Seminar

Immutable databases at the example of Datomic

University of Applied Sciences Rapperswil
Autumn Semester 2016
December 24th, 2016

Supervisor: Prof. Stefan Keller
Author: Tobias Stauber
Abstract

By now the concept of immutable databases has been around some time but still there is no real definition to it. For example to streaming databases (the topic that was worked by the other seminar attendees) there are the 8 requirements for real-time stream processing by Michael Stonebraker et al. (5).

At the time this paper was written, there was no established definition to immutable databases. Due to that, this paper contains an attempt to synthesise a short description about how a immutable database should look like from the existing examples for immutable databases.

In addition this paper contains and a short comparison between streaming databases and immutable databases at the example of datomic, followed by an introduction into datomic covering its architecture and datomics declarative query language: datalog.

Immutable databases did not turn out to do a job that could not be achieved by other means but it turned out beeing able to simplify quite a few popular usecases.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Introduction</td>
<td>3</td>
</tr>
<tr>
<td>2 Immutable Databases</td>
<td>4</td>
</tr>
<tr>
<td>3 Comparison</td>
<td>5</td>
</tr>
<tr>
<td>3.1 Streaming Databases</td>
<td>5</td>
</tr>
<tr>
<td>3.2 Immutable Databases</td>
<td>5</td>
</tr>
<tr>
<td>4 Datomic</td>
<td>7</td>
</tr>
<tr>
<td>4.1 Query language</td>
<td>8</td>
</tr>
<tr>
<td>4.1.1 Queries</td>
<td>8</td>
</tr>
<tr>
<td>4.1.2 What is datalog?</td>
<td>8</td>
</tr>
<tr>
<td>4.1.3 Datalog examples</td>
<td>10</td>
</tr>
<tr>
<td>4.1.4 Adding entries to database</td>
<td>16</td>
</tr>
<tr>
<td>4.2 Environment</td>
<td>17</td>
</tr>
<tr>
<td>4.2.1 Peer Library (Application Server)</td>
<td>17</td>
</tr>
<tr>
<td>4.2.2 Peer Server (Standalone JVM Server)</td>
<td>17</td>
</tr>
<tr>
<td>4.2.3 Transactor (Database Server)</td>
<td>18</td>
</tr>
<tr>
<td>4.2.4 Client Library</td>
<td>18</td>
</tr>
<tr>
<td>4.2.5 Shell</td>
<td>18</td>
</tr>
<tr>
<td>4.2.6 Console</td>
<td>19</td>
</tr>
<tr>
<td>4.3 Backends</td>
<td>19</td>
</tr>
<tr>
<td>4.4 CRUD / ARAR</td>
<td>20</td>
</tr>
<tr>
<td>List of Code-samples</td>
<td>20</td>
</tr>
</tbody>
</table>
1 Introduction

This document is the output of the "Databasesystems" seminar in my master of science engineering studies. The topic of the seminar was intended to be streaming databases but as of the limited amount of working examples for streaming databases Prof. Stefan Keller decided that I shall investigate into the topic of immutable databases.

As I delved into this subject, I soon realised that I need to narrow the topic down to just one example. My choice fell on datomic because every time I looked for a working sample of a specific database the only ready to deploy product mentioned was datomic. During my researches I could not find a lot of literature about datomic. So what I want to do is a small guide into datomic covering the query language.
2 Immutable Databases

the first thought that came to my mind was: How is an immutable database actually defined? There must be a way to change the dataset. Is it more like a Hadoop data-lake, where one can add new files but not manipulate existing ones?

In a classical relational database every cell in a table holds a value. Using the UPDATE statement the content of any cell can be changed to a new value, but now there is no way to find out, what the content was before.

Now immutable databases do not allow the user to change the content of a cell. Although the usecases of a database to which only new datasets can be added, but not updated seem to be very limited. So the only way to allow updating a cell and still keep the history is to maintain many different versions of the cell; to store a whole history of all the values this cell has ever held. To be still operable like a classic database the cell's handle always points to the newest value of the cell.

To modify a cell's handle for accessing other than the most current value, immutable databases are bringing an additional element to the classic relational data model; the element of time. By using time as an additional identifier to a cell, the cell's content itself can be made immutable.

