20 Putting into Practice: COUCHDB, a JSON Semi-Structured Database

This PiP chapter proposes exercises and projects based on COUCHDB, a recent database system which relies on many of the concepts presented so far in this book. In brief:

1. COUCHDB adopts a semi-structured data model, based on the JSON (*JavaScript Object Notation*) format; JSON offers a lightweight alternative to XML;
2. a database in COUCHDB is schema-less: the structure of the JSON documents may vary at will depending on their specific features;
3. in order to cope with the absence of constraint that constitutes the counterpart of this flexibility, COUCHDB proposes an original approach, based on *structured materialized views* that can be produced from document collections;
4. views are defined with the MAPREDUCE paradigm, allowing both a parallel computation and incremental maintenance of their content;
5. finally, the system aspects of COUCHDB illustrate most of the distributed data management techniques covered in the last part of the present book: distribution based on consistent hashing, support for data replication and reconciliation, horizontal scalability, parallel computing, etc.

COUCHDB is representative of the emergence of so-called *key-value store* systems that give up many features of the relational model, including schema, structured querying and consistency guarantees, in favor of flexible data representation, simplicity and scalability. It illustrates the "No[tOnly]SQL" trend with an original and consistent approach to large-scale management of "documents" viewed as autonomous, rich pieces of information that can be managed independently, in contrast with relational databases which take the form of a rich graph of interrelated flat tuples. This chapter will help you to evaluate the pros and cons of such an approach.

We first introduce COUCHDB and develop some of its salient aspects. Exercises and projects follow. As usual, complementary material can be found on the Web site, including JSON data sets extracted from the DBLP source. We also provide an on-line testing environment that lets you play with COUCHDB, insert data and run MAPREDUCE scripts.

20.1 Introduction to the COUCHDB document database

This section is an introduction to the COUCHDB features that will be explored in the project and exercises. We left apart many interesting aspects (e.g., security, load balancing, view management) that fall beyond the scope of this introductory chapter. The presentation successively covers the data model, the definition of views, and data replication and distribution.

20.1.1 JSON, a lightweight semi-structured format

JSON is a simple text format initially designed for serializing Javascript objects. For the record, Javascript is a scripting language (distinct from Java) which is intensively used in Web browsers for "dynamic HTML" applications. In particular, a Javascript function can access and modify the DOM tree of the document displayed by a browser. Any change made to this document is instantaneously reflected in the browser window. This gives a means to react to user's actions without having to request a new page from the server (a development technique know as AJAX), and therefore enables the creation of rich, interactive client-side applications.

Although JSON comes from the Javascript world, the format is language-independent. There exist libraries in all programming languages to read and parse JSON documents, which makes it a simple alternative to XML. This is particularly convenient when persistent data must be tightly integrated in a programming environment because objects can be instantiated from the JSON serialization with minimal programming effort.

Key-value pairs

The basic construct of JSON is a key-value pair of the form *"key": value*. Here is a first example, where the value is a character string:

```
"title": "The Social network"
```

Usual escaping rules apply: the character '"' for instance must be escaped with '\'. Special characters like tabs and newlines are also escaped:

```
"summary": "On a fall night in 2003, Harvard undergrad and computer\n
    programming genius Mark Zuckerberg sits down at his computer\n
    and heatedly begins working on a new idea. (...)"
```

JSON accepts a limited set of basic data types: character strings, integers, floating-point numbers and Booleans (`true` or `false`). Non-string values need not be surrounded by '"'.

```
"year": 2010
```

Complex values: objects and arrays

Complex values are built with two constructors: objects and arrays. An *object* is an unordered set of name/value pairs, separated by commas, and enclosed in braces. The types can be distinct, and a key can only appear once. The following is an object made of three key-value pairs.

```
{"last_name": "Fincher", "first_name": "David", "birth_date": 1962}
```

Since constructors can be nested, an object can be used as the (complex) value component of a key-value construct:

```
"director": {
      "last_name": "Fincher",
      "first_name": "David",
      "birth_date": 1962
  }
```

An array is an ordered collection of values that need not be of the same type (JSON definitely does not care about types). The list of values is enclosed in square brackets []. The following key-value pairs represents a list of actors' names.

```
 "actors": ["Eisenberg", "Mara", "Garfield", "Timberlake"]
```

