

# CouchDB IN ACTION

Chris Chandler

Unedited Draft

MEAP



 MANNING



**MEAP Edition  
Manning Early Access Program**

Copyright 2009 Manning Publications

For more information on this and other Manning titles go to  
[www.manning.com](http://www.manning.com)

# *Contents*

Preface

## **Part I: Getting started with CouchDB**

Chapter 1: Hello CouchDB

Chapter 2: Creating your first database and document and becoming familiar with Futon

Chapter 3: Getting more familiar with documents and storing data

## **Part II: Couch core ideas**

Chapter 4: Writing Map Functions

Chapter 5: Map/Reduce Mapping: Searching through documents

Chapter 6: Map/Reduce Reducing: Making all those search results meaningful

## **Part III: Advanced CouchDB**

Chapter 7: Key collation or "don't call them joins": Combining complex records

Chapter 8: Paging through data

Chapter 9: Replicating data for redundancy and parallel processing

Chapter 10 Reporting style use

Chapter 11: Example app converting a paper process to an electronic one

Chapter 12: Pragmatic CouchDB on the way

Appendix A: Installing CouchDB

Appendix B: JavaScript basics

---

## List of Figures

1.1. REST vs Binary Protocol .....	3
1.2. Basic SQL table .....	4
1.3. Referring to a document unambiguously .....	4
1.4. Requests to CouchDB .....	5
1.5. Requests being reverse proxied .....	6
1.6. Structure of CouchDB documents .....	8
1.7. Map function as a funnel .....	11
1.8. Reduce as a funnel that takes a view .....	14
2.1. Futon .....	18
2.2. The configuration screen .....	18
2.3. Replication interface .....	19
2.4. Creating a database .....	21
2.5. Database view in Futon .....	22
2.6. New document view .....	23
2.7. Saved document with new name field .....	24
3.1. Class diagram of models .....	32
3.2. SQL table .....	33
3.3. one-to-one table .....	34
3.4. one-to-one w/ different cardinality .....	35
3.5. one-to-many 2 tables .....	36
3.6. Many to many .....	39
3.7. Many to many example with friends .....	40
4.1. Document to Map function to View .....	53
4.2. Non-unique key behavior .....	55
4.3. Screenshot of Futon's temporary view interface .....	60

---

## List of Tables

2.1. Configuration options .....	19
2.2. Replication results .....	20
2.3. Database information .....	21
2.4. Intrinsic document values .....	23
3.1. Document validation function parameters .....	44
4.1. Metadata from view in response .....	58
4.2. View/query parameters .....	59

---

## List of Examples

1.1. First document .....	9
1.2. Adding keys .....	9
1.3. Documents with arrays .....	9
1.4. Documents with hashes .....	9
1.5. Documents with complex mix of hashes and arrays .....	10
1.6. Map function document .....	12
1.7. ....	12
1.8. ....	13
1.9. Skeleton of a reduce function .....	14
1.10. Map for reduce .....	15
1.11. Reduce counting users .....	15
2.1. Replication JSON response .....	20
2.2. Example JSON output of database statistics .....	21
2.3. Curl comand for creating a document .....	24
2.4. CouchDB response for document creation .....	24
2.5. Curl command for creating a document with a nested hash .....	25
2.6. CouchDB JSON response to creating the nested hash .....	25
2.7. Curl command to read a document .....	25
2.8. CouchDB JSON resposne to reading the nested hash document .....	25
3.1. Creating another document .....	28
3.2. Getting a document from CouchDB .....	28
3.3. Document with email added .....	29
3.4. Updating the document .....	29
3.5. Deleting a document .....	29
3.6. Retrieving a deleted document .....	30
3.7. Retrieving a previous version .....	30
3.8. Updating a document after deletion .....	31
3.9. Example document without a type specifier .....	31
3.10. Example Person document with type field .....	32
3.11. Example modeling single table inheritance like behavior .....	33
3.12. Example of two documents needing to be merged .....	37
3.13. Example document of merged one-to-many with type on the sub document .....	38
3.14. Example document of merged one-to-many relationship with specific keys .....	38
3.15. Example many-to-many document skeleton .....	41
3.16. Example document with a many-to-many relationship .....	41
3.17. Basic design document with a view .....	42
3.18. Design document with multiple views .....	42
3.19. User document for validation .....	43
3.20. Skeleton of a validation function .....	44
3.21. Basic validation of keys and content .....	44
3.22. Contextual validation .....	45
3.23. Document validation: Verifying changing attributes .....	46
3.24. Complete validation example with security .....	47
3.25. Curl example to store a binary attachment .....	48
3.26. Curl example to store a binary attachment with a specified content header .....	48
3.27. Curl example to read a document with a binary attachment .....	48
3.28. Curl example to retrieve a binary attachment .....	49
3.29. Curl example for updating binary attachment .....	49
3.30. Curl example for deleting a binary attachment .....	49
4.1. User document for map examples .....	52
4.2. ....	52

4.3. ....	52
4.4. Map function for non-unique keys .....	56
4.5. Map function for non-unique keys returning usernames .....	56
4.6. Map function for non-unique semantic key returning full documents .....	57
4.7. Creating a design document via API .....	57
4.8. View JSON response .....	58
4.9. Passing a map function to the temporary view API .....	60
4.10. Example document for paging .....	61
4.11. ....	61
4.12. Generic document for complex keys .....	62
4.13. Map function with complex keys .....	62
4.14. Querying ranges of complex keys .....	63
B.1. Basics of if/else .....	67
B.2. Javascript loops .....	68
B.3. Examples of working with JSON .....	68

---

# Chapter 1. Hello CouchDB

- A database to address Internet needs
- RESTful access to information
- Document-oriented data for easier organization
- Map/Reduce as a means of accessing data

"The current database won't scale to meet that need." This sentence has been uttered in businesses the world over to describe what happens when either the current architecture becomes insufficient, or when the "big iron" SQL database in the basement has finally reached capacity. In an era of media driven web applications, high concurrency, and global availability the time has come for us to rethink the way in which we manage our vast stores of data.

