CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF ELECTRICAL ENGINEERING

DIPLOMA THESIS

Distributed database for mobile devices

Diploma Thesis Supervisor: Ing. Tomáš Bařina

2014

Otakar Chasák

Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Computer Graphics and Interaction

DIPLOMA THESIS ASSIGNMENT

Student: **Bc. Otakar Chasák**

Study programme: Open Informatics
Specialisation: Software Engineering

Title of Diploma Thesis: **Distributed database for mobile devices**

Guidelines:

- Analyze concepts of NewSQL databases.
- Analyze RDBMS for Android mobile device platform and describe them.
- Analyze methods of accessing NewSQL from a mobile device. Propose a solution allowing access to data from a mobile device while minimizing network traffic.
- Install and document deployment of a chosen database.
- Design a method of data transfer between RDBMS on a mobile device and IaaS NewSQL database.
- Implement a prototype of the proposed method and demonstrate its functionality in a sample application.
- Document everything properly. Support important architectural decisions with arguments.
- Test the proposed solution and provide estimate of basic parameters.
- Perform integration and functional testing.

Bibliography/Sources:

Sedivy, Jan; Barina, Tomas; Morozan, Ion; Sandu, Andreea - MCSync - Distributed, Decentralized Database for Mobile Devices - Bengaluru 2012
Lars Vogel - Android SQLite and ContentProvider - US, 2012
Williams, D. - Optimal parameter selection for efficient memory integrity verification using Merkle hash trees - Ithaca, NY, USA - 2004
Eben Hewitt - Cassandra: The Definitive Guide - US, 2010
Grant Allen, Mike Owens - A The Definitive Guide to SQLite - US, 2006
Tudorica, B.G. - A comparison between several NoSQL databases with comments and notes - Romania, 2011
Hecht, R.; Jablonski, S. - NoSQL evaluation: A use case oriented survey - Germany, 2011

Diploma Thesis Supervisor: Ing. Tomáš Bařina

Valid until the end of the summer semester of academic year 2014/2015

L.S.

prof. Ing. Jiří Žára, CSc.
Head of Department

prof. Ing. Pavel Ripka, CSc.
Dean

Prague, February 20, 2014

Abstract

The goal of this diploma thesis is to explore new category of databases – NewSQL databases. Several NewSQL database are described, one of them is selected for next work. In this work selected NewSQL database is installed. For this installation a sample application is developed. The application uses selected NewSQL database as primary storage. The database is accessible from Android mobile device. Android application uses its embedded database to cache data from primary database to minimize network traffic and to allow to be used offline. Application synchronizes the cached data, when it is online. Possible solutions for this requirements are discussed and the selected is implemented in sample application.

Abstrakt

Cílem této diplomové práce je prozkoumat novou kategorii databází – NewSQL databáze. Vybrané NewSQL databáze jsou popsány, jedna z nich je zvolena pro další práci. Tato databáze je následně nainstalována. Pro tuto instalaci je vyvinut prototyp aplikace. Aplikace používá vybranou NewSQL databázi jako primární úložiště. Tato databáze je zpřístupněna pro mobilní zařízení s platformou Android. Android aplikace používá vestavěnou databázi jako cache pro data získaná z primární databáze, aby byl minimalizován síťový přenos a umožněna práce bez síťového připojení. Při dostupnosti síťového připojení dojde k synchronizaci dat. Možná řešení pro tyto požadavky jsou diskutována a vybrané řešení je implementováno v prototypu aplikace.

Prohlášení

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

V Praze dne 11. 5. 2014

…….

Acknowledgement

I would like to thank my supervisor Ing. Tomáš Bařina for valuable advices and help he has provided.

Content

1. Introduction

Nowadays needs to store and access huge amount of information rise. Solutions based on standard technologies are not sufficient or are very expensive. There are some problems related to this activity and there are new types of database, which tries to solve these problems. NewSQL databases are one of the directions. These quite new products try to solve problems related to storing big data, while preserving ACID properties.

The goal of this thesis is it to explore NewSQL databases and develop a sample application, which uses NewSQL database. This is the first part of the application, the second one is client application running on Android mobile device. The client application uses embedded SQLite relational database as a cache. While the network connectivity is not always present, user can access and modify cached data when is offline. These changes are synchronized to server, when the network connectivity is available.

In the first chapter several concepts of NewSQL databases are explained and one is selected for the sample application. Next chapter describes the SQLite database, which is embedded in Android devices and used in client application.

Next chapters describe method of accessing selected NewSQL database from mobile device and problems, which can happen when replicating server and client database.

Next chapter describes features of developed solution and documents the application, its deployment and development for selected NewSQL database.

The last chapter describes tests, which are part of the developed application.

2. NewSQL

The network traffic is still growing within last few years. More and more data must be stored and be available online. It means higher load for OLTP (online transaction processing) databases. The OLTP has following characteristics[9]:

- transactions are short-lived
- transactions touch small amount of data - no full scans or large distributed joins
- transactions are repetitive - executing same queries with different inputs

NoSQL databases, which became popular during last years, takes this into consideration. They are faster than original RDBMS and can be easily scaled. Many of them are key-value and support low level query language. But they lack full ACID transaction support. It is not problem for some types of application, but for some other application is it necessarily. This leads to creation of new databases class called NewSQL. NewSQL databases provide similar performance as NoSQL databases, are easily scalable, use SQL, but keeps full ACID transaction support. NewSQL databases often use shared-nothing architecture, it means each node is independent and none of the nodes share memory or disk storage. Categorization[1] of NewSQL databases is based on different approaches, used by vendors.

- **new architectures** - these databases written from scratch, running in cluster of shared-nothing nodes, where each node has its own subset of data. These are VoltDB, H-Store, NuoDB, Google Spanner or Clustrix
- **new MySQL engines** - new storage engines provides better scalability and original MySQL interface. New MySQL engines use TokuDB, MemSQL or TokuTek
- **transparent sharding** - pluggable layer, which enables split database across multiple nodes, but database remains same. The examples are MySQL Cluster or dbShards

NewSQL databases

ClustrixDB

ClustrixDB is a scale-out database designed for cloud, which was developed in Clustrix, Inc. Clustrix served customers since 2008, today is available as software in Amazon Web Services or can be installed on cloud. Designed for cloud, Clustrix supports real-time analytics.

ClustrixDB runs on cluster of shared-nothing nodes, it uses per-index distribution strategy to split data across the nodes. Each table contains one or more indexes referred as representation of the table. Each representation has its own distribution key, which is used to slice the data in the table. Clustrix breaks each representation into a collection of logical

slices using consistent hashing. To ensure fault tolerance and availability. Clustrix has always at least two physical replicas of each slice, which are stored on separate nodes. It enables to specify number of replicas for each representation. If some of the slice exceeds a maximum size, it is re-sliced. Thanks to consistent-hashing, only this slice needs to be rehashed.

When a query is executed, Clustrix first looks up, on which node requested data resides. Then it reads the data from node and returns them. When join is used, each returned row can be processed on more nodes. When more rows are returned, process can run parallelly on different nodes.

In production environment ClustrixDB requires SSD disks as data storage. Clients can use MySQL drivers for interacting with database. Multi version concurrency control (MVCC)[8] is used to ensure lockless readers. To achieve consistency MVCC uses timestamp or transaction id to determine state the database. ClustrixDB enables setting of isolation level - read committed, repeatable read and serializable read. There are supported standard data types. ClustrixDB can perform online schema changes without blocking reads or writes. It creates a new container into which it copies the data. All DML commands are applied to data in container and logged in temporary log. These are then applied to changed data and the container is dropped. [3]

NuoDB

Development of this database started in 2008 in company called NimbusDB. In 2011 it changed its name to NuoDB. Architecture of NuoDB consists of three layers - administrative tier, transactional tier and storage tier. Transactional tier maintains atomicity, consistency and isolation. It is in-memory tier, that acts as on-demand cache. The storage layer ensures durability. It makes data durable on commit and provides data access, when there is a cache miss. These tiers consist of processes running over hosts. All processes are peers - no single point of failure. NuoDB defines these layers by running single executable in one of two modes - Transaction engine or Storage manager. These modes communicate with each other over peer-to-peer coordination protocol. Transaction engines accept client SQL calls, parse them and run SQL against cached data. When there is a cache miss, it can gets data from any of its peers - from different transaction engine, which has data in cache, or from storage manager, which has data stored. The minimum NuoDB configuration consists of one transaction engine and one storage manager running on one host. New transaction engines or storage managers can be started anytime.

