root Controller

This controller manages a version resource (`GET /version`) that provides version information for the API itself as well as the underlying Elektra installation. A comprehensive description of the API can be found at the root of the API (`GET /`, optionally with the parameter `?raw=true` for a machine-readable version) as well.

3.1.2 Authentication Controller

Although the whole service is meant to be publicly available, we decided to implement a user and permission system to restrict access to certain functions, which allows for administrative features as well. Authenticating users and their requests is the job of the *authentication controller*. To do so, it provides a resource that allows to generate a session token based on the credentials of a stored user, which can then be used in consecutive requests for authentication instead.

3.1.3 User Management Controller

In order for the *authentication* to work, we need users in our system. This is where the *user management* comes into play. It allows to create new users (`POST /user`), to list information for multiple users at once (`GET /user`) and to retrieve more detailed information for a single user (`GET /user/<name>`). It is also capable of manipulating stored users, e.g. performing an update (`PUT /user/<name>` with the updated data in the payload) or a deletion (`DELETE /user/<name>`). Of course it requires some permissions higher than a normal user has to fully utilize all user management functions. Only own information can be retrieved, changed and deleted by normal users themselves.

3.1.4 Snippet Management Controller

Very similar to the *user management*, this controller can be used to manage *configuration snippet entries* in the database. While all the other controllers utilize features of Elektra in some way as well, it is this part that is the most important. Not only because it handles everything directly related to the topic of the service, but also because the other

application parts often reuse functionality already used or implemented by this one. It basically supports exactly the same functions than the *user management controller*, but on the `/database` instead of the `/user` resource.

3.1.5 Snippet Conversion Controller

An additional feature that can be seen as show case for Elektra is the *snippet conversion*. Internally it reuses functionality of the *snippet management*, but with the difference that converted snippets get never stored in the key database at all. Its sole purpose is to convert configuration snippets between different configuration formats. It also can be used to list configuration formats supported by the API.

3.2 Data Structure in the Key Database

When trying to implement an API with a hierarchical database, defining a structure for the data is important. Other than relational databases, in Elektra it is not possible to define tables with fixed columns and types. Storing data in an appropriate format and schema is the responsibility of the application, which in this case is the REST service.

The data structure itself is very simple and best explained by looking at the data structure used in the application. There we use models for our data (classes) which have multiple attributes. These models are then stored in the key database, using several keys per model. All data that belongs to a model shares the same key prefix, which depends on one of the models properties, e.g. the name when speaking of a user. Additionally the keys are prefixed with an application and model specific identifier. For example: *<application>/<model>/<unique-identifier>/<model-data>*

For user *Jeff* it looks like this (root key has the password hash as value):

```
/myapp/users/Jeff 3fc48...313cb
/myapp/users/Jeff/email jeff@example.com
/myapp/users/Jeff/rank 10
/myapp/users/Jeff/createdat 1478250143
```

It is also possible to store data as meta data [2] of a key if storing it as descendant of the key is inappropriate or impossible. This is the case for the snippet model as the actual configuration snippet is stored in the sub-keys. Other attributes like the snippet title and description need therefore to be stored as meta data. Think of:

```
/myapp/snippets/org/app/v1/example
/myapp/snippets/org/app/v1/example/setting1 on
/myapp/snippets/org/app/v1/example/setting2 off
```

[2] **Meta data** or **meta keys** are keys that belong to another key. They are often used to describe a key (e.g. its data format, like *string*, *integer*, *boolean*, ...) or to store comments.

```
[meta for /myapp/snippets/org/app/v1/example]
  title "Example for my app"
  description "Even with a helpful description"
```

If we would try to store the title or the description below the normal key `/myapp/snippets/org/app/v1/example/title`, it could be overwritten by parts of the configuration snippet.

3.2.1 Data Validation

Even though we could utilize Elektras *specification* feature to ensure that no keys with wrong (i.e. inappropriate, not wanted or of the wrong type) data get stored in the key database [Raa16], the REST service itself has also to do (at least some) validation to prevent runtime issues. It is therefore more convenient to validate all inputs within the API itself instead of handing the job to the key database and mounted plugins.

Of course the developed web front-end does also validate user inputs before they are sent to the back-end. But as we cannot expect to only receive requests being sent by the front-end and the data does only get stored in the back-end, the validation in the latter is actually the only one that counts towards security. The validation done in the front-end serves more the purpose of usability.

3.3 Problems

Throughout the development of the REST service, several problems came up. Of course most of them were rather small and not noteworthy at all, but a few, especially the ones related to Elektra, were interesting enough to be mentioned here.

3.3.1 Inconsistent Plugin Behavior

Something that was noticeable quite early during development already is the inconsistent behavior of storage plugins. Some of the plugins store absolute key paths in their configuration files, while other plugins (correctly) use relative paths. Although this inconsistency has been noticed a long time ago already, not many plugins had been adapted since then yet.