JSON documents

A *document* is an object. It can be represented with an unbounded nesting of array and object constructs, as shown by the following example which provides a JSON representation of the movie *The Social Network*.

```
{
    "title": "The Social network",
    "year": "2010",
    "genre": "drama",
    "summary": "On a fall night in 2003, Harvard undergrad and computer
    programming genius Mark Zuckerberg sits down at his computer
    and heatedly begins working on a new idea. In a fury of blogging
    and programming, what begins in his dorm room soon becomes a global
    social network and a revolution in communication. A mere six years
    and 500 million friends later, Mark Zuckerberg is the youngest
    billionaire in history... but for this entrepreneur, success leads
    to both personal and legal complications.",
    "country": "USA",
  "director": {
      "last_name": "Fincher",
      "first_name": "David",
      "birth_date": "1962"
  },
  "actors": [
    {
        "first_name": "Jesse",
        "last_name": "Eisenberg",
        "birth_date": "1983",
        "role": "Mark Zuckerberg"
    },
    {
        "first_name": "Rooney",
        "last_name": "Mara",
        "birth_date": "1985",
        "role": "Erica Albright"
    },
```

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

```
    {
        "first_name": "Andrew",
        "last_name": "Garfield",
        "birth_date": "1983",
        "role": "  Eduardo Saverin "
    },
    {
        "first_name": "Justin",
        "last_name": "Timberlake",
        "birth_date": "1981",
        "role": "Sean Parker"
    }
  ]
}
```

To summarize, JSON relies on a simple semi-structured data model, and shares with XML some basic features: data is self-described, encoded independently from any application or system, and the representation supports simple but powerful constructs that allow building arbitrarily complex structures. It is obvious that any JSON document can be converted to an XML document. The opposite is not true, at least if we wish to maintain all the information potentially conveyed by XML syntax. Namespaces are absent from JSON, documents are always encoded in UTF-8, and the language lacks from a built-in support for references (the ID-IDREF mechanism in XML).

Moreover, at the time of writing, there is nothing like a "JSON schema" that could help to declare the structure of a JSON database (some initiatives are in progress: see the references). The attractiveness from JSON comes primarily from its easy integration in a development framework, since a JSON document can directly be instantiated as an object in any programming language. The absence of typing constraint requires some complementary mechanisms to ensure that a JSON database presents a consistent and robust data interface to an application. Documents should be validated before insertion or update, and data access should retrieve documents whose structure is guaranteed to comply to at least some common structural pattern. COUCHDB is an interesting attempt to provide answers to these issues.

20.1.2 COUCHDB, architecture and principles

A COUCHDB instance is based on a Client/Server architecture, where the COUCHDB server handles requests sent by the client, processes the requests on its database(s), and sends an answer (Figure 20.1). Unlike most of the database management systems that define their own, proprietary client-server communication protocol, COUCHDB proposes a REST-based interface. Requests sent by the Client to the server are REST calls, and take actually the form of an HTTP request, together with parameters transmitted with one of the basic HTTP method: `GET`, `POST`, `PUT` and `DELETE`.

It is probably worth recalling at this point the essential features of the HTTP requests that constitute a REST interface. First, we aim at manipulating *resources*, in our case, essentially JSON documents or collections of documents. Second, each resource is referenced by a *Universal Resource Identifier (URI)*, i.e., a character string that uniquely determines how and where we access the resource on the Internet. And, third, we apply *operations* to resources. Operations are defined by the HTTP protocol as follows:

GET retrieves the resource referenced by the URI.

PUT creates the resource at the given URI.

POST sends a message (along with some data) to an existing resource.

DELETE deletes the resource.

The difference between `PUT` and `POST`, often misunderstood, is that `PUT` creates a new resource, whereas `POST` sends some data to an existing resource (typically, a service that processes the data). The difference is harmless for many applications that ignore the specificities of REST operation semantics, and simply communicate with a web server through HTTP. A so-called RESTful service takes care of the meaning of each operation, and deliberately bases its design of the concept of resources manipulation.