Client can be connected to database using one of the driver developed by NuoDB (Hibernate, JDBC, ODBC, .NET, PHP PDO) or community driver (Node.js, PHP Zend etc.). Under the front-end layer of the transactional tier, which supports SQL, the data are stored and managed in objects called Atoms. The front end layer maps SQL to associated Atoms. When transaction engine starts first, it populates only one object in cache - the root catalog Atom (Master catalog). From this Atom all other elements can be discovered (on which node a needed element is cached or if it must be obtained from storage manager). To provide consistency and prevent conflicts NuoDB uses Multi-version concurrency control. All data are versioned and each update or delete makes new version. It can be set isolation

level for transaction (consistent read, write committed, read committed). NuoDB by default runs with a Snapshot Isolation model. In this mode one transaction can read the data, which another transaction updates and there is no conflict. It uses MVCC as ClustrixDB. When two transactions update or delete same data, all changes are made on transaction engine, where the data are cached. This transaction engine (called Chairman) can detect conflicts. When any change is made in transaction, reliable message is sent to all peers, that need to know about it (all storage managers and transaction engines with cached atom). NuoDB supports standard data types. There could be performed online schema changes.[6]

VoltDB, H-Store

VoltDB is in-memory NewSQL database, it is an implementation of academic OLTP database called H-Store. This application runs on a cluster of shared-nothing nodes. H-Store instance is a cluster of one or more nodes. Node is one physical computer, where run one or more execution sites and one transaction coordinator. Execution site (logical operational entity) is a single-threaded execution engine that manages some part of the database and executes transaction on behalf of its transaction coordinator. The transaction coordinator ensures serializability of transactions. All transactions has serializable isolation level.

Tables in H-Store are horizontally divided into fragments (shards), this is done by a partition attribute (column). Related fragments from multiple tables are combined together into a partition. Hash value of the partition attribute determines fragment, where the tuples are placed. Each table can have more partition attributes.

The traditional relational database management systems use disk as primary storage for data. There is often some kind of main memory cache in front of the database installation. The objects may reside both in cache and on disk. It requires some application logic to keep these objects synchronized and also disk access is needed for it. Use of caches can also be problem for scalability. The H-Store architecture reverses the traditional storage hierarchy, because main memory is used as primary storage device. If the database does not fit into main memory, cold data are swapped to hard disk. This technique is called anti-caching.
To ensure availability in case of node failure data are replicated. H-Store creates snapshots (disk based copies) of the database to provide durability. It is possible to use command logging (synchronous or asynchronous). After the restart database is restored from the last snapshot and logged commands are replayed.

Client interacts with the database via stored procedures, which are written in Java. Each procedure is identified by its unique name. Stored procedures are assigned by a partitioning attribute consisting of one or more input parameters. Each instance of a stored procedure made as a response to client request is called transaction. When a client request is made, transaction coordinator hashes value of the partitioning attribute and redirects it to site with the same id as hash.

VoltDB creates transaction model (Markov model) for each stored procedure. These models and stored procedure input parameters enable database to predict the actions of transaction

at runtime. If the database know what the transaction will do, it can select optimization for each transaction. Following optimizations are possible[9]:

- execute the transaction at the node with partition, which is most accessed
- lock the accessed partitions only
- disable undo logging for non-aborting transactions
- speculatively commit the transaction at partitions, which will not be accessed

VoltDB support changing schema or stored procedure while the database is running.[9][10][11][12]

MySQL cluster

MySQL cluster is in-memory database running in shared-nothing cluster. It combines standard MySQL server with clustered storage engine - NDB (Network DataBase). Running configuration contains computers (host), each running one or more processes (node). Nodes can be of following types: MySQL servers - for accessing NDB, data nodes for data storage and management node. When NDB is used as a storage engine, data are stored at data nodes and accessible via all MySQL servers in cluster. There could be all data types supported by MySQL database. MySQL cluster has only read committed isolation level.

Data node is a process, which stores replicas. Each data node should be located on separate computer and is member of node group. Node group consists of one or more nodes and stores partitions or sets of replicas. Partition is a portion of data stored by the cluster. MySQL cluster partitions table automatically, but it is possible to use user-defined partitioning. Replica is a copy of a cluster partition. As long as each node group has at least one operable node, the cluster has all data available.
MySQL cluster can be used with existing MySQL applications. These applications communicate with MySQL server node. Applications can be written directly for MySQL cluster. There can be used NDB API (C++) or MySQL cluster connector for Java (high level API similar to ORM frameworks), these cases connect directly to NDB. There can be performed online schema changes.[5]

Database selection

I have selected H-Store for application, which I will implement in my diploma thesis. H-Store differs from other compared databases in technology used for storing data. It keeps all the data in main memory, other compared databases use main memory as cache. Access to the data should be faster, because it has no disk stalls. It runs on shared-nothing cluster as most of described databases. This architecture is convenient for scaling.

H-Store provides features, that other databases do not offer:

- It optimizes data location to keep accessed data together on one node and reduce network traffic.
- It can select optimization for each transaction in runtime, because parameters of stored procedures and model of the transaction.

H-Store is research database, it is reason, why the features are described better than in other compared databases, where it is not explained how they achieve better performance. It is open-source.

NewSQL database comparison

	ClustrixDB	**NuoDB**	**VoltDB/H-Store**	**MySQL cluster**
outside interface	MySQL drivers	Hibernate, JDBC, ODBC, .NET, PHP PDO and other community drivers	Java, C++, C#, PHP, Python, Node.js and community libraries	MySQL drivers
data storage	SSD	HDD	main memory/HDD	
data types	standard data types	standard data types	tinyint, smallint, integer, bigint, float, decimal, varchar, varbinary, timestamp	standard data types
isolation levels	read committed, repeatable read, serializable	consistent read, write committed, read committed, repeatable read, serializable	serializable	read committed
online schema changes	yes	yes	yes	yes
source code	no	no	yes	yes
programming language	C		Java, C++	C++

3. RDBMS for Android device platform

There is only one RDBMS officially supported in Android - SQLite. It is widely used as embedded database for local storage in web browsers (Google Chrome, Firefox, Opera) and other programs (Skype).

SQLite

SQLite is distributed as a small single library without any dependencies. It implements most of the SQL92 standard.

- RIGHT OUTER JOIN and FULL OUTER JOIN are not supported.
- Views are read only.
- ALTER TABLE is not fully supported.
- FOR EACH STATEMENT triggers are not supported.
- GRANT and REVOKE are not supported, because they are meaningless in embedded database

SQLite does not have a separate server process. Instead the SQLite library can be linked in or called dynamically by the application. The entire database is stored in a single cross-platform file. SQLite implements serializable transactions, which keeps ACID properties. SQLite uses read and write lock, locking is done over the entire database file. It enables concurrent reading, but only one writing at the time. Even when modifying different table.[16][17]

In other databases the datatype is determined by the column in the database table (static typing). SQLite uses dynamic type system, the datatype of a value is associated with the value, not with the container. In SQLite value each datatype is stored as one of the following types[16]:

- NULL
- INTEGER
- REAL
- TEXT
- BLOB.

Android API for SQLite interacting

Android platform has embedded SQLite database. There are also classes, which provide methods to access this database. By default the database is located in the directory: DATA/data/APP_NAME/databases/FILENAME. Where APP_NAME is the name of the application and FILENAME is name of the database.

Classes for working with SQLite is placed in package android.database.sqlite.*

SQLiteOpenHelper takes care of database creation and version management. It provides methods getReadableDatabase() and getWritableDatabase() to access SQLite in read or write mode.
SQLiteDatabase class has methods for inserting, updating querying and deleting. It also provides method for executing raw SQL.
Cursor interface is returned by the query. Implementation of this interface represents result of the query and can be used to retrieve values of columns and iterating through the resultset.[18][19]

When working with the SQLite database, the file system is accessed. Therefore it is recommended to perform database operation asynchronously.

SQLite and H-Store comparison

Both of these databases are row-storages relational databases. H-Store is NewSQL database designed to run in cluster of shared nothing. This concept provides high throughput and high availability. SQLite database is a single file - when writing the while file is locked. It does not provide high throughput compare to server databases. The H-Store uses main memory as primary medium, SQLite is stored on filesystem. Both databases supports serializable isolation level only.

4. Methods of accessing H-Store from mobile device

The H-Store communicates by its stored procedures. These procedures can be called by Java API provided in H-Store. To enable communication from mobile devices, these stored procedures have to be accessible via a web service. This enables to develop client application without dependency on specific H-Store API and Java programming language. There are two possibilities when developing web services.

- SOAP WS
- REST WS

SOAP WS are oriented on operations. This is mostly used in enterprise solutions. Libraries implementing this standard are needed on client side, which is problematic for development on Android and other mobile platforms.

REST (Representational state transfer) is an architectural style for distributed hypermedia. It defines constraints to build stateless services. REST uses standardized protocols. When HTTP protocol is used, the REST service can use its support for caching by using standardized HTTP header to minimize network traffic. Implementation of calling REST service on mobile device is easy and no new libraries are needed.

The REST WS was selected, because it is a standard for mobile platforms, no dependencies are needed. The functionality of HTTP headers for caching can be used.

5. Data transfer

Data contained in the application can be divided into two groups.

User specific data

Each user has its own data, which he accesses in application. When reading this data, the application have to check that local cache contains all the records and all of them are up to date, otherwise data must be downloaded or updated. The user can access data from more client devices (e.g. mobile and web browser), so the changes in data must be detected and applied. In case of mobile device we cannot rely on the connection to server (Internet is switched off).
The application should support postpone commit. In case of postpone commit the user can theoretically perform changes from different device with access to the Internet, so a conflict can occur when the connection to the server is present.

