Modern Databases

Leen Torenvliet Stephan van Schaik Robin de Vries

February 12, 2015

Contents

Chapter 1

Introduction

In this course about modern databases will focus on some non-SQL databases. The text is tailored to a seven weeks course on databases, that consists for roughly 50% of theory and a project of about three weeks. In the theory section, we start out with an SQL language. In particular we treat the probably oldest still in use SQL language PostgreSQL (Chapter 2, which is easy to download, readily available, easy to install and very stable. With the relational model in mind, we treat subjects as transaction management, serializability and locking protocols. These notions play an important roole in distributed environments where not only the queries must be treated more or less in parallel, but also the database systems are distributed over many nodes. Using the relational model for introduction of these notions however, gives us a more relaxed way of dealing with them. Moreover subsequent chapters need to deal with a host of different subjects that will claim our attention.

In the old days there was only one model of databases available, the relational model. SQL (standard query language) type databases are still very popular and widely used. Their long history of theory and practice gives them the stability, reliability and speed needed in critical applications. A relational model is also very apt to handling parallelization of transactions. Modern locking stratgegies allow for handling batches of transactioins offered more or less simultaneously in a serializable fashion (i.e., the result in the database is as if these transactions were handled in parallel). There is however also an inherent non-parallel, or rather non-distributed property in relational databases, and that is the storage of data. Large tables of data on which joins must be performed, a very important action in the relational model, cannot be easily or intuitively be distributed over a large number of machines. Joining two tables that are stored distributed over machines that have a large geographical distance is a pain. Joining two tables on different machines requires huge amounts of data to be transported. Therefore, different concepts were developed. In this course we will make acquaintance with some representatives of these models. As with many elaborate concepts, a categorization can be made in very many different ways and along many different dimension. We choose a categorization that is most common.

1. Relational. These are of the SQL type. Think of storing data in tables with entities as rows and properties (attributes) as columns. Tables can also be relations between tables. Therefore the model is also known as the entity-attribute-relationship (EAR) model.
   Examples: Postgres, MySQL, Oracle.

2. Graph. If relations between data are important (i.e., there are many connections), we

could decide to represent the data as a graph. Distributed maintenance of data can easily be realized by letting each node in the network of machines represent a subset of the nodes of the graph.
Examples: Neo4J, Graphbase, InfiniteGraph.

3. Document. Document databases store hierarchical values called documents. They support distributed storage by storing sets of documents on different machines. Examples: CouchDB, MongoDB, Couchbase.

4. Columnar: This form of database scales along multiple servers, and groups similar data into column families. Column values can be individually versioned and managed, though families are defined in advance. This model shares the most properties with the relational model, though the orientation is perpendicular. It was made popular by Google's "Big Table." Examples: HBase, Cassandra, BigTable

5. Key-value: This model is perhaps the simplest form of modern databases. Basically the database is modeled just as a big (distributed) hash table. Values are stored in buckets defined by keys. A value can be literally anything. Examples: Riak, Redis, Voldemort.

From the range of modern databases nowadays available, we pick three examples as representatives of discernible categories in database language, the key-value databases, the document databases, and the graph databases. In particular in Chapter 3, we will look at CouchDB as an example of document oriented databases in Chapter 4, we will look at Neo4J as an example of graph databases databases and in Chapter 5, we will look a bit at the key-value database system RIAK, which will also sort of model the system that we will set up in the project belonging to this course. The setup of this document is a bit like the "seven databases" book by Redmond and Wilson [**?**]. We will have, however, a less extensive selection of databases than do Redmond and Wilson, to make room and time for the project.

All database systems must perform the basic Create, Read, Update and Delete (CRUD) actions necessary to maintain a large data set. In this text we will study how in modern database management systems (DBMS) these tasks are often fulfilled in a distributed environment.

Finally the project, defined and set up in the last chapter, is an implementation of a key-value type database. We use RIAK sort of as a coat rack for our project. The students will implement a working example of a key-value database in a distributed environment of Linux type machines.

The documentation that backs this text is gathered from many places and not always correctly referenced, or referenced at all. We summarized chapters from database books like [**?**] and copied examples without altering them. As the objective of this text is purely non-commercial we feel free to do so. Students who feel the need to gather more background material are urged to consult (and buy) these original texts. We keep a list of all texts used in the bibliography.

Chapter 2

PostgreSQL

We will assume the reader to have at least some knowledge of an sql-type database language. In this first chapter on modern databases, we will in fact discuss features of an old sql-language, which is nonetheless still widely used. In this chapter, we will discuss the query language PostgreSQL, with a particular interest on general aspects of transaction management like serializability, locking and consistency. These features are of interest of course also in later chapters.

A relational database consists of a collection of tables. Most tables consists of rows and columns, where the rows in the table represent *entities* for which data is maintained. These data consist of *attributes* which are the columns of the table. Each attribute has *values* picked from a certain *domain*. For every table with entities as rows, there is a (combination of) attribute(s) that identifies the entity uniquely. Such a (combination of) attribute(s) can be used as a *key* to the entity for the table. Preferably, a key consists of a single attribute but this is not always the case. Keys can be used to build an *index structure* on the table(s). Very popular in classical databases are the so-called B+trees, which are always perfectly balanced trees of depth logarithmic in the number of keys stored in the databases. Another popular access mechanism is through so-called *hash tables*. In particular linear and extendable hashing functions are much used in classical databases. In more modern versions of databases, the hash table is often stored and maintained over many nodes. These *distributed hash tables* form an important subject in this text.

Some tables are mere relations between other tables. These relation tables contain multiple columns, each of which have key values from other tables. This is done mostly to help save space. Combining all attributes in a single table might force the administration to enter lots of null values, or duplicating lots of data. Both mean a waste of space, could lead to integrity problems (when data updates are not performed at all necessary places) and certainly leads to big tables and therefore to performance issues. Avoiding these issues leads to so-called Normal Forms in traditional databases, and is a big issue there. We speak of a functional dependency between attributes if the value of one attribute can be computed from (a set of) other values. In particular, a key must give a unique value to a lot of other values. The purpose of bringing the database into a normal form is to limit the different forms of functional dependencies to a maintainable minimum, preventing inconsistencies.

2.1 Queries in PostgreSQL

Before we can play with PostgreSQL, we need to install a database. This text assumes that an environment is given that can respond to SQL queries. If not, then installations of PostgreSQL are readily available from various sources and for various operating systems, most notably from pgfoundry.org/projects/dbsamples. Installing a PostgreSQL database is also almost trivial as a database in an SQL language can usually be stored *in* that language. In an environment that understands PostgreSQL, a database can be created using the createdb command. Next, a file containing an SQL database can be loaded. Typically such a file first sets up a number of tables, and uses a number of INSERTs to fill the tables with data. Examples of such files can be downloaded from many places. For our purposes a simple database, like the world database (see e.g., pgfoundry.org/projects/dbsamples) will suffice.

Download this database and install PostgreSQL, if it is not already on your system. A simple `sudo apt-get` wil often do the trick.

There are many intricacies connected to running a PostgreSQL database that will allow or forbid users connecting to databases from various stations. This is not our present concern. To avoid all this, we can get administrator access by posing as the database administrator postgres through the command `sudo su postgres`.

Now we can issue the command `createdb` world and thus have an empty postgress database. by typing `psql world`, we enter the world of postgres and can start populating and querying this database.

The file world.sql can be used for this purpose by typing

`\i world.sql`

note: the `psql world` command will always give you access to the world database, once you are the user postgresql.

Now your database has several tables. For instance, it has a table called city. Try and inspect the contents of this table by typing

```
SELECT * FROM city;
```

Don't forget the ";"!

Now, let's have fun and draw some information from the database. The city table and the country table were separated deliberately to make it possible to store information more efficiently. We can regain access to the information scattered over the tables by "join"ing them together and then select the information searched for from this virtual larger table.
Try:

```
select city.name, country.name, city.population from city inner join country on
city.id = country.capital;
```

and see what happens.

PostgreSQL has many, if not all the query possibilities that third year students already know from other sql languages like, e.g., mysql. We will practice some more with these in the exercises section at the end of this chapter.

One type of queries in particular deserves our attention, the `WITH` query. Look at a query like the following.

```
WITH regional_sales AS (
     SELECT region, SUM(amount) AS total_sales
     FROM orders
     GROUP BY region
```

```
),   top_regions AS (
     SELECT region
     FROM regional_sales
     WHERE total_sales > (SELECT SUM(total_sales)/10 FROM regional_sales)
)
SELECT region,
     product,
     sum(quantity) AS product_units,
     SUM(amount) AS product_sales
FROM orders
WHERE region IN (SELECT region FROM top_regions)
GROUP BY region, product;
```

taken from the internet, which displays per-product sales totals in only the top sales regions. This example could have been written without `WITH`, but then we would need two levels of nested sub-SELECTS. It's a bit easier this way.

`WITH` has, however, another, more complicated form, the *recursive* with. Consider the following query.

```
ll WITHRECURSIVE t(n) AS (
       VALUES(1)
       UNION ALL
       SELECT n+1 FROM t WHERE n < 100
)
SELECT sum(n) from t;
```

There is, or need not be, a table `t` in the database. What `t` looks like is dealt with in the recursive part of the query. A recursive `WITH` starts with a non-recursive term, then a `UNION` (or `UNION ALL`) then a recursive term, where only the recursive term can contain a reference tot the query's own input. Such a query is executed as follows.

1. Evaluate the non-recursive term. For UNION (but not for UNION ALL), discard duplicate rows. Include all remaining rows in the result of the recursive query, and place them in a temporary working table.

2. So long as the working table is not empty, repeat these steps.

   (a) Evaluate the recursive term, substituting the current contents of the working table for the recursive self-reference. For UNION (but not UNION ALL) discard duplicate rows and rows that duplicate any previous result row. Include all remaining rows in the result of the recursive query, and also place them in a temporary intermediate table.

   (b) Replace the contents of the working table with the contents of the intermediate table.

In the example above, the working table has just a single row in each step, and it takes on the values from 1 through 100 in successive steps. In the 100th step, there is no output, because of the WHERE clause and so the query terminates.

2.2 Transaction Management

A *Transaction* is a set of operations that form some logical unit. For instance we want to transfer a sum of money from one account to another account. This means that one account must be checked to have enough money in it to pay the amount, unless some credit has been agreed upon, in which case it must be checked that the credit limit won't be violated by the transfer and a date must be set for interest accumulation. It has to be checked whether the amount to be transferred does not supercede the limits set by the bank. It has to be checked whether the command for the transfer stems from a legitimate source, which may require extra communication and so on and so on.

In normal database management systems it is also true that many (millions) of transactions are seemingly going on practically at the same time, and many of them might want to read from and write to the same places in the database. Finally, though modern databases seem less and less vulnerable, an error might occur whether accidentally or deliberately when the system is attacked.

A transaction must have certain properties for a database to be workable. From the outside it must seem that a transaction is an *atomic* concept. That is, it must seem that all of the action is done in one go uninterrupted. Furthermore, though transactions take time and the data in the database might change while the transaction is going on, the result of a single transaction must be the same as it would have been if the transaction had been applied to the database in *isolation* (i.e., on a static database). Of course a transaction may not jeopardize any consistency constraints of the database (i.e., it must be *consistent* and finally, if a transaction succeeds in completion its actions can only be deliberately be undone (i.e., it must be *durable*)

Atomicity, Consistency, Isolation, and Durability are often called the ACID properties of databases, though the negative connotation that acid has on robust structures has not contributed to this name giving.

In our money transfer example above, it is clear that there are many points in such a transaction in which in particular consistency is violated. When moving an amount A form account B to account C, there is some time between the moment that A is deducted from B and added to C. In this period there is actually less money in the database than there should be. Another transaction that drives by and wants to either charge account B or account C may find this impossible for insufficient funds, but it gets worse. Transaction T_1 may read the account B and conclude that there is enough money there to transfer A to C, decides to subtract A from B and transfer the money, but before T_1 can write the new amount to B, transaction T_2 might drive by and read the, meanwhile old, value of the account, decide that there is enough money to transfer to D and go ahead. Then, T_1 writes the new value, T_2 writes another new value, and in fact, money has been *created*. It is clear that most database systems would not look favorably on such a transaction management system.

An easy solution to this problem would be to perform all transactions one after the other, but that would mean terrible performance. Transaction management systems (TMS) therefore are looking for ways to perform transactions *as if* they were performed in sequence.

One might also impose the restriction on the transaction management system that a transaction may not make use of the resources that another transaction wants to use unless that other transaction is completed. This however presents another problem. A transaction may claim access to a resource, perform some actions there and then die. A locking system as proposed would imply that other transactions may *never* be able to access the resource.

Therefore transactions need to be monitored, timed out, and rolled back if they take too much time. Generally, a transaction can be in the states *Active*, *Partially committed*, *Failed*, *Aborted* or *Committed*, where each of these states (except maybe for the last) signals to the TMS which action to take next. Once a transaction is committed the TMS must ensure durability by writing the updates to disk.

2.2.1 Atomicity and Durability

There are several ways in which atomicity and durability can be guaranteed. Most database books have extensive chapters on this. We give only a very brief overview here. Under normal circumstances when transactions are not performed concurrently both atomicity and durability are not a problem. Only when abnormal circumstances occur (e.g., system crash) these properties are in jeopardy. There are basically two ways to protect the system against data loss. One, very inefficient, method is to maintain a shadow copy of the database and update a database pointer, making the shadow copy the actual database, after each commit of a transaction. The other method requires maintaining a log of the transactions. This log can then be used to either make the changes to the database only after a commit has taken place or undo and redo certain parts of transactions after failure during a write to the database. These two cases are dubbed *deferred database modification* and *immediate database modification* respectively.

Since processing the entire log could become rather costly, it is also customary to write so-called checkpoints into the log files. These represents points in time beyond which a roll-back must not occur. At the time of a checkpoint all data is written to non-volatile memory.

Most schemes consider non-volatile memory as *stable* memory (i.e., cannot be changed by system crashes by definition). This is of course not true. Non-volatile memory should therefore be backed up on a regular basis as well. This is however further beyond immediate atomicity and durability requirements.

2.2.2 Concurrent Executions

The world would be so much easier when transactions could just be put in a queue and then executed one after another whenever the DBMS felt like it. This is however a bit problematic even when making reservations for a single airplane with 300+ seats. The last customer would have to wait like what seemed forever before her reservations could be confirmed. Therefore most DBMS's handle their transactions "concurrently," which, more often than not, means interleaved, but sometimes real parallelism and even distributed computing is involved. This will become even more important in later sections, where we discuss large databases, distributed over multiple nodes.

Concurrent execution of transactions brings about problems especially when the transaction consists of multiple actions that can interfere with each other (i.e., want to use the same resource, write into the same memory location or even have one read from a location, while the other writes into that location). It is the job of the system's *concurrency control mechanism* to ensure that transactions are performed "as if" they were performed serially (i.e., leaving the database in the same state after all transactions are finished).

Let us look at a very standard example of two transactions T_1 and T_2 where one transaction tries to transfer an amount of \$50 from account A to account B and transaction T_2 tries

to tax A, that is tries to move 10% from A's account to account B. This might look as follows.

T_1:	read(A)
	$A = A - 50$
	write(A)
read(B)	$B := B + 50$
	write(B).

T_2:	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
	read(B)
	$B := B + temp$
	write(B).

Now suppose the two transactions are separately, but concurrently handled by the DBMS. It could be in the order $T1; T2$ or $T2; T1$, in which case there is no problem of consistency (though both A and B might have preferences), but it can also be performed in an interleaved manner like the following.

