Migrating a mobile application towards a distributed database for simplified synchronisation

Knut Lorenzen
Abstract

Migrating a mobile application towards a distributed database for simplified synchronisation

Knut Lorenzen

As mobile applications are often dependant on cloud services and connected through unreliable radio networks, application developers may find themselves implementing custom caching and synchronisation algorithms if the application is to operate flawlessly while offline.

Relational databases have been the predominant architecture for persistent storage for a long time. With the emergence of the real time web, distributed schema free databases known as NoSQL have gained widespread adoption in recent years.

This thesis evaluates the benefits of a distributed, document-oriented database over a relational database for a mobile business application that needs to provide offline usage.

A prototype for an existing building inspection application invoking an embedded NoSQL database has been developed for this purpose.

While the NoSQL database provides built-in replication capabilities for the mobile application, it is clearly limited compared to SQL when it comes to modelling highly structured data.
Contents

1 Introduction 3
   1.1 Mobile applications ........................................ 3
   1.2 NoSQL Databases ........................................... 3
   1.3 Paperless construction inspections .......................... 4
   1.4 About this thesis ............................................ 4

2 Problem Description 6
   2.1 About iSpect .................................................. 6
   2.2 iSpect Requirements .......................................... 6
      2.2.1 Architecture .............................................. 6
      2.2.2 Construction Fault Data Model .......................... 8
   2.3 Current Implementation ...................................... 11
      2.3.1 System Architecture ...................................... 11
      2.3.2 Fault Model Relational Schema .......................... 12

3 Background 16
   3.1 Mobile Application Backend Architectures .................. 16
      3.1.1 Mobile Cloud Computing ................................. 16
      3.1.2 iCloud Storage APIs ...................................... 16
   3.2 Distributed Databases ....................................... 17
      3.2.1 Relational Databases ..................................... 17
Chapter 1

Introduction

1.1 Mobile applications

Smartphones and tablet computers have made a breakthrough recent years as they advanced technologically and became more consumer oriented. Devices mainly from Apple, but also Samsung, HTC and others running operating systems like Apple’s iOS or Google’s Android are replacing classical personal computers (including laptops) for uses such as accessing web services or digital entertainment. For third party application developers these platforms have considerable limitations compared to desktop platforms: Due to limited resources in terms of battery life, memory and computational power, applications can not expect to run as persistent daemons and are automatically terminated and revived on demand by the system. On some platforms applications are also subject to approval by the manufacturer, and the available APIs and libraries are usually strongly limited. Initially this had prevented such devices from widespread business use. However this is slowly changing as tablets and smartphones have become quite ubiquitous and employees push companies to use these in the work place, too.

1.2 NoSQL Databases

In recent years so-called NoSQL databases, which do not store data in relational table schemas, have become increasingly popular with web services. Their main advantages over relational databases include load scalability and better performance for very large data sets. NoSQL databases are designed for a distributed architecture and allow simpler replication between nodes than traditional relational databases. While they sacrifice well established
consistency features, most NoSQL databases provide what is called eventu-

al consistency, i.e. the delayed propagation of updates throughout a clus-
ter. CouchDB is a document-oriented database that stores JSON formatted
objects and provides a RESTful HTTP interface. CouchDB is especially
popular for its master-to-master replication capability.

1.3 Paperless construction inspections

Construction projects involve a multitude of contracting work, e.g. masons,
roofers, plumbers, electricians etc. hired by a supervising organisation. To
ensure quality, each segment of work given to a contractor needs to be in-
spected and approved throughout all stages of a construction project. For
this purpose, inspection service providers have emerged. The interaction
between supervisors, contractors and inspection companies creates a lot of
paper work that can delay the contractors payment, putting them under
financial pressure and leading to poor quality. Carrying out inspections
paperless with a PDA or tablet computer can speed up the inspection work-
flow significantly, as inspection reports do not have to be noted on a sheet
and subsequently entered into an IT-system by an office worker, but rather
directly on location by the inspector. It also creates potential to classify
construction faults and analyse them statistically, which can be helpful to
improve the overall construction process [8].

1.4 About this thesis

This thesis documents the migration process from a relational database to
a NoSQL database used by a mobile application. This has been done in
cooperation with Selling Solutions AB who is developing an iPad application
for building inspections. The application is designed to tightly interoperate
with a CRM web service which is also developed by Selling Solutions.

The data sharing between the mobile app and the web service has been one
of the key deficiencies of the overall system. Because the app is used on
construction sites, it must operate seamlessly even when no connection to
the internet is available. The approach to this problem so far has been to
extract a subset of the data from the shared database and cache it on the
device. This however has introduced the problem of cache invalidation and
the necessity to update the on-device cache as frequently as possible. The
solution proposed here is the use of NoSQL database technology because it
is designed for distributed architectures. As mobile devices often work in
conjunction with a backend service, we want to showcase how this technology
can be adapted for this common architectural pattern.

While switching to a distributed database frees the application developer from implementing synchronisation algorithms, it introduces new challenges:

**Schema Migration** At the beginning of this project, a complex relational database schema already existed on which other software components depend. While the data migration from SQL to NoSQL itself is trivial, the available query operations in NoSQL are less powerful and can make it challenging to migrate more complex schemas.

**Conflict resolution** NoSQL can solve the problem of synchronisation between distributed nodes, but it leaves the programmer to deal with conflicts that will occur when the same data record is modified on two separate nodes. Application specific conflict resolution and merge strategies need to be considered.

**Eventual consistency** Replacing a relational database with a distributed NoSQL database usually means sacrificing hard consistency. Since in this case the architecture is distributed prematurely (and not by choice), this is not a problem introduced by the migration to NoSQL. But still it needs to be analysed whether the delayed propagation of updates between mobile device and backend poses any problems and how to mitigate them.

In conclusion we found that the migration enables almost effortless synchronisation between mobile device and backend database. The NoSQL database components available for the mobile platform still lack maturity and could provide stronger abstraction. Unfortunately, the document-oriented NoSQL database does not provide a query and update API as feature rich as SQL. Modelling graph relations proved to be cumbersome.
Chapter 2

Problem Description

2.1 About iSpect

iSpect is an iPad application tailored for construction inspection business. It assists building inspectors in carrying out inspections and submitting reports to their back office.

As a testing ground, a small subset of iSpect’s feature set, the construction fault creation, was taken to build a prototype for a CouchDB based backend.

The interaction between the customer, the back office, the web CRM, the inspector and the mobile app of an inspection is depicted in figure 2.1:

2.2 iSpect Requirements

2.2.1 Architecture

Master DB

The iPad application is part of a distributed IT system and shares its data model with a customer relationship management system (CRM) and potentially other components.

The construction fault model is stored in this database, and inspection reports created with iSpect need to be written to it.
1. Customer calls and asks for an available inspector

2. Order department fills in the required parameters and queries the CRM for available inspectors

3. The CRM invokes the web service

4. Web service replies with available inspectors

5. The web service sends the inspection to the mobile application

6. The inspector carries out the inspection

7. The inspector sends the inspection data back to the web service

8. The Web service pushes the new report back in to the CRM

**Figure 2.1: Use Case**

**Offline Mode and Local Storage**

Since this is a mobile application, iSpect must be able to perform optimal even under high latency to the master db (e.g. over cellular network) or
even an interrupted connection. Therefore all data must be stored to a local database.

**Synchronisation**

The local and remote database should be synchronised as frequently as possible to minimise the occurrence of conflicts.

### 2.2.2 Construction Fault Data Model

**Decomposition Of A Construction Fault**

![Construction Fault Diagram]

Figure 2.2: Construction Fault

A construction fault is comprised of five essential properties:

**Space**: Any top-level division inside an inspection object; typically rooms, e.g. kitchen, living room, balcony, garage, . . .
Unit: An integral, fixed part of a space, e.g. window, wall, ceiling, floors, ...

Object: Removable objects inside a space or attached to a unit, e.g. furniture, electrical and sanitary installations, outlets, handles, ...

Detail: A detail to an object or a unit.

Error Code: A textual description of what is actually wrong.

Only space and error code are mandatory. Unit, object and detail are optional. iSpect presents the inspector with predefined values for each of the construction fault properties to choose from.

Combinatorial Constraints And Graph Relations

Figure 2.3: Construction fault graph relations (compact view)

The given construction fault property values may not be combined arbitrarily. Instead, sensible combinations are defined by the data model. The following relations exist, as depicted in fig. 2.3 and fig. 2.4:

- space - error code, e.g. 'bathroom' + 'dirty'
- space - unit - error code, e.g. 'bathroom' + 'door' + 'handle missing'
- space - unit - detail - error code, e.g. 'bathroom' + 'door' + 'paint job' + 'faulty'
- space - object - error code, e.g. 'bathroom' + 'radiator' + 'leaking'
- space - object - detail - error code, e.g. 'bathroom' + 'radiator' + 'thermostat' + 'faulty'
Figure 2.4: Construction fault graph relations (expanded view)

- space - unit - object - error code, e.g. 'bathroom' + 'window' + 'glass' + 'crack'
- space - unit - object - detail - error code, e.g. 'bathroom' + 'wall' + 'cold water outlet' + 'faucet' + 'leaking'
Selection Sequence

The user creates a construction fault by selecting a sequence of values for each property (with some of them being optional). After every selection by the user, the app updates the view with valid choices as defined in the data model:

- The app will initially present a list of all available spaces.
- On space selection, related units, objects and error codes are displayed.
- On unit selection, the objects, details and error codes are updated.
- On object selection, the details and error codes are updated.
- On detail selection, the error codes are updated.

2.3 Current Implementation

2.3.1 System Architecture

The current version of iSpect uses Microsoft SQL Server (fig.2.5). SQL Server provides a web service API. iSpect uses the web service API for all data that doesn’t need to be accessed during an inspection:

- Inspection orders can be updated before arriving at an inspection site
- Inspection reports can be buffered and submitted later in case of missing connectivity.

However, the construction fault model is needed to log construction faults and therefore must be cached on the device. This is done through a periodic file export from SQL Server. A Sqlite database file is extracted from the main database and distributed to the devices through an App update.

One can easily see that this workflow is anything but efficient. Microsoft SQL Server is not a good choice as a backend for an iOS app. There is no SDK provided. A programmer has to interact with the bare bones of a HTTP API and parse the ODATA XML format, which results in a lot of boiler plate code.
2.3.2 Fault Model Relational Schema

The database schema for the construction fault model is as follows: For each construction fault property an entity table exists, with an ID column that serves as primary key. All relations (i.e. sensible combinations) between any of these entities are defined in the same relations table, DM_DefectRelations. As shown in figure 2.6, DM_DefectRelations consists of a primary key, dm_relID and foreign keys to each entity table. The foreign keys for non-essential construction fault properties (i.e. unit, object and detail) are allowed to be null.

Table 2.1 outlines the contents of DM_DefectRelations: Each row defines a graph relation as was described earlier. The columns spaceID and errorCodeID should always contain a foreign key, while unitID, objectID and detailID are allowed to have null values.