This is the case of a COUCHDB server. It implements a REST interface to communicate with the client application. HTTP calls can be encapsulated either by a REST client library, or even expressed directly with a tool like *curl* (see below). The server answers through HTTP, with messages encoded in JSON. Here is a first, very simple communication with an hypothetical COUCHDB server located at, say, `http://mycouch.org`: we send a `GET` request and receive a JSON-encoded acknowledgment message.

```
$ curl -X GET http://mycouch.org
{"couchdb":"Welcome","version":"1.0.1"}
```

A nice feature of the approach is that is quite easy to directly communicate with a server. Keeping in mind the three main REST concepts (resource, URI and operation semantics) helps figuring out the purpose of each request.

The server maintains one or several collections of JSON *documents*. In addition to the JSON structure which constitutes the description of a document d, non-structured files can be attached to d. COUCHDB adds to each document an *id* and a *revision number*. The id of a document is unique in the collection (an error is raised by COUCHDB if one attempts to create a document with an already existing id), and is stored as the value of the `_id` key in the JSON document. In case the value of `_id` is not part of the inserted document, COUCHDB automatically assigns a unique value (a long, obscure character string).

Revisions correspond to the versioning feature of COUCHDB: each update to a document creates a new version, with the same `_id` but a new revision number, represented by the value of a `_rev` key.

A collection in a COUCHDB collection has no schema: a document d_1 with a structure A can cohabit with a document d_2 with a structure B, potentially completely different from A. Basically, this means that the application is in charge of checking the structural constraints before insertion or updates.

COUCHDB proposes some support to solve the problem. First, *validation functions* can be assigned to a collection: any document inserted or updated must be validated by these functions; else the modification request is rejected. This is somewhat equivalent to implementing a specific type-checking function instead of a declarative specification.

Second, COUCHDB allows the definition of *views*. A view is a new key-document collection, specified via a function (actually the specification is based on MAPREDUCE: see below), and organized as a B-tree built on the key. Defining a view is tantamount to a virtual restructuring of the document collection, generating new keys, new documents and new ways of exploring

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

the collection. Since the view is structured as a B-tree, an important advantage is that it supports efficient key and range queries. Views are a means in COUCHDB of presenting a structured and well-organized content to applications.

Finally, the last COUCHDB aspect illustrated by Figure 20.1 is data replication. It is possible to ask a COUCHDB instance to synchronize itself with another instance located anywhere on the Web. This enables a replication mechanism that copies each document to the remote servers. Replication is useful for security (replicated data is safer data) and for scalability. It also gives rise to consistency concerns, since two client applications may modify independently two replicas of a same document. COUCHDB detects update conflicts and reports them, but does not attempt an automatic reconciliation.

These are the basics of COUCHDB principles. Let us now delve into practice.

20.1.3 Preliminaries: set up your COUCHDB environment

From now on, we will guide you through a step-by-step exploration of a few salient features of COUCHDB: creating data and views, replication and distribution. You are invited to download a sample of our *movies* data set, encoded in JSON, from the book web site. You also need an access to a running COUCHDB server. You can set up your own environment (see the site `http://couchdb.apache.org`), or use our on-line environment. Please look at the site for details. In the following, `$COUCHDB` will refer to the IP of the COUCHDB server. In case you would use a Unix console, this variable can be defined with:

```
export COUCHDB=http://<couchIP>:5984
```

where `couchIP` denotes the IP address of the host[1]. In order to communicate with the server, you need a client application that sends HTTP requests to your COUCHDB server. The universal command-line tool to do so is *curl*, which should be available on any Unix-like system. For instance the following command:

```
curl $COUCHDB
```

sends an HTTP request `GET` (the default method) to the server, which should answer:

```
{"couchdb":"Welcome","version":"1.0.1"}
```

Using the option *-v* unveils the details of the HTTP protocol.

```
$ curl -v  $COUCHDB
* About to connect() to xx.xxx.xxx port 5984 (#0)
*   Trying xx.xxx.xxx... connected
* Connected to xx.xxx.xxx (xx.xxx.xxx) port 5984 (#0)
> GET / HTTP/1.1
> User-Agent: curl/7.19.7
> Host: xx.xxx.xxx:5984
> Accept: */*
>
< HTTP/1.1 200 OK
< Server: CouchDB/1.0.1 (Erlang OTP/R13B)
```

[1] If you use our COUCHDB server, you must add you login/password to this IP.

```
< Date: Tue, 09 Nov 2010 08:34:36 GMT
< Content-Type: text/plain;charset=utf-8
< Content-Length: 40
< Cache-Control: must-revalidate
<
{"couchdb":"Welcome","version":"1.0.1"}
* Connection #0 to host xx.xxx.xxx left intact
* Closing connection #0
```