T_1	T_2
read(A)	
$A := A - 50$	
write(A)	
	read(A)
	$temp := A * 0.1$
	$A := A - temp$
	write(A)
read(B)	
$B := B + 50$	
write(B)	
	read(B)
	$B := B + temp$
	write(B)

Simple calculation shows that in this case the total amount of money in the database is still the same as in the sequential execution, so the database is still consistent, but the astute reader will notice that this might not always be the case as the following schedule shows.

T_1	T_2
read(A)	
$A := A - 50$	
	read(A)
	$A := A - temp$
	$temp := A * 0.1$
	write(A)
	read(B)
write(A)	
read(B)	
$B := B + 50$	
write(B)	
	$B := B + temp$
	write(B)

T_1	T_2
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

T_1	T_2
read(A)	
write(A)	
	read(A)
read(B)	
	write(A)
write(B)	
	read(B)
	write(B)

T_1	T_2
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Figure 2.1: Serializing a Schedule

2.2.3 Serializability

If we let transactions be performed concurrently, then any order in which the transactions are handled by the database should leave the database in a legal consistent state. The best thing that we can do, if we allow transactions to be handled in an interleaved manner, is to demand that the state of the database after successful completion of these transactions is *as if* the transactions are done sequentially. Some schedules guarantee this, others don't. We call schedules of the former kind *serializable*. An action causes an inconsistency if performed according to a schedule if it would leave the database in a state that cannot be reached were the transactions be performed in any sequential manner. This can be read actions that produce values that could not be in the database, or write values that produce wrong values in the database, or a combination of these. Since only read and write actions can cause these inconsistencies, we concentrate on these and leave all other actions out. Furthermore, read and write actions that are performed on separate resources cannot give rise to inconsistencies, no matter which order they are performed in. Therefore we concentrate on read and write actions that are performed on a single resource. Serializing the schedule is done by swapping actions in the schedule such that conflicts are resolved. A schedule that can thus be fixed is called *serializable*. Suppose that some resource Q is subject to read and write instructions in T_1 and T_2. If only read(Q) is involved, then there is no problem, but if read(Q) and write(Q) are involved or even if both transactions try to write(Q) then the contents of the database after completion might depend on the order in which these instructions occur and we cannot blindly swap these instructions in the schedule. Figure 2.1 shows three phases of consecutively rescheduling actions in a schema. The first table shows the original schedule, the second shows the result of swapping two instructions pertaining to different resources, and the final table shows the result of performing all non-conflicting swaps, resulting in a serial schedule.

Because these swaps are possible and leave an equivalent state, we call the original schedule realizable.

There are two basic forms of serializability, *conflict* serializability and *view* serializability.

Conflict Serializability

We say that two transactions T_1 and T_2 are in conflict, if they contain operations on the same data item and at least one of these instructions is a write operation. If a schedule S can be rearranged into a serial order S' by swapping instructions without changing read and write

T_1	T_2	T_3
read(Q)		
	write(Q)	
write(Q)		
		write(Q)

Figure 2.2: a view serializable schedule

instructions that could leave the database in a different state than would be reached after performing S, then we call S conflict serializable.

View Serializability

View serializability is just a bit more complicated than is conflict serializability, since it gives a little bit more freedom. Every conflict serializable schedule is also view serializable, but not every view serializable schedule is also conflict serializable. Two schedules S and S' are said to be *view* equivalent, if the following conditions are met.

1. Every transaction that reads an initial value of a data item in S, must also read an initial value of that data item in S'.
2. Every read instruction in every Transaction reads a value in S' that is produced by the same write transaction as it was produced by in schedule S.
3. If a transaction performs the final write operation on some data item in schedule S, then that same transaction performs the final write instruction on that same data item in schedule S'.

Serializing actions are taken by swapping instructions in a schedule as was done in the conflict serializing game. If none of the above conditions is violated, and S' is a serial schedule, then S is said to be view serializable.

Some transactions write to a data item without ever having read it. These writes are called *blind* writes. Blind writes occur in every schedule that is view serializable, but not conflict serializable.

2.2.4 Recoverability

So far, we have only discussed the concurrent handling of transactions that came to a good end. In reality however, many transactions will fail for all kinds of reasons. Suppose that transactions T_1 and T_2 want make use of the same resource Q, and that T_1 writes a value that T_2 subsequently reads. Now if T_2 should commit *before* T_1 finishes, it could be that T_1 fails, and T_2 is allowed to work with a value *that should never have been in the database*. It may be a bad idea to have transactions read values from a non-committed transaction.

As long as T_1 has not committed the value of Q, it is OK to let T_2 run, but should T_1 fail, a rollback not only of T_1, but also of T_2 is in order.

Cascading Schedules

A situation as sketched above could not only happen for a pair of transactions, but other transactions could be involved as well. Worse, a transaction T_1 could be involved in a write

which is used by T_2, which writes a value that is used by T_3,..., which writes a value that is used by T_n, *after which* T_n writes a value that is used by T_1. A value of *any* of these transactions involves a roll back of *all* transactions. Such a rollback is called a *cascading* rollback. Needless to say this situation is undesirable. It is therefore good to strive for a situation where schedules are *cascade-less*. This situation can be achieved when for each pair of iterations T_i and T_j such that T_j reads a data item previously written by T_i, the commit operation of T_i appears before the read operation of T_j. Note that this can also resolve the cyclic problem mentioned above, as long as transactions are allowed to do partial or intermediate commits.

2.2.5 Isolation

last in this series of ACID properties is isolation. Handling of a transaction by a DBMS should be performed "as if" no other transactions are present. We've seen that this can be done nicely if no other transactions interfere with read and write operations done by this transaction. If transactions would be handled one after another, that would be the perfect situation. However, performance of the database system would soon leave it on the shelf never to be used again. If transactions would only use different resources then there would be no problem letting them run concurrently. Unfortunately, in most situations, transactions *do* use the same resources, most notably in situations in which databases operate.

Now what? Well, give a transaction exclusive access to a data item for as long as it needs that data item. In database terms: give a transaction a "lock" on the resource. Just doing this might leave the database with a performance just a little better than the situation sketched above, therefore this is not very desirable. Nonetheless, methods used in practice are basically all locking methods albeit that they are a little more clever than just giving a transaction exclusive access. We will discuss some of these methods in section 2.2.7 since some of the algorithms used very much resemble the algorithms presented in the next section.

2.2.6 Serializability Tests

We have established in the section above conditions under which a schedule is either conflict or view serializable. In both cases the schedule must be equivalent to a serial schedule. The question is now of course: how do we determine if this is the case and to which serial schedule(s) is our schedule equivalent. For conflict serializability this question is easily answered, since conditions are quite strict, for view serializability is much harder. In fact, Papadimitriou [**?**] showed that this problem is NP-complete, so that in practice we only know exhaustive search algorithms for this problem.

As said, for conflict serializability the situation is considerably easier. If two transactions read from and/or write to the same data item Q, then they must appear in the same order in a serial schedule if either one action is read and the other is write, or both are write. So, we can translate the transactions to nodes in a digraph, and the actions to edges in a graph, and a transaction schedule is serializable if and only if we can apply topological sorting on this graph (i.e., if the graph is cycle free).

2.2.7 Locking Strategies

As observed before, we cannot *really* allow a database to perform concurrent reads and writes on the same data item. Worse even than in more general concurrency problems, where only

the consistency of the data item itself is considered, concurrent transactions must maintain consistency of the entire database. Therefore other transactions must be forbidden to change the value of a data item not only while it is read, but the whole time that its value is involved in computations until it is written back to the database. Locking data items is the only thing the database community has come up with so far, and in fact perhaps it is the only thing possible.

There are however some variations on the theme. We could note that reading a data item when another transaction is busy doing it can cause no harm *unless* the other transaction intends to write a new value to it. Therefore, two transactions that only wish to read an item can share the value of that item, as long as this is known by the database. This type of "lock" is called a *shared* lock (S). Writing to a data item requires that no other transaction can read *or* write that data item for as long as is needed by the transaction. This type of lock is called an *exclusive* lock (X).

Locking is realized in DBMS's by the operation of a *concurrency-control manager*, also called a *locking* manager, most often on a messaging base. That is, transactions send messages to the manager *requesting* locks, after which the manager *grants* locks. The transaction takes no action on the data item on which a lock is requested (i.e., it *waits*) until the lock is granted. The transaction then keeps the lock until such time as it is done with the data item, and it subsequently *releases* the lock.

The control manager only grants a lock to a transaction for a given data item if there are no *incompatible* locks outstanding on that data item. For instance read locks, if shared, are compatible. A write lock is always exclusive and never compatible with any other lock. If an incompatible lock request is received, it is not granted until all incompatible locks previously granted are unlocked.

Of course, there are situations in which a transaction waits forever for a lock to be granted. It is then said to be *starved*. In other cases two or more transactions wait for *each other* to release a lock before they can continue. Such a situation is called *deadlock*. Both situations are undesirable, and need resolving. Most often this implies rolling back one or more transactions.

In Figure 2.3, we see two transactions and the concurrency-control manager in action. Although in this schedule the database is left consistent after completion of the two transactions, we can observe that T_2 displays an incorrect value, because the unlock comes before the transfer of the amount to A.

Figure 2.4 gives an example in which deadlock occurs. A lock is requested and granted on item B, after which another transaction requests a lock on item A, which is granted and subsequently a lock on item B which is held, because it is not yet released. Now if T_1 requests a lock on item A, it is put on hold. However T_1 cannot release the lock on B now, which is needed for T_2 to release the lock on A.

Two Phase Locking

The problem of inconsistency due to an early release of the lock is overcome by the two-phase locking protocol (TLP). This name might sound more impressive than the protocol actually is. It is just meant to be a separation of requesting and releasing the lock. All transactions are (re)organized in the schedule so that two phases can be discerned, the *growing* phase and the *shrinking* phase. During the growing phase, a transaction may request a lock but not release any lock, and during the shrinking phase, a transaction my release a lock, but not obtain any new locks. The point where a transaction has obtained its final lock is called the

T_1	T_2	concurrency-control manager
lock-X(B)		
		grant-X(B,T_1)
read(B)		
$B := B - 50$		
write(B)		
unlock(B)		
	lock-S(A)	
		grant-S(A,T_2)
	read(A)	
	unlock(A)	
	lock-S(B)	
		grant-S(B,T_2)
	read(B)	
	unlock(B)	
	display($A + B$)	
lock-X(A)		
		grant-X(A,T_2)
read(A)		
$A := A + 50$		
write(A)		
unlock(A)		

Figure 2.3: A locking scheme

T_1	T_2
lock-X(B)	
read(B)	
$B := B - 50$	
write(B)	
	lock-S(A)
	read(A)
	lock-S(B)
lock-X(A)	

Figure 2.4: deadlock

lock point. If we order transactions by their lock points we can show that this order is in fact a conflict serialization of the schedule.

Two phase locking unfortunately *does not* prevent deadlock. In fact, the transactions in Figure 2.4 are two phased (they don't release locks in the part of the schedule shown). Two-phase locking does not prohibit cascading schedules either. Therefore, just demanding TLP organization of transactions is not considered satisfactory. Most modern databases that use locking, work with either *strict* TLP or *rigorous* TLP. In strict TLP the extra demand is that all exclusive locks are held until the transaction commits, preventing any other transaction from even reading the data. In rigorous TLP even *all* locks are held until the transaction commits. Rigorous TLP gives serializability in the order in which the transaction commits. Needless to say that both protocols influence the performance of the DBMS.

Upgrading and Downgrading

Consider transactions T_1 and T_2, where T_1 executes a series of reads on data items $A_1, \ldots, A_n$, at the end of which T_1 writes A_1 and T_2 reads A_1 and A_2 and displays $A_1 + A_2$. Since a write is performed by T_1, it must request an exclusive lock on A_1 which results in the wait of T_2 until such time the lock is released (i.e., not before n steps). However, it doesn't *need* the exclusive lock until this write is performed to be sure that no other transaction is reading A_1 at that point. Some databases therefore allow *lock conversions* meaning that shared locks can be upgraded to become exclusive locks, whereas exclusive locks can be downgraded to become shared locks. Since upgrading is a mild form of locking, and downgrading is a mild form of releasing, DBMS's that implement TLP only allow upgrading in the growing phase, and downgrading in the shrinking phase. Strict and rigorous TLP implementations only allow upgrading.

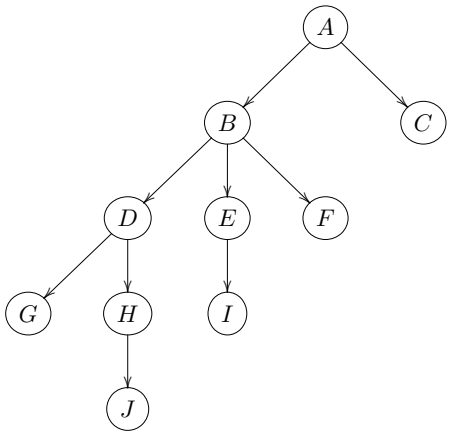
An easy and widely used scheme can be used to add locking requests to schedules that have only read and write instructions.

- If a transaction T_i issues a read(Q) operation, the system issues a lock-S(Q) instruction followed by the read(Q) instruction.
- If a transaction T_i issues a write(Q) operation, the system checks whether T_i already holds a shared lock on Q. If so, then the system issues an upgrade(Q) instruction followed by the write(Q) instruction, if not, the system issues a lock-X(Q) instruction, followed by the write instruction.
- All locks obtained by a transaction are unlocked only after a commit or abort.

This scheme also ensures rigorous TLP.

Graph Based Protocols

We treat one other locking strategy in this section that allows for non-two-phase locking of the items in the database, the so-called *tree protocol* (TP). TP requires prior knowledge of the order in which the data elements in the database are accessed, the so called *database graph*, which is a partial order on $D = \{d_1, \ldots, d_h\}$ where $d_i \mapsto d_j$ implies that any transaction accessing both d_i and d_j must access d_i before accessing d_j. In TP only the lock-X instruction is allowed and any T_i can lock a data item only once, according to the following rules.



T_1	T_2	T_3	T_4
lock-x(B)			
	lock-x(D)		
	lock-x(H)		
	unlock(D)		
lock-x(E)			
lock-x(D)			
unlock(B)			
unlock(E)			
		lock-x(B)	
		lock-x(E)	
	unlock(H)		
lock-x(G)			
unlock(D)			
			lock-x(D)
			lock-x(H)
			unlock(D)
			unlock(H)
		unlock(E)	
		unlock(B)	
unlock(G)			

Figure 2.5: A tree schedule

1. The first lock by T_i may be on any data item.
2. Subsequently, a data item Q can be locked by T_i only if the parent of Q is currently locked by T_i.
3. Data items may be unlocked at any time.
4. A data item that has been locked and unlocked by T_i cannot be re-locked by T_i.

All schedules that are legal under the tree protocol are conflict serializable.
As an example, the schedule of Figure 2.5 gives a tree and a serializable schedule.

Locking in a distributed environment; Brewer's CAP theorem

Practical Tradeoffs When working in a distributed envrionment, we cannot always enforce write consistency. In a classical RDBMS writes are committed and an elaborate locking mechanism maintains that reads always return values that are written last as if all reads and writes happen in serial order. We discussed this extensively. In a distributed system, the data is divided between nodes, and redundancy is kept (i.e., data is written to more than one node) to take insurance against the failure of one or more nodes. We cannot write new data to all nodes simultaneously because of relativity theory, and we cannot lock all nodes until a data element is written to all relevant nodes each time, since that would seriously hinder the availability of the data. Enter Brewer's CAP theorem. C stands for consistency, A stands for Availability and P stands for partition tolerant (i.e., new nodes can be activated and share the data). Brewer's CAP theorem simply says: "Choose two."

When data is shared on an uneven number of nodes, conflicts are often resolved by taking a majority vote, a "last write wins" strategy, then writes have to be time stamped and care must be taken that differences are true and significant, or vector clocks. We will come to speak about vector clocks extensively later in this document.

ACID and BASE As we've seen above, RDBMSs pay great attention to ACID transactions. Since in distributed environments writing to multiple nodes makes this impossible, or at least very uncomfortable, this requirement is changed to the "BASE" requirement, meaning that the database is Basically Available, Soft State and Eventually consistent. We can always access the data, though the data read is not necessarily coming from the last write, and the database will be consistent when enough time has passed to synchronize all nodes. Of course, as we have seen, there may be situations in which inconsistent data is simply not acceptable. In this case, the distributed system also provides some kind of locking mechanism, but this will have an impact on both performance and availability.

2.3 Exercises

1. For each of the schedules of transactions T_1,T_2 and T_3 below

   (a) $r_1(A)$; $r_2(B)$; $r_3(C)$; $w_1(B)$; $w_2(C)$; $w_3(A)$;
   (b) $r_1(A)$; $r_2(B)$; $r_3(C)$; $r_1(B)$; $r_2(C)$; $r_3(D)$; $w_1(C)$; $w_2(D)$; $w_3(E)$;
   (c) $r_1(A)$; $r_2(B)$; $r_3(C)$; $r_1(B)$; $r_2(C)$; $r_3(A)$; $w_1(A)$; $w_2(B)$; $w_3(C)$;
   (d) $r_1(A)$; $r_2(B)$; $r_3(C)$; $w_1(B)$; $w_2(C)$; $w_3(D)$;
   (e) $r_1(A)$; $r_2(B)$; $r_3(C)$; $r_1(B)$; $r_2(C)$; $r_3(D)$; $w_1(A)$; $w_2(B)$; $w_3(C)$;

   (where $r_i(X), w_i(Y)$ means transaction T_i reads (resp. writes) data element X (resp Y), do each of the following:

   i. Insert shared and exclusive locks, and insert unlock actions. Place a shared lock immediately in front of each read action that is not followed by a write action of the same element by the same transaction. Place an exclusive lock in front of every other read or write action. Place necessary unlock at the end of every transaction.
   ii. Tell what happens when each schedule is run by a scheduler that supports ahred and exclusive locks.
   iii. Insert shared and exclusive locks in a way that allows upgrading. Place a shared lock in front of every read, an exclusive lock in front of every write, and place the necessary unlock at the end of the transactions.
   iv. Tell what happens when each schedule from (iii) is run by a scheduler that supports shared locks, exclusive locks, and upgrading.
   v. Insert shared, exclusive, and update locks, along with in-lock actions. Place a shared lock in front of every read action that is not going to be upgraded, place an update lock in front of every read action that will be upgraded, and place an exclusive lock in front of every write action. Place unlocks at the end of transactions, as usual.
   vi. Tell what happens when each schedule from (v) is run by a scheduler that supports shared, exclusive, and update locks.

2. For each of the following schedules, insert appropriate locks (read, write or increment) before each action, unlocks at the end of transactions. Then tell what happens when the schedule is run by a scheduler that supports these three types of locks.

(a) $r_1(A)$; $r_2(B)$; $inc_1(B)$; $inc_2(A)$; $w_1(C)$; $w_2(D)$;
(b) $inc_1(A)$; $inc_2(B)$; $inc_1(B)$; $inc_2(C)$; $w_1(C)$; $w_2(D)$;
(c) $r_1(A)$; $r_2(B)$; $inc_1(B)$; $inc_2(C)$; $w_1(C)$; $w_2(D)$;

3. Consider the following transactions:

$$T_1 : r_1(A); r_1(B); inc_1(A); inc_1(B);$$
$$T_2 : r_2(A); r_2(B); inc_2(A); inc_2(B);$$

Answer the following:

a) How many interleavings of these transactions are serializable?
b) If the order of incrementation in T_2 were reversed, how many serializable interleavings would there be?

4. Fun with PostgreSQL.
The *musicbrainz* database offers a large variety of tables to maintain and update a database on artists and their records. A schema is included as Figure 2.6. Your first challenge might be to obtain a more readable version of this figure.

The bitbucket repository owned by lalinsky (https://bitbucket.org/lalinsky/mbslave) has a sequence of instructions on how to make a local copy of this database to play with. It is currently several gigabytes large, and it is updated weekly. If, by the time of your using this text, the database administrators have decided to remove the option of maintaining a local copy, your friendly neighborhood instructor has made a(n older) copy that you can work with.

First try, following the instructions to download a recent dump of the database, and install it as a postgres database on your device.

The command `\dt` will provide us with an overview of the tables in the database. If that does not work try the following.

```
SELECT
    table_schema ||'.' ||table_name
FROM
    information_schema.tables
WHERE
    table_type = 'BASE TABLE'
AND
table_schema NOT IN ('pg_catalog', 'information_schema');
```

(a) INSERT use the INSERT INTO command to add a line to some table. It can be necessary, for example because there could be a foreign key constraints that demands that lines be inserted in other tables first.
(b) DELETE Remove an entry from some table don't forget to be very specific with the WHERE in this query, otherwise strange things may happen
(c) create a list of artist and their songs by joining a sequence of tables and selecting the appropriate columns from the joins (you may not want to see the entire list, but only a top 50 or so).

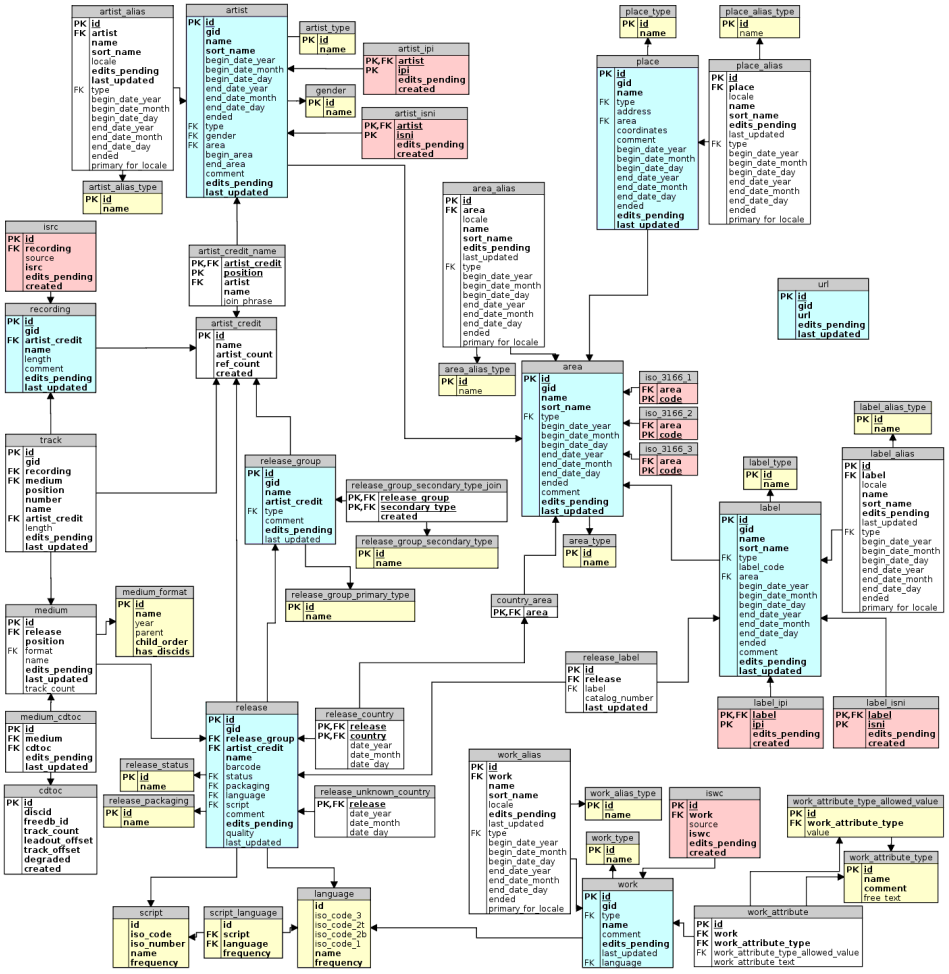


Figure 2.6: The musicbrainz database

Chapter 3

CouchDB

3.1 Introduction

The first database of the "modern type" we will discuss here is CouchDB. It is a so-called document database, which means it stores entire documents and retrieves entire documents. Querying the database is done by creating views as we will see later on.

3.1.1 REST and cURL

CouchDB databases (as are later discussed RIAK databases) take their CRUD orders trough actions on urls. The CRUD responsibilities are carried by the well known methods POST (for Create), GET (for Read), PUT (for Update) and DELETE (for Delete) respectively. cURL is the language of choice for communication with the database though some of these can also be hidden from the user through a web-interface. In the case of CouchDB, this interface is called FUTON. Installing Couchdb on a Linux machine should be easy, just use "sudo apt-get install couchdb." When installed, the FUTON web interface, needed for creating databases should be available standard on port 5984, so go ahead and open a web browser and point the web browser to "http://localhost:5984/_utils." You should now get a picture like Figure 3.1.

In futon, you can control the database, in particular create a database. In the next sections, we will set up a small books database to illustrate the workings of FUTON and the interaction of Couch with Curl.

The name of handling the CRUD tasks in this way is REST, which stands for Representational State Transfer, and using cURL as our REST interface allows us to speak directly to the database, without the need for any (other) programming languages (in sofar cURL can be called a programming langauge).

3.1.2 Map Reduce

Map Reduce is the winning strategy introduced by Google [**?**] for handling queries in distributed database environments. The idea is simple. In traditional database queries, the query selects a number of records, then the machine handling the query extracts from these records the desired information, and presents the results. In Map Reduce, part of the work extracting the desired information is shifted to the database nodes that subsequently return a list containing only the relevant part of the data, which is merged by one or more reduce

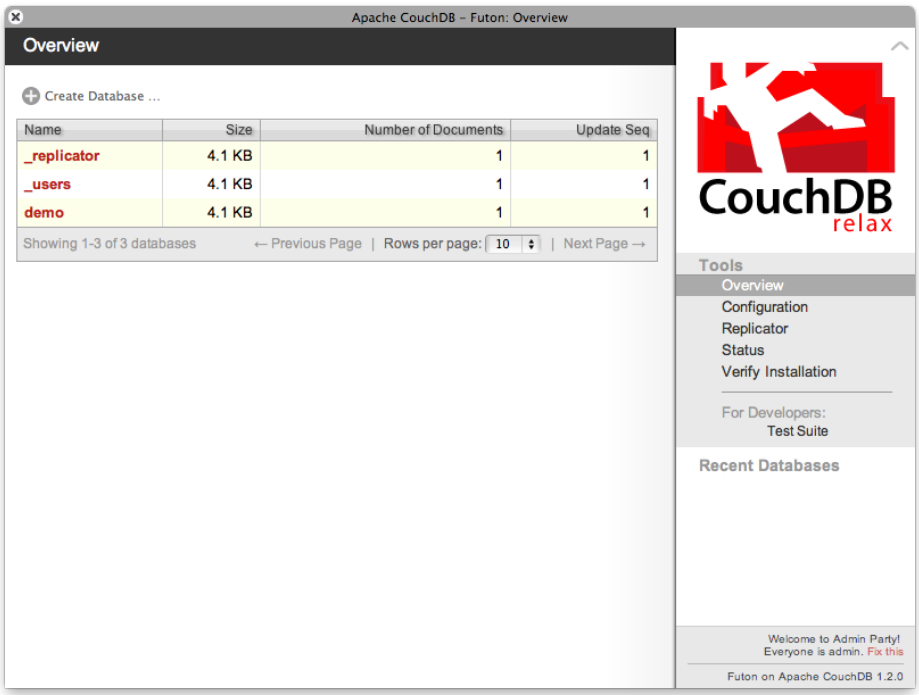


Figure 3.1: The FUTON interface

functions to the relevant value, much like the role "reduce" plays in LISP, or "fold" does in other functional language.

For instance, if a distributed database contains data records of the population of some country, and we would be interested in finding out how many records contain an age attribute over 35, then a traditional database system, would first retrieve all the records WHERE this attribute is greater than 35, and next count the number of records it retrieves. In a typical map reduce environment, each of the database nodes gets a map request, to return a list of values for the records where the age is greater than 35 (this list would look like $\{1, 1, 1, \ldots, 1\}$ and subsequent reduce functions for the node would sum this list to a per-node total, after which the next reduce function sums up these values of the per-node counts, to form the value queried for. The paper [**?**] contains various other examples along the same lines. Part of the power of the Map Reduce mechanism is that the map functions, as well as the reduce functions, are user-defined and could therefore be much more complicated than in our example, leading to the implementation of much more powerful queries.

3.1.3 FUTON

A good couch deserves a futon to lie comfortable. CouchDB has a web interface by which databases can be created and controlled, called FUTON. The web interface FUTON is used to create and control documents. When fired up by pointing the web browser to "http://localhost:5983/_utils" we see a picture like in Figure 3.1

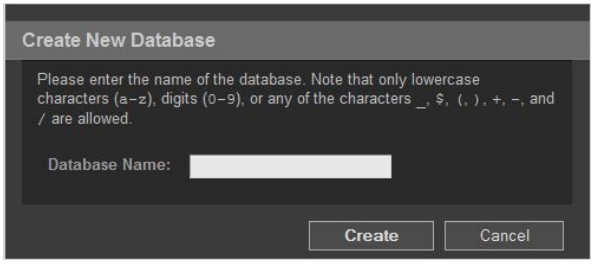


Figure 3.2: Pop Up

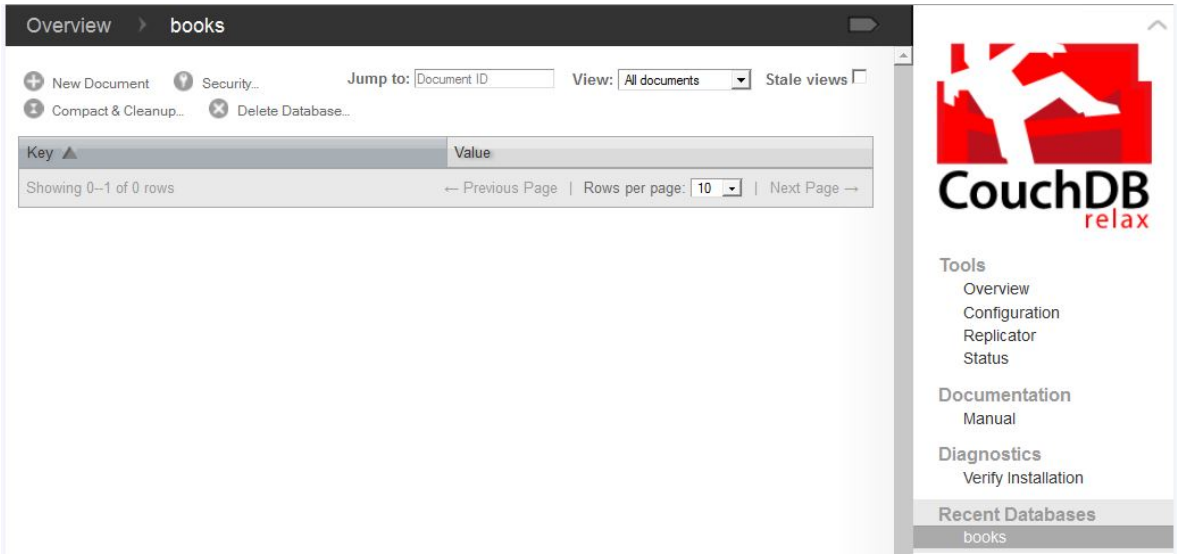


Figure 3.3: An empty database

3.2 Playing on the Couch

We will use this interface later to create a database on which we will extensively experiment with queries. For now, we will create a toy database called books, which will contain our collection of books. Futon has a button with which to create new databases. It is found next to the plus-sign on top of the page. After clicking, it should produce a pop-up like Figure 3.2, which after filling in "books" brings us to our new (empty) books database (see Figure 3.3).