When creating backups for example, this behavior does not matter, as it is the goal of a backup to restore an earlier state anyway. Within the developed REST service it introduces a problem though as the service uses a special path prefix for all its keys (which creates some sort of virtual database in the (already virtual) key database). If plugins do now use absolute key paths during export of configuration snippets, the created configuration files cannot be re-imported by users in their installation of Elektra because the desired import path does not match the export path (which quite certainly will be the case for almost every application).

What this means is that if we have the following keys in our database

```
user/sw/apps/myapp/version
user/sw/apps/myapp/version/major 1
user/sw/apps/myapp/version/minor 0
user/sw/apps/myapp/version/micro 5
```

and we attempt to export everything below the path `user/sw/apps/myapp` in the INI format, we would expect to receive something like

```
[version]
major = 1
minor = 0
micro = 5
```

while the output would be

```
[user/sw/apps/myapp/version]
major = 1
minor = 0
micro = 5
```

for a broken INI storage plugin. An import of the wrong snippet for a different path, e.g. `user/sw/apps/otherapp`, would therefore fail.

3.3.2 Supported Format of Storage Plugins

Another problem was to retrieve the format supported by a storage plugin. It was and is quite easy to find a suitable plugin if a certain configuration format (xml, ini, json, ...) is given, because either there is a plugin named after the format (e.g. ini plugin) or a plugin contains the format in its provider list [3]. But it was not possible to determine the reverse - the format if a plugin was given.

The reason for that is that the format information was only given implicitly as part of the provider list, which can also contain unrelated information. It was impossible to know which value of the list represents the format. Only by adding a prefix (like `storage/`) to the provider value in question, it was possible to lookup the format consistently.

For the yajl plugin, which implements the JSON configuration format, this means a change of the contract from

```
infos/provides = storage json
```

to

```
infos/provides = storage/json
```

which gets expanded during build to

```
infos/provides = storage/json storage json
```

[3] The provider list of a plugin lists features the plugin offers. A storage plugin for example offers *storage*, while a resolver plugin offers *resolver*. Certain provider values can be interpreted as *type* of the plugin. This list allows for easy builds and enables developers to look for available plugins by their type.

This way, all necessary information to lookup the format is available and we still kept backwards-compatibility for all other lookups and functions.

3.3.3 Plugin Variants

The last larger issue we faced was the retrieval of plugin variants (i.e. differently configured instances of a plugin [4]). As we already know, plugins may receive a configuration, of which each parameter can be either mandatory or optional. Normally only the mandatory configuration parameters are of major interest though as we cannot use a plugin without them (which does not apply for the optional parameters of course).

Before we added an option that allowed us to generate configuration variants for plugins, it was not possible to automatically retrieve sufficient information to configure them. Because it seems natural to let the plugins decide themselves what is best for them, it is now possible to add an (optional) function to plugins that does exactly that - generate variants for the plugins.

Of course it is not strictly defined what this function has to generate (if it was, we would not have to let the plugin generate the variants anyway). This means that the function does not just generate configuration permutations (which would also be impossible in some cases, imagine a plugin with five configuration parameters of which each may have a value between 1 and 1000; we cannot generate 5^{1000} permutations). Instead the function can generate whatever it wants, starting from an empty set of configurations, via a subset of all possible configurations to the full set of all permutations (if possible).

The new function named `genconf()` (for *generate configuration*) may for example return the following for a plugin that prints one key-value pair per line:

```
system/space
system/space/config
system/space/config/format = % %
system/dollar
system/dollar/config
system/dollar/config/format = % $ %
```

As we can see, each configuration variant has besides its set of configuration parameters also a unique name (here *space* and *dollar* indicating the separator). This allows us to disable and override them via the system configuration stored in the global key database (assuming the plugin is called *sep*):

```
system/elektra/plugins/sep/variants/space/disable = 1
system/elektra/plugins/sep/variants/dollar/override = 1
system/elektra/plugins/sep/variants/dollar/config
system/elektra/plugins/sep/variants/dollar/config/format = % $$$ %
```

[4] Plugins are configured with the help of a *KeySet*, containing one key per parameter. So a plugin variant is one *KeySet* filled with a valid plugin configuration.

The main reason we required this feature was the *augeas* plugin, which supports a large number of configuration formats and should therefore definitely be part of the snippet sharing service. The plugin takes one mandatory parameter, which defines the Augeas lens of the configuration format to use. With this feature, the *augeas* plugin does now return one plugin variant per available lens.

It was also planned to support the overriding of plugin contracts (e.g. override the *format* a plugin offers for a specific plugin variant). But as currently the technical preconditions are not met to implement such a feature, this plan was discontinued for the moment.