Considerable research and effort has gone into developing the next stage of data storage options. Toward the top of this crowd is the big Giant google themselves. Several years ago they invested time and effort to figure out how to scale their data operations. <sup>1</sup>The result of that effort was BigTable. BigTable was the product of considerable research establishing that the way in which we utilize data for most Internet applications is not the same original set of assumptions that went into the original SQL designed databases of the 70's, and 80's. Google researchers realized that for most, but not all, applications we don't require most multi-record locking mechanisms, and that key integrity was easy to manage from an application stand point. By designing a very simple columnar data system they were able to grow their data operation to previously unheard of levels.

But how does BigTable relate to CouchDB? Several years ago Damien Katz, the original creator of CouchDB and former senior developer of Lotus Notes, saw the potential to take his experience with Lotus Notes, a document management framework, and what Google had accomplished with BigTable's MapReduce style data and create a beautiful fusion of the two. His background working on document-oriented data systems like Lotus Notes made a perfect segue into writing a database for the Internet that could merge the best features of both together. A schema-less data system that leveraged the distributed computability of MapReduce.

In this first chapter we are going to explore the core features and motivations behind CouchDB, and more importantly, what advantages it will confer to your IT department. The first, and potentially most salient, point to start with is: so what makes it a database for the Internet?

## What makes CouchDB a database for the Internet?

From its inception, CouchDB was designed to be a new type of database to meet the growing challenges of database development. Today, web application development is not only one of the hottest segments of the IT industry in general, but it's also the principal focus of this book when discussing CouchDB's role in the Internet ecosystem. However, saying it's a database for the Internet means we need to explore two principal concerns of Internet systems architecture: connectivity and redundancy. Firstly, let's explore how amazingly simple it is to interface with CouchDB.

---

<sup>1</sup>Fay Chang, Jeffrey Dean, Sanjay Ghemawat, et al. "Bigtable: A Distributed Storage System for Structured Data". OSDI 2006