To query for rows in one of the entity tables with a given foreign key, iSpect uses joins. For instance, after the user has selected a space, iSpect will display the matching error codes. The query executed when doing this is shown in figure 2.3.2:

Another example shown in figure 2.3.2 demonstrates how error codes related to a pair of space and unit as selected by the user are queried. Other queries run after the user makes a new selection follow the same pattern.
Figure 2.6: Fault model relational schema
Table 2.1: Fault model relations table

<table>
<thead>
<tr>
<th>relID</th>
<th>spaceID</th>
<th>unitID</th>
<th>objectID</th>
<th>detailID</th>
<th>errorCodeID</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(auto) FK Spaces null null null FK ErrorCodes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(auto) FK Spaces FK Units null null FK ErrorCodes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(auto) FK Spaces null FK Object null FK ErrorCodes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(auto) FK Spaces FK Units FK Object null FK ErrorCodes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(auto) FK Spaces FK Units null FK Details FK ErrorCodes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(auto) FK Spaces null FK Object FK Details FK ErrorCodes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>(auto) FK Spaces FK Units FK Object FK Details FK ErrorCodes</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
SELECT DISTINCT
DM_ErrorCodes.dm_ErrorCodeID, DM_ErrorCodes.dm_ErrorCodeName
FROM DM_ErrorCodes
INNER JOIN DM_DefectRelation
ON DM_ErrorCodes.dm_ErrorCodeID =
DM_DefectRelation.dm_ErrorCodeID
WHERE DM_DefectRelation.dm_SpaceID = ? AND
DM_DefectRelation.dm_UnitID IS NULL AND
DM_DefectRelation.dm_ObjectID IS NULL AND
DM_DefectRelation.dm_DetailID IS NULL AND
DM_DefectRelation.dm_ErrorCodeID IS NOT NULL
```

Figure 2.7: SQL statement to query error codes related to a given space
SELECT DISTINCT DM_ErrorCodes.dm_ErrorCodeID, 
        DM_ErrorCodes.dm_ErrorCodeName 
FROM DM_ErrorCodes 
INNER JOIN DM_DefectRelation 
ON DM_ErrorCodes.dm_ErrorCodeID = 
        DM_DefectRelation.dm_ErrorCodeID 
WHERE 
        DM_DefectRelation.dm_SpaceID = ? 
        DM_DefectRelation.dm_UnitID = ? 
        DM_DefectRelation.dm_ObjectID IS NULL AND 
        DM_DefectRelation.dm_DetailID IS NULL AND 
        DM_DefectRelation.dm_ErrorCodeID IS NOT NULL

Figure 2.8: SQL statement to query error codes related to a given space, unit pair
Chapter 3

Background

3.1 Mobile Application Backend Architectures

3.1.1 Mobile Cloud Computing

The recent generation of mobile applications is tightly entangled with cloud computing[1]. This is founded partly in scarce resources (battery life, computational power, network bandwidth), where outsourcing demanding computation tasks to remote infrastructure has become a common pattern. Other driving factors are social networks which are built on scalable cloud infrastructure and where mobile apps are an add-on option to the web.

The dependence on cloud infrastructure has lead a big weakness in dealing with connectivity issues, which are far more prevalent in mobile networks than in their wired counterparts. Many mobile applications that rely on cloud infrastructure deal poorly with network partitions.

3.1.2 iCloud Storage APIs

In 2011, Apple introduced its own cloud service called iCloud. iCloud is used to exchange data between Apple’s own mobile and desktop devices including calendars, contacts and photos, and storing backups of mobile devices. iCloud also includes storage APIs for iOS and OS X developers[13], whose purpose is to sync application state and other user content between devices. A user who has an iPhone and an iPad and is using the same application on both devices is supposed to seamlessly switch between the two devices.

iCloud storage works as a local, embedded database that replicates with
iCloud. However, it is restricted to native applications running on Apple’s mobile and desktop operating systems. Also, data is contained within an iCloud user account. It is therefore not possible to share state between users, which gives iCloud APIs very limited use for application developers.

3.2 Distributed Databases

3.2.1 Relational Databases

The relational model, in conjunction with SQL as definition and query interface originally proposed by E. Codd in 1970, has triumphed most other architectures. The relational model defines so-called schemas, a decomposition of data into predefined tables, and those tables consist of cells who contain scalar, non-structured data. This allows to store data with minimal redundancy so that write operations are unlikely to cause inconsistent states.

The most widely used today are MySQL, Microsoft SQL Server, PostgreSQL, Oracle DB (all of which are self-contained daemons) and sqlite (an embedded database).

3.2.2 ACID Consistency And Transactions

Most relational database management systems offer so-called transactional consistency. A transaction is defined a sequence of operations, that, on the outside, appears to be executed either completely or not at all by the RDBMS. This is to ensure that only states that satisfy given integrity constraints are stored persistently. There are cases in which a sequence of operations, if executed only partly, would introduce an inconsistent state to the database or return inconsistent results. Potential causes of interruptions that could lead such partial executions are multi-user environments and system failures.

In database theory there are four famous consistency properties referred to by the acronym ACID[2] that stands for:

- Atomicity - All or nothing - A Transaction is either executed completely or not at all.
- Consistency - At all times, only consistent states appear on the outside
• Isolation - A transaction is not affected by other, concurrent transactions

• Durability - Once the transaction is completed, changes made by it are stored persistently.

There are different strategies to satisfy all of these four properties. A naive approach could be mutual exclusion, or pessimistic locking, where all transactions are serialised. This however would be inefficient as some operations are more time consuming than others, so small, fast operations would have to wait for complex operations to finish first.

3.2.3 CAP Theorem

In 2000, Eric A. Brewer of Berkely University stated one of the most widely cited theorems in distributed and database systems of the last decade, the CAP theorem. It contends that any network-shared data system can have at most two out of three properties[5]:

• consistency (C): All operations appear to be atomic and serialised

• availability (A): Operations will be performed without delay

• tolerance of network partitions (P): Data is distributed over two or more network nodes

If we apply this to relational databases, their design emphasises consistency and availability over partitioning tolerance (or distribution/replication capabilities) in CAP terms. Relational databases have long been the gold standard for storing persistent data fast and reliable. Their capacity has over time scaled with the development of computer hardware.

With the emergence and expansion of the world wide web, data-intensive applications such as search engines and social networks seem to have pushed RDBMS to their limits, or at least single-node computing. Web companies like Google and Amazon seemed to have just too much data for a single computer to handle and developed cluster-based architectures.

In other cases the bottle neck was not data, but load. Even if the amount of data didn’t exceed a single machine’s capacity, the sheer number of requests did. The solution to this were load balancing and replication.
3.2.4 The NoSQL Movement

Historically, there have been and still exist various models for database architecture other than the relational model, such as hierarchical or directory services, graph and object-oriented databases. But none of these have been able to break the (perceived) dominance of the relational model[4].

NoSQL is the keyword for a recent and rather significant development in database technology. Literally, it stands for “not only SQL“: NoSQL is a category of database management systems characterised by

- (obviously) query and manipulation APIs other than SQL,
- non-relational data storage and
- distributed cluster architectures.

NoSQL databases have emerged with the need for horizontal scaling (i.e. clusters) due to new application classes that demanded more resources than a single hardware node could provide, which can be summarised by real time web. The main forces behind this development have been web companies such as Google and Amazon, who needed to scale their web services to millions of users.

As of today, a wide variety of not only implementations but even categories of DBMS is associated with the term NoSQL (table 3.1):
<table>
<thead>
<tr>
<th>Category</th>
<th>Implementations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Eventually-Consistent Key-Value-Store</td>
<td>Dynamo</td>
</tr>
<tr>
<td></td>
<td>Voldemort</td>
</tr>
<tr>
<td></td>
<td>Dynamite</td>
</tr>
<tr>
<td></td>
<td>SubRecord</td>
</tr>
<tr>
<td></td>
<td>Mo8onDb</td>
</tr>
<tr>
<td></td>
<td>Dovetaildb</td>
</tr>
<tr>
<td>Ordered-Key-Value-Store</td>
<td>Tokyo</td>
</tr>
<tr>
<td></td>
<td>Tyrant</td>
</tr>
<tr>
<td></td>
<td>Lightcloud</td>
</tr>
<tr>
<td></td>
<td>NMDB</td>
</tr>
<tr>
<td></td>
<td>Luxio</td>
</tr>
<tr>
<td></td>
<td>MemcacheDB</td>
</tr>
<tr>
<td></td>
<td>Actord</td>
</tr>
<tr>
<td>Data-Structures Server</td>
<td>Redis</td>
</tr>
<tr>
<td>Tuple Store</td>
<td>Gigaspaces</td>
</tr>
<tr>
<td></td>
<td>Coord</td>
</tr>
<tr>
<td></td>
<td>Apache</td>
</tr>
<tr>
<td></td>
<td>River</td>
</tr>
<tr>
<td>Object Database</td>
<td>ZopeDB</td>
</tr>
<tr>
<td></td>
<td>DB4O</td>
</tr>
<tr>
<td></td>
<td>Shoal</td>
</tr>
<tr>
<td>Document Store</td>
<td>CouchDB</td>
</tr>
<tr>
<td></td>
<td>Mongo</td>
</tr>
<tr>
<td></td>
<td>Jackrabbit</td>
</tr>
<tr>
<td></td>
<td>XML Databases</td>
</tr>
<tr>
<td></td>
<td>ThruDB</td>
</tr>
<tr>
<td></td>
<td>CloudKit</td>
</tr>
<tr>
<td></td>
<td>Perservere</td>
</tr>
<tr>
<td></td>
<td>Riak Basho</td>
</tr>
<tr>
<td></td>
<td>Scalaris</td>
</tr>
<tr>
<td>Wide Columnar Store</td>
<td>Bigtable</td>
</tr>
<tr>
<td></td>
<td>Hbase</td>
</tr>
<tr>
<td></td>
<td>Cassandra</td>
</tr>
<tr>
<td></td>
<td>Hypertable</td>
</tr>
<tr>
<td></td>
<td>KAI</td>
</tr>
<tr>
<td></td>
<td>OpenNeptune</td>
</tr>
<tr>
<td></td>
<td>Qbase</td>
</tr>
<tr>
<td></td>
<td>KDI</td>
</tr>
</tbody>
</table>

Table 3.1: List of NoSQL DBMS[9]

Different approaches to categorise NoSQL DBMS exist The major distinc-
tions are in terms of data model and persistence design:

- **Persistence:** NoSQL DBMS make a compromise between persistence and performance. Some of them are pure in-memory databases and therefore very fast (e.g. Scalaris, Redis). Data loss is either avoided through replication or techniques such as Memtable or SSTable, which mean periodic snapshots and/or transaction logs to disk. Other implementations offer fully persistent on-disk storage (e.g. CouchDB, Neo4j).

- **Data model:** Columnfamily, key/value, document, graph are the most common categories. The tradeoff here is between raw speed for trivial operations (key/value) and the ability to handle more sophisticated data models gracefully. Key/value stores for instance are completely unstructured and are a good choice for applications with very large datasets with simple queries. Document stores on the other hand (also referred to as document stores, document databases or document oriented databases) offer the storage of encoded (e.g. JSON, XML, …), structured values and typically have more sophisticated query APIs.

### 3.2.5 Eventual Consistency

One big caveat of NoSQL DBMS is their weak consistency compared to relational DBMS. Due to their distributed nature updates need to be propagated between network nodes, which is subject to delay. It is therefore impossible to guarantee that two nodes have the same state at a given time, and therefore two clients accessing different nodes at the same time might get different results.

This renders NoSQL RDBMS unsuitable for certain classes of applications, especially in business and finance where ACID consistency is essential. For web search engines and social networks or sensor data it might acceptable if two users get different results, considering that the ability to handle large volume of clients or very large data sets (big data) is a prerequisite for some of these applications[4].

### 3.2.6 Document-Oriented Databases

Document-oriented databases are considered a compromise between fast, but devoid of (query-)feature key/value stores and fully structured, feature rich relational SQL based databases. The most widely used implementations today are MongoDB and CouchDB.
The name-inspiring term *document* is defined as a structured unit of data that is encoded in some standardised format such as JSON or XML. A document is usually associated with a unique key to be looked up by, and usually document stores allow the definition of secondary indices based on the content the documents. Unlike key/value stores, document databases have semantics about the structure of the data. But unlike relational databases, document databases are schema-free, i.e. the structure of individual documents is arbitrary, which makes them more flexible to changes in the data model.

A foreign key mechanism may be implemented by documents referencing each other, but most implementations lack basic referential integrity features such as foreign key constraints. It is therefore up to the application programmer to ensure that such internal references are valid.
Chapter 4

Prototype Implementation

Now that the underlying concepts and technological circumstances are established we move on to the design of the prototype implemented as a part of the thesis.

4.1 System Architecture

Looking at the overall system architecture, we identify three main software components interacting via CouchDB’s http API: The CRM web service, a CouchDB instance, and the mobile application. Or, more precisely, both the web service and the mobile application access a shared CouchDB database. The mobile app itself is composed of the application logic and a local DBMS. The app interacts only with its local DBMS and never with the CouchDB
service or the CRM. The local DBMS will replicate opportunistically (i.e. whenever the app is given execution time on the device and has an uplink) with the CouchDB service.

4.2 Software components

CouchDB [6]: An instance of CouchDB is set up to test the synchronisation. CouchDB is a server application and runs as a system daemon on the development work station. It has a web interface for administration, database inspection and manipulation, which is useful for testing.

CouchDB is a document-oriented database that uses JSON (Javascript object notation) encoded documents and provides a RESTful http API for all operations.

Documents are the smallest addressable units of data, i.e. whenever data is accessed or altered, a complete document is affected. They contain arbitrarily nested structures consisting of:

- Dictionaries, i.e. key-value collections. The root of a document must be a dictionary
- arrays, i.e. number-indexed collections,
- strings,
- booleans,
- numbers.

A document has a unique key or id (a string), that is unique inside a CouchDB database (an instance of CouchDB may host any number of databases).

CouchDB allows the definition of views through map/reduce functions written in Javascript. Views are a way to aggregate and index documents, and queries are run against these views. The Javascript code for the map/reduce functions are stored inside special documents, called design documents.

CouchDB uses multi version concurrency control (MVCC). The version history of every document is retained and updates will create a new revision that is appended to the B-tree storage on disk. To reclaim disk space, a compaction process can be scheduled by the user that will purge all non-current revisions on the local instance.

CouchDB is renowned for its master-to-master replication protocol. Like the query API, the replication protocol uses only http and JSON.
Any CouchDB instance can replicate passively or actively with any number of other instances. The replication is incremental and (optionally) filtered according to parameters. Replication can be performed once or set up to run continuously.

CouchDB detects conflicts, i.e. cases where single documents have been modified concurrently. On detection it resolves these preliminarily with a deterministic algorithm that choses a winning revision while retaining the losing revision(s). The benefit of this behaviour is that a CouchDB instance can make this decision without having to negotiate with other nodes (e.g. by determining the timely sequence of the changes and making a decision based on that), which would cause delays. The application programmer can override CouchDB’s automatic conflict resolution with custom merging algorithms.

**TouchDB** [7]: The iSpect prototype is deployed bundled with TouchDB

TouchDB is an embedded database (i.e. a library) and an attempt to port CouchDB to mobile devices. Since CouchDB is written in Erlang and runs as a self-contained process, it is not allowed to install it on iOS or Android because of platform restrictions by Apple and Google. Apple for instance prohibits app developers to ship applications that rely on interpreters or virtual machines like Erlang and enforces the use of iOS’s proprietary APIs that are exclusively available in C and Objective-C.

TouchDB is therefore written in Objective-C and actually uses Sqlite[10] internally. Unlike CouchDB it does not provide a passive http API, but an object-oriented one in Objective-C that mimics CouchDB’s http API. The compatibility to CouchDB is achieved through the ability to replicate with CouchDB instances.

**iOS** is Apple’s mobile operating system, providing the essential frameworks for networking and graphical multitouch user interface.

**Web CRM service** The CRM web service is treated as a black box throughout this document. It is simply mentioned to outline the actual application’s environment.

### 4.3 Schema Migration

The overall goal is to duplicate iSpect’s construction fault creation work flow with TouchDB as local DBMS. To get there, we first migrate the relational model described in ch.2.3.2 to CouchDB, as it is a more feature rich development environment than TouchDB. Unlike TouchDB, CouchDB has a web
interface to inspect and manipulate data, and to test and experiment with views and queries. Once the model is proven to be transferred to CouchDB, it can be easily synced to TouchDB.

4.3.1 Document design

The migration from the relational fault model schema is pretty straightforward: For each entity table (DM_Spaces, DM_Units, etc.) a corresponding document type is defined and each document contains the column values of one row. So for instance, a row from DM_Spaces

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>HasNrList</th>
<th>IsIndoors</th>
<th>IsOutdoors</th>
<th>SortingPriority</th>
</tr>
</thead>
<tbody>
<tr>
<td>0001</td>
<td>VARDAGSRUM</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>7</td>
</tr>
</tbody>
</table>

is translated into the document fig.4.2. There’s three observations to be made about it:

1. The document has a property called `type`. This is just a convention and nothing CouchDB recognises semantically.

2. The integer primary key column ‘ID’ is discarded. This column is used solely for foreign key referencing in the relational schema and has no semantic value of itself.

3. The document’s id `space_VARDAGSRUM` is a string concatenation between the document’s type and its name field.

As for the document’s id other options would have been possible, for instance a randomised string (which CouchDB automatically generates if no id is given on document creation). The point is, we want to have only one document with the name ‘VARDAGSRUM’, because the user shall only see one entry with that name when he picks a space. Unlike a relational DBMS, CouchDB has no unique constraints on fields. The only thing where CouchDB enforces uniqueness is the document’s id. So by this concatenation of a documents type and name fields in the id, we make these our primary key and prevent duplicates from being introduced.

The tables DM_Units DM_Object, DM_Detail and DM_ErrorCode are migrated in the same fashion as shown for DM_Spaces.

4.3.2 Entity Relations

There two principal approaches to modelling many-to-many entity relationships in CouchDB[11]:

26
Figure 4.2: A document representing a space

1. One document type holding a list of document ids to another.

2. Relationship documents, i.e. dedicated documents containing the references to two related documents.

Since there are no actual join operations in CouchDB’s querying API, the relationship documents would make querying a cumbersome two-step operation on the application side. In order to avoid this, the first option is preferable.

Example: Error Codes and Spaces

We want to showcase the relationship modelling, the necessary map/reduce functions and query parameters on the example of the relationship between spaces and error codes, which is the simplest one:

Let’s assume we have a space document like in fig. 4.2, and that this space has a direct relation to one or more error codes. We’ve just in the previous section decided to implement this by one side keeping references (i.e. document ids) to the other, which still leaves us with the choice which one. Either seems valid, but for now let’s just say the error code documents hold references to the spaces. We will see later that this turns out to be a good choice.

The error code document shown in fig. 4.3 is related to several spaces. If we recall, in iSpect the user is to be shown a list of error codes (if any) after he has selected a space. In order to find all the error codes related to that space, it would be convenient to have a view with all the relations between spaces and error codes. Because we are looking for error codes to a given space, the keys of the view should contain the spaces’ document ids and the values should be the names of the error codes.

Such a view is depicted in fig.4.4. The view can be queried with a space id
Figure 4.3: A error code document with relations to spaces

Figure 4.4: Rows of the view 'error codes by space'
function( doc ) {
    if ( doc.type == "error_code" )
        for each ( space in doc.spaces )
            emit( space, doc.name );
}

Figure 4.5: Map function for the view 'error codes by space'

as a key and will return just the error codes for that particular space (just
the values, or if necessary the full error code documents).

In fig.4.5 a map function is shown that will generate the view shown in fig.4.4.
This function is applied to all the documents in the DB. The if-clause filters
any documents that are not of type error_code. A for-each loop iterates
over the array called spaces (compare fig.4.3) and creates key-value-pairs
using the emit function, with the entries of the array (space document ids)
as key and repeating the error codes’s name as value.

Example: Error Codes, Spaces And Units

After seeing how a many-to-many relationship between two entities can be
implemented in CouchDB, the next step is to show a graph relationship
between three entities can be modelled. We therefore chose the relation
'space-unit-error code' as the next example. To establish the first leg of the
relation between spaces and units, we can use the same method as seen in
the previous chapter, just replacing error code with unit. A unit document
for instance would look similar to the error code document shown in fig.4.3,
with an array named 'spaces' containing document ids of related spaces.
So after the user has selected a space, not only does iSpect offer matching
error codes, but also units to build a space-unit-error code relation. If the
user does select a unit at this point, the error codes column must be updated
to show error codes that are part of this three-tier graph relation.

As the error codes are determined by the space and the unit the user has
selected, a view with unit-space pairs as keys and error codes as values would
be desirable. Keys in CouchDB views are not restricted to strings or scalar
values, but may also be compound structures like arrays, as we see in fig.4.6.

A way to achieve such a view could be to store space, unit pairs in the error
code documents. Making use of JSON’s nested structures, one could also
put arrays of unit document ids inside a dictionary with space ids as keys, as
shown in fig.4.7. A map function similar to the one seen in fig.4.5 can then
iterate over the structure under the property space_units and generate the

29
<table>
<thead>
<tr>
<th>key</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>[space_ALTAN, unit_BARRIÅR]</td>
<td>EJ KOMPLETTERAD</td>
</tr>
<tr>
<td>[space_ALTAN, unit_FÖNSTER]</td>
<td>EJ KOMPLETTERAD</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>[space_BADRUM, unit_FÖNSTER]</td>
<td>EJ KOMPLETTERAD</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>[space_BASTU, unit_FÖNSTER]</td>
<td>EJ KOMPLETTERAD</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>[space_FASAD, unit_FÖNSTER]</td>
<td>EJ KOMPLETTERAD</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>[space_WC/DUSCH, unit_FÖNSTER]</td>
<td>EJ KOMPLETTERAD</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Figure 4.6: Rows of the view 'error codes by space, unit'

view, which if queried with a space, unit id pair would return the matching error codes.

All Views

Having established how the implementation of the graph relationships work in detail, in tab. 4.3.2 is a list of all the views that model the graph relations in the construction fault model. In principle, a single graph relation is not stored in one document, but split into parts and scattered in separate documents.

4.4 TouchDB API

This section gives a brief overview of the relevant classes of TouchDB’s API used in the prototype implementation.

CouchDocument represents a CouchDB document. In figure 4.8 the (instance) methods properties: and putProperties: can be seen. Instances of this class are a wrapper around a dictionary (e.g. NSDictionary in Objective-C). The premise of this class is that contents of a document already been brought into memory. To modify it, a programmer would retrieve the top-level dictionary with properties(), make his changes to it, and then write the changes back to the DBMS using putProperties(). putProperties is a non-blocking call that returns a RESTOperation object, which can be passed a callback to be executed on finish. This is convenient as allows to make use
Figure 4.7: An error code document with relations to spaces and units

<table>
<thead>
<tr>
<th>View name</th>
<th>Key</th>
<th>Relations stored in</th>
</tr>
</thead>
<tbody>
<tr>
<td>units by space</td>
<td>space id</td>
<td>unit.spaces</td>
</tr>
<tr>
<td>objects by space</td>
<td>space id</td>
<td>object.spaces</td>
</tr>
<tr>
<td>error codes by space</td>
<td>space id</td>
<td>error_code.spaces</td>
</tr>
<tr>
<td>objects by space, unit</td>
<td>[space id, unit id]</td>
<td>object.spaces.spaces_units</td>
</tr>
<tr>
<td>details by space, unit</td>
<td>[space id, unit id]</td>
<td>detail.spaces.spaces_units</td>
</tr>
<tr>
<td>error codes by space, unit</td>
<td>[space id, unit id]</td>
<td>error_code.spaces.spaces_units</td>
</tr>
<tr>
<td>details by space, unit, object</td>
<td>[space id, unit id, object id]</td>
<td>detail.space_unit.object</td>
</tr>
<tr>
<td>error codes by space, unit, object</td>
<td>[space id, unit id, object id]</td>
<td>error_code.space_unit.object</td>
</tr>
<tr>
<td>error codes by space, unit, object, detail</td>
<td>[space id, unit id, object id, detail id]</td>
<td>error_code.space_unit.object_detail</td>
</tr>
</tbody>
</table>

Table 4.1: All graph relationship views
of concurrency without the need to manage threads.

Just like CouchDB, TouchDB has design documents to store meta data about a database and, most importantly, code to generate map-reduce views. In TouchDB design documents are represented by the class CouchDesignDocument (figure 4.9). The two most common functions are the creation and the querying of views.

One key difference between CouchDB and TouchDB is the available languages for view and other functions. While in CouchDB the default is Javascript, TouchDB only supports Objective-C. To query data, a programmer would first create a view by defining a map function (and an optional reduce function, which is omitted in the example below):

```objective-c
CouchDesignDocument* design;
[design defineViewNamed: @"emailByName" mapBlock:
    MAPBLOCK({
        NSString* name = [doc objectForKey: @"name"];
        if (name) emit(name, [doc objectForKey: @"email"]);
    }) version: @"1.0"];
```

Let’s assume we have a database that contains information about users or employees etc. The map function above will create a view (index in CouchDB terminology) with the users’ names as keys and email addresses as values. This would be useful to look up a users email address for a given name. Bear in mind that the creation of the view is done initially and not at query time.

Now, in order to query a view, the application programmer will call CouchDesignDocument:queryViewNamed:
CouchQuery+ query = [design queryViewNamed: "emailByName ";

Figure 4.10: TouchDB CouchQuery class

This will yield a CouchQuery object (figure 4.10). The CouchQuery class has a start: method that will start the query, returning a RESTOperation object. The RESTOperation, if we recall from the CouchDocument class, represents an active request and has a callback mechanism that can notify the application on finish. The main purpose of the CouchQuery class is to set various query parameters (not shown here, but all of which can be found in the CouchDB http API), most importantly the keys.

Once the query has finished, the result can be accessed via the rows method. The CouchQueryEnumerator it returns is merely a container for CouchQueryRow objects (figure 4.11):

Figure 4.11: TouchDB CouchQueryRow class

Each row contains a key and value pair that was created by the map function above. Optionally, full documents can be also be fetched and accessed through the document method. For optimal performance this should be avoided though.

4.5 iOS Application

The iOS application developed along with this thesis serves the purpose of testing and demonstrating the creation of construction faults and replication between an iPad and a CouchDB instance.

The user interface (fig.4.12) is divided into two screens. Each screen contains three table views:
1 switch to toggle replication

2a navigates to screen 2

2b navigates back to screen 1

3 'save' creates a construction fault

---

Figure 4.12: iOS Prototype Screen 1 & 2

Screen 1:
- Spaces
- Units
- Objects

Screen 2:
- Details
- Error codes
• Locations (this is just a simple list of prepositions to be included in the construction fault, it is always the identical no matter what values are selected in the other columns)

The user interface behaves just as was specified in sec.2.2.2: Initially, only a list of spaces is visible in the left hand table of screen one. The user can make (and revert) selections in each table and the values shown in the tables will be updated according to the selection. The ‘save’ button in the top right corner of screen will create a construction fault in the local database.

The switch UI element in the top bar can enable or disable the replication with the CouchDB instance (whose location is 'hard-wired' into the code). The real iSpect would have no such option, as it should always replicate when possible. The switch is merely to test the behaviour in case of connectivity loss inside the iOS simulator.

4.6 Replication

If we recall the overall system architecture (fig.4.1), the main point of the prototype is replication with a CouchDB server. The advantages of this is that the app has its local replica of the whole database, allowing low latency operations without network delays and connectivity issues, and also very important for a mobile device, saving battery by avoiding frequent communication over wireless networks. As was pointed out earlier, TouchDB is compatible to CouchDB through CouchDB’s replication protocol, which is built on web technology i.e. HTTP and JSON.

Replication is a unidirectional operation between two database nodes. It makes sure all changes made to the source database since the last replication are applied to the target database. In CouchDB the direction of the replication is referred to by the terms pull (replication from a remote source) and push (r. towards a remote target).

Under the hood CouchDBs replication protocol build up on its multi version concurrency control: Each document has an update sequence number, and so has the overall database. This way CouchDB can easily determine which documents were changed since the last replication.

For our iOS prototype, we want changes to the construction fault model (which are done through a different application) to be replicated to the devices from the master database. We also want construction faults that are created on the devices to be replicated back to be master database, so therefore we need both push and pull.
NSURL* url = [NSURL URLWithString:
    @"http://127.0.0.1:5984/defect_model"];

[database replicateWithURL:url exclusively:YES];

Figure 4.13: Setting up replication in TouchDB

This is rather trivial to set up in TouchDB (4.13): We tell the local (TouchDB) database to replicate with the remote CouchDB database at a given HTTP URL. This will make both databases catch up first (replaying all changes since the connection got separated), and after that keep the connection alive to live replicate new changes. This is not a polling mechanism, but rather a 'sleeping' Websocket[12] TCP connection which allows both sides to asynchronously push changes and send packets only when necessary, preserving resources. TouchDB will automatically re-establish the connection in case it is interrupted, unless of course the application is terminated by the operating system (which may occur when the user opens other applications on his iPad).
Chapter 5

Evaluation

Let’s briefly recall the goals and requirements for our iPad prototype:

1. Use TouchDB to represent iSpect’s construction fault model.
2. Creation of construction faults using the construction fault model.
3. Replicate opportunistically both ways with a CouchDB instance.
4. An implicit, non-functional requirement is the performance. The iPad application should not suffer from any noticeable UI delays.

5.1 Construction Fault Model Migration

The construction fault model constrains how values for each fault property may be combined. The user selects values for the properties one by one, and after each selection the available values for unselected properties are updated according to the selection (see sec. 2.2.2). To verify that the values shown are in accordance with the relational representation of the model, one could compare the results of the TouchDB/CouchDB queries with the corresponding SQL queries.

Considering the large number of possible permutations this is not a practical approach. Instead, we resorted to a combination of a formal verification and random samples:

- The CouchDB construction fault model is created with the aid of a migration tool. The migration tool creates the relations by querying the table DM_DefectRelation (tab. 2.1) and adding the relations to
the CouchDB database (as illustrated in sec.4.3.2). These queries are identical to the ones used by the original iSpect implementation and should therefore translate the relations correctly. For instance the query shown in fig.2.3.2 is translated into the view 'error codes by space' (see tab.4.3.2). Each of these views was constructed with a corresponding SQL query.

• Sample verifications have been done manually by comparing the behaviour of the prototype with the original iSpect application.

• Though a complete verification was too complex, we have compared the CouchDB views (tab.4.3.2) quantitatively to their SQL equivalents and found that the numbers are in accordance.

5.2 Two-Way replication with CouchDB

We have intensively tested the continuous replication between the iPad application and the CouchDB instance.

5.2.1 Push

When construction errors are created with the prototype, they instantly appeared in the CouchDB database. To demonstrate this, we use the http testing utility curl to listen to CouchDB’s changes API:

```bash
curl -X GET "http://localhost:5984/defect_model/_changes?feed=continuous"
```

Now if we create a construction fault in the iPad application, we instantly see an event in curl:

```json
{"seq":1008,"id":"C649BEB7-F98F-4380-8D21-1A51E2A8B91E","changes":[{"rev":"1-8e329865f01e7515df36514438a77daa"}]
```

This tells us that a document with the given id was updated to the given revision. We could easily open CouchDB’s web interface to verify that the updated document is indeed the construction fault we have just created in the iPad prototype.

5.2.2 Pull

The simplest way to demonstrate pull replication it is to select a space in the iPad app and then manipulate relations through CouchDB’s web
So for instance, if we select the space ALLMÄNT, the unit BARRIER is shown. If we inspect the document BARRIER in CouchDB, we can remove the link to ALLMÄNT. As soon as we commit the change in CouchDB, the unit BARRIER disappears in the iPad app as well.

5.2.3 Offline Mode

We have tested these pull and push tests also with an interrupted connection between iPad and CouchDB (by disabling the replication through the iPad app’s UI). As soon as the replication is turned on again, all changes were replayed.

5.3 Performance

One obvious concern is the performance of TouchDB compared to Sqlite. We have benchmarked all TouchDB queries and compared them to their corresponding SQL queries. To benchmark TouchDB we used its built-in logging facilities (fig:5.1). The Sqlite command line utility has built-in benchmarks that we can use (fig.5.2). We found that the TouchDB queries required approximately the same execution time as their respective Sqlite statements, in some cases TouchDB even outperformed Sqlite.
SELECT * from DM_ErrorCodes
INNER JOIN DM_DefectRelation
ON DM_ErrorCodes.dm_ErrorCodeID = DM_DefectRelation.dm_ErrorCodeID
WHERE DM_DefectRelation.dm_SpaceID = 146
AND DM_DefectRelation.dm_UnitID = 186
AND DM_DefectRelation.dm_ObjectID = 468
AND DM_DefectRelation.dm_DetailID = 61 AND
DM_DefectRelation.dm_ErrorCodeID IS NOT NULL;
...
CPU Time: user 0.009249 sys 0.000290

Figure 5.2: Time measuring the equivalent statement in Sqlite
Chapter 6

Conclusions And Future Work

6.1 Conclusions

In summary, the project goals have been met:

- The replication works seamlessly and can cope with disconnects.
- The data model with graph relations was translated into documents.

The migration to TouchDB comes at a cost though:

- The iSpect development team needs to build competencies for CouchDB and TouchDB’s concepts, which represent rather specialised expertise compared to relational databases.
- TouchDB’s querying APIs are limited are have poor data aggregation capabilities compared to SQL. This results in a somewhat convoluted data model compared to the relational one.
- TouchDB is still an unstable, bleeding edge product. Basing a commercial product on it poses a somewhat of a risk, if the APIs should change frequently or the project were discontinued.

In general terms, NoSQL technology and its distributed approach are a good foundation for mobile application development, as this technology has already solved some of the fundamental pitfalls of distributed databases, especially as shown in in this case, replication protocols.
So far, not many mobile applications use replication, but most of them perform poorly or not at all when no internet connection is available. Database replication enables a new class of mobile applications, especially in the business sector where the ability to use an app independently of the location and network connectivity is more important than in the consumer sector.

6.2 Future Work

As for our industry partner, the next logical step is to migrate the overall system’s database towards CouchDB and completely replace MS SQL Server.

TouchDB’s architecture is, in the words of the developers, a compromise to save development time. Sqlite was chosen as storage backend because it is a proven, highly reliable and performant persistent storage, but building a document database on top of a relational database is likely to have scalability limitations. It might be desirable for the TouchDB project to replace Sqlite with a custom, native b-tree storage at a later point.

Over the course of the project, there have been some important developments to TouchDB. TouchDB originally started as lab project of a CouchBase Inc. employee. CouchBase Inc. has in the mean time decided to incorporate it into its portfolio and rebranded it as CouchBase Lite. Additionally, a Core Data[14] layer has been added which provides stronger abstraction. This could be adopted by iSpect to make the code even more compact and readable.
Bibliography