Shared data

The application can contain data, which is used by more or all of the users. When reading the data the application have to detect whether the data is complete and up to date. If not the changed or missing data must be downloaded.
If the users are allowed to change this data, there is probability of conflicts, when users are modifying same record. If the postpone commit is enabled following scenario could happen. One device is offline and some record is modified. After that second device modifies same record and commits this change to the server database. Then the first device becomes online and tries to create postpone commit. If this commit is successful the update made from second device could be missed, however it was the last update.
If this data was often modified or new records were added, it could constrain other users to update their cached data frequently.

Transmission of larger files

Downloading or uploading larger files over the Internet on mobile device can be affected by speed of mobile network, or network failure. The connection can fail while uploading or downloading the data (eg. weak signal) and so the transmission is not completed. Application has to detect this situation and recover from it.

Synchronization

H-Store database used on server and SQLite database on mobile client are both relational databases. These databases will use same or very similar database scheme. Synchronization between these databases can be made on different levels.

The first possibility is to make the changes in SQLite database and modified rows mark as changed and when the postpone commit is present send only the changed rows to the server. If same row would be modified in offline mode in client device and on server before the postpone commit will begin, the client modification should be reverted. But there are some use cases, which can be applied on the record based on its present value, for example rating the record. If the conflicts would not be checked, some modifications could be lost.
In the second approach the client can change the rows in the database and log the operation, which is made, when the device is offline. When the device connects to server, the actions, which are in this log will be made on server. Potential conflicts will be resolved on the server. This approach enables to resolve potential conflict based on the action doing the change. If the change will be reverted, related record changes made in the local database will be also reverted.

When the postpone commit is present, the following situation can occur. Some transactions, which were made offline, can be rollbacked, when the postpone commit is done. This rollbacked transaction can affect the transactions, which were made offline after the rollbacked one (cascade of transactions). There can be transactions, which can be committed independently of result of previous transactions and transactions which must be rollbacked, if the previous transaction failed.

Summary

Many problems can occur in specific user data and in shared data. It is likely that these problems will arise in shared data.

- data completeness detection, deletion detection
- data integrity detection
- conflict detection
- postpone commit
- cascade of transactions
- network failure

Solutions

A conflict can occur under above mentioned conditions. It means the client tries to modify row. The value of that row is different from the value, which was changed on the client device. There are more approaches that solve this problem.

The vector clock is common mechanism used for replica conflict resolution. It is an algorithm to generate a partial ordering of events. It is a vector of N logical clocks, where N is a number of processes in system. Vector clock enables to determine relationship between the versions. It is not suitable for environment with many processes, because the vector will be too long, as in sample application, where it is supposed to be many users.[22][23][24][25]

Simpler solution is to use optimistic locking. This technique requires to add column to database table, where the version or timestamp is kept. Whenever the record is updated, the value of this column is checked. If it is equal to the original the change is committed, otherwise the transaction is rollbacked. This technique enables to detect conflicts in postpone commit, but does not enable to determine relationship between versions.

For data integrity test hash function can be used. The checksum is calculated from the record. This checksum is sent over network. The received checksum is then compared with the checksum calculated from the local record. Sending only checksum over the network reduces the volume of the transferred data.

Merkle tree can be used for data completeness detection. Merkle tree also called hash tree was patented by Ralph Merkle in 1979. It is a tree of hashes, where the leaves are hashes of data blocks. The inner nodes are hashes of their children. There is a root hash on the top of the tree. This hash can be used to verify consistency between huge datasets. It reduces amount of the transferred data. It is possible to discover, which records are changed or missing, by comparing hashes contained in the nodes of the Merkle tree. When the hash tree is built, it can be updated, when the data are changed. It is common to keep the Merkle tree in memory, which can lead to large memory consumption. For mobile client devices it is not optional to build Merkle tree every time the application is started. There can be used different approach, where the leaf nodes are stored in database and the tree is fixed-size. Only the non leaf nodes are stored in memory, which would save memory and speed up the tree reconstruction. [21][22]

When using REST API and HTTP protocol, its caching support can be used. This can be done by using following HTTP headers:

- Last-Modified - response header, contains time of last modification
- If-Modified-Since - request header, client has already data and verifies its validity
- Etag - response header, contains tag (hash) of requested data
- If-None-Match - request header, contains tag of locally cached data

For purposes of sample application, the header If-None-Match is rewarding. The client application can send hash of data retrieved from its local database and server can compare received hash with hash of its own data.

Creating new record offline

When mobile client is offline and new record is created, SQLite database creates a primary key for this record. Here could be a conflict between primary keys created on client device and primary keys created in server database. For example the SQLite database could generate primary key, which is yet used on server. Because the SQLite database is used as a

cache of the part of the server database, the primary keys created in offline mode must be synchronized with the server.

Records created offline could be stored in the same table in SQLite as records cached from the server. In this case the primary key would be changed during synchronization. This could lead to cascade update and a new primary key obtained from the server could be in conflict with primary key of another newly created record.

Different approach is to keep newly created records in a different table, temporary one for example. During the postpone commit these records will be send to server, where they will be saved and their id obtained from the server will be used to store them in SQLite.

It is also possible to use universally unique identifiers (UUID)[26]. Its intent is to uniquely identify information in distributed systems without any significant central coordination. Java and Android provide API to generate UUID. It is a 128 bit number, represented by 32 lowercase hexadecimal digits, which are displayed in five groups separated by hyphens. The first 3 sequences are interpreted as hexadecimal digits, the remaining 2 sequences as plain sequence of bytes.
Use of UUID as a primary key makes creating new records in offline mode easier. The UUID can be generated in offline mode and when the connection is available the record is send to server. Use of UUID increases size of the database and can slow down database operations, when used as a primary key.
In SQLite a primary key can be of any type, but when using some part of Android API to access database (android.database.Cursor) column with primary key must have name "_id". Here it seems possible to create own implementation, but there can be changes in future versions and support will become complicated. It is easier approach is to use integer as primary key in SQLite database (it is faster and all parts of API can be used) and to use UUID only when communicating with the server. There can be database table for looking local id of UUID and vice versa.
In H-Store types of primary key column are not restricted, but partitioned columns can be only of type integer or string. If UUID would be used as primary key it must be of type string, because primary keys will be partitioned.[34]

For sample application UUID is used as primary key, because it simplifies creation of records in offline mode - the new record can be sent to server with the same primary key.

Postpone commit

If the cascade of transactions should be solved, there must be log of transactions, which were made offline. It must contain order of transactions and affected rows. When the postpone commit is being made and any of the transactions is rollbacked, the following transaction can be rollbacked too, or rules could be defined. These rules can define, which of the following transactions will not be synchronized to the server. Rows affected in client database by the rollbacked transactions will be updated from the server.

When a mobile client is offline, the application can enable a user to perform some operation in the application. Changes are made in the SQLite database and performed operations are logged in separate table. This approach enables to solve the dependencies between transactions made offline (cascade of transactions), when synchronizing with server.

If the connection is available, the logged operations are sent to the server one by one ordered by time. If the operation succeeds, next logged operation is sent. If the operation failed due to conflict or database constraint, following operations are cancelled. Application must ensure updating of records, which were affected by cancelled operations.

6. Installation of H-Store database

For building and running H-Store database one of the following platform is needed[20][32][33]:

- Ubuntu Linux 9.10+ (64-bit)
- Red Hat Enterprise Linux 5.5 (64-bit)
- Mac OS X 10.6+ (64-bit)

The following packages are required to be installed local machine before building:

- gcc/g++ +4.3
- JDK +1.6 - OpenJDK or Oracle distribution, 1.7 recommended
- Python +2.7
- Ant +1.7
- Valgrind +3.5

Before running H-Store the OpenSSH have to be enabled and login to localhost without a password must be allowed.

H-Store source code can be downloaded from GitHub:

git clone git://github.com/apavlo/h-store.git

The entire H-Store project is build by executing the build target:

ant build

H-Store project comes with set of predefined benchmarks. Before the benchmark can be started, a jar file containing compiled stored procedures, system catalog and cluster configuration of selected benchmark muse be build. This is done by the following command:

ant hstore-prepare -Dproject=project_name

The resultant jar file is created in H-Stores base directory with the name of project. By default the command hstore-prepare cluster configuration of only one host with two sites, each with two partitions. This can be changed via configuration file or by specifying cluster configuration directly in command line.

To execute benchmark the following command is used:

ant hstore-benchmark -Dproject=project_name

It starts up the database, loads the data and executes defined stored procedures, then the database stops. Additional options can be used, to remain running (parameter noshutdown), to prevent it from executing whole benchmark (parameter noexecute).

ant hstore-benchmark -Dproject=project_name -Dnoexecute=true -Dnoshutdown=true

If the database is running the stored procedures can be invoked by following command:

ant hstore-invoke -Dproject=project_name -Dproc=procedure_name -Dparam0=param_value

7. Sample application

One goal of this thesis is to develop a sample application to evaluate the designed solution. The application is named MusicCatalog and enables users to search, add and mark songs. A freely available collection of song metadata is used to populate the database. This application should have following attributes:

- H-Store as server database
- Multiple user access
- Client application written for Android
- Client application caches the data to enable offline usage
- Each user can use application on multiple devices

Use cases

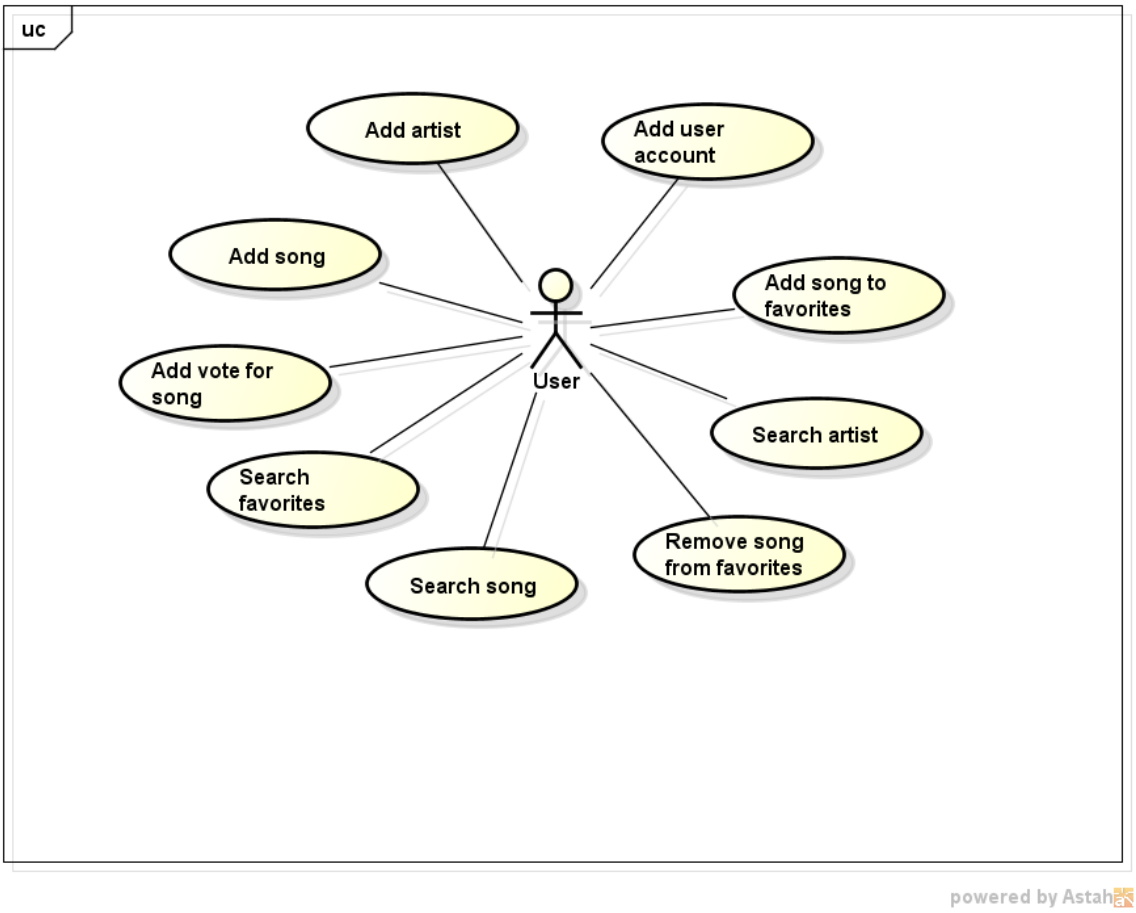

Figure 1 - Use case diagram

Add user account

1. System shows form
2. User fills in nick and password
3. System save creates new user

Add artist to dataset

1. User selects ‘add artist’
2. System shows a form for artist metadata
3. User fills in the form
4. System saves new artist

Add song to dataset

1. User selects ‘add song’
2. System shows a form for song metadata
3. User fills in the form
4. System saves new song

Add vote for song

1. User selects ‘add vote’
2. System increments current count of votes

Search artist

1. User selects ‘search artist’
2. System shows a screen for song searching
3. User fills in informations about artist
4. System shows search result

Search song

1. User selects ‘search song’
2. System shows a screen for song searching
3. User fills in informations about song
4. System shows search result

Search favorite songs

1. User selects ‘favorite songs’
2. System shows user’s favorite songs

Add song to favorites

1. User selects ‘add to favorites’

2. System saves current song to user's favorites

Remove song from favorites

1. User selects 'remove from favorites'
2. System removes current song from user's favorites

Architecture

Developed application consists of server part and client part running on Android device. The server part contains two projects. First one is the project named MusicCatalog_Database. It contains implemented stored procedures and configuration files for H-Store database. It is built together with H-Store database. Second part is Java project named MusicCatalog_Server. There is REST API implementation for serving request from users. REST API communicates with H-Store database via stored procedures.

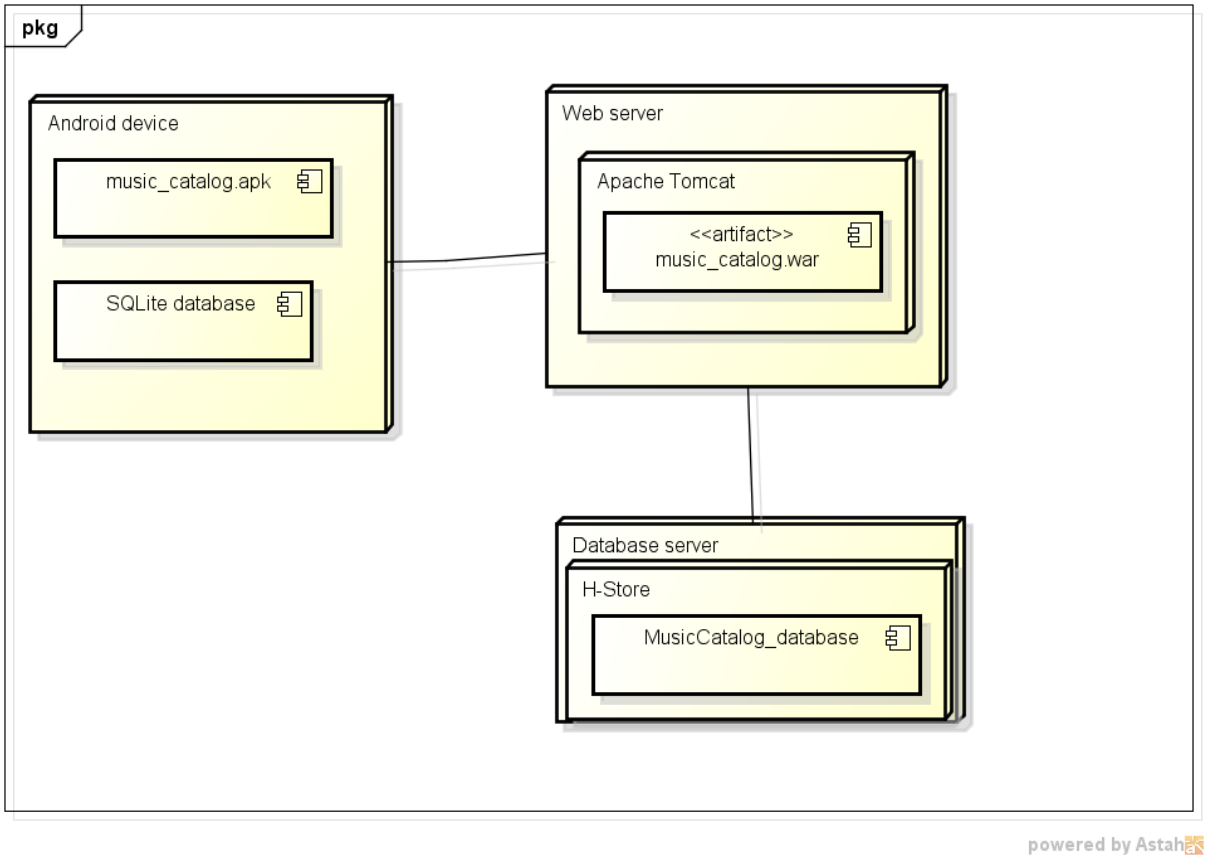

Figure 2 - Deployment diagram

Server part

MusicCatalog_Database

MusicCatalog_Database is a Java project containing stored procedures for communicating with H-Store database, database schema and Java classes for H-Store configuration.

Package cz.cvut.fel.music_catalog contains following configuration files:

- music_catalog-ddl.sql - SQL script with SQL DDL for schema creation. The name of the file must begins with project name and continue with suffix -ddl.sql.
- ProjectBuilder.java - Java class, which extends AbstractProjectBuilder class from H-Store API. ProjectBuilder defines project name, stored procedures, DataLoader class and ClientDriver class.
- MusicCatalogLoader.java - Java class, which extends Loader class. It used for populating the database after startup
- MusicCatalogClient.java - Java class for automatic benchmark

Database schema

Database schema contains following tables:

- Artist - stores the names of artist
- Songs - contains metadata related to each song, column votes stores count of votes given to each song
- User - contains user's accounts, nick is unique user name used for signing into the application, database contains hashed password, which was used when creating user account
- favorite - it is intersection table, which realizes binding between user and his favorites songs

Primary keys used in this database are generated by UUID Java class.

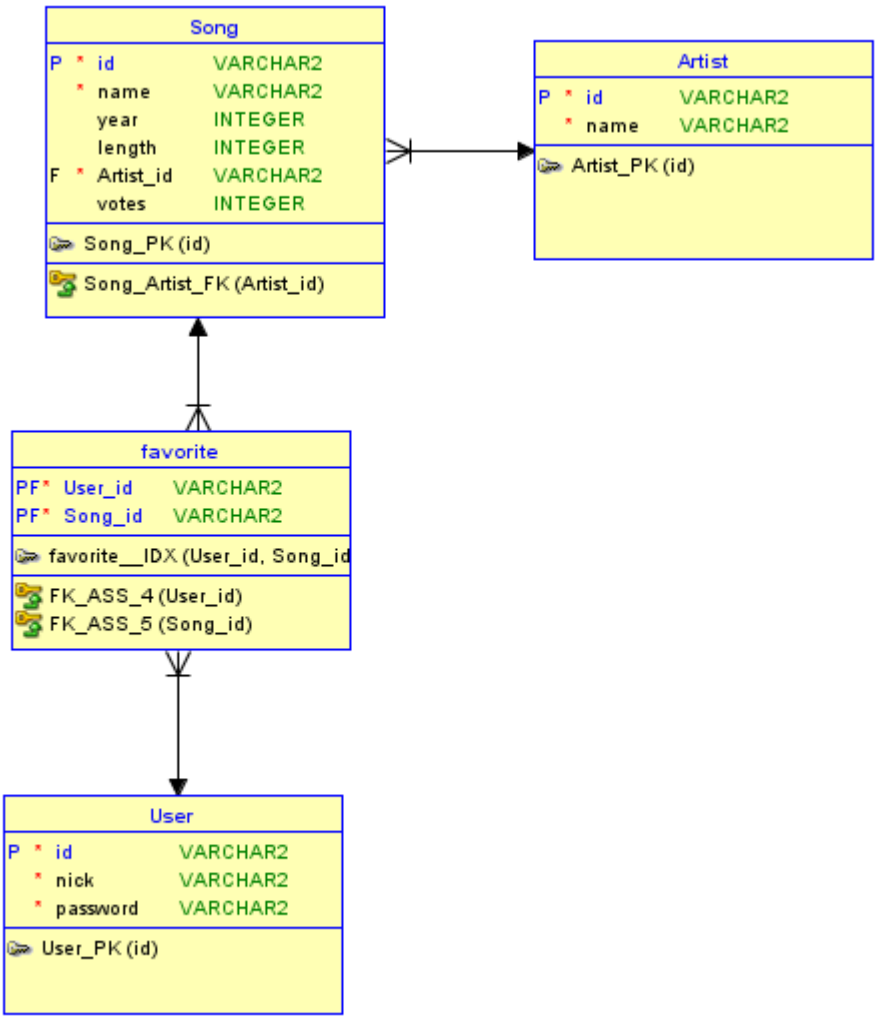

Figure 3 - H-Store database schema

For better performance of H-Store database an estimation of volume and frequency of stored procedures must be known, so the partitioning can be made. The goal of partitioning is to ensure, that the most frequently stored procedures are single-sited. This is mainly important for procedures modifying data.[34]

There are also following constraints for partitioning:

- Only columns of string or integer types can be partitioned
- There can be only one partition column per table
- Partition column do not need to be unique, but they can not be null

In this application HTTP BASIC authentication is used, so method for finding user by nick is called in each incoming request.

User specific data

Server database contains all the user specific data. H-Store is row-store base relational database, it means all the data are stored in database tables. The database must contain table with user accounts, other informations belonging to each user are binded by foreign keys in other tables.
Users are authenticated in REST service, so each request is connected to specific user and can modify only his data.

Data

Data used in this application are obtained from Million Song Dataset. It is a freely-available collection of audio features and metadata for a million contemporary popular music tracks. The whole dataset consists of HDF5 files, each file for one song. HDF5 is a file format developed by NASA to handle large datasets. The data can also be retrieved from SQLite database distributed with HDF5 files.
There are over 40 metadata for each song including artist and track title, year or duration. I have selected only a subset of this 40 metadata for my application.

Stored procedures

Communication with H-Store database is made through stored procedures. Each stored procedure is a Java class, which extends VoltProcedure class from H-Store API. In H-Store the stored procedure is one transaction - it succeeds, or rolls back whole[4]. The stored procedures are called from REST API. There is one stored procedure for each operation provided by the application.

All the stored procedures created in this project are in package cz.cvut.fel.music_catalog.procedures. These are:

- AddArtist
- AddSong
- AddToFavorites
- AddUser
- AddVoteToSong
- GetArtistById
- GetArtistByName
- GetFavorites
- GetSongById
- GetSongByName
- GetSongByNameArtistId
- GetSongsByArtist
- GetUser
- RemoveFromFavorites

Example of H-Store stored procedure

By design H-Store uses stored procedures to execute queries and operation over the database. It is also possible to use ad hoc queries, but they are not optimalized, so they should not be used to repetitive transactions. In H-Store the stored procedure and transaction are the same, it succeeds whole or rollbacks. So there is no need to use transactions.

Stored procedure is separate Java class, which extends VoltProcedure API. It must contain predefined SQL statements and run method, which is called, when the stored procedure is started. The run method can return long or VoltTable datatypes. Input parameters of the run method could be following types or their arrays[34]:

- Integer types: byte, short, int long, Byte, Short, Integer, Long
- Floating point types: float, double, Float, Double
- Fixed decimal point: BigDecimal
- Timestamp types: org.voltdb.types.TimestampType, java.util.Data, java.sql.Date, java.sql.Timestamp
- String and binary types: String, byte[]
- VoltDB types: VoltTable
-

```
public class GetArtistById extends VoltProcedure {

    public final SQLStmt getArtistStmt = new SQLStmt("SELECT * FROM "
    + DbNames.ARTIST_TABLE + " WHERE " + DbNames.ID_COLUMN + "=?;");

    /**
     * Searches artist by id
     * @param artistId
     * @return
     */
    public VoltTable run(String artistId){
        this.voltQueueSQL(getArtistStmt, artistId);
        VoltTable[] queryresults = this.voltExecuteSQL();
        VoltTable result = queryresults[0];
        return result;
    }

}
```

Figure 4 - Stored procedure example

In stored procedure GetArtistById is defined SQL statement for selecting record from Artist table by its primary. The run method takes one parameter, the id of artist. The returning parameter is of type VoltTable, it means the procedure returns result from one table. There is a possibility to return results of more queries. This can be done by calling method voltQueueSQL multiple times and return the result of method voltExecuteSQL. The run method would then return array of VoltTable.

The run method can also contain logic in Java programming language, processing of result of query and then next query and so on.

When the Java runtime exception occurs in run method, or database constraints are violated, the transaction is automatically rolled back. In situations, when there is need to rollback the transaction manually, VoltAbortException can be thrown, which causes transaction to rollback.[34]

Deployment

The project containing stored procedures and H-Store configuration is built by Ant build.xml located in H-Store's base directory. To build and run the project following steps must be done[4]:

- copy project's source including package hierarchy into src/benchmarks directory in H-Store's base directory
- create benchmark specification file, it is located in properties/benchmarks directory in H-Store's base directory, the file must have same name as benchmark (music_catalog.properties) and the canonical path to the ProjectBuilder call must be specified as follows:

  builder = cz.cvut.fel.music_catalog.ProjectBuilder

- build the entire H-Store distribution by command ant build
- create a project's jar by command:

  ant hstore-prepare -Dproject=benchmark_name

- start the benchmark with parameters to remain running:

ant hstore-benchmark -Dproject=benchmark_name -Dnoexecute=true -Dnoshutdown=true

MusicCatalog_Server

The project MusicCatalog_Server is written in Java programming language and uses Jersey library as implementation of JAX-RS specification. Data are sent in JSON format.

REST API is used for serving client's requests. I have selected REST architecture style, because it is easily scalable and can be accessed from mobile devices or web applications.

resources:

- context/artist/
  - POST - adds new artist, if artist with specified name already exists, HTTP status SEE OTHER (303) is returned

- context/artist/{id}/
  - GET - returns concrete artist identified by the id, if specified id does not exists, HTTP status NOT FOUND (404) is returned

- context/artist?artistName={name}
  - GET - returns collection of artists with name equals to query parameter name

- context/artist/{id}/songs
  - GET - returns all songs from artist identified by the id, if specified id does not exists, HTTP status NOT FOUND (404) is returned

- context/song/
  - POST - adds new song

- context/song?songName={name}
  - GET - returns collection of songs with name equals to query parameter name

- context/song/{id}
  - POST - adds vote for song specified by id, if specified id does not exists, HTTP status NOT FOUND (404) is returned

- context/user/
  - POST - adds new user into system, authentication is not required

- context/favorites
  - GET - returns collection of favorites songs for user
  - POST - adds song to user's favorites

Transferred data are serialized and deserialized to JavaScript Object Notation (JSON) by Jackson library. The following objects are mapped:

Artist:


```
{
"uuid":"ea9a0e4b-1efd-4975-88c0-d7ea44c3f114",
"name":"Mastodon",
"uri":"/MusicCatalog_Server-1.0/rest/artist/ea9a0e4b-1efd-4975-88c0-d7ea44c3f114"
}
```


Song:


```
{
"uuid":"f8b268b0-b1b2-475c-9e87-8f0a9e80c476",
"name":"Deep Sea Creature",
"year":2001,
"length":280,
"votes":0,
"artist":{
"uuid":"ea9a0e4b-1efd-4975-88c0-d7ea44c3f114",
"uri":"/MusicCatalog_Server-1.0/rest/artist/ea9a0e4b-1efd-4975-88c0-d7ea44c3f114"
},
"uri":"/MusicCatalog_Server-1.0/rest/song/f8b268b0-b1b2-475c-9e87-8f0a9e80c476"
}
```


User:

```
{
"uuid":"c86dda27-aeba-4787-827f-b8f64c1e7ed2",
"nick":"username",
"password":"userpassword"
}
```


Authentication is realized by HTTP BASIC. Clients send their credentials as Base64 encoded string in HTTP header Authorization in each request. This mechanism does not provide confidentiality protection, so HTTPS protocols should be used when communicated over public networks.
The authentication functionality is made in own implementation of interface javax.ws.rs.container.ContainerRequestFilter. HTTP header Authorization is checked in each incoming request and credentials in header's value are checked against H-Store database. If this control succeeds id of the signed user is set in requests property. Otherwise HTTP status Unauthorized is returned in response. This control is performed over all incoming requests except creating new user.
The user id can be retrieved from request in further processing when needed, so the favorites songs of each user can be modified.

Resources in MusicCatalog_Server supports caching. When client obtains resource, it can sends string representation of locally cached data in HTTP header If-None-Match. Server finds requested data and compute its representation. The server's representation and values ot the HTTP header are compared. If these representations are equal the client's data are up to date and may not be sent over network. In this case server sets HTTP status NOT MODIFIED in response. Otherwise server returns requested data in the response. This approach reduces network bandwidth.
The representations must be created by same procedure on server and on client side. The digest algorithm MD5 is used. Transferred object is represented as digest of concatenation if its properties transferred to string. In case of collection of objects (result of search), the ArrayList is used to keep ordering. Collection is represented as digest of concatenation of object's representations contained in the collection.

The following picture depicts Java method in REST API. The value of If-None-Match HTTP header is one of the method parameters. It is than compared with hash representation of object searched in server database. The appropriate status in HTTP response is set.

```
@GET
@Path("/{id}/")
@Produces("application/json")
public Response getArtist(@PathParam("id") String artistId, @HeaderParam("If-None-Match") String tag){
    try {
        ArtistDTO artist = Artist.getArtist(artistId);
        if(artist == null){
            throw new WebApplicationException(Response.Status.NOT_FOUND);
        }
        String currentTag = artist.getETag();
        if(currentTag != null && currentTag.equals(tag)){
            return Response.status(Response.Status.NOT_MODIFIED).header("ETag", tag).build();
        }else{
            ArtistMapping am = Mapping.getArtistMapping(artist, this.uriInfo);
            return Response.status(Response.Status.OK).header("ETag", currentTag).entity(am).build();
        }
    } catch (HstoreException e) {
        throw new WebApplicationException(Response.status(Response.Status.INTERNAL_SERVER_ERROR)
                .entity(e.getMessage()).build());
    } catch (ProcCallException e) {
        logger.log(Level.SEVERE, e.getMessage(), e);
        throw new WebApplicationException(Response.Status.BAD_REQUEST);
    }
}
```

Figure 5 - Caching implementation example

H-Store database is accessed via its stored procedures. H-Store provides API for invoking stored procedures. Firstly a connection to the database must be created. Variable ipAddress contains IP address of arbitrary node of the H-Store cluster.

```
Client client = ClientFactory.createClient();
client.createConnection(ipAddress, 21212);
```

Stored procedure can be invoked by method callProcedure on Client instance. First parameter is name of the procedure, following arguments are passed to the stored procedure:

```
client.callProcedure("AddArtist", id, name);
```

Stored procedure can return long or VoltTable data types. In case the the stored procedure returns anything else long it must return VoltTable instance. From this instance client retrieve the data by the data type and column name or index:

```
VoltTable[] results = client.callProcedure("GetArtistByName", name).getResults();
                        VoltTable result = results[0];
                                for(int i=0; i<result.getRowCount(); i++){
                                        VoltTableRow row = result.fetchRow(i);
                                        String id = row.getString("id");
                                }
```

The MusicCatalog_Server project can be built by Maven into war, which can be deployed into web server. This project has dependencies to H-Store API. It is not distributed via public Maven repositories, so it have to be installed into your local Maven repository to enable building the application. These dependencies can be found in attached CD.

The builded archive can be deployed to servlet container running on the same machine as the H-Store database

Development for H-Store database

As H-Store database communicates via stored procedures the development process takes more time than development for other databases. If new stored procedure must be added or changed, the H-Store must be stopped and then the whole process of building must be made to change the project jar. There is one more annoyance binded with the building. If an error occured during the building error messages are not always relevant.

During the development a bug in H-Store has been detected. The H-Store claims to support same SQL syntax as VoltDB, but using SELECT statement nested in JOIN throws an error while building. This SQL syntax is supported by VoltDB documentation[4][36].

The H-Store database enables command logging to preserve durability. But a functionality for restoring the database after shutdown is not yet implemented. This is implemented in VoltDB paid licences only.

Client part

Client part is Android application, it communicates with server via REST API and caches data into its local SQLite database.

Android

Android is an operation system based on Linux kernel. It was primarily developed for touchscreen mobile devices. In September 2013 one billion devices using Android operating system has been achieved. Android's source code is released by Google under open source license, but in most devices this open source system is combined with proprietary software from hardware manufacturers. Since 2008, when the first commercial version of Android was released, several versions has been released. The current version is Android 4.4.2 KitKat.[29]

Development for Android platform

The Android Software Development Kit (Android SDK) contains Java libraries and developer tools for building, testing and debugging Android application inside supported development environment. The application can be deployed and debugged in virtual Android device or in real Android device, which is connected to computer. There is also possibility to use Android NDK, which allows you to develop part of the application in C or C++ programming language. This is primarily for CPU - intensive operations, but most of the applications does not need it.

Database

The SQLite database embedded in Android is used as a cache for data retrieved from the server and for tracking changes, which are done when the device is offline. Primary keys are generated by Java class UUID and are named _id, which is required by some Android classes for accessing database. Schema of the database is very similar to the one used on the server. Database is accessed via singleton instance of DatabaseHelper, which extends SQLiteOpenHelper from Android API.

The table User is missing, because only one user can work with the cached data. If credentials are changed the content of the database is deleted.

- Artist - contains artist metadata
- Song - contains metadata about song, this database contains data for only currently signed user, so no bindings to user is necessary
    - favorite - marks favorites song, integer datatype is used, because SQLite does not have boolean data
    - typeChanges - table for tracking changes made when offline
    - _id - identification of the row in table
    - source_table - name of the table, where the change happened
    - type - type of the change, eg. ADD or REMOVE for change of favorite song
    - changed_id - identification of changed row
    - created - time, when the change was performed

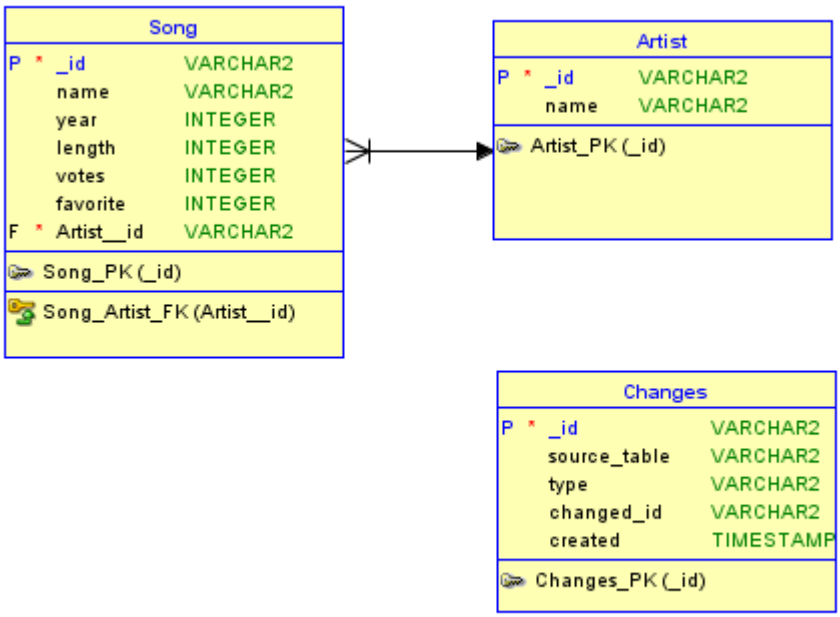

Figure 6 - SQLite database schema

Functionality

There is the functionality implemented for accessing H-Store database via REST API. The same functionality is also implemented for local SQLite database for caching. This is used when the device is offline. Moreover it is implemented storing of changes performed offline and retrieving these changes from database, when the synchronization comes into play.

The following parts of application functionality are described:

- searching
- favorites
- addition of new record
- changing of record
- changing favorites
- synchronization

Searching

User can search artists or songs. When one of the search is started, first the application tries to find result in local cache (SQLite). Then one of the following scenarios happens.

If the applications is currently in offline state, the result from local cache is returned.
If network connectivity is present, application perform same searching via REST API. The HTTP header If-None-Match with value of hashed representation obtained from local cache is set in HTTP request. If the actual data on server are different from the client's data, server sends its actual data back to client. In case the data on the server are same as local data, response has status NOT MODIFIED.
If the response has the status NOT MODIFIED, data obtained from the local cache are returned.

Following picture depicts calling REST API from Android application. The header If-None-Match is set, its values is retrieved from object passes as parameter – data from local cache. When a response is returned from the server, its status is mapped to returned value, because of later processing. This processing depends on context, when the function is called.

```
public DtoResponse<ArtistDTO> getArtistById(ArtistDTO artist) throws RestException{
    HttpClient client = new DefaultHttpClient();
    HttpGet get = new HttpGet(this.resourceAddress + "/" + artist.artistId);
    get.setHeader("Accept", "application/json");
    get.setHeader("Authorization", this.getAuthorizationHeaderValue());
    get.setHeader("If-None-Match", artist.getETag());
    try {
        HttpResponse response = client.execute(get);
        int code = response.getStatusLine().getStatusCode();
        DtoResponse<ArtistDTO> resp;
        switch(code){
        case 200:
            String result = EntityUtils.toString(response.getEntity());
            JSONObject json = new JSONObject(result);
            artist = getArtist(json);
            resp = new DtoResponse<ArtistDTO>(artist, Status.MODIFIED);
            return resp;
        case 304:
            resp =  new DtoResponse<ArtistDTO>(artist, Status.NOT_MODIFIED);
            return resp;
        case 303:
            resp = new DtoResponse<ArtistDTO>(artist, Status.SEE_OTHER);
            return resp;
        case 400:
            resp = new DtoResponse<ArtistDTO>(artist, Status.BAD_REQUEST);
            return resp;
        case 500:
            resp = new DtoResponse<ArtistDTO>(artist, Status.INTERNAL_SERVER_ERROR);
            return resp;
        default:
            resp = new DtoResponse<ArtistDTO>(artist, Status.ERROR);
            return resp;
        }
    } catch (ClientProtocolException e) {
        Log.w(this.getClass().getName(), e);
        throw new RestException(e.getMessage());
    } catch (IOException e) {
        Log.w(this.getClass().getName(), e);
        throw new RestException(e.getMessage());
    } catch (JSONException e) {
        Log.w(this.getClass().getName(), e);
        throw new RestException(e.getMessage());
    }
}
```

Figure 7 - Caching client implementation example

In case the data obtained from local cache are not actual. The data obtained from server are stored in SQLite database in case the record does not exists in SQLite. Otherwise existing record is updated. Actual data are returned. If the songs are searched, artist of each found song is also verified.

Favorites

When user wants to search his favorites, the local database is firstly searched. Then following scenarios can occur:

If the mobile device is offline the result from local database is returned

Otherwise the checksum of favorites songs retrieved from SQLite is calculated and sent to server in If-None-Match HTTP header. Server finds favorites songs for user, which is binded with the request and compare digest calculated from local data with value of HTTP If-None-Match header sent by client.
If these values are equals HTTP status NOT MODIFIED on response is set. Client returns data retrieved from local cache
Otherwise server returns list of favorites songs, which were found in H-Store database. Client then compares the list of song retrieved from SQLite database with the list returned from the server. These list are ordered by song id, so the client can recognize, which song was added to favorites or removed from favorites on the server. This can happen, when user was modifying his favorites songs from different device.

Addition of new record

When the new artist or song is added it is checked, it is first checked if the same record already exists in the local cache. If the mobile device is online, the existence is also checked by calling the server. If the record exists, new one is not added. If it does not exist the record is always stored into local cache.

If the device is currently online, the new record is sent to server to be stored. If the device is offline, the new record in SQLite's table Changes is created. Value of column source_table is set to Artist or Song, depends what record was added. Column type is set to ADD. Changed_id column contains _id of the newly inserted record. In the column created the actual time is stored.

Changing of record

In this application user changing of record can be shown on adding a vote to song. This can have two scenarios:

If the application is online, the request is sent to server, it adds vote to the specified song and returns back the song with actual values. The song is then updated in SQLite database
If the application is offline, vote count is incremented in local cache and new record in table Changes is created. The source_table column has value Song. Type column is set to VOTE value. Changed_id column contains _id of the voted song and created contains the actual time.

Changing favorites

User can change his favorites songs in online or offline mode. In both scenarios the selected song is first marked as favorite in local cache.

- In online mode a request is sent to the server
- In offline mode new record is inserted into Changes table. The column source_table has value Favorites. Type column is set to one of the values ADD or REMOVE, depends whether adding or removing from favorites songs. Changes_id contains _id of the song. Current time is set in created column.

Synchronization

When the application is used in offline mode the performed changes are stored in table Changes in SQLite database. For each record in the table it can be reconstructed, which change was done, which row in table Artist or Song was modified and when the change was done.

When the user performs some action in the application, the network connectivity is being checked. If the mobile device is connected to the network, first the table Changes is checked, whether contains any record. If the table is empty application continues with the original task. If not the list of changes, ordered in ascendence by created column, is iterated and each change is replayed to the server. If the change succeeds, it is deleted from the Changes table. After processing of all the changes, the application continues with the original task.

Records in Changes table can have following values in source_table column:

- Artist - new artist was added, when the device was offline. Type column can contain only value ADD, so REST method for adding new user is called, body of request contains artist serialized to JSON. Artist is referenced by the column changed_id.
- Song - the type column can contain one of the two values:
    - ADD - new song was added, REST method for adding new song is called, body contains serialized song, which is referenced by changes_id column
    - VOTE - use added vote for song, which is referenced by changed_id column, request is made to resource for adding vote
- Favorites - user could add or remove song in his favorites
    - ADD - song referenced by changed_id was marked as user's favorite, request to resource favorite is made, body contains serialized id of the song
    - REMOVE - song referenced by changed_id was unmarked as user's favorite, request is sent to resource favorites to remove the song from favorites

A failure can happen during the process of synchronization. It can be caused by several reasons. The device can loose the network connectivity while communicating with the server, server can be unavailable, REST API can return HTTP status INTERNAL SERVER ERROR, or database constraint can be violated - server returns HTTP status BAD REQUEST or SEE OTHER, if the record already exists.

- In case of missing network connectivity or INTERNAL SERVER ERROR, process of synchronization is paused
- In case of BAD REQUEST or SEE OTHER currently processed change and the following changes will be deleted

Application description

The application consists of nine activities (screens), where the user performs described actions. When the application is launched a main activity is started. First the user has to sign in or add new user account, otherwise no operations are enabled. After the user fills in his credentials he can access the functionality via the menu button with following items.

- Search song - starts an activity, which enables to search song by song's name and displays result in list below the form, after click on found song, activity displaying selected song's metadata is started, user can also add new song via menu
- Search artist - launches an activity for searching artists, after clicking on found artist, list of his songs is displayed, by clicking on song, the activity with song's metadata is activated, user can add new artist via menu button
- Favorites - start activity, which lists user's favorites songs, after clicking on song the activity with song's metadata is displayed
- Changes - for debugging reason, it launches activity displaying content of SQLite's table Changes
- Delete database content - for debugging reasons, deletes whole SQLite database

In the activity, which displays metadata of selected song, can user add vote for song and add/remove song from favorite songs via menu button.

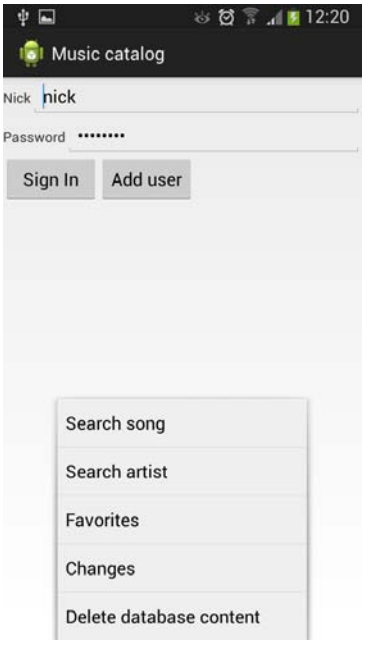

Figure 8 - Main activity printscreen

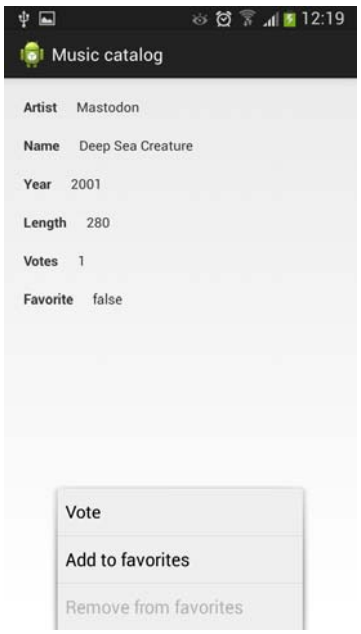

Figure 9 - Song activity printscreen

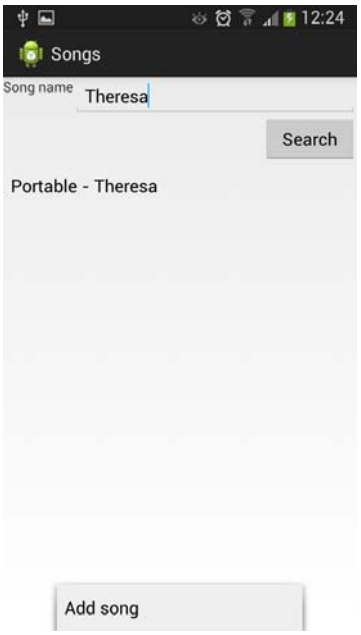

Figure 10 - Songs activity printscreen

8. Testing

The developed solution, which has to be tested, consists of three components (Android client application, web application and H-Store benchmark). The projects MusicCatalog_server and Android application contains test suites to verify integration and functionality of the solution.

Tests in musiccatalog_server

First test suite is in project MusicCatalog_server. The TestNG framework is used to test a communication between web application running in servlet container and H-Store database. These components communicates via calling stored procedures for H-Store API. The stored procedure is called by invoking method callProcedure, its first parameter is name of the stored procedure, followed by arbitrary number of Objects. Wrong parameters of the callProcedure method are not explored in compile time, but in runtime. So it is very important to test every method, which calls stored procedures.

TestNG testng.xml configuration file is located in package cz.cvut.fel.music_catalog.server.testing. The tests can be started by this file in Eclipse or by maven. The H-Store database must be running, when tests are started.

Android testing

The Android testing framework[30] is integrated in Android SDK, it is testing framework based on JUnit. This framework provides own TestCases to test individual components of Android application - for Android Activities, Services or UI. Pure Java classes without dependency on Android SDK can be tested by original JUnit. Mock objects for resources of Android system are also provided. Android testing framework also provide so called monkeyrunner for monkey test. The tests can run on Android virtual device or on physical Android device.

The Android tests are in standalone project called MusicCatalogTest, this project is binded to MusicCatalog Android application. The project contains five test cases. In these test cases an existence of specific user is tested, otherwise the new user is register and SQLite database is cleared. This is done in setUp method, which is called before the test methods. In test methods new artist, song etc. are added. The methods for querying local database and server REST API are used, status of response is also checked to test caching via HTTP headers. The tests can be started via Eclipse or command line, but installation of Android SDK is required. The server part of the solution must be running, when the Android tests are started.

Testing data

To test RESP API or Android client manually some data in server database are needed. For this purpose a sample Java application was developed. This application must be placed in the same directory as H-Store base directory. The application fills the H-Store with data from SQLite database, which is part of Million Song Dataset distribution. First the application creates user with nick “nick” and password “password”, then it populates the H-Store with data. The project is named MusicCatalog_Loader and can be built and started by following ant command:

ant invoke

9. Conclusion

In this thesis several NewSQL databases were described. As most of them are proprietary and their concepts are not described into details. The H-Store database selected for sample application is open-source and its principles are desribed in the documentation. The solution for accessing this database from mobile device was proposed and implemented. The developed application was documented, also examples how to develope an application for H-Store were described.

As the cloud infrastructure was not present, development and testing was done on single computer. Virtualization was used to run H-Store database. Single computer is not common environment for running NewSQL database, because it lacks main memory capacity and processor cores. Because of this the amount of data used to populate the database was limited. Measurement of parameters was not done, because it would not be relevant.

10. Bibliography

[1] NewSQL – The new way to handle big data [online] <http://www.linuxforu.com/2012/01/newsql-handle-big-data/>
[2] NewSQL [online] <http://en.wikipedia.org/wiki/NewSQL>
[3] Clustrix Documentation [online] <http://docs.clustrix.com/display/CLXDOC/Home>
[4] H-Store Documentation – Writing new benchmark [online] <http://hstore.cs.brown.edu/documentation/development/new-benchmark/>
[5] MySQL Cluster Overview [online] <http://dev.mysql.com/doc/refman/5.5/en/mysql-cluster-overview.html>
[6] NuoDB Online Documentation [online] <http://doc.nuodb.com/display/doc/NuoDB+Online+Documentation>
[7] VoltDB Documentation [online] <http://voltdb.com/dev-center/documentation/>
[8] Multiversion concurrency control [online] <http://en.wikipedia.org/wiki/Multiversion_concurrency_control>
[9] A. Pavlo, E. P. C. Jones, and S. Zdonik, "On Predictive Modeling for Optimizing Transaction Execution in Parallel OLTP Systems
[10] R. Kallman, H. Kimura, J. Natkins, A. Pavlo, A. Rasin, S. Zdonik, E. P. C. Jones, S. Madden, M. Stonebraker, Y. Zhang, J. Hugg, and D. J. Abadi, "H-Store: a High-Performance, Distributed Main Memory Transaction Processing System
[11] J. DeBrabant, A. Pavlo, S. Tu, M. Stonebraker, and S. Zdonik, "Anti-Caching: A New Approach to Database Management System Architecture"
[12] A. Pavlo, C. Curino, and S. Zdonik, "Skew-Aware Automatic Database Partitioning in Shared-Nothing, Parallel OLTP Systems"
[13] Sedivy, Jan; Barina, Tomas; Morozan, Ion; Sandu, Andreea - MCSync - Distributed, Decentralized Database for Mobile Devices – Bengaluru 2012
[14] Lars Vogel - Android SQLite and ContentProvider – US, 2012
[15] Williams, D. - Optimal parameter selection for efficient memory integrity verification using Merkle hash trees - Ithaca, NY, USA - 2004
[16] SQLite [online] <https://sqlite.org/>
[17] SQLite [online] <http://en.wikipedia.org/wiki/SQLite>
[18] Saving data in SQL databases [online] <http://developer.android.com/training/basics/data-storage/databases.html>
[19] Android SQLite database and content provider – Tutorial [online] <http://www.vogella.com/tutorials/AndroidSQLite/article.html>
[20] H-Store GitHub repository [online] <https://github.com/apavlo/h-store>
[21] Merkle tree [online] <http://en.wikipedia.org/wiki/Merkle_tree>
[22] Riak hash tree [online] <https://github.com/basho/riak_kv/blob/master/docs/hashtree.md>
[23] Vector clock [online] <http://en.wikipedia.org/wiki/Vector_clock>
[24] Why Vector Clocks are easy [online] <http://basho.com/why-vector-clocks-are-easy/>

[25] Why Vector Clocks are hard [online] <http://basho.com/why-vector-clocks-are-hard/>
[26] Universaly unique identifier [online] <http://en.wikipedia.org/wiki/Universally_unique_identifier>
[27] Million song dataset [online] <http://labrosa.ee.columbia.edu/millionsong/>
[28] What is HDF5 [online] <http://www.hdfgroup.org/HDF5/whatishdf5.html>
[29] Android (operating system) [online] <http://en.wikipedia.org/wiki/Android_(operating_system)>
[30] Testing Fundamentals [online] <http://developer.android.com/tools/testing/testing_android.html>
[31] H-Store Documentation – Executing H-Store [online] <http://hstore.cs.brown.edu/documentation/deployment/executing/>
[32] H-Store Documentation – Environment setup [online] <http://hstore.cs.brown.edu/documentation/deployment/environment/>
[33] H-Store Documentation – Building H-Store [online] <http://hstore.cs.brown.edu/documentation/deployment/building/>
[34] VoltDB Documentation – Designing the data access [online] <https://voltdb.com/docs/UsingVoltDB/DesignProc.php>
[35] VoltDB Documentation – Designing your VoltDB application [online] <https://voltdb.com/docs/UsingVoltDB/ChapAppDesign.php#SecDBDesign>
[36] VoltDB documentation – SELECT [online] <https://voltdb.com/docs/UsingVoltDB/sqlref_select.php>

11. Attachments

Included CD content

Included CD contains following directory structure:

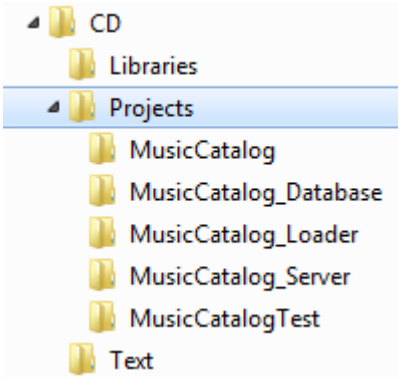

Figure 11 - CD content

Content of directories

- Libraries – jar dependencies, which must be installed into local Maven repository
- Projects
  - MusicCatalog – Android application source codes
  - MusicCatalog_Database – Source codes of H-Store project
  - MusicCatalog_Loader – application for populating the H-Store
  - MusicCatalog_Server – source codes for web application
  - MusicCatalogTest – Android tests
- Text – diploma thesis text