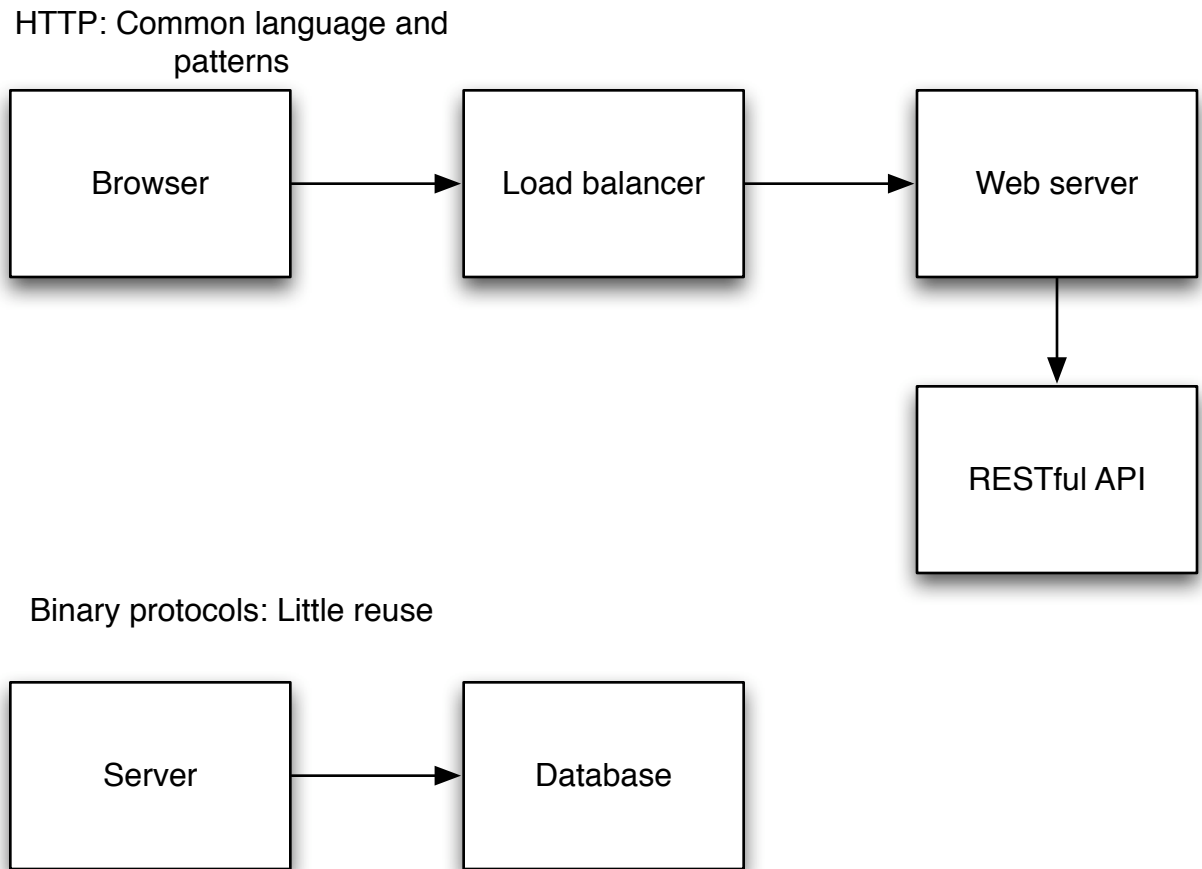
## Connecting to the database with REST

The magical term today in web architecture is REST. REST is an acronym for representational state transfer; which essentially means that we can describe "resources" in terms of 4 fundamental actions that are expressible in terms of HTTP verbs. These verbs are:

- GET
- POST
- PUT
- DELETE

On the surface this probably seems like a fairly simple explanation. The power behind this system is that it allows us to leverage the phenomenal amount of HTTP infrastructure already available that has made the Internet the success that it is. Rather than design a proprietary binary protocol, or a special set of XML based protocols, we can define a communications model that any client that understands HTTP can interact with.

The secondary advantage to a database that speaks the language of HTTP for communication is that it creates a platform that is language agnostic. Today, in the enterprise, it is not uncommon to encounter IT development shops that employ a broad number of languages. While databases like MySQL have done a great job making drivers available for a multitude of platforms, there's still a considerable gap between the supported features on one language's platform versus another because the drivers are maintained along their own development path and lifecycle. The second step CouchDB has made toward language agnosticism is observing that all incoming data on the REST interface will be treated as an uninterpolated string. Rather than have the database worry about the correct typing of the data, we'll let the application layer be principally concerned with this task.

**Figure 1.1. REST vs Binary Protocol**

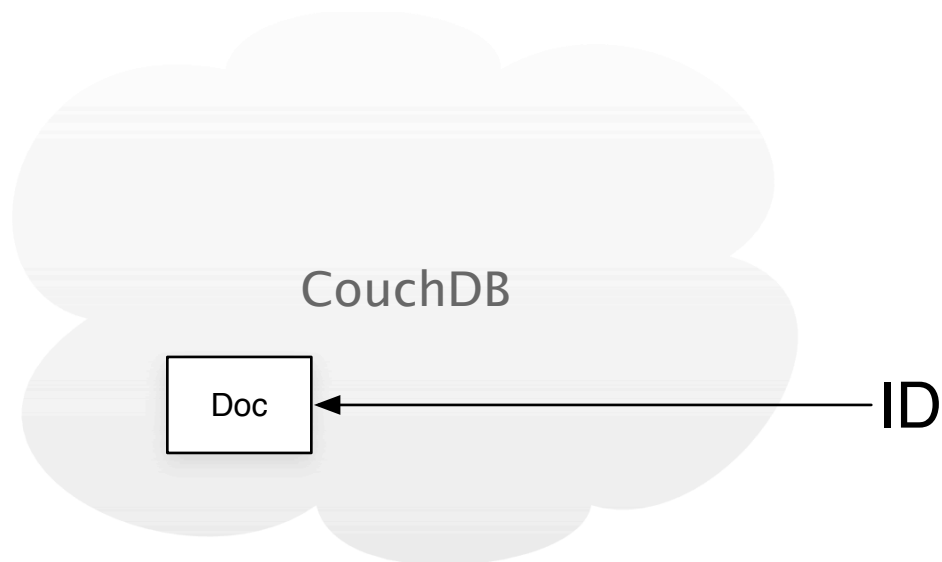
Now that we've introduced some of the language agnostic capabilities of the REST interface, let's explore the RESTful model from a systems architecture standpoint and how we should think about the individual components that operate from within the database.

This RESTful way of referring to components makes even more sense when we start to think of database records as unique objects within a system. The most effective way to see this illustrated is to compare it with the SQL notion of a primary key. The principal function of a primary key in SQL database is to uniquely identify a singular record within a table space, as illustrated in Figure 1.2, "Basic SQL table". In CouchDB we have a similar concept that we just uniformly refer to as the document id. However, the function of this key in a SQL database is mostly to support third normal form, an idea that we will cover in depth later in chapter 3 when it comes to denormalization. If these terms don't make a lot of sense yet, that's fine, we'll cover them in depth. Let's look at this though as if we were designing a web application and we wanted to refer to a specific record within our system.

**Figure 1.2. Basic SQL table**

ID	Dimension 1	Dimension 2	Dimension 3	Dimension 4
1				
n				

If we knew that we wanted the first record we could craft a SQL statement that would `SELECT` the contents of that row and return it, but how would we uniquely refer to that one single record? The short answer is that we most likely cannot refer to that single record without also referring to the table that contains it. CouchDB on the other hand, similar to Google's BigTable, just has one unified table space (though it's not really a table space per se). Because all documents within a database all coexist in one large data space, we can refer to a single record unambiguously through its document ID as illustrated in Figure 1.3, "Referring to a document unambiguously".

**Figure 1.3. Referring to a document unambiguously**

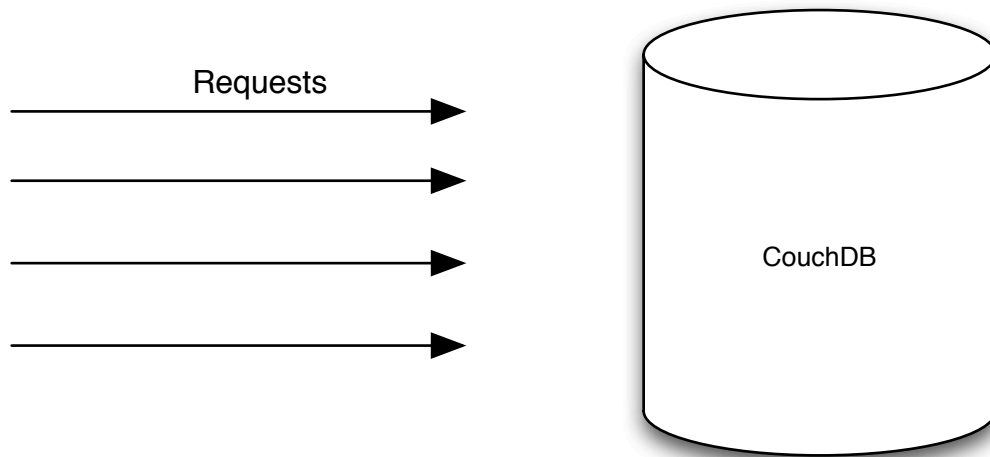
This unambiguous reference takes the form of an HTTP URI like the following:

`http://dbaddress:5984/database/identifier`

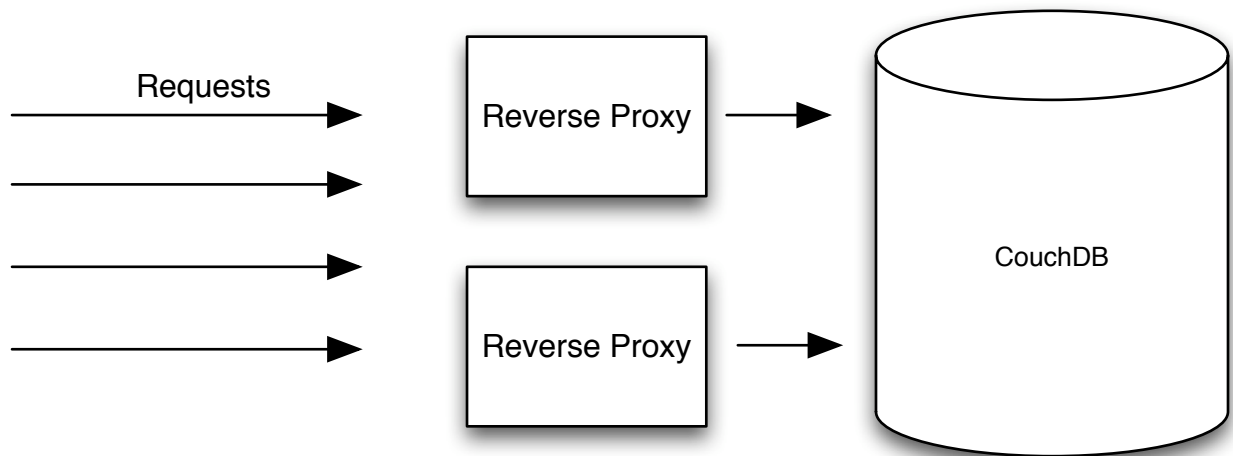
The consistency of this model confers full functionality to any client that speaks the language of HTTP. We can, and will, write a client that can access database information simply by building calls to `curl`. The client logic expressed in ruby will be the same client logic expressed in Erlang. All that changes is the language we use to express the same common set of semantics.

The advantages of writing a client that understands REST should be clear, but that is not the end of the advantages of this type of system. Not only does it make client code easier to create but it also allows us to leverage a considerable amount of power from our IT infrastructure. We are also going to explore advantages to a RESTfully accessible database in terms of operations and systems management.

**Figure 1.4. Requests to CouchDB**



To really illustrate the power of an HTTP/RESTful based system imagine a scenario in which our database accesses skyrocket shown in Figure 1.4, “Requests to CouchDB”. Our popular application is now being hit by social networks, users, and mobile devices and our hardware resources are almost fully saturated. It makes sense that we would need to alleviate this system load through caching. With binary clients this change is almost certainly a modification to the application code. Depending on the complexity of the optimizations being made, this could include a fairly costly set of modifications. Because we know that CouchDB uses the same HTTP protocol that other web infrastructure uses we can leverage a reverse proxy system to sit as a cache in between our application servers and our databases. By adding a product like squid in place we have an actual "drop-in" solution. Imagine being able to alleviate system load by altering your infrastructure, rather than modifying your code to suit the situation. This idea is shown in Figure 1.5, “Requests being reverse proxied”

**Figure 1.5. Requests being reverse proxied**

Another unique implication of a completely RESTful/HTTP driven system is that we can actually write applications entirely in Javascript that resides on the CouchDB server itself. This special classification of applications, called couchapps, is under active development. Since javascript is capable of issuing requests back to the machine of origin it is like running an application directly on the database. In special situations where the information is truly light-weight (and trusted!) a tremendous cost savings could be realized by literally combining the application with the database.

All this talk about RESTful advantages would be incomplete without discussing the implications on the security model. In a traditional HTTP server authentication is enforced by means of returning an HTTP 401 error message and asking the client to return a username/password in order to continue. It turns out that this same system works amazingly well for CouchDB. The same squid example from above can easily be reworked to include an authentication step. To add authentication to this system we merely have to instruct squid to only allow authorized access to the backing CouchDB instance. To carry the example even one step further, we could continue the squid proxies to use LDAP authentication, in which case we take advantage of our currently available authentication infrastructure without having to modify either our application or CouchDB itself!

I will do my best throughout this book not to continuously make a joke about how all this RESTful discussion makes me want to take a nap. But joking aside, now that we have walked through a discussion of REST it makes sense that we should walk through the data structures that we are actually communicating about. A phenomenally simple communications scheme only makes sense if we have something specific to talk about.

## Replication in CouchDB for high availability

If the first principal concern of Internet databases is connectivity, then the second is very likely replication. I was always a fan of the expression " 'Robust' means never having to say 'I am sorry'." Simply put, the database needs to be fault tolerant, and luckily CouchDB has such safeguards built in.

CouchDB supports true bi-directional replication of changes from one database to another via the same REST interface that clients would use to communicate with the database. Because CouchDB uses an internal MVCC (multi-version concurrency control system) document updates are tracked, change by change,

on a particular node and then the changes are replicated to the others in the cluster. Since the replication process is also incremental, if something stops the current replication from completing, then CouchDB can easily recover replication in the next round exactly from where it left off.

CouchDB's replication support doesn't just include high-availability and redundancy. Some of the more interesting features in CouchDB also include support for offline replications and partial replicas. Because views and data replicate together, it's possible to completely replicate an online database to a laptop or offline server if bandwidth becomes unavailable or because the data will need to be accessed at an offsite location.

\*Example of partial replica function

## Similar but different ideas: object oriented databases

Many of you reading to this point might have already asked: isn't CouchDB just an object-oriented database? The answer is somewhat tricky, but ultimately no. Object oriented databases indeed represent complex data structures very similarly to CouchDB's documents, but the key intrinsic difference is that an object-oriented database is meant to exist as a seamless persistence layer for a specific language. CouchDB was consciously designed to be as language agnostic as possible. If the data is always represented in JSON, then any language that is capable of parsing JSON can access the data. This theme of language agnostic decisions has already come up before though in the case of the RESTful interface. Just as any language can read JSON, any language can also speak HTTP.

### So why focus on CouchDB?

The breadth of available options for data storage today can truly be staggering, so naturally we had to restrict our scope to a single option. We chose to focus on this particular subject not only because it represents an amazing fusion of modern technologies to create a new way of dealing with data, but because it also provides an amazing cross-language and cross-platform solution. The individual parts have been available for a considerable amount of time, but the fact they are now available in a single database option is compelling indeed.

## Working with documents in a document oriented database

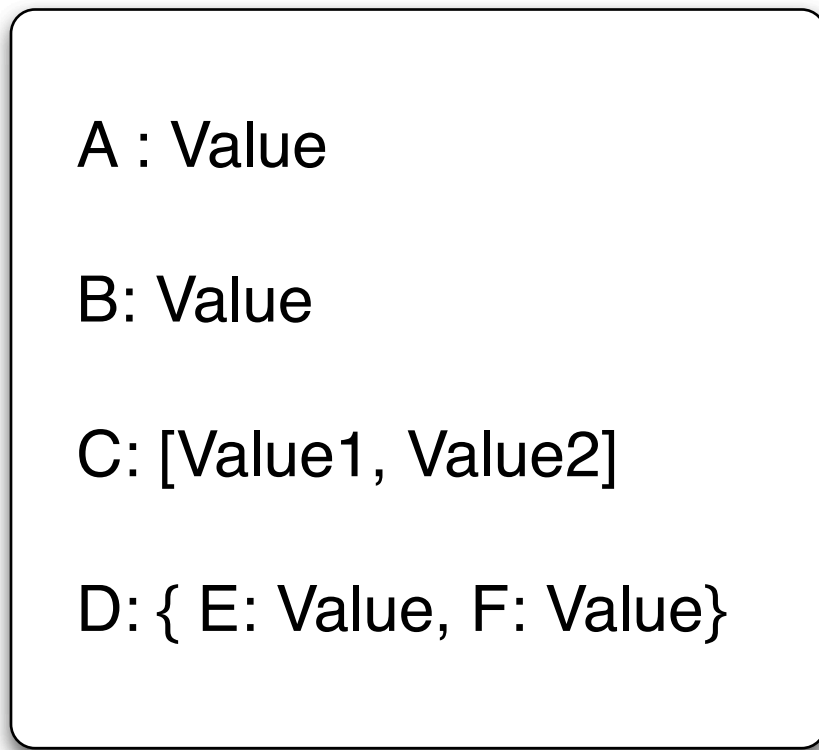
Now for one of the most important parts of introducing a document oriented database: documents! In order to utilize a system like CouchDB it is imperative to understand the fundamental semantics of documents. Luckily, they are quite simple and easy to pickup. In the next two sections we are going to focus on learning to think about data from the perspective of a document, and then explore the specific structure of these documents in terms of CouchDB.

## Keys, not rows - Learning to think about data as documents and not rows

One of the most fundamental differences between SQL based databases and document oriented databases like CouchDB is the fundamental unit of data. In SQL, the fundamental unit of data we deal with is the

row. We write a SQL statement that will select some number of rows from the system and return them so that our application can consume them. In a document oriented database we don't actually have rows to manipulate, we have documents. In the case of CouchDB we can even make one more assertion and say that we have JSON documents. Since JSON can express arbitrarily complex data structures it does not make sense to conceptualize the information in terms of simple rows. Any record in our system is potentially a rich, hierarchical set, of arrays, hashes, and even hashes of hashes, hashes of arrays, etc. ad nauseam (Figure 1.6, "Structure of CouchDB documents").

**Figure 1.6. Structure of CouchDB documents**



This fundamental difference underscores one of CouchDB's most powerful features: schema-less data. Once again, in a traditional SQL system one of the most important decisions the developer is faced with is how to model the application domain. Decisions made in the process of describing the application schema can have very far reaching implications. Very good decisions lead to a manageable and growable schema, bad decisions however create a nightmare of bad code trying to organize that information into something meaningful. One place where this is readily apparent is the act of storing hierarchical data as rows. Does every row contain a complete path to the root? Does every row contain just a reference to its parent? A combination thereof? What if we could just store the hierarchy of information in its usable state so we did not have to decide between heavy application processing to make it meaningful versus N+1 SELECT statements hammering our database to recreate that data in its correct structure.

You might even be asking at this point, "Hasn't this been done?" The short answer is "yes," but a better answer would include the explanation for why some of those ideas never took hold. Sometimes these solutions are referred to as object-oriented databases, since objects are what are being stored. However, CouchDB is not a true object-oriented database for a few key reasons.

## The anatomy of a CouchDB document

Many people have mentioned that CouchDB is a document oriented database, but what does that mean? Perhaps the best way to understand document oriented data is to compare and contrast it with other forms of data storage. To make this easier let's look at a simple document in CouchDB in:

### Example 1.1. First document

```
{
  "my_key" : "my_value"
}
```

This is a very simple example of a CouchDB document represented in JSON form. It shows a singular key named `my_key` that refers to a single value called `my_value`. If we were dealing in terms of a SQL database we might rather think of `my_key` as the name of a column and the `my_value` as a specific row's data. So with SQL we could have many such rows all with differing values, but of a consistent shape and design. This is where document oriented data and the columnar way of thinking start to diverge. CouchDB has no schema like a RDMS. We could just as easily add an additional document to the system with structure:

### Example 1.2. Adding keys

```
{
  "my_key" : "another value",
  "different_key" : "another value still!"
}
```

and the system won't complain because there is no schema to conform with.

The other key advantage of CouchDB's document-oriented data is that we we can store non-trivial hierarchical data in these fields. What would the data for a stored array look like?

### Example 1.3. Documents with arrays

```
{
  "my_array_key" : ["value 1", "value 2", "etc"]
}
```

is all it takes to store an array directly within a document. Rather than construct relationships between different tables to form a "has many" relationship we can simply store the "many" part directly within the document. The same came be said for hash tables

### Example 1.4. Documents with hashes

```
{
  "my_hash_key" : { "internal_key" : "internal_value" }
}
```

This case might seem even more obvious since the outer document itself is really nothing but a hash too. Since we can nest hashes like this it's very possible to create documents of documents in a sense. Or if we really wanted to go all out we could use a mixture of all three at the same time to express virtually any data structure we need

### Example 1.5. Documents with complex mix of hashes and arrays

```
{
  "my_array_key" : [ {"nested_hash" : "nested hash value",
                    "double_array" : ["scalar"]}
  ]
}
```

## Accessing your data via MapReduce

In the previous section we introduced a fairly radical concept when thinking about databases, that idea is that our fundamental unit of data is the document, not the row. In the transition from the relational database world we must learn to think about modeling our data not in terms of SQL statements, but in terms of functions that can work on that data and transform it into a useable state.

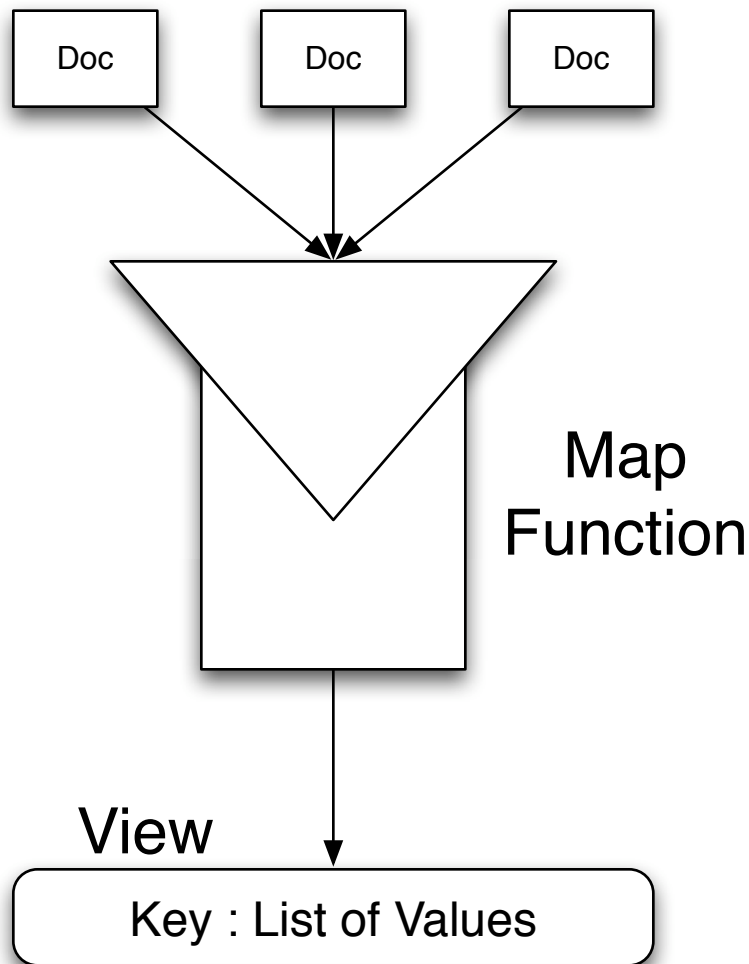
The convention for accessing data held in a SQL RDMS is via SQL, the structured query language. However, CouchDB as discussed, is not a RDMS and thus requires something different from SQL to access and view its data. This is where Map/Reduce comes in.

Map/Reduce, originally, was a pair of high-order parallel functions introduced originally in LISP to partition information and then reconstitute it in an a final processed and useable state. What we are most interested in is the idea of Map/Reduce's partitioning and combination, not the specifics of how it originally worked in LISP.

At this point let's take a quick look at some pragmatic examples of this way of thinking rather than going into the philosophy of Map/Reduce. Let's begin by demonstrating the functionality of Map.

## Accessing data with a Map function

The Map function is the beast responsible for taking in a stack of documents and then "mapping" it into key-value pairs. It may help to think of the Map function as a mathematical function that maps documents into a hash table, shown in Figure 1.7, "Map function as a funnel". The biggest reason this is important is because the Map function requires no information external to the current document that is being processed, thus it is highly parallelizable.

**Figure 1.7. Map function as a funnel**

Consider the following example in SQL. Let's assume we have a table of users with various columns (e.g., username, password, first name, etc).

\* Table of sample data

If we wanted to know all the information for user "Alice" we would write a SQL query similar to the following:

```
SELECT * FROM users WHERE username = "Alice"
```

Now let's translate this data into something that would make sense for CouchDB in Example 1.6, "Map function document".

## Example 1.6. Map function document

```
{ "type": "user",
  "username" : "Alice",
  "password" : "kittycat"
}
```

Now that we are in the world of documents we'll need to translate our previous SQL statement into a Map function to extract the same information. First, let's break apart the SQL statement and actually think about what it does piece by piece in terms of clauses.

The first major clause is the SELECT. If we really think about it, SELECT is responsible for saying what attributes of the table username we are interested in retrieving. In this particular case we are saying SELECT \*, so go ahead and grab the entire row of data. The second major clause is the FROM. Our example is interested in extracting data from the table users, and only the table users (Note: CouchDB doesn't have JOINS per se, instead they called collations and will be covered later). Lastly, is the WHERE clause which expresses a set of criteria to choose which rows from the table in question should be retrieved. These three attributes express the same core idea of a CouchDB Map function: What do we want back?, Where should we find it?, and What qualifies the record for retrieval?.

Let's write a very quick and dirty Map function that will allow us to lookup a user from our user documents based on username. The skeleton of a Map function is just a single Javascript function that takes a single argument of a document (JSON object).

```
function(doc)
{
}
```

Let's start adding in pieces until we have the same functionality as the previous SQL statement. Firstly, let's add in the equivalent of the FROM clause. Since this function will be applied to every document in our system, including non-user documents, we'll need to distinguish which documents we will actually want to pull from. Note that our example above has a field called "type". Since all documents in a database inhabit a single data space, we'll need to only consider documents of type user.

## Example 1.7.

```
function(doc)
{
  if(doc.type == "user")
  {
    //Now we know we have a user document
  }
}
```

Now that we know we are working in the context of a user document, we can put our logic inside that if statement.

Now let's add in the SELECT \* and WHERE username = 'Alice' parts together. Previously I said that Map functions convert documents into a hash table-like structure, and that's what this function will need to do. It will perform this task through a function called emit. The Map function will emit a key-value pair to construct a working view of our data. Emit takes two parameters: 1) the key that should exist in our hash table, and 2) the value that should be associated with that key. In this example we want all data associated with username Alice. So the key in our table will be the criteria we use to lookup the data we're interested in. Thus, we need to emit the user document's username as the key, and the entire document as the value.

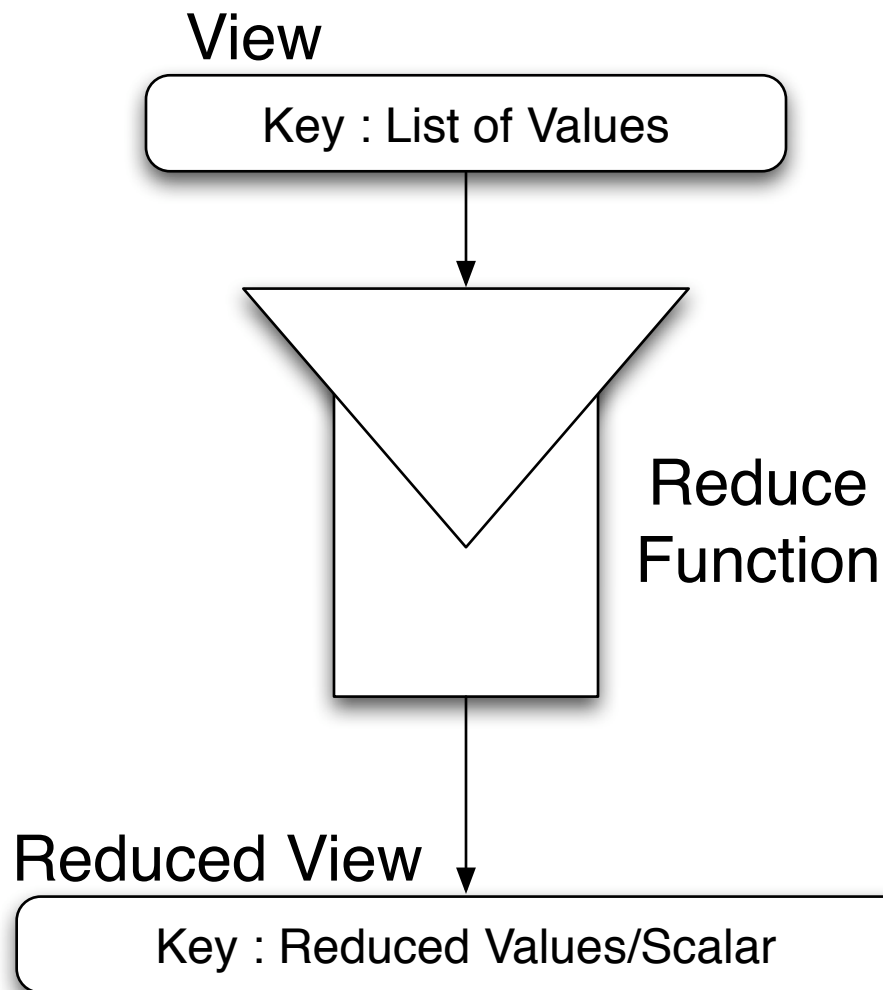
### Example 1.8.

```
function(doc)
{
  if(doc.type == "user")
  {
    emit( doc.username, doc );
  }
}
```

This hash table is really what makes CouchDB powerful. We can now lookup all of our user data via usernames by accessing this view. Now that we have a simple Map example under our belts, let's take a look at how a Reduce function could potentially change this hash table down into a different form of data.

## Making the Map a little more useable with Reduce

Since the Map function was responsible for "mapping" documents into an intermediate hash table result, the Reduce function (Figure 1.8, "Reduce as a funnel that takes a view") is responsible for reducing the values under a particular key in the hash-table down into a smaller result set, potentially a single record.

**Figure 1.8. Reduce as a funnel that takes a view**

The reduce function takes three arguments as in Example 1.9, “Skeleton of a reduce function”

**Example 1.9. Skeleton of a reduce function**

```
function(key, values, rereduce)
{
}
```

For now, let's focus principally on the first two arguments: key and values. Since we know the reduce function works AFTER the map function, this reduce function will process data from the Map function's hash table. Thus, the "key" argument is a singular key from the hash table and "value" is whatever was emitted as the value under that particular key.

The only potential difference between usual hash tables and CouchDB's hash table is what occurs upon a key collision. CouchDB will append a new value onto a previous key creating a linked list style of

behavior. Let's write a quick Map function like Example 1.10, "Map for reduce" that emits non-unique keys so we can illustrate this behavior.

### Example 1.10. Map for reduce

```
function(doc)
{
  if(doc.type == "user")
  {
    emit( 1, doc );
  }
}
```

This Map function will generate a single key in the resulting hash table, but that single key will have a number of values equal to the number of user documents in the system. The immediate question is, why would we need this?

Let's take a look at another SQL comparison. This time let's count the number of user records we have with a SQL statement:

```
SELECT count(*) FROM users
```

Without the count() in the SELECT clause this would return all the user data we have in our system, but we are only interested in a piece of meta-data: the number of records which is a single integer. Now let's switch gears back to our Reduce function and see how our new Map function does something similar.

The reduce function will be called with key = "1" and values = "<array of all user records>" because that's exactly what our Map function was set to generate. The second argument is the key to this puzzle. We want the number of user records in the system, so a count operation is equivalent to determining the size of that array. The reduce can be seen in Example 1.11, "Reduce counting users"

### Example 1.11. Reduce counting users

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

If we run this Map/Reduce pair together we will get a singular integer result back we will see a single hash table with key "1" and a reduced value set equal to the number of user records in our system. If we had 20 user records, then the result would be {"1" : "20"}.

## Summary

CouchDB represents a phenomenal amount of possibilities for the future of data management. As a reasonable alternative to SQL based data storage we have another sophisticated tool in our IT toolbox to address the situations in which 3-normal form data isn't quite the most appropriate solution. The scalability of MapReduce also changes the playing field when we talk about larger data sets and specific scalability issues of Internet-based applications later in this book.

The first major take-away from this chapter should be that CouchDB communicates RESTfully with the rest of the world. Using the HTTP infrastructure that we have all come to know and love we can write clients and interact with CouchDB using whatever language is the most appropriate for the task at hand. By removing the need for a complex binary protocol we can express our data interactions using a composition of simple basic operations.

We also took the first steps to working with Map/Reduce as a means to access your important data. It probably still seems a little strange and foreign, but in coming chapters you will learn to express virtually all the SQL statements you already know and work with into Map/Reduce style functions.

Lastly, was a whirlwind tour of document-oriented data. The ability to store information as an all-together bundle is one of CouchDB's more compelling features. Expressing information hierarchically in single atomic records is one of the major shortcomings of SQL that document-oriented data hopes to solve.

Now that we have introduced the major concepts let's actually get up and running with our own CouchDB instance. In the next chapter we will cover how to get CouchDB installed and how to start creating your first databases.