Every interaction with the server can in principle be handled by *curl*. Of course, a graphical interface is more pleasant that a command-line tool. Our site proposes such an interface. If you have your own installation of COUCHDB, Futon is an application, shipped with any COUCHDB environment, that lets you manage your databases. Futon is actually a Javascript application natively stored in a COUCHDB server, so that it works without any further installation step. It can be accessed at the following URL:

```
$COUCHDB/_utils
```

Fig. 20.2 shows a screen copy of the Futon home page. In the following, we will describe the interactions with COUCHDB through the *curl* command line interface. Most of them can also be expressed with Futon.

20.1.4 Adding data

Let us create our first database. We simply send a `PUT` request to the COUCHDB server, asking for the creation of a resource. The following command creates the *movies* database.

```
$ curl -X PUT $COUCHDB/movies
{"ok":true}
```

Now the resource exists at the given URI, and a `GET` request will retrieve some information about the database.

```
$ curl -X GET $COUCHDB/movies
{"db_name":"movies",
 "doc_count":0,
  "doc_del_count":0,
  "update_seq":0,
  "purge_seq":0,
  "compact_running":false,
  "disk_size":79,
  "instance_start_time":"1289290809351647",
  "disk_format_version":5,
  "committed_update_seq":0}
```

That's all: we send HTTP requests to COUCHDB, which answers with a JSON document. COUCHDB offers an "API" which takes the form of REST services whenever appropriate (not all services can be conveniently implemented as REST calls). The `_all_dbs` service for instance returns an array with the list of existing databases.

```
$ curl -X GET $COUCHDB/_all_dbs
["movies","_users"]
```

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

It is time now to add documents. Get the JSON-encoded documents of the movies database from the book's web site. We shall first insert *The Social Network*.

```
$ curl -X PUT $COUCHDB/movies/tsn -d @The_Social_Network.json
{"ok":true,"id":"tsn","rev":"1-db1261d4b2779657875dafbed6c8f5a8"}
```

This deserves some explanations. First, we follow the very same logic as before, asking COUCHDB to create a new resource at URI `$COUCHDB/movies/tsn`. The resource content must be JSON-encoded: we transfer the content of the *The_Social_Network.json* file. Note the `-d` curl option which must be followed by the content of the HTTP request message. Note also that this content can be extracted from a file with the special syntax `@fileName`.

COUCHDB answers with a resource that contains several information. First, `"ok":true` means that the document has been successfully inserted. Then we get the id, and the revision number. The id (or key) of the document is the means by which it can be retrieved from the database. Try the following request:

```
$ curl -X GET $COUCHDB/movies/tsn
```

You should get the document just inserted. In this case, the document id, `tsn` is user-defined. We must be careful to ensure that the id does not already exist in the database. COUCHDB generates an error if we attempt to do so:

```
$ curl -X PUT $COUCHDB/movies/tsn -d @The_Social_Network.json
{"error":"conflict","reason":"Document update conflict."}
```

A *conflict* has been detected. COUCHDB uses an “eventually consistent” transaction model, to be described next.

If we want COUCHDB to generate the id of the document, we must send a `POST` request, along with the content and its MIME encoding. Recall that `POST` sends some data to an existing resource, here the database in charge of inserting the document:

```
$ curl -X POST $COUCHDB/movies -d @The_Social_Network.json \
  -H "Content-Type: application/json"
{"ok":true,
 "id":"bed7271",
 "rev":"1-db126"}
```

A new id has been generated for us by COUCHDB, and the document has been stored as a resource whose URI is determined by this id, e.g., `$COUCHDB/movies/bed7271`.

In order to update a document, you must send a `PUT` request that refers to the modified document by its id *and* its revision number. The multi-version protocol of COUCHDB requires that both values must be given to refer to a document value. The usual update mechanism involves thus (i) getting the document from the database, including its revision number, (ii) modify locally the document and (iii) put back the document to COUCHDB which creates a new version with a new revision id.

