UNIVERSITÀ DI PISA

SCUOLA DI INGEGNERIA

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Master of Science in Computer Engineering

THESIS

Analysis, design and implementation of alternative solutions for data persistence on mobile devices

Supervisors:
Prof. Alessio VECCHIO
Prof. Marco AVVENUTI
Ing. Stefano ZINGARINI

Candidate:
Francesco PIRAS

Academic Year 2015/2016

Abstract

Data availability on mobile applications is a key feature required to have a better user experience, especially in offline scenarios. Mobile applications should be available both when the device is online and when it is offline. The assumption that the device is always connected to the Internet is in general too strict. Relational databases provide a solid but limited solution to store persistent data locally. In order to satisfy dynamic needs of mobile applications, alternative storage solutions should be used to offer a schema-free data layer. JSON can be used as a data interchange format for lightweight and easy manipulation. However, problems like performance and security must be considered in such mobile scenarios especially for enterprise applications. This thesis describes the analysis, design and development of a mobile storage solution. The proposed approach relies on JSON and it has been integrated into a proprietary enterprise framework in order to improve performances of the existing data layer. First of all, state-of-the-art data storages for mobile devices have been analysed, in terms of data and query model, concurrency, encryption, cross-platform support. After that, we designed and developed a new solution, based on SQLite with JSON1 extension. This standalone library has been integrated into a proprietary framework in order, to provide a performing schema-free data storage, able to store and query non-relational data in a very efficient way.

Acknowledgements

I would thank my academic supervisor Professor Alessio Vecchio that supported me carefully during this work.
Thanks to my tutor Stefano Zingarini that guided me during this work at Apparound.
Thanks to the Apparound company that gave me this big opportunity and thanks to all my colleagues that helped me every time I needed.
Thanks to my family and to my girlfriend Diletta that supported me during university years.
Thanks to all my friends who were beside me during this journey.

Contents

List of Figures

List of Tables

List of Abbreviations

JSON	**J**ava**S**cript **O**bject **N**otation
CMS	**C**ontent **M**anagement **S**ystem
DBMS	**D**ata**B**ase **M**anagement **S**ystem
RDBMS	**R**elational **D**ata**B**ase **M**anagement **S**ystem
SQL	**S**tructured **Q**uery **L**anguage
NoSQL	**N**ot **O**nly **S**tructured **Q**uery **L**anguage

Dedico questo lavoro di tesi ai miei genitori per il loro costante e prezioso supporto durante questi anni di studio…

Chapter 1

Introduction

Data availability on mobile applications is a key feature required to have a better user experience especially in offline scenarios. For enterprise applications, having data available on the device allows the user to work continuously, even if he does not have an internet connection available. Data can be synchronized as soon as the Internet connection is available. Working offline permits to save energy that on mobile devices is a limited resource. In order to store, retrieve and synchronize such data in an efficient way, a lightweight and flexible format is needed. In the last years, the JSON data format, an open standard that uses an human-readable text to transmit data, became the most common data format used to transfer data on the web. It permits to represent semi-structured data using a schema-free data model, very suitable for the dynamic needs of the mobile world. However, whereas most of database systems that can manage JSON data are available for server and desktop computers, we have few choices about embedded databases for mobile devices. New technologies for mobile device data persistence come just in the last years, so many companies had to implement their own home-made solutions. This work describes the analysis, design and development of a mobile storage solution that relies on JSON and is based on SQLite with JSON1 extension. It has been integrated into a proprietary enterprise framework, in order to improve performances of the existing custom solution.

1.1 Goals

Summarizing, the goals of this work are:

- Analyse and evaluate alternative data storage solutions considering the state-of-the-art of storage engine for mobile devices, evaluating their features with

particular attention to the support for the JSON data format.

- Design and implement a new standalone library for JSON data persistence in order to replace a legacy solution, allowing to change the underlying storage engine easily.
- Integrate the library into a custom framework and evaluate the performances compared against the old solution.

1.2 Related work

In [1] are presented three architectural principles that facilitate a schema-less development style within an RDBMS, so that users can store, query, and index JSON data. The first principle is *Storage Principle for JSON* that stores JSON data natively in VARCHAR, CLOB, RAW, or BLOB columns without any relational decomposition. A JSON object collection is modelled as a table having one column storing JSON objects. Each row in the table holds a JSON object instance in the collection. The second principle is *Query Principle for JSON* that extends SQL with SQL JSON operators. These operators embed a simple JSON path language to navigate and query JSON object instances. The third principle is *Index Principle for JSON* that uses indexes on JSON data to efficiently support queries on JSON object collection. It can use partial-schema-aware indexing to build secondary structures on top of the primary JSON object collection or schema-agnostic indexing relying on information retrieval techniques such as inverted index. We used *Storage Principle for JSON* and *Query Principle for JSON* in our work.
In [2] is proposed a novel embedded data storage system, called EJDB, for JSON data which can be used as a shared executable library. It is a C library based on modified version of Tokyo Cabinet [3]. However, this project seems to be abandoned and without a concrete support for mobile devices.
In [4] is presented Argo, an automated mapping layer for storing and querying JSON data in a relational system. Their results point to directions of how one can marry the best of both worlds, namely combining the flexibility of JSON to support the popular document store model with the rich query processing and transactional properties that are offered by traditional relational DBMSs.

1.3 Outline

This work is divided in several Chapters as follows:

- **Chapter 2**
  We present the context on which this work has been developed, introducing the mobile applications world. We introduce the Android platform architecture [5] and some principles about hybrid mobile applications [7][6] , illustrating advantage of writing a single code base and targeting multiple mobile platforms. We show the architecture of a popular framework for hybrid mobile applications development, Apache Cordova [8], and we introduce the Apparound framework [9].

- **Chapter 3**
  We start with a brief introduction about DBMS, highlighting RBDBMS and NoSQL main characteristics [10][11][12][13][14]. Then, we present the state-of-the-art of most widely used storage engines for mobile applications i.e. SQLite [15][16], Couchebase Mobile [20], Realm [21].

- **Chapter 4**
  We present Apparound LocalDB, a custom mobile NoSQL database developed by Apparound.

- **Chapter 5**
  We present LocalDB, the system that we designed and implemented as major part of this work.

- **Chapter 6**
  We present the performance test setup, the tools used and the result obtained in terms of exectution time and speedup [24][25].

- **Chapter 7**
  We summarize the work, draw conclusions and discuss possible future work.

- **Appendix A**
  We present the The JSON Data Interchange Standard according to its official reference [26].

Chapter 2

Background

In this chapter we present the context on which this work has been developed, starting from a brief introduction about mobile operating systems (with particular focus on Android), mobile applications, hybrid mobile applications and Apparound mobile applications. At the end will be presented the specific problem that this work is willing to solve.

2.1 Mobile Operating Systems

A mobile operating system is an operating system for smartphones, tablets, PDAs, or other mobile devices. Several operating systems like Android, iOS and Windows are available for those devices. It combines features of a personal computer operating system with other features useful for mobile or handheld use, usually including a touchscreen, cellular radio, Bluetooth, Wi-Fi, GPS, camera, speech recognition, voice recorder, music player and NFC.

2.1.1 Android Operating System

Android is a widely known 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. Android runs on a Linux kernel written in C. On top of the kernel the Android Runtime (ART) is deployed. Furthermore, a wide set of libraries is available, which handle for instance media, video processing and network communications and can be accessed through APIs provided by the Android SDK. 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. Android application are written using the Java language.

It is a garbage collected language, which means that it allows developer to do not care about explicitly memory deallocation. This task is carried out by the Android Runtime through a special process called Garbage Collector, which is in charge of finding data objects that cannot be accessed, and to reclaim the resources used by those objects. However, when this process runs it consumes additional resources, impacting performance and stalling program execution.

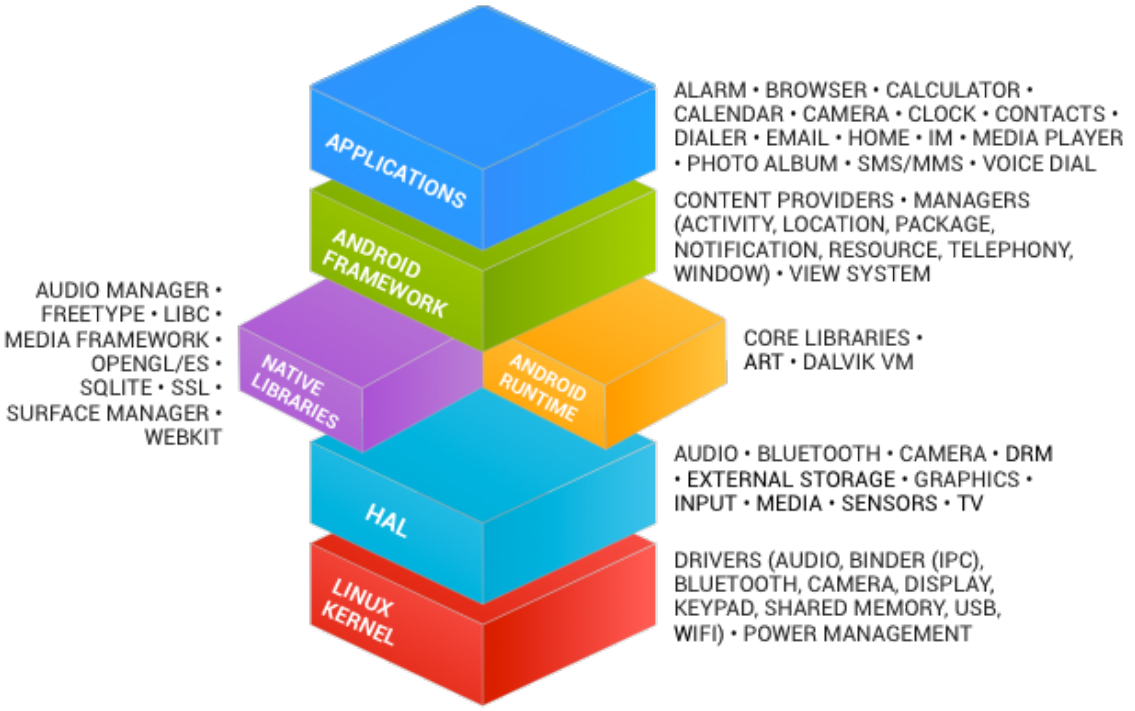


FIGURE 2.1: The Android stack[1]

2.2 Mobile Applications

A Mobile application is a computer program designed to run on mobile devices such as smartphones and tablets. Several operating systems like Android, iOS and Windows are available for those devices to support such new generation mobile software. Some applications come together with the device as pre-installed software to offer basic functionalities such as web browser, email client, calendar etc. Although it is ordinary software, developed using common programming languages such as

[1] https://source.android.com/

Java, Objective-C, C# etc., the main difference between desktop applications and mobile applications is that the latter are designed to offer a better user experience on mobile devices, removing everything not needed in order to keep them light and fast as much as possible against limited hardware resources offered.
Every operating system offers to the developers a rich framework, also known as SDK, in order to provide high-level API to access device functionalities and everything needed to develop complete mobile applications.

A generic mobile application is generally structured as a multi-layered application consisting of:

- **Presentation Layer**
  This layer is the topmost layer that provides the application user interface. It is in charge of displaying data to the user and handling the user's inputs.
- **Business Layer**
  This layer coordinates the application. It makes logical decisions, processes and moves data between the other two layers.
- **Data Layer**
  This layer stores and retrieves data from several sources using local cache services or remote services. Data are sent back to the Business Layer for processing.

2.3 Hybrid Mobile Applications

Hybrid Mobile applications look and behave like any other mobile application but they are different in the technologies in which they are developed. Hybrid Mobile applications are built with a combination of native technologies and standard web technologies like HTML, CSS and JavaScript (like websites on the Internet). The web part of these applications runs hosted inside a *WebView*.

A *WebView* is a native component that behaves like a web browser, typically configured to run in a fullscreen window. This component has a rendering engine to show web pages and enables these applications to access device capabilities such as camera, gps, accelerometers, filesystem, database, and more through a set of API

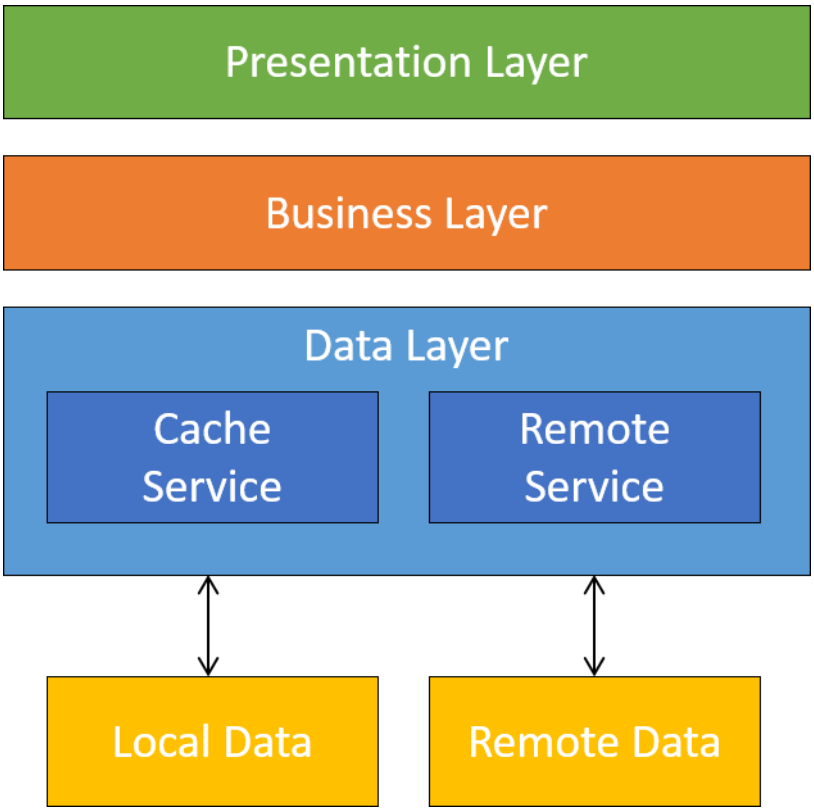


FIGURE 2.2: Generic Mobile Application Layers

bindings. These capabilities are often restricted for mobile web browsers, limiting the capabilities of standard web applications respect to hybrid mobile applications.

So, why should we build an hybrid mobile application?

- **Simple and fast development**
  Web languages are generally simpler than native languages, enabling the developers to rapid prototype and build full-featured mobile applications.

- **Cross-Platform deployment**
  Hybrid applications relays on WebView. Each mobile platform provides this component as part of its SDK, allowing developers to write a single web application that can run on different platforms. This is very convenient especially for the Presentation Layer and for the Business Layer that represent the most significant part of an application. There are several frameworks that provide all necessary tools to develop, build and deploy hybrid applications.

However, each platform comes with a set of caveats when it comes to its web runtime or WebView. This is especially true with Android, which is inconsistent between OS versions. Moreover, there might be unique capabilities of platforms to which a developer may wish to target. In those instances, a combination of plugins and platform-specific code must be utilized in order to take advantages of those capabilities.

An example of a widely used framework that allows developers to build hybrid cross-platform mobile applications is *Apache Cordova*. It provides a set of JavaScript API to develop mobile applications using web technologies and a set of plugins to enable web and native components to communicate.