Now, couchDB is a *document* database, which means it is filled with documents. No wonder there's a button that lets us create document. Pushing it results in creating a new document for our database. Documents in CouchDB consists of fields that have value. A field that is always present in CouchDB the "_id" field. It's value is generated uniquely, but the value is editable. We will need this field dearly later on and it might be helpful to give it a name that would type in more easily. We will let it be for now. Let us add some fields to make the document more meaningful. A book that has been sitting on my desk for a while could be added to the document, like is done in Figure 3.4

We need to store the document of course, and therefore we need to push the save button. Now we witness an important event (see Figure 3.5. Not only is the document stored, but, out of thin air, a new field "_rev" is added. This too is an important field, the value of which

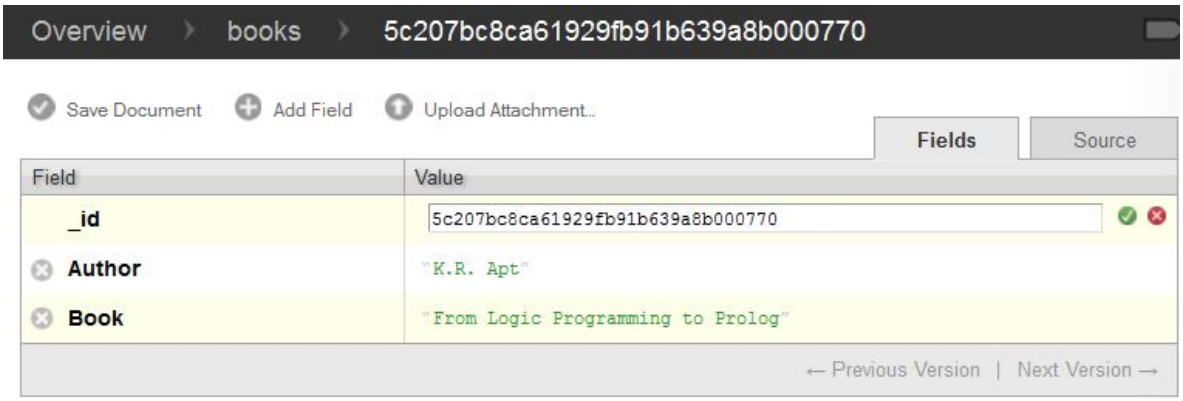


Figure 3.4: A first document

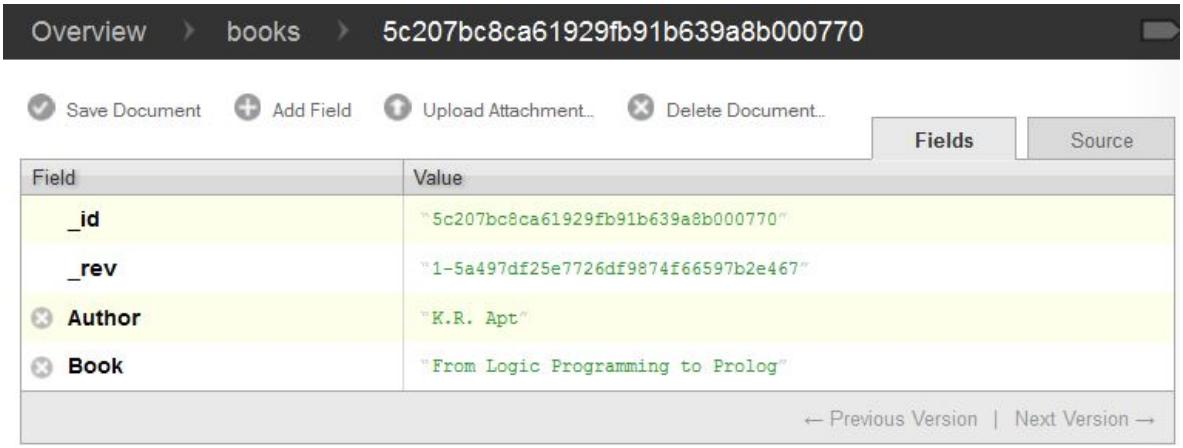


Figure 3.5: Storing the first document

we will also need dearly in the near future.

An important thing to note is that previous "revisions" remain available, and futon can browse through them. Couch always replaces an entire document. Even when the document is erased from the database, the document is not really destroyed. This may not always be the best choice performance wise, but might come in handy in some situations.

An author may have more than one book, so we might want to update the "Book" field with

```
[
"From Logic Programming to Prolog",
"Principles of Constraint Logic Programming",
"Constraint Logic Programming Using ECLiPSe",
"Verification of Sequential and Concurrent Programs"
]
```

Also, the title may not be the only thing there is to a book. We might try another update, like

```
[
```

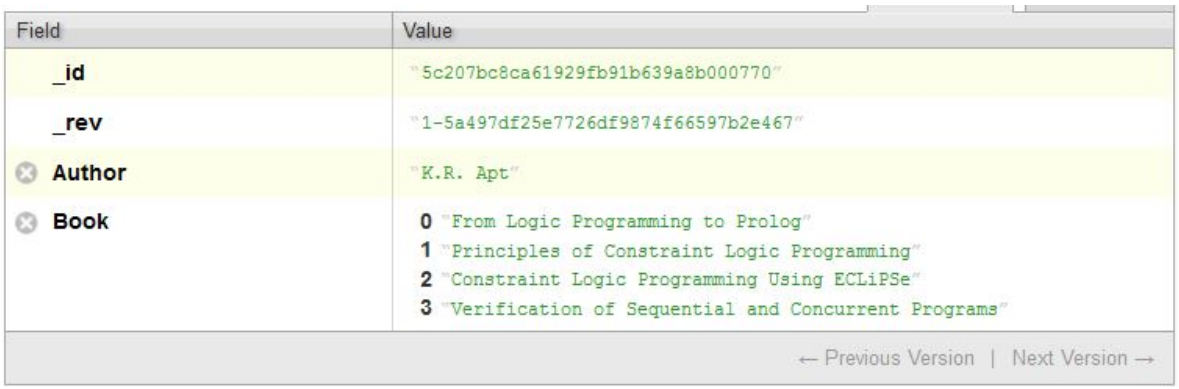


Figure 3.6: updated document

```
{"title":"From Logic Programming to Prolog",
"year":1997
},
{"title" : "Principles of Constraint Logic Programming",
"year":2006
},
{"title" : "Constraint Logic Programming Using ECLiPSe",
"year" : 2009,
"CoAuthor":"M.G. Wallace"
},
{"title" : "Verification of Sequential and Concurrent Programs",
  "year" : 2009,
  "CoAuthor": "F.S.de Boer and E.R. Olderog"
}
]
```

Try typing the text instead of copy-paste. You may notice that Couch is very forgiving relaxed kind of program. If it doesn't understand, it still accepts the value, but just doesn't parse it like we would like to. Instead it just accepts the value as a string. The remaining tasks of CRUD, we will illustrate with cURL, the handy command line interface for dealing with urls.

3.2.1 curl

The language of choice for dealing with many internet databases is curl. A curl command consists of a url, and several parameters that appear after option statements. A full description of curl can be found after "man curl," but we describe the effect of the very few parameters we use in these examples below.

-i (HTTP) Include the HTTP-header in the output. The HTTP-header includes things like server name, date of the document, HTTP version and more.

-H (HTTP) Extra header to use when getting a web page. You may specify any number of extra headers. Note that if you should add a custom header that has the same name

as one of the internal ones that curl would use, your externally set header will be used instead of the internal one.

-d (HTTP) Sends the specified data in a POST request to the HTTP server, in the same way that a browser does when a user has filled in an HTML form an presses the submit button. This will cause curl to pass the data to the server using the content-type application/x-www-form-urlencoded.

3.2.2 Read with GET, Create with POST, Update with PUT and Delete with DELETE

The following commands illustrate the use of curl in GET, POST, PUT and DELETE situations.

```
curl http://localhost:5984/books/aae2614505b0dfc58ec2752aea001872

{"_id":"aae2614505b0dfc58ec2752aea001872",
"_rev":"3-138c668fd536cdf9d97af3d3f6a3e985",
"Author":"K.R.Apt",
"Book":[
{"title":"From Logic Programming to Prolog",
"year":1997},
{"title":"Principles of Constraint Logic Programming",
"year":2006},
{"title":"Constraint Logic Programming Using ECLiPSe",
"year":2009,"CoAuthor":"M.G. Wallace"},
{"title":"Verification of Sequential and Concurrent Programs",
"year":2009,"CoAuthor":"F.S.de Boer and E.R. Olderog"}
]
}
```

(Note: I've added some line-breaks and indentation for readbility

```
curl -i -X POST "http://localhost:5984/books/"\
 -H "Content-Type: application/json"\
 -d '{"Author" : "L.Torenvliet"}'
HTTP/1.1 201 Created
Server: CouchDB/1.0.1 (Erlang OTP/R14B)
Location: http://localhost:5984/books/aae2614505b0dfc58ec2752aea00268d
Date: Fri, 08 Aug 2014 12:37:00 GMT
Content-Type: text/plain;charset=utf-8
Content-Length: 95
Cache-Control: must-revalidate

{"ok":true,"id":"aae2614505b0dfc58ec2752aea00268d",
"rev":"1-2a6b3b3dd7832c7b773c6bf512ae3308"}

curl -i -X PUT "http://localhost:5984/books/aae2614505b0dfc58ec2752aea001872" \
-H "Content-Type: application/json" \
```

```
-d '{ "_id":"aae2614505b0dfc58ec2752aea001872",
"_rev":"3-138c668fd536cdf9d97af3d3f6a3e985",
"Author":"L.Torenvliet",
"Book":[{"title":"Theory of Semi-Feasible Computation", "year":2002}]}'

HTTP/1.1 201 Created
Server: CouchDB/1.0.1 (Erlang OTP/R14B)
Location: http://localhost:5984/books/aae2614505b0dfc58ec2752aea001872
Etag: "4-b3fe0f1b4172b7309c4c3ef99acd3e90"
Date: Fri, 08 Aug 2014 13:06:58 GMT
Content-Type: text/plain;charset=utf-8
Content-Length: 95
Cache-Control: must-revalidate

{"ok":true,"id":"aae2614505b0dfc58ec2752aea001872",
"rev":"4-b3fe0f1b4172b7309c4c3ef99acd3e90"}
```

Note: both the _id and the _rev fields must address exactly those in the database, otherwise you will get a conflict updating the document. The system will return a HTTP 409 Conflict with a JSON object describing the problem as `reason : "Document update conflict"`. You may *think* you can edit individual fields, but you cannot. Couch is rewriting the entire document and creating a new version.

```
curl -X DELETE "http://localhost:5984/books/aae2614505b0dfc58ec2752aea001872"\
 -H "If-Match: 4-b3fe0f1b4172b7309c4c3ef99acd3e90"

HTTP/1.1 200 OK
Server: CouchDB/1.0.1 (Erlang OTP/R14B)
Etag: "5-7b98108657572ce67e9c6eebbcef6a64"
Date: Fri, 08 Aug 2014 13:14:23 GMT
Content-Type: text/plain;charset=utf-8
Content-Length: 95
Cache-Control: must-revalidate

{"ok":true,"id":"aae2614505b0dfc58ec2752aea001872",
"rev":"5-7b98108657572ce67e9c6eebbcef6a64"}
```

3.2.3 Views and Stuff

Now it is time to play with a database of a bit larger size. For this purpose we have selected a book database that is a little bit edited for creating a couch database, derived from the English library open data set.

In the exercises, we will prepare a library database from a public source. For the next examples, we have already done so. Let us first look at a typical document in the database (see Fig 3.7).

As we see, a typical document has a title, a creator (author), and one or more contributors. The database was imported using a ruby script from an xml/rdf source (rdf stands for

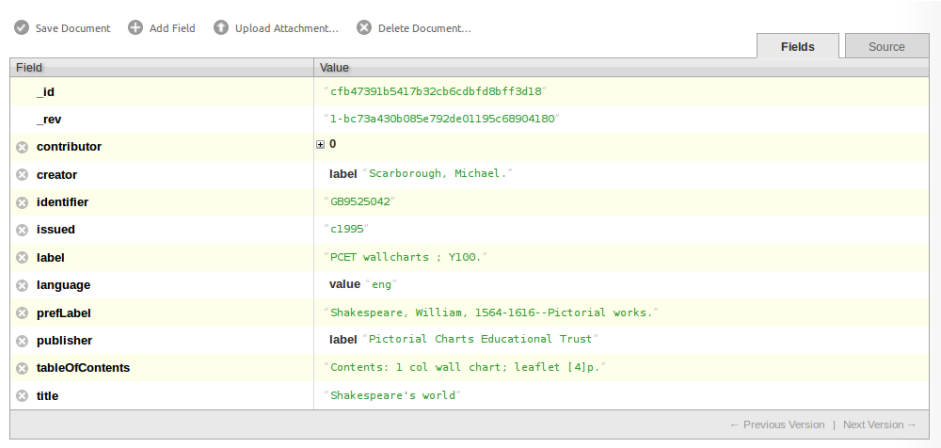


Figure 3.7: Structure of a typical book document

"resource description framework" a format that is becoming standard for these type of files) in which the <creator>field had a sub-field called <label>which encloses the name of the the creator. We see the effects of this in the structure of the document. Not all documents may have a creator, and or contributors. We can ask couchdb for the ones who do. In the database field, we can point to the field **View** and set it to "Temporary view." We can now program a Map Function that lets us select the documents that do have a creator. This is shown in Figure 3.8.

For these docs, we might actually want to know who this creator might be. No problem (see Figure 3.9).

We can also produce more than one item per document, if a document has multiple field of the same kind. In our case that might be the contributor field. Look at the example in Figure 3.10.
Notice that each of the key listings is equipped with an ID of the document that is clickable.

Storing Views

You may have noticed that it takes quite some time to compute the map function on a sizable database. It may therefore be profitable to store the view obtained by the map in the database. On the top right corner of the futon interface we see that there is a "save as" button. Pushing this button brings up a dialog that asks us for a name of the "design" and a name of the "view." The design is sort of a category that stores views together that have the same nature. This is of course mostly a matter of taste, but should worry us when we are really trying to design a database. For now let us store the database in the design "titles" and name the view "by_author_and_coauthor," as is a reflection of the information in the views.