Let us show how to update a document by adding an *attachment*. We execute a `PUT` request on the previous movie to associate its poster (a JPEG file). We must provide the file content in the request body, along with the MIME type. The version of the document which is modified is referred to by its revision number. Here is the curl command, which specifies the MIME type of the attachment:

```
$ curl -X PUT $COUCHDB/movies/tsn/poster?rev=1-db1261 -d
@poster-tsn.jpg   -H "Content-Type: image/jpg"
{"ok":true,"id":"tsn","rev":"2-26863"}
```

As a result, a new revision `"2-26863"` has been created. The poster can be retrieved from COUCHDB with the URI `$COUCHDB/movies/tsn/poster`.

Finally, a document can be deleted with the REST `DELETE` command. The revision number must be indicated. Here is an example:

```
$ curl -X DELETE $COUCHDB/movies/tsn?rev=2-26863
{"ok":true,"id":"tsn","rev":"3-48e92b"}
```

A surprising aspect of the result is that a new revision is created! Indeed, the deletion is "logical": old revisions still exist, but the latest one is marked as "deleted", as shown by the following query that attempts to retrieve the current version of `tsn`.

```
$ curl $COUCHDB/movies/tsn
{"error":"not_found","reason":"deleted"}
```

We invite you now to load in your collection the *movies* documents available on our Web site. The following section shows how to query COUCHDB databases with views.

20.1.5 Views

A *view* in COUCHDB is the result of a MAPREDUCE job. The main rationale behind this seemingly odd choice is, first, the ability to run in parallel the evaluation of view queries in a distributed environment, and, second, the incremental maintenance of view results. Both aspects are closely related: because the MAP phase is applied to each document independently, the evaluation process is inherently scalable; and because COUCHDB records any change that affects a document in a collection, view results can be maintained by re-evaluating the MAPREDUCE job only on changed documents.

Views definition are stored in the COUCHDB database as special documents called *design* documents. Temporary views can also be created using the Futon interface which provides a quite convenient tool for interactive view definition and testing. From your favorite browser, access the `$COUCHDB/_utils` URL, move to the *movies* database and select the `temporary views` choice form the `Views` menu. You should obtain the form shown on Figure 20.3.

The form consists of two text windows: the left one (mandatory) for the MAP function, and the right one (optional) for the REDUCE function. Functions are written in Javascript (we use simple examples that are self-explanatory). We begin with a simple MAP function that takes as input a document (i.e., a representation of a movie, see above) and produces a $(key, value)$ pair consisting of the movie title (key) and the movie's director object (value). Write the following text in the left window and press the `Run` button: you should obtain the list of $(title, director)$ pairs shown on Figure 20.3.

```
function(doc)
{
   emit(doc.title, doc.director)
}
```

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

A MAP function always takes as input a document of the collection. The reader is referred to Chapter 16 for a detailed description of the MAPREDUCE parallel computation principles. Essentially, the point is that the above function can be spread over all the nodes that participate to the storage of a COUCHDB database, and run on the local fragment of the database.

Here is a second example that shows how one can create a view that produces a list of actors (the key) along with the movie they play in (the value). Note that, for each document, several pairs are produced by the MAP function

```
function(doc)
{
   for each (actor in doc.actors) {
      emit({"fn": actor.first_name, "ln": actor.last_name}, doc.title) ;
   }
}
```

Note that the key component may consist of a complex JSON object. From Futon, save the first function as "`director`" and the second one as "`actors`" in a design document called ("`examples`"). The views are now stored in the *movies* database and can be queried from the REST interface, as shown below:

```
$ curl $COUCHDB/movies/_design/examples/_view/actors
{"total_rows":16,"offset":0,
"rows":[
 {"id":"bed7271399fdd7f35a7ac767ba00042e",
   "key":{"fn":"Andrew","ln":"Garfield"},"value":"The Social network"},
 {"id":"91631ba78718b622e75cc34df8000747",
   "key":{"fn":"Clint","ln":"Eastwood"},"value":"Unforgiven"},
 {"id":"91631ba78718b622e75cc34df80020d3",
   "key":{"fn":"Ed","ln":"Harris"},"value":"A History of Violence"},
  ...
 {"id":"91631ba78718b622e75cc34df800016c",
   "key":{"fn":"Kirsten","ln":"Dunst"},"value":"Spider-Man"},
  {"id":"91631ba78718b622e75cc34df800028e",
    "key":{"fn":"Kirsten","ln":"Dunst"},"value":"Marie Antoinette"},
 ...
 ]
}
```

Two comments are noteworthy. First, COUCHDB keeps in the view result, for each $(key, value)$ pair, the id of the document from which the pair has been produced. This might be useful for getting additional information if necessary.

