ESQLite

a relational database solution for JSON data with applications in mobile computing

Hermkens, E.G.W.

Award date:
2016

Technische Universiteit
Eindhoven
University of Technology

Department of Mathematics and Computer Science

ESQLite: A relational database solution for JSON data with applications in mobile computing

Master Thesis

E.G.W. Hermkens

Supervisor:
dr. G.H.L Fletcher

Committee:
dr. G.H.L Fletcher
dr. A.M. Wilbik
dr. M. de Leoni

1.0

Eindhoven, March 2016

Abstract

This thesis concerns the problem of working with large amounts of partially non-relational or non-structured data on lightweight clients, such as mobile devices. The amount of devices and the data generated from them grows at an unprecedented speed, propelling the development of new database systems. Whereas the vast majority of data transfers on the web is standardized through for instance the *JavaScript Object Notation (JSON)*), storage and querying techniques for this type of data are still very limited and do not meet today's needs. Although a lot of research is done in this field, when looking at (lightweight) client solutions it leaves much to be desired.

We introduce and develop ESQLite, an extension on top of the relational SQLite database which is able to fill the void in JSON data handling. ESQLite can store and foremost query non-relational data represented as JSON. ESQLite offers both efficient storage, as well as an XPath like query language extension to perform path queries. Before storing, the JSON is shredded at insertion into key-value pairs, along their tree-structure meta data. When disk space is of concern, the shredding can also be done temporary at query time. This approach enables us to implement efficient path querying on JSON properties. Reconstructing the JSON can be done with a tree walk or self-join algorithm. Finally, ESQLite is able to extract relational parts from the JSON, and handle these as relational data, further boosting performance.
Special effort is put into developing an easy to use, backwards compatible and open source available library, in the form of an Android specific wrapper. Android is the platform of choice since it is open source, has more than one billion users, built-in SQLite support and a lot of data intensive applications.

Using two datasets, of one which is a real world medication database, the performance of ESQLite is benchmarked. A set of queries and dimensions of interest is chosen, with speed and disk space as the main focus. Whereas ESQLite insertion queries are on average an order of magnitude slower than normal SQLite, it can handle selection XPath queries on JSON data as fast as normal SQL SELECT queries. Therefore there is a trade off between insertion and query speed. Several optimizations and variations are discussed, boosting the performance on average 8% to 50%.

While ESQLite as an open source library may be a useful tool for those interesting in working and querying JSON data, it may also form a starting point for further research. A lot remains to be done; for instance improvements in determining and extracting the relational base of the JSON, efficient reconstruction and extension of the query language.

Acknowledgments

This report concludes the results of my graduation project for the master program of Business Information Systems at the department of Computer Science at the Eindhoven University of Technology. From company experience I was really eager to work on a project which has something to do with document stores and/or querying non-relational data, this brought me to dr. Fletcher. It took quite some wanderings to come to the actual thesis subject, and during the project converging turned out to be the hardest part. In retrospective the subject completely suited my interests, and the practical twist enabled me to get fully motivated.

First of all I would like to thank my graduation supervisor dr. George Fletcher for guiding and helping me during my project. Your constructive feedback, critical questions and open approach to new ideas gave me confidence and stimulated me during my project. The collaboration during my master project has been a pleasant and a great learning experience. I would also like to thank dr. A.M. Wilbik and dr. M.D. Leoni for joining my assessment committee.

Last, but not least I would like to thank my parents, brother and sister, friends and Lotte for their encouragement, interest and support. I enjoyed my time as a student both intellectually as socially, I had te opportunity to learn a lot. From time to time I struggled with my intrinsic motivation in relation to other alluring activities, and would never have been at this point without the support of my parents, Lotte and my family.

Edwin Hermkens
March 2016

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

Nowadays (web) applications are highly data intensive and often include a combination of relational, non relational, unstructured, schema-less or sparse data. A great effort of research is put into the storage and querying of these types of data, resulting in the development of a wide landscape of so called 'Not only SQL' (NoSQL) databases. Examples of such database systems are; key-value stores or graph databases. In Chapter 2.1 we discuss NoSQL in more detail. Interesting to observe is that both server-side storage as (web) transfer of data has evolved to handling this types of data, but lightweight client side database systems are less developed in this area. Since less effort is put into enabling lightweight mobile clients to cope with this type and size of data, a gap falls between the capabilities of server- and client-side databases. This can be partly explained by the fact that client-side applications often request only small subsets of preprocessed and pre-queried data from the server, the data is then used and the device does not hold a copy of the dataset in non-volatile memory. For applications where this not possible, due to for instance security, privacy or off-line usage, this leaves us with a complex storage and query problem. This approach does not leverage the possibilities of database systems to their full potential.

When looking from a technological perspective, the de facto standard for storing data on mobile devices is SQLite. SQLite is a relational database system based on the programming language C. It is developed as a lightweight version of MySQL [2], able of handling relational data in an ACID transactional way. SQLite tables have fixed schema's describing the data. The de facto standard for transferring data on the web is *Javascript Object notation* (JSON), which contrary to what the name may suggest, is language-independent. It is a text format which contains one or more objects of attribute-value pairs, possibly in arrays. JSON is schema-less and highly flexible.

We observe that there is a mismatch between the technologies for storing these types of data and how its transferred and queried. Furthermore the data is often a combination of relational and non relational data, making it hard to have good performance on both types within the same system. For the above reasons the few existing NoSQL solutions for mobile devices are usually unsatisfactory. This motivates to design an approach and implementation which is introduced as the Android wrapper library ESQLite.

The remainder of this Chapter is organized as follows. Section 1.2 grasps the problem as described into a problem statement after which we further materialize the statement into three research questions. These questions will give guidance throughout the research. In Section 1.1 aims to make the problem more tangible by elaborating on a real world example. Afterwards we will outline the main contributions in Section 1.3 and finally Section 1.4 will briefly elaborate on the thesis outline.

1.1 Real world example

The problem at hand is derived from a problem we came across in a company setting. The company operates in the healthcare domain of patient drug management. For that reason the company handles a large medication database of prescription medication available in the Netherlands. This case in particular is interesting since the data is both hierarchical (relational) as partly not relational. Whereas from a patient perspective, medication can be seen as simple items, for instance *that red round pill*, the healthcare professional uses a more hierarchical approach. Medication can be divided into substances, products and articles, each with their own characteristics. A patient who is prescribed the drug item *Finimal 500MG* is actually using the brand *Finimal* of the substance *Paracetamol* with a dose of *500mG*. For each substance a large variety of brands and doses are available. Since *Finimal* comes in different forms, sizes and quantities the professional also needs notion on a item level such as *Finimal 500MG 30 pieces Oral*. The actual hierarchical structure is even far more complex, containing logistic information such as barcodes, clinical rules, usage- and pricing information and so on. In Figure 1.1 the complete structure as handled is shown.

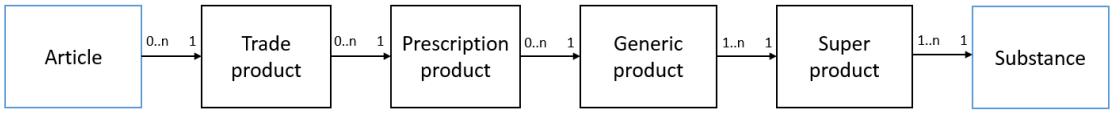


Figure 1.1: Overview of medicine data Hierarchy

Besides this data, the company also stores other media oriented (meta)data such as information leaflets, *bijsluiters*, instruction-videos and more. All of which are related to one or more items, products or substances. This results in an complex stack of data which can be relational, non-relational or a combination of both. To clarify, in Figure 1.2 an example of semi unstructured data for the drug *Ibuprofen* is represented as a JSON document.

When storing the above data in a normal relational database such as SQLite one would have to use a lot of tables. For all levels as shown in Figure 1.1 as well as all other subtree objects, a table has to be created containing columns for every property. This will introduce a lot of sparse data and a demoralized database, introducing foreign keys, joins and thus poor performance. Because the schema-less properties of some of the subtrees would introduce a column for each attribute, this will result in a very sparse table.

For example, since the brand name is stored at the 'trade product' level, whereas the substance name is stored at the 'substance' level, Figure 1.1 shows that selecting the combination of brand and substance name would require at least five joins. Keeping in mind that there are over 100.000 regularly updated medicines available, which have to be stored to a regular phone. One can imagine that a normal relational database would not satisfy storage and query needs for this type of data, the above therefore illustrates the need of a renewed form of JSON data handling. Please mention that the above case is partly relational data, there is also a vast amount of completely non-relational data use cases available.

Figure 1.2: Example of JSON stored medicine data

1.2 Problem statement

The unsatisfactory storage and query methods for (partially) non-relational data on mobile devices, as described in the previous section, leaves room for a (real world usable) solution which bridges this gap. We underpinned this observation and placed it in context by using a company example. In this section we will formulate a problem statement and corresponding research questions which will form our basis to tackle the problem at hand. The effort that needs to be done can be grasped in the following problem statement:

Current mobile storage of partially unstructured, spare or schema-less data has many shortcomings. To tackle this problem an extension to SQLite, the de facto standard for mobile databases, has to be designed and developed, which is capable to efficiently store JSON, as well as SQL relational data.

Since the problem is complex and highly multidimensional it is important to first define what boundary conditions a possible solution has to address to. First, the approach to be chosen has to be able to efficiently store and query partly non-relational JSON data, while not detracting normal SQL performance and query abilities. Furthermore to ensure usability we constantly have to look at the behavior of our approach in real-world use cases and come up with an implementation that requires minimum effort and maximum backwards compatibility. Finally disk space usage has to maintain within reasonable boundaries.

The above problem statement and boundary conditions can be translated to the following main

research question.

'How and in what way can we extend SQLite, so that is equipped to efficiently store and reason over JSON data?'

From this we can derive three subquestions, listed below.

1. Could SQLite be extended so that it can efficiently store document data?
2. Could SQLite be extended so that it can efficiently reason over document data?
3. Could SQLite be extended without effecting normal SQLite performance and usability on relational data?

Since for different data and queries the above questions can differ in importance, Section 4.1.2 further defines evaluation criteria and Section 6.4 defines a set of queries for our experimental evaluation.

1.3 Contributions

The main contribution of this thesis is the design of ESQLite, an open source wrapper library on top of SQLite, especially designed for Android applications [25]. For ESQLite to come to existence we first looked at the current shortcomings in handling JSON data and defined a general problem statement and research questions.

The main concept within our approach is the shredding an reconstruction of JSON to key-value tuples which can be stored relational. This storage technique enables us to query the JSON very efficiently. We discuss several shredding techniques. We investigated at both insertion time shredding as selection time shredding alongside some optimization techniques. Next we discussed three possible JSON reconstruction approaches, again with several optimizations. At last our approach enables us to use an XPath like Query language extension which is capable of querying JSON on key,value and path basis.

The approach at hand is then translated to our implementation ESQLite en thoroughly evaluated. Results show that while ESQLite decreases the insertion and selection speed significantly, it shows great advantages when querying the relational stored data. ESQLite furthermore does not effect normal SQL performance for relational data. ESQLite meets the requirements as denoted in our research questions, and for several known limitations improvements are at hand. Based on these observations we conclude that ESQLite turns out to, however there is much to improve, successfully meet our demand of handling JSON data efficiently while maintaining usability.

1.4 Thesis overview

In the next chapter, Chapter 2, will give a preliminary overview of the field of interest. Concepts as SQL, NoSQL, Mobile Storage and SQLite are discussed in more detail, providing the necessarily background.

In Chapter 3 related work is studied. First the overall architectural approaches are discussed, after which questions are proposed which are used to further discuss four different specific solutions available.

In Chapter 4 the problem is formalized an a general (implementation in depended) approach is discussed. First evaluation criteria are outlined, after which the necessarily notations are elaborated. Then from a mathematical basis all different concepts which add up to what will be the

ESQLite framework are elaborated and materialized in the form of pseudocode. The chapter is concluded with an architectural overview of the approach as proposed.

Chapter 5 discusses the translation of our approach to a usable implementation framework called ESQLite. In the previous chapter different approaches for specific subproblems are outlined, which in this chapter are translated to different implementation variations. Since one of the tree main evaluation criteria is usability, special interest is put into this aspect in this section.

In Chapter 6 the experimental setup is discussed, elaborating on the approach, environment, objective, chosen queries and datasets. When the evaluation setup is laid out, in Chapter 7 the different variations of the implementations are evaluated. After some preliminary tests are performed, a detailed look at the performance of JSON1 and ESQLite in specific is discussed.

Finally Chapter 8 concludes the results of the thesis, by means of the evaluation research. Also we elaborate on suggestions for future work and limitations of the current implementation.

Chapter 2

Preliminaries

This chapter serves as background on the topic by providing context, techniques and definitions in the field of mobile storage, which will be used to propose a solution. For a domain expert reader this chapter may be superfluous. In Section 2.1 we take a look at the concepts of SQL and NoSQL, after which in Section 2.2 we take a brief look at the field of mobile storage in specific.

2.1 SQL and NoSQL

Database management systems (DBMS) can roughly be divided in relational and non-relational databases. The Relational databases, as first described by Edgar Frank Codd [1], have been dominant for the last three decades. But the fast increase in the amount of data generated pushes relational database systems beyond their limits. Researchers therefore looked at other approaches that can meet the modern day requests, resulting in the rise of what is called *Not only SQL* or *NoSQL* database solutions. For a complete survey on NoSQL data-stores one could look at the work of Jing et al [3].

2.1.1 SQL

The rational approach, gained popularity by its simplicity of storing rows representing instances of an entity and columns representing the corresponding attribute values describing each instance. Also the Structured Query Language SQL, is one of the competitive advantages of relational RDBMS. SQL is a declarative language, so how data is stored on the actual system is abstracted from. The programmer has no notion of the data storage layer.

Relational database's have several strong features. The most important one is ACID transactions, which guarantee the availability, consistency, isolation and durability of the database transactions. ACID is a set of properties which guarantee that transactions in a database system maintain the databases integrity. Atomicity requires that transactions are all or nothing, so no partial transaction is possible. Consistency ensures that every given transaction will bring the database from one valid state to another valid state. Isolation ensures that concurrent transactions give the same final state as executing these states in serial, and finally durability ensures that committed transactions remain. ACID was introduced by Jim Gray in the 1970s [9]. For these properties to be possible relational databases demand strict schema's to which the data has to adhere. Relational databases are also normalized, meaning that data is only stored once, ensuring data consistency and integrity while maintaining storage needs to a minimum.

Relational databases also have several shortcomings, most notably listed below:

1. To maintain a RDBMS a lot of specific domain knowledge is needed to facilitate a specific programming language to interact with SQL.

2. Scaling and distribution of RDBMS is difficult, particularly when scaling to multiple hardware nodes or physical locations is needed.

3. The data has to adhere to a strict schema. Schema-less data such as log-files are not easy to store and query.

4. ACID transactions introduce latency and computational bottlenecks.

5. New loosely defined data formats such as JSON, GSON, XML are not supported by RDBMS and are all viewed as simple (non queryable) text.

6. SQL queries can become very complex and non-intuitive. There is a mismatch between application logic and database logic.

2.1.2 NoSQL

NoSQL database systems address the shortcomings of relational databases. The fast growing amounts of data, web 2.0 like applications and large amounts of user-generated content foster the development of alternatives to relational databases. One of the first real world implementations of NoSQL was Googles BigTable [10]. There are three common types of non relational RDBMS systems: Key-value stores, which store information as key-value pairs in associative arrays. Graph databases which store interconnected objects as nodes and edges. Finally column-oriented stores, which store data into columns rather than into rows. These systems are particularly strong in not only storing relational data but also log-files, binary, graph, unstructured, XML, JSON and document data.For the scope of this project we will not go into details regarding NoSQL techniques.

The main advantages of NoSQL are; the massive scalability, distributed architecture, possibility to store schema-less data, native querying and coping with high volumes of read/write data. NoSQL fills a void in the database landscape, especially when ACID transactions are of limited interest.
The most notable shortcomings of NoSQL are listed below:

- NoSQL typically has no ACID transactions and a trade-off between availability, consistency, integrity and distributivity has to be made, making them less suitable for all round tasks.

- The concept is relatively young and therefore limited standardization and corporate ready solutions are in place.

- The underlying models differ greatly between different solutions, making interoperability difficult.

Imported to note is that while NoSQL fills some of the shortcomings of RDBMS both fill a different part of the changing database landscape.

2.2 Mobile storage

Whereas most database systems are hosted on large servers or computers, the upcoming amount of lightweight clients such as smart-phones, tablets and other smart devices increase the need for lightweight mobile data-stores. Especially since these kind of devices fulfill an interface role between user, other connected devices and the Internet. Mobile devices are often equipped with a large amount of sensors and are capable of registering large amounts of data.

The spectrum of mobile devices is very broad, varying from tables, drones, phones to watches. In this research we focus on mobile phone clients equipped with a fully functional operating system such as Android, iOS, Windows or Linux. For these platforms the database system SQLite is the most dominant solution, on which we will elaborate further in the next section.

2.2.1 SQLite

SQLite is a relational database system based on the programming language C. It is developed as a lightweight version of MySQL, capable of handling relational data with ACID properties is fully supported. SQLite tables have fixed schemes describing the data, and in that regard can be seen as a traditional relational database system (RDBMS). However there are some great differences between SQLite and other RDBMS. The most important are outlined below:

- SQLite is a self-contained single file based database, making it enormously portable. SQLite as a whole is 0.3 MB in size.
- SQLite's single file strategy makes it possible to load the entire database into memory.
- SQLite does not offer security, user management, or multi-threading support.
- SQLite only supports a single write operation at a given time, and has no concurrency control whatsoever.
- SQLite has a zero-configuration policy, creating a database is a single action.
- SQLite has no server-client model, and directly interacts with the operating system.
- SQLite has a very flexible variable-length record system which stores rows as simple linked lists.
- SQLite supports fewer data-types compared to a fully fledged RDBMS.

Below in Figure 2.1 a schematic overview of SQLite is given, explaining the basic architecture of SQLite consisting of three parts; The core, backend and SQL compiler. The core contains the interface with application logic, SQL processor and virtual machine. Whereas the SQL compiler consists of a tokenizer which splits up the queries, parser and code generator converting the query to byte code. The backend assures the representation and storage of the data by means of a B-tree, pagination system and OS interface. For a detailed explanation of the architecture, one could look at [12].

2.2.2 Android OS

Android is widely know as an mobile operating system developed by Google. Android runs on more then one Billion devices worldwide and its currently last stable release is Android 6.0. For the scope of this report a detailed android explanation is superficial, but some technical details will be shortly explained below.

Figure 2.2 shows that Android runs on a Linux kernel written in C. On top of the kernel a wide variability of libraries is deployed, which handle for instance media, video processing and network communications and can be accessed through *Application Programmer Interfaces*. One of these libraries is SQLite. Furthermore on top of the kernel the *Android runtime* is deployed. This is a Java virtual machine which is capable of compiling Java to byte-code on runtime. This immediately sheds light on the Java oriented top level in which the application framework and all applications are accommodated. Whereas Java from a programmer perspective is almost completely Java oriented, the underlying core including SQLite is programmed in C.
Android applications run in a Sandbox on top of the Android framework. This ensures that applications have no access to the rest of the system and can be allocated a fixed amount of resources, the latter one is important for testing purposes.

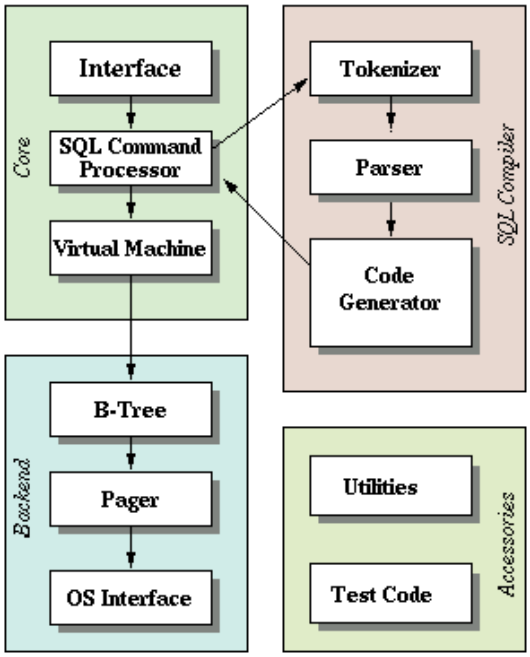


Figure 2.1: Architectural overview of SQLite [12]

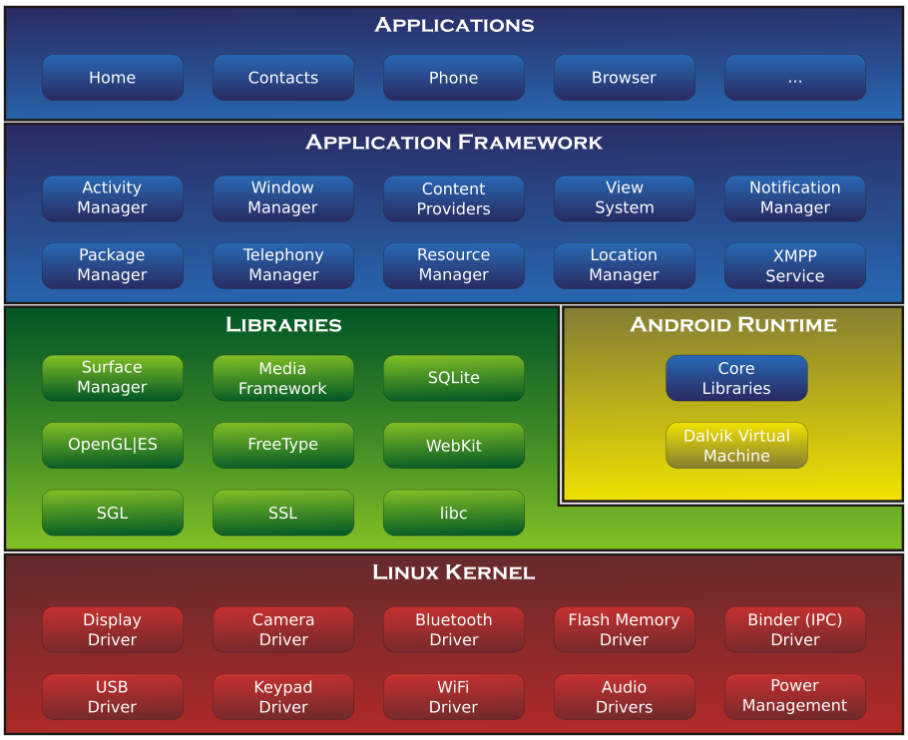


Figure 2.2: Architectural overview of Android [11]

2.2.3 JSON

JSON, which stands for *Javascript Object Notation* is an open standard for transmitting data objects in the form of attribute-value pairs. The standard is human readable and does not force a certain schema or language. Key-value pairs can be of type *text*, *int*, *boolean*, *null*, *object* or *array*, and the latter two can be of any of these types, facilitating nesting. JSON is first introduced by Douglas Crockford and at the time of writing JSON is the most common data transfer format used on the Internet [14]. In Section 4.2.1 we further elaborate on the notation of JSON.

Chapter 3

Related work

This chapter enables to place our project in the context of previous performed research and development. In recent years a lot of research is done which looked beyond the scope of traditional Relational Database Systems (RDBMS). Resulting in the development of NoSQL on one side and on the other side extensions and enhancements to RDBMS. However most of the research is focused on non-relational XML, document data or data in general, less is known about how to cope with JSON data. In this section the current state of art is assessed. For the scope of this chapter we look primarily at JSON enabled RDBMS, but also take a look at XML and document-data oriented studies, since there is significant similarity.

First a brief classification on the architectural approaches which are currently available is done in Section 3.1. Architectural approaches of interest are *shredding*, *native* storage, *JSON type* storage and *JSON mappers and layers*. We then focus on four specific solutions derived from these approaches in Section 3.2. In detail we discuss the *Argo* framework, *native storage*, *PostgreSQL* and *MySQL* and finally *JSON1*. Later on these solutions are used as a source of inspiration when designing our own approach to address the problem of JSON storage.

3.1 Overall architectural approaches

At architectural level there are several different approaches to storing and querying JSON data, the most common are outlined below. Each approach has different strengths, weaknesses and use cases, which will be elaborated very briefly. For a more extensive analysis and benchmarking one could look at referred studies.

Shredding One of the popular approaches in handling JSON is decomposing the JSON objects into tables containing *id-name-value* triples. This is referred to as shredding. Examples of this approach are Argo, and LegoDB [4] [6]. We will discuss Argo in detail in the next section. Shredding is efficient in querying specific properties of objects, but is extremely slow when documents have to be inserted or reconstructed. To reconstruct a document multiple tables have to be accessed and a number of self-joins are needed.

Native (blob/text) The second approach is storing JSON as a string in a native *blob* type or *text* (variation is *varchar*) type. In its simple form this approach does not need any RDBMS modification and the application layer takes most of the heavy lifting. Storing as a *blob* is the fasted solution since it has no further translation steps, but comes with the price of limited query possibilities, because most RDBMS systems can only perform searches on *text* data-types. We discuss this in more detail in Chapter 5. This approach is less I/O intensive compared to shredding and therefore is very fast, but offers very limited query and index capability. All selection querying has to be done in memory or application logic, limiting the maximum document size.

This approach offers full flexibility in term of schema-less storage. Since this approach does not require any changes to the RDBMS it is often used as a quick fix.

native JSON type To enhance query and index capabilities the RDMBS can be extended with the introduction of a native JSON type. The most well known example is PostgreSQL which recently introduced the support of such a native JSON type. [15].

JSON functions, mappers and layers Another approach to enhancing the functionality of a JSON enabled RDBMS is by defining *custom functions*. The best known example is the recently introduced MySQL JSON support [16]. Besides custom functions also custom operators and interfaces can be implemented resulting in more complex mappers and layers. There are several projects available such as NoSQLite, Ecstortive or MongoLite, which are SQLite and python based document stores [18] [19] [17]. Both SQL and SQLite support custom functions, which enable developers to write python or C functions to, for example, verify, import, export or minimize JSON. For the scope of this research no in-depth look at index optimizing techniques as proposed by some of these studies is taken.

Interpreted One might store JSON in a shredded form to avoid null values for sparse data. Sparse data exists of records with many attributes that are null for most records. One could choose a 'vertical' storage approach such as shredding, but as mentioned this comes with a price regarding performance. Another approach is to extend the RDBMS tuple storage format to allow the representation of sparse attributes as interpreted fields. Interpreted storage differs from normal SQL storage in that the association between a data value and its attribute is represented by a tag in the data value. An interpreted record has a header containing relation-id, tuple-id and tuple length. When a value for an attribute is known the record than contains the attribute identifier, length and value for this attribute. The attribute identifier corresponds to an attribute catalog. Interpreted storage gives more efficient and flexible querying. In the studies of Beckmann et all a PostgreSQL based implementation is discussed [7].

3.2 Specific solutions

In the previous section four distinctively architectural approaches to storing and querying JSON are described. In this subsection several specific implementations will be discussed. To streamline the discussion, the following four questions will be addressed.

1. What is the query language or API?
2. How and in what way is the query language related to the RDBMS?
3. How is the JSON data stored and/or represented in the RDBMS?
4. Where is the solution located in the database engine?

3.2.1 Argo (shredding)

In the paper of Craig Chasseur et al, an automated mapping layer for storing and querying JSON data in a RDBMS called Argo is proposed [4]. Argo claims to provide all advantages of document stores while providing higher performance and richer functionality. It supports ACID transactions and native joins, which normal document stores are not capable of. Argo consist of two main components; the mapping layer to convert JSON objects to relational triples and vice versa and a SQL like JSON query language called Argo/SQL. Argo is benchmarked using NoBench which shows some interesting results. Argo performs very well compared to MongoDB when the store contains less than four million records. Especially insert and selection queries are fast, up to 70 times faster then MongoDB. Join queries however are significantly slower.

14ESQLite: A relational database solution for JSON data with applications in mobile computing

What is the query language/API? Argo introduces Argo/SQL which is a SQL like query language for collections of JSON objects, it supports INSERT, SELECT and DELETE. For the SELECT and DELETE queries argo is capable of evaluating simple predicates such as mathematical operators or LIKE patterns. Finally Argo is capable of doing single INNER joins, but it does not support complex joins. Below is a simple example of a Argo/SQL INSERT.

```
1 INSERT INTO collection_name OBJECT{..}
```

Listing 3.1: Argo notation of INSERT query containing JSON

How and in what way is the query language related to the RDBMS? Argo is a mapping layer on top of the RDBMS. When performing an INSERT query it begins as a SQL transaction on the RDBMS. Argo then recursively walks through the structure of the JSON document while executing INSERT statements on the underlying table structure. When finished, Argo commits the transaction to the RDBMS. When performing a SELECT query on Argo, it fetches the matching object ID's and stores these in a temporary table. Then it fetches all attribute values from the desired tables. The paper of Chasseur et all discusses the techniques used in more detail [4].

How is the JSON data stored/represented in the RDBMS? Argo decomposes JSON objects to triples. for all JSON attributes a record with a *objid*, *key* and *value* column is created. Argo uses three tables, one for each primitive type of JSON namely; *string*, *number* and *boolean*. Below in Table 3.1 the decomposition of the given JSON document in 3.2 is shown:

```
1  {
2      "name ": "",
3      "age": 35,
4      "kids": [
5          {
6              "name": "Michael",
7              "age": 5
8          }
9      ]
10 }
```

Listing 3.2: JSON data example as passed to Argo.

Objid	keystr	valstr
1	name	George
1	kids[0].name	Michael

Objid	keystr	valnum
1	age	35
1	kids[0].age	5

Table 3.1: Argo table representation of JSON data.

Where is the solution located in the database engine? Because argo is a mapping layer, it sits on top of the existing RDBMS. The actual query engine and storage layer is not altered.

3.2.2 Native storage

The studies of Liu et al considered native storing of JSON data within a RDBMS, the study is based on three architectural principles for storage, querying and indexing [5].
For storage, the JSON data is stored natively without shredding it into relational form. Because JSON objects are schema-less they are stored as one aggregated object in a column without any

decomposition.
The second architectural principle regarding querying considers SQL as a set oriented query language by extending SQL with a JSON path language based on XPath. This path language is able to query over a collection of JSON documents.
Finally the study proposes a partial schema-aware indexing architecture. when Common JSON attributes, sub-documents or members can be identified in a given collection this can be seen as a partial schema. On top of the JSON collection one could then store a secondary structure in the form of a B+ tree representing this partial schema. Lui et al. did a benchmark on Argo and concluded with a solution which is a factor two more efficient in runtime, since no objects have to be build from triples.

What is the query language/API? When storing JSON natively no query language extension is needed, however SQL then offers no support for querying SQL other then as plain text. SQL supports custom data-types and SQL functions, which one could use to further enhance functionality. For instance Lui et al mention the introduction of an *IS-JSON* function which checks if the JSON is syntactically correct.

The Query language is extended to SQL/JSON by introducing a set of operators and functions which embed a JSON path language. SQL/JSON is derived from the ISO SQL/XML standard. Functions as *JSON-EXISTS*, *JSON-VALUE JSON-TABLE* are introduced. JSON-TABLE for instance expands a JSON array into a set of relational rows with each corresponding to an element in the array. In this way *JSON-TABLE* bridges between JSON and relational data. *JSON-VALUE* is used to extract scalar values from JSON and cast them to corresponding SQL types. Also *JSON-QUERY* is capable of extracting parts of JSON and create new JSON objects. Finally default SQL INSERT, DELETE, and UPDATE statements can be used.
The SQL/JSON path language is similar to XPath and JsonPath [20] [22].

How and in what way is the query language related to the RDBMS? SQL/JSON is an extension of SQL. Lui et al propose a stream processing architecture which feeds JSON streams to a SQL/JSON path language processor. No further detail on the implementation of the processor is given in the paper.

How is the JSON data stored/represented in the RDBMS? A collection of JSON objects is stored as records in a table with a JSON column. The column type can be *varchar*, *blob* or *raw* in which the JSON is natively stored.

Where is the solution located in the database engine? The SQL/JSON functions can be implemented as custom functions or as built-in functions in the core of the specific RDBMS.