For better understanding we will replace the term cell from the relational model by the term item. Whenever one wants to update an item in the database, a new item, containing the changes made to the old item, will be created and put in place of the current state. In a relational model instead of an item, whenever one wants to update a cell a whole new row has to be inserted. In this process of adding a new item and adjusting the item-handle to the newly inserted item, none of the information contained in the old item will be lost. Rather it will just be a previous state of the item.

As one can imagine the one big problem with this architecture is, that huge amounts of data accumulate fairly quickly. But today storage is not anymore that big of a problem as it has been a decade ago.

But back to the advantages, where will this history come in handy? Today we use a lot of computers to process data. Especially when it comes to machine learning and IoT these amounts of data can be huge. And mostly these data consists of measured results, for example vibrations in trains, voltage fluctuations in power grids, consumer profiles and so on. And there are always points in time that will make all the difference.

If one plans on creating statistics or use data for machine learning, long-time studies are inevitable.

Today there seems to be no real definition of immutable databases. The definition comes from the reference implementation. Next to datomic there was another immutable database released close to the end of the seminar. This database is called mentat developed by mozilla and is really similar to, or at least heavily inspired by datomic.
3 Comparison

3.1 Streaming Databases

This chapter covers the difference between streaming databases and immutable databases.
Streaming databases are designed to detect changes over time, trends, maximums and minimums, correlations and so on.
For doing this they need to know only a limited window in time. If the interest is on detecting to quick temperature changes, or logging the amount of errors per hour, or things like that. In the end one needs a buffer for this time but not a persistent database.
Streaming databases can be understood more like a business-logic layer sitting atop of a database. It can extract additional data from the stream, or it can segment parts in a stream, or adding data. In short it can do a lot. But in the end the structured data ends up in the database.
According to Stonebraker a system must fulfil eight requirements to qualify as a real-time streaming processing system. The requirements are discussed in the section bellow to point out that these requirements cant be fulfilled by a immutable database. In short streaming databases are designed for fast processing of fast changing datasets and input-streams.

3.2 Immutable Databases

In contrast immutable databases can be understood as persistent storage with a persistent from the beginning to now understanding of time. In a immutable database time is not a relative concept used to categorise data. It rather is a part of the data itself.

Stonebraker’s 8 requirements of real-time streaming processing

Immutable databases are not compatible with these requirements.
<table>
<thead>
<tr>
<th>Comparison</th>
<th>1. Keep the data moving</th>
<th>In comparison to the streaming databases that operate on fast changing databases (sliding window on a stream), the immutable database works on slowly- or not changing input. The requirement to process messages in-stream can not be met as no in-stream processing takes place in a immutable database.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>2. Query using SQL on Streams</td>
<td>The only representable examples for immutable databases, datomic and mentat, are not using SQL but datalog (a prolog like language for operations on datastructures) for their queries. Therefore this requirement is not met.</td>
</tr>
<tr>
<td></td>
<td>3. Handle stream imperfections</td>
<td>Immutable databases are not processing streams. But even the queries on the dataset are by the usage of datalog forced to complete. There is no such feature for operations to time-out as there is in streaming databases.</td>
</tr>
<tr>
<td></td>
<td>4. Generate Predictable Outcomes</td>
<td>As of the immutability of the database this requirement is always guaranteed to be met as long as the time of the changes is limited by since - to.</td>
</tr>
<tr>
<td></td>
<td>5. Integrate stored and streaming data</td>
<td>Even if immutable databases are capable of doing many of the things like comparing past with present and so on, no streams are integrated in this process.</td>
</tr>
<tr>
<td></td>
<td>6. Guarantee data safety and availability</td>
<td>Even though there is no stream processing, datomic is able to run in high availability mode. This feature is only available in the full pro version.</td>
</tr>
<tr>
<td></td>
<td>7. Partition and Scale Applications automatically</td>
<td>Datomic is scaling by the architecture. For details about the environment see chapter 4.2</td>
</tr>
<tr>
<td></td>
<td>8. Process and respond instantaneously</td>
<td>As the queries are by design terminating, the processing can take a considerable amount of time.</td>
</tr>
</tbody>
</table>
4 Datomic

Minimal data unit in datomic is a so called datom. A datom is an immutable, point-in-time fact: [entityid attribute value transaction added?]

The entityid is the unique id of this entity. Everything that is able to be stored in a datomic database has an entityid. This id is used to perform the implicit joins that datomic applies to execute the queries.

The attributes are facts that can be applied to something. Every attribute has a name like :birthday and a value type to represent the assigned value (:db.type/instant). Names can be qualified by namespaces. Names beginning with :datomic/ are reserved for internals.

The value that is assigned to this attribute. The type must match the value-type of the attribute. The transaction is an id to an entity with an attribute :db/txInstant containing the time when the datom was added to the database.

Claims

On their website they make many big claims about the features of datomic. These features and if they provide what they promise will be discussed in detail below.

Datomic avoids:

- Manual caching and replication, by using a built in automatic caching in the peer library / peer server. Every dataset replication will be done automatically after an update in the dataset took place.
- Complex configuration, by providing configuration file templates for every data backend available. Only minor adjustment have to be done and one is good to go.
- Sharding (automatic or manual). Datomic itself does simply not do that. It can be done by using sharding in the storage backend like pg_shard in PostgreSQL, but it is highly discouraged [4].
- Logging does not seem to be avoided. There is even a manual to setup the logging [3]. So as far as one could deduct from the given information at the time this paper was written, this claim could not be met.
- Locking and Latching are avoided by the architecture. Data changes only slowly. Queries run on a snapshot of the dataset, mistakes can be detected and removed from the up-to-date state of the dataset. The underlying storage backend however is still able to lock and latch.
• Disk management is not actually avoided but rather delegated to the storage backend. This statement is true, but considering that datomic is not supposed to manage disks makes it look rather misleading.

4.1 Query language

In contrast to the classic structured databases like PostgreSQL, MySQL and so on, datomic does not use SQL for queries. Instead they use datalog. All the datalog examples are expressed using the extensible data notation.

4.1.1 Queries

Queries are executed on the peer (peer library / peer server). The dataset that the query operates on is the cached database. Therefore datomic is better suited for slow changing datasets.

4.1.2 What is datalog?

Datalog is a subset of the logical programming language prolog optimised for relations between data structures. Compared to prolog there is one big advantage: datalog is truly declarative. This means datalog programs are always guaranteed to terminate (As long as a finite data-structure was the target of the query) and the order of the statements is irrelevant. This makes it possible to elide the, in prolog well known, cut operator.

The second big difference that one may realise is that the order of the statements is not relevant. I may state this in an example:

The following prolog example will never terminate. This is caused by the way the prolog interpreter tries to solve this program. It first tries to solve descend by the rule that the second argument might be the child of the first one. If this is not true, it will try the first rule. But the first step in the evaluation calls descend with the exact same arguments before it even could evaluate the the next step. This forms an infinite loop.

```
Listing 4.1: Not terminating prolog sample

descend(X,Y) :- descend(Z,Y),
               child(X,Z).

descend(X,Y) :- child(X,Y).
```
In contrast to two datalog examples that will not only terminate but also return the same result.

<table>
<thead>
<tr>
<th>Listing 4.2: Similar rules in datalog</th>
</tr>
</thead>
<tbody>
<tr>
<td>```datalog</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>[[(descends ?X ?Y)</td>
</tr>
<tr>
<td>[descends ?Z ?X]</td>
</tr>
<tr>
<td>[?X :person/child ?Z]]</td>
</tr>
<tr>
<td>[(descends ?X ?Y)</td>
</tr>
<tr>
<td>[?X :person/child ?Y]]</td>
</tr>
<tr>
<td>]</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>; Does the same as</td>
</tr>
<tr>
<td>```datalog</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>[[(descends ?X ?Y)</td>
</tr>
<tr>
<td>[?X :person/child ?Z]</td>
</tr>
<tr>
<td>[descends ?Z ?X]]</td>
</tr>
<tr>
<td>[(descends ?X ?Y)</td>
</tr>
<tr>
<td>[?X :person/child ?Y]]</td>
</tr>
<tr>
<td>]</td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>There are other minor differences that can be disregarded.</td>
</tr>
</tbody>
</table>

## Features

### Bindings

A binding can occur in following forms:

- `?X`  
  **Scalar**
- `[?X ?Y]`  
  **Tuple**
- `[?X ...]`  
  **Collection**
- `[[?X ?Y]]`  
  **Relation**

A **scalar** is a simple variable that can contain a column of data.

A **tuple** represents multiple values that all must match in the same iteration of the query. It forms an “and” statement for all the contained values.

A **collection** binding binds a single variable `?X` to multiple passed variable-values like this `"Arnold Schwarzenegger", "Halle Berry", "Harrison Ford"`). This results in a kind of “or” statement. I could for example ask for all the movies in which one of these actors were casted.
A relation binding combines the tuple binding and the collection binding. It forms a collection of tuples. For example one can pass a relation between external information about earnings related to names. With this information one could calculate which movies were the most expensive in average.

Rules

If there are blocks that are used repeatedly in queries they can be extracted into so called rules. For example this rule that checks if a person is associated with an album or not:

Listing 4.3: Datalog rule example

```
[(associated ?person ?album)
 [?album :albums/artist ?person]]
[(associated ?person ?album)
 [?album :album/producer ?person]]
```

Datalog rules could be compared to prepared statements in SQL. In contrast to rules prepared statements are not integrated in the query language but in the language that operates SQL.

In a where statement the rule can be called like this: `(associated ?person ?album)`. And it does not matter whether one passed a person or an album. It will just resolve the variable that was not passed. If both variables were given, the rule returns true if the condition is met. If no variable was provided, all the possible relations will be output.

Using `?[var]` requests the provided argument being bound when it was provided. If an unbound variable is provided an exception will be thrown.

Per default the rules are passed to the query using the `:in` statement as the parameter named `#`.

4.1.3 Datalog examples

Statement :find

The :find statement marks the columns in the result. The :find statement is also the right place to express aggregates like `sum`, `count`, `min`, `max` and others.
Listing 4.4: :find statement examples

; Breaks the results down by name and counts how often this person borrowed a comic.
:find ?name (count ?noOfBorrowedComics)

; Returns the values in the variable ?attributes
:find ?attributes

These statements translated in SQL would look like:

Listing 4.5: :find statement SQL equivalent

/* Breaks the results down by name and counts how often this person borrowed a comic */
SELECT name, COUNT(noOfBorrowedComics) ...

/* Returns the values in the variable ?attributes */
SELECT attributes ...

Possible aggregates are:

• :find (min ?variable)
• :find (max ?variable)
• :find (count ?variable)
• :find (count-distinct ?variable)
• :find (sum ?variable)
• :find (avg ?variable)
• :find (median ?variable)
• :find (variance ?variable)
• :find (stddev ?variable)

Depending on the type of binding one wants to find the :find statement looks a little bit different:

<table>
<thead>
<tr>
<th>Type</th>
<th>Default result type</th>
<th>:find syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td>Relation</td>
<td>The default result type</td>
<td>:find ?X ?Y</td>
</tr>
<tr>
<td>Collection</td>
<td>For single variable finds</td>
<td>:find [?X ...]</td>
</tr>
<tr>
<td>Tuple</td>
<td>For multiple related values</td>
<td>:find [?day ?month ?year]</td>
</tr>
<tr>
<td>Scalar</td>
<td>For single value variables</td>
<td>:find ?totalEarnings . (note the dot)</td>
</tr>
</tbody>
</table>
Statement :where

This statement is used to specify what we are looking for. All the data-clauses in :where statements are implicitly joined.

A data-clause is a statement in the form of [database entityid attribute value transaction]

The database is implicitly $ this is the default name of the database if there is no explicit :in statement. The database can be explicitly set. This is useful, if a single query is intended to operate on multiple databases.

### Listing 4.6: :where statement examples

; Returns all the entities that have an attribute :movie/title and the attribute :movie/year of these entities.
[:find ?movieTitle ?releaseYear
 :where
 [:movie :movie/title ?movieTitle]
 [:movie :movie/year ?releaseYear]]

; Returns all the movie titles in which a person, named Harrison Ford, acted.
[:find ?movieTitle
 :where
 [:person :person/name "Harrison Ford"]
 [:movie :movie/cast ?person]
 [:movie :movie/title ?movieTitle]]
Listing 4.7: :where statement SQL equivalent

/* Returns all the the movie titles their release years. */
SELECT movieTitle, releaseYear FROM movies;

/* Returns all the movie titles in which a person, named Harrison Ford, acted. */
SELECT M.movieTitle 
FROM movies as M 
INNER JOIN cast as C 
    ON M.id_cast = C.id_cast 
INNER JOIN persons as P 
    ON C.id_person = P.id_person 
WHERE P.name = 'Harrison Ford';

In a data-clause every irrelevant part can be ignored by replacing it by an underscore _. After the last implicitly given argument all the remaining arguments are set to _.

For example if one knows that persons have an attribute :person/name one can simply find out about all the other attributes using following query:

Listing 4.8: Where statement with elided values

[:find ?attributes 
 :where 
 [:person :person/name] 
 [:person ?attributes]]

In an SQL database everything is stored in tables. To check how a entry looks, one should take a look on the table schema.

Statement :in

If we want to reuse the query about in which movies an actor acted then an input can be passed to the query. The first parameter is used for the database $. Further parameters can be used to pass bindings.
Most SQL dialects don’t know variables. Transact-SQL knows a thing called local variables. In SQL one would use prepared statements.

This query can be called in clojure using a statement like `(q query db "Arnold Schwarzenegger")

**predicates**

Predicates are a specialised function that return a boolean. Operators that are implemented into datalog are: =, !=, <, <=, >, and >=.

```clojure
[:find ?movieTitle ?year 
 :in $ ?minYear 
 :where 
 [?movie :movie/year ?year] 
 [?movie :movie/title ?movieTitle] 
 [(< ?year ?minYear)]]
```

```sql
SELECT title year FROM movies 
WHERE year < 1991;
```
functions

A function can be one of the simple operators +, -, *, and / or any clojure function or java method. The used functions must be fully qualified to be resolved correctly.

For example this query that prints the approximate age in years:

```
[:find ?ageInYears
 :in $ ?today ?name
 :where
 [?pers :person/name ?name]
 [?pers :person/born ?born]
 [(?today - ?born) ?age]
 [(?age / (* 1000 60 60 24 365) ?celsius)]
```

SQL supports similar arithmetic operations:

```
SELECT age/(1000*60*60*24*365) AS "Age in Years"
FROM persons;
```

If java functions are used the qualification using dots is replaced by slashes. For example to call a static method to get the current time `(LocalTime/now)` ?now. To call instance methods one would use this syntax: `(getHour ?now) ?hour`. To bind this arguments to the methods datomic uses java reflection to determinate the type of the arguments. The whole execution can be speed up massively if the type of the argument gets specified like that: `(.isAfter \^LocalTime ?now ?otherTime) ?isAfter`

**Time**

The one element that makes all the difference, time. Querying data back in time is one of the big features offered by datomic.

This time-travelling is done using filters. The available filters are:

- asOf: Includes everything up to a specific point in time.
- since: Does the exact opposite of asOf.
- history: Includes the whole history and all previous states.
missing?

To detect if an attribute is not set or was retracted one can use the \( \text{missed?} \) \$ \ ?hansZimmer :person/born \) function. If born is not set on the variable, the statement will return true.

### 4.1.4 Adding entries to database

Adding entries to a database is rather simple. For example using the datomic-shell:

```
(def db (client/db conn))

(datomic.api/transact conn
  [[:db/add "ac" :person/name "Mel Gibson"]
   [:db/add "ac" :person/born "-441788719"]])
```
4.2 Environment

A datomic environment consists of some individual components.

4.2.1 Peer Library (Application Server)

Communicates with the transactor. It sends transactions and receives the resulting changes from the transactor. Another feature is the internal caching mechanism. Received changes update the cache. The peer supports queries on the cache.

For small projects the peer library can work in standalone mode. Instead of a transactor it then uses an in-memory database.

At the moment only JVM based languages are supported. For other languages the client library is available.

An example of an architecture with peer library can be seen in figure 4.1.

4.2.2 Peer Server (Standalone JVM Server)

The peer server does the same as the peer library but is a standalone process. It is needed as the communication counterpart for clients using the client library.

An example of an architecture with peer server can be seen in figure 4.2.
4.2.3 Transactor (Database Server)

The transactor is in charge of the connection to the storage service. It accepts and executes transactions and promotes back the resulting changes. (Uses storage service to save data)

4.2.4 Client Library

The client library is a lightweight implementation of the peer library designed for the use in microservices. Features like caching and query support is implemented in the peer server. To use these features the library needs to communicate with the peer server. In previous versions the communication was done via a REST API, in current versions the REST API (now considered legacy) was replaced by the datomic clojure client API which is at the moment considered alpha state. The library itself is also implemented in clojure.

According to the datomic documentation, the protocol was designed to support non-JVM languages too. After some reverse engineering it turns out that the clojure client library also makes use of a not jet official http API. Official implementations in other languages are to be expected.

4.2.5 Shell

This java-shell allows to interface with the datomic java-api. Creating or deleting a database is as simple as shown in the example below:
Listing 4.18: Creating / deleting a database

```java
// Create
String uri = "datomic:sql://datomic?jdbc:postgresql://" +
    "localhost:5432/datomic?user=datomic";
Peer.createDatabase(uri);
Connection conn = Peer.connect(uri);

// Delete
Peer.deleteDatabase(uri);
```

4.2.6 Console

The console allows to connect to a transactor and then run queries on the dataset, inspect the schema or add/remove attributes.

To attach the console to a transactor with sql backend, the connection URI must be passed including the jdbc string.

If the connection URI is copied from the transactor output, the `<DB-NAME>` tag must be removed. It must not be replaced by the actual database name.

4.3 Backends

In the pro version datomic supports many different backends.

- Dev mode (dev)
- SQL database (sql)
- DynamoDB (ddb)
- Cassandra (cass)

Dev mode

In dev mode no real backend is used. Instead the transactor creates files on the local persistent storage device. This type of storage does not require explicit configuration.
**SQL**

Using the java database connectivity (JDBC) driver a connection to a SQL database can be established. Datomic provides scripts to setup a database, tables and users for PostgreSQL, MySQL and Oracle. The PostgreSQL JDBC driver is already included in the provided transactor delivery. Drivers for other databases must be included into the java / clojure project.

Altering tables using other means than the transactor is highly discouraged and might result in inconsistent data.

**DynamoDB**

DynamoDB is an Amazon Web Service offering a database as a service. This backend was not tested by the author during this seminar.

**Cassandra**

If a cluster, featuring at least three nodes, is available, Apache Cassandra can be used as a memory backend. This backend was not tested by the author during this seminar.

**4.4 CRUD / ARAR**

In contrast to classic relational databases, which are know for its four main operations (Create, Read, Update, Delete), datomic uses ARAR (Assert, Read, Accumulate, Retract).

An assertion in datomic is a fixed date-in-time fact that is represented by one datom. Instead of updating these assertions with the latest facts, new assertions are created representing the latest facts at the current time. All these assertions are accumulated.

Rather than deleting one retracts. A retraction does not delete any assertions. What it does can be described to taking a snapshot of how it was before and setting it as latest state. Mistakes that were made will not be forgotten. This can be helpful if decisions were made using the wrong data. Tracking these mistakes down and correcting them will be a lot easier that way.

If a record needs to be purged from the database (for example because a customer deleted its account and privacy requires to delete personal data) there is the excise function. After this function it is as if the user never existed.
List of Code-samples

4.1 Not terminating prolog sample .................................................. 8
4.2 Similar rules in datalog ........................................................... 9
4.3 Datalog rule example ............................................................... 10
4.4 :find statement examples ........................................................ 11
4.5 :find statement SQL equivalent ................................................ 11
4.6 :where statement examples ...................................................... 12
4.7 :where statement SQL equivalent ............................................ 13
4.8 Where statement with elided values ........................................ 13
4.9 In statement ............................................................................ 14
4.10 In statement ........................................................................... 14
4.11 :in statement SQL equivalent .................................................. 14
4.12 Conversion function ................................................................. 15
4.13 Conversion function ................................................................. 15
4.14 Time-travelling asOf ............................................................... 16
4.15 Time-travelling since ............................................................... 16
4.16 Time-travelling history ............................................................. 16
4.17 Adding entries to database ....................................................... 16
4.18 Creating / deleting a database .................................................. 19
Bibliography