3.4 Lessons Learned

Besides aforementioned problems and their solutions, we also learned that planning and describing a REST API is not as trivial as initially thought. Not only is the selection of the right language and tools important, but also the API design itself should be consistent and properly considered (e.g. to avoid redundancies and to spare the server from too high loads). This consistency should also be ensured across different tools of the same project, which we did by using the same conventions as another bachelor project (REST API for cluster management by Daniel Bugl).

We also learned that the elektrification of existing applications can be difficult. The REST service, which was written based on CppCMS (a high-performance C++ web framework), required application integration due to its nature of being an Elektra tool (more from an ideological than a technical standpoint). As full integration of Elektra in CppCMS itself was impossible, a transformation of Elektra *Keys* to CppCMS' own implementation of JSON was written. This transformation came with some problems, which could, in the end, be solved by using a configuration specification. But the attempt to write the transformation showed us that it is easier to use the data structures that come with Elektra instead of making an own implementation.

Finally, even if it sounds rather trivial in the description given in section 3.2, the storage of structured data comes with some pitfalls. The retrieval of stored objects for example involves bundling *Keys* that belong together into proper objects. This process does require a lot of comparisons, which is very time consuming and does therefore also not scale very well. Consequently it makes sense to cache bundled objects instead of bundling them together for each request over and over again.

CHAPTER 4

Evaluation

Because of the way we are using Elektra in this project, namely as database for structured data, it seems quite natural to benchmark the developed service against a solution based on a relational database system, which will be MySQL in our case. The REST part of the service is irrelevant for the benchmark as it would be the same for any data storage solution, therefore we only benchmark specific functions of the service classes that access the data store directly. This allows for an easy one-to-one comparison between Elektra and MySQL as well as each type of benchmark (e.g. lookup by the unique identifier).

For the benchmarks we will focus on the read performance of the systems. Although the implemented service does support inserts, updates and deletes (and so does MySQL), reads will occur much more frequently, which makes it a more interesting operation to measure. Reads are also the only functions that depend on the cache of the REST service; inserts, updates and deletes open a connection to the key database, which is quite time consuming in its current implementation. A possible solution for this issue, which would lead to a compromise regarding execution times of different database operations, can be found in the Appendices.

4.1 Test Setup

All benchmarks were written in C++ with the MySQL benchmark application being based on the official MySQL C++ Connector binding. Although execution times can be measured even in nano seconds (if the compiler supports it), only micro seconds were used for the evaluation because of the high numbers in larger scale scenarios. To improve the quality of the evaluation, always the median time of three benchmark executions was taken. The data used for both benchmarked systems is equal and stems from the exact same generator.

We will have a look at the following common operations:

- Lookup of an entry by its unique key
- Lookup of entries by only some part of their key
- Lookup of entries by different meta information

For each type of benchmark the data is freshly inserted into the data store previous to measuring the execution time. The MySQL database uses three relations (tables) for *users*, *snippets* and *tags*, whereas one *user* can have multiple *snippets* and one *snippet* multiple *tags*. The Elektra key database uses two mount points (with the *dump* storage plugin [1]) for *users* and *snippets* (*tags* are part of snippet entries), while entries of each type are stored within multiple keys below the corresponding mount point as described in the previous chapter.

Depending on the tested data store, indexes can be applied to the created database or the data store can be read into the in-memory cache before executing the benchmark itself. After that, the tested lookup function has to fill models for *Snippet Entries* as used by the service classes (especially relevant for the MySQL benchmark). This ensures proper comparability.

Which entries have to be searched by the benchmarks is calculated from the amount of test data being inserted. The test data is sorted and contains continuous numbers (e.g. for 500 entries, we have numbers 1 to 500). This allows to search in any area of the data. For this benchmarks we used data from the center of the test data (i.e. `number-of-entries / 2`).

Although the benchmarks support the lookup of multiple entries at once, we always used data that allowed to look for only one entry as we did not want to measure the filling of models with data, which is the same for both systems. The numbers presented by the benchmarks are therefore always for lookups with one single entry as result.

The benchmark results were collected on a virtual machine with 4 assigned CPU cores and 4096 MB of RAM. The VM hosting a Debian Jessie installation was operated using Oracles VirtualBox version 5.1.4, which was running on a desktop PC with an Intel Xeon E3-1231 v3, 8192 MB of RAM and a Windows 7 64bit installation. The virtual machine files were stored on a SanDisk Ultra II SSD with 480 GB (60% occupied).

For MySQL the version 5.7 was used because v5.5 (the Debian Jessie standard version at the time of this writing) does not support full-text indexes for the InnoDB engine. The used Elektra version for the benchmarks was 0.8.19 with an almost full installation of plugins, bindings and tools. Debian itself had version 8.6 with the Linux kernel 3.16.0-4-amd64.

[1] During a pre-benchmark comparison of storage plugins, the *dump* plugin delivered best speeds, although the *ni* plugin is known for being more performant normally. This is most likely because of the high amount of meta keys being used in the service data. The *ni* plugin stores normal keys and meta keys separately, while the *dump* plugin stores them side by side. Attaching meta keys during `kdbGet()` is therefore more expensive because an additional `ksLookup()` has to be performed.

Everything used for the following benchmarks can be found in the Elektra repository (commit a6d587d and below).

4.2 Results

4.2.1 Lookup by unique Key

Measuring performance for this operation is interesting because of the way Elektra stores its data and MySQL retrieves records by a unique index [2]. While Elektra stores its data directly besides the key, MySQL uses a special index which tells the DBMS where the actual data is located. This allows MySQL to reduce the search space with Elektra at the same time having to parse the whole mount point where the key is stored in.

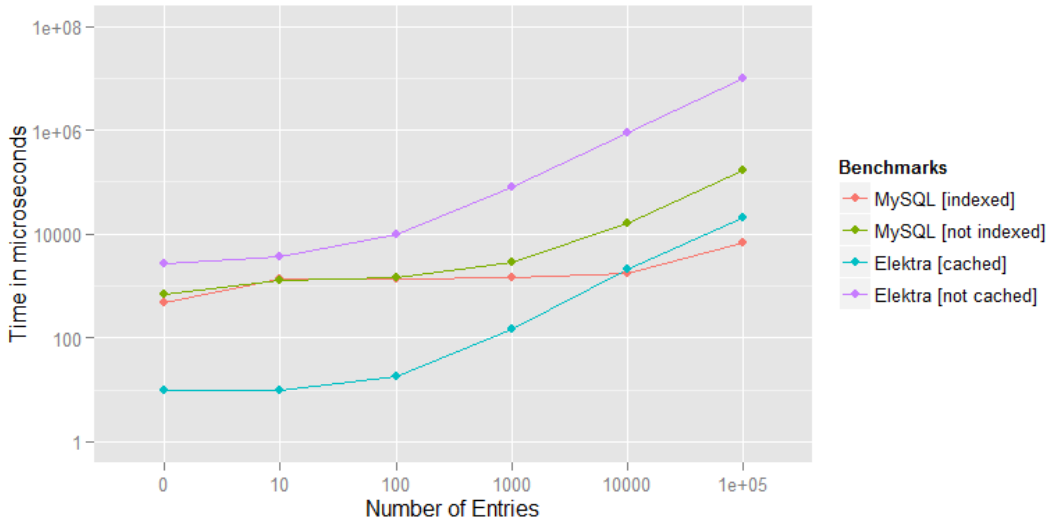

Figure 4.1: Benchmark: Lookup of single entry by its unique key

(*Note: axes are scaled logarithmically*)

As we can see in Figure 4.1, the cached variant of the Elektra solution is incredible fast. This is not surprising at all considering that all data is stored in-memory and only the right entry has to be found by searching the cache. Without caching, the Elektra solution is very slow though - a lot slower than the MySQL database. The reason for this is that several steps are necessary until the data stored in the key database is available:

1. Resolving the mount point (accessing and parsing the configuration file where mount point information is stored).
2. Accessing and parsing the configuration file where the entry we are looking for is stored in.

[2] A unique index was used for the *key* of snippet entries in favor of a primary key, because the use of a numeric primary key (*ID*) makes more sense when dealing with foreign keys (cf. *tags* table).

3. Transforming all Elektra keys into *Entry* objects being used by the service.
4. Searching for the right entry.

Obviously the time it takes to parse the configuration file where all entries are stored increases with the size of the file (i.e. the number of stored entries). The time does also depend on the used storage plugin; a custom plugin which operates on a well-defined, minimalistic structure may be (a lot) faster. The only step in above list that remains pretty much constant is the resolution of the mount point.

With the current implementation of the REST service (see 4.1 for commit details) it is not possible to disable the cache entirely. The tests differ only in the time when the cache is loaded. That means also the lookup of a single entry will force the application to load the whole cache, even if the configuration snippets are stored within different mount points (to reduce the size of individual mount points). The service itself will fill the cache during application start, so that users do not experience delays.

When looking at the chart precisely, it can be seen that MySQL is overtaking the key database in the end. This is not really surprising, as the unique index (B-Tree) used by MySQL improves speed a lot. While Elektra (and its storage plugins) will struggle with higher number of records, MySQL was designed for such amounts of data, which makes it handle them easily.

Unfortunately it was not possible to execute the benchmarks for one million and more entries. The MySQL benchmark aborted the insert of the data (most likely because of the transaction size and that even with an increased limit), while the Elektra benchmark had to keep all data for the mountpoints in-memory, which is impossible for this amount of entries. By allowing the operating system to load the data into the swap, the benchmarks would be falsified though.

4.2.2 Lookup by Part of a Key

It has already been noted that each *Snippet Entry* is stored below a unique key which consists of several parts (see 3.2). Because of that it is possible to look for all keys that start with a certain string (*part of a key*). This way we can, for example, look for *<organization>* to find all snippets of an organization, but we cannot look for *<application>* without providing an *<organization>* when utilizing the key comparison function. With additional filter functions also that is possible though.

When realizing this feature with MySQL, it is even simpler because each part of the key is stored separately and therefore direct access to the *application* name is also possible (despite that there might be multiple applications with the same name of different *organizations*). So the comparison basically comes down to `entry.getName().startsWith("myorganization")` vs. `SELECT * FROM 'snippets' WHERE 'organization' LIKE 'myorganization'`).

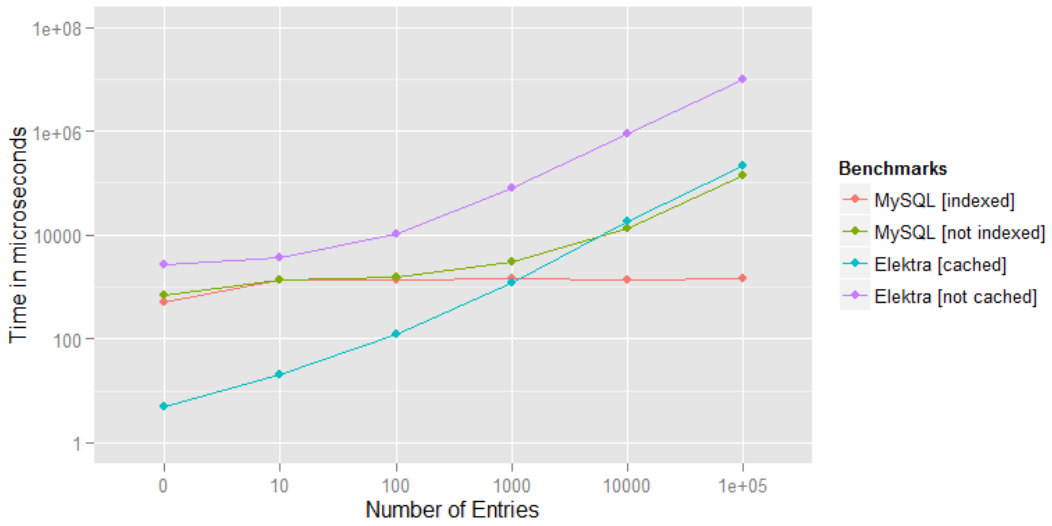

Figure 4.3: Benchmark: Lookup of entries by part of their key

(*Note: axes are scaled logarithmically*)

This does not mean that the solution based on Elektra is faster than a MySQL database, as can be seen in Figure 4.3. The indexes used for the key name parts are really powerful and allow for very fast lookups. They even seem to provide constant lookup time for the different benchmark sizes.

4.2.3 Lookup by the Authors Name

The reason for this being an interesting benchmark is that the authors name gets stored in a separate relation (table) in the MySQL solution and only the users ID (primary key) is referenced in the snippet entry, which obviously reduces the database size. On the contrary, the solution based on Elektra stores the authors name as part of the entries meta data, which leads to data duplication, but also to easier access at the same time.

In Figure 4.5 it can be seen that again the MySQL solution seems to be more performant when the amount of data increases. The reason for that could be that the authors name has only to be found once (as it does exist only once in the database). After doing so, the ID of the corresponding user can be used to find snippet entries with a equivalent foreign key reference (i.e. an integer comparison), which is a lot cheaper (from the time point of view) than comparing a string (the name) over and over again.

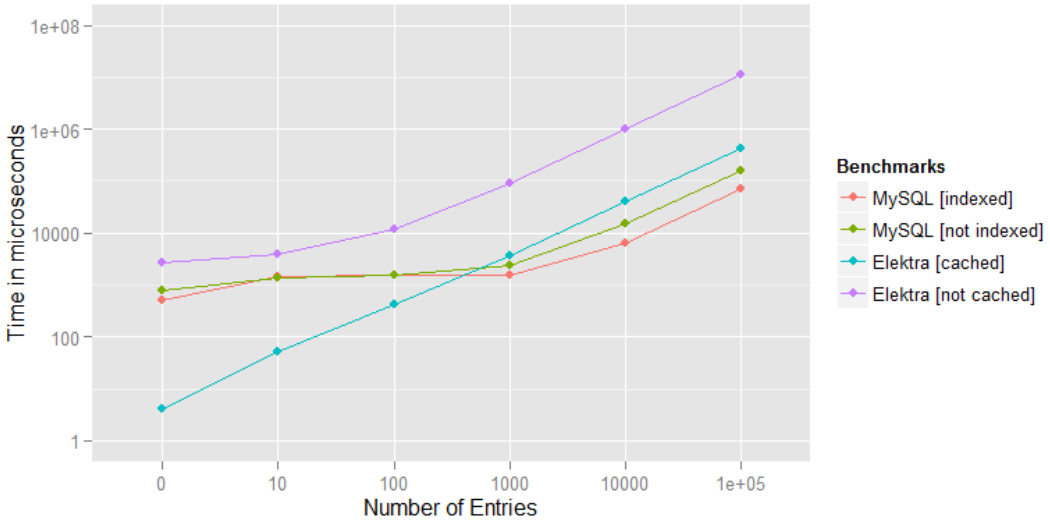

Figure 4.5: Benchmark: Lookup of entries by their author

(*Note: axes are scaled logarithmically*)

4.2.4 Lookup by a Tag

Also a common way to look for entries is searching by a tag. In Elektra the tags are stored in a space-separated list as meta data of an entry [3]. As soon as the entries are loaded in-memory, the tags are available as vector and can be iterated. In MySQL the tags are stored in a separate table (1:n relation).

Figure 4.7 shows us that Elektra is also in this scenario slower compared to MySQL. Interesting is also that there is an index applied to the *tags* table, which does not seem to have any effect at all.

4.2.5 Lookup by Part of Description

As last benchmark we tested searching through the description of entries, which can be considered a full-text search problem. For the benchmark, the searched part of the description is always the last word of the target description, forcing the solutions to search through the whole description for each entry (if not optimized).

In Figure 4.9 we see a quite similar result compared to the previous two benchmarks (cf. 4.5 and 4.7). The MySQL indexes do not seem to have any real effect and Elektra gets worse when the database size increases.

Speaking of size, the test data for the very last benchmark, which searched through descriptions of 100.000 thousand entries of one user, occupied 133.408 KB of space within

[3] Meta data is stored in meta keys, which are basically normal keys belonging to another key.

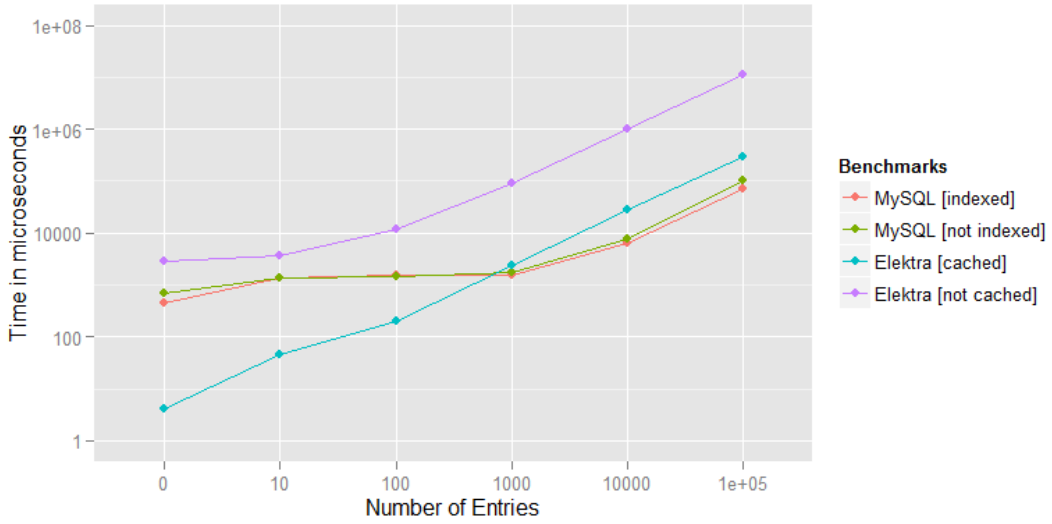

Figure 4.7: Benchmark: Lookup of entries by a tag

(*Note: axes are scaled logarithmically*)

the MySQL database system (with indexes; 26.704 KB without) and 202.052 KB in the Elektra key database (using the *dump* storage plugin). The reason for the difference is that every single attribute of the entries in the key database is stored along with its full key, wasting a lot of space. This is special behavior of the *dump* storage plugin though; other plugins like the *ni* plugin (*ini* format) occupy 162.246 KB for the exact same data and a storage plugin optimized for the used data structure may consume even less disk space than the MySQL database as already hinted in section 2.2.5.

4.3 Discussion

Based on our benchmarks we can conclude that a MySQL database performs (significantly) better than Elektras key database when it comes to lookups, especially with larger data sets. Furthermore, we learned that indexes in the MySQL solution do not seem to have as much of an impact as the in-memory cache of the Elektra solution - at least not for the size of our test sets. We also ascertained that in MySQL lookups of entries with the help of a unique index are faster than lookups with other types of indexes, such as full-text indexes (cf. lookup 4.2.1 and 4.2.2 vs. 4.2.5).

Another point that caught our attention is that the last three benchmarks (cf. 4.2.3, 4.2.4 and 4.2.5) produce quite similar results (especially the Elektra benchmarks). That the uncached benchmarks deliver this result is not really surprising considering that parsing of configuration files takes a lot more time than iterating and searching objects. But also the cached benchmarks are quite similar, what leads us to the assumption that not

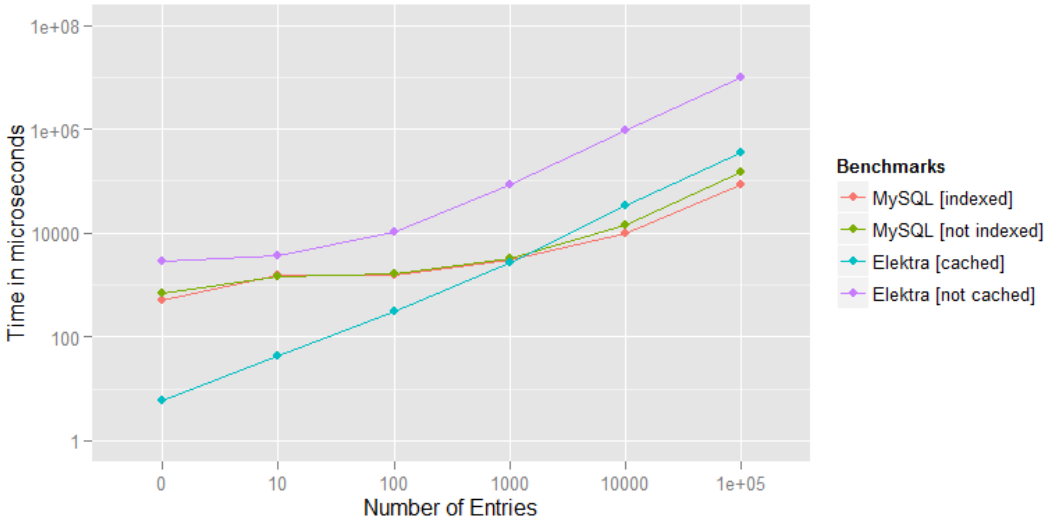

Figure 4.9: Benchmark: Lookup of entries by part of their description

(*Note: axes are scaled logarithmically*)

searching different attributes of entries (e.g. the description) takes most time, but rather iterating the cache does.

CHAPTER 5

Conclusion and Future Work

After a short introduction to Elektra, we have learned important differences and similarities of Elektra and MySQL in this paper, starting from information retrieval, via user management and data access through to stored routines and plugins. We have also learned that developing for and with Elektra does sometimes come with difficulties, which we could demonstrate using the example of our case study. Finally, we benchmarked our developed case study subject against a solution based on MySQL to get an idea of Elektras capabilities.

The previous chapters showed us that Elektra, despite being meant to store simple configuration parameters in a global, hierarchical key database, is also capable of providing decent performance when it comes to storing structured data for a web service. We have seen that Elektra will not be a market leader in this area because it performs worse than other solutions in larger-scale scenarios and that without its cache, Elektra would furthermore suffer from lookup times that were inappropriate and impractical for a web application.

But based on this knowledge, we assume that improvements could be made to the developed REST service (and Elektra itself) in several areas. The first and most straight forward approach would be to use both benchmarked systems together - MySQL for storage and Elektra for data manipulation as well as data transformation [1]. Not only would it give us better performance regarding the data store, but also shift some important responsibilities to the database management system. A MySQL or SQLite plugin for Elektra, supporting the storage of structured data, could also serve as alternative.

Another improvement, in case MySQL as data store is no option, could be to use dynamically created mount points for different parts of the stored data (e.g. to split entries by their *organization* or *application* into different mount points). This alone would

[1] Currently *data transformation* only means *configuration conversion*, but *script-based transformation* of configuration snippets is another idea for the future.

not be enough though, a new cache implementation (see 5 for a possible solution) would be required as well. By reducing the size of the mount points, faster lookups would be possible as the time required to parse a single mount point would be reduced significantly. Rarely accessed configuration snippets could also be kept out of the cache entirely this way.

The last possible improvement we want to mention is an additional command for the `kdb` tool (or an own CLI tool) that can send configuration snippets taken from the local key database to the REST service. It would enable end-users to bypass the web front-end and to replace complex cURL requests. In other words it would allow for easier sharing of configuration snippets, especially on systems without a graphical user interface.

To conclude, we can say that Elektra, in its current state, is sufficient to run a REST service that allows for sharing, converting and downloading of configuration snippets. Scalability is definitely an issue which should be addressed in future work, but first the service should prove its strengths and weaknesses to allow for further improvements.

Quite certainly the Elektra project itself will also learn from the service and its users. But not only about errors and mistakes that were made in single areas or modules, but also about user expectations and possible improvements, which will hopefully influence Elektras road-map in the future.