3.2.3 PostgreSQL and MySQL

PostgreSQL, together with MySQL are the most well known RDBMS, since PostgreSQL 9.2 JSON it has a natively supported JSON data type. This enabled more efficient storage rather than just storing json as *blob* or *varchar*. This version also introduced a few JSON specific functions, for instance *is-valid-json*, *array-to-json* and *row-to-json*. As of the release of 9.3 a more complete JSON support is introduced.
MySQL only recently introduced JSON support. In contrary to PostgreSQL, no native JSON type is introduced but rather functionality is added on top of the *varchar* data type. Microsoft chose to do this for compatibility and migration reasons. Likewise to PostgreSQL it has custom functions and operators. MySQL is also capable of exporting, importing and transforming between JSON and relational form and next to that has indexing capabilities, based on B-trees.

What is the query language/API? For both MySQL as PostgreSQL the normal SQL query language is extend with new functions and operators. PostgreSQL also introduced a native JSON data type.

How and in what way is the query language related to the RDBMS? The same as in normal MySQL and PostgreSQL databases.

How is the JSON data stored/represented in the RDBMS? The JSON data is in both cases stored as text in a table column. For the indexing capabilities of MySQL also system tables for storing B+ trees are generated.

Where is the solution located in the database engine? The solution is implemented on query engine level. The query planner, optimizer and executor are improved to handle the new functionality.

3.2.4 JSON1

JSON1 is a SQLite extension that implements thirteen functions capable of managing JSON data. Compared with solutions above JSON1 is quite limited, however it is one of the few JSON enabled implementations already build on SQLite. The '1' annotation is on purpose, since the developer expects new extensions to follow shortly.

What is the query language/API? The query language is an extension on SQL. Examples of implemented functions are *json*, which verifies and minifies a JSON string. *Json-array*, which returns a json array and *json-extract*, *json-insert*, *json remove*. For example, query 3.3 below uses *json-extract* to extract the titles of all entries that contain the given tag. More examples can be found in the SQLite documentation.

```
1 SELECT json_extract("t1"."data", '.title') AS title FROM "entry"
```

Listing 3.3: JSON1 notation of INSERT query containing JSON

Furthermore virtual table functions are available such as *json-each* and *json-tree* which are capable of performing tree walks on objects. When a table stores zero or more phone numbers as a JSON array object called user.phone, one could use *json-each* to find all users have some specific area code.

```
1 SELECT DISTINCT user.name
2   FROM user, json_each(user.phone)
3  WHERE json_each.value LIKE '704-%';
```

Listing 3.4: JSON1 notation of SELECT query containing JSON

How and where is the query language related to the RDBMS? The same as normal SQL.

How is the JSON data stored/represented in the RDBMS? JSON data is stored as type *text*. No new type is introduced, so that backwards compatibility is ensured.

Where is the solution located in the database engine? JSON1 can be loaded as an extension to the core of SQLite.

3.2.5 Others

Below a list of other less known projects is given. These are only briefly described.

NoSQLite NoSQLite is lightweight, document-oriented and based on on Python and SQLite. It provides a very lightweight and easy way to use SQLite and has analogies to MongoDB [18].

MongoLite MongoLite is a schema-less database on top of SQLite. It is referred to as 'What UnQLite is to MongoDB as SQLite is to PostgreSQL' [17].

JsonLite JsonLite is a wrapper. Since it is very limited and has no query interface, no further investigation is performed, but an interested reader could take a look at the documentation [21].

Chapter 4

Our approach: ESQLite

In the previous chapters we discussed the problem of storing and querying partially unstructured, sparse or schema-less data. In this chapter we will introduce ESQLite as our approach for solving the problem at hand. To support decision making in terms of our approach we will first discuss design criteria in Section 4.1. We aim to grasp all design criterion and make the best possible combination, adding up to the ESQLite framework. When addressing the problem one has to look at all three aspects of the database system and their mutual interaction. These are; the database storage engine, query engine and query language. Therefore in Section 4.2 we look at the storage aspect, in Section 4.3 we investigate the desired query language extension, and finally in Section 4.4 focuses on the query engine. Note that this chapter does not concern implementation design decisions, this will be discussed in Chapter 5.

4.1 Design criteria

Before we can propose an approach to the problem at hand, the design criteria it has to meet will be discussed. In Section 4.1.1 we elaborate on use cases on which our approach has to address to. In Section 4.1.2 we then determine evaluation criteria that form the basis of our approach. The evaluation criteria are closely aligned to the research question as opposed in Chapter 1.

4.1.1 Use cases

As there is a large amount of use cases to elaborate on, we will focus on cases closely aligned to our field of studies. First two generic cases are addressed, followed by more specific use cases derived from the Drug database use case used in this research.

One-to-may data When inserting a large amount of records to a single relational table, SQLite is able to batch insertions into a single transaction. This means a number of records is written to the disk at once, rather then each record separately. This ensures that SQLite is capable of handling large amounts of insertions reasonably fast.

When inserting relational data which has to be split over two tables, A and B linked with a foreign key constraint, a problem arises. Batching the insertions becomes impossible since SQLite first has to store the first part in table A and use the ID of that item to construct the foreign key as saved in table B. This results into a decrease of performance. Also lookups of data become slower since joins over multiple tables are needed. An example of this type of data is a system which logs all sessions of a user on a certain software system. The database then stores users in a user table and sessions in a a session table. There is a foreign key relation between one user and zero or more sessions.

Hierarchical data The above use case elaborates on a one-to-may relation. We can further generalize this problem when data is highly hierarchical, as we saw for our real-world medicine example in Section 1.1. Relational storing of this data is done over multiple tables, as well as querying which takes multiple joins. The level of hierarchy within the data determines the number of joins needed.

Storing sparse properties When certain data-objects or items have a lot of possible properties, but only a few of them are used for each specific item, the result is very sparse data. For instance every car has some properties such as the brand, type, color, and license plate. But one specific car can have one or more of the following; stereo installation, navigation, parking sensors, climate control, and so on. When storing this in a normal relational table, one would get a very high number of columns containing only null values. Often a workaround is to make a separate table which stores triples with a parent identifier, property-name and property-value. When single column data has a large set of sparse properties, the storage in a single table results in a large amount of null values, which occupies a lot of disk space.

Non-relational data When handling (partially) non-relational data, often precisely this part of the data, stored as JSON, is of interest to query. SQL does not support querying on this data, other then searching for specific substrings by use of the LIKE clause. When for instance a specific key-value pair within the document at a specific location (path) is of interest, path querying techniques are needed.

4.1.2 Evaluation criteria

Since we now have an understanding of the use cases of interest, it can be concluded that the landscape of data storage and handling is very complex. For this reason a proposed solution can never be equipped to offer an *one stop shop*. For the purpose of this project we define four main evaluation criteria ro implement and evaluate a solution. These evaluation criteria are derived from our research question as stated in Section 1.2 and are outlined below.

Runtime performance The first and most important criterion is performance. Since this is rather abstract, we will focus on the runtime performance in specific. Important to note is that this pure statistical metric may turn out to be a inadequate proxy for the overall performance as experienced by the user, and therefore some additional criteria have to be defined. In relational database systems the four elemental operations *Create*, *Read*, *Update* and *Delete* can be distinguished, alongside with other more complex statements. For the scope of this project we focus on an approach which tries to satisfy performance requirements to these four operations, whereas the focus on for instance *Join* operations is left as future work.

Disk space performance The second most important criterion is the disk space usage. Since databases handle vast amounts of data, their disk footprint can grow rapidly. While this is a key element of interest for database systems, it is even more important for our desired implementation explicitly designed for lightweight mobile devices. Runtime versus disk space performance are at odds, which will be extensively discussed in the coming sections.

Query Capabilities The third evaluation criterion is regarding the query possibilities of the JSON document part. Though SQL turns out to be very suitable to query relational data, it offers limited to none support for querying on the JSON part of the data. Therefore the approach of choice has to address the ability to query the JSON data efficiently. This criterion is tightly connected to the first two.

Usability The last criterion we take into account is from a non technical perspective. Our approach has to maintain high usability. Meaning that it can be used easily by programmers, is compatible with current available implementations and works in a real world situation. The latter one implies that our approach has to perform well over a broad spectrum of use cases and data-types, providing extra challenges to the runtime and disk space criteria as denoted above.

ACID Since normal relational database systems as investigated in the related work chapter are all ACID database systems, the requirement of a full ACID database system remains unchanged.

4.2 Storage

The first aspect of interest is the storage of JSON data within SQLite. The most efficient way of storing JSON data, in terms of disk size and insertion speed, is as native *blob*. As clarified in Chapter 3, this also eliminates all query possibilities on the JSON data, so this approach does not satisfy our query capability evaluation criterion. Another approach is to shred the document into tuples, introducing advance query possibilities. We chose to use this technique and tackle the subsequent performance and size drawbacks by implementing several optimizations. In Section 3.2.1 we investigated *Argo* as a solution that uses shredding, which will be of inspiration to our approach. We will now explain how JSON shredding is done, and how it can be optimized.
Before we can dive into more detail we first walk trough some basic notations.

4.2.1 Key-value pair notation

Before we dive further into our approach we first discuss some basic notations regarding SQL and JSON document data. As shown in the previous chapter, JSON data can be represented as document entities. A given JSON document consists of a set of m (possibly nested) key value pairs, denoted as follows:

We now have a clear understanding of the use cases and the evaluation criteria our approach has to address to, which will be used throughout the coming sections.

$$\{k_1 : v_1, k_2 : v_2, ..., k_m : v_m\} \tag{4.1}$$

$$\{k_1 : \{k_{n1} : v_{n1}, k_{n2} : v_{n2}, ..., k_{nm} : v_{nm}\}, k_2 : v_2, ..., k_m : v_m\} \tag{4.2}$$

As can be seen in equation 4.1 and 4.2, key-value pairs can be both single atomic values as well as nested key-value sets.

The JSON syntax denotes three atomic types and two complex types. In table 4.1 all JSON types are listed.

Type	Description
$Number$	A signed decimal number
$String$	A sequence of zero to more unicode characters
$Boolean$	1 bit integer, having values $\{true, false\}$
$Null$	

Table 4.1: Overview of JSON atomic data-types

Besides these atomic data-types, the JSON syntax contains the type *object*, which is an unordered collection of key-value pairs, this is shown in equation 4.3. Finally JSON contains an $array$ object, which is an ordered list of key-value pairs, shown in equation 4.4.

$$JSON_{object} = \{k_1 : v_1, k_2 : v_2, ..., k_n : v_n\} \tag{4.3}$$

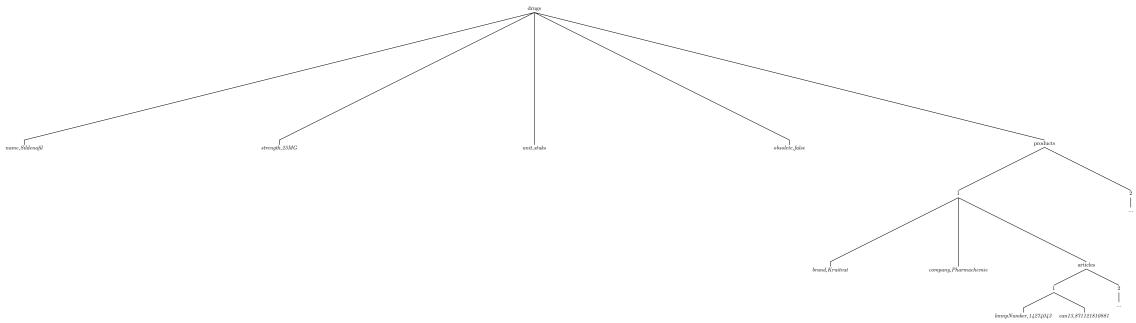
$$JSON_{array} = [JSON_{object_1}, JSON_{object_2}, ..JSON_{object_n}] \tag{4.4}$$

In equation 4.5 we show a set of atomic key-value pairs in set notation and their translation to a set of 2-tuples in 4.6.

$$\{k_1 : v_1, k_2 : v_2, ..., k_m : v_m\} \tag{4.5}$$

$$\{(k_1, v_1), (k_2, v_2), .., (k_m, v_m)\} \tag{4.6}$$

Since JSON documents often have a highly nested structure, one could represent a JSON object as a tree of key-value pairs, all key-value pairs containing an atomic value represent leaf nodes, whereas the non atomic values; *object* and *array* are intermediate nodes. In Figure **??** a simple tree representation of a document is given.


drugs
name,Sildenafil
strength,25MG
unit,stuks
obsolete,false
products
1
2
...
brand,Kruitvat
company,Pharmachemie
articles
1
2
...
knmpNumber,14274043
ean13,8711121810881

To conclude this section we showed that a JSON document consists of a set of key-value pair of either atomic or complex types. A given key-value pair can be expressed as a set, tuple or (sub)tree.

4.2.2 Shredding

As described in Section 4.2.1 a JSON document can be denoted a set key-value pairs. To be able to fit the nested structure of a document into a relational form, we shred a document into 4-tuples as shown in equation 4.7.

$$\{k1 : v1\} \rightarrow (id, parent, key, value) \tag{4.7}$$

While shredding the document into 4-tuples, the internal structure of the document has to be stored as well. Since in Section 4.2.1 we show that a document can be expressed as a tree, we therefore store the key, value and parent pointer of a given key-value pair.

We define two functions F_{shred} and K_{shred} to shred a nested document while preserving its structure this is done by storing parent pointers. Performing 4.8 and 4.9 on a given document returns te union of all its child tuples. 4.9 returns a tuple for atomic JSON types and the union of the tuple with a recursive call on F_{shred} for complex types. The recursiveness of the combination of 4.8 and 4.9 ensures that the parent id of a nested tuple is stored within the tuple.

$$F_{shred}(JSON) = \cup_i K_{shred}(v_i) \text{ where } v_i \in JSON \tag{4.8}$$

$$K_{shred}(v_i) = \begin{cases} (id, parent, key, value) & \text{if } k_i \text{ is atomic} \\ (id, parent, key, null) \cup F_{shred}(v_i) & \text{if } k_i \text{ is an object} \end{cases} \tag{4.9}$$

The above set notation can be rewritten to pseudocode as shown in listing 4.1.

```
1  function shred(object)
2    for(int i < keys.count)
3      key = keys[i]
4      value = values[i]
5      if (value instanceOf JSONObject)
6        shred(value)
7        store::(id,parents,key,null)
8      else if (value instanceOf (number|boolean|string|null))
9        store::(id,parents,key,value)
```

Listing 4.1: Pseudocode representation of shred function

Since JSON supports both unordered lists of key-value pairs in the form of $JSONObjects$, as well as ordered sets of key-value pairs in $JSONArrays$, the above is not sufficient. The following pseudocode in Listing 4.2 gives the array support needed to shred JSON.

```
1  function shred(object)
2    for(int i < keys.count)
3      key = keys[i]
4      value = values[i]
5      if (value instanceOf JSONObject)
6        shred(value)
7        store::(id,parents,key,null)
8      else if (value instanceOf JSONArray)
9        shredArray(value)
10       store::(id,parents,key,null)
11     else if (value instanceOf (number|boolean|string|null))
12       store::(id,parents,key,value)
13
14 function shredArray(array)
15   for (int i < array.length)
16     shred(array[i])
```

Listing 4.2: Pseudocode representation of extended shred function

There is another approach to shredding. This approach does not need both the storage of the tuples ID and parent ID, but only an unique ID for every nesting level. This means that siblings have the same ID, but parent-child relations have a distinct ID. Tree reconstruction is still possible in this case, but it has a higher runtime. On the other hand this approach reduces the storage needs. Because of the higher runtime, this approach is not further investigated in this research.

Shredding can be done both at insertion- and selection-time of a given document, both of which will be evaluated in Chapter 7.

4.2.3 Reconstruction

This section concerns the reconstruction of tuples to JSON documents. There are two main strategies to distinguish here; first is the reconstruction of the document on query level, and second the reconstruction on application (in code) level.

IN_QUERY A commonly used reconstruction method for JSON documents from tuples is done through self-joins. Since all tuples are stored as rows in the same table, in order to reconstruct the tree one has to join all tuples to form that table. In the paper of Roijackers, this approach is discussed in detail [8]. Below in Listing 4.3 a query illustrating this reconstruction technique is shown for a simple JSON object.

```
SELECT shred_0.key, shred_0.value, ...shred_n.key, schred_n.value
FROM shred as shred_0
LEFT JOIN shred as shred_1
 ..
LEFT JOIN shred as shred_n
WHERE shred_0.id = <id>
```

Listing 4.3: Example of IN_QUERY reconstruction

The example of the simple selection in Listing 4.3 shows the complex nature of this reconstruction method, forcing a large number of expensive *join* operations on the same table. This approach can be further improved when the root and tree level of each tuple is stored, removing the need for multiple self joins as shown below.

```
SELECT shred0.*
FROM shred as shred_0
WHERE shred_0.root = <id>
ORDER by level
```

Listing 4.4: Example of IN_QUERY reconstruction

However the above seems to be quite efficient, still a lot of post processing needs to be done to reconstruct the JSON tree structure, bringing us to the concept of IN_CODE reconstruction.

IN_CODE To overcome the problems of the previous mentioned IN_QUERY approach, the IN_CODE approach fetches the required tuples form the database and reconstructs the object in application logic. The main advantage of this approach is that complex document reconstructions are divided in separate database calls, and the object can be build op stepwise and intermediately stored in memory. Needles to say this approach introduces more overhead in the application logic. In Listing 4.5 a pseudocode example of IN_CODE reconstruction is shown:

```
function reconstruct(ID)
  object = new JSONOBject
  shred = database.get(WHERE parent = ID)
  for(int i < keys.count)
    key = keys[i]
    value = values[i]
    if (value instanceOf JSONObject)
      object.put(key, reconstruct(shred.id));
```

```
9       else if (value instanceOf (number|boolean|string|null))
10        object.put(key,value)
11    return object
```

Listing 4.5: Pseudocode representation of reconstruct function

4.2.4 Optimizations

As pointed out in the beginning of this chapter; while the approach of shredding and reconstructing a JSON gives us vast query possibilities, it has a severe impact on the performance and space requirements. Both will be examined in Chapter 7. For this reason several optimization techniques are discussed in this section. The first optimization technique is *relational base schema storing*, followed by extending the tuples with additional meta data. Note that the IN_CODE reconstruction technique of the previous chapter can be seen as a type of the first optimization.

Relational base schema

JSON documents are in essence schema-less. However in most use cases the data seems to have some sort of schema overlap in between them in the form of recurring objects or key-value pairs. We call this the *relational base schema*. Given a set of two documents, as showed below:

```
1 {
2   firstName:"Edwin",
3   lastName:"Hermkens",
4   grades:[
5       {"0":10},
6       {"1",6}
7     ]
8   "graduated":true
9 }
```

Listing 4.6: JSON relational base - 1

```
1 {
2   firstName:"Bob",
3   lastName:"Williams",
4   grades:[
5     {"0":8},
6     {"1",6}
7     {"2",7}
8     ]
9 }
```

Listing 4.7: JSON relational base - 2

In the above example one can see that both $firstName$ and $lastName$ occur in all documents, whereas *graduated* only occurs in one and *grades* has different number of child objects. We can therefore derive a common base schema for this set of two documents, containing the keys $firstName$ and $lastName$.
Since this data is relational, it leaves room to store these properties as additional relational columns outside of the JSON document. For the scope of this thesis we limit ourselves to only extracting a relational base on the root level of a document. Equation 4.10 $F_{base}(JSON)$ and Equation 4.11 $K_{base}(v_i)$ are shown below and can be used for extracting the relational base of a document. 4.10 takes the union of all child objects, then 4.11 returns every tuple that is of an atomic type.

$$F_{base}(JSON) = \cup_i K_{base}(v_i) \text{ where } v_i \in JSON \tag{4.10}$$

$$K_{base}(v_i) = \begin{cases} (id, parent, key, value) & \text{if } k_i \text{ is atomic} \\ & \text{if } k_i \text{ is an object} \end{cases} \tag{4.11}$$

The above notation can be rewritten to pseudocode, as shown in Listing 4.8. The function calculates a base by dismissing any keys that at some point, when storing a new JSON document, are absent. This ensures that only keys that are in every document will be evaluated as relational base. The drawback of this approach is that our optimization does not perform well when the relational base changes over time.

```
1 function calculate(JSON) {
2   cache = new Cache()
3   if (!cache.active)
4     for(int i < keys.count)
5       if (value instanceOf (number|boolean|string|null))
6         if(!cache.contains(key))
7           cache.add(key)
```

```
8       cache.active = true;
9     else
10      for(int i < keys.count)
11        if (!JSON.has(key))
12          cache.remove(key)
```

Listing 4.8: Pseudocode respresentation for calculating (relational base) function

Root, path, type storage

The 4-tuple can be extended to further improve the reconstruction and query performance by storing additional information such as the root, path and type of the key-value pair. This additional stored information off course has impact on the disk space requirements of the database. Equations 4.12, 4.13, 4.14 and 4.15 show the corresponding notation for each of them, we also give a brief explanation below.

$$\{k1 : v1\} \rightarrow (id, root, parent, key, value) \tag{4.12}$$
$$\{k1 : v1\} \rightarrow (id, path, parent, key, value) \tag{4.13}$$
$$\{k1 : v1\} \rightarrow (id, root, path, parent, key, value) \tag{4.14}$$
$$\{k1 : v1\} \rightarrow (id, root, path, parent, key, value, type) \tag{4.15}$$

Root Storing the root for a given tuple, eliminates the need of traversing the tree from the selected key-value pair to its root when querying. In Listing 4.9 a pseudocode example to get the root of a given key-value tuple is shown.

```
1   getRoot(id) {
2   shred = database.get(WHERE parent = ID)
3   for(int i < keys.count)
4     String parent = shred.parent
5     if (parent == id) { //is root
6       return guid
7     else
8   return getRoot(parent)
```

Listing 4.9: Pseudocode representation of getRoot function

Path Storing the path for any given tuple ensures that not all matching key-value pairs for a query have to be evaluated, but only those with the path structure matches that of the XPath, without first having to traverse the tree and storing the path. In Listing 4.10 a pseudocode example to get the path of a given key-value tuple.

```
1 getPath(id) {
2   shred = database.get(WHERE parent = ID)
3   for(int i < keys.count)
4     String parent = shred.parent
5     if (parent == id) { //is root
6       return;
7     else
8       ret = getPath(parent) + "/" + key
9   return ret
```

Listing 4.10: Pseudocode representation of getPath function

Type Storing the type of a given key-value tuple eliminates the need of calculating the type at each query.

Dynamic shredding

The proposed shredding technique handles every new JSON document exactly the same. If for a given set of documents there is a high document similarity, this means a redundant storage of tuples. In the most extreme case a collection of n exactly the same documents result in $n-1$ duplicates of exactly the same data tuples. This can be prevented by using *dynamic shredding*. If a tuple already exists, then store multiple root ID's to that tuple, rather then the whole tuple. This technique is often referred to sub document embedding in NoSQL systems and can be applied to individual tuples or (sub) objects. For the scope of this project *Dynamic shredding* is left as future work.

4.3 Query language extension

The second aspect of the approach concerns the query language. In Section 4.2 shredded storage is proposed, under the presumption that it would leverage query possibilities on the JSON data, in this section we will elaborate on this further. Since SQL is not equipped to reason over tree structured data, we need some form of query language extension. Because one of the four evaluation criterion for our approach is usability, only backwards compatible extensions are considered.

We showed that JSON documents have a tree like structure, therefore it seems favorable to propose some sort of path query language to leverage this structure. In the next section a XPath like query structure is proposed. Our related work investigation in Chapter 3 showed some interesting starting points, which we incorporate in our approach.

4.3.1 Tags

First a convenient system for indicating ESQL extensions within the SQL code is needed. For this we define JSON tags. A tag can be one of the types shown in Table 4.2. Our query engine extension has to detect tags and deal with them accordantly. $TYPE\_JSON$ indicates that the JSON within this tag has to be handled by ESQLite rather then natively. $TYPE\_XPath$ tells ESQlite that an XPath query has to be evaluated. Both tag types are invoked by typing $json()$ within the SQL statement.

Tag	Description
$TYPE\_JSON$	a tag containing JSON.
$TYPE\_XPATH$	a tag containing a XPath query descriptor

Table 4.2: Overview of available JSON tag types in ESQLite

4.3.2 XPath Querying

Since JSON is tree like and has close resemblances to XML, the proposed query language bares great resembles with the widely known XPath language [20]. XPath is the path traversal related subset of XQuery (XML Query Language), which in its turn is a language able to select and manipulate subsets and substructures from a set of XML files. There are several variations to XPath available which are able to handle JSON, for example; JsonPath and JAQL [22] [23]. JsonPath differs from XPath mainly on notation, which is aligned to JSON. JSONiq is based on XQuery. Since the XPath notation is widely known, and our approach has only need for simple path traversal and attribute selection functionality, we propose a query language inspired on XPath. First we define a set of selection expressions which can be used in Table 4.3.

We define complex path expressions in Table 4.4. Important to note is that XPath, and thus our

Expression	Description
nodename	Selects all nodes with the name 'nodename'
/	Selects from the root node
.	Selects the current node
@	Select attribute

Table 4.3: Overview of XPath expressions available in ESQLite

approach, supports all logic operators such as $<, >, =$.

Expression	Description
$//title[@lang]$	Selects all title elements that have an te named lang
$//title[@l =' en']$	Selects all title elements that have a 'l' attribute with a value of 'en'
$/bookstore/book[price > 35]$	Selects all book elements of the bookstore element that have p > 35

Table 4.4: Overview of XPath predicates available in ESQLite.

The notation as given in Table 4.3 and 4.4 gives us the tools needed to query the JSON data. In Section 4.4 we elaborate on the use of this query language in respect to the query engine in detail.

4.4 Query engine

The previous section elaborated on how JSON data can be efficiently stored and reconstructed, this section will further explain the role of the query engine. We do this by translating the required queries and use the previously introduced function F_{shred} to facilitate the transformation from JSON to relational storage. Since we aim to remain complete backwards comparability, this approach does not alter the query engine or query planner inner workings.

4.4.1 Shredding

To explain we use the following example query on a table *data* containing only one column called *json*. Suppose one would be interested in inserting a JSON document into the table, the programmer would use the SQL statement as shown in Listing 4.11

```
1 INSERT INTO data VALUES({k1:v1, k2:v2, .., km:vm});
```

Listing 4.11: Example INSERT query with JSON

Applying the function F_{shred} on the JSON part of the above query, we get the following result:

$$F_{shred}(\{k_1 : v_1, k_2 : v_2, .., k_n : v_n\}) = \{(1, 0, k_1, v_1), (2, 0, k_2, v_2), .., (n, 0, k_n, v_n)\} \tag{4.16}$$

For which each tuple in the tuple-set can be stored as a native SQL record in the shred table as shown in Listing 4.12.

```
1 INSERT INTO data VALUES(<identifier>);
2
3 INSERT INTO data_shred VALUES(1,0,k1,v1);
4 INSERT INTO data_shred VALUES(1,0,k2,v2);
5 ..
6 INSERT INTO data_shred VALUES(1,0,kn,vn);
```

Listing 4.12: Example INSERT query with shredded JSON

To ensure that the JSON tuples later can be restored and linked to the correct record, we store an identifier to the tuple-set root object.

4.4.2 Reconstruction

In this section, selection and reconstruction of JSON data will be discussed. Listing 4.14 shows an arbitrary SQL SELECT WHERE statement.

```
1 SELECT {k1, k2, .., km}
2 FROM R1,..,RN
3 WHERE <condition>
```

Listing 4.13: Example SELECT query

The statement can be expressed in the following relational algebra notation;

$$\Pi_{k_1,k_2,..,K_N}\sigma_{condition}(R_1 \times .. \times R_n) \quad \text{for n tables} \tag{4.17}$$

$$\Pi_{k_1,k_2,..,K_N}\sigma_{condition}(R_1) \quad \text{for 1 table} \tag{4.18}$$

For the ESQL statement in Listing 4.14, containing an XPath construct, the selection and reconstruction works slightly different. Note that for this example we eliminate the complexity of multiple tables.

```
1 SELECT {k1, k2, .., km}
2 FROM R1
3 WHERE <condition> AND json(\\json\[@K1=V1])
```

Listing 4.14: Example ESQL SELECT XPath query

The first step is to get all shreds which satisfy the XPath statement. Equation 4.19 returns a set of root ID's of documents matching the XPath query. For this example we used *root storage* and *path storage* optimizations as shown in Equation 4.14 and 4.15, to eliminate the number of tree traversals.

$$\Pi_{root}\sigma_{key=K1 \wedge value=V1 \wedge path=\backslash}(R1\_shred) \tag{4.19}$$

Then with the set of root ID's available, we will now use the inverse function of F_{shred} to reconstruct the JSON of the documents containing these root ID's.

$$F_{reconstruct}(ID) = F^{-1}_{shred} \tag{4.20}$$

Function 4.20 uses 4.21 to fetch al needed shreds and then performs the reconstruction. Since a reference to the root ID is stored in the original table, the reconstructed JSON can be linked to the correct record.

$$\Pi_{id,root,path,parent,key,value,type}\sigma_{shred.root=ID}(R_1) \tag{4.21}$$

If no *root storage* optimization is used, the algorithm is slightly more complicated, needing for m levels of nesting, m self joins as shown in 4.22.

$$\Pi_{id,root,path,parent,key,value,type}\sigma_{shred_0.parent=ID,s_1.parent=s_0.id,..,s_m.parent=s_{m-1}.id}(S_0 \times S_1 \times .. \times S_m) \tag{4.22}$$

This concludes our approach to the storage and reconstruction of JSON data. In the next chapter we will discuss our implementation in more detail

Chapter 5

Implementing ESQLite

In Chapter 4 an approach for handling JSON data by extending SQLite is developed and evaluated. Therefore it is now possible to implement the proposed solution as a library. This chapter outlines and discusses the implementation of ESQLite as a Java Android application. In Section 5.1 we first look at the native SQL implementation. We do this since it sheds light on how JSON storage is normally done, and provides a baseline for our evaluation in Chapter 7. In Section 5.2 ESQLite is proposed. At first we give a brief library outline and architecture overview in Section 5.2.1, followed by an example of the ESQLite query life-cycle in Section 5.2.2. Finally the chapter is concluded with a more detailed explanation of all the configuration options in Section 5.2.3. All configuration options are closely aligned to the proposed approaches in Chapter 4.

5.1 Native SQL

Before we discuss our implementation ESQLite we first take a brief look at the native SQL approach of storing JSON data. For the native implementation the JSON part of the data can be stored as *text* or *blob* column in the corresponding table of that item. The *text* type stores data as a string using the database's encoding. Java stores strings encoded as UTF-16 whereas default SQLite implementations use UTF-8. The database system can perform string functions such as comparisons and searches on *text* type columns. *Blob* stores the data as a Binary Large Object exactly as it was. A *blob* can be text, PDF, image or anything else. The database is not able to search, compare or manipulate *blob* values, but they can be inserted and selected very fast due to the absence of type and encoding translations. Ideally the data is encoded in the same format as received, for instance JSON or XML. In the native approach the query engine, storage engine and query language all remain unchanged, hence the name native.

To illustrate, the synthetic dataset as given in Listing 6.1 will be inserted as shown in listing 5.1

```
1 INSERT INTO Users (name, data) VALUES
2 ('administrator','[{"action":"login","datetime":"2015-10-01
3 10:00:01"},{"action":"logout","datetime":"2015-10-01 11:00:01"}]');
```

Listing 5.1: Example INSERT query with JSON data

5.2 ESQLite

ESQLite is written in JAVA SE7 on top of the Android Development platform. ESQLite is supported for mobile devices running Android 4.0 onwards. ESQLite does not require any additional packages libraries or bindings and can be easily included as an *Java library project*. ESQLite can

be freely downloaded from *Gitlab* [25]. In Section 2.2 more background details about Android and SQLite can be read.

Android SQLite engine Android comes with its own custom developed *Java Database Driver* under the name of *android.database.sqlite* [13]. It is also possible to use other Java Database client interfaces but this is strongly discouraged. If one would be require to use a custom driver, then this driver has to be shipped with the application. With respect to our usability condition, this is not desirable and we choose to use the android driver. The custom Android SQLite engine does not need any configuration and is available for use.

Query language extension For the framework to determine if JSON has to be processed, we extend the query language with XPath inspired $ESQLTags$. In section 4.3 this query language extension is discussed in more detail. Tags can be of type $JSON$ or $XPath$. Whereas the $JSON$ tag indicates the presence of JSON data in need of processing, the $XPath$ tag contains an XPath query. Since XPath querying is not natively supported by SQL, the use of such a tag is unavoidable. Listing 5.2 shows how to select all records of table *user* for which the column *data* contains the path */name* with key-value pair $first = edwin$.

```
1 SELECT * FROM user WHERE json(//data/name/[@first='edwin']);
```

Listing 5.2: Example ESQLTag Xpath

5.2.1 Architectural overview

The main component of ESQLite is the $ESQLiteDatabaseWrapper$ class. From a programmer perspective this class replaces the $SQLiteDatabase$ class references and provides all interfacing with ESQLite. Below in Listing 5.3 an example is given of how ESQLite compared to SQLite is initialized.

```
1 //normal SQLite
2 SQLiteDatabase db = this.openOrCreateDatabase(DB_PATH);
3 db.execSQL(sql);
4
5 //ESQLIte
6 ESQLiteDatabaseWrapper dbWrapper = ESQLiteDatabaseWrapper.openOrCreateDatabase(
      DB_PATH);
7 dbWrapper.execSQL(sql);
```

Listing 5.3: Example Usage of ESQLite

The library is backwards compatible, meaning that a programmer does not have to change any existing code or queries other than using the wrapper class. Since the wrapper class is a static, no instantiation is required.
ESQLite comes with several configuration options regarding shredding and reconstruction, these configuration options are the result of the different approaches as discussed in Chapter 4. We use a Builder pattern to enable programmers to easily tune the configuration to their needs, below in Listing 5.4 the builder pattern is shown. In the next section all configuration options will be explained.

```
1 ESQLiteDatabaseWrapper.Builder builder = new ESQLiteDatabaseWrapper.Builder()
2 .debug(true)
3 .shred_type(ShredType.UPDATE)
4 .reconstruct_type(ReconstructType.IN_CODE)
5 .store_base(true)
6 .store_path(truePATH)
7 .store_root(true);
8 ESQLiteDatabaseWrapper dbWrapper = ESQLiteDatabaseWrapper.openOrCreateDatabase(
      DB_PATH, builder);
```

Listing 5.4: Example Usage of ESQLite with configuration builder

The UML diagram in Figure 5.1 provides an architectural overview by listing all classes and relations within the ESQLite framework. Please note that all helper classes and functions as well all *get*() and *set*() functions are left out.
The main class of our framework is *ESQLiteDatabaseWrapper*. An instance of this class is created (i.e. *mainActivity*) in the Android application when a database connection is needed. The *ESQLiteDatabaseWrapper* processes and stores SQL queries as *ESQLQuery* objects. The ESQLQuery instance checks whether the query contains *ESQLTag* objects, indicating the presence of JSON functionality and if this tag is present stores these as *ESQLTag* objects. A given *ESQLiteDatabaseWrapper* instance handles zero or more *ESQLQueries* with zero or more *ESQLTags*. For a given insertion query, depending on the *ShredType* and *ReconstructType* configurations that are passed by the builder, the framework will then shred the JSON into *ESQLShreds*. These shreds will eventually be stored as tuples in a relational table. Configuration options concerning the storage of a *path*, *root* and *base* ensure these references are stored within the tuples and *ESQLRelationalBase* objects are instantiated. From a performance perspective the instances *ESQLBaseCache*, *ESQLShredCache* and *TempTableCache* are used to speed up the process. When a selection query is processed by the framework the reconstruction and XPath evaluation is done. In Figure 5.2 a more detailed overview of query reconstruction is provided. Figure 5.1 also outlines the test framework related classes and builder configurations which will be further discussed in Chapter 6.

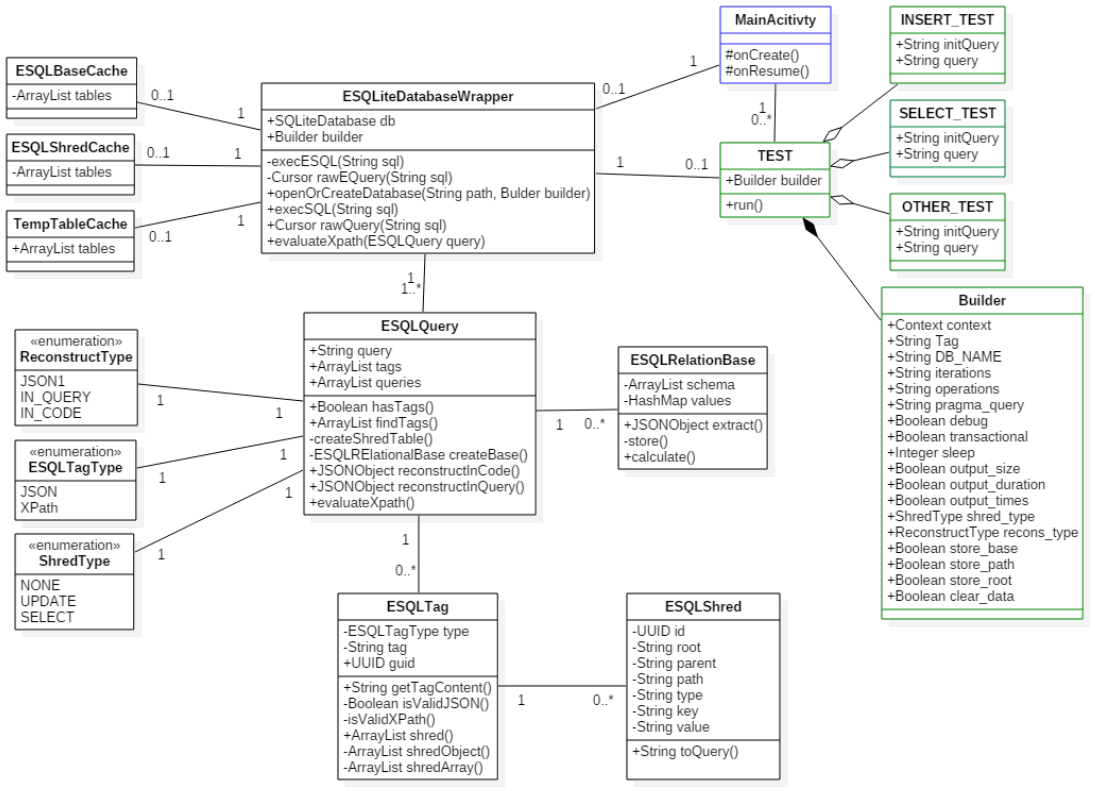


Figure 5.1: Architectural overview of the ESQLite library

5.2.2 ESQLite query lifecycle

In Figure 5.2 the ESQLite query life-cycle is shown. In this example a very basic JSON document is used with two key-value pairs. At insertion (step 1) the JSON string is replaced by an identifier

and inserted in the database (2). If it does not exists already a shred table is created (3) after which the shredded key-value tuples are stored in this table(4). If there is a relational base, e.g. the key *age*, that tuple is inserted in a relational base table(5b). If it does not exists the base table is created at that time (5a).
At selection time (6) the (potential) XPath tag is evaluated and the corresponding records are returned as a Cursor (7). If no XPath tag is present, the query is executed normally after which it is possible the Cursor detects that there is a JSON reference present. If so it fetches the corresponding shreds and base table records (9,11). In both cases the JSON has to be reconstructed (8) and passed back to the application logic(12).
This example is the most basic approach of ESQLite, several configurations and optimizations are available and discussed in the next section.

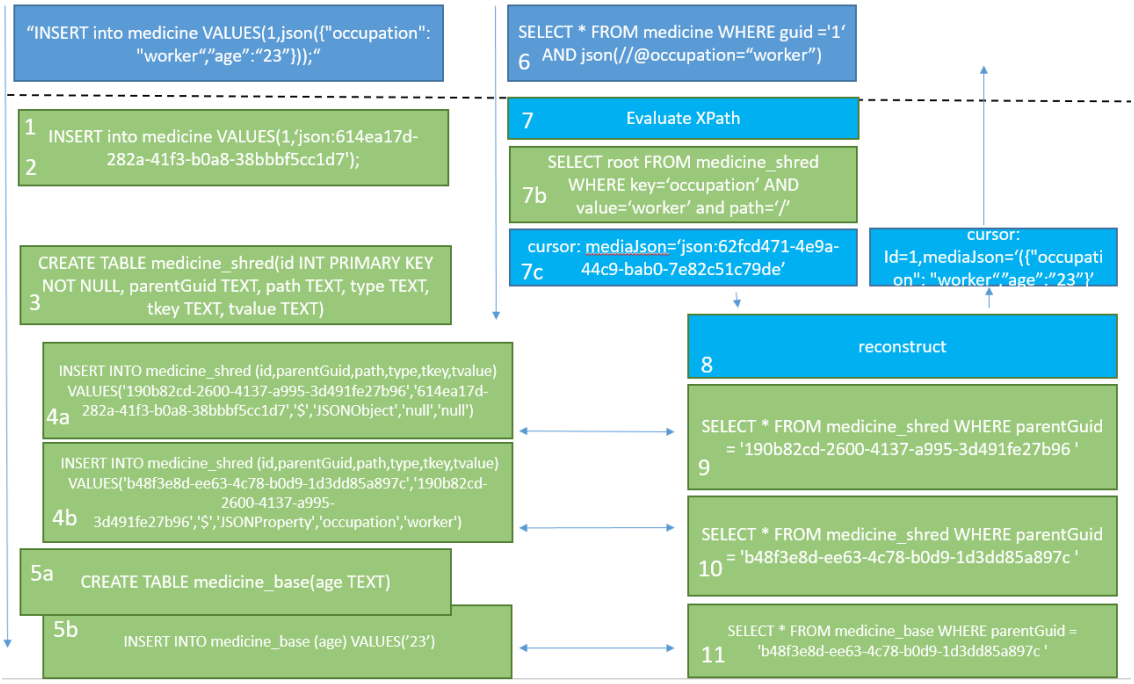


Figure 5.2: Example of the ESQLite query life cycle

5.2.3 ESQLite configurations

ESQLite comes with a set of configuration options which can be easily set at instantiation time as shown in Listing 5.4. Besides a debug logging option it is possible to determine the ShredType, reconstuctionType as well as with the base, path and root storage options. All configuration options are closely aligned to our approach chosen and are implemented not only from a usability perspective but will be used in Chapter 7 for evaluation.

ShredType The framework depicts three different shredding types. The *None* type handles JSON data as normal *text* in exactly the same way as described in Section 5.1. The *UPDATE* type shreds the data as described in the previous chapter at insertion time. Whereas this approach is relatively slow at insertion time, it is very fast at selection time. If the performance requirements are the other way round the option *SELECT* is available. This option also has less storage requirements since the shredding is done in temporary tables and dropped after querying. The latter option is not available in conjunction with relational base storage. A more detailed evaluation of the performance is done in Chapter 7.

ReconstructType For reconstructing the JSON from tuples there are three approaches available. The default configuration is IN_CODE which performs the actual JSON reconstruction within the wrapper application logic. The option IN_QUERY however does the complete reconstruction within the query itself. Since this approach works with nested self joins (see Section 4.2.3) it becomes very inefficient or even impossible on large JSON objects. If the JSON object at hand is rather simple, this solution turns out to be quite efficient since it eliminate coding overhead in reconstruction. The third option available is $JSON1$ which, inspired by the SQLite JSON1 extension, does only a partial reconstruction. $JSON1$ returns an Cartesian product of the relational table with all its corresponding shreds, leaving all querying to the programmer. This option can not be used in conjunction with XPath query expressions.

Root storage In its most basic form ESQLite needs to do a complete tree walk based on the shreds' parent pointers to reconstruct the JSON from all shreds. To speed up the process of reconstruction it is possible to store the root of each shred. Off course this introduces more storage overhead.

Path storage Analog to the above the XPath evaluation needs to do a tree walk in order to determine if a JSON key-value pair matches a given path. To speed up the process of querying it is possible to store the path of each shred. Off course this introduces more storage overhead.

Base storage When storing a large number of JSON documents, it is possible that all the documents have partially the same structure or schema. The *baseStorage* option extracts the shared relational base from each document and stores these properties as normal columns within a relational table. When documents have highly similar structures this increases the performance significantly and reduces the disk space needed (as shown in 7.5). Since in the scope of this project this implementation is rather naive, it performs bad when the schema changes over time.

Chapter 6

Experimental setup

In the previous Chapter, 5, different implementations where proposed. With the ESQLite Library encapsulating these implementation variations we can now focus on developing a testing environment and derive evaluation objectives from the main research objectives as stated in Chapter 1. The evaluation of our implementation is of vital importance to asses whether ESQLite can be proposed as a solution to the problem at hand. This chapter therefore presents the setup for our experimental evaluation of the proposed implementation. The chapter starts with an explanation of the desired environment in terms of hard- and software in Section 6.1. Then Section 6.2 states the evaluation objective followed by the designed set of queries needed to evaluate in Section 6.4 and datasets in Section 6.3. We conclude this chapter with a brief explanation of the testing platform in Section 6.5.

6.1 Environment

The focus of this research lies within mobile storage. For that reason it is of essence that the experiments are conducted on mobile devices or machines that mimic their specifications and behavior as closely as possible. The preliminary evaluation uses a virtual private machine running Linux, while the ESQLite benchmark and remainder of the evaluation are performed on a virtual Android device running in an emulator on top of a Windows host system. In the follow subsections the hardware and software which we used is discussed in more detail.

6.1.1 Hardware

The preliminary evaluation uses a virtual private machine running Linux. The machine has a single core 2.2Ghz CPU, 1024MB of RAM and 50GB of SSD storage. This setup closely matches a normal phone, while having the benefit of excluding a lot of device specific variables. The remainder of the evaluation uses a virtual Android device instance, so called emulator, running on a Windows 8 host machine. Using the emulator minimizes the dependencies on external factors compared to a real device. Exact specifications are outlined below.

Windows hostmachine

- : CPU: Intel Core i7-3770 3.40Ghz quadcore.
- : Memory: 16GB DDR3
- : Storage: Solid State Disk 128G. Read 480MB/s Write 200MB/s
- : OS: Windows 7

Linux virtual machine

- : CPU: Intel E5600 2.40Ghz singlecore.
- : Memory: 1GB DDR3
- : Storage: Solid State Disk 50GB (virtual file system). Read 9000MB/s Write 100MB/s
- : OS: Ubuntu LTS 14.04.3

Nexus 5 Android emulator

- : CPU: Emulated ARM processor.
- : Memory: 1.5GB
- : Storage: Solid State Disk 1GB (virtual file system).
- : OS: Android 6.0 MarshMellow x86 64bit

6.1.2 Software

The hostmachine uses a clean Windows 7 installation, whereas the emulator uses a clean Android 6.0 MarshMallow installation, MarshMallow is the current last stable release of Android. The *Android virtual device manager* is used to create and manage virtual machines, which runs on top of *Oracle Virtual Box*.

6.1.3 Android emulator

The Android emulator is an application that provides a virtual mobile device on which one can run Android applications. It runs a full Android system stack, down to the kernel level. The Android emulator mimics all of the hardware and software features of a *Nexus 5* mobile device. It provides a variety of navigation and control keys, as well as a screen in which the application is displayed. The emulator uses the host network adapter to emulate the devices network connectivity. Also, the emulator provides dynamic binary translation of device machine code to the OS and processor architecture of the host machine. For this evaluation an emulator is preferable over a real device since it is a more stable controlled environment.

6.1.4 SQLite Wrapper

The proposed ESQLite solution aims to provide an easy to use library for Android/Java on top of SQLite. A more detailed description of SQLite is provided in Chapter 2 whereas Chapter 5 elaborated on the ESQLite wrapper.

6.2 Objective

The experiments we want to conduct have to be closely aligned to the research objective as stated in Chapter 1. Furthermore the evaluation criteria which formed the basis of our approach as outlined in Section 4.1.2, have to be addressed closely. We primarily focus on runtime and disk space performance but also focus our boundary condition of usability. The objective of our evaluation therefore can be denoted as follows.

Measure the performance of ESQLite on different types of datasets in terms of runtime and disk space.

Our implementation has several optimizations and configurations, which all have some impact on the overall performance. Our evaluation objective can therefore be divided into the following sub-objectives.

1. What is the performance in terms of query runtime for different types of datasets.
2. What is the performance in terms of disk space usage for different types of datasets.
3. What is the runtime performance for different shredding alternatives.
4. What is the runtime performance for different reconstruction alternatives.
5. What is the performance compared to SQLite and JSON1.

In the next section we denote several datasets of interest, which will form the basis of our evaluation.

6.3 Datasets

To get a clear understanding of how our implementation performs, we first define datasets and queries on which it will be evaluated. We have to make sure that the datasets as chosen provide a solid assessment and address all of the in Section 6.2 mentioned objectives. For this we use two artificial datasets representing two extremes, as well two real world complex datasets.

Example set 1: Artificial In listing 6.1 below, an example document of our first artificial constructed dataset is shown. The actual dataset contains 100 or 1000 documents depending on the specific test. This set contains all the five JSON types introduced in 4.2.1, has limited nesting, and, since all top level keys are present in all documents, a large relational base. This set can therefore be used to verify the basic workings and performance of ESQLite.

```
{
"_id": "1",
"prefix": "a_",
"nestedArray1": [{
  "key1": "value1"
}],
"nestedObject1":{
  "key1": "value1"
},
"stringKey": "value1",
"intKey": 1,
"booleanKey": true,
"key4": "value4",
"key5": "value5",
"key6": "value6"
}
```

Listing 6.1: Example dataset 1: Artificial

Example set 2: Artificial The second dataset is artificially constructed as well, and also contains up to 1000 documents of the structure as shown in listing 6.2 below. In contrary to the first set, this one contains no relational base and has a highly nested structure. In conjunction with set 1 these sets cover both highly nested as non-nested sets with different key-value types. Therefore these two sets are used to benchmark the corner-cases of our solution.

```
{
  "_id": "1",
  "prefix": "a_",
  "nestedArray4": [{
    "nestedArray3": [{
      "nestedArray2": [{
        "nestedArray1": [{
        "key1": "value1"
        }]
      }]
```

```
11        }]
12    }]
13  }
```

Listing 6.2: Example dataset 2: Artificial

Example set 3: Medicine dataset In Section 1.1 we elaborate on a real-world example in the field of medicine. Since one of our stated boundary conditions is usability in real world situations, we use this dataset as one of our evaluation sets in Chapter 7. The set shown below in Listing 6.3 is a small subset of the actual dataset at hand, important to note is the hierarchy of medicine, products, articles and substances which provide both a combination of hierarchical and unstructured data.

```
1  {
2    "_id": "e7287b00-296f-11e5-bee2-cb50caea7737",
3    "name": "Paracetamol",
4    "strength": "360MG",
5    "form": "Zetpil",
6    "number": "16018958",
7    "amount": "12",
8    "RVG": "3843",
9    "driving_ability": false,
10   "atc": "N02BE01",
11   "products": [{
12     "brand": "Finimal",
13     "company": "Bayer",
14     "unit": "stuk",
15     "_id": "e728c030-296f-11e5-a4cd-7b07b8392b38",
16     "articles": [{
17       "supplier": "Umcg Ziekenhuisapotheek",
18       "obsolete": false,
19       "_id": "e7291ad0-296f-11e5-a1e6-3f5c8f5"
20       }]
21   }],
22   "substances": [{
23     "name": "Methylparahydroxybenzoaat",
24     "active": false,
25     "_id": "e729b5e0-296f-11e5-91f8-d7d09ebe06fc"
26   }, {
27     "name": "Propylparahydroxybenzoaat",
28     "active": false,
29     "_id": "e729ed60-296f-11e5-9235-bbf593d9302d"
30   }]
31 }
```

Listing 6.3: Example dataset 3: Medicine

Example set 4: Magic the Gathering This dataset contains meta-data of playing cards for the game *Magic the Gathering*. Whereas the previous dataset contains a highly hierarchical structure, with some non-relational parts, this set has a larger schema-less component, and has a lot of Array values. Each card has a wide variety of properties, all of which do not have to occur for each card. Below, in Listing 6.4 the JSON representation of one card within the dataset is given.

```
1  {
2    "layout": "normal",
3    "name": "Air Elemental",
4    "manaCost": "{3}{U}{U}",
5    "cmc": 5,
6    "colors": ["Blue"],
7    "type": "Creature \u2014 Elemental",
8    "types": ["Creature"],
9    "subtypes": ["Elemental"],
```

```
10     "imageName": "air elemental",
11     "printings": ["LEA", "LEB", "2ED", "CED", "CEI", "3ED", "4ED", "5ED", "PO2", "
           6ED", "S99", "BRB", "BTD", "7ED", "8ED", "9ED", "10E", "DD2", "M10", "DPA"
           , "ME4", "DD3_JVC"],
12     "legalities": [{
13       "format": "Commander",
14       "legality": "Legal"
15     }, {
16       "format": "Freeform",
17       "legality": "Legal"
18     }, {
19       "format": "Legacy",
20       "legality": "Legal"
21     }, {
22       "format": "Modern",
23       "legality": "Legal"
24     }, {
25       "format": "Prismatic",
26       "legality": "Legal"
27     }, {
28       "format": "Singleton 100",
29       "legality": "Legal"
30     }, {
31       "format": "Tribal Wars Legacy",
32       "legality": "Legal"
33     }, {
34       "format": "Vintage",
35       "legality": "Legal"
36     }],
37     "colorIdentity": ["U"]
38 }
```

Listing 6.4: Example dataset 4: MTG

6.4 Queries

Whereas the different datasets as depicted in the previous section influence the performance of our implementation, the queries performed on those sets are important as well. The type and complexity of queries determine how the SQLite query planner and ESQLite library will generate and execute a plan. To evaluate our implementation we use three basic queries which are derived from our approach in Section **??**. These queries encapsulate the insertion, selection and XPath query aspects, which make up the most vital operations of SQL. The insertion query QO as shown in Listing 6.5 is used to evaluate our approach as described in Section 4.2.2, selection queries Q1 and Q2 as listed in 6.6 and 6.7 are used to evaluate the in Section 4.2.3 proposed reconstruction methods. For all queries the corresponding table does not hold any keys or indexes. The queries abstract from implementation and will be used to evaluate both our implementation as well as regular SQL.

```
1 INSERT INTO <table> VALUES {set};
```

Listing 6.5: Q0: INSERT

```
1 SELECT * FROM <table>;
```

Listing 6.6: Q1: SELECT

```
1 SELECT * FROM <table> WHERE <XPath expression>
```

Listing 6.7: Q2: SELECT WHERE

The above queries are general. Depending on the dataset and test case, the *table* and *expression* is substituted. In Listing 6.8 Q2 is materialized for each of the datasets as discussed in the previous section.

```
dataset 1: SELECT * FROM data WHERE json('//json/[@stringKey=value1]')

dataset 2: SELECT * FROM data WHERE json('//json/nestedArray4/nestedArray3/
    nestedArray2/nestedArray1[@key1=value1]')

dataset 3: SELECT * FROM data WHERE json('//json/[@number=16018958]')

dataset 4: SELECT * FROM data WHERE json('//json/[@power=3]')
```

Listing 6.8: Q2: SELECT WHERE PER SET

For evaluation of native SQLite and JSON1 as done in 7.3 we need additional queries. Q3 as shown in Listing 6.9 and Q4 as shown in Listing 6.10, illustrate how JSON1 respectively native SQL could be used to mimic the behavior of Q2 as closely as possible, with the absence of XPath query language extension. For explanatory reasons these queries are materialized using dataset 3.

```
SELECT count (*) FROM allCards, json_each(allCards.json) WHERE json_each.key =
"power" AND json_each.value = "3";
```

Listing 6.9: Q3: SELECT WHERE

```
SELECT count(*) FROM allCards
WHERE json LIKE '%"power":"3"%';
```

Listing 6.10: Q4: SELECT WHERE

6.5 Testing framework

For the purpose of testing, the ESQLite wrapper framework as discussed in Chapter 5 is further extended to a full fledged testing framework. By use of a single Builder construct all necessarily parameters, variations and optimizations can be set as desired. A detailed architectural overview can be seen in Figure 5.1. Below an example of the testing framework Builder is given. Appendix A shows a more detailed example of the testing suite.

```
INSERT_TEST.Builder builder = new INSERT_TEST.Builder()
    .tag("ESQLWRAPPER")
    .dbName("test.db")
    .debug(false)
    .context(this)
    .operations(1000)
    .iterations(20)
    .pragma("PRAGMA synchronous = 0")
    .transactional(false)
    .sleep(1500)
    .store_relational_base(true)
    .store_path_of_shred(true)
    .store_root_of_shred(true)
    .reconstruction_type(ReconstructType.IN_CODE)
    .shredding_type(ShredType.UPDATE)
    .log_size(true)
    .clear_data(true)
    .log_dump_of_times(true)
    .log_iteration_duration(true);

INSERT_TEST insert_test = new INSERT_TEST(builder);
insert_test.start();
```

Listing 6.11: ESQlite testing framework initialization.

The testing suite defines a base *TEST* class which can be extended for each specific test. All tests are performed on a single non-blocking non UI-thread. Each test will run n operations in m iterations, for which all individual run times as well as average and disk space usage is stored, ensuring stable results.

Chapter 7

Experimental evaluation

Now that we have developed the ESQLite library, we can perform experiments to evaluate its effectiveness. We do this in accordance with our evaluation objectives as stated in Section 6.2. This means that we measure the performance of our implementation for three types of queries and four types of datasets. The runtime and disk space performance are the primary evaluation criteria, though we will also keep the query capabilities and usability in mind. At first, in Section 7.1, some preliminary results are shown which form the basis for further investigation. In Section 7.2 we show an extensive evaluation of ESQLite. After which we evaluate JSON1 in Section 7.3. However JSON1, is not part of our implementation, as it served as inspiration for our approach, and therefore is of interest. With these results at hand we do a final comparison between how ESQLite, JSON1 and native SQLite handle JSON data in Section 7.4. The chapter is concluded with a discussion of the results in Section 7.5.

For all tests in this chapter we use the testing framework as proposed in Section 6.5. All queries are executed n times in m runs, and all measurements given in this chapter are the average of these $n * m$ runs, to ensure stable results. The default is 20 runs of 100 queries each. Limitations in available disk space on the virtual Android environments restricted us to a maximum database size of 1GB.

7.1 Preliminary

In this section some preliminary results are discussed, providing a baseline for the rest of the evaluation and ensuring that optimal parameters are chosen to rule out as much background noise as possible.

Firstly, the default Android SQLite implementation is tested. For this we used the experimental setup as discussed in Section 6.1. Also, we benchmarked ESQLite with all JSON support disabled, acting as default SQLite but introducing the ESQLite library overhead.

Type	none	transactions	synchronized	both
SQL native TEXT type	5	10	135	139
SQL native BLOB type	5	7	128	130
ESQL native TEXT type	5	10	138	142
ESQL native BLOB type	5	9	133	133

Table 7.1: Query Q0: 100x1 records INSERT (different settings)

As can be seen in Table 7.1 the SQL *blob* storage type is about 8% faster than SQL *text* type,

which matches our expectations from Chapter 3. Furthermore it shows that full disk synchronization results in a factor 25 performance drawback, since a SQL write command has to wait for drive confirmation when a record is stored before proceeding. Forcing transactions for every query only has a limited effect on the performance.

From the above results we can conclude that synchronization has a large impact on performance whereas forced transactions does not. *Blob* storage gives a small performance increase compared to *text* but since *blob* storage does not provide any query possibility, it is in most cases not desired to use *blob* types. To minimize noise we put both parameters on false for all coming tests.

7.2 Benchmark ESQLite

In this section we benchmark the ESQLite wrapper, our implementation as discussed in Chapter 5. First we discuss the runtime performance in Section 7.2.1 followed by the disk space performance in Section 7.2.2. While these two evaluation dimensions provide us with a good understanding of the ESQLite performance, we take a more detailed look at the disk space saving *selection time shredding* in Section 7.2.3 and two reconstruction alternatives in Section 7.2.4. We introduced some implementation configurations in Section 5.2.3, and we first take quick look at their performance impact in terms of runtime. This section is concluded with some scalability evaluation results in Section 7.2.5, providing further decision support.

All benchmarks are performed with the three queries types we defined in Section 6.4, and the four datasets as defined in Section 6.3. To be able to obtain valid results we need to minimize environment parameters. Therefore the following parameters are fixed; synchronization is turned off, SQLite does not wait for the disk to callback a successful write. Default SQLite transaction settings are used. No indexes or primary keys are used.

7.2.1 Runtime performance

For evaluating the runtime performance we observe three different dimensions; implementation, query types and datasets. The implementation dimension consists of three implementations; Native SQLite, our basic ESQLite implementation and finally our fully fledged SQLite implementation. Whereas the basic ESQLite implementation does not have root, path and relational base storage optimizations (discussed in Chapter 4), the ESQLite full implementation does. The second dimension of interest is about the query types as denoted in Section 6.4; insertion, selection and XPath type queries. The final dimension we measure is the datasets. We use four datasets, two synthetic and two real world sets. In the figures below we show native SQLite, basic ESQLite and the ESQLite full configuration for all datasets. All running times shown are per 100 insertions or selections.

Set 1 From our evaluation as shown in Figure 7.1 the following conclusions can be derived. The native approach is about three 30 times faster for inserts, and 8 times faster for selection queries compared to ESQLite. This clearly indicates the added costs of enabling JSON XPath query capabilities. Later in Section 7.2.3 we evaluate the *selection time shredding* improvement, to eliminate these costs.
We observe that the native implementation is not able to perform query Q2. Since Q2 is our XPath tag containing query, this is expected behavior since SQL does not have any JSON query capabilities. We also see that the full ESQLite implementation performs about 25% faster for Q0, about 10% faster for Q1 and 50% as fast for Q2 . These results are conform what we expect. Dataset 1 has limited nesting and a large relational base which in our full implementation will be stored relational. We therefore see an increase in performance for insertion query Q0 since less shreds have to be stored. We can conclude that the path and root storage functionality really shows its added value for Q2 on dataset 1, since the XPath evaluation (tree walk) can be done

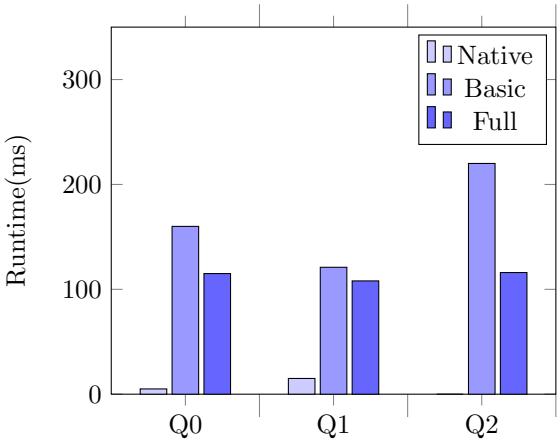


Figure 7.1: Runtime comparison - Set 1

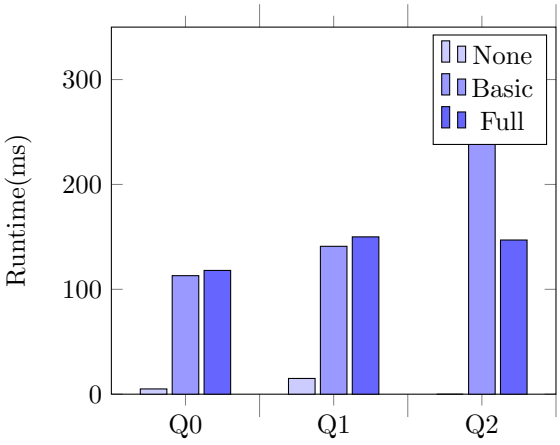


Figure 7.2: Runtime comparison - Set 2

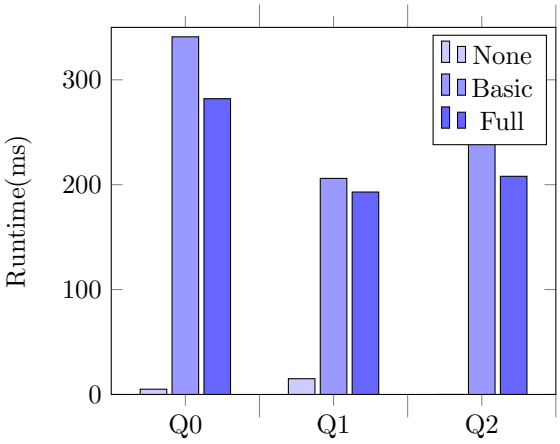


Figure 7.3: Runtime comparison - Set 3

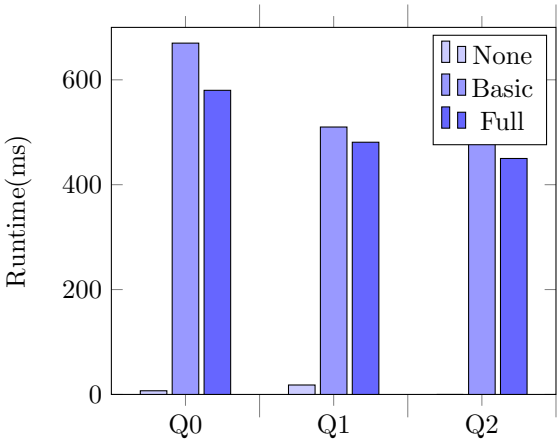


Figure 7.4: Runtime comparison - Set 4

much more faster. In this test all 100 records are in the result set of Q2. When the query only returns 1 out of 100 records, Q2 takes only 83ms, since less JSON objects have to be reconstructed. From this we can conclude that the XPath evaluation is 37% of the runtime, whereas 63% of the runtime costs is concerning the reconstruction. In Section 4.2.4 we elaborated on this in detail.

Set 2 In Figure 7.2 we benchmark our solutions for the second dataset. This dataset is highly nested and has no relational base. Equivalent to the previous results of dataset 1, we see that ESQLite is 10 times slower for insertion than the native approach. Again we see that the native approach is not capable of performing Q2, for the same reasons as mentioned in the previous set. The performance of ESQLite differs however for this dataset compared to the previous, for query Q1 and Q2. It is clearly visible that the full ESQLite implementation does not benefit from relational base storage in Q1 (as there is none in this set) which can be deduced from both having roughly the same runtime. However we can conclude that for Q2 our full approach still benefits from path and root storage, realizing a factor two improvement in speed. Again in this test the result set of the Q2 XPath query gives all 100 records as result.

Set 3 The previous two datasets contained synthetic data. The set used in this evaluation (Figure 7.3) is a real-world medicine dataset. In this evaluation we are interested how our implementation performs on real world data. We see that ESQLite becomes even more slower then the native approach, with run times up to 60 times that of the native approach. However, we also see that both our implementations are able to execute Q2 in under 250ms per 100 records, which, given the fact that this dataset is larger then the previous two, is within expectation. Finally we can conclude that our full ESQLite implementation still offers an improvement over our basic approach, whereas in the previous tests this improvement was up to 50% it is now limited to 18%. This, because the relative fraction of relational base values compared to the whole document is rather small for this set, and the JSON is highly nested.

Set 4 Whereas the previous dataset is highly nested, the fourth dataset as tested above in Figure 7.4 has a lot of array typed values. For this query we again see that ESQLite is up to 100 times slower than native SQL, and our full implementation performs 10% to 20% better then our basic implementation. Across the board the ESQLite implementation performs about twice as slow for this set in comparison to the previous set which contains roughly the same amount of key-value pairs.

Set comparison In Figure 7.5 we finalize this section by comparing the full ESQLite solution for all datasets as tested above. We see that ESQLite performs best on dataset 1 whereas dataset 4 has by far the highest runtime. We also see that the relative performance per query is the same for each dataset. We can conclude that ESQLite thus performs best on a JSON with a large relational base and a limited amount of nested arrays.

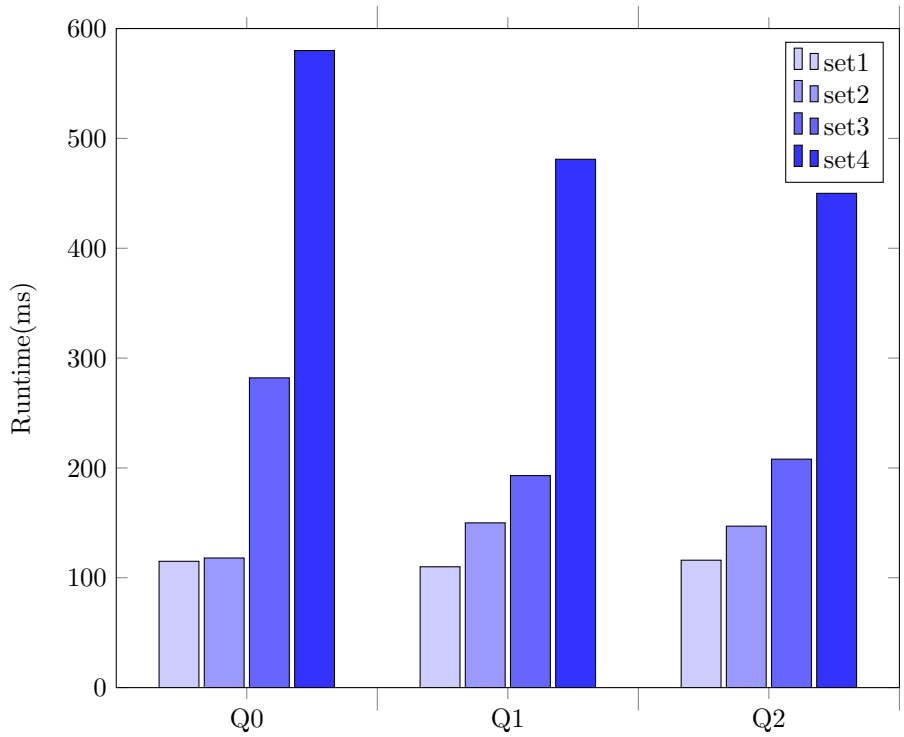


Figure 7.5: Runtime comparison Q0, Q1 and Q2 for all sets

7.2.2 Disk space performance

In Figure 7.6 we compare the disk space usage per solution. We do this only for one dataset since only the relative difference is of interest. Whereas all previous tests we only looked at the basic ESQLite versus full ESQLite implementation we now explicitly also look at the configuration options individually, to derive which option has what effect on the disk size. From Figure 7.6 we see that our ESQLite shredding increases the disk space usage by a factor 7.8, which is quite substantial. If we store the relational base separate we see that half of this increase can be recovered (for this dataset). Furthermore we see that the root storage has only a marginal effect whereas path storage costs about 20% extra storage space. We can conclude that the full ESQLite implementation thus costs 4.8 times the disk size compare to native storage. To overcome this storage burden one could use selection time shredding, which is benchmarked in the following section.

7.2.3 Selection time shredding

In Figure 7.7 we compare the runtime performance for the configuration options; selection-time and update-time shredding. Selection time shredding is explained in Section 5.2.3. Please note that the results of the *update* series are the same as in the previous section, and for both we use the full ESQLite implementation. We use dataset 1 for this test.

We immediately see the increased performance for the insertion query Q0 and decline for selection query Q2 when using selection time shredding. We also see a performance increase by factor 8 for Q1. This can be explained since there is no need to do selection time shredding if no XPath query is involved. We can conclude that one could therefore choose to use selection time shredding if insertion and simple selection queries have to be fast. Furthermore insertion time shredding has only a temporary need for additional storage, since shredding is performed in temporary tables.

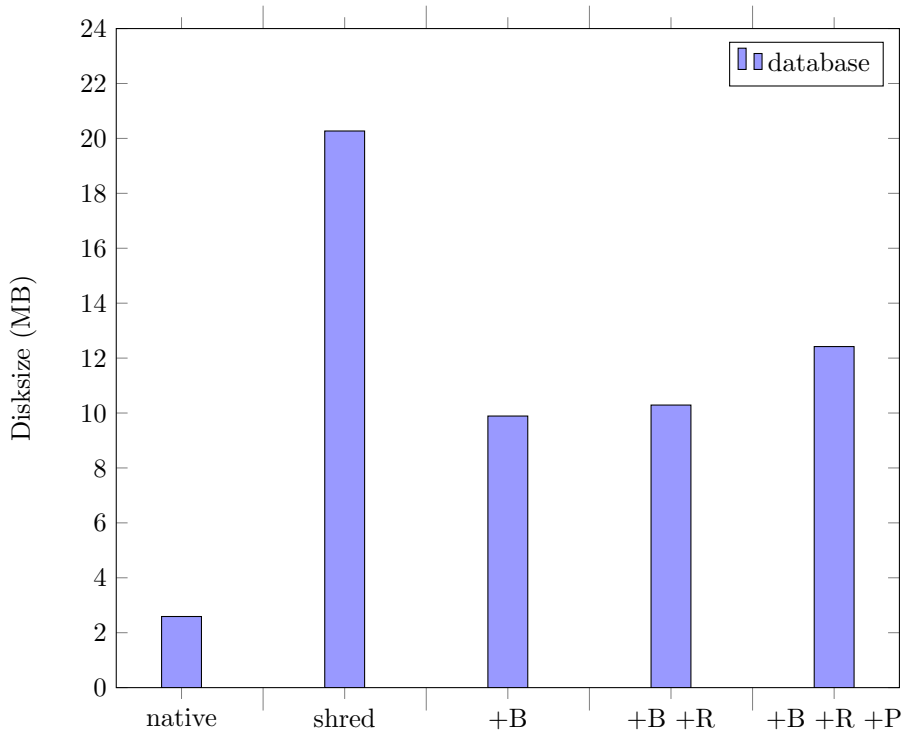


Figure 7.6: Disksize ESQLite implementations

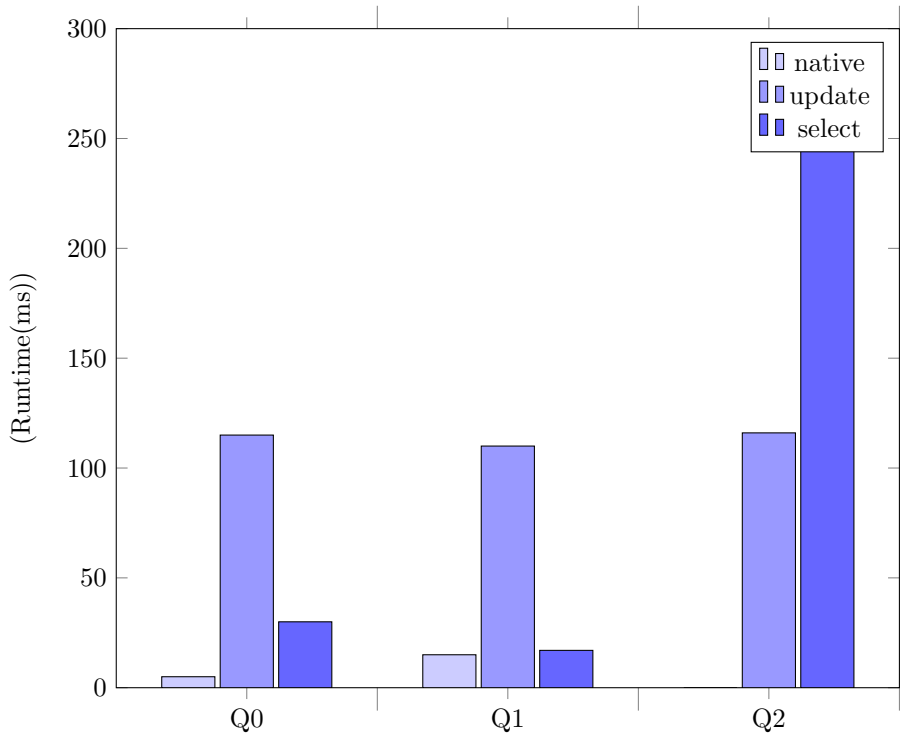


Figure 7.7: Update- vs selection time shredding

7.2.4 Reconstruction alternative

In Figure 7.8 as shown below we compare runtime for JSON reconstruction with the configuration options *in_code* and *in_query*, both of which are explained in Section 5.2.3 of our implementation. Please note that the results of the *update* series are those of the previous section. For this benchmark we use the full ESQLite implementation and dataset 1. Before we can draw any

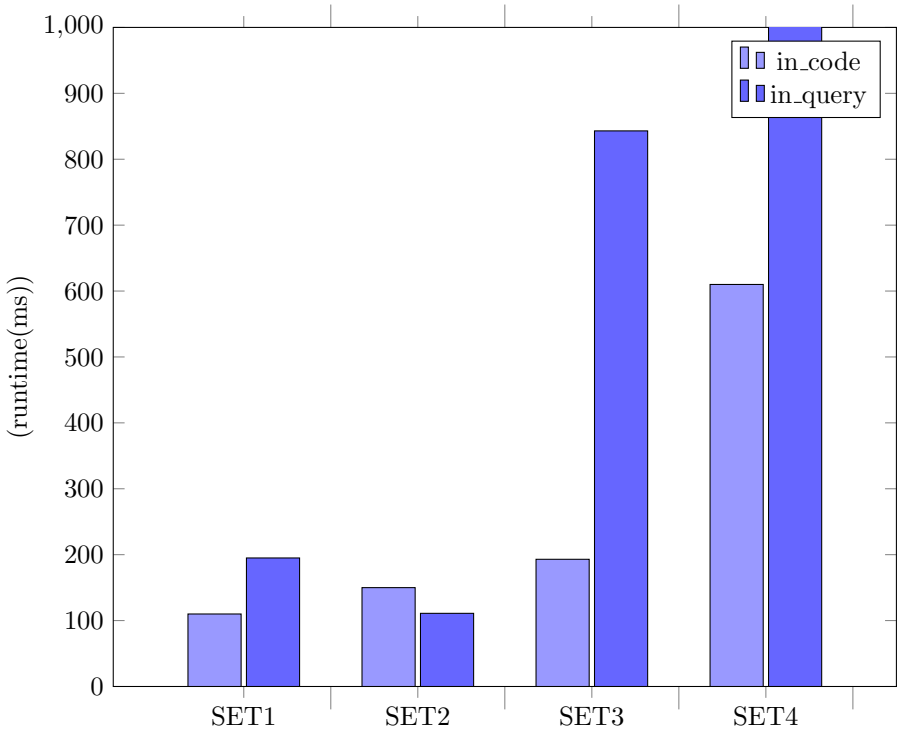


Figure 7.8: In_code vs in_query reconstruction

conclusions from the above results, please note that this is only the *in_query* selection of all necessary key-value pairs, the actual tree reconstruction of the query based on the cursor results still has to be done in code.
The first observation is that for set 4 *in_query* reconstruction is completely out of scope, since its runtime is 5530ms. We see that for set 1 both reconstruction methods are comparable, but taking into account the remaining reconstruction work for *in_query* we can conclude that set 1 is faster. For set 2 *in_query* reconstruction performs slightly better, but for the real world sets 3 and 4, *in_query* reconstruction becomes tremendously slow. From the above we can conclude that there is limited ground for using this type of reconstruction, other than for very simple JSON data.

7.2.5 Scalability

For all previous evaluations we inserted or selected 100 queries. In this section we do a brief evaluation of how scalable ESQLite is. In Figure 7.9 the runtime for the XPath query Q2 is plotted against the number of records in the result set. We see that the runtime increases more than linear, which one would expect form a normal SQL selection query. When we try to approximate the runtime we find a function of the following form.

$$Runtime(n) = (an) * (b^2 logn) \tag{7.1}$$

This can be explained since, for every selection we also need to do a tree reconstruction. The tree reconstruction takes *logn* since we have to perform a query for each level of each subtree. Due

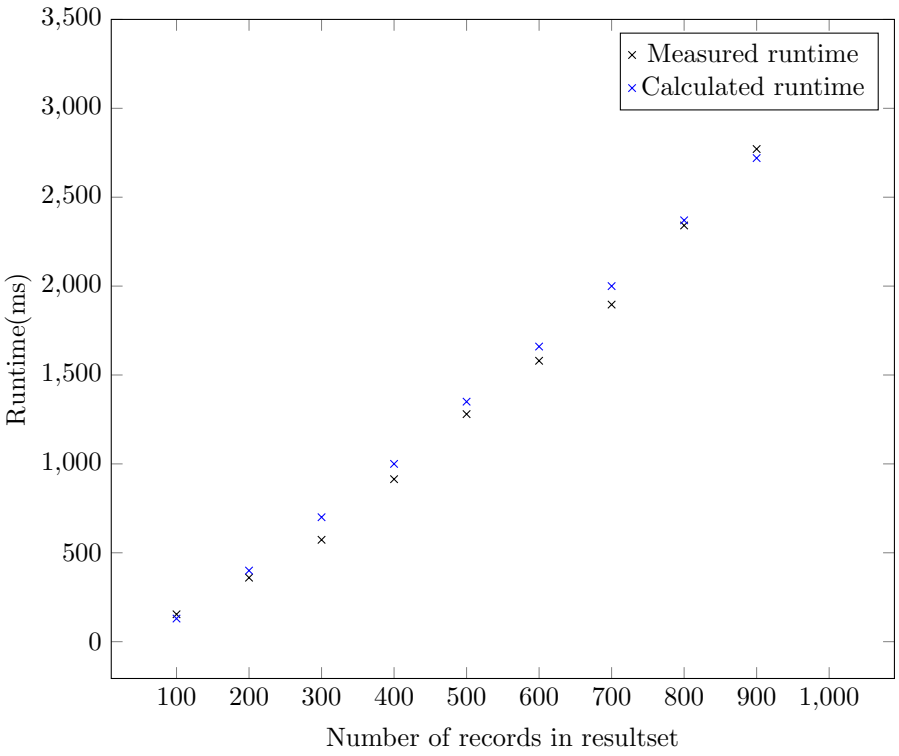


Figure 7.9: Q3: Runtime SELECT WHERE increased number of records in resultset.

to implementation limitations it is not possible to perform an ESQL query for which te expected result set is larger than 1000.

7.3 Benchmark JSON1

JSON1 is briefly but explicitly benchmarked, since it forms the inspiration of this project. As can be read in Chapter 3 it is the only mobile database system having any support for JSON queries. In Section 3.2.4 we investigated JSON1 as related work in our implementation. Since JSON1 is not available for android, we benchmark SQLite and JSON1 both on the Linux setup as introduced in Section 6.1. For all tests we used dataset 1. Since JSON1 does only return the shredded JSON and leaves the *where* evaluation to the user, it is not able to perform Q2.

Type	Native	JSON1
Q0	1.3	1.4
Q1	6.3	50
Q2	n.a	n.a.

Table 7.2: Runtime(ms) JSON1 Set 3 100 records

Runtime performance Table 7.2 shows the runtime of JSON1 compared to Native SQLite. Since JSON1 does the shredding on runtime, we see fast insertions for query Q0. For the selection query Q1 we see that *json_each* is about 10 times slower, since it has to output a Cartesian product of the JSON with all its key-value pairs.

Type	Native text	JSON1
set 1	7	7
set 2	10.3	10.3

Table 7.3: Disk space usage JSON1 (MB)

Disk space performance From a disk space usage perspective, we see that JSON1 has the same footprint as normal SQLite, this can be explained since SQLite does the shredding in memory when a selection query is executed.

7.4 ESQLite compared to SQLite and JSON1

In this section we compare the performance of SQLite, ESQLite and JSON1. Since SQLite and JSON1 cannot evaluate XPath queries, these results are left out. We perform this evaluation on dataset 1.

Since we can only run JSON1 on our Linux test setup, and ESQLite on the Android testing framework, in order to compare both, we first determine what the runtime on both setups is for SQLite. In Figure 7.1 we see that for dataset 1 Q0 took 5ms and Q1 15ms, whereas in this setup Q0 took 1.3ms and Q1 took 6.3ms. From this we conclude that our Android setup introduces roughly a factor 3 performance decrease. Despite this limited proxy, it gives us the order of magnitude to take into account. From Table 7.4 we conclude our evaluation with a comparison of native SQLite,

Type	Q0	Q1	Q2
Native(ms)	5	15	n.a.
JSON1(ms)	4	150	n.a.
ESQLite(ms)	115	108	116

Table 7.4: Runtime comparison SQLite, ESQLite and JSON1

ESQLite and JSON1. We confirm our earlier results that both ESQLite and JSON1 are about 10 times slower for selection queries compared to native SQLite. Both however are not able to perform XPath query Q2. The two differ in that JSON1 performs the same as SQLite for insertion query Q1, since it only does shredding on selection time. The main conclusion we can derive from the above is that for dataset 1, JSON1 is slightly slower than ESQLite. When we also take into account that JSON1 results are only a Cartesian product rather than the constructed JSON as delivered by ESQLite we can safely conclude that ESQLite outperform JSON1. Important to note is that the performance conversion factor between our two test setups is taken into account. Since dataset 1 is rather simple compared to our real-world datasets 3 and 4, we expect that the difference will be significantly larger for these sets, but that evaluation is out of the scope of this research.

7.5 Discussion

In this chapter we evaluated our ESQLite implementation using the experimental setup as outlined from the previous chapter. In Section 6.2 we stated the following evaluation objective; *Measure the performance of ESQLite on different types of datasets in terms of runtime and disk space.* We divide this objective into five sub-objectives regarding; the runtime and disk space performance, evaluation of our shredding and reconstruction alternatives and the performance of ESQLite against JSON1 and native ESQLite.

The evaluation of the first sub-objective; *what is the performance in terms of query runtime for different types of datasets*, leads to the following conclusions. The runtime performance of ESQLite for insertion queries turned out to be 20 to 100 times slower than native SQLite, depending on the complexity and level of nesting of the JSON data. For selection type queries the runtime is in the range of 8 to 20 times slower. The added costs however enable ESQLite to reason over JSON with XPath queries, which native SQLite is not capable of. Furthermore ESQLite does not effect runtime for data which does not contain JSON. We determined that the XPath evaluation took 83ms (per 100 records), whereas the complete XPath query costs 286ms (per 100 records) for dataset 1, and up to 588ms (per 100 records) for dataset 4.
Most interesting is the increase in performance between the basic ESQLite implementation compared to our fully fledged implementation. We see, that depending on the dataset, this results in an 8% to 50% runtime performance increase. ESQLite optimizations turn out to have great effect on limited nested datasets and a large relational base.

Regarding the second sub-objective; *What is the performance in terms of disk space usage for different types of datasets*, we conclude that the basic ESQLite implementation has a 7.8 times larger disk space need than native SQLite, whereas a 4.8 times larger footprint for our optimized implementation.

ESQLite also offers selection time shredding. The results to our corresponding third sub-objective; *What is the runtime performance for different shredding alternatives*, lead to the following conclusions. Selection time shredding can be useful when insertion performance and/or disk space is important. Selection time shredding partially redeems the added runtime costs of ESQLite. Furthermore we showed that selection time shredding is fast when no XPath evaluation has to be done, since shredding is then completely skipped.

What is the runtime performance for different reconstruction alternatives, is the fourth sub-objective of interest. Looking at the ESQLite *in_query* and *in_code* reconstruction alternatives we conclude that both perform equally on dataset 1 and 2, but *in_code* performs significantly better on the real world datasets, and therefore will be preferable in most cases.

The final sub-objective of our evaluation is stated as follows; *what is the performance compared to SQLite and JSON1.* JSON1 is benchmarked and ESQLite is compared to both JSON1 as native ESQLite. We again conclude that both ESQLite as well as JSON1 are about 10 times slower for selection queries compared to native SQLite and we also concluded that ESQLite performs better than JSON1 on the selected queries and datasets.

The conclusion to our main evaluation objective as denoted above can therefore be summarized as follows; while ESQLite turns out to be a order of magnitude slower than native SQLite, it is able to handle JSON data with a reasonable performance. In any case, for both synthetic as well as real-world datasets, ESQLite is faster than the current available JSON1.

Chapter 8

Conclusions

This thesis introduces the problem of handling partially non-relational data on lightweight clients such as mobile devices. Whereas this type of data increasingly emerges and new techniques for coping with this data are developed, the mobile field seems to fall behind. In this chapter we summarize our contributions and underpin these with the evaluation results as obtained. Afterwards in Section 8.2 we look at ESQLite's limitations and what future work has to be done.

8.1 Summary of contributions

The current dominant database system for mobile devices is the relational database SQLite. The main contribution of this thesis is the design and development of ESQLite, a wrapper library on top of SQLite, enabling it to store and reason over JSON data. ESQLite shreds JSON into key-value pairs and stores them as tuples in a relational table, then on selection the JSON is reconstructed. This approach not only gives us a way of storing the data relational, but allows for extensive querying on the JSON data. A XPath like query language extension is proposed to perform this querying. Several variations and optimizations to ESQLite are discussed and implemented. ESQLite is backwards compatible and easy to use, which besides performance requirements was an explicit boundary condition for our implementation.

We then set up an evaluation framework with the objective of verifying the performance of ESQLite on different types of real-world data. We conclude that while ESQLite is an order of magnitude slower then native SQLite, these additional costs enable us to query efficiently over JSON data. Important to note is that ESQLite does not affect performance when JSON data is not present. Evaluating a single JSON XPath query takes 83mS and the selection and evaluation of 100 real-world medicine records takes 203ms, which turns out to be reasonably fast and significantly faster then the current available JSON1. We improved our basic ESQLite solution to store the relational base of a JSON document in a table, store a root pointer and store the path of each tuple to further improve the reconstruction runtime. We showed that the these improvements boost the performance with 8% to 50% depending on the dataset. Since ESQLite has a 4.8 times larger disk space requirement then native SQLite, we also introduced a *selection time* shredding option. This option uses temporary memory for shredding and has no additional storage needs.

We address both storage and reasoning of JSON data, and proved to do this in an efficient matter, while not effecting the normal SQL performance. So it is safe to say that ESQLite answers our research question: *How and in what way can we extend SQLite, so that it is optimal equipped to store and reason over JSON data.*

8.2 Limitations and future work

The proposed, implemented and evaluated framework is merely a prototype and offers perspective for further investigation. There remains a lot to be done to further fill the void currently present in the field of JSON storage. Below some of the most interesting limitations and future work proposals are briefly discussed.

Dynamic relational base extraction The current implementation can only observe simple key/value pairs and cannot evaluate arrays or nested objects. Also the current implementation is fairly naive and performs poor when the relational base is calculated suboptimal.

Topological sorting and reconstruction With the use of topological sorting algorithms it is possible to further improve the reconstruction runtime. There are several different approaches to this, the easiest is storing the level of a node.

Dynamic shredding In the current implementation shredding is done for every document. When for instance n exactly the same documents are inserted to ESQLite, one would have $n - 1$ duplicate tuples in the database, effecting runtime and disk space need. An optimization could be to store multiple parent pointers when a tuple has multiple occurrences. Note that this has a negative effect on the topological sorting as suggested above. In Section 4.2.4 we briefly discussed this topic.

Indexing However some of the related work showed interesting results regarding indexing techniques, this was out of scope for this project.

Query language extension The current XPath query language extension can evaluate simple expressions but has no support for more complex aggregation functions.

Bibliography

[1] Edgar F. Codd. *A relational model of data for large shared data banks.* Communications of the ACM, Volume 13 Issue 6, June 1970, Pages 377-387 7

[2] Donald D. Chamberlin, Raymond F. Boyce. *A structured English Query Language.* SIGMOD Workshop, Vol. 1 1974: 249-264 1

[3] Han, J., et al. (2011, October). Survey on NoSQL database. In Pervasive computing and applications (ICPCA), 2011 6th international conference on (pp. 363-366). IEEE. 7

[4] Chasseur, Craig, Yinan Li, and Jignesh M. Patel. 'Enabling JSON Document Stores in Relational Systems.' WebDB. 2013. 13, 14, 15

[5] Liu, Zhen Hua, Beda Hammerschmidt, and Doug McMahon. 'JSON data management: supporting schema-less development in RDBMS.' Proceedings of the 2014 ACM SIGMOD international conference on Management of data. ACM, 2014. 15

[6] Bohannon, Philip, et al. 'LegoDB: Customizing relational storage for XML documents.' Proceedings of the 28th international conference on Very Large Data Bases. VLDB Endowment, 2002. 13

[7] Beckmann, Jennifer L., et al. 'Extending RDBMSs to support sparse datasets using an interpreted attribute storage format.' Data Engineering, 2006. ICDE'06. Proceedings of the 22nd International Conference. IEEE, 2006. APA 14

[8] Roijackers, John, and George HL Fletcher. 'On bridging relational and document-centric data stores.' Big Data. Springer Berlin Heidelberg, 2013. 135-148. 25

[9] Gray, Jim, and Andreas Reuter. Transaction processing: concepts and techniques. Elsevier, 1992. 7

[10] Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber. Bigtable: A Distributed Storage System for Structured Data. ACM Transactions on Computer Systems, Volume 26 Issue 2, June 2008 8

[11] 'Android Anatomy and Physiology' (PDF). Google I/O. May 28, 2008. Retrieved May 23, 2014. ix, 10

[12] https://www.sqlite.org ix, 9, 10

[13] http://developer.android.com/reference/android/database/sqlite/package-summary.html 32

[14] Crockford, Douglas (May 28, 2009). 'Introducing JSON'. json.org. Retrieved July 3, 2009. 11

[15] http://www.postgresql.org/docs/9.4/static/datatype-json.html 14

[16] https://dev.mysql.com/doc/refman/5.7/en/json.html 14

[17] https://github.com/aheinze/mongo-lite 14, 18

[18] https://gist.github.com/maxpert/5341461 14, 18

[19] http://www.creapptives.com/post/47494540287/a-document-store-with-sqlite3-and-python 14

[20] http://www.w3schools.com/xsl/xpath-syntax.asp 16, 28

[21] https://github.com/nodesocket/jsonlite 18

[22] https://github.com/jayway/JsonPath 16, 28

[23] http://jsoniq.org/ 28

[24] https://code.google.com/archive/p/jaql/

[25] https://gitlab.com/edwinhermkens/esqlite 4, 32

Appendix A

Code examples of ESQLite Testing framework

```
protected void onResume() {
  super.onResume();
  INSERT_TEST.Builder builder = new INSERT_TEST.Builder()
     .tag("ESQLWRAPPER")
     .dbName("test.db")
     .debug(false)
     .context(this)
     .operations(100)
     .iterations(1)
     .pragma("PRAGMA synchronous = 0")
     .transactional(false)
     .sleep(1500)
     .store_root_of_shred(true)
     .store_path_of_shred(true)
     .store_relational_base(true)
     .reconstruction_type(ReconstructType.IN_QUERY)
     .shredding_type(ShredType.UPDATE)
     .log_size(true)
     .clear_data(true)
     .log_dump_of_times(true)
     .log_iteration_duration(true);

  INSERT_TEST insert_test = new INSERT_TEST(builder);
  insert_test.start();
  //other tests are started from the insert test thread.
}
```

Listing A.1: Code example ESQLite testing framework initialization

```
public class INSERT_TEST extends TEST {

  public INSERT_TEST(Builder builder) {
    super(builder.context, builder);
  }

  @Override
  public void run() {
    String DB_PATH = "/data/data/" + context.getPackageName() + "/" + DB_BAME;
    ESQLiteDatabaseWrapper.Builder wrapperBuilder = new ESQLiteDatabaseWrapper.
        Builder()
      .debug(DEBUG)
      .shred_type(SHRED_TYPE)
      .reconstruct_type(RECONSTRUCT_TYPE)
      .store_base(STORE_BASE)
      .store_path(STORE_PATH)
```

```
16          .store_root(STORE_ROOT);
17        ESQLiteWrapper = ESQLiteDatabaseWrapper.openOrCreateDatabase(DB_PATH, null,
              null, wrapperBuilder);
18        try {
19          if (sleep > 0) Thread.sleep(3000);
20          if (OUTPUT_SIZE) Log.e("SIZE", getDBSize(DB_BAME));
21          ESQLiteWrapper.execSQL(INIT_QUERY);
22          if (PRAGMA_QUERY != null) {
23            ESQLiteWrapper.execSQL(PRAGMA_QUERY);
24          }
25          for (int i = 1; i < ITERATIONS; i++) {
26          //INSERT_QUERY = INSERT_QUERY.replace("key"+(i-1), "key" + i);
27            if (OUTPUT_ITERATION_DURATION)
28              Timer.get().start(OPERATIONS + " INSERTED " + i + "/" + (ITERATIONS -
                    1) + " RUNS");
29            for (int j = 0; j < OPERATIONS; j++) {
30              if (TRANSACTIONAL)
31                ESQLiteWrapper.execSQL("BEGIN TRANSACTION");
32              ESQLiteWrapper.execSQL(INSERT_QUERY);
33              if (TRANSACTIONAL)
34                ESQLiteWrapper.execSQL("COMMIT");
35            }
36            if (OUTPUT_ITERATION_DURATION) Timer.get().stop();
37            if (OUTPUT_SIZE) Log.e("SIZE", getDBSize(DB_BAME));
38            }
39          if (OUTPUT_DUMP_TIMES) Timer.get().dumpTimes();
40        } catch (Exception e) {
41          Log.e("TIMER", "ERROR ");
42          e.printStackTrace();
43        }
44        //run second test
45        builder.operations(1);
46        builder.iterations(20);
47        builder.log_size(false);
48        builder.clear_data(false);
49
50        SELECT_TEST select_test = new SELECT_TEST(builder);
51        select_test.start();
52        }
53      }
54  }
```

Listing A.2: Code example ESQLite testing framework