Storing the view has the advantage that we can retrieve it. Let us try it. (Warning: be prepared to hit ctrl-C)

type:

```
curl http://localhost:5984/ruby_books/_design/title/_view/by_author_and_coauthor
```

We can also search the view by adding a limiting value with "?" as in

```
curl 'http://localhost:5984/ruby_books/_design/titles/_view/by_author_and_coauthor?key="Yugoslavia"'
```

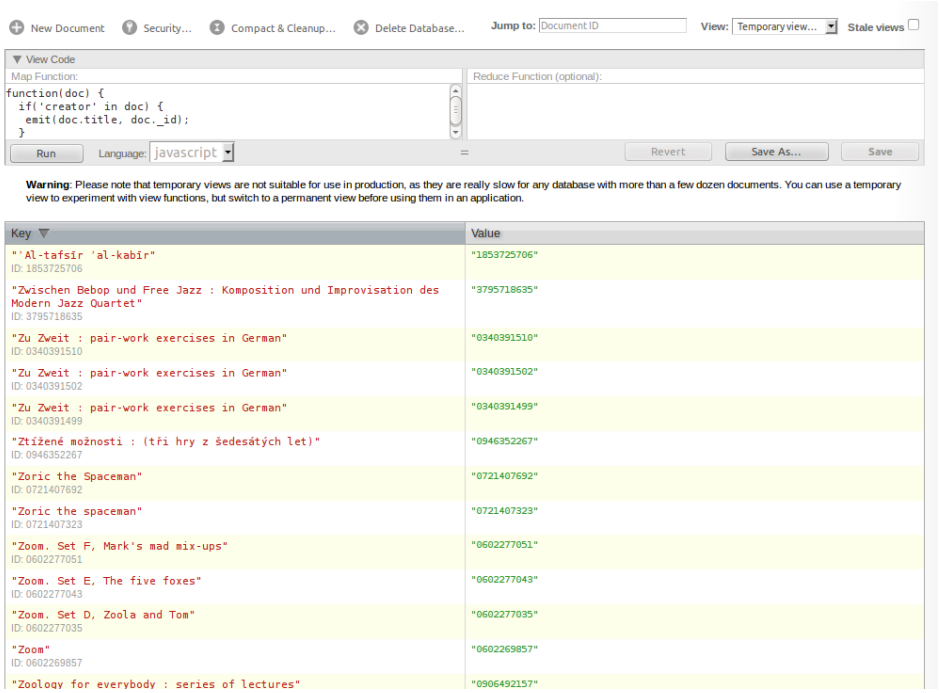


Figure 3.8: Map Creator ID

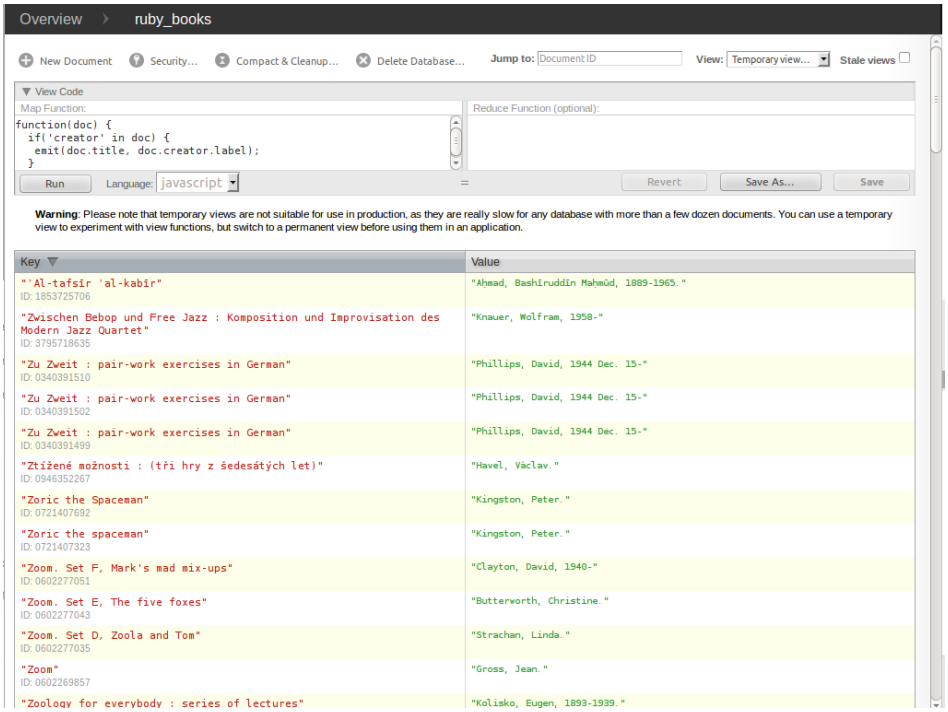


Figure 3.9: Title and creator

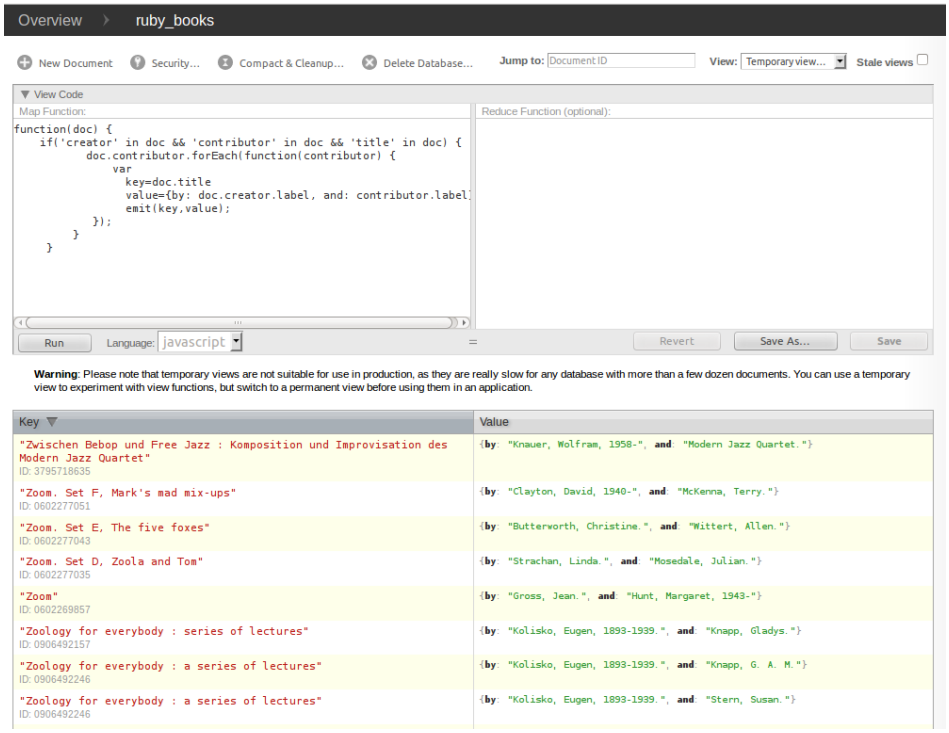


Figure 3.10: per contributor

Note the single quotes around the url. If you don't place them, you might get the (very informative)

```
{"error":"bad_request","reason":"invalid UTF-8 JSON"}
```

and spend another hour or so tracking the fact that bash has eaten your double quotes around Yugoslavia.

Now we get:

```
{"total_rows":37117,"offset":37057,"rows":[
{"id":"0850901812","key":"Yugoslavia","value":{"by":"Hawkes, John Ryder, 1910-",
"and":"Hawkes, Moira, 1917-"}},
{"id":"0850902444","key":"Yugoslavia","value":{"by":"Hawkes, John Ryder, 1910-",
"and":"Hawkes, Moira, 1917-"}},
{"id":"0850902703","key":"Yugoslavia","value":{"by":"Hawkes, John Ryder, 1910-",
"and":"Hawkes, Moira, 1917-"}},
{"id":"0850903513","key":"Yugoslavia","value":{"by":"Hawkes, John Ryder, 1910-",
"and":"Hawkes, Moira, 1917-"}},
{"id":"0850903718","key":"Yugoslavia","value":{"by":"Hawkes, John Ryder, 1910-",
"and":"Hawkes, Moira, 1917-"}}
]}
```

as we would expect.

3.2.4 Map and Reduce

Having a set of documents that have authors (creators) and contributors we might be interested, for example, in how proficient certain co-authors were on our documents. The map/reduce system in modern databases, here in Couch, gives an excellent mechanism to do so. First let us define a Map function that can count.

```
function(doc) {
  (doc.contributor || []).forEach(function(contributor) {
   emit(contributor.label, 1);
  });
}
```

Now we also put something in the (optional) "Reduce Function" field of Futon, namely the following.

```
function(key,values,rereduce) {
   return sum(values);
}
```

Pushing "Run" should give something like Figure 3.11

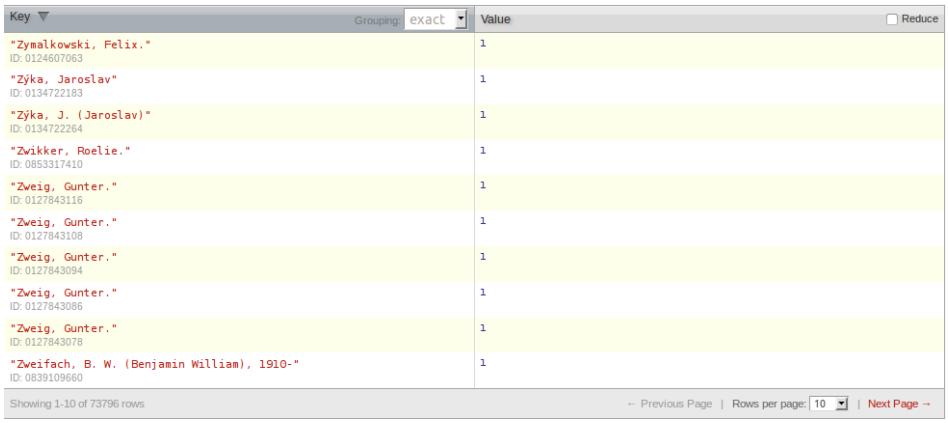


Figure 3.11: counting contributions

which is reasonable, since we only asked to return a 1 per counted contribution. However, notice the "Reduce option" in the upper left corner and check it. Now we see the output transforming to 3.12

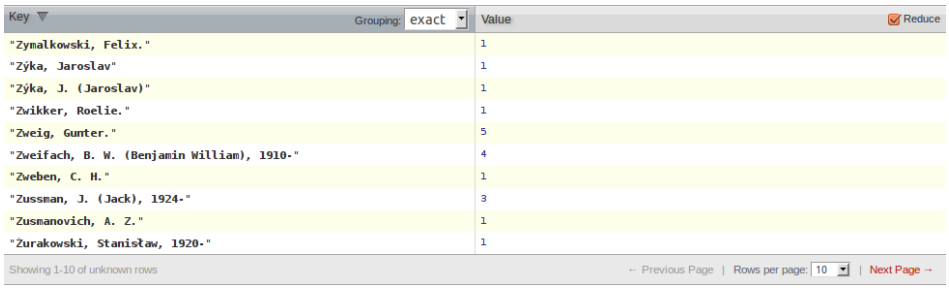


Figure 3.12: summing contributions

which shows the format we wanted. Apparently some co-authors have contributed to more than one document.

3.3 Exercises

1. First, download the "BNB Basic" data set from the British Library `http://www.bl.uk/bibliographic/download.html` (Please use the RDFXML verson to avoid confusion). Secondly write a script that inserts this data set to your local CouchDB installation. Please note that you have to make some changes in the structure of the data set to make it suitable for CouchDB, otherwise executing queries will be rather painful. Both an RDF package and a Couchdb interface are available for python. Please use these if you plan to use this language.

2. Reproduce the examples in this chapter with the database produced.

3. Using map reduce queries get a per country overview of the number of authors and the (number of) languages their respective works have appeared in. Again, you may want to limit the output (more elegantly than hitting ctrl-C)

Chapter 4

Neo4J

4.1 Introduction

The next database in the series of modern databases is the graph database Neo4j. A graph database is a totally different concept than the previously mentioned databases. A graph database represents data using nodes and different relationships (edges) between those nodes. When executing queries on the database you can search for nodes (having a specific relationship) and paths between those nodes defined by those relationships. Nodes and relationships can also have specific properties which can also be used when querying the graph. Figure 4.1 shows the relationship between nodes, properties, labels and relationships using a graph. This figure also is a good example of how a database looks like in Neo4j: just nodes and edges. Let's try!

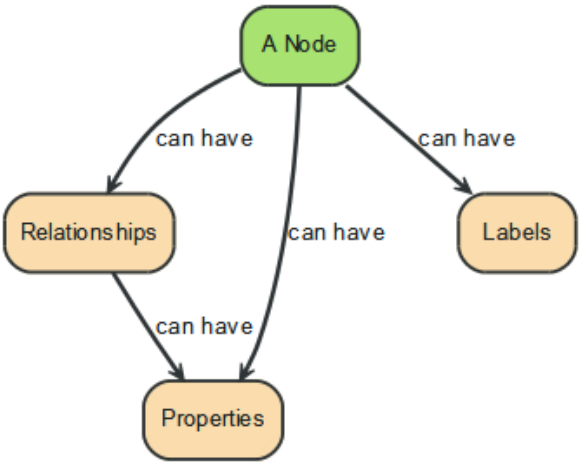


Figure 4.1: The properties of a node in Neo4j

4.1.1 Installing and starting Neo4j

Download the tarball from Blackboard and extract it somewhere on your computer. Open a terminal and execute the command "`bin/neo4j start`". You can check whether Neo4j has been started succesfully by browsing to `http://localhost:7474`, here you will find the web-interface of Neo4j. If Neo4j does not start, please make sure that your system

does not bind to ports 7474, 7373 and 1337. It is likely that Neo4j will give some warning about the maximum open files allowed by the operating system. You can remove this warning by following the instructions in the documentation: `http://neo4j.com/docs/2.1.6/linux-performance-guide.html#_setting_the_number_of_open_files`. This requires root-access to your system and removing this warning might not be neccesary for this course (but be aware of it).

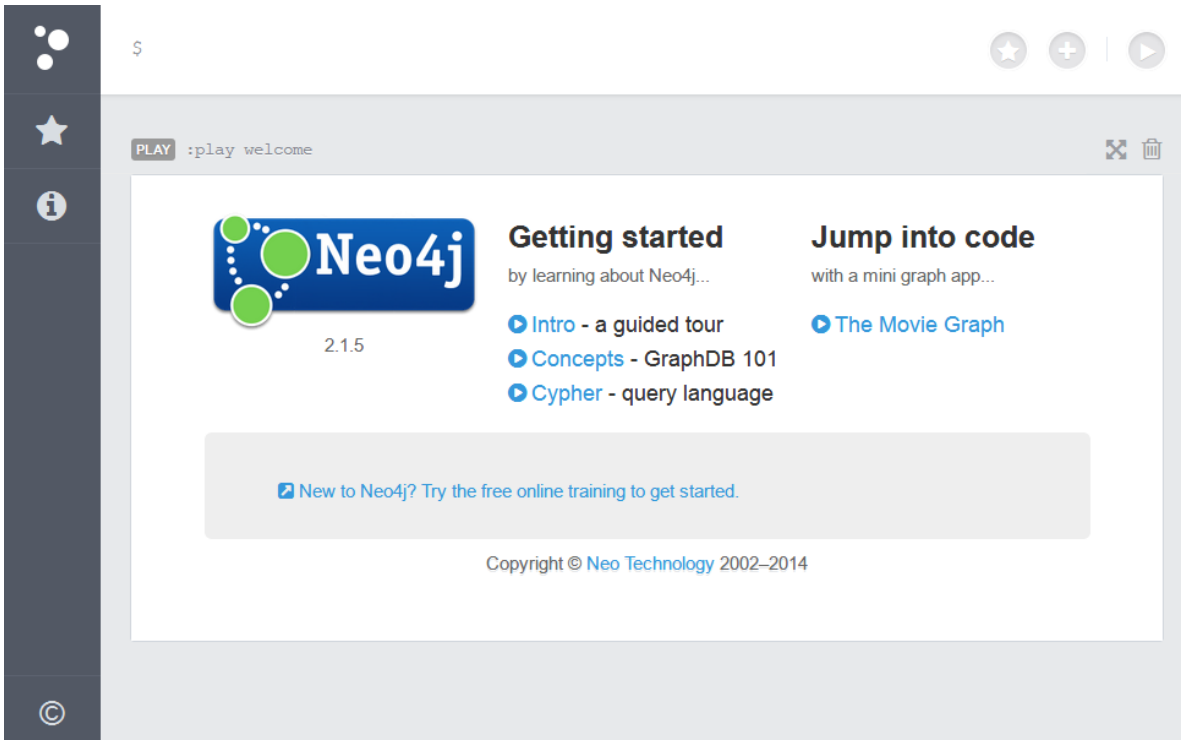


Figure 4.2: Welcome to the Neo4j web-interface.

4.1.2 Loading demo data

The Neo4j-archive on Blackboard contains some demo-data: a small movie database. If you have downloaded another installation of the Neo4j database (because you run on another platform), you can find the queries to insert this demo-data on Blackboard. Also this data is included in the builtin-demo, which you can start by entering ":play movie graph" in the command-line on top of the web-interface.

4.2 Cypher

Cypher is the query language that ships with Neo4j. It is a declarative language (you describe your results and the computer generates it) and it has some similarities with SQL. If you like SQL, you will find yourself comfortable with Cypher very quickly. But the main difference is that SQL is designed to work with tables, columns and rows and Cypher to deal with nodes and edges. A Cypher-query consists of several clauses (like SQL for example has a SELECT-command with FROM and WHERE-clauses). If you want to retrieve data from the

database, you have to create a query with “MATCH” and “RETURN” clauses.

The query `MATCH a RETURN a;` will find every node in the database (because we do not filter, only assign to a variable called a) and return those nodes. So in general, this query will return the complete database! When entering it in the Neo4j web-interface, the result will look like Figure 4.3. Besides this very awesome graphical representation, we can also ask for results in the form of tables. Tables are more practical response when creating a program, but less fancy for the eye. Using the toggle-button in the right-bottom corner, you can switch to table-layout. Figure 4.4 shows the result in form of tables, less fancy but more useful!

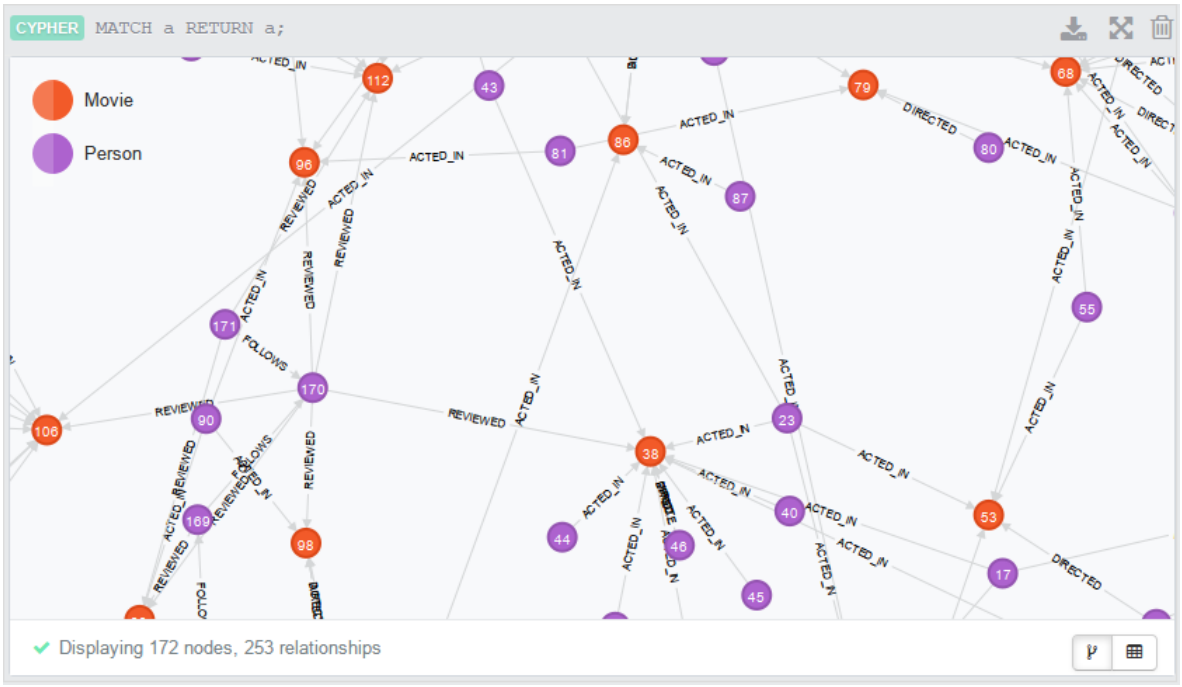


Figure 4.3: `MATCH a RETURN a;` result represented as a graph

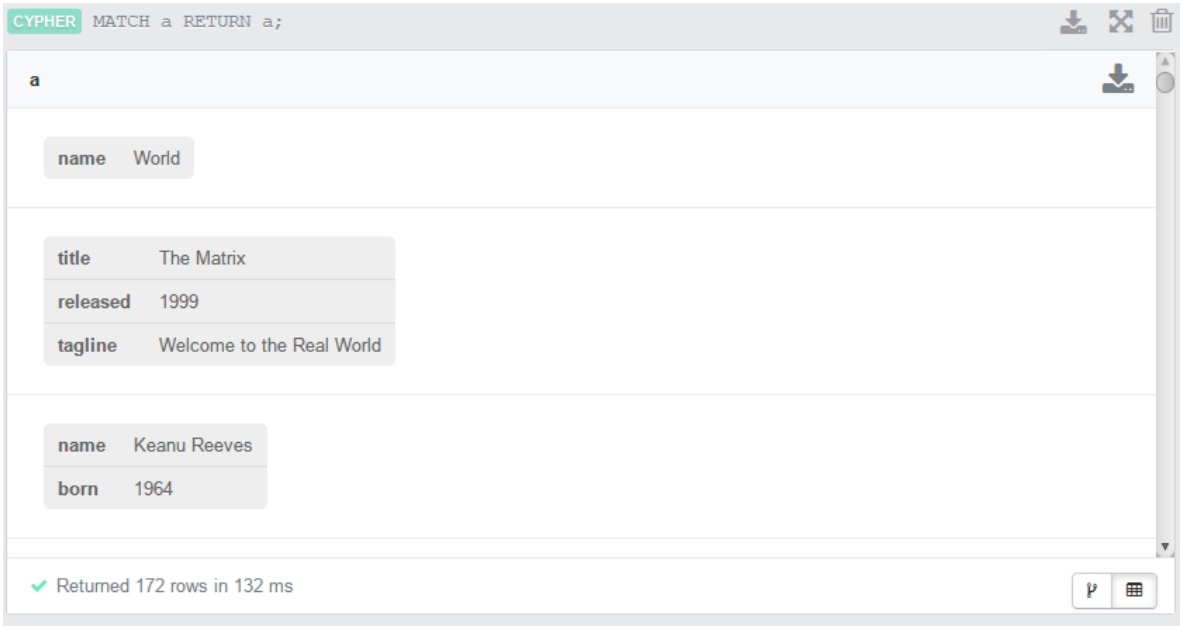


Figure 4.4: `MATCH a RETURN a;` result represented as tables

4.2.1 WHERE-clauses

Of course we can filter some of the nodes by means of a WHERE-clause, this looks very familiar to SQL. For example, finding the node(s) with the name "Tom Hanks" can be done by executing `MATCH a WHERE a.name = "Tom Hanks" RETURN a;`. In this case "name" is an attribute of the node on which we can apply a filter. It is also possible to filter on attribute directly when binding the node to a variable. The query `MATCH (tom name: "Tom Hanks") RETURN tom;` gives an equivalent result.
As with SQL we can add multiple selection criteria using an "AND"-clause. For example `MATCH a WHERE a.born > 1980 AND a.born < 2000 RETURN a.name` will return the name attributes of the nodes with a born attributes with values between 1980 and 2000.

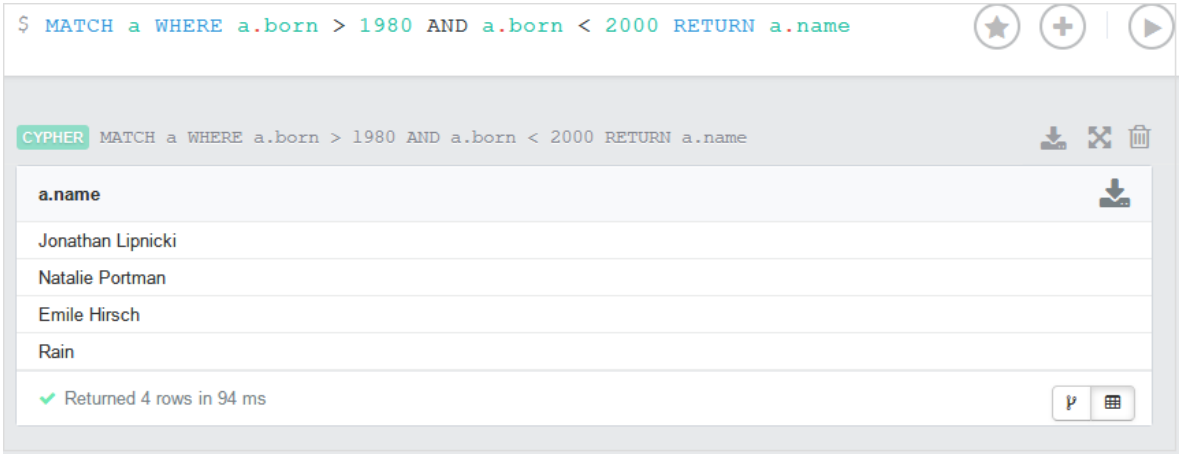


Figure 4.5: A slightly more interesting Cypher-query.

4.2.2 Querying relationships

In the match-clause we can also query multiple nodes (with an edge between them) like: `MATCH a--b RETURN a,b;`. This example will return two pairs of nodes with a relationship. In case of the "`--`" the direction of the edge does not matter (all edges in Neo4j have a direction), so this query will give (a,b) pairs with (movie,person) but also with (person,movie). If you want to specify the direction of the edge, one can use "`-->`" instead of "`--`" when writing a query. Because relationship can contain properties, they can also be bound to a variable-name. This can be done by placing brackets with a variable name between the dashes of the arrow, (like `a-[r]->b`). Also the type of a relationship can be returned by using `type(r)` where "r" is the name of the variable bound to the relationship.
Figure 4.6 shows the execution of the query `MATCH a-[r]->b WHERE a.name="James Marshall" RETURN a,b,r,type(r);` which shows how this relationship querying works in practise. Please note that some people are not just actors, but also directors or producers of other movies.

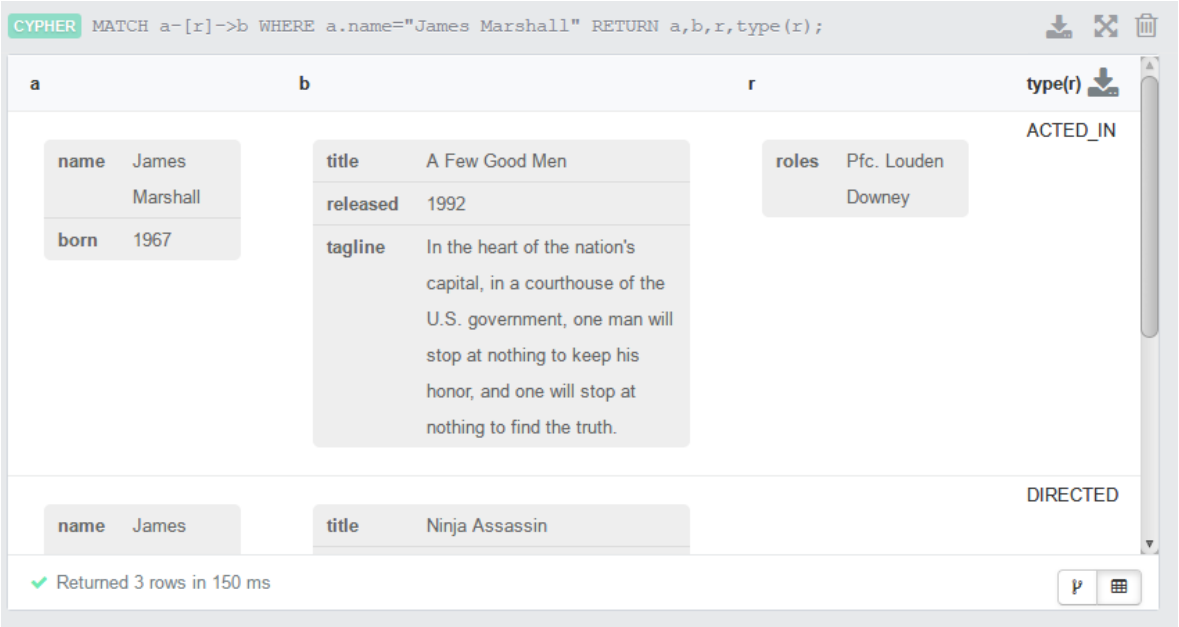


Figure 4.6: Querying a relationship.

One can filter the type of a relationship using the WHERE-clause, but an another (cleaner) way of selecting the type of a relationship between nodes is by the putting the relation type in the MATCH-clause like: `MATCH a-[r:‘ACTED_IN‘]->b RETURN a,b,type(r) LIMIT 25`. This query will return 25 nodes that are connected with an "ACTED_IN"-relationship.

4.3 Python

Using the Neo4j web-interface is fun, but not very useful when building a real application. Fortunately there are several packages to query a Neo4j-database using your favourite programming language! For example, we will now show a Python package, but you can also use Java, Ruby, PHP or any other language using the REST API.

4.3.1 Installing the Py2neo package

We will show how to use the "Py2neo" package, which can simply be installed by running "`pip install py2neo`" on your computer (you might need to execute this as root or using sudo). If you do not have pip installed on your computer yet, you can install it by running "`sudo apt-get install python-pip python-dev build-essential`" on Debian/Ubuntu-based systems, and using similar commands on other Linux distributions.

4.3.2 Py2neo basics

Once installed, create a Python-file and connect to the database running on your local-machine using the code listed below.

```
from py2neo import Graph
graph = Graph("http://localhost:7474/db/data")
```

Now we can find our favourite actor by executing a Cypher-query using the Python binding and print the results:

```
results = graph.cypher.execute('MATCH (a{name:"Tom Hanks"})  RETURN a;')
print(results)
```

```
   | a
---+-------------------------------------------
 1 | (n72:Person {born:1956,name:"Tom Hanks"})
```

The `results` variable will contain an array which contains the several variable(s) defined in the Cypher-query. Like a dictionary you can retrieve the properties of this variable:

```
print(results[0].a['name'])
```

```
Tom Hanks
```

You can find more about the Py2neo binding on their website: `http://py2neo.org`.

When more of the moviedatabase is implemented, you might find pictures like this.

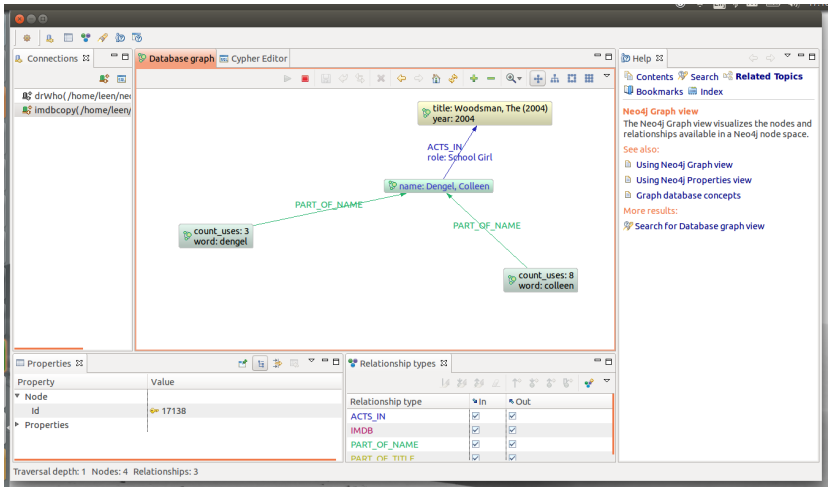


Figure 4.7: The imdb database as seen through neoclipse

4.4 Exercises

1. Give two pros and two cons of a graph database (like Neo4j) compared to a relational database (like PostgreSQL). Use the movie-database to strengthen your arguments.

2. Find a copy of the imdb database on the internet. Load it into Neo4j and find out how many movies it has in which your favorite actor/actress plays a role.

3. Load the database schema in the postgresql chapter into neo4j. The nodes are tables and the edges are...what?

Chapter 5

RIAK

5.1 Introduction

An example of a simple key-value type database is RIAK. In this section we will fire up a couple of nodes on a single computer that will play a ring of processors talking to each other and (redundantly) storing data. Riak makes use of the Chord algorithm for assigning data to nodes. To have full appreciation of the working of RIAK's data store, it is better to have a (simulated) ring of nodes.

The way we will mostly approach RIAK is through curl commands, so it is best to have curl installed on your machine. It's easy to get just through sudo apt-get. You can download a current version of RIAK from http://www.basho.com. There are several compiled versions of the server available, but, as with many of these projects, it is a moving target and code is not always in an optimal form. The first time I tried just downloading the binaries, the system complained that the code was unstable and dangerous.

Fortunately, there is also source available, and al that is needed is (at the time of this writing) Erlang and C++, both of which can also be installed throught the sudo apt-get command.

If all is well then getting an up to date version of Riak *might* be as easy as downloading and unpacking the zipped source files and typing `make devrel` into the console.

This might take some time and will almost surely reveal lots of warnings and may even require you to sudo apt-get some more software along the way, but at the end of the session you should have a new dev directory in between your riak files in which you find dev1 through dev8 as subdirectories, representing eight nodes in a ring of database servers. With the commands

```
dev/dev1/bin/riak start
dev/dev2/bin/riak start
dev/dev3/bin/riak start
```

you can fire up any number of servers of course, but three to five will suffice. These servers "listen" to commands coming to them on different ports of the localhost. You can read *and change* the names of these ports in the config files belonging to the individual servers. When I tried it, the file was called etc/riak.conf in the dev/devX directory of the server concerned and server 1 listened to port 10018. (In later experiments, the port changed to 41962 and others.) Let us call the port that RIAK listens to PORT for now.

The three servers are not yet connected. We can tell them that they should join up to form a single database by using the administrative console in the bin directory belonging to

the server. This looks like
`dev/dev2/bin/riak-admin join dev1@127.0.0.1`
Note: In the version you have downloaded the join command may already have been deprecated. Newer versions use `riak-admin cluster`. The command may then look like `dev/dev2/bin/riak-admin cluster join dev1@127.0.0.1`. to make the second server join forces with the first, and a similar command to make the third server join forces with the second.

OK. Now we are ready to exchange some data. You might first want to check that the servers are alive by typing `http://localhost:PORT/stats` into the addres space of your favorite webbrowser and get a host of, mostly unintelligible, information on the status of all three nodes.

5.2 CRUD

Just as before, REST (REpresentational State Transfer) is taken care of completeley by curl commands in RIAK. POST stands for Create, GET takes care of Read, PUT takes care of Update and DELETE stands for Delete.

We use curl first to see whether our newly formed three sized ring of servers is alive by typing
`curl http://localhost:PORT/ping`
It should answer saying `OK`.

Then we are ready to store some stuff in the RIAK database. No need to create anything, just store. Let's first try to put some html code that we can later retrieve. As noted above PUT will store data in the database by using the -X option of curl. The -H option will provide a html header, and the -d option (also data option) will let us store some text in the database. It looks like `curl -v -X PUT http://localhost:PORT/riak/favs/db -H "Content-Type text/html" -d "<html><body><h1>Here's RIAK your friendly neighborhood distributed database system</h1></body></html>"` .

Point your browser to the address localhost:PORT/riak/favs/db to see the text you just stored there.

We may want to see what buckets are in the database so-far. We can use thet GET command for that as in `curl -v -X GET http://localhost:PORT/riak?buckets=true`, which will give us a listing of the buckets put in the database.

Now it is time to build our little book databases. We use the `PUT` command to create the data as follows.

```
curl -v -X PUT http://localhost:PORT/riak/books/apt?returnbody=true \
-H "Content-Type:application/json" \
-d '{"Author" :  "K.R. Apt" , "Title" :  "From Logic Programming to Prolog", "Year"
:"1997" }'
```

It's a bit tedious to add new books to the database in Riak. If we have new books by the same author, we have to update the bucket, this means copying the entire text while adding new info. Like so.

```
curl -v -X PUT http://localhost:PORT/riak/books/apt?returnbody=true \
-H "Content-Type:application/json" \
-d '[{"Author" :  "K.R. Apt" , "Title" :  "From Logic Programming to Prolog", "Year"
:"1997" }, { "Author" :  "K.R. Apt" , "Title" :  "Principles of Constraint Logic
```

`Programming", "Year" :"2006"}, { "Author" : "K.R. Apt" , "Title" : "Constraint Logic Programming Using ECLiPSe", "Year" :"2009", "CoAutor":"M.G. Wallace" }, {"Author" : "K.R. Apt" , "Title" : "Verification of Sequential and Concurrent Programs", "Year" :"2009", "CoAuthor" : "F.S. de Boer and E.R. Olderog"}] '` Getting tired of curl as your interface of choice? Try installing Rekon, a nifty little tool developed by Basho. After installing, you can point your browser to "localhost:PORT/buckets/rekon/keys/go" and enjoy the interface as pictured below:

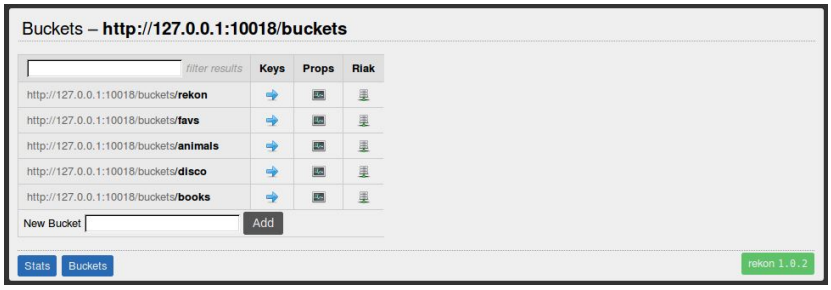


Figure 5.1: localhost:PORT/buckets/rekon/keys/go

Figure 5.2: which keys

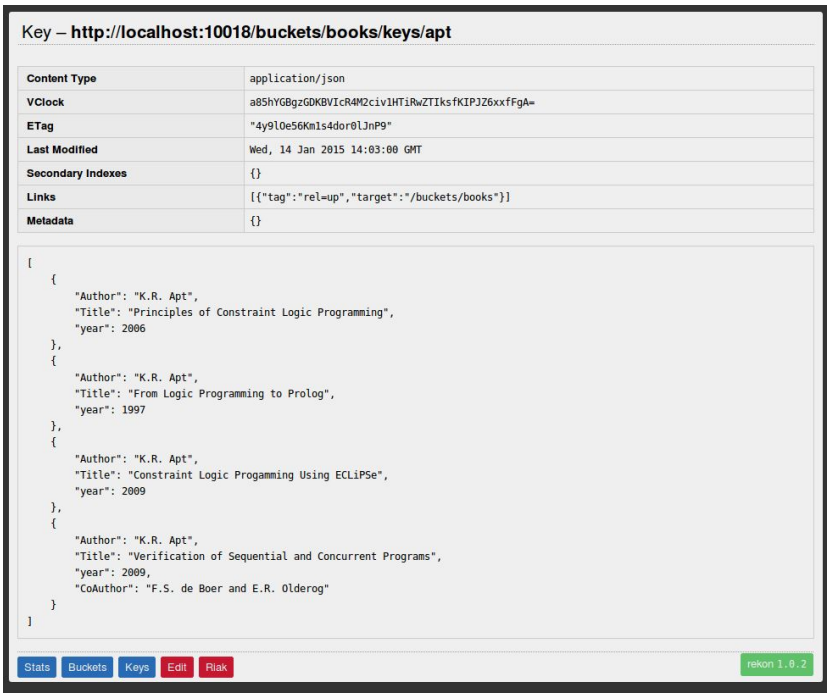


Figure 5.3: What content

Oops. Did something go wrong? What's with all the backslashes. Not to worry, that is just the way Rekon presents its data. After typing `curl -v -i http://localhost:PORT/bucket/books/key/apt`, you will find: [{"Author" : "K.R. Apt" , "Title" : "From Logic Programming to Prolog", "Year" :"1997" }, { "Author" : "K.R. Apt" , "Title" : "Principles of Constraint Logic Programming", "Year" :"2006"}, { "Author" : "K.R. Apt" , "Title" : "Constraint Logic Programming Using ECLiPSe", "Year" :"2009", "CoAutor":"M.G. Wallace" }, {"Author" : "K.R. Apt" , "Title" : "Verification of Sequential and Concurrent Programs", "Year" :"2009", "CoAuthor" : "F.S. de Boer and E.R. Olderog"}]}
as expected.

We can also see this when we expand the key in rekon. Thus

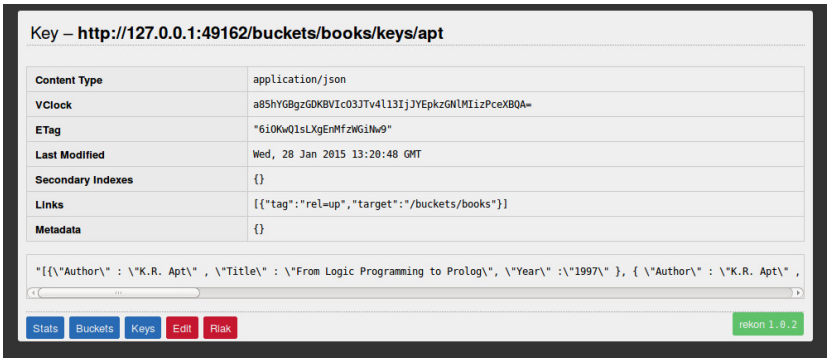


Figure 5.4: What keys

One more aspect remains of CRUD, which is the DELETE. This can also be done with

curl. Type `curl -i -X DELETE http://localhost:PORT/riak/books/apt` and the key is gone.

5.3 Map Reduce in RIAK

Map reduce is definitely a bit harder to do using CURL than it was using the FUTON interface. The commands in curl look like

```
curl -X POST -H "content-type:  application/json"
http://localhost:PORT/mapred --data @-
{
       "inputs":[
             [bucket1,key1],[bucket2, key2], [bucket3, key3],...
       ],
       "query":[
       {"map:" {
           "language":"javascript",
           "source":
           function(v) {
           //PARSE THE Riak Object as JSON
           }
       }}
    ]
}
```

The sequence of bucket,key data items can also be given as just "bucket" (e.g. [books,apt] or just "books"), but then in some cases a lot of data can be returned.

Let us flesh up the database of books a bit. I've copy/pased the list of the 100 most visited books of the Gutenberg project, edited it, sorted it in a couple of curl commands and then we get bucket books and keys like below. (Don't use rekon on a database with many keys, or you'll find you have to wait a while).

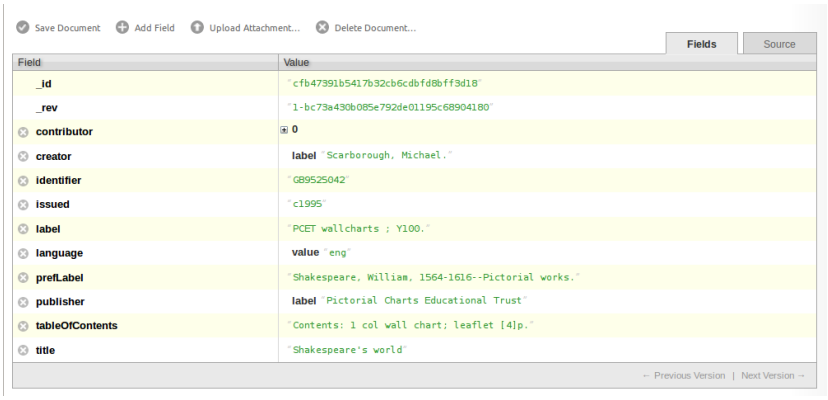


Figure 5.5: booksB key

5.4 Exercises

1. Download the riak sources (there's a git repository at basho.com) and install a test cluster of five nodes.

2. Carnegy Melon University (cmu.com) has a tarfile that contains all the mailfiles of the Enron executives at the time of the big fraude case. Install the mail folder as a riak database (e.g., make buckets for the persons and keys for their mailfolders and a hashfile for the individual mailfiles and their contents).

3. Create a map-reduce search for sensitive words (e.g., political, fraud, scam etc.)

4. Make an implementation of the Chord algorithm (e.g., in python) that resembles Riak's search algorithm. Nodes in the graph should maintain local hashtables and one of the nodes should be responsible for handling search request. A request should hash to a primary machine and that machine, if unsuccessful, should make the requests to other nodes in its finger table. The algorithm should be robust to adding and deleting nodes in the ring.

Chapter 6

Project

6.1 Introduction

In the last three week of this course you will implement your own distributed database. In a separated document you can find a detailed project description. In this chapter we will discuss some of the key components of your database. The first component is the Chord algorithm for location management of the data. Although Chord is described well in it's own article [**?**], we will spend some words on this algorithm in the next section. After this, there is an introduction to hashing. Hashing is a very important component in the Chord implementation and choosing the wrong algorithm can be catastrophic. The last section will be about vector clocks, in which you learn how to get your database eventually consistent.

6.2 Chord

Chord is an algorithm, or rather a protocol, for storing a distributed hash table in a set of computers called "the ring." A (distributed) hash table consists of pairs of keys and values. The value is to be stored in the memory of the machine, be it volatile or more permanent memory, and the key designates the place where the value is to be stored. The hash function computes the address of the processor from its argument. As such, the hash function just determines the place in memory where the value will sit for a given machine in the ring, called a *node*. Before the place in the table of a given node is computed, Chord determines from that same key the number of the node, actually its ip-address, that will be given the task to store the node. The matter is complicated for several reasons. We have already seen in the chapter on Riak, a key-value database, some of these complications. To wit:

1. New machines can either leave or enter the ring.
2. Data can and will be stored in more than one node to increase reliability of the system and availability of the system in case of a crash.

Therefore, storing the data will not give a unique name of the node responsible for the data, but rather a range of nodes in which one principal node may be addressed first when attempting to retrieve the data, but we need to take into account that it is possible that the main node responsible for the data crashes or leaves the ring for other reasons (maybe just network speed, or bad weather), and the data must be searched in other nodes as well. In

a ring, each node has a predecessor and an a successor, of which the addresses are stored in the node. The identifiers of these need not be consecutive. In fact the hash algorithm will work better if they are uniformly spread in the address space. This will give a lesser form of collision or congestion and a more even spread of the values between the nodes.

When a node receives a request for some key, it searches its memory for the corresponding value, but independently passes the request to its successor. This of course may lead to a worst case performance of $O(n)$ where n is the number of processors, mutliplied by the complexity of the local hash function.

To fix this problem, the Chord protocol provides for each node a so-called *finger table*. This is a table of nodes that are, in the network, powers of two apart. So the first node in this table is the nodes successor, the second node is the successor of the successor (to be found in the successors finger table). From here on, it gets a bit more complicated. So suppose that Node N has a successor called Node S, and that Node S is stored in the finger table of Node N, and that the successor of Node S, Node SS is stored both in the finger table of node N and in the finger table of Node S. Node SS, will also have a finger table, in which it stores its successor, Node SSS and the successor of this node, Node SSSS. Now Node N can know which address to store in the third position of its finger table. It's Node SSSS, whose address it can learn by asking it of Node SS, whose address is already learned from asking Node S.

In a ring (see Figure 6.1), processors can with a little communication store a finger table of successors with exponential gaps. Now if Node N receives a request for a value belonging to some key, the request is immediately put out to all the nodes in its finger table. It can be shown that, if the key is present in the system, the system will report success in time bounded by $O(logn)$ multiplied by the local search time.

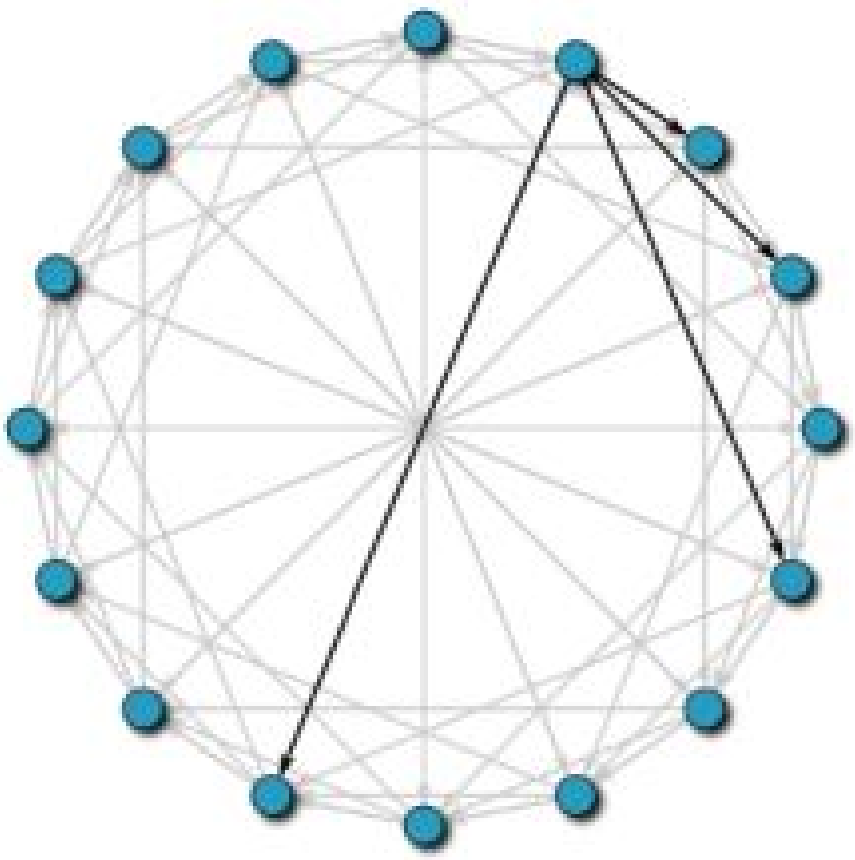


Figure 6.1: ring with finger tables

6.3 Hashing

There are plenty of applications where one has to be able to make certain statements about the similarity of two or more strings of data, or where one has to be able to map strings of data to records for future retrieval. For most of those applications the use of hashing functions is almost inevitable. Therefore we will be looking into the particular subject of hashing functions, or more specifically we will be looking at:

- what properties a hashing function should have in order to be considered a good hashing function;
- how hashing functions are constructed in order to fulfill the conditions of those properties;
- how to test whether a hashing function does actually satisfy those criteria;
- and finally, what hashing functions are commonly used and how well do they meet those requirements.

6.3.1 Properties

Formally, a hashing function maps an input consisting of a variable amount n of boolean states $\{s_1 \dots s_n\}$ to an output consisting of a fixed amount m of boolean states $\{d_1 \dots d_m\}$.

Furthermore, the pigeonhole principle states that if n items are put into m pigeonholes with $n > m$, then at least one pigeonhole must contain more than one item. Similarly, for n distinct input values that are to be mapped to their respective output values where there are m distinct output values, and where $n > m$, there must be at least one output value for which there is more than one input value that will be mapped to it. This shows the concept of hashing collisions.

However, if for the previous case $n <= m$, then it is possible to construct a perfect hashing function for any specific set of input data, i.e., a hashing function that will map the input values in such a way that no single one of them maps to the same output value.

Considering that hashing collisions are inevitable in almost all cases, but the most specialized, the primary goal when developing a hashing function is to minimize the amount of collisions. In order to achieve this, a hashing function will distribute its input values in a uniform manner so that they will be scattered over the output space, similar to a pseudo-random number generator. In addition to the uniform distribution, it is also important that a hashing function doesn't neglect any information in the input in order to achieve this particular goal.

However, unlike a PRNG the process of a hashing function has to be deterministic, i.e., a hashing function should always return the same output value for a given input value.

Furthermore, it can be said that a hashing function reaches avalanche when toggling a single bit state in the input value results in a completely different output value. Even though the concept of avalanche is fundamental for cryptographic hashing functions, it is also a useful property for hashing functions in general. Another concept is the idea that hashing functions shouldn't be invertible, i.e., given the output value it should be either impossible or at least improbable to derive the input value from that.

Finally, a hashing function should be easy to compute to serve its full purpose. Having that said, we can conclude that the following properties are going to be important for a hashing function to be useful database systems:

- A hashing function should be deterministic.
- A hashing function should attempt to distribute its input uniformly, i.e., it should attempt to minimize the amount of collisions.
- A hashing function should use all of the information provided in the input, so that there is no bias.
- A hashing function should be easy to compute.

Nevertheless, it should be noted that these properties are not useful for every kind of application. For instance, on the one hand an important property for locality-sensitive hashing is that the more similar to the input values are, the more likely it is for collisions to occur. On the other hand, collisions are disastrous for cryptographic hash functions where it should be easy to compute the hash of a given a message; but where it should be infeasible to generate a message given a hash value, change the message without changing the hash value, and to find two different messages with the exact same hash value.

6.3.2 Understanding hashing functions

In order to grasp hashing functions, we will be looking at their general structure:

- they will use one or more registers to store the current state, henceforth it will be named the internal state of the hashing function;
- they will use a sliding window to select a part of the input sequence that will properly fit into those registers;
- and they will use two common operations, namely:
  - combining the sliding window with the register(s).
  - permuting the internal state.

The prior operation, combining the sliding window with the register(s), is rather straightforward. There are two candidates as for what operator to apply to both the sliding window and the register(s), namely: addition and exclusive or, where addition is essentially the same as the exclusive or operator with the difference being that addition propagates the overflow bit, whereas this is completely being neglected in the case of exclusive or.

The rationale behind using exclusive or and addition is that, given two boolean states for which it is known that for each of those states the probability for it to be true is the same as the probability for it to be false, the result will also have the same characteristic, i.e., the operation itself is unbiased.

With this knowledge we can construct both the algorithm for additive hashing, and the algorithm for XOR-hashing, as illustrated below:

```
function additive_hash(input):
    hash = 0

    for byte in input:
        hash = hash + byte

    return hash
```

```
function xor_hash(input):
    hash = 0

    for byte in input:
        hash = hash ⊕ byte

    return hash
```

However, there is an important observation that should be made concerning these two hashing algorithms: both addition and exclusive or are commutative operations. Therefore hashing "abc" will yield the same result as hashing "cba", which does not fulfil one of the properties we have talked about before, namely that the amount of collisions should be minimised.

Another less direct observation that should be made is that for specific kinds of data such as ASCII-encoded strings, the bytes tend to follow common patterns that can be split up into a set of static, or biased, states, and a set of dynamic, or random, states.

With those observations in mind, it is now time to look at the latter operation, which is all about permuting the internal state. There are various ways to get this done, such as bit rotation, multiplication, and, in the case of multiple registers, changing the order of the registers.

6.4 Vector clocks

6.4.1 Why do we need vector clocks

In distributed databases nodes might not actually have the most recent version of information, or worse, if a low level of consistency was chosen in the design, may actually contain inconsistent information. Which means that while a revision of the database item might be newer then another, if they aren't causally [1] related, they are actually representing two conflicting version of history of the item.

This introduces two problems when we actually want to query data on such a set of nodes.

1. Which version is the most recent
2. How do we resolve a conflict

The rest of this is written independent of the fact that enforcing, $\lfloor \frac{(\texttt{nodes})}{2} \rfloor + 1$ being involved in operations would avoid many issues, but not entirely.

Why not use wall clock time stamps?

Given just the first problem, the easy solution would be to use the current wall clock time. This however doesn't work in practice, mainly because of the fact that accurately synchronizing time across a cluster of nodes is not reliable.

Even if you'd use a perfect solution it is still possible that a transaction on two nodes happens at the exact same time. This would still make it impossible to establish the causal relation between these two events.

So using just time stamps is not the answer (in practice they do help solve some practical issues with implementing vector clocks)

Why not use Lamport timestamps?

Lamport time stamps are a conceptually simpler model then vector clocks and simply consist of a counter internal to each item or a single global counter. It becomes obvious that neither model can work in practice as any node can increment the counter in an item to the exact

[1] i.e., in a cause and effect relation

same value as another one could. This would make it impossible to detect that there is a conflict.

While with a single global counter (or a broker permitting access to the per item counter) would solve this problem, it also introduces a new one, namely we now have a single point of failure. So this isn't a good solution either (although such a model is used in a few distributed file systems).

So why vector clocks?

Vector clocks work around these problems by essentially distributing the clock itself. With vector clocks it is easy to determine both if two vectors are causal and in case they are causal, which preceded the other.

It does this by maintaining a Lamport clock per client, which is then used to update the time per item.

6.4.2 Vector clocks

So now to define the vector clock in proper terms. Vector clocks are made up of vectors in an essentially linear algebra way. So the following is the case.

$VC(x)$ is the vector clock for event x where $VC(x)_z$ is the element that belongs to node with the identifier z.

Each of the element in the clock represents a client that has done an event (has modified) the object the vector clock refers to. So one has to take into account there is an infinite number of possible clients, yet a limited number might be in the vector because most of them have not yet modified it. In practice this also means that we need to use a unique identifier (for example a UUID) for each client.

Next the successor $VC(x+1)$ for a given $VC(x)$ is created. Which luckily is the obvious case:

$$\bigcup VC(x)_z + 1$$

Now to actually establish an ordering we need to check two things, namely, what is the temporal ordering of the vector clock and are the two vector clocks actually causal. Where the actual temporal ordering only makes sense when we establish that they are causal.

While the temporal ordering is relatively simple.

$$VC(x) < VC(y) \iff \forall z[VC(x)_z \leq VC(y)_z]\exists z'[VC(x)_{z'} < VC(y)_{z'}]$$

Or in English, each element of $VC(y)$ is equal or lager than the same element in $VC(x)$ and atleast one of them is actually larger in $VC(y)$. This is also equivalent to $|VC(x)| < |VC(y)|$ obviously.

Establishing which clocks are causal is slightly harder however. Two clocks are causal when one clock leads into the other with simple operations of creating successors. While this is a suitable formal definition, it is time intensive to check it that way.

So a simpler way of stating it is based on the fact that the difference between the two vectors should contain only negative or positive elements (with 0 considered to be both). As this difference is then a simple set of successor or predecessor statements. So this is gives us the expression.

$$\text{causal}(VC(x), VC(y)) \iff \forall z VC(x)_z - VC(y)_z \geq 0 \vee \forall z VC(x)_z - VC(y)_z \leq 0$$

Then there is a final possible action we might want to do on vector clocks. While knowing we have two non-causal vector clocks is useful and there is no automatic merging possible on them, we can however with some more knowledge or by asking the user, merge them, in which case we need a new $VC(z)$ which has the property of $VC(z) > VC(x) \wedge VC(z) > VC(y) \neg \text{causal}(VC(x), VC(y))$, which is something we can construct.

$$\begin{aligned} &\forall w \\ &VC(x)_w \geq VC(y)_w \rightarrow VC(z)_w = VC(x)_w. \\ &VC(x)_w \leq VC(y)_w \rightarrow VC(z)_w = VC(y)_w \\ &VC(z)_n + 1 \text{ for client } n \end{aligned} \tag{6.1}$$

This establishes all the mathematical definitions and constructions that are necessary to build vector clocks that are usable in a simple databases. However there are some further issues that need to be dealt with.

6.4.3 Real world vector clocks

Vector clocks in RIAK

Let us briefly look at the implementation of vector clocks in RIAK, which could be an inspiration for the implementation of vector clocks in our project later on. We copy the example from the little Riak book mentioned earlier. Suppose we have three people trying to reach consensus on their favorite food: Aaron, Britney, and Carrie. These people are called *actors*,i.e., the things responsible for the updates. We will track the value each actor has chosen along with the relevant vector clock.

When Aaron sets the `favorite` object to `pizza`, a vector clock could contain his name and the number of updates he's performed.

noindent`bucket: food`
`key: favorite`
`vclock: Aaron: 1`
`value: pizza`

Britney now comes along and reads `favorite`, but decides to update `pizza` to `cold pizza`. When using vclocks, she must provide the vclock returned from the request whe wants to update. This is how RIAK can help ensure you're updating a previous value, and not merely overwriting with your own.

```
bucket: food
key: favorite
vclock: Aaron: 1, Britney:1
value: cold pizza
```

At the same time as Britney, Carrie decides that pizza was a terrible choice, and tried to change the value to `lasagna`.

```
bucket: food
key: favorite
vclock: Aaron:1, Carrie: 1
value: lasagna
```

This presents a problem, because there are now two vector clocks in play that diverge from `Aaron: 1`. By default, RIAK will store both values.

Later in the day Britney checks again, but this time, she gets the two conflicting values (aka siblings) with two vclocks.

```
   bucket:food
key:favorite
vclock:  Aaron:  1, Britney:  1
value:  cold pizza
---
vclock:  Aaron:  1, Carrie:  1
value:  lasagna
```

It's clear that a decision must be made. Perhaps Britney knows that Aaron's original request was for `pizza`, and thus two people generally agreed on pizza, so she resolves the conflict choosing that and providing a new vclock.

```
bucket:food key:favorite vclock:    Aaron:1, Carrie:1, Britney:2 value:  pizza
```

Now we are back to the simple case, where requesting the value of `favorite` will just return the agreed upon `pizza`.

Some small remaining details

As you might imagine the size of a vector clock, with a field for each client and incremented to infinity, would grow to ridiculous sizes. So in real world implementations [2] the vector clock is pruned usually using a time stamp to prune old vector components.

For more details on this, see the references for this section.

[2] Not your own for this project, don't worry

Bibliography

[JG04] J.Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. research.google.com/archive/mapreduce.html, December 2004.

[Pap79] C.H. Papadimitriou. The serializability of concurrent database updates. *Journal of the ACM*, 26(4):631–653, 1979.

[RW12] E. Redmond and J.R. Wilson. *Seven Databases in Seven Weeks*. Pragmatic Bookshelf, 2012.

[SKS06] A. Silberschatz, H.F. Korth, and S. Sudarshan. *Database System Concepts*. Mc Graw-Hill, 5th edition, 2006.

[SMK+01] I. Stoica, R. Morris, D. Karger, M.F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. *ACM SIGCOMM Computer Communication Review*, 31(4):149–160, 2001.