Second, you will notice that the result is sorted on the key value. This relates to the underlying MAPREDUCE process: the $(key, value)$ pairs produced by the MAP function are prepared to be merged and aggregated in the REDUCE phase, and this requires an intermediate "shuffle" phase that puts together similar key values. In the above results samples, movies featuring Kirsten Dunst are consecutive in the list.

From this sorted representation, it is easy to derive "reduced" result by applying a REDUCE function. It takes as input a key value k and an array of values v, and returns a pair (k, v') where v' is a new value derived from v and, hopefully, smaller. Here is a first, generic example, that return the number of values associated to a key:

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

Add this REDUCE function to the `actors` view, and compute the result (be sure to set the "reduce" option in Futon, or pass a `group=true` parameter to activate the reduction). You should see indeed with each actor's name the number of movies s/he features in.

```
$ curl $COUCHDB/movies/_design/examples/_view/actors?group=true
{"rows":[
{"key":{"fn":"Andrew","ln":"Garfield"},"value":1},
{"key":{"fn":"Clint","ln":"Eastwood"},"value":1},
...
{"key":{"fn":"Kirsten","ln":"Dunst"},"value":2},
{"key":{"fn":"Maria","ln":"Bello"},"value":1},
{"key":{"fn":"Morgan","ln":"Freeman"},"value":1}
...
]}
```

20.1.6 Querying views

In COUCHDB, views are *materialized*. The MAPREDUCE job is run once, when the view is created, and the view content is maintained incrementally as documents are added, updated or deleted. In addition, this content is a represented as a B-tree which supports efficient search on either key value or ranges. Create a third view, called "`genre`", with the following definition.

```
function(doc)
{
  emit(doc.genre, doc.title) ;
}
```

This is tantamount to issuing the following command in a relational database:

```
create index on movies (genre);
```

Now the database system (whether relational or COUCHDB) can efficiently evaluate a query that refers to the key value. Here is the REST request searching for all documents in `genre` with key value "`Drama`".

```
$ curl $COUCHDB/movies/_design/examples/_view/genre?key=\"Drama\"
{"total_rows":5,"offset":2,"rows":[
{"id":"91631ba78718b622e75cc34df800028e",
  "key":"Drama","value":"Marie Antoinette"},
{"id":"bed7271399fdd7f35a7ac767ba00042e",
  "key":"Drama","value":"The Social network"}
]}
```

Range queries can be expressed by sending two parameters `startkey` and `endkey`.

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

View creation (based on MAPREDUCE) and view querying (based on view materialization and B-tree indexing on the results' key) constitute in essence the solution proposed by COUCHDB to the challenge of satisfying both the flexible data structuring of semi-structured models, and the robust data representation needed by applications. Views provide a means to clean up and organize a collection of documents according to the regular representation expected by application programs. MAP and REDUCE functions act first as filters that check the content of input documents, and second as data structuring tools that create the virtual document representation put as values in a view.

The cost of computing the view representation each time a query is submitted by such a program is avoided thanks to the incremental materialization strategy. The design is also strongly associated to the distribution features of a COUCHDB instance, described next.

20.1.7 Distribution strategies: master-master, master-slave and shared-nothing

Several distribution strategies can be envisaged with COUCHDB. The system does not impose any of them, but rather provides a simple and powerful *replication* functionality which lies at the core of any distribution solution.

Replication

Replication is specified in COUCHDB with a `POST` request sent to the `_replicate` utility. The following example requires a replication from local database *movies* to the local database *backup*. The *continuous* option is necessary to ensure that any future update to a document in *movies* will be reflected in *backup* (otherwise a one-shot replication is made).

```
curl -X POST $COUCHDB/_replicate \
   -d '{"source": "movies",  "target": "backup", "continuous": true}' \
   -H "Content-Type: application/json"
```

Futon proposes actually an interface which makes trivial the specification of a replication. Note that the command defines a one-way copy of the content of a database. Full, symmetric replication can be obtained by submitting a second command inverting the `target` and `source` roles.

You are invited to experiment right away the replication feature: create a second database on one of your available COUCHDB server, and replicