Czech Technical University in Prague
Faculty of Electrical Engineering
Department of Cybernetics

Vladimir Burian

Distributed Database for Mobile Devices

Master's Thesis

Prague, 2013

Thesis Advisor: Ing. Tomas Barina

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Cybernetics

DIPLOMA THESIS ASSIGNMENT

Student: Bc. Vladimír B u r i a n

Study programme: Cybernetics and Robotics

Specialisation: Robotics

Title of Diploma Thesis: Distributed Database for Mobile Devices

Guidelines:

1. Analyze concepts of column family NoSQL databases and describe them in detail.
2. Analyze relational database systems available for Android mobile device platform and describe them.
3. Describe differences between column family NoSQL databases and relational databases. When considering data replication between these types of databases, deduce what are the limitations resulting from hybrid architecture and propose methods to mitigate their effects.
4. Install and document deployment of a database suitable for solving the replication between these types of databases.
5. Design the method of data transfer between relational database on a mobile device and NoSQL database in an IaaS infrastructure.
6. Implement a prototype of the proposed replication method and demonstrate its functionality with a sample application.
7. Document everything properly. Support important architectural decisions with arguments.
8. Provide and measure/estimate basic parameters of the proposed solution.
9. Perform integration and functional testing.

Bibliography/Sources:

[1] Sedivy, Jan; Barina, Tomas; Morozan, Ion; Sandu, Andreea - MCSync - Distributed, Decentralized Database for Mobile Devices – Bengaluru 2012.
[2] Lars Vogel - Android SQLite and ContentProvider – US, 2012.
[3] Williams, D. - Optimal parameter selection for efficient memory integrity verification using Merkle hash trees - Ithaca, NY, USA – 2004.
[4] Eben Hewitt - Cassandra: The Definitive Guide – US, 2010.
[5] Grant Allen, Mike Owens - The Definitive Guide to SQLite – US, 2006.

Diploma Thesis Supervisor: Ing. Tomáš Bařina

Valid until: the end of the summer semester of academic year 2013/2014

prof. Ing. Vladimír Mařík, DrSc.
Head of Department

prof. Ing. Pavel Ripka, CSc.
Dean

Prague, January 10, 2013

České vysoké učení technické v Praze
Fakulta elektrotechnická

Katedra kybernetiky

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: Bc. Vladimír B u r i a n

Studijní program: Kybernetika a robotika (magisterský)

Obor: Robotika

Název tématu: Distribuovaná databáze pro mobilní zařízení

Pokyny pro vypracování:

1. Analyzujte koncepty sloupcových NoSQL databází a detailně je popište.
2. Analyzujte relační databázové systémy dostupné pro mobilní platformu Android a popište je.
3. Popište rozdíly mezi sloupcovými NoSQL a relačními databázemi. Pokud budeme uvažovat replikaci mezi těmito typy databází, tak vyvoďte omezení, které z této hybridní architektury plynou, a navrhněte způsoby mitigace jejich dopadů.
4. Nainstalujte a zdokumentujte nasazení vhodných databází pro daný problém.
5. Navrhněte způsob přenášení dat mezi relační databází na mobilním zařízení a NoSQL databází umístěné v IaaS infrastruktuře.
6. Naimplementujte prototyp navržené replikace a na vzorové aplikaci demonstrujte jeho funkčnost.
7. Vše řádně zdokumentujte. Důležitá architektonická rozhodnutí vždy podložte argumentací.
8. Proveďte odhad a měření základních parametrů navrženého řešení.
9. Vytvořte integrační a funkční testy.

Seznam odborné literatury:

[1] Sedivy, Jan; Barina, Tomas; Morozan, Ion; Sandu, Andreea - MCSync - Distributed, Decentralized Database for Mobile Devices – Bengaluru 2012.
[2] Lars Vogel - Android SQLite and ContentProvider – US, 2012.
[3] Williams, D. - Optimal parameter selection for efficient memory integrity verification using Merkle hash trees - Ithaca, NY, USA – 2004.
[4] Eben Hewitt - Cassandra: The Definitive Guide – US, 2010.
[5] Grant Allen, Mike Owens - The Definitive Guide to SQLite – US, 2006.

Vedoucí diplomové práce: Ing. Tomáš Bařina

Platnost zadání: do konce letního semestru 2013/2014

L.S.

prof. Ing. Vladimír Mařík, DrSc.
vedoucí katedry

prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 10. 1. 2013

Prohlášení autora práce

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použitéi informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

V Praze, dne 10. 5. 2013 .....

.....

Podpis autora práce

Acknowledgement

I would like to thank my supervisor, Ing. Tomas Barina, for the patient guidance, encouragement and advice he has provided.

Abstract

The goal of this thesis is to investigate possibilities to replicate data between columnar NoSQL database used as a central storage in an IaaS infrastructure and arbitrary number of Android mobile devices with relational databases. Proposed system is multi-tenant. NoSQL DB serves as a storage for arbitrary number of client databases. And proposed system allows their efficient replication/sharing among arbitrary number of devices. It could be seen as a service synchronizing user data between different devices or as a service providing synchronized database for collaborative work of multiple users. The thesis discusses NoSQL DB Cassandra as it was used to develop the system, characteristics of the system as a whole and methods to efficiently synchronize database changes and meaningfully resolve collisions. Emphasis in placed on a system performance and resistance to failures, especially those caused by an unreliable mobile device connection. Both the server and the client is implemented and their basic principles are described in detail.

Abstrakt

Účelem této práce je prozkoumat možnosti replikace dat mezi sloupcovou NoSQL databází požitou jako výkonné centrální úložiště na straně cloudu a mezi blíže neurčeným počtem mobilních zařízení s relačními databázemi na platformě Android. Navržený systém je víceuživatelský. NoSQL databáze slouží jako úložiště pro libovolný počet klientských databází. A systém je umožňuje snadno a efektivně replikovat či sdílet mezi libovolným počtem zařízení. Na toto se lze dívat jako na službu poskytující možnost synchronizace uživatelových dat mezi jeho zařízeními, nebo jako na službu poskytující synchronizovanou databázi pro účely spolupráce a společné editace více uživatelů. V práci je diskutována především NoSQL databáze Cassandra, protože byla použita při implementaci, dále charakteristiky distribuovaného systému jako celku a metody jak efektivně synchronizovat změny v databázích a smysluplně řešet jejich kolize. Důraz je kladen především na výkonnost systému a jeho odolnost proti chybám, především pak proti chybám způsobeným nespolehlivým mobilním připojením. V rámci práce byl implementován klient i server a jejich základní principy jsou detailně popsány.

Contents

Chapter 1

Introduction

The goal is to develop a multi-tenant service based on a powerful columnar NoSQL database located in a cloud infrastructure allowing seamless replication and sharing of mobile relational databases between mobile devices, particularly Android devices in this case. There are two main reasons to do so. At first it would be availability of a simple method to keep user data synchronized among user devices and backed-up, or a method to allow collaborative modification of a synchronized database. The second reason is that it is unpleasant for users when service or application is dependent on a network connection. This limits its usability usually. The proposed solution solves this problem in a way that user has always the data available in a local database and can modify it regardless of current network connection. And changes made while being off-line can be seamlessly synchronized with cloud and made globally available as soon as there is a network connection again.

In the chapter 2, columnar NoSQL databases are described in detail because their knowledge is crucial to be able to develop system using them as a backend storage. Especially Cassandra database is described as it was used to implement this project. Then relational databases available for Android platform are briefly described in chapter 3.

Differences between both types of databases and especially characteristics of the distributed system as a whole are described in chapter 4. There is also analysis of possible solutions to synchronize modifications made in multiple databases and resolve conflicting modifications.

In the chapter 5, system is proposed and defined in more details with respect to analyses from previous chapters. Installation of a development environment is also described.

Finally server and client implementations are described in chapter 6 and 7 respectively. Description of both sides focuses mostly on internal data organization and processes manipulating that data. Emphasis is placed mostly on a system performance and its resistance to failures caused by unreliable environment.

Chapter 2

Columnar NoSQL DBs

NoSQL databases in general address big data problem which arises with web 2.0 era and global services (i.e. Twitter or Facebook) in recent years in situations where usage of traditional RDBMS is infeasible. They are from the beginning designed with horizontal scalability in mind, e.g. system performance scales well with number of machines present in a database cluster. There is a plenty of various NoSQL databases nowadays, each solving specific big data problem, utilizing specific data model and providing specific set of operations over the data, which is usually by no means as flexible and powerful as SQL is.

Several taxonomies were proposed to categorize NoSQL databases. One of these taxonomies, based on utilized data model, propose 4 categories[5] Key value stores, Document stores, Column family stores and Graph databases. In the following text we will focus specifically on columnar NoSQL databases, but all NoSQL databases have some common attributes.

Data in NoSQL databases are in general usually stored in the denormalized form to minimize number of queries, and so number of I/O operations, disk seeks etc. Ideally only one query and one correspondent read of continuous block of data should be sufficient to fulfil our need for data at that moment. So NoSQL DBs do not provide such a complex queries like JOIN statement in SQL and totally avoid constructions like that.

The next very usual attribute is absence of data types. All the data is treated as uninterpreted chunks of bytes. An exceptions are e.g. document stores which store documents in a JSON format.

NoSQL DBs are commonly said to be schema-free. That is, the data in DB is not stored in entities with some fixed schema (as tables in RDBMS have), but rather schema of each data entry can be different. E.g. each row in a columnar DB can have different columns or each document in document store can have different structure.

Another phenomenon is absence of relations which can be seen in RDBMS. Columnar databases are missing relations entirely. Some document stores have concept of relations, but they are usually not intended for frequent usage and have limited abilities. But this is totally opposite to graph NoSQL databases where relations between entities are basic building blocks and where relations itself have associated data and are as important as entities they connect. This is a clear evidence that NoSQL databases are king of specialized databases each solving specific problem, which can have very different properties.

To provide horizontal scalability, NoSQL DBs utilize mechanisms to partition data and distribute load to different machines in a cluster. And to ensure data availability in a case of node failure or network unavailability (and to improve scalability moreover), they provide possibility to replicate data as a core function.

Columnar NoSQL databases were pioneered by Google's BigTable and other stores are mostly inspired by this one. To name some others: HBase and Hypertable are open source implementations of BigTable. Cassandra is another popular columnar NoSQL DB, it is also based on BigTable but has few distinctions. Although all these databases share basic ideas, they are by no means the same. They have differences, but furthermore these differences are changing quite often as new features are being implemented. So it is quite hard to make comprehensive and especially up-to-date comparison.

In the following text we will focus especially on Cassandra DB version 1.2 [2][10][15], but many described principles are common to all columnar NoSQL databases.

2.1 Data model

Columnar NoSQL databases are the most similar to the traditional RDBMS in terms of the data model used. Data model of Cassandra DB is in the figure 2.1.

The top level component is a keyspace $\langle \mathbf{KS} \rangle$, it is roughly corresponding to database in relational databases. Keyspace is identified by a string. By convention usually one keyspace per application is created. One reason to create more that one keyspace is because some settings (e.g. replication factor) are keyspace-wide.

Keyspace is a container for column families ⟨**CF**⟩ or their extension super column families ⟨**SCF**⟩. These can be seen as counterparts of tables in RDBMS, both have rows and columns, but it is rather only term compliance, principles are different. Column family is nothing more than a sorted collection of rows. Count of column families in a keyspace should be kept low, as is the case also for tables in relational databases.

Each row is addressed by one unique key. Each row itself is a sorted collection of columns. And column, the smallest indivisible entity, is a triplet consisting of column name, value and timestamp. Keys, column names and values can be arbitrary chunks of bytes. Timestamps have type of long integer. Columns can be freely created or deleted per row and they are sorted by column names. Comparator to compare column names can be specified during column family creation and cannot be changed further.

To summarize the data hierarchy in columnar DB: a value stored in a column family is addressed by four parameters:

`[Keyspace][ColumnFamily][Key][ColumnName]`

Super column family adds one level of nesting. (Super)Columns in ⟨**SCF**⟩ are containers

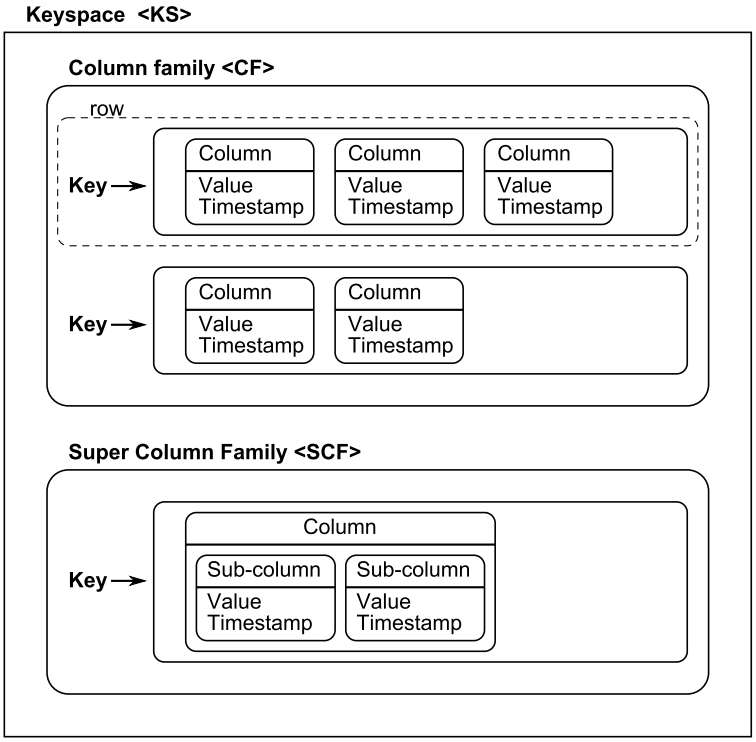

Figure 2.1: Illustration of data model and component hierarchy in Cassandra DB.

for collections of sub-columns which are then containers for column name, value and timestamp triplets. So to address a value stored in a super column family, five parameters needed to be specified:

```
[Keyspace][ColumnFamily][Key][ColumnName][SubColumnName]
```

Another level of nesting is not possible. And it must be remarked, that using super column families is currently not recommended and possibly never will be as it is only remainder of historical development. The reason is that sub-columns are not indexed and must be all deserialized into the memory before any of them can be read. But super column families can be replaced by ordinary column families and design pattern called "composite columns" which will be discussed later.

What is the purpose of column families when each row can contain virtually anything without any relations and constraints to the content of other rows? It is the way the data in column family is physically organized and stored so that we can generally read and write any continual range of columns or whole rows very quickly. The name "column family" can be interpreted as a container (family) for columns which are frequently queried together.

2.2 Design patterns

Design of data storage scheme in columnar NoSQL databases is not driven strictly by some set of rules and semantic meaning of the data, but it is driven by our demand for high system performance. That is, we design such a data scheme so that queries we need to execute are executed as quickly as possible. It is sometimes being said that required queries are the first thing that need to be identified in the process of data scheme design.

There are few very basic design patterns which are new to people who are used to working with RDBMS.

Probably the most basic decision is to design row(s) to be skinny or wide. This categorization is based on count of columns in a row and properties of data used as columns names (e.g. whether column names are static).

2.2.1 Skinny rows

Skinny rows are characterized by relatively small count of columns, which is not varying a lot between other rows in a column family, and by column names which are usually static and same in each row. Skinny rows are most similar to schema of traditional RDBMS table. For example skinny rows are used in a column family storing basic users informations. Each row has then the same, basic set of columns, like user's name, gender, date of birth, etc. That is the pure form of skinny rows.

But it can still contain columns which are not common for all rows, e.g. email. For example each user can have several email addresses and all of them can be stored in one row but several columns. Of course column names for each email address in a row must be exclusive and created in a systematic way, so that we can easily read and identify all email addresses, e.g. "mail:1", "mail:2", ... But there can a better way to do so. Anyway the space used to store email addresses of one particular user is proportional to the number of his email boxes. If he does not have any email boxes, no space is used. And at the same time another user can have virtually unlimited number of email addresses. But what is most important, we can obtain allthe user's email addresses easily and quickly with one query only.

2.2.2 Wide rows

Contrary, wide rows are characterized by highly variable number of columns with column names which are usually not known beforehand and which vary from row to row. In this case, presence of column and its column name carry some information in itself. Common usage of wide rows is to store time series data, like data logs, users' comments, etc. In such cases, columns are given names containing unique timestamp, e.g. time-based UUID, and value contain some useful data like message. Time-based UUID is concatenation of timestamp and machine MAC address, hence it ensures column name uniqueness and causes that records are sorted by time, because columns in rows are sorted by their column names. With such a schema it is easy and very quick to query, for example few of the most recent comments, as they are stored in a continuous memory block.

Wide row is very common and useful design pattern, but it has some pitfalls, which cannot be overlooked. By design, each row is always stored entirely on one node (machine). To take advantage of having cluster of nodes, we have to somehow split records to multiple rows. One possibility is to create one row per hour and use hour-of-day timestamp as a

key to the row. Then records are distributed across all the nodes and we can still quickly query few last records. But if the stream of records being written down is fast, this cannot protect us against occurrence of hot-spots as always only one node serve all the requests on an hourly basis. To solve this problem, rows can be created with shorter time period or records placement in rows do not have to be based only on time, but e.g. time and record category simultaneously, for example. Then rows containing records of different categories will be (probably) distributed across different database nodes and load will be distributed.

2.2.3 Valueless columns

Another pattern which usually appears in conjunction with wide rows are so called "valueless" columns. In this pattern all the data is stored in column names and values are then null.

This is particularly useful when creating custom indexes. For example to get all products of particular category it could be sufficient to read a row which key is the key of category and column names are keys of products. There is no need to have some data in column values then, but in normal situation they would probably contain some useful information like product name, which we probably want to know likewise.

Another usage of valueless column pattern is to implement M:N relations. One column family with valueless columns can be used to assign elements B to elements A, i.e. row keys are keys of elements A and column names are keys of elements B, and vice versa.

2.2.4 Composite column names

And one more pattern, which is also quite common with wide rows, is called composite column name. It means that column names are composed of several parts. Parts can be distinguished for example by position in the column name or they can be separated by colon symbol, etc. For example super column families can be replaced by column families with column names in the form of "column-name:sub-column-name". Then all the sub-columns can be also queried at once, but to get names of sub-columns, returned column names must be parsed. This is a little disadvantage but it still remains recommended approach.

2.2.5 Conclusion

These were the very basic design patterns, but there are still some recommendations which should be accounted during design time. As all values are stored as triplets (column name, value, timestamp), column name is repeated many times in memory, so it is convenient to keep column names short to save memory space. As was written previously, data can be stored in the form of column names, but it is good to know that column name is limited to 64 KB in size.

2.3 Cluster operation

NoSQL databases are distributed systems and to understate how they work, how data is processed and how they are used, several terms and concepts have to be explained.

2.3.1 Partitioning

There is a mechanism called partitioning which assigns rows to arbitrary nodes based on row keys, and so it distributes load and data across cluster nodes. Partitioning could be theoretically handled by a single master node which could easily load balance all slave nodes, but this would be probably system bottleneck and surely would be a single point of failure. Cassandra adopted solution where all nodes are equivalent. All nodes act as a master and any node can fulfil any query regardless of actual data placement. Partitioning is deterministically based only on row key. There are two basic partitioners: `RandomPartitioner` and `ByteOrderedPartitioner` . But any other custom partitioner can be created by implementing appropriate java interface.

RandomPartitioner

`RandomPartitioner` is the most common partitioner and as it is also the default one. Row placement is determined by hash of the row key. All nodes in a cluster are arranged in a ring and each node is assigned a token. Token value acts as a border of row key hashes which belong to that particular node. This way, if the hash function being used is strong and tokens are generated correctly, data is uniformly (and semi-randomly) spread across the cluster. Maybe the only disadvantage is, that read operation must be very probably executed on several nodes concurrently when querying a range of row keys, and this leads to higher nodes utilization and higher network traffic.

ByteOrderedPartitioner

This is not the case for `ByteOrderedPartitioner` as it partitions rows based on their raw values of row keys. So each node then serves particular range of row keys. And because rows are kept sorted by row keys within each node, querying range of row keys is globally cheaper as it would be very probably served by one node only. Inconvenience is that we have to know (or control somehow) distribution of row keys and assign tokens properly to ensure data being uniformly distributed. Or we would end up with data being distributed very non-uniformly. So this partitioner is used only very occasionally in very special cases, where it is actually advantageous.

2.3.2 Replication

Another important topic is data replication. Number of replications (i.e replication factor) is configurable but it is always common within whole keyspace. This is one case when there is no other way than creating more keyspaces per application, if we want to have column families with different replication factors.

Cassandra is datacenter aware and rack aware. It is possible to specify exact number of replications per datacenter and it places replicated data in different racks within datacenter (if possible) to minimize probability of failure.

2.3.3 Consistency

Because columnar NoSQL DBs are distributed system, data (rows) can be replicated across several machines and so could be independently and concurrently modified on several machines, there must be mechanism to keep the data consistent. Consistency in this situation means that all DB nodes handling that particular row have the right version of the data.

In the case of Cassandra there is a possibility that inconsistency can appear, until the replicated data in all locations is synchronized again. This condition is called **eventual consistency**, i.e. consistency is not guaranteed, but it's guaranteed that system will evolve into consistent state eventually. Cassandra is sometimes said to be consistency tunable. It is possible to specify required consistency level for any read or write query in several steps. Consistency level specifies how many replicas have to successfully execute

that particular operation in order to fulfil the query. If for some reason that operation cannot be successfully executed on the required number of replicas, then the query fails. But there is nothing like commit protocol, so the operation still could have been successfully executed on lower number of replicas and eventually propagate to all replicas.

Some of possible consistency levels are:

ANY	Operation has to be successful on any node. It even do not have to be owner of that row if so called hinted handoff is enabled.
ONE	Operation has to be successful on one replica.
TWO	Operation has to be successful on two replicas.
QUORUM	Operation has to be successfully executed on more than a half of all replicas in cluster.
ALL	Operation has to be successful on all replicas.

Conflicts resolution

The question is what happens when data is inconsistent and it is being synchronized. What is the result and how is the data merged? Data conflicts are resolved between columns (triplets: column name, value and timestamp), not between whole rows. And that is what these timestamps are there for – conflict resolution. Winner between multiple versions of the same column is resolved using the following rules sorted by their priority:

1. Column with higher timestamp is preferred.
2. Column deletion is preferred over column write.
3. Column with higher value resolved by binary comparison is preferred.

It means that if there is a record about column deletion and column write both with the same timestamp, then column is deleted as a result and write is ignored. If there are two column writes with the same timestamp, then column version with greater value is winner. The same set of rules applies also during write operation.

The timestamp could be set while preparing modification (write or delete) query to the database. And it is usually good practise to do so. Although it is called timestamp and usually unix time stamp is used as this value, it is not a requirement. Any meaningful number of type long can be used and it is perfectly fine. The above rules still applies.

2.4 Node operation

2.4.1 Data write

NoSQL DBs are generally designed to be able to record massive data streams, so the write operations must be very fast. This requirement is a source of most limitations/relaxations in a sense of data consistency and other guaranties usual in relational databases. Write operation is understood to be write of one column into a row. There is only one more complicated data operation called batch mutate which is discussed later.

Write operation on Cassandra DB node consist only of appending write to a commit log for durability and write to an in-memory structure called a memtable. Then the write operation is acknowledged as finished. There are no other memory operations, random seeks etc. in between so the write operation is really very fast.

SSTables

Next processing of written data is done asynchronously in background. Memtables contain data sorted by row keys and column names. Time to time, as required, memtables are flushed to disk in form of SSTables which have the same data organization as memtables. SSTables and memtales are maintained for each column family independently. SSTables are immutable, once created they cannot be modified. This implies that data is physically stored in sorted order as required but only within SSTables. And there could be a lot of these SSTables, columns of particular row can be distributed across several SSTables and even old versions of a column can be present in various SSTables. It is not convenient situation but it is just trade of for speed.

Compaction

However there is another asynchronous process called compaction. During compaction several SSTables are merged into a new one whereas unneeded data is discarded, e.g. old data or expired data and tombstones. Then the source SSTables are deleted. For the new SSTable it still holds that contained data is sorted by row keys and column names. So during compaction there is an I/O activity and space usage peak, but as a result physical data organization is improved toward the ideal situation and the space usage is usually reduced.

There was mentioned that expired data is discarded during compaction. It is another possibility of Cassandra DB: when writing column it can be assigned optional parameter called TTL – Time-To-Live. This parameter says after how many seconds from the write execution the data is considered deleted. This feature is not dependant on column timestamp value.

2.4.2 Data deletion

Problem with data deletion is that it cannot be just deleted and forgotten. If such a situation occurred in one replica then it would be impossible to find out whether the data was deleted in this replica or it was inserted in other replicas. For this reason the data is never immediately forgotten but so called tombstone is written instead of that to indicate that column is deleted. The process of tombstone write is the same as ordinary data write and tombstones also have timestamp parameter as columns have, to resolve conflicts.

Length of tombstones life is set with parameter `gc_grace_seconds` during column family creation. It default to 10 days. After this period expires and compaction occurs then particular tombstones are discarded forever. It is crucial that any replica is not down longer than the configured period. Otherwise once deleted column could appear again as it was never deleted.

2.4.3 Batch mutation

The basic and also the only data unit to work with is a column, it holds in all situations. However there are so called batch mutations. It is nothing more then possibility to execute set of mutations (either writes or deletes) at once as a batch. Besides it is more efficient to network traffic, it can provide some guarantees. As of recent Cassandra version 1.2 there are two types: ordinary batch mutate and atomic batch mutate.

While ordinary batch mutate does not provide additional guarantees (it is equal to concurrent execution of all mutates: order of execution is random and fail results in a partially applied batch mutate), atomic batch mutate does.

Atomic batch mutate guarantees kind of atomicity – if any part of a batch succeeds then all of it will. It is accomplished with commit logs and system of nodes cooperation. Introduced overhead is roughly 30 %.

Although it guarantees that either all or nothing of a batch will be applied, it does not guarantee isolation. It is still possible to read partially applied batch. And also order of mutates is undetermined.

Only exceptions are mutates within rows. Mutates modifying particular row are guaranteed to be executed in isolation. It is impossible to read row with partially applied batch mutation.

2.4.4 Data read

Data read is rather complicated operation. In-memory bloom filters per SSTable are used to find out whether a node contains required data and in which SSTable the data possibly is. Then SSTables contain indexes to quickly seek the required data.

It is obviously slower operation than data write. Moreover the speed highly depends on physical data arrangement and whether the required data is nicely compacted into one SSTable or it is rather fragmented. This highly individual and affected by the way and frequency of data modifications and cluster and column family settings.

2.4.5 Secondary indexes

Cassandra has possibility of secondary indexes – accessing column family rows by expression with indexed column values. Like many times before, consistency is sacrificed for speed and so the secondary indexes are maintained asynchronously in background. They are internally implemented as hidden column family.

Secondary indexes performance is highest when indexed columns can contain only small set of unique values. The performance is lowest when column with unique values is indexed. In such a case its recommended to implement custom indexing if performance is an issue.

2.5 Programming interface

To be able to design system using Cassandra DB, it is necessary to know what queries for handling of data are available. Historically the first (and most low level) interface to Cassandra DB is provided via Thrift RPC library. In recent versions Thrift interface is

being replaced/complemented with custom solution CQL – Cassandra Query Language. CQL is becoming preferred over Thrift. It mimics SQL and is probably easier to use in the beginning. Additionally CQL is available for more programming languages. But it does not provide any functionality which is not accessible via Thrift. That is why only basics of Thrift API will be discussed.

2.5.1 Thrift API

When concerning direct data manipulation, there are 3 basic kinds of operations: insert, get and remove.

insert() – Inserts/updates just one column.
remove() – Removes one column or whole row based on arguments.
batch_mutate() – Batch execution of a set of insert and remove operations.
atomic_batch_mutate() – Like batch_mutate but with more guarantees.

All data modification functions requires specification of desired consistency level. All operations allow specification of timestamp per column too. When writing column it is also possible to specify its TTL.

It is evident from previous sections, but again there are not operations like locking or transactions as they would be too resource expensive. The only way to execute several operations atomically is to call atomic_batch_mutate() function.

One rule that implies from lack of complex atomic operations, consistency issues and DB operation as such is that queries and data model should be designed so that executed queries are idempotent, i.e. executing the same query repeatedly leads to the same DB state. The reason is that there exist possibility that function call fails but the modification is applied partially or even fully despite of that. Then the query have to be executed again until it is executed successfully to fulfil our requirements, hence the desire for idempotent queries.

As a result, application should be created to accept inconsistencies during normal operation and deals with them without an impact on application functionality and availability.

The most comprehensive set of functions is the one providing various get operations. These are:

get() – Get one column from DB.

get_slice() – Get continuous range of columns given by lowest and highest column name or columns given by a list of column names. Read is performed within a specific row.

multiget_slice() – Same as get_slice() but it performs read of specified columns on a non-continuous list of given row keys in parallel.

get_range_slices() – Same as get_slice() but it operates on a continuous range of row keys given by lower and higher key bounds. It is most meaningful to execute when ByteOrderedPartitioner is used. Then rows are sorted by their value and function truly returns range of keys. If RandomPartitioner is used then function actually does not returns range of keys and result seems to be meaningless. But set of returned keys is deterministic anyway and this function can be used e.g. to iterate over all rows.

get_indexed_slices() – Same as get_slice() but it operates on rows satisfying provided IndexClause. IndexClause is a set of logical expressions with columns being indexed. Clause must contain minimally one equal relation to be acceptable.

Like functions modifying data, these functions take required consistency level as an argument. The result is obtained as a merge of data reads from the specified number of replicas. If the data cannot been read from the given number of replicas then read operation fails.

When querying for a range of columns or keys (or both) it is possible to specify whether the selected range should be reverted and what is the maximum number of items being returned.

2.5.2 Hector client

Although Thrift API is sufficient to fully operate Cassandra DB, it is rather unfriendly. And it is also not considered a good practise to use Thrift API directly. Because of that, various third-party libraries are being developed on the top of Thrift API. One of the most popular is Hector client for Java. Not only provides it better interface, but it also addresses features like connection pool, distribution of queries to different nodes in a cluster, i.e. load balancing, and many others which are crucial to operate cluster successfully.

Chapter 3

Databases for Android platform

To list important relational database systems available for Android, the situation is quite simple. There is only one build-in and officially supported database and also the only one widely used – it is SQLite[11]. And it is not a surprise because SQLite is very popular especially among mobile devices and embedded systems for its small footprint and overall simplicity.

Another RDBMS for Android could be found, although they seem not to be wide-spread. To mention some of them:

- ITTIA DB SQL
- IBM Mobile Database
- Berkeley DB
- SQL Anywhere

All of them with exception of IBM Mobile Database are multi-platform databases and Android is only one of many supported platforms. Whereas IBM Mobile Database is a solution exclusively for Android. All these products are parts of larger product lines and allow seamless synchronization with another relational databases, often only with databases developed by the same company.

They are not in our focus as we are rather concerned about bare relational databases and SQLite would serve our requirements sufficiently.

3.1 SQLite

SQLite is a lightweight database in a form of a self contained software library without any dependencies. It is not a server and cannot be operated over network. Current SQLite version is almost full featured SQL database with few limitations and distinctions.

In terms of supported queries, SQLite implements almost all of SQL92 standard. Only features not fully implemented are triggers, table alternations (it is possible to rename table and add columns) and some types of JOIN operations. Also views are read-only.

Transactions

In terms of operation, SQLite strongly holds transaction ACID properties (Atomicity, Consistency, Isolation and Durability) as these are core functionalities of relational databases. With transactions is connected another issue – locking. SQLite supports only database level locking and it distinguish read locks and write lock. So it is possible to read concurrently from DB, but it is not possible to do concurrent modifications even if they are on different rows or tables. And as a consequence it is also not possible to perform concurrently several transactions containing INSERT or UPDATE operations.

Data types

What is more interesting are peculiarities of SQLite internals and tables schema. SQLite internally supports only five different data types: integer, real, text, blob and null. So whatever data type was specified during table creation, the data is in the end stored as one of these five types. And moreover, SQLite is not statically typed (like major of relational DBs are). There is no type check during data insertion or update, but with one exception only. It is primary key with type of integer. In that case, integer value is required. In all other cases, type of data being inserted/updated is not constrained at all. Data type is not property of table column, but rather property of each individual stored value. Data types defined during table creation are only used to identify preferred column data types. The concept is called type affinity. Even length of text and blob type values is not constrained during table creating. SQLite does not have fixed sized rows, instead of that each stored record is variable in length. These attributes around data types are considered being feature, not imperfection. Backward compatibility with others SQL DBs is not affected anyway. But in the opposite direction compatibility is not guaranteed.

Primary keys

Another curious peculiarity is that every table has unique integer primary key named rowid by design. If table is created with integer primary key, then this defined primary key will be identical with rowid. If table is created with primary key(s) of other types then unique rowid will exist besides them anyway.

3.1.1 Android integration

SQLite is built into Android. Classes accessing SQLite are part of Android API and there are also further classes which could use SQLite indirectly as data source, e.g. `CursorAdapter` . Although it is possible to have arbitrary keys in SQLite tables, to take full advantage of provided advanced API (like mentioned `CursorAdapter`) there is a convention, that tables must contain `_id` primary key of type integer.

Chapter 4

Database differences

Columnar NoSQL DBs differ from relational databases in all aspects as these systems have totally opposite strategies and goals.

One difference is data schema: schema free vs. fixed schema, uninterpreted data vs. static typing (which is not completely true in a case of SQLite) and only one uninterpreted arbitrary row key vs. primary key optionally composed of several columns with given types in general. But these differences are relatively not a big deal.

Much more crucial are differences in database architectures. Relational database is a centralized system (not necessarily in these days, but it is absolutely true for SQLite) with support of ACID transactions, concept of relations as a core functionality and mechanisms to keep DB in consistent state, e.g. constraints on foreign keys, triggers, etc. On the other hand, columnar NoSQL databases are totally opposite. They are decentralized systems being usually eventual consistent, with support of nothing more than batch operations which are absolutely not as powerful as ACID transactions of relational databases. Also concept of relations is completely omitted as well as mechanisms to put some constraints on the data.

ACID properties

Lets see what are the guarantees of ACID transactions in more detail:

Atomicity – Transaction must be applied either as whole or nothing. If any part of transaction fails, then whole transaction fails and DB state is restored (rolled back) to the original state.

Consistency – Transaction cannot change DB to inconsistent state with respect to internal constraints, e.g. violation of references between entities with relations.

Isolation – Even if multiple transactions and executed concurrently, the result must be as if all the transactions were executed serialized, one-by-one. So for example it is impossible to see modifications made in one transaction in another transaction.

Durability – Once a client is acknowledged about a transaction being successfully committed, then the transaction must be preserved. It cannot be just forgotten in a case of DB failure, e.g. server crash or power loss. Transaction is usually considered durable if it is recorded in a non-volatile memory. Of course if non-volatile memory is destroyed, then the transaction is lost too.

CAP theorem

System being developed could be categorized as decentralized and distributed and as such it cannot fulfil all characteristic properties of relational databases and all characteristics of ideal distributed systems at once. Three main characteristics of a distributed system and their relations are discussed in CAP theorem. These are:

Consistency – This property ensures that all nodes in a distributed system have the same, up-to-date data. (Not to confuse with consistency in ACID.) It is not possible to have multiple versions of the same data in various segments of the system.

Availability – System is always available regardless of some indeterministic special conditions or failures in any part of the system. If availability is ensured then it is always possible to query the system and immediate response is expected.

Partition tolerance – System function is not affected by system partitioning, i.e. when parts of network become unavailable and some nodes cannot communicate with each other.

According to the CAP theorem, it is impossible to build distributed system which holds all of these properties, which is consistent, available and partition tolerant in the same time. It is only possible to build distributed system which holds two of CAP properties, remaining one has to be always sacrificed.

4.1 Guaranteed operations

Our goal is to provide Android app developers with local database which would be transparently synchronized with columnar NoSQL database in cloud acting as a central DB. It would be also possible to synchronize the database across multiple devices which moreover need not to be on-line during operation, so the local DB availability would be still guaranteed regardless of current network availability.

Proposed system should be partition tolerant and available, so the data across devices cannot be consistent. And hence the local client DB cannot guarantee transactions to be ACID, because ACID transactions depends on knowledge of operations being executed over the data in whole system.

It means that inconsistencies can occur and must be resolved properly as the data is synchronized with central NoSQL database. In terms of ACID properties, durability cannot be guaranteed as simultaneous modifications can occur and one modification must be always preferred over the others in general. But it should be possible to guarantee operations to meet remaining ACI properties.

The question is, what should be the complexness of operations meeting these requirements? There are three basic options: keep ACI properties at level of transactions or row modifications or column modifications only. The answer is not definite. It is a trade-off between level of guarantees exposed by the system and its performance/scalability. Lets discuss these options in detail:

4.1.1 Level of transactions

Doing DB synchronization on level of transactions would make the system most perfect in terms of its guarantees and similarity to relational databases. But it is not convenient. The reasons are following:

Columnar NoSQL DBs sacrifice most of RDBMS properties to gain good horizontal scalability. There are also distributed relational DBs with support of ACID transactions, but they do not provide such a performance and horizontal scalability. If they did, there would be little reasons to use columnar NoSQL DBs because they introduce many complications with their weak guarantees.

If we were to develop system with ACID transactions on the top of a distributed system with virtually no guarantees, i.e. columnar NoSQL DB, the solution would have to

introduce so much overhead that almost certainly the performance would not be as good as of distributed RDBMS, which provide desired guaranties as a core functionality. So it makes no sense to develop such a system using columnar NoSQL DB.

An example of a freely available distributed relational database feasible to solve this task is MySQL Cluster.

4.1.2 Level of row modifications

This is a much simpler goal. Using NoSQL DB as a central storage in this case would be probably an advantage in terms of performance. Though issues like transactions, consistency or reference integrity are dismissed it is still not a trivial task because the system as a whole is distributed and should be partition tolerant. It implicates that multiple versions of the same data can exist across the system and there must be some mechanism to resolve these conflicts – more in section ??.

4.1.3 Level of column modifications

There is a little difference in complexity in comparison with previous case. But choosing level of operations to support is not only trade-off between performance and guarantees, but it is also question of possible amount of information created in one place and being lost during synchronization because of some conflicts afterwards.

For example consistency at level of row changes keeps most of changes done across the system, but does not guarantee consistency at transaction level (and transactions became useless). And vice versa, consistency at level of transactions can lead to great amount of transactions being cancelled during synchronization, which could be undesirable too.

It is different from case to case and in particular cases it could be desired to preserve concurrent modifications of different columns within the same row. There is not a right answer but lets suppose that this is not a frequent requirement and that it makes sense to preserve only modifications of rows.

4.2 Row conflicts resolution

With respect to previous analysis of options, it is chosen to resolve conflicts at level of row operations. This leads to another set of design options – how to resolve these conflicts.

Lets discuss them from the simple ones to more tricky ones.

4.2.1 Last modification wins

This is the most straightforward solution because it utilizes conflict resolution mechanism built into the Cassandra – timestamps. When a row is changed locally by a client, it is given a current clock timestamp. And when the row is being synchronized, the row with the highest timestamp always wins.

This solutions assumes that whenever any row is modified by a client, he does so because he has some recent informations about the object represented by the row and wants to store them. So it does not consider previous version of the row (and if the assumption holds it is fine). This solution also assumes that clients have synchronized clocks which need not to be true because clients are operated by people.

So there are weaknesses, but it could be reasonable solution in some case and for sure it would be far the most efficient one without any overhead.

4.2.2 Lamport clock

Issues of the previous solution could be resolved with Lamport timestamps. Lamport timestamps are kind of counters or logical clocks (it is not dependent on real time) used to estimate partial causal ordering of events – in our case row modifications. It can be simply described as a version number assigned to every row. Whenever a modified row is being published its version is incremented. Then it is possible to make the following conclusions on the server side when modified row B is received form a client and A is current version of the row on server:

version(A) > version(B) – Version B is either ancestor of A or it is not in the same sequence of modifications as A is (but they have some common ancestors). In this case version B is dismissed.

version(A) = version(B) – In this case it is unclear which version should be preferred in general. If we were to guarantee one single sequence of causal modifications, then all other row modifications with the same version number would have to be declined. Otherwise it would be possible that different clients have rows with the same version number but different content. And it would mean that the sequence of modifications has branched.

Simply said once row with version number X is accepted by server it won't be replaced by any other row with the same version number.

version(A) < version(B) – If the previous case is resolved as described then this case means that B is a direct successor of A. And so it is accepted as a newer row version.

But there is a pitfall. To resolve conflicting rows with the same version number as described previously, it is necessary to be able to execute atomically read-modify-write operation or just conditional write. And this is kind of operations unavailable in Cassandra.

Of course it could be possible to implement these atomic operations on the top of Cassandra and some distributed locking system. But it seems not to be a good idea to use locks whenever a row is being synchronized to the server, when performance is our goal. It would be probably more feasible to use server-side storage with built-in locking mechanism or more complex atomic operations available than making workarounds involving just another system.

What are possibilities not utilizing those atomic operations?

Firstly, it is possible to abandon requirement for a single row history sequence entirely. Just use Lamport clock as a row timestamp instead of the current time of modification on a client side. As a consequence, branches could be occasionally created in a history of a row, actual row state distributed across the system through the server could occasionally skip to an another branch and more frequently these branches would be pruned. But it would still guarantee that a row with a too old parent (and possibly itself being too old) won't overwrite a newer one, without introducing any additional overhead.

To lower the probability of "imperfections" occurrence it is possible to utilize read-modify-write operation though not being atomic. Doing so on server side would introduce one read operation (otherwise unneeded). It is question whether it is worth it.

But similar principle can be used conveniently during synchronization between server and client. If client synchronizes updates from server at first, it can immediately resolve conflicts with local modifications and then upload modifications which are less likely to cause "imperfection" on server-side. The time window where race conditions can occur is longer in comparison with resolution on server side. But it does not introduce any additional overhead as an advantage. So this seems to be reliable solution.

4.3 Detection of client-server DB differences

This section discusses possibilities to detect differences between contents of NoSQL DB on server-side and SQL DB on client-side, e.g. which rows are different and need to be synchronized, which rows have been deleted, inserted etc.

4.3.1 Hash tree

Common method to detect differences between two sets of data entities, e.g. files in file systems or rows in databases, or more accurately their hashes is (Merkle) hash tree. Great advantage is its speed – to find just one difference, the time complexity is logarithmic. Another advantage is that given a data set the hast tree can be built from scratch. Then it can be updated simultaneously as data is being modified, or it can be completely rebuilt the nest time it is needed. On the other hand its

Although the basic idea behind a hash tree is quite simple, it is rather complex data structure. And it is even more complex to use hash trees in a multi-server environment while data is placed in a distributed decentralized storage (NoSQL database). These are few reasons why another method is proposed.

4.3.2 Modifications timeline

The basic idea is that client knows what data have been locally modified as it has control over the local SQL storage. And on server side it is sufficient to maintain list of row modifications/synchronizations monotonous in time as newer row versions are received. Each client keeps pointer to that list according to its last synchronization and then it is able to query for relevant modifications only and stay up-to-date.

Of course it is not necessary to keep list of all modifications, but it would be rather list of rows sorted by the time of last "publication". It is unnecessary to keep information about modification which is outdated.

So far it may seem like ideal solution because it is quite simple, time complexity is constant and it is more memory efficient than hash trees. But it is not completely true in all conditions, there is a complication with deleted rows. Records about deleted rows still must be present in timeline so that clients can be notified about that. Problem occurs when rows are frequently created and deleted. Then it is possible that clients are being

unnecessarily notified about deletion of rows which they have never seen – this is not a case of hast tree.

And then it is hard to estimate when it is safe to delete record about row deletion. One option is to never delete these records. Columnar NoSQL databases are designed to store high volume data. So it could be acceptable, although it is not for free. Or it could be specified that e.g. records older than 3 months will be garbage collected. And if client requests recent modifications and it is found out that there is a gap due to garbage collection, it would be required to download list of all rows and determine iteratively which rows no longer exists.

Assuming that this system won't be used in environments where rows are rapidly created and deleted, it is perfectly acceptable then.

Chapter 5

System overview

Proposed system should have the following attributes:

- Multi-tenancy, stores great amount of different databases.
- Each database can be synchronized among arbitrary number of devices.
- Local database can be operated during off-line state.
- Transactions and consistency checking are dismissed.
- Synchronization is done on a row level.
- It is based on modification/history timelines.
- Conflicts are resolved using timestamps, e.g. Lamport timestamp, preferably configurable.

The central part of this system is a high scalable cloud infrastructure containing cluster of Cassandra DB nodes and Tomcat servers running java servlets and providing clients with RESTful interface.

Workspace consist of several project:

acs-api-base – Library being shared between client and server. It contains classes used to transfer parameters to the server or results from server.

acs-server – Java servlet implementing server side of the proposed synchronization process.

acs-server-provision – Tools and recipes to automatically install tomcat and cassandra nodes. Contains also Vagrantfile instructions to install development environment with one command.

acs-client – Android client with sample application – Todo list.

acs-client-test – Some tests of the system.

5.1 Development environment installation

Development environment can be installed locally as a virtual machine using tool called Vagrant and VirtualBox. It is sufficient to execute the following command:

```
$ cd acs-server-provision/vbox
$ vagrant up
```

Virtual machine is automatically created, required packages installed (especially Cassandra and Tomcat) and configured. If everything succeeds then development environment with one machine, one Cassandra node and one Tomcat instance is available.

The process is driven by `Vagrantfile` configuration. But because virtual machine for development has network card bridged with the host network card, it would be almost certainly necessary to update bridged interface name in `Vagrantfile`.

Concrete instructions how to deploy particular package are in so called cookbooks in `chef/cookbooks` directory. These are basically ruby scripts executed with tool called Chef by Opscode company. Cookbook for Cassandra deployment was custom made. It is simple, but easy to use and is able to deploy multi-node Cassandra cluster. Details are in its readme file.

5.2 Compilation

Compilation of `acs-api-base` and `acs-server` is managed with maven tool. At first api library has to be compiled:

```
$ cd acs-appi-base
$ maven install
```

and then servlet project can be deployed:

```
$ cd acs-server
$ maven tomcat:redeploy
```

Compilation of Android projects `acs-client` and `acs-client-test` is managed by Android plugin for Eclipse. But it is necessary to manually copy compiled `acs-api-base` library to the client libraries.

Another, but hopefully the last problem with development environment, are IP addresses. Because the virtual server is assigned IP by DHCP server in bridged mode, it is necessary to update accordingly IP addresses in some project files. These are:

- `acs-server/pom.xml`
- `acs-server/src/com/vladaburian/acs/server/dao/ACSCluster.java`
- `acs-client/src/com/vladaburian/todoapp/TodoSyncDb.java`
- `acs-client-test/src/com/vladaburian/todoapp/test/TestDb.java`

Chapter 6

Server implementation

In this chapter we will discuss server-side internals, i.e. designed database schema, mechanisms and principles ensuring right system operation and fault tolerance and the exposed REST application interface for clients cooperation.

6.1 Database schema

Designed database schema is in figure 6.1. It may seem confusing at first as there is not a common technique to visualize complicated schema of data in NoSQL DBs. Constants are in bold, all other values are variables. Colons are used as delimiters for composed values. And ellipsis means that row has undetermined number of columns, i.e. it is used to store list of elements – these are all with exception of rows inside `CF:DATA` and `CF:TABLE_META`. Of course all column families have arbitrary number of rows.

To simplify understanding of relations between column families, these are outlined in figure 6.2.

Column families can be split into three categories by their purpose: `CF:DATA` is used to store actual data from client databases. Column families whose name begins with `CF:TIMELINE` are used to implement history timeline – it allows efficient data synchronization with clients. And finally `CF:TABLE_ROWS` and `CF:TABLE_META`.

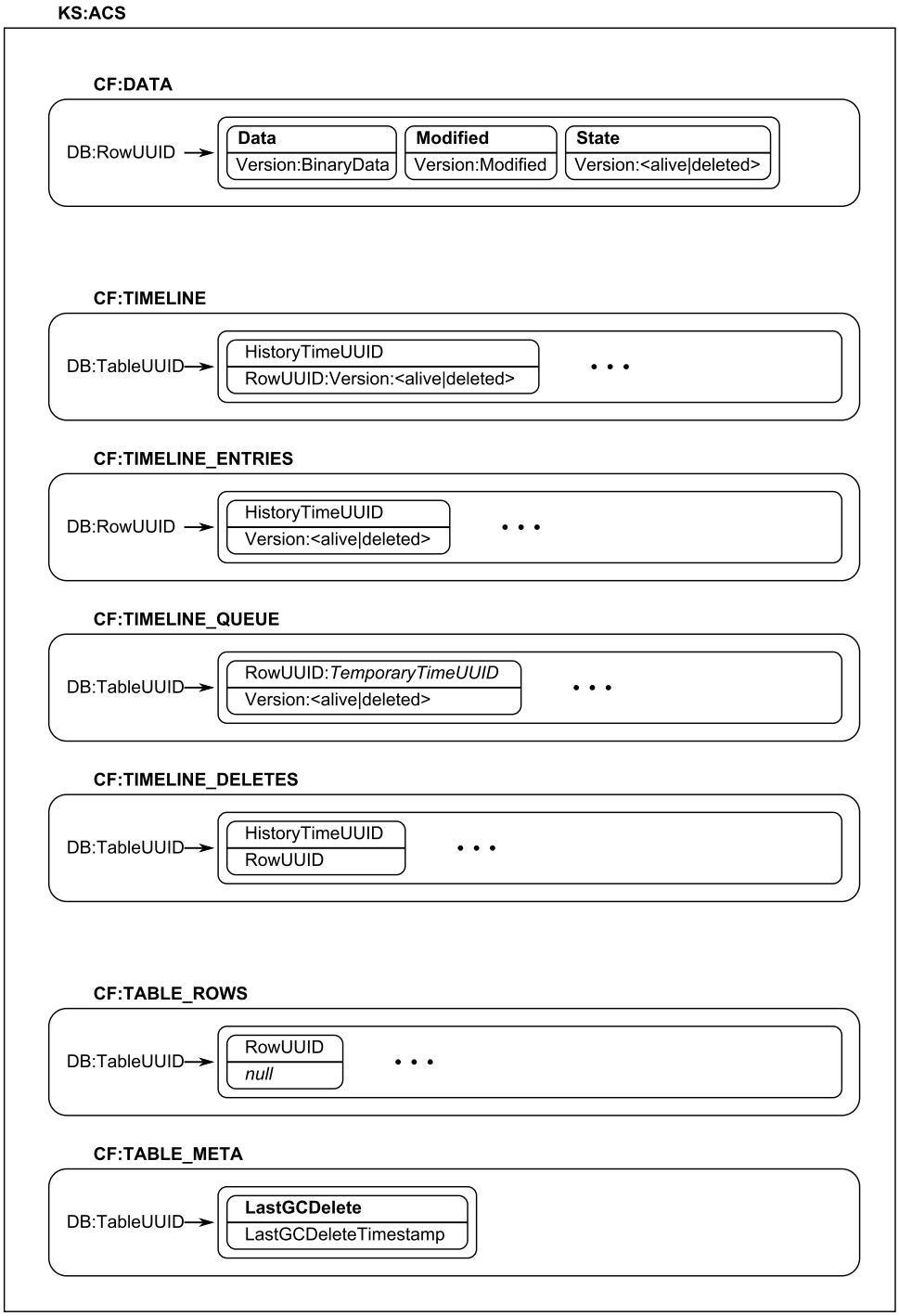

Figure 6.1: Designed database schema used in Cassandra NoSQL DB. (Diagram is analogous to fig. 2.1.

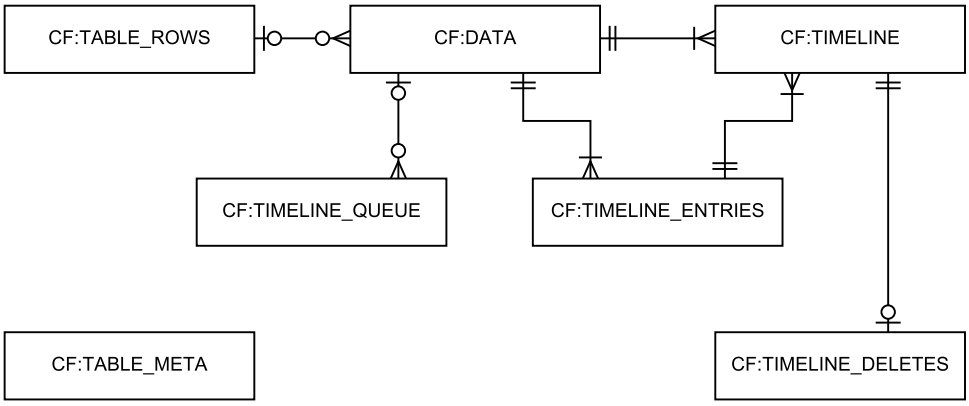

Figure 6.2: Logical relations between CFs from figure 6.1 partially expressed with ER diagram and crow's foot notation. (However it is still little confusing as some entities has meaning of rows, some has meaning of columns.)

6.1.1 Client data storage

All rows from all tables of all users are stored in a single column family – `CF:DATA`. Because the proposed system is multitenant and should be able to serve a great amount of different users, possibly tens of thousands or more, it is not feasible to create new column family per user and table. The reason is that each Cassandra node has a set of in-memory data structures per column family, so the nodes of Cassandra would run out of memory or performance would be bad otherwise.

It is possible to concatenate all the data into one CF because each row is addressed by UUID – Universally Unique Identifier, which is also its primary key on client side. There are multiple types of UUIDs. Random-UUID, which is implemented in Android SDK, is practically unique. But there exists Time-UUID which is guaranteed to be globally unique by design.

There is a summary of variables connected with a row:

DB – Unique identifier of a database. This value is part of all keys in all column families. Its purpose is to isolate values belonging to different databases. Authorization mechanism can be implemented using this identifier.

RowUUID – Unique row identifier and its primary key.

Version – UUID identifying uniquely particular row version. When a row is modified locally on a client side, it is assigned a new version UUID (by client).

BinaryData – Actual row content is stored in binary format as a serialized HashMap mapping column names to values.

Modified – Timestamp associated with row. It could be e.g. time of modification or Lamport clock value, or anything else. It is client responsibility to set this values meaningfully. And it is concurrently value used as timestamp when data was being written to the Cassandra DB.

<**alive**|**deleted**> – This value signalizes whether the row actually exists (is alive) or has been deleted (and these data is there to be able to inform other clients about delete).

6.1.2 History timeline

This is the most tricky part. The main component is `CF:TIMELINE` which contains sequences of modifications (or rather UUIDs of modified rows, lets say timeline entries) by table and which is queried by clients to synchronize data. Each entry is identified with Time-UUID (HistoryTimeUUID) and so the entries are sorted by time. Each client memorize the last known HistoryTimeUUID and queries only for newer timeline entries. And this effectively means that new entries must be appended to the end of timeline. It is unacceptable to insert new entry between any already existing entries.

This is not easy to satisfy in a distributed environment. To accomplish this, distributed locking is involved and that means great performance hit. Because of that, timeline is being built asynchronously in background. It could be said that actual row data and timeline entries are eventually consistent.

When a row is modified, alongside with new row version written into `CF:DATA` there is a request atomically (using atomic batch) written into `CF:TIMELINE_QUEUE` to process this modification, i.e. update `CF:TIMELINE`. TemporaryTimeUUID is newly generated and is used to ensure request uniqueness, to ensure that each request actually appears in `CF:TIMELINE_QUEUE` and won't be lost because of interactions with another requests for the same row as a result of Cassandra weak consistency model.

Description of column families involved in timeline processing follows:

`CF:TIMELINE` – Timely monotonous sequence of modified rows for each table. Number of columns is mostly equal to number rows in particular table plus number of deleted rows.

`CF:TIMELINE_ENTRIES` – List of entries in `CF:TIMELINE` for each row. It is used to delete outdated entries and keep only the most actual one. `CF:TIMELINE` and `CF:TIMELINE_ENTRIES` and always modified atomically together as it is crucial that each entry is tracked in both CFs.

`CF:TIMELINE_QUEUE` – Records about row modifications for each table waiting to be inserted into `CF:CF:TIMELINE`.

`CF:TIMELINE_DELETES` – Similar to `CF:TIMELINE`, but contains only records about deleted rows. It is used to quickly iterate over records of deleted rows and garbage collect those which are possibly too old and useless.

Processing of requests from `CF:TIMELINE_QUEUE` is done in two phases: 1) Delete outdated entries both in timeline or queued. 2) Append up-to-date entry to timeline.

There is a sequence of operations to perform the first step:

1. Select all queued requests for the given row from `CF:TIMELINE_QUEUE`.
2. Find out which request matches winning row version according to Cassandra conflict resolver (highest timestamp and highest version). Other requests are immediately deleted.
3. Compare winning queued request with entries yet present in timeline (by reading `CF:TIMELINE_ENTRIES`) and delete outdated ones.
4. Possible outcomes:
   (a) All entries yet present in timeline were outdated by queued one. Winning queued entry remains in queue and execution continues to second phase.
   (b) Winning queued entry was outdated by entry already present in timeline. Queued entry is deleted and execution is finished.

In the second phase, following operations are executed.

1. Acquire table lock.
2. Generate new Time-UUID for use as a new HistoryTimeUUID to be appended to timeline.
3. Check that new HistoryTimeUUID is greater than the last one in `CF:TIMELINE`. If not then terminate execution and repeat whole process in near future. (This point is necessary because Time-UUIDs monotonously generated over several servers need not to have monotonous values.)

4. Write newly generated entry into timeline (this step involves atomic batch write to `CF:TIMELINE`, `CF:TIMELINE_ENTRIES` and possibly `CF:TIMELINE_DELETES`).
5. Check that lock acquired in step 1. has not failed. If it has there is a possibility of concurrent writes to timeline and violation of monotonousness. In that case execution is terminated and process is run once again.
6. Delete queued entry as it is now ensured that it was correctly appended to the timeline.
7. Release table lock.

6.2 Application interface

Application interface exposes two main paths:

- `/{db-id}/tables/{table-id}/rows/{row-id}`
- `/{db-id}/tables/{table-id}/history`

Both parameter {`db-id`} and {`row-id`} are UUIDs, to guarantee global uniqueness. Parameter {`table-id`} can be arbitrary value, preferably table name. The former path is used to manipulate and read rows. The later path is used to read history of a table and so synchronize changes from server side to client.

6.2.1 Row GET method

Optional query parameters:

timestamp	Does not return row with lower timestamp than this.
version	Required row version.

Possible error results:

unavailable	Row is not present in DB. It has been possibly deleted, or there is server-side eventual inconsistency.
try_again	Row satisfying timestamp and version requirements is not available yet. Possibly also caused by eventual inconsistency.
deleted	Row has been deleted.

Result data format:

```
{
  "modified": <timestamp>,
  "version": <version UUID>,
  "data": {
    <column name 1>: <value 1>,
    <column name 2>: <value 2>,
    ...
  }
}
```

6.2.2 Row PUT method

Optional query parameters:

rmw Check the timestamp of existing row and insert this one only if it has higher timestamp.

Input data format: Same as a format of result data in GET method.

6.2.3 Row DELETE method

Optional query parameters:

lastId Returns history entries newer than history entry with specified UUID. If unspecified then the result begins with the oldest history entry.

limit Limits number of returned history entries.

Possible error results:

history_gc If query cannot be satisfied because some delete history entries were garbage-collected after the lastId specified.

Result data format:

```
{
  "limit": <number of entries>,
  "history": [
```

```
        {
          "rowId": <UUID of changed row>,
          "rowTimestamp": <timestamp of modification>,
          "rowVersion": <row version UUID>,
          "historyId": <this history entry UUID>,
          "isDeleted": <whether row is deleted>
        },
        ...
    ]
}
```

Chapter 7

Client implementation

7.1 Architectural decision

Decision that synchronization is driven by history timeline mechanism (??) has direct consequences – it is necessary to catch up all modifications made on a local database and keep records of them. With these records it is then possible to synchronize local changes latterly.

Note: This is one reason for hash trees because they can be always rebuilt. But anyway, it would be impractical to rebuilt hash trees before each synchronization. And to update hash trees along with DB modifications, it would be also necessary to catch up all modifications. Besides that, updating hash tree has higher time complexity than proposed solution.

So in the first place it is necessary to make a decision where database modifications would be caught up.

7.1.1 SQLite modification

Modifying SQLite internals to provide seamless background synchronization would be nearly optimal solutions.

+ Developer is still provided with SQLite API.
+ Highly portable as SQLite itself is.
− Hard to implement and debug.

Due to the complexity this is not considered a relevant option. But there are another possibilities to extend SQLite without actually modifying its internals: hooks and so called virtual tables – vtab.

SQLite hooks

SQLite API provides functions to register several hooks: `sqlite3_update_hook()`, `sqlite3_commit_hook()`, `sqlite3_rollback_hook()`, `sqlite3_wal_hook()`. This option seems promising, but callbacks have several limitations and it is not clear whether it is possible to e.g. atomically update system metadata along with actual data modifications. But probably it would.

+ Original SQLite API.
+ Highly portable.
+ Easy to develop

– Possibility of hidden drawbacks.

SQLite virtual tables

Virtual table is a data source registered into SQLite which can be used like any other real table. They are managed by a custom module loaded in SQLite which provides given interface to operate data hidden behind these virtual tables. So the module has full control over the data (which could be almost anything, CSV, spreadsheet, another table, etc.).

In this case, developer would be given ordinal SQLite DB and custom module/plugin providing synchronized virtual tables.

+ Original SQLite API.
+ Highly portable.
+ More control over the data.

– Do not support creation of indices and triggers.

Drawbacks of SQLite on Android

So far proposed solutions have major drawbacks on Android platform. Android applications are written in Java, but SQLite is C library. Compilation of C++ code for Android is possible with NDK – Native Development Kit, but it has many limitations. One of them is that built-in SQLite is not exposed in NKD. SQLite can be accessed only through Android java SDK wrapper class `SQLiteDatabase` which however cannot be inherited

and modified as it has only private constructor. And SQLite ability to load native module by calling SQL function is disabled.

All these limitations probably have some reasons, but as a result it is impossible to extend functionality of build-in SQlite in any way. The only possibility is to distribute own SQLite binary together with own java wrapper and basically with substitution for Android database framework. But they would not be completely compatible anyway due to Android object model.

7.1.2 Content provider

Content providers are basic part of Android framework providing content to applications. This mechanism is used to access e.g. contacts, call logs, calendars, etc. (android.provider) In short, content provider is a class extending `ContentProvider` and implementing given CRUD operations to manipulate underlying data or whatever it is. Content provider has to be assigned an URI address and registered to Android system. Android has full control over the content provider and its life time then. Content provider is not accessed directly, but it is accessed through Android framework calls which are redirected to particular registered providers based on passed URI. It is similar to REST interface.

The main feature of content providers is that they are system-wide, i.e. it is possible to access particular content providers from other applications than they live in, and of course to specify access rights. If this is not a requirement then using `ContentProvider` is overkill in comparison with `SQLiteDatabase` .

+ Clean solution in a sense of Android framework.
− Overkill in comparison with `SQLiteDatabase` .

7.1.3 Custom solution

Modifying SQLite function with native module is technically the most purest solution with respect to SQLite. But in context of Android platform, it is the most complicated one. The only reason to choose it would be its easy portability.

Content provider is a pure solution with respect to Android framework, but it is too inconvenient and unnecessarily interconnected with Android system if only basic function of database is needed.

For these reasons custom solution was chosen – `SyncDB` . This class encapsulates `SQLiteDatabase` and provides basic CRUD functions with the same signature. So it is lightweight and easy to use as original `SQLiteDatabase` is.

7.2 Database schema

Database schema of a client-side SQLite DB being synchronized with the proposed cloud service is in the figure 7.1. All the custom user tables are represented with table `table_name` . For each user table there is a table `_table_name` used to store row metadata. And there is one `__sync_meta` table used to store the last synchronized timeline entry of each user table.

Description of columns follows:

`_id` Primary key which is required by Android framework in some situations. Not used by proposed synchronization client.

`_guid` Globally unique row identifier (UUID) used to identify row in proposed synchronization service.

`_table` Table name of a user table.

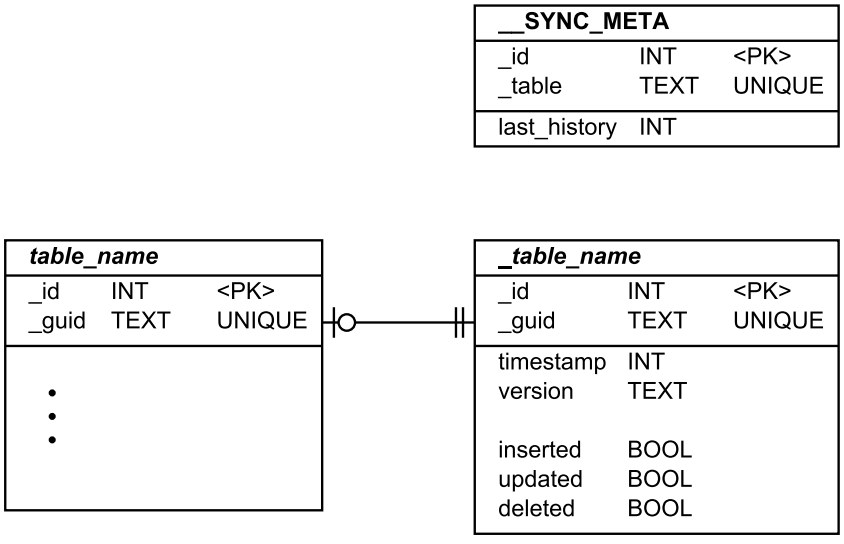

Figure 7.1: Database schema of a client-side SQLite DB.

`timestamp` Row timestamp for conflict resolution. It could be e.g. Lamport timestamp as discussed in ??.

`version` UUID associated with row version, i.e. particular column values.

`inserted` Flag that row has never been sent to the server and could not have been observed by other clients, i.e. row is still local only.

`updated` Flag that row has been modified in some way and modification has not been synchronized yet.

`deleted` Flag that row has been deleted.

`last_history` UUID of the last applied history entry.

7.2.1 Database operation

During each operation executed by database user, appropriate row metadata is updated transactionally. It always involves generating new `version` UUID and setting `updated` flag. In a case of a row insertion new `_guid` is generated and `inserted` flag is set additionally.

In a case of row deletions, both row and its metadata are deleted immediately if flag `inserted` is set. Otherwise row is deleted immediately, but metadata is kept and updated with flag `deleted` being set

7.2.2 Database synchronization

Database synchronization is low level decomposed into functions synchronizing one row at a time and doing either synchronization of local modification to cloud or vice versa.

Synchronization of local changes to server is done in 3 steps to minimize length of transactions because of communication with server.

1. Read row and metadata and possibly modify metadata.
2. Communicate with server.
3. Update metadata with respect to their possible change meanwhile.

Steps 1. and 3. are executed in transactions. But in step 2. database is unlocked and can be accessed by user. This has great effect on DB responsiveness at the cost of more complexity and more frequent DB access.

In the first step row metadata is modified only if `inserted` flag is set – then it is reset to signalize that row can be possibly shared with server and it is no more safe to delete it without server being notified.

In the second step appropriate server REST method is called, i.e. either PUT or DELETE. And if it succeeds then execution continues by step 3. Otherwise row synchronization fails and can be rerun.

In the third step it is checked whether `version` has been changed in comparison with the first step. If so then synchronization fails and can be rerun. Otherwise row synchronization succeeds and flag `updated` is reset, or row metadata is finally deleted if `deleted` flag was set.

Synchronization of each table is done separately. At first server is queried for a vector of table history entries. If value `last_history` for the given table exists in table `__sync_meta` , it is appended to the query so only newer entries are returned. Then entries are processed sequentially one at a time.

If affected row has local unsynchronized modifications (`updated` flag is set) then winner is resolved by the client according to the same rules as are applied on server. And either local change is synchronized to server or row version from server overwrites local modification.

As a history entry is processed, its id formed by a time-UUID is written to the `__sync_meta` table as a new `last_history` value.

7.3 Usage

Using proposed client is as simple as can be seen in the listing 7.1. It is preferred to extend provided `SyncDB` class and do initialization inside of a constructor. The only important part is calling of the super–class constructor and passed arguments:

1. Context where DB is created.
2. Resolver used to manipulate timestamps. Can be `LastWinsTimestamp` , `LamportTimestamp` or custom one.

3. Filename of the local database file.
4. Url of the server REST interface with appended DB identifier.
5. Object of type `DatabaseSchema` describing database schema.
6. Version of the database schema. Has the same meaning as in the case of `SQLiteDatabase` .

Listing 7.1: Example of creating synchronized DB on a client – sample Todo application in this case.

```
1  public class TodoSyncDB extends SyncDB {
2
3      private static final String DB_NAME = "todo.db";
4      private static final int VERSION = 17;
5
6      private static final String SYNC_REST_URI =
7          "http://192.168.212.105:8080/acs-server/rest/tododb/";
8
9      public TodoSyncDB(Context context) {
10         super(context, new LastWinsTimestamp(), DB_NAME,
11             Uri.parse(SYNC_REST_URI), createSchema(), VERSION);
12     }
13
14     static private DatabaseSchema createSchema() {
15         DatabaseSchema schema = new DatabaseSchema();
16         schema.addTable("todos")
17             .addColumn("date", Datatype.TYPE_DATE)
18             .addColumn("description", Datatype.TYPE_STRING);
19
20         return schema;
21     }
22 }
```

Chapter 8

Conclusion

During the work, problem was analysed and prototypes of a synchronization server, client library and sample application were successfully developed. Although it could be said that differences between columnar NoSQL databases and relational databases were not mitigated enough, i.e. transactions are missing, it is probably the only one feasible solution. As was explained in the text, NoSQL DBs dismiss most of guarantees RDBMS have in order to provide top performance and scalability. If it was possible to still provide these attributes but with transactions being preserved, it would be implemented yet. But it is not. So it does little sense to implement transactional system on the top of a distributed database, because it is even more complex problem and solution would be infeasible.

Another issue is a performance measurement. Development and testing was done on a single node virtual machine with very limited resources available. Cloud infrastructure was not available in the time. This is a very uncommon configuration for these types of databases. It is extremely unusual in a sense of hardware performance and number of database nodes. Because of that, no meaningful measurement could be done.

Bibliography

[1] *SQLite*.
URL: ⟨http://www.sqlite.org/⟩

[2] *Apache Cassandra^{TM} 1.2 Documentation*. 2013.
URL: ⟨http://www.datastax.com/doc-source/pdf/cassandra12.pdf⟩

[3] Abadi, A. T. D.: The problems with ACID, and how to fix them without going NoSQL. 2010.
URL: ⟨http://dbmsmusings.blogspot.cz/2010/08/problems-with-acid-and-how-to-fix-them.html⟩

[4] Browne, J.: Brewer's CAP Theorem. 2009.
URL: ⟨http://www.julianbrowne.com/article/viewer/brewers-cap-theorem⟩

[5] Bucur, T.: A comparison between several NoSQL databases with comments and notes. *Roedunet International Conference (RoEduNet)*, 2011.

[6] Coelho, F.: Remote Comparison of Database Tables. *Conference on Advances in Databases, Knowledge, and Data Applications*, 2011.

[7] Datastax: *Cassandra documentation*.
URL: ⟨http://www.datastax.com/docs⟩

[8] Ellis, J.: 2012 in review: Performance.

[9] Google Inc.: *Android reference*.
URL: ⟨http://developer.android.com/reference/packages.html⟩

[10] Hewitt, E.: *Cassandra: The Definitive Guide*. Sebastopol: O'Reilly Media, Inc., 2011, ISBN 978-1-449-39041-9.

[11] Owens, G. A. M.: *The Definitive Guide to SQLite*. USA: Apress, 2010, ISBN 978-1-4302-3226-1.

[12] Ravi, S.: A survey on data and transaction management in mobile databases. *Database Management Systems (IJDMS)*, 2012.

[13] Sandu, S. B. M.: MCSync – Distributed, Decentralized Database for Mobile Devices. 2012.

[14] Seetha, R. K.: A Stateful Database Synchronization Approach for Mobile Devices. *Soft Computing and Engineering (IJSCE)*, 2012.

[15] Strauch, C.: *NoSQL Databases*. Stuttgart: Hochschule der Medien.
URL: ⟨`http://www.christof-strauch.de/nosqldbs.pdf`⟩

[16] Vogel, L.: Android SQLite Database and ContentProvider - Tutorial. 2013.
URL: ⟨`http://www.vogella.com/articles/AndroidSQLite/article.html`⟩