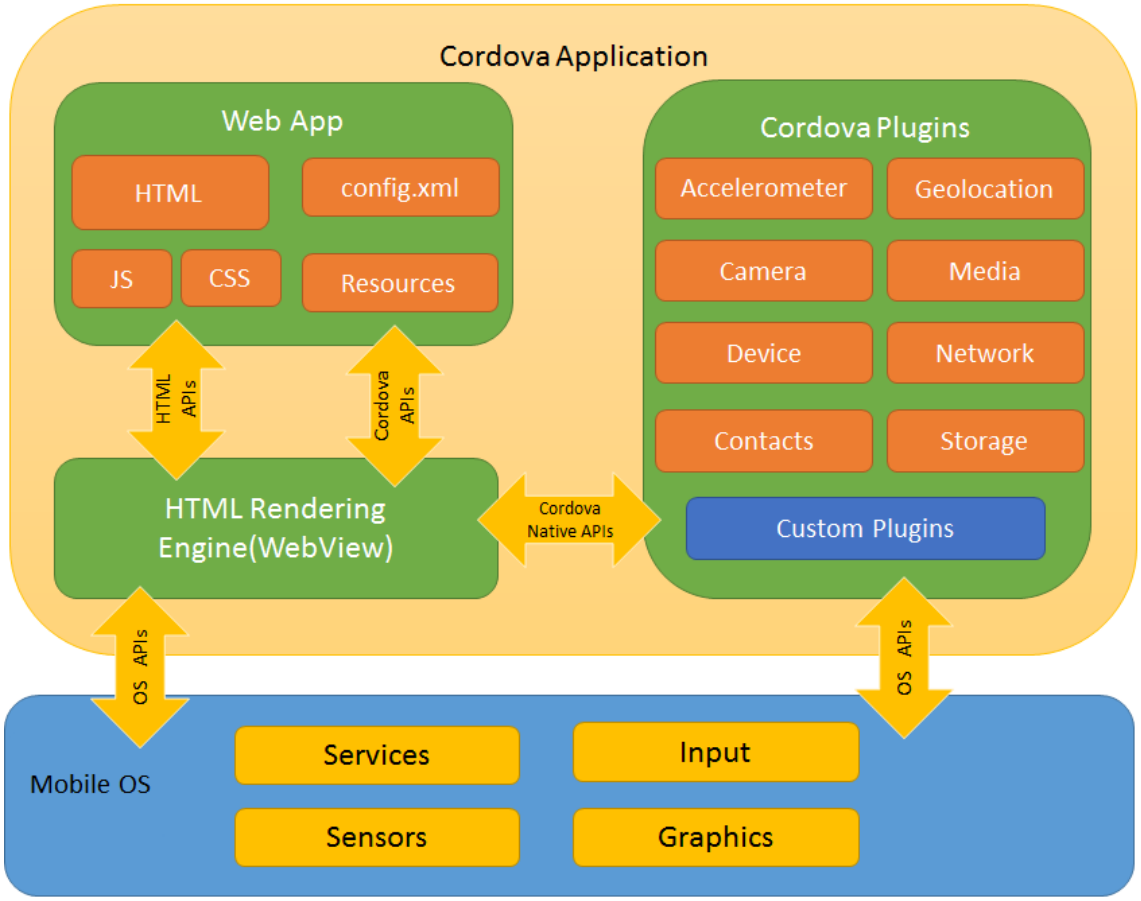


FIGURE 2.3: Apache Cordova Architecture[2]

[2] https://cordova.apache.org/docs/en/latest/guide/overview/

2.3.1 Android JavaScript Interface

On Android platform, we can deliver a web application or just a web page as a part of our application using a *WebView*. The *WebView* class is an extension of Android's *View* class that allows to display web pages as a part of an activity layout. All that *WebView* does, by default, is to show a web page. If the web page we plan to load in our *WebView* uses JavaScript, we must enable JavaScript explicitly for our *WebView*. In Listing 2.1 we can see an example.

LISTING 2.1: Enabling JavaScript on Android WebView

```
1 WebView myWebView = (WebView) findViewById(R.id.webview);
2 WebSettings webSettings = myWebView.getSettings();
3 webSettings.setJavaScriptEnabled(true);
```

We can create interfaces between our JavaScript code and client-side Android code. To bind a new interface between our JavaScript and Android code, we must call *addJavascriptInterface()*, passing it a class instance to bind to our JavaScript and an interface name that our JavaScript can call to access the class. In Listing 2.2 we can see how to create a new interface and in listing 2.3 how to register it on the *WebView*. In listing 2.4 we can see how to call our native method from the JavaScript

LISTING 2.2: Create a JavascriptInterface

```
1  public class WebAppInterface {
2      Context mContext;
3
4      WebAppInterface(Context c) {
5          mContext = c;
6      }
7
8      /** Show a toast from the web page */
9      @JavascriptInterface
10     public void showToast(String toast) {
11         Toast.makeText(mContext, toast, Toast.LENGTH_SHORT).show();
12     }
13 }
```

LISTING 2.3: Register the JavascriptInterface on WebView

```
1 WebView webView = (WebView) findViewById(R.id.webview);
2 webView.addJavascriptInterface(new WebAppInterface(this), "Android");
```

LISTING 2.4: Call the native method from JavaScript

```
1 function showAndroidToast(toast) {
2     Android.showToast(toast);
3 }
```

2.4 Apparound Mobile Applications

Apparound is a company focused on developing sales force automation tools based on mobile technologies.

2.4.1 Architecture

Apparound mobile applications can be classified as hybrid mobile applications because they are developed using a mix of native and web technologies. Apparound developed a proprietary framework on top of common mobile SDKs that supports the embedding of web applications enabled to access device capabilities regardless the underling platform. The supported mobile platforms are Android, iOS and Windows.

In Figure 2.4 we can see the architecture of an Apparound mobile application. The main layers of an Apparound mobile application are:

- **Native Layer**
  It implements the core functionalities on each mobile platform. These functionalities are available through a Native API and allow to access device capabilities such as network, sensors, filesystem, databases etc. APIs are exposed through a Widget Server that manages the requests coming from the above layer.

- **Widget Layer**
  It provides all needed tools to run a complete web application. This layer is composed by a native part and a JavaScript part. The native part is a custom WebView, specific for each mobile platform, which acts as HTML and JavaScript engine. The JavaScript part is a proprietary library, called JavascriptWrapper, that implements the JavaScript API, which allows the web application to interact with the Native API.

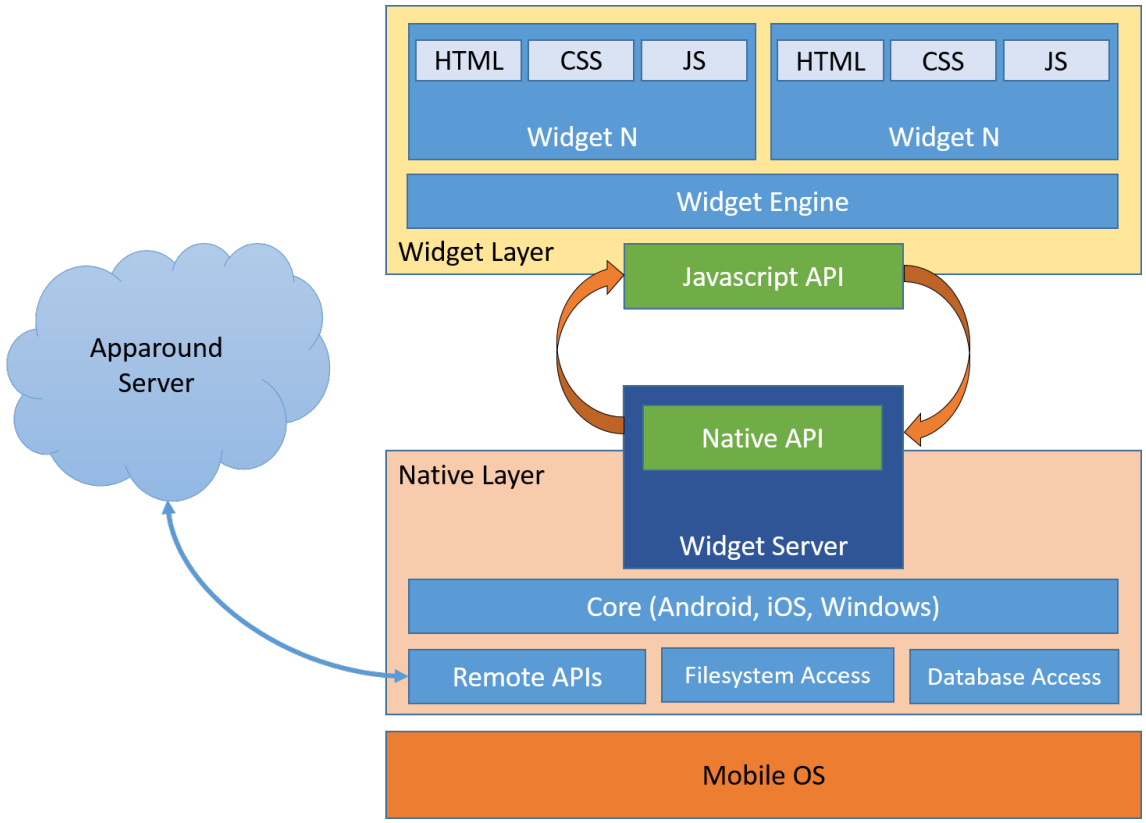


FIGURE 2.4: Apparound Mobile Application Architecture

A web application that runs inside this environment is called Widget. There can be more than one Widget inside a single mobile application. A Widget behaves as a standalone application but it can also communicate with other Widgets using several mechanisms provided by the framework. The JavascriptWrapper, which abstracts all implementation details to the Widget, enables to access to the Native API calling a function with some parameters. Every JavaScript API has its Native API counterpart.

A Widget can be shipped inside the application package or it can be downloaded from the back-end server as soon as it is needed. The Widget is published once to the Apparound back-end server using a proprietary CMS, making it available for download by the mobile applications when needed. This allows making changes to the Widget without the need to publish the whole native application to the different distribution platforms (Google Play, Apple Store and Windows Store) which can get into a long approval process.

This architecture permits to develop complex applications regardless the platform on which they will be deployed, using a single source code for the business and the presentation layer.

2.4.2 Offline feature

A key feature of Apparound applications is that they support full offline functionality, ensuring users to have the application fully working even if he has not an Internet connection available. At the first login, all necessary resources are downloaded from the back-end server in order to be available in the future. During offline work, all collected data are stored in the mobile application storage. As soon as the Internet connection is available, data are immediately synchronized with the back-end server.

Chapter 3

Storage engines for mobile devices

In this chapter we present the state-of-the-art of most widely used storage engines for mobile applications with a brief introduction about Database Management System, Relational and Non-Relational databases.

3.1 Database Management Systems

Database management systems (DBMS) are software applications that allow users to access an organized collection of data stored in a database. Databases can be roughly divided in relational and non-relational databases. Relational databases, commonly referred as SQL databases, 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. Therefore, researchers 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.

3.1.1 Relational Databases

In Relational Databases data are organized in structures called tables. Every table can have zero or more rows. Each row is an instance of the entity that the table represents. It is composed by one or more columns and each column represent an attribute of the entity. Relational databases are defined by ACID properties:

- **Atomicity**
  All of the operations in the transaction will complete, or none will.
- **Consistency**
  Transactions never observe or result in inconsistent data.

- **Isolation**
  The transaction will behave as if it is the only operation being performed.
- **Durability**
  Upon completion of the transaction, the operation will not be reversed.

3.1.2 Non-Relational Databases

Due to their normalized data model and their full ACID support, relational databases are not suitable for many scenarios with large amount of data, because joins and locks influence performance of the system negatively especially in distributed settings. NoSQL is a term often used to describe a class of non-relational databases that scale horizontally to very large data sets but do not in general make ACID guarantees. The CAP Theorem states that it is impossible for a distributed service to be consistent, available, and partition-tolerant at the same instant in time:

- **Consistency**
  All nodes see the same data at the same time
- **Availability**
  Every request receives a response about whether it succeeded or failed.
- **Partition tolerance**
  The system continues to operate despite arbitrary partitioning due to network failure.

Even if ACID properties can't be guarantees, Non-Relational storages are defined by BASE properties:

- **Basically Available**
  Replication and sharding techniques are used in NoSql databases to reduce the data unavailability, even if subsets of the data become unavailable for short periods of time.
- **Soft State**
  NoSql systems allow data to be inconsistent and provides options for setting tunable consistency levels.

- **Eventual consistency**
  The system will eventually become consistent once it stops receiving input. The data will propagate to everywhere it should sooner or later, but the system will continue to receive input and is not checking the consistency of every transaction before it moves onto the next one

3.1.3 Data Models

Although the main differences between SQL and NoSQL can be seen into a distributed environment, we are more interested about the data models they offer. SQL and NoSQL data stores can be classified by their data model. The most common are:

- **Relational data stores**
  Data are stored into tables of rows and columns as mentioned before. Each row has its own unique identifier but rows may be accessed also using a query language like SQL, which allows to filter data imposing conditions on column values. SQL permits also to do more complex operations like data projection, aggregation and join. Rows from different tables can be linked together using the foreign key constraint.

- **Key-Value stores**
  Data are stored as key-value pairs and can be accessed as an hash table or dictionary. Values may be simple strings or complex lists or sets. Values can be accessed only using the key so they are opaque to the system. Data can be aggregated into collections.

- **Document stores**
  Data are stored as a collection of documents. Each document has a unique identifier and encapsulates key-value pairs. Keys are unique within a document and values can be atomic values or documents as well, enabling the possibility to embedding nested documents. Key-value pairs are not opaque to the systems and can be queried as well. The most common formats used are XML, JSON and BSON. These formats permit to store schema-free semi-structured data giving the possibility to have documents with different attributes that can added at runtime. Document may contain references to other documents (also from a different collection) similarly to the foreign key concept in relational databases.

- **Column stores**
  Data are stored in a column of closely related data. Each column has an identifier on which are associated one or more attributes. In this way, data can be aggregated with less I/O operations.

- **Graph Databases**
  Data are stored as a collection of key-value pairs representing nodes in a graph. This type of storage offers an efficient way to manage data heavily linked together. Operations like recursive joins or shortest path calculation can be done very efficiently.

3.2 Mobile Storages

On mobile devices, data storages are very different with respect to those systems that can be found on back-end servers. Applications can store persistent data into files on the device memory or into embedded databases. There are some embedded databases available for mobile devices, designed explicitly for them.

3.2.1 SQLite

SQLite is a software library that implements a self-contained, serverless, zero configuration, transactional SQL database engine. It is written in C and the code is in the public domain. Unlike classical SQL databases, SQLite does not have a separate server process. SQLite reads and writes an ordinary disk file where it stores tables, indices, triggers and views. The database file format is cross-platform so the same database file can be moved freely across multiple systems. The library has a very small footprint (500 KB) and a minimal stack and heap impact on main memory during execution making SQLite a popular database on mobile devices. In fact, Android includes it as part of its SDK. Also iOS and Windows devices can run SQLite. Many pre-compiled binaries are available on SQlite web site.

Architecture In Figure 3.1 the architecture of SQLite is presented.

[1] https://www.sqlite.org/arch.html

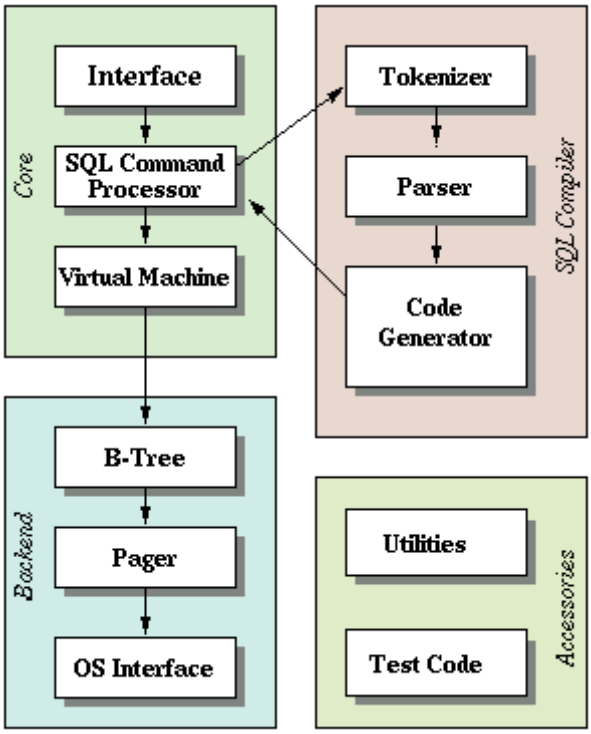


FIGURE 3.1: SQLite Architecture[1]

JSON1 Extension The JSON1 extension [17] is a SQLite loadable extension that implements 14 application-defined SQL functions and two table-valued functions that are useful for managing JSON content stored in an SQLite database. This extension is not provided in the SQLite released binaries neither in the version included into the Android SDK. In order to enable the JSON1 extension, we have to compile the SQLite source code adding a compile-time option as specified in the official documentation. The JSON1 extension currently stores JSON as ordinary text. The present implementation parses JSON text at over 300 MB/s.

JSON1 Extension functions The main functions of interest are essentially two:

- *json(X)*
  Verifies that its argument X is a valid JSON string and returns a minified version of that JSON string (with all unnecessary whitespace removed). If X is not a well-formed JSON string, then this routine throws an error.

- *json_extract(X,P1,P2,...)*
  Extracts and returns one or more values from the well-formed JSON at X. If only a single path P1 is provided, then the SQL datatype of the result is NULL for a JSON null, INTEGER or REAL for a JSON numeric value, an INTEGER zero for a JSON false value, an INTEGER one for a JSON true value, the dequoted text for a JSON string value, and a text representation for JSON object and array values. If there are multiple path arguments (P1, P2, and so forth) then this routine returns SQLite text which is a well-formed JSON array holding the various values. A well-formed PATH is a text value that begins with exactly one '**$**' character followed by zero or more instances of *".objectlabel"* or *"[arrayindex]"*. In listing 3.1 we show some examples.

LISTING 3.1: Examples of json_extract function

```
1 json_extract('{"a":2,"c":[4,5,{"f":7}]}', '$') ==> '{"a":2,"c":[4,5,{"f":7}]}'
2 json_extract('{"a":2,"c":[4,5,{"f":7}]}', '$.c') ==> '[4,5,{"f":7}]'
3 json_extract('{"a":2,"c":[4,5,{"f":7}]}', '$.c[2]') ==> '{"f":7}'
4 json_extract('{"a":2,"c":[4,5,{"f":7}]}', '$.c[2].f') ==> 7
5 json_extract('{"a":2,"c":[4,5],"f":7}','$.c','$.a') ==> '[[4,5],2]'
6 json_extract('{"a":2,"c":[4,5,{"f":7}]}', '$.x') ==> NULL
7 json_extract('{"a":2,"c":[4,5,{"f":7}]}', '$.x', '$.a') ==> '[null,2]'
```

Data Model In SQLite we can model data using the relation model. We have tables containing rows with several columns. In listing 3.2 we can see an example on how to create a SQLite table.

LISTING 3.2: SQLite create table

```
1 CREATE TABLE ExampleTable (
2     _id     INTEGER PRIMARY KEY AUTOINCREMENT,
3     column1     INTEGER,
4     column2     TEXT,
5     column3     TEXT
6 );
```

Because we are interested in storing JSON data, in listing 3.3 we show an example of how we can store data in JSON format. We can see that we don't have to define the JSON schema ahead of time, and we can store it as plain text.

LISTING 3.3: SQlite create table for JSON data

```
CREATE TABLE ExampleTable (
    _id     INTEGER PRIMARY KEY AUTOINCREMENT,
    jsonData    TEXT
);
```

After the creation of the table, we can start inserting data. In listing 3.4 we show an example. We use the function *json(X)* in order to verify and minify the provided text containing our JSON data.

LISTING 3.4: SQlite insert JSON data

```
INSERT INTO ExampleTable(jsonData)
VALUES (json('{"firstName":"Saul","address":{"city":"Los Angeles"}}'));
```

Query Model We can use SQL language in conjunction with *json_extract(X,P)* function in order to query inserted data. In listing 3.5

LISTING 3.5: SQLite query JSON data

```
SELECT jsonData
FROM ExampleTable
WHERE
    json_extract(jsonData,'$.firstName') = 'Saul'
  OR
    json_extract(jsonData,'$.address.city') = 'Los Angeles';
```

Concurrency At thread level, SQLite supports three different threading modes:

- **Single-thread** In this mode, all mutexes are disabled and SQLite is unsafe to use in more than a single thread at once.
- **Multi-thread** In this mode, SQLite can be safely used by multiple threads provided that no single database connection is used simultaneously in two or more threads.
- **Serialized** In serialized mode, SQLite can be safely used by multiple threads with no restriction.

At file level, SQLite allows multiple processes to have the database file open at once, and for multiple processes to read the database at once. When any process wants to write, it must lock the entire database file for the duration of its update.

Encryption The encryption for SQLite can be provided by SQLCipher [18], an open source library that provides transparent, secure 256-bit AES encryption of SQLite database files. It uses the internal SQLite Codec API to insert a callback into the pager system that can operate on database pages immediately before they are written to and read from storage, so it does not operate on the entire database file. This makes it very efficient. It is an extension of SQLite, but it does not function as a loadable plugin for many reasons so it is maintained as a separate project. SQLite has also an extension called SQLite Encryption Extension (SEE)[19]. SEE allows SQLite to read and write encrypted database files. All database content, including the metadata, is encrypted so that to an external observer the database appears to be white noise. In order to be used, this extension needs a software license.

3.2.2 Couchbase Mobile

Couchbase Mobile is a complete systems (shown in Figure 3.2) for mobile application data storage composed by:

- **Couchbase Lite**
  It is an embedded database that manages and stores data locally on the device in a document-oriented JSON format. It has full CRUD, query, and indexing functionality, all from a native API. Couchbase Lite has a small footprint at 500KB and supports all major device platforms. It is based on SQLite.

- **Couchbase Sync Gateway**
  It is built-in for replicating data between the embedded database and the database server. It includes multi-master replication, and both automatic and custom conflict resolution. It also supports peer-to-peer replication.

- **Couchbase Server**
  It is a NoSQL database server that manages and stores data in the cloud in a document-oriented JSON format. It scales easily to billions of records and terabytes of data, and it provides sub-millisecond response time for reads and writes.

It offers a complete solution to manage many mobile scenarios along with a synchronization layer and a back-end server. Currently, Couchbase Lite is supported on Android, iOS and Windows platforms.

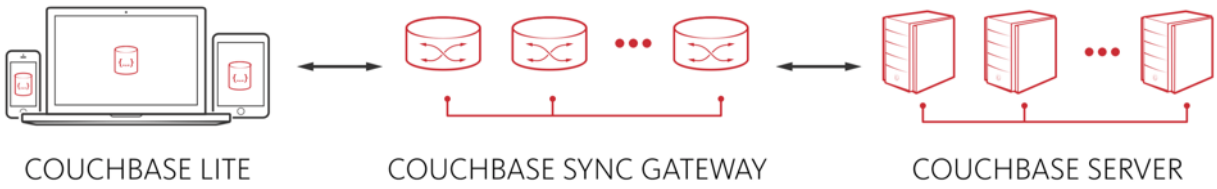


FIGURE 3.2: Couchbase Mobile Architecture

Data Model On Couchbase Lite, data are stored in a Document that is referenced by an unique ID within the database. Data documents consists of arbitrary JSON objects not enforced by rigid schemas. This provides the advantage of designing data models that can be naturally nested like a dictionary throughout the object model it is representing. In listing 3.6 we can see an example on how to create a new document and store it within the database.

LISTING 3.6: Couchbase data modelling and storing example

```
1  // Get the database (and create it if it doesn't exist).
2  Manager manager = new Manager(new JavaContext(), Manager.DEFAULT_OPTIONS);
3  Database database = manager.getDatabase("mydb");
4
5  // Create a new document (i.e. a record) in the database.
6  Document document = database.createDocument();
7  Map properties = new HashMap();
8  properties.put("firstName", "Saul");
9  properties.put("lastName", "Hudson ");
10
11 // create inner properties
12 Map innerProps = new HashMap();
13 innerProps.put("street", "Fuller Avenue");
14 innerProps.put("city", "Los Angeles");
15 innerProps.put("state", "California");
16 properties.put("address", innerProps);
17
18 // save the document
19 document.putProperties(properties);
```

Query Model On Couchbase Lite, everything is based on indexes created using a *MapReduce* function executed on all the documents stored in the database. The *MapReduce* function takes two arguments: a document and an emitter. The emitter is an object used to add entries to an index. The document is a JSON document stored in the database. This Map function will be executed on all of the JSON documents available. The resulting index in Couchbase terminology is called a *View*. In listing

3.7 we show an example on how to create a *View*, mapping city name and firstName of a person, and how to perform a query on the created *View*. Couchbase Lite limits queries only on previously created *Views*.

LISTING 3.7: Couchbase view and query

```
1  // Create a view and register its map function:
2  View citiesView = database.getView("cities");
3  citiesView.setMap(new Mapper() {
4      @Override
5      public void map(Map<String, Object> document, Emitter emitter) {
6          String city  = (List) document.get("address").get("city");
7          emitter.emit(city, document.get("firstName"));
8      }
9  }, "2");
10
11 // perform a query on the created views
12 Query query = database.getView("cities").createQuery();
13 query.setMapOnly(true);
14 QueryEnumerator result = query.run();
15 for (Iterator<QueryRow> it = result; it.hasNext(); ) {
16     QueryRow row = it.next();
17     String firstName = (String) row.getValue();
18     Log.i("MYAPP", "The␣fistName␣is␣%s", productName);
19 }
```

Concurrency Because Couchbase Lite relays on SQLite for data persistence, it has the same concurrency model.

Encryption Couchbase Lite has a built-in enterprise level security that includes user authentication, user and role based data access control (RBAC), secure transport over TLS, and 256-bit AES full database encryption.

3.2.3 Realm

Realm is a mobile database that runs directly inside phones, tablets or wearables that can be considered a replacement for SQLite. Instead of wrapping SQLite as many libraries do, it has its own core engine, written in C++. It uses the *zero-copy* principle when possible, allowing to have a direct access to the raw database instead of have to copy on memory all needed data. This is possible because each Realm object talks directly to the underlying database with a native pointer to the data in the database file that is always memory-mapped. Realm stores data at the vertical level

as columns, allowing to avoid read the entire row from the database. It is ACID compliant and supports transactions. Currently, Realm is supported on Android and iOS platforms.

Data Model On Realm, data model classes are created by extending the *RealmObject* base class, so we have to know the structure of our model ahead. In listing 3.8 we can see an example.

LISTING 3.8: Reaml data modelling

```
1  // Define the model class by extending RealmObject
2  public class Address extends RealmObject {
3      @PrimaryKey
4      private long id;
5      private String street;
6      private String city;
7      private String state;
8      // ... Generated getters and setters ...
9  }
10 // ...
11 public class Person extends RealmObject {
12     @PrimaryKey
13     private long id;
14     private String fistName;
15     private String lastName;
16     private Address address;
17     // ... Generated getters and setters ...
18 }
```

On Realm, all write operations (adding, modifying, and removing objects) must be wrapped in write transactions. A write transaction can either be committed or cancelled. During the commit, all changes will be written to disk, and the commit will only succeed if all changes can be persisted. By cancelling a write transaction, all changes will be discarded. Using write transactions, data will always be in a consistent state. In listing 3.9 we can see how to persist data within a Realm database. It is possible to add *RealmObjects* represented as JSON directly to Realm.

LISTING 3.9: Realm data persisting

```
// Initialize Realm and get a Realm instance for this thread
Realm.init(context);
Realm realm = Realm.getDefaultInstance();

// Use class model like regular java objects
Person person = new Person();
person.setFistName("Saul");
person.setLastName("Hudson");

// create object from a JSON string
Address address = realm.createObjectFromJson(Address.class,
    "{\"street\":\"Fuller Avenue\",\"city\": \"Los Angeles", \"state\": \"California\" }");

person.setAddress(address);

// Persist data in a transaction
realm.beginTransaction();
final Person managedPerson = realm.copyToRealm(person);
realm.commitTransaction();
```

Query Model Realm's query engine uses a Fluent interface to construct multi-clause queries. In listing 3.10 we can see an example.

LISTING 3.10: Ream query

```
// Build the query looking at all people
RealmQuery<Person> query = realm.where(Person.class);

// Add query conditions:
query.equalTo("fistName", "John");
query.or().equalTo("lastName", "Peter");

// Execute the query:
RealmResults<Person> result1 = query.findAll();

// Or alternatively do the same all at once (the "Fluent interface"):
RealmResults<Person> result2 = realm.where(Person.class)
                                    .equalTo("fistName", "John")
                                    .or()
                                    .equalTo("lastName", "Peter")
                                    .findAll();
```

Concurrency Realm allows to work with data on multiple threads without having to worry about consistency or performance because objects are auto-updating at all times. We can operate on live objects in different threads, reading and writing to

them, without worrying about what other threads are doing to those same objects. If we need to change data we can use a transaction. The other objects in the other threads will be updated in near real time. The only limitation with Realm is that we cannot pass Realm objects between threads. If we need the same data on another thread, we just query for that data on the other thread.

Encryption The Realm file can be stored encrypted on disk by passing a 512-bit encryption key (64 bytes). This ensures that all data persisted to disk is transparently encrypted and decrypted with standard AES-256 encryption. The same encryption key must be supplied each time a Realm instance for the file is created.

3.2.4 Summary

The table 3.1 summarizes the characteristics of each analysed database.

TABLE 3.1: Mobile Databases comparison

	SQLite	**Couchbase**	**Realm**
Data model	Relational + Json Documents with JSON1 extension	Json Documents	Realm-Objects
Query model	SQL	Custom native map-reduce query	Custom pipelined native query
Concurrency	Thread-safe, File level lock	Thread-safe, File level lock	Thread-safe, Database level lock
Encryption	Yes, with SQLCipher or SSE	Yes, built-in	Yes, built-in
Cross-Platform	Android, iOS and Windows	Android, iOS and Windows	Android and iOS

Chapter 4

LocalDB

In this chapter we present Apparound LocalDB. It can be classified as a NoSQL Database Engine that uses a JSON document based data model. The need to have a flexible data model offered by JSON documents arise primarily from the fact that data may come from external services that do not provide the schema in advance so the use of a relational data model may be not feasible. It is used in several application processes like contents synchronization, search, and quoting.

4.1 Internal Structure

LocalDB internally stores data using several files which contain text in JSON format (see Appendix A), eventually encrypted.
For each entity type we want to store, a table is created. Each table stores items named records. Physically, a table consists in some files that have a maximum dimension of 10 KBytes for performance reason. A file belonging to a table is named according the following rules:

- *TableName.json* if it is the first file
- *TableName_n.part.json* if it is the n^{th} file, where $n \geq 2$

Files are linked together using some informations contained into the stored JSON data as we will see in next section. In figure 4.1 we can see an example of the internal structure based on JSON files.

4.1.1 File Structure

File content has a fixed structure. The information contained inside the root JSON object are the following:

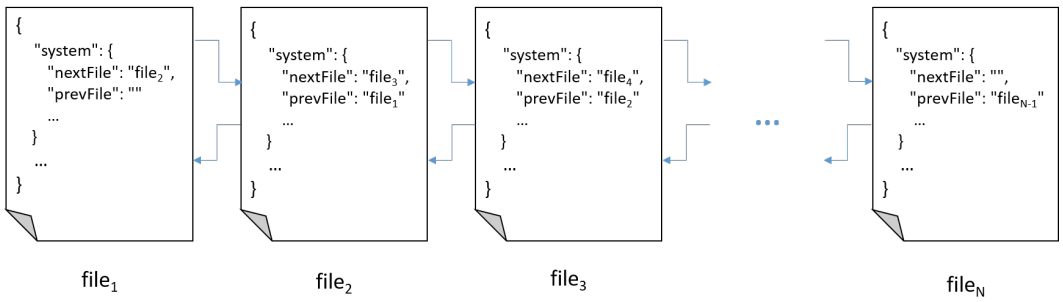


FIGURE 4.1: LocalDB files

- *"system": <Object>*
  Informations about how current file is linked with other files:
  - *"prevFile": <String>*
    the previous file name
  - *"thisFile": <String>*
    the current file name
  - *"nextFile": <String>*
    the next file name
  - *"counter": <Number>*
    the order inside the file list

  The value of *"prevFile"* and *"nextFile"* may be empty, indicating that there is no previous file (i.e. the current file is the first file) or there is no next file (i.e. the current file is the last file) respectively

- *"remoteIds": <Object>*
  Informations about mapping between *remoteId* and *localId*

- *"records": <Object>*
  List of stored records, accessible by their *localId*

In the following an example of the internal structure of a LocalDB file is presented:

LISTING 4.1: File structure

```
{
  "system" : {
    "prevFile" : "ExampleTable_2.part.json",
    "thisFile" : "ExampleTable_3.part.json",
    "nextFile" : "ExampleTable_4.part.json",
    "counter" : 3
  },
  "remoteIds" : {
    "e123456" : "20160414-1554-0732-1234",
    "e789102" : "20160414-1554-0732-5678"
  },
  "records" : {
    "20160414-1554-0732-1234" : {
      "localId" : "20160414-1554-0732-1234",
      "remoteId" : "e123456",
      "entity" : {
        "name": "The name 1",
        "isReadOnly" : false,
        "total" : 115.231
      }
    },
    "20160414-1554-0732-5678" : {
      "localId" : "20160414-1554-0732-5678",
      "remoteId" : "e789102",
      "entity" : {
        "name": "The name2",
        "isReadOnly" : false,
        "total" : 102.456
      }
    }
  }
}
```

4.1.2 Record Structure

Records have a fixed external structure, with several information used for synchronization purpose.

- *"localId"*: <String>
  The local identifier
- *"remoteId"*: <String>
  The remote identifier. It may be null if record has been created locally and never synchronized

- *"entity": <Object>*
  The actual record content
- *"localLinks": <Object>*
  Information about linking with other records of other tables
- *"flags": <Object>*
  Information useful during the synchronization process
- *"originalEntity": <Object>*
  The record content before last synchronization

Following, we can see an example of the internal structure of a record:

LISTING 4.2: Record structure

```
{
  "localId" : "20160127-1643-1936-0497-359593065860",
  "remoteId" : "29243343-3d1c-460c-97e5-e2f7cd65c4ad",
  "entity" : {
    /* entity data */
  },
  "localLinks" : {
  },
  "flags" : {
    "locked" : false,
    "toBeSynchronised" : false,
    "readOnly" : false,
    "deleted" : false
  },
  "originalEntity" : {
    /* entity data before last synchronization */
  }
}
```

4.2 Internal Operations

LocalDB exposes some API allowing to create, retrieve, update and delete stored items.

- *String addItem (String table, Object item, Boolean allowReplace)*
- *Object getItem (String table, String id)*
- *Array<Object> findItems (String table, Array<Object> criteria)*

- *Boolean deleteItem (String table, String id)*

4.2.1 addItem

Add a new item to the table. If an item with same localId or remoteId already exists and allowReplace flag is true, the item is updated, else the operation fails. It returns the localId of the added/updated item if the operation ends successfully or an empty string otherwise. If localId is null or empty, it is needed to generate a new one. In addition, this function has to keep the table files in a consistent state. See Listing 4.3 for more details.

LISTING 4.3: Add item

```
input: String TABLE, Object ITEM, Boolean REPLACE
output: String


begin
    localId ← ITEM.localId
    if localId is null or ""
        localId ← newGUID()
    end
    remoteId ← ITEM.remoteId

    files ← getTableFiles(TABLE)
    foreach file in files
        if localId not in file.records.keys
            continue
        end
        if REPLACE is false
            return ""
        end

        file.updateRemoteIds(remoteId, localId)
        file.updateRecord(localId, ITEM)
        file.write()
        return localId
    end

    if files.count > 0
        lastFile ← files.lastFile
        if lastFile.hasSpace()
            lastFile.addRecord(localId, ITEM)
            lastFile.write()
        else
            newFile ← createNewFile()
            newFile.addRecord(localId, ITEM)
            newFile.updateSystemInfo(lastFile.filename)
            newFile.write()
            lastFile.updateSystemInfo(newFile.filename)
            lastFile.write()
        end
    else
        newFile ← createNewFile()
        newFile.addRecord(localId, ITEM)
        newFile.write()
    end
    return localId
end
```

4.2.2 getItem

It returns an item from the table whose localId or remoteId is equal to the given id. It returns the item if it was found, null otherwise. See Listing 4.4 for more details.

LISTING 4.4: Get item

```
1  input: String TABLE, String ID
2  output: Object
3
4
5  begin
6      item ← null
7      localId ← ID
8
9      files ← getTableFiles(TABLE)
10     foreach file in files
11         item ← file.records[localId]
12         if item is null
13             localId ← file.remoteIds[ID]
14             if localId not null
15                 item ← file.records[localId]
16             end
17         end
18         if item not null
19             return item
20         end
21     end
22     return null
23 end
```

4.2.3 findItems

It returns an array containing the items that match the given criteria. Criteria has the structure shown in Listing 4.5. It is a JSON array where each element is a JSON object that represents a prototype for the items that we want to get. In order to be included into results, an item has to match at least one of the given criteria.

LISTING 4.5: JSON criteria

```
[
  {
    "entity" : {
      "company" : {
        "city" : "Pisa",
        "type" : "srl"
      }
    }
  },
  {
    "entity" : {
      "company" : {
        "city" : "Firenze",
        "type" : "srl"
      }
    }
  }
]
```

LISTING 4.6: Find items

```
1  input: String TABLE, Array<Object> CRITERIA
2  output: Array<Object>
3
4
5  begin
6      res ← []
7      files ← getTableFiles(TABLE)
8      foreach file in files
9          foreach record in file.records
10                 if record match CRITERIA
11                     res.push(record)
12                 end
13         end
14     end
15     return res
16 end
```

4.2.4 deleteItem

Delete an item from the table whose localId or remoteId is equal to the given id. It returns true if the operation ends successfully, false otherwise. In addition, this function has to keep the table files in a consistent state. See Listing 4.7 for more details.

LISTING 4.7: Delete item

```
input: String TABLE, String ID
output: Object


begin
    files ← getTableFiles(TABLE)
    foreach file in files
        localId ← ID
        if file.records[localId] is null
            localId ← file.remoteIds[ID]
        end
        if localId not null
            delete file.records[localId]
            if file.records.length == 0
                prevFile ← openFile(file.system.prevFile)
                if prevFile not null
                    prevFile.system.nextFile ← file.system.nextFile
                    prevFile.write()
                end
                nextFile ← openFile(file.system.nextFile)
                if nextFile not null
                    nextFile.system.prevFile ← file.system.prevFile
                    nextFile.write()
                end
                file.delete()
            end
            return true
        end
    end
    return false
end
```

4.3 Issues

As we seen in previous sections, LocalDB stores data on several files, which contain data stored using JSON as serialization format. This solution provides a flexible data model and allows to interchange data between native side and JavaScript side easily. However, these data have no index support so, whenever there is the need to find data items matching some criteria, a full-scan search has to be performed in order to retrieve these data. This process may potentially involve all files so, a large amount of reads from the persistent storage may be required making the process very long. In particular, on Android, this process allocates memory for a large amount of objects. These object are accessed for a small time so the Garbage Collector has to enter

in action many times to reclaim the unused resources. Its action consumes CPU time and slows down the application responsiveness. In Figure 4.2 we can see the RAM and CPU usage during a a full-scan search operation. We can see how the Garbage Collector action is related to the RAM usage, observing a memory a peek followed by a drain appearing. When the Garbage Collector enters in action, we can observe also a peek on CPU usage.

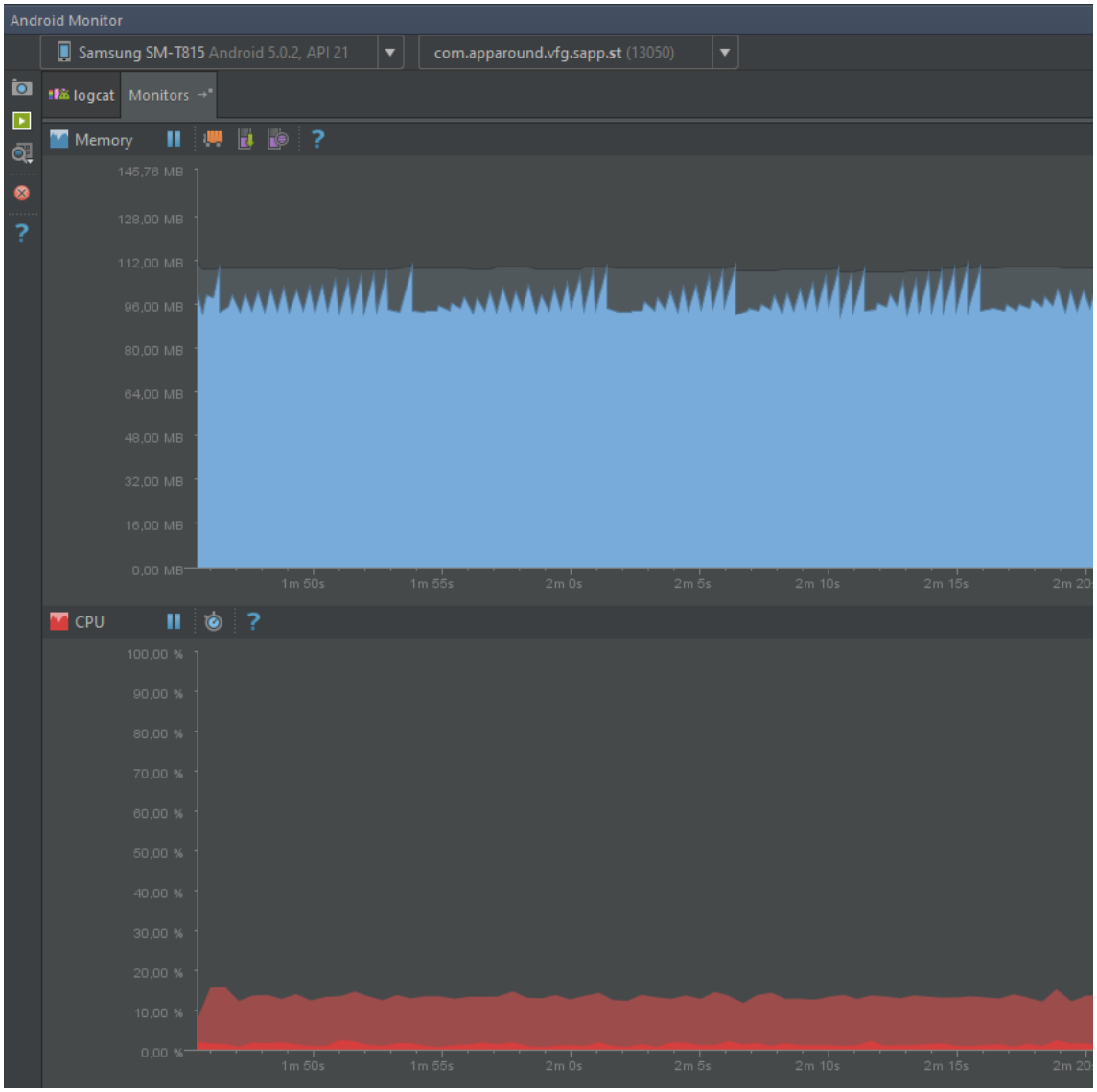


FIGURE 4.2: RAM and CPU usage during a search operation

Chapter 5

Improvements to LocalDB

In this chapter we present the improvements for LocalDB that we designed and implemented as major part of this work. The goal is to develop a new standalone library to provide a data persistence layer that must meet some requirements in order to improve performance, maintainability and robustness respect to the existing solution.

5.1 Requirements

In this section we analyse the requirements that our library must meet.

5.1.1 Functional requirements

- **Use JSON as data format**
  The library must store and retrieve items in JSON format
- **Use localId and remoteId as identifiers**
  Each stored item must be identified by a localId locally unique and eventually by a remoteId. If not present, the localId must be generated on the fly as GUID
- **Interfaces backward compatible**
  The library publicly available interface must be compatible with the old LocalDB in order to be easily integrated into the Apparound framework. For this, it has to expose a minimal set of API in order to create, retrieve, update and delete data items.
- **Same Business logic of old LocalDB**
  The library internal functions must implement the same logic of the old LocalDB

- **Interchangeable storage engine**
  The library must allow to change the underlying storage engine without changing its own functionality
- **Manage multiple database instances**
  The library must allow to instantiate and retrieve multiple database instances
- **Thread safe**
  The library must be thread safe, handling concurrence on data access

5.1.2 Non-Functional requirements

- **Performance**
  The library shall guarantee good performance even if it has to large amounts of data
- **Reusability**
  The library shall be standalone in order to be reusable in different contexts
- **Portability**
  The library shall be design to be implemented in different mobile platforms
- **Security**
  The library shall encrypt persistent stored data
- **Maintainability**
  The library shall be as modular as possible in order to be maintainable even if it becomes complex
- **Reliability**
  The library shall guarantee atomicity and data durability
- **Robustness**
  The library shall handle unusual data and internal errors
- **Testability**
  The library shall allow to test each module independently

5.2 Library overview

The library shall be integrated into a generic application as well as into a generic framework. The core of the library is implemented using native technologies. The architecture includes a cross-platform storage engine written in C/C++ and uses a platform specific compiled binary with its own bindings for data persistence. The library can be accessed from JavaScript by implementing an interface on the native side to access its functionalities. In Figure 5.1 we can see how these components can be placed together showing a full stack integration, highlighting the fact that some platform specific implementations are needed.

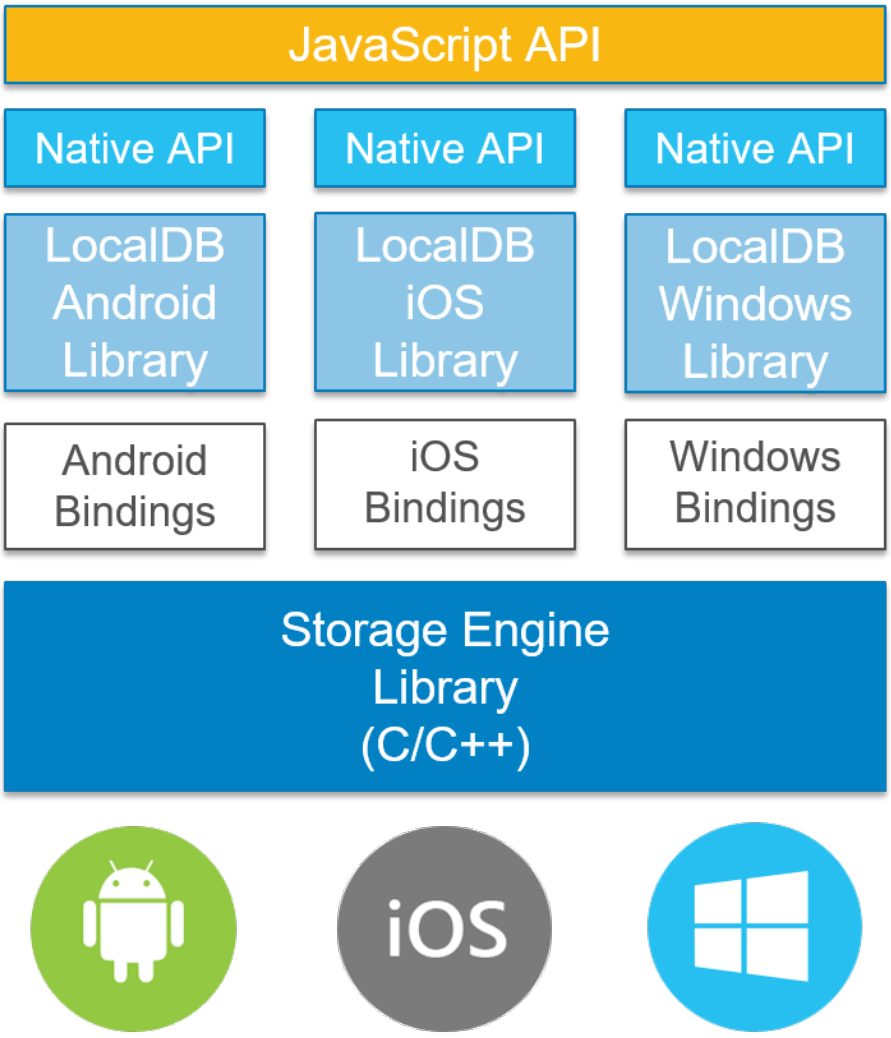


FIGURE 5.1: LocalDB library integration example

5.3 Architecture

The LocalDB library is composed of several hierarchical components in order to provide a wide modularity and separation of concerns. The goals of this architecture is to provide a good abstraction on how data are persistently stored by the underling storage engine.
The main modules are:

- **LocalDB Manager**
  This module exposes the API to create or retrieve LocalDB instances. It is in charge of instantiate, cache and return a new instance of LocalDB if it does not exist or return a cached instance if it has been create before.

- **LocalDB API**
  This module exposes the API to add or update, get, find, remove and iterate stored items. The API is compliant with old LocalDB version.

- **Storage Engine Abstraction**
  This module abstracts storage engine implementation details. It knows which of the underling storage engine methods to call in order to perform the desired operation.

- **Storage Engine**
  This module implements core functionalities to create, retrieve, update and remove data items physically. It can use different technologies to persist data but the interface is the same across all different storage engine implementations.

In Figure 5.2 we can see the LocalDB Library architecture and how this can support different types of storage engines.

5.4 Interfaces

In this section we describe the interfaces that our modules must expose to make internal functionalities available to other modules. According to JSON format (see Appendix A) we use Object and Array type indicating JSON object and JSON array respectively.

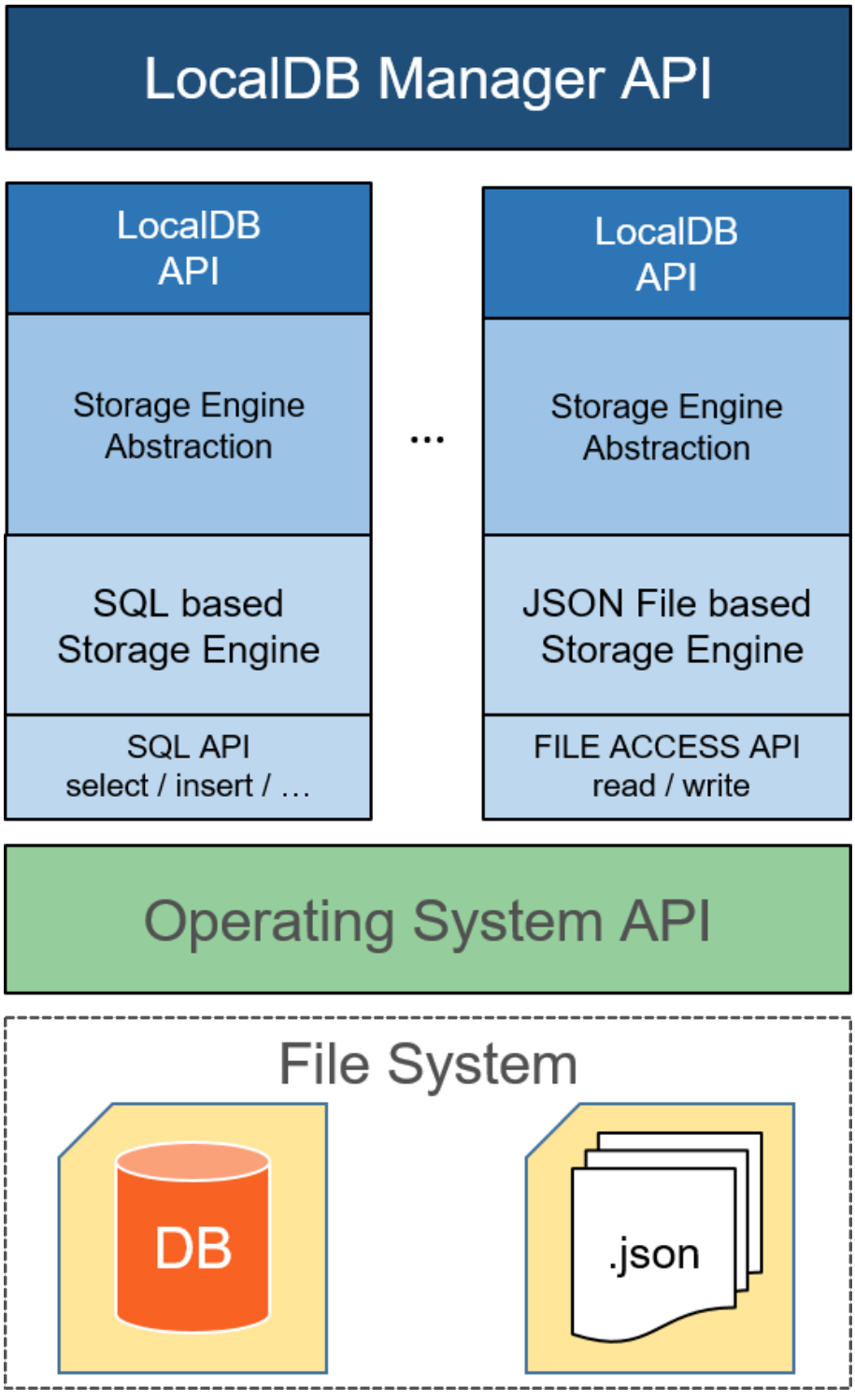


FIGURE 5.2: LocalDB library architecture

5.4.1 LocalDB interfaces

Here, we describe the LocalDB interfaces, derived from the old LocalDB interfaces. The expected behaviour of the new API is the same shown in Chapter 4 with some improvements in order to extend LocalDB functionalities and to support future improvements. The needed APIs are:

- *String addItem (String table, Object item, boolean allowReplace)*
  It does the same as in old LocalDB
- *Object getItem (String table, String id)*
  It does the same as in old LocalDB
- *Array<Object> findItems (String table, Array<Object> criteria, Object options)*
  It does the same as in old LocalDB. In addition, it accepts options in order to sort results according some criteria, limit the results or start from a particular offset. In Section 5.5 we can see the findItems options structure.
- *boolean removeItem (String table, String id)*
  It does the same as in old LocalDB.
- *void iterate (String table, Object options, IterationListener listener)*
  Start a new iteration on the given table using options to establish the blocks size and the sorting criteria of the items. The progress of the operation is notified through the *IterationListener* interface. Each iteration has its own identifier that will be passed to the listener for each notification
- *void continueIterate (String iterationId)*
  Continue the execution of the iteration with the given id
- *void stopIterate (String iterationId)*
  Stop the execution of the iteration with the given id

Furthermore, the *IterationListener* must expose the following API:

- *void onContinueIterate(Object result)*
  Notify that there are more items and passes the result of the iteration(Listing 5.2)
- *void onStopIterate()*
  Notify that there are no more items or that the operation has been stopped

5.4.2 Storage Engine Abstraction interfaces

The interface of the storage engine abstraction is the same as LocalDB in order to allow LocalDB to call them easily, so it is not described again.

5.4.3 Storage Engine interfaces

Now, we describe the storage engine interfaces, valid for every storage engine we want to implement. We need to expose the API for basic CRUD operations that will be invoked by the Storage Engine Abstraction module in order to implement the LocalDB operations. The API has the following methods:

- *String createItem (String table, Object item)*
  Create an item
- *Object retrieveItem (String table, String id)*
  Retrieve a single item
- *Array retrieveItems (String table, Array<Object> criteria, Object options)*
  Retrieve items that match given criteria with options (Section 5.1) about limit, offset and sorting criteria.
- *String updateItem (String table, Object item)*
  Update an item
- *Boolean deleteItem (String table, String id)*
  Delete an item

5.5 Data Design

In this section we show the data structures needed by our library.

5.5.1 Criteria structure

The criteria structure is the same seen in Listing 4.5

5.5.2 Options structure

LISTING 5.1: Find options structure

```
{
    "limit": 50,
    "offset": 100,
    "sort": [{
        "orderBy": "<some json field>",
        "orderByAsc": true
    },{
        "orderBy": "<some json field>",
        "orderByAsc": false
    }]
}
```

- *limit*: the maximum number of items to return
- *offset*: the offset inside the result set from which start
- *sort*: an array of JSON objects with a field *orderBy* that indicates the JSON field on which results have to be sorted and a field *orderByAsc* that indicates the sort direction (ascendant or descendant)

5.5.3 Iteration result structure

LISTING 5.2: Iteration result structure

```
{
    "iterationId" : "<guid>",
    "items" : [
        {
            < item >
        }
        ...
        {
            < item >
        }
    ]
}
```

- *iterationId*: the id of the iteration, useful to continue it
- *items*: the result items for the current iteration

5.6 Library Design

In this section we show the library design with the class diagram (Section 5.3) needed to implement the architecture shown previously.

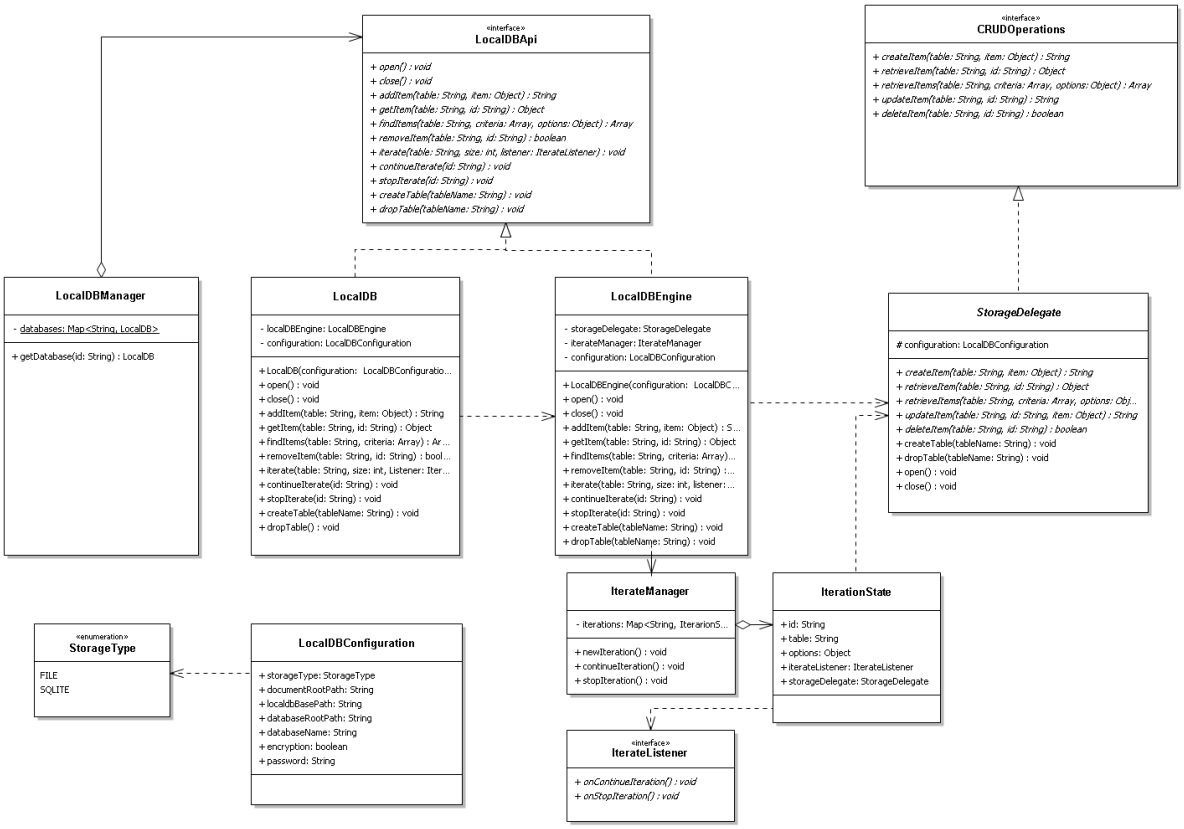


FIGURE 5.3: LocalDB library class diagram

The main classes and their tasks are:

- **LocalDBConfiguration**
  Contains all configuration data like the storage type, the storage path, the encryption flag and the password.

- **LocalDBManager**
  Creates new instances of *LocalDB* based on a path in which data will be stored. The path is specified into *LocalDBConfiguration*. Created instances are cached

so if an instance that has been already created is requested, it is returned instead of recreating it.

- **LocalDB and LocalDBEngine**
  Implement *LocalDBApi* interface, which provides the public APIs of LocalDB Library. Every *LocalDB* instance has a reference to a *LocalDBEngine* instance. These classes implement a Proxy Pattern[22] in order to delay the instantiation of the *LocalDBEngine* (and related heavy objects) until the first time it is needed and control the access to it. Furthermore, *LocalDBEngine* uses the Bridge Pattern[23] in order to decouple and hide implementation details and the interaction with the *StorageDelegate* and the *IterateManager*.

- **StorageDelegate**
  It is an abstract class that implements *CRUDOperations* interface. Every concrete *StorageDelegate* must extend this base class in order to implement the effective Storage Engine. Concrete *StorageEngine* classes transform upcoming JSON data into the right storage format and persist it. They are also in charge of interpret criteria in order to perform filtering over the stored data. The concrete type of the *StorageDelegate* can be decided at the time of *LocalDB* instance creation. In particular, this can be done by providing an instance of *LocalDBConfiguration*, and using the *StorageType* enumerator.

- **IterateManager**
  Helps *LocalDBEngine* to perform iterations on blocks of stored items using its *StorageDelegate*. It stores a map of *IterationState* instances identified by an identifier and takes a reference to the underlying *StorageEngine*.

- **IterationState**
  Contains informations about the state of an iteration such as the id, the table on which the iteration is done, the options with inside the iteration size and offset.

5.7 Implementation

In this section we discuss the implementation details of the library.

5.7.1 Storage Engine based on SQLite

From the analysis of the state-of-the-art of storage engines for mobile devices conducted in Chapter 3 and from requirements stated on Section 5.1, SQLite with JSON1 extension appears as the best candidate to be used as underling storage engine for our library. It allows to have a schema-free data model in conjunction with a powerful query model using the JSON1 extension. To be more precise, we actually use SQLCipher which is an extension of SQLite that encrypts SQLite database files, in order to achieve the security requirement. Since SQLCipher provides the same interface provided by SQLite, we will refer to it as SQLite for simplicity. The only difference stays in the method used to open the database file. SQLCipher requires the encryption password as extra parameter respect to SQLite. It is also possible to avoid database encryption passing an empty password as parameter.

5.7.2 Architecture Details

Starting from the architecture seen in Section 5.3, we show a detailed view of our library architecture. In particular, we show what components the storage engine needs and what is the data flow through these components. In figure 5.4 we can see LocalDB architecture with SQLite as storage engine. We can see that LocalDB API and the Storage Engine Abstraction do not change even when using a specific underling storage engine.

The SQlite based storage engine needs these two components:

- **Query builder**
  This component is in charge of translating criteria into a SQL query
- **Data converter**
  This component has two main tasks:
  - Wrap JSON data into an insert or update SQL query
  - Convert records coming from the underling SQLite engine into JSON formatted data

Further more, whenever a **retrieve** item operation is requested, the storage engine must:

1. Take up-coming JSON criteria

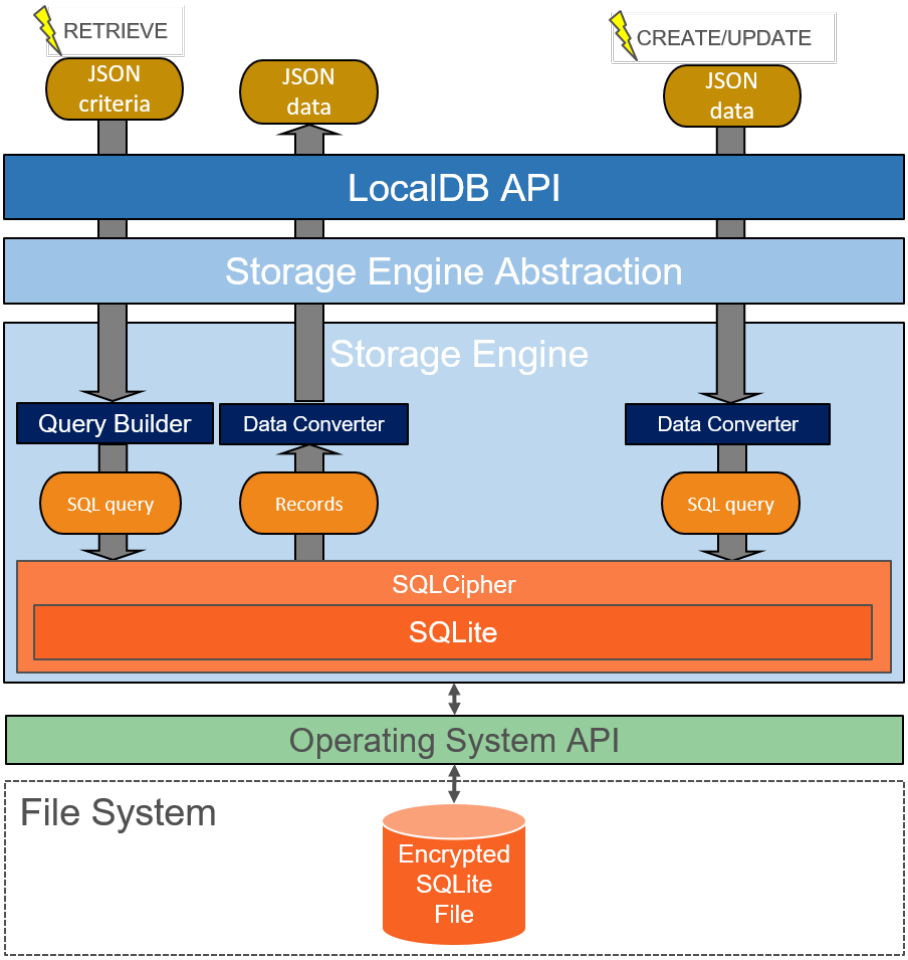


FIGURE 5.4: LocalDB library architecture

2. Translate criteria in order to build a valid SQL query
3. Perform the query
4. Take the result records
5. Convert records data into JSON data
6. Pass JSON data to the up-level module

And as counterpart, whenever an **add** or **update** item operation is requested, the storage engine must:

1. Take up-coming JSON data
2. Wrap data in order to build a valid insert or update SQL query

3. Perform the query

4. Handle the success or fail result

5.7.3 Design

In the following we present the class diagram with a new class called SQLiteStorageDelegate that implements the low level functionalities needed to store and retrieve persistent data items. SQLiteStorageDelegate must extend the StorageDelegate base class and implement all abstract methods need to perform required operations.

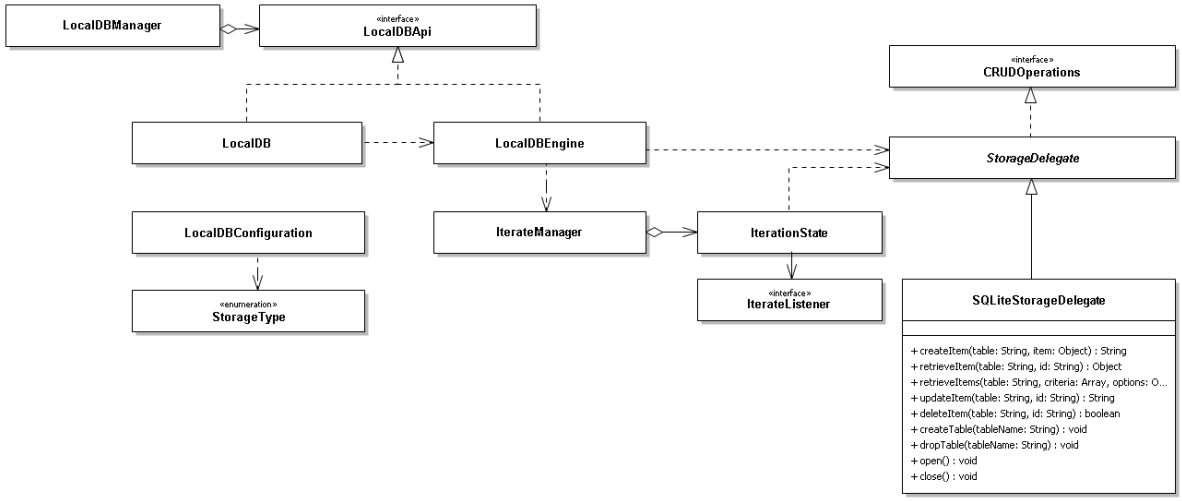


FIGURE 5.5: LocalDB library class diagram with SQLiteStorageDelegate

5.7.4 SQlite JSON1 extension exploitation

In the following we show how SQLiteStorageDelegate leverages the SQlite JSON1 extension in order to store and query JSON data.

Create table In order to store data into the SQLite database, it is needed first to create a table for each entity type. The table can be very simple, just two columns:

- The row identifier. An auto-incremented integer can be used

- The JSON data content. It has to be stored as plain text

Listing 5.3 shows an example of the SQL command to create the table.

LISTING 5.3: Create table command

```
CREATE TABLE ExampleTable (
    _id     INTEGER PRIMARY KEY AUTOINCREMENT,
    jsonData    TEXT
);
```

Insert JSON data After the creation of the table, JSON data storing can start. It can be used the *json(X)* SQL function to verify that its argument X is a valid JSON string and to obtain a minified version of that JSON string. Listing 5.4 shows an example of the SQL command to create insert a new data row into the database.

LISTING 5.4: Insert JSON data command

```
INSERT INTO ExampleTable (jsonData)
VALUES (
    json('{"localId":"1234","field":"example␣text"}')
);
```

Retrieve JSON data Now the inserted data can be retrieved. It can be used the *json_extract(X,P)* SQL function which extracts and returns one or more values from the well-formed JSON at column *X*, using *P* as path for the JSON value is wanted to extract. Listing 5.5 shows an example of the SQL command to retrieve inserted data from the database.

LISTING 5.5: Retrieve JSON data command

```
SELECT jsonData
FROM ExampleTable
WHERE json_extract(jsonData,'$.localId') = '1234';
```

Update and Delete JSON data For completeness, we show how to update and delete data in Listing 5.6 and Listing 5.7.

LISTING 5.6: Update JSON data command

```
UPDATE ExampleTable
SET   jsonData=json('{"localId":"1234","field":"updated␣text"}')
WHERE json_extract(jsonData,'$.localId') = '1234';
```

LISTING 5.7: Delete JSON data command

```
1 DELETE
2 FROM ExampleTable
3 WHERE json_extract(jsonData,'$.localId') = '1234';
```

5.7.5 JSON criteria handling

The main effort to be done at this level is to convert JSON criteria seen in 4.5 into a valid SQL query, exploiting the JSON1 extension in order to have a flexible and performant data layer. We need to write an utility class called **SQLiteCustomQueryBuilder** to help us to build SQL valid queries. The idea is to take the JSON criteria, which can be a JSON Array of complex nested JSON Object, and obtain a flatten structure for each criterion of the Array. The flatten structure obtained for each criterion must be a *Dictionary* where each entry is a <*key*,*value*> couple. The *key* represent the JSON path from the root node to a leaf node of the JSON tree and the *value* is the value of the JSON node for that path. Once obtained this *Dictionary*, we can iterate it in order to build the *WHERE* expression for the SQL query. For each entry in the *Dictionary* we want to obtain an expression like *json_extract(X,P) = value* that must be putted in *AND* with others. The expression obtained for each *Dictionary* must be putted in *OR* with other expressions in order to impose that the wanted JSON data must match at least one criterion.
In Listing 5.8 we present the **buildDictionary** algorithm used to build a *Dictionary* from a criterion. *Any* type is used to indicate that the value of a variable of type *Any* can be either a JSON Object or a base value like string, number or boolean. The algorithm is called initially passing to it an empty string as initial path, the JSON Object criterion as value and a reference to an empty *Dictionary*. Then, recursively, whenever it finds a nested JSON Object as the current value do the following:

- iterate over the its keys
- building a new path by concatenating the current path and the current key
- extract the value for the current key from the JSON Object
- call the **buildDictionary** passing to it the new path, the extracted value and the reference to the *Dictionary* to be built

Otherwise, it adds to the *Dictionary* a new entry using the couple <*path*, *value*>

LISTING 5.8: buildDictionary algorithm

```
input: String PATH, Any VALUE, Dictionary DICT
output: Object

begin
    if VALUE instanceof Object
        foreach key in VALUE.keys
            path ← PATH + "." + key
            val ← VALUE[key]
            buildDictionary(path, val, DICT)
        end
    else
        DICT.add(PATH, VALUE)
    end
end
```

In the following an example of the desired SQL query output (Listing 5.10) starting from a JSON criteria (Listing 5.9) is shown. In this example it is wanted to retrieve items whose *entity.company.city* field is equal to "*Pisa*" and *entity.company.type* field is equal to "*srl*" or items whose *entity.company.city* field is equal to "*Firenze*" and *entity.company.type* field is equal to "*spa*".

LISTING 5.9: JSON criteria

```
[
  {
    "entity" : {
      "company" : {
        "city" : "Pisa",
        "type" : "srl"
      }
    }
  },
  {
    "entity" : {
      "company" : {
        "city" : "Firenze",
        "type" : "spa"
      }
    }
  }
]
```

LISTING 5.10: Select with criteria

```
SELECT jsonData
FROM ExampleTable
WHERE (
  (
    (json_extract(jsonData,'$.entity.company.city') = 'Pisa')
    AND
    (json_extract(jsonData,'$.entity.company.type') = 'srl')
  )
  OR
  (
    (json_extract(jsonData,'$.entity.company.city') = 'Firenze')
    AND
    (json_extract(jsonData,'$.entity.company.type') = 'spa')
  )
);
```

JSON data indexing It is also possible to index some data inside the JSON stored into the database using the SQLite *Indexes On Expressions*. This functionality allows to reduce the execution time of a query involving an indexed JSON field. In the following some examples about *Indexes On Expressions* using *json_extract* function are shown.

LISTING 5.11: Unique index using json_extract

```
CREATE UNIQUE INDEX Example_localId_index
ON ExampleTable (json_extract(jsonData, '$.localId'));
```

LISTING 5.12: Index using json_extract

```
CREATE INDEX Example_field_index
ON ExampleTable (json_extract(jsonData, '$.field'));
```

Chapter 6

Performance Evaluation

After the implementation of the library, we integrated it into the Apparound Framework in order to test performance against the old solution. For this work, we implemented and integrated the library only for the Android platform.

6.1 Setup

For the performance tests, we use an application developed using the Apparound Framework and a pseudo-real dataset whose logical structure is shown in Figure 6.1.

We need to introduce some details about the data involved on the performance tests. There are several entities involved, that are:

- Account
- Contact
- Customer
- Lead
- Opportunity
- Quote
- Task

Even if the logical structure appears a little bit complicated, it is only needed to know that the entity *Account* is the main entity to which other entities are linked.

[1] 1SF entities belong to SalesForce provider https://www.salesforce.com

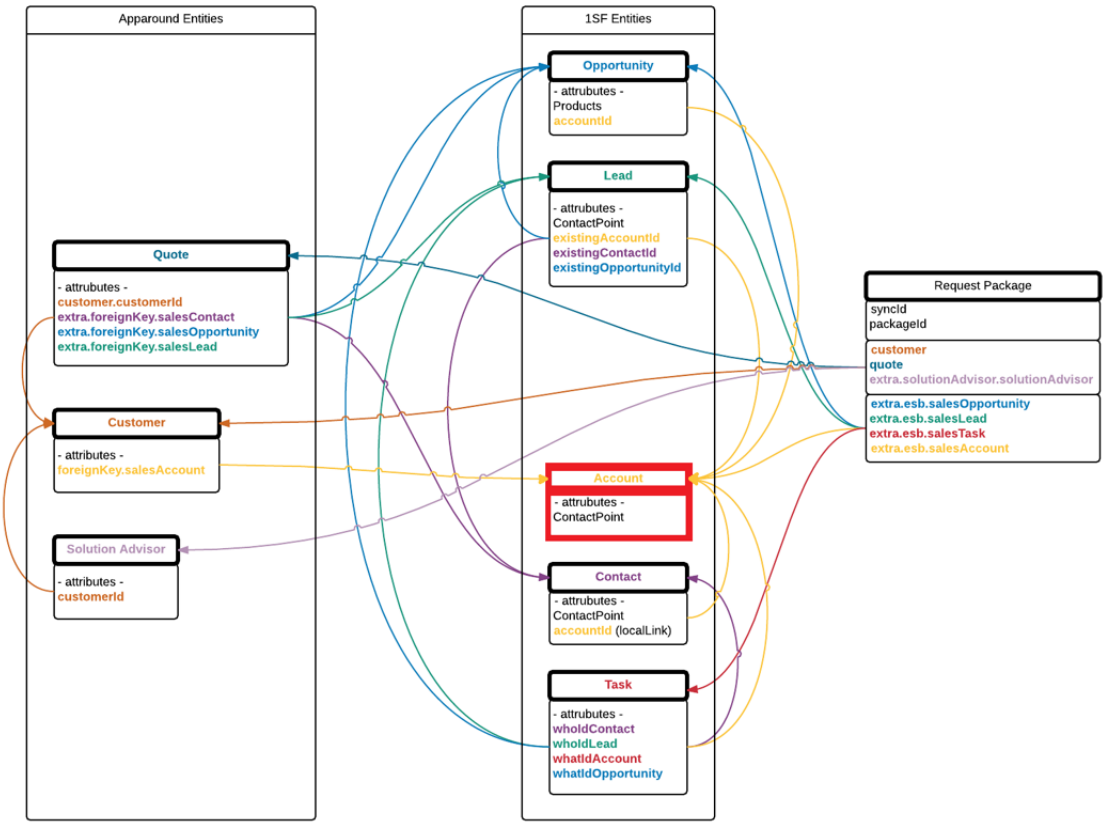


FIGURE 6.1: LocalDB logical structure sample[1]

So, starting from an instance of an *Account* we can retrieve all related instances of other entities using the *Account* identifier.
We need also to introduce some statistics about this scenario, relatively to the number of items for each entity, the average size of an item, the approximate size of the table into the disk, the average size of a table file and the number of files needed by old LocalDB to store them. In Table 6.1 we summarize these statistics. We have one file for each *Quote* item due to some implementation requirements that impose this. Files of other entities have a different average size because the add item algorithm inserts a new item at the end of a file if the current file size does not exceed the maximum size (10 KBytes) else it creates a new file, so some files are a little bit larger. There is also to take into account that each file has some extra informations as shown in Section 4.1.1.

TABLE 6.1: LocalDB scenario

Entity	Number of items	Average Item size (bytes)	Total size (bytes)	Average File size (bytes)	Number of files
Account	186	1.850	344.100	11.400	30
Contact	179	1.850	331.150	11.000	30
Customer	600	630	378.000	10.500	36
Lead	250	2.900	725.000	11.600	63
Opportunity	148	2.300	340.400	10.500	32
Quote	584	3.600	2.102.400	3.600	584
Task	139	1.500	208.500	10.500	20

6.2 Tools

In order to perform the tests, we wrote some scripts in JavaScript that call the native APIs through the interface provided by the framework. We use the tool *Chrome DevTools* provided by *Google Chrome* browser to inspect and debug the *WebViews* present in our application. This tool allows also to run JavaScript snippet using the *WebView* environment. We chose to proceed in this way in order to test common cases, simulating the user interaction with a part of the application running in a WebView. In such way, we have almost the same overhead needed to transfer data between native and JavaScript side. Due to some limitation of the scripts, we also perform a manual test doing a real journey inside the application, in order to test a more complex data flow and processing. We prepare a log system to collect informations about the execution time of the involved operations. Results are averaged to obtain statistically sound conclusions.

As device for the tests, we use a Samsung Tab S2 running Android 6.0.1 with an Octa-Core CPU at 1.9 GHz, 3 GB of RAM. We also use an older device during the journey test, a Samsung Galaxy Tab 4 running Android 5.0.2 with a Quad-Core CPU at 1.2 GHz and 1.5 GB of RAM.

6.3 Tests

6.3.1 Retrieve items with non-indexed field test

The test consists on calling the *findItems* API from JavaScript code. We use the *localId* as identifier for the item we want to retrieve. The localId this time is not an indexed field. We repeat the test 100 times for each entity, collecting the execution time and then the average execution time is calculated. The results are summarized in Figure 6.2. We highlight in green the execution time on native side and in orange the overhead to transfer data between the native side and JavaScript side. We can see that new LocalDB on native side is faster than old solution. The small difference in the overhead time is due to some optimizations in the data serialization.

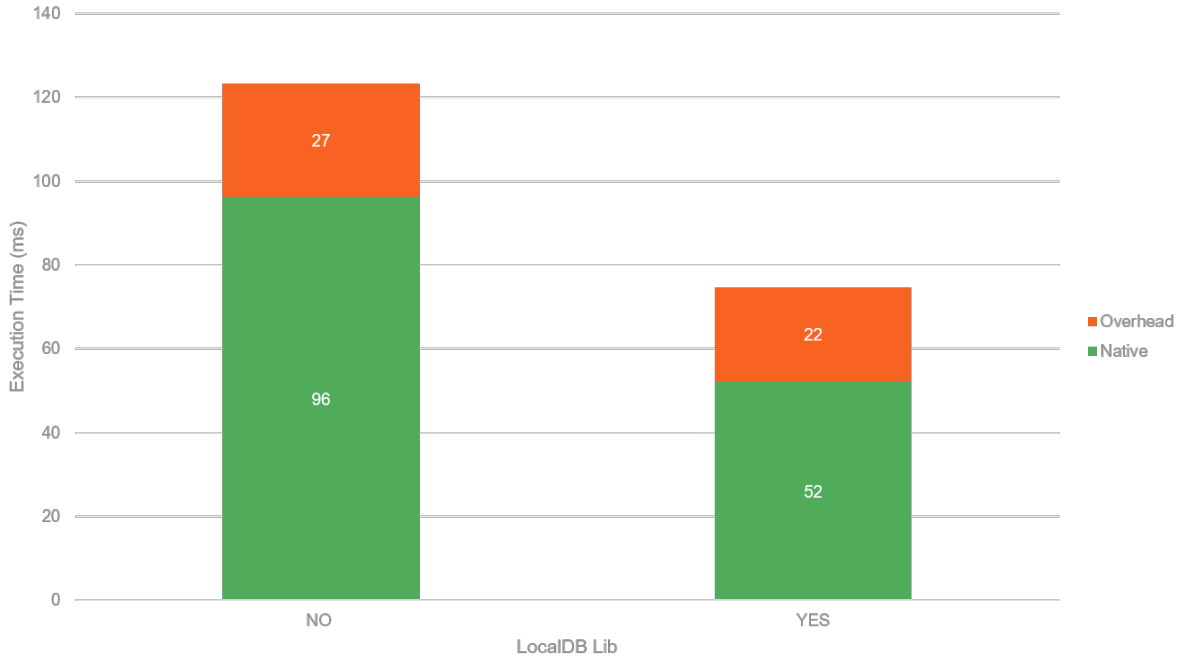


FIGURE 6.2: Retrieve on non-indexed field test results

6.3.2 Retrieve items with indexed field test

The test consists on calling the *getItem* API from the JavaScript. We use the *localId* as identifier for the item we want to retrieve. The localId is an indexed field with *UNIQUE* clause. We repeat the test 100 times for each entity, collecting the execution

time and then the average execution time is calculated. The results are summarized in Figure 6.3. We highlight in green the execution time on native side and in orange the overhead to transfer data between the native side and JavaScript side. We can see that new LocalDB on native side is faster than old solution. The small difference in the overhead time is due to some optimizations on data serialization.

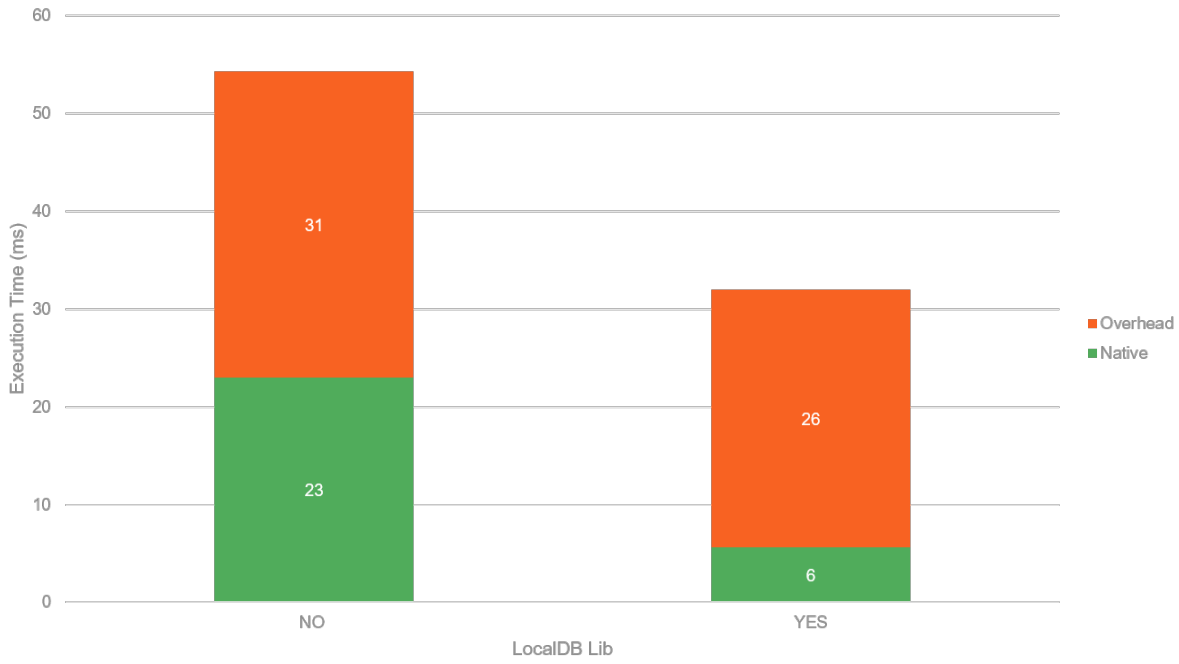


FIGURE 6.3: Retrieve on indexed field test results

6.3.3 Advanced retrieve items test

The test consists on iterating over the items of the entity *Account* and retrieve all related linked items of entities *Contact, Lead, Opportunity, Task , Customer and Quote* using the *findItems* API. On the new LocalDB library, an index on fields used to link items is added in order to improve performance. The results are summarized in Figure 6.4 on which is shown the average execution time of *findItems* operation for each entity type. In Figure 6.5 we summarize the average global execution time and can see that new LocalDB is, on average, about 4.5 times faster than the old solution. In Figure 6.6 we show the RAM usage comparison between the test done with the new LocalDB Library and the old solutions on Android platform. Due to the accuracy of the memory management guarantee by the underling SQLite library,

we can see that there is a better memory usage on our solution, making the Android garbage collection less intensive. This is important because, even if there is a little bit more RAM usage, the major impact on performance is the Garbage Collector action so making it less intensive guarantee a better device resources utilization.

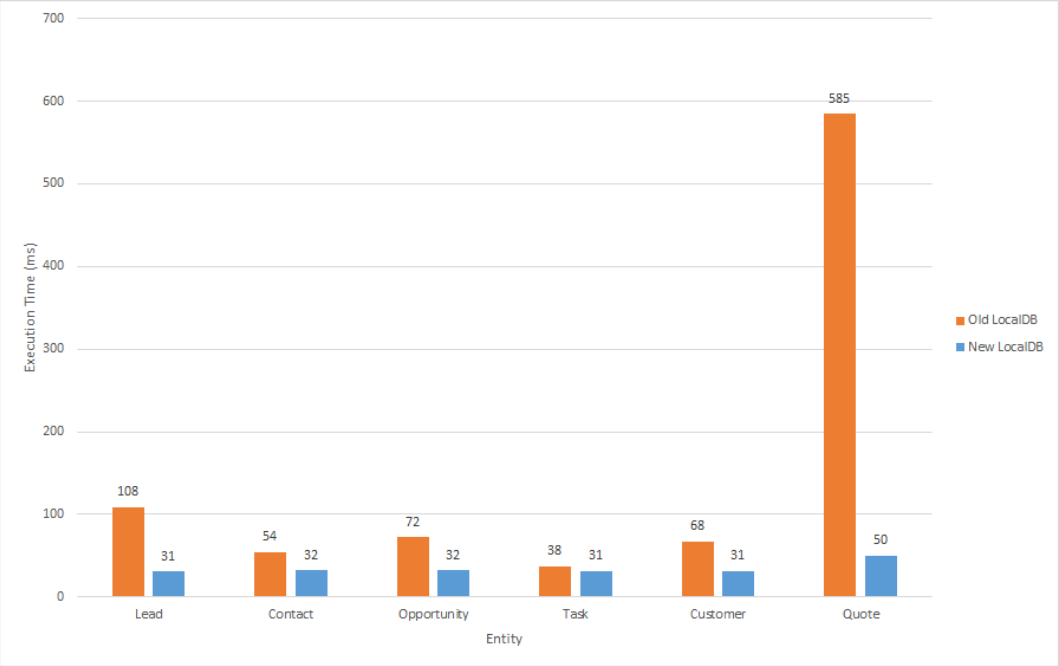


FIGURE 6.4: Advance Retrieve test results (for each entity)

6.3.4 Manual journey test

The test consists in a manual journey inside the application that aims to produce some local data to be sent to the server. After the data are sent to the server, the response is processed and the local data are updated with the processed data. The Figure 6.7 outlines these operations.

We need to add information about internal logics. The data update, due to relation between entities, implies that every time the entity *Account* is modified, it is needed to push update to every related entity item. The Figure 6.8 clarify this procedure, highlighting that many operations are required in order to push the update to every item involved.

Now we can see the results of this test in Figure 6.9. It is clear that again our solution is better in terms of execution time respect to the old solution. We test

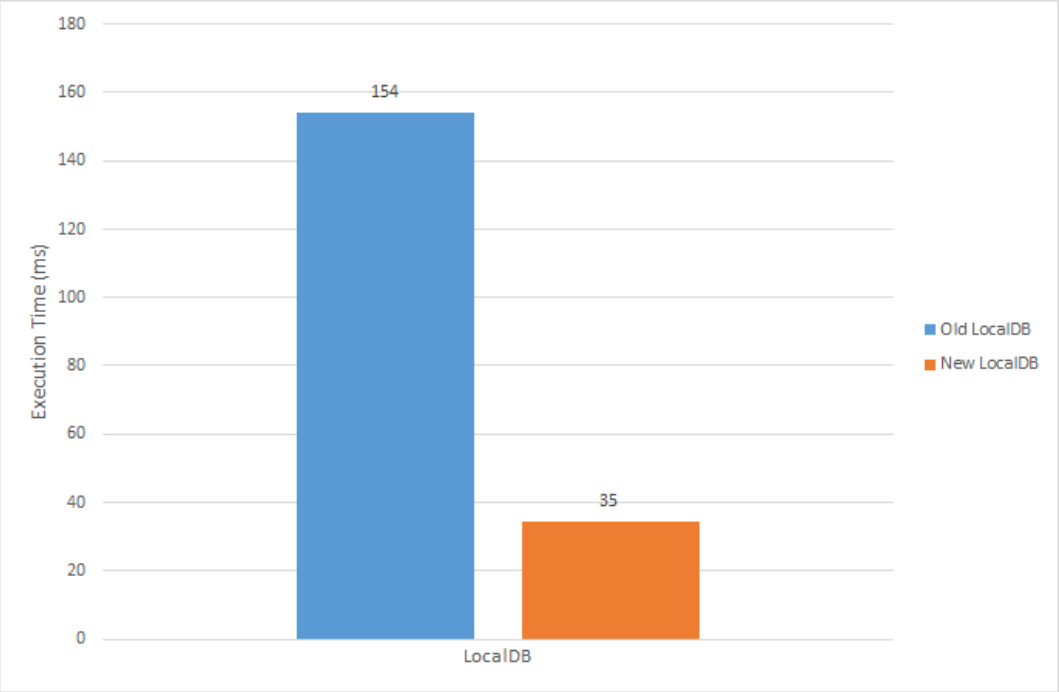
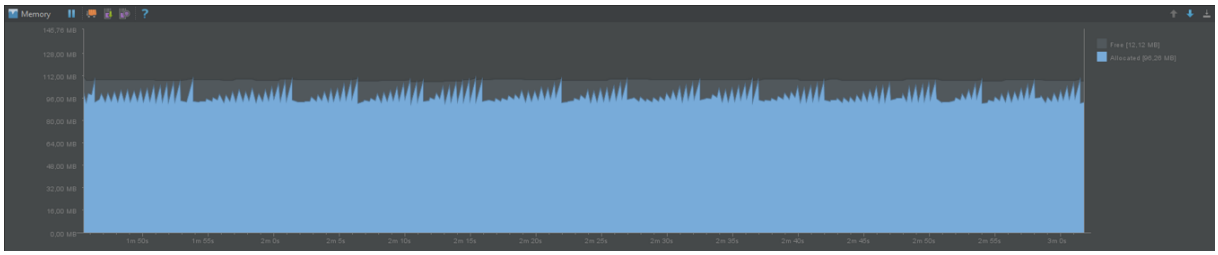
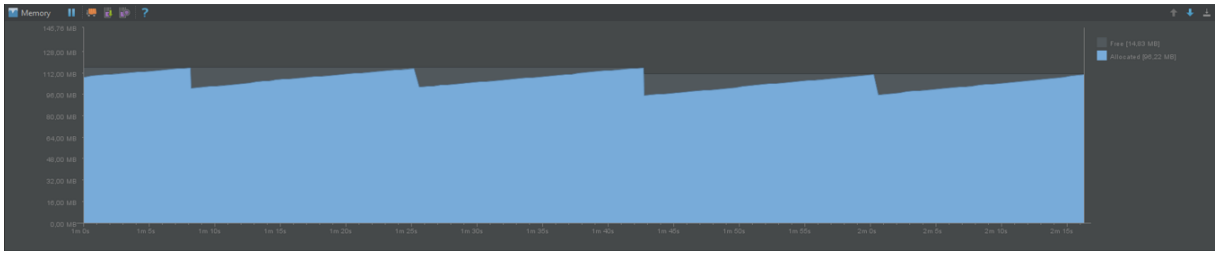


FIGURE 6.5: Advance Retrieve test results (average)



(A) Old LocalDB



(B) New LocalDB

FIGURE 6.6: RAM usage comparison

this journey also using an old Android device. The use of the new library increase significantly the data layer performance.

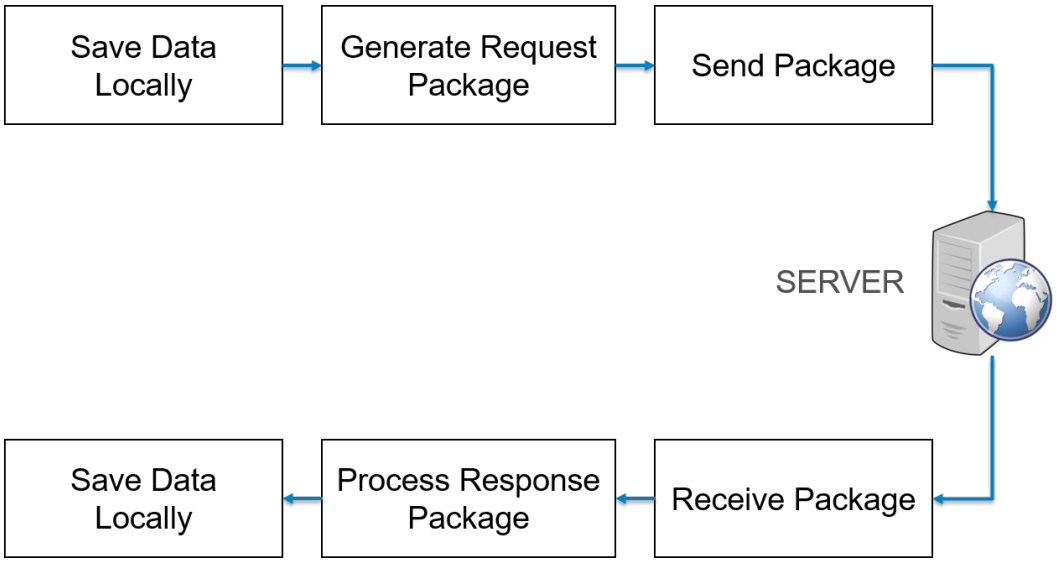


FIGURE 6.7: Data processing operations

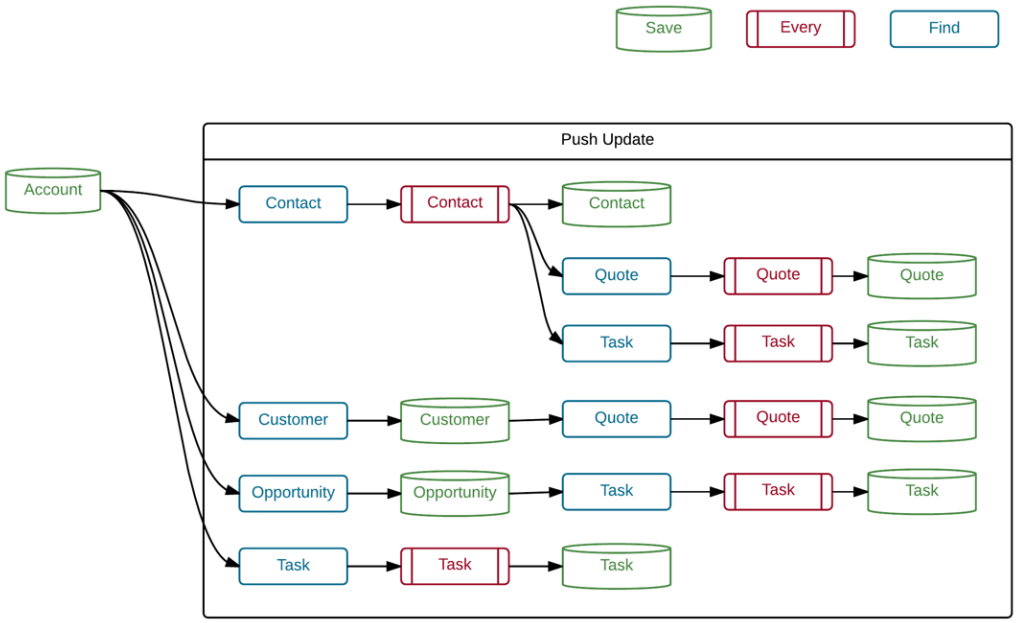


FIGURE 6.8: Push updates flow when an Account is modified

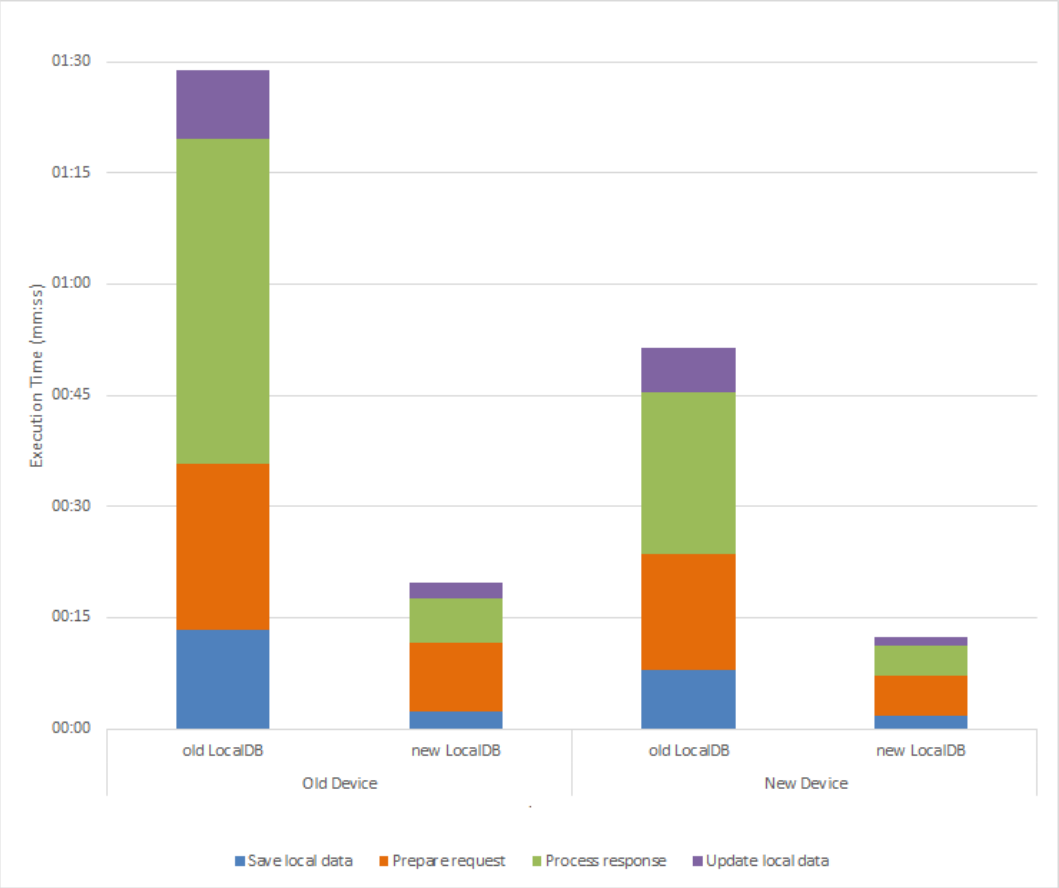


FIGURE 6.9: Complete journey test results

6.4 Summary

The table 6.2 summarize the result obtained from the performance tests. We use the *Speedup Formula* applied to the execution time in order to compare the performance increase between the two solutions:

$$S_E = \frac{E_{old}}{E_{new}} \tag{6.1}$$

TABLE 6.2: LocalDB performance summary

TEST	Old LocalDB	New LocalDB	Speedup abs	Speedup %	Time Gain
Retrieve on non-indexed field	123 ms	74 ms	1,66	66%	40%
Retrieve on indexed field	54 ms	32 ms	1,69	69%	41%
Advanced Retrieve	154 ms	34 ms	4,53	353%	78%
Manual Journey	51,20 s	12,4 s	4,13	313%	76%

Chapter 7

Conclusions

In this thesis work, we presented a solution to store and retrieve efficiently semi-structured data in JSON format. We presented the context on which this work has been done and the reasons to use JSON as data interchange format. We analysed existing solution and state-of-the-art for data persistence on mobile devices, evaluating pros and cons. We designed and implemented a solution to solve the problem of storing and retrieving efficiently JSON data using the well-know library SQLite together with an extension useful for managing JSON content stored into the database. We presented details about the architecture and implementation of our solution that has been deployed on the Android platform. The same solution can be easily deployed on other platforms like iOS and Windows. We integrated the library into the proprietary framework belonging to Appaorund, where this work has been carried out. At the end, we evaluated performance of the new solutions compared against the old one. We found that our solution has better performance respect to the old one and that can be safely used on enterprise applications to guarantee a better user experience. The main difficult was to design and develop a new standalone library sufficiently general, flexible, adaptable to many contexts and easily integrable into the custom framework.
The solution has been integrated as part of enterprise products, starting the porting of this solution on iOS and Windows platforms.

7.1 Future work

As future work we identify some relevant improvements to extend out library.

- Extend the high level query language in order to build more complex queries, exploiting the power of the underling storage engine

- Provide an integrated and configurable synchronization manager
- Provide a way to monitoring changes on the underling stored data

Appendix A

The JSON Data Interchange Standard

JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write. It is easy for machines to parse and generate. It is based on a subset of the JavaScript Programming Language. JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages, including C, C++, C#, Java, JavaScript, Perl, Python, and many others. These properties make JSON an ideal data-interchange language.

JSON is built on two structures:

- A collection of name/value pairs. In various languages, this is realized as an object, record, struct, dictionary, hash table, keyed list, or associative array.
- An ordered list of values. In most languages, this is realized as an array, vector, list, or sequence.

These are universal data structures. Virtually all modern programming languages support them in one form or another. It makes sense that a data format that is interchangeable with programming languages also be based on these structures:

object An *object* (A.1) is an unordered set of name/value pairs. An object begins with (left brace) and ends with (right brace). Each name is followed by : (colon) and the name/value pairs are separated by , (*comma*).

array An *array* (A.2) is an ordered collection of values. An array begins with [(left bracket) and ends with] (right bracket). Values are separated by **,** (*comma*).

value A *value* (A.3) can be a string in double quotes, or a number, or true or false or null, or an object or an array. These structures can be nested.

string A *string* (A.4) is a sequence of zero or more Unicode characters, wrapped in double quotes, using backslash escapes. A character is represented as a single character string. A string is very much like a C or Java string.

number A *number* (A.5) is very much like a C or Java number, except that the octal and hexadecimal formats are not used.

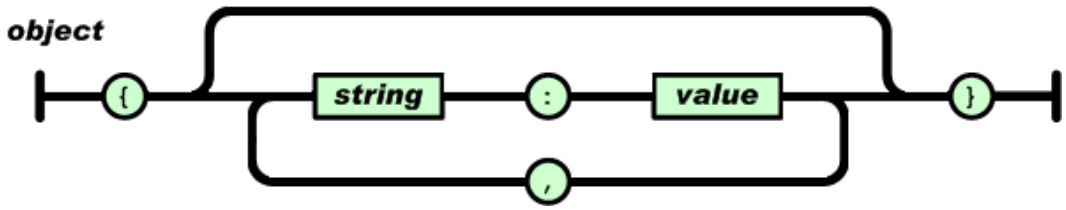


FIGURE A.1: JSON object

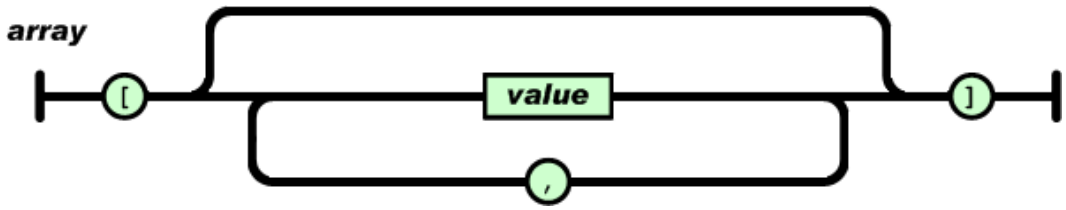


FIGURE A.2: JSON array

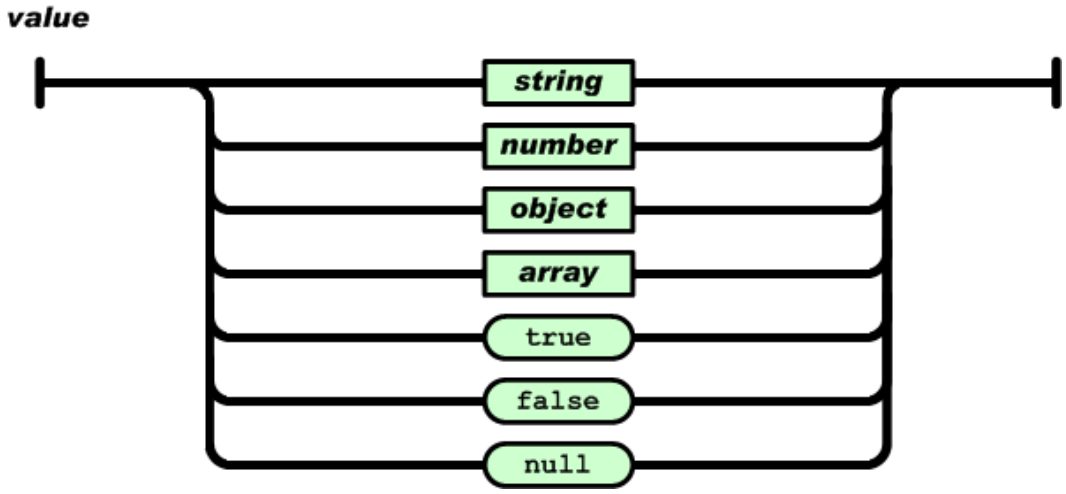


FIGURE A.3: JSON value

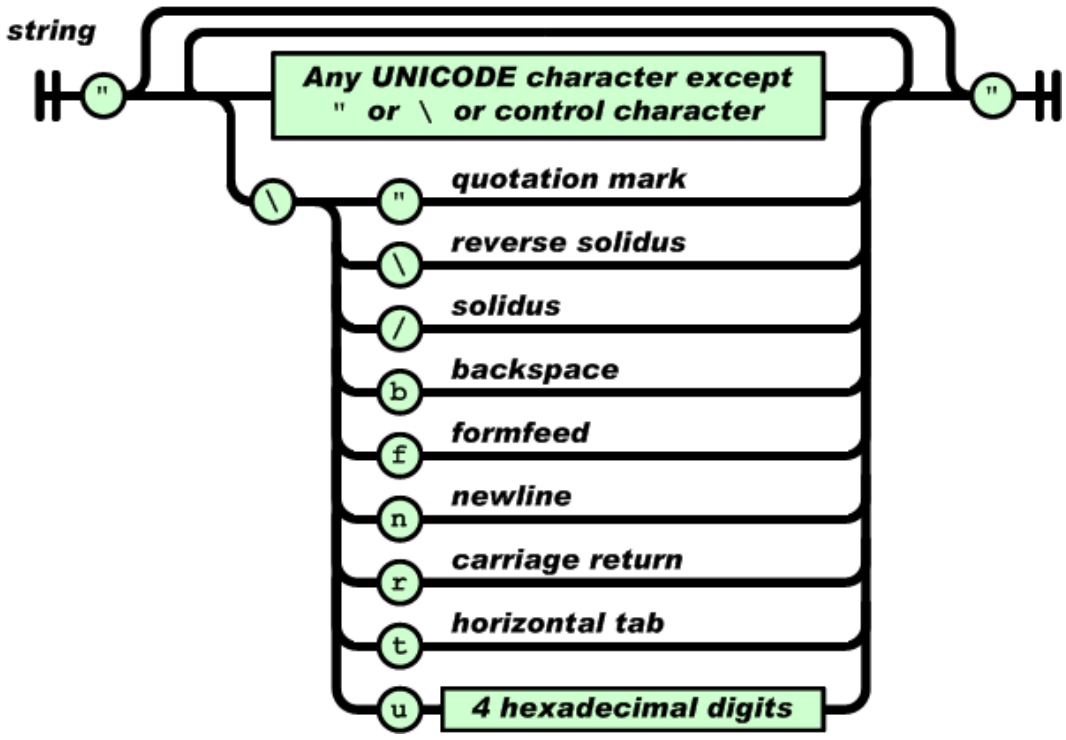


FIGURE A.4: JSON string

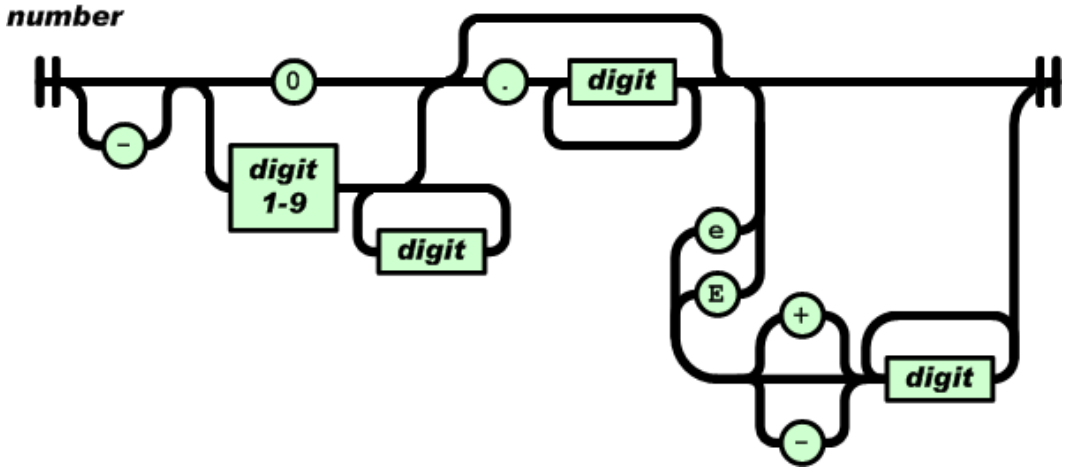


FIGURE A.5: JSON number

Bibliography

[1] Zhen Hua Liu, Beda Hammerschmidt, Doug McMahon. *JSON Data Management – Supporting Schema-less development in RDBMS*. Oracle Corporation, 2014.

[2] Anton Adamanskiy, Andrey Denisov. *EJDB - Embedded JSON database engine*. Fourth World Congress on Software Engineering, 2013.

[3] Tokyo Cabinet: a modern implementation of DBM
`http://fallabs.com/tokyocabinet/`

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

[5] *Android platform architecture*
`https://developer.android.com/guide/platform/index.html`

[6] *Building Web Apps in WebView*
`https://developer.android.com/guide/webapps/webview.html`

[7] *What is a Hybrid Mobile App*
`http://developer.telerik.com/featured/what-is-a-hybrid-mobile-app/`

[8] *Apache Cordova*
`https://cordova.apache.org/`

[9] *Apparound*
`http://www.apparound.com/`

[10] Abraham Silberschatz, Henry F. Korth, S. Sudarshan *Database System Concepts*, Sixth Edition. McGraw-Hill, International Edition, 2–35, 2011.

[11] Rabi Prasad Padhy, Manas Ranjan Patra, Suresh Chandra Satapathy. *RDBMS to NoSQL: Reviewing Some Next-Generation Non-Relational Database's*

(IJAEST)INTERNATIONAL JOURNAL OF ADVANCED ENGINEERING SCIENCES AND TECHNOLOGIES, Vol No. 11, Issue No. 1, 015 - 030

[12] Mason, R. T. *NoSQL databases and data modeling techniques for a document-oriented NoSQL database*. Proceedings of Informing Science & IT Education Conference (InSITE), 259-268, 2015.

[13] Manoj V. *COMPARATIVE STUDY OF NOSQL DOCUMENT, COLUMN STORE DATABASES AND EVALUATION OF CASSANDRA* International Journal of Database Management Systems (IJDMS) Vol.6, No.4, August 2014

[14] A B M Moniruzzaman and Syed Akhter Hossain. *NoSQL Database: New Era of Databases for Big data Analytics - Classification, Characteristics and Comparison* International Journal of Database Theory and Application, Vol. 6, No. 4. 2013

[15] *SQLite*
`https://sqlite.org/`

[16] Sibsankar Haldar. *SQLite Database System Design and Implementation [Second Edition]* Self-Publishing

[17] *SQLite JSON1 extension*
`https://sqlite.org/json1.html`

[18] *SQLCipher*
`https://www.zetetic.net/sqlcipher/`

[19] *SQlite SSE*
`https://www.sqlite.org/see`

[20] *Couchbase Mobile*
`http://www.couchbase.com/nosql-databases/couchbase-mobile`

[21] *Realm.io*
`https://realm.io/`

[22] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Grady Booch *Design Patterns: Elements of Reusable Object-Oriented Software* Addison-Wesley Professional Computing Series, 207, 1995.

[23] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Grady Booch *Design Patterns: Elements of Reusable Object-Oriented Software* Addison-Wesley Professional Computing Series, 151, 1995.

[24] Abraham Silberschatz, Henry F. Korth, S. Sudarshan *Database System Concepts*, Sixth Edition. McGraw-Hill, International Edition, 1045–1046, 2011.

[25] Milo Martin, Amir Roth *Performance & Benchmarking* `http://www.cis.upenn.edu/ milom/cis501-Fall12/ lectures/04_performance.pdf` Slides of Computer Architecture Lectures, University of Pennsylvania

[26] *The JSON Data Interchange Standard* `http://www.json.org/`