Appendices

The Cachefilter Plugin

Key Database Returns too much or less Data

Because of the way Elektra is implemented, accessing the key database can return more keys than actually requested (e.g. siblings of the lookup-key instead of children only). Elektra will return all keys stored below a mount point if one of its keys is requested. For example if we use `kdbGet()` with the parent key `/myapp/primary`, we would expect to only retrieve the following keys

```
/myapp/primary/version
/myapp/primary/version/major
/myapp/primary/version/minor
```

while the key database would also return

```
/myapp/secondary/version
/myapp/secondary/version/major
/myapp/secondary/version/minor
```

if they are stored within the same mount point (i.e. file).

For successive calls to `kdbGet()` with the same parent key (or a descendant of it), the key database will return only changed keys. In a long running application that does not want to manage multiple *KeySets* on its own, this may be inappropriate.

It is implemented this way because storage plugins can only operate on whole files, not on parts of them (which has also to do with the structure of various configuration formats). Storage plugins should also not be responsible for caching data. They should only be able to parse the content of a file (a configuration snippet) into a *KeySet* and serialize a *KeySet* back into the same configuration format if necessary. So they return everything stored within a mount point, no matter what exact parent key was used for `kdbGet()`.

The Solution

To fix this issue, the *cachefilter* plugin should be developed. The goal is to match user expectations when it comes to dealing with the key database. To do so, the plugin should cache all retrieved keys and remove unwanted keys (i.e. keys not starting with the parent

key) during `kdbGet()` from the result. It should also add keys of the cache to the result for recurring calls to `kdbGet()`. Cached keys should then be added back to the *KeySet* during `kdbSet()`, which should ensure that no keys get lost.

A first draft of the plugin that lacks most functionality can be found in the folder `/src/plugins/cachefilter` of the Elektra repository. The development of the plugin was discontinued for the moment because the Elektra framework itself lacks the necessary functionalities to deal with global plugins as it would be required by the cachefilter plugin. The plugin requires state-awareness (i.e. it needs to know in which state it was called) and especially invocation in the right state.

Impact on the REST Service

For the REST service it would mean that we can replace the internal cache implementation by the plugin, which would speed up inserts, updates and deletes significantly. Read operations on the other hand would probably suffer from the new cache implementation if the old cache was removed, because all keys belonging to an entry would have to be bundled together for each request once again. A mix of both cache implementations could therefore be the best solution after all.

List of Figures

Bibliography

[Aah03] Aahz. Typing: Strong vs. weak, static vs. dynamic, 2003. http://www.artima.com/weblogs/viewpost.jsp?thread=7590.

[Aut16] The SQLite Authors. Sqlite 3.15.2 documentation, 2016. https://www.sqlite.org/docs.html.

[AWT16] David Axmark, Michael 'Monty' Widenius, and MySQL Documentation Team. Mysql server documentation (version 5.7), 2016. https://dev.mysql.com/doc/refman/5.7/en/.

[BRA12] Alexandru Boicea, Florin Radulescu, and Laura-Ioana Agapin. Mongodb vs oracle - database comparison, 2012. Presented at the Emerging Intelligent Data and Web Technologies (EIDWT), 2012 Third International Conference.

[EN10] Ramez Elmasri and Shamkant Navathe. *Fundamentals of Database Systems*. Addison-Wesley Publishing Company, sixth edition, 2010.

[GH10] Sergei Golubchik and Andrew Hutchings. *MySQL 5.1 Plugin Development*. Packt Publishing, 2010.

[Gro16] The PostgreSQL Global Development Group. Postgresql 9.6.1 documentation, 2016. https://www.postgresql.org/docs/9.6/static/.

[Iss14] Luke P. Issac. Sql vs nosql database differences explained with few example db, January 2014. http://www.thegeekstuff.com/2014/01/sql-vs-nosql-db/?utm_source=tuicool.

[Pak10] Matti Paksula. Persisting objects in redis key-value database. 2010. Available at https://www.cs.helsinki.fi/u/paksula/misc/redis.pdf.

[Pow06] Gavin Powell. *Beginning Database Design*. Wiley Publishing, Inc., 2006.

[Raa10] Markus Raab. A modular approach to configuration storage, September 2010. Available at http://www.libelektra.org/ftp/elektra/thesis.pdf.

[Raa16] Markus Raab. Elektra: universal framework to access configuration parameters. *The Journal of Open Source Software*, 1(8), dec 2016.

[RR07] Leonard Richardson and Sam Ruby. *RESTful Web Services*. O'Reilly, first edition, 2007.

[Str] Christof Strauch. Nosql databases. Found at `http://www.christof-strauch.de/nosqldbs.pdf`.

[SZT12] Baron Schwartz, Peter Zaitsev, and Vadim Tkachenko. *High Performance MySQL*. O'Reilly, third edition, 2012.

[Xia11] Yang Xiaojie. Analysis of dbms: Mysql vs postgresql. 2011. Available at `https://www.theseus.fi/bitstream/handle/10024/27471/Final_Thesis_Xiaojie_Yang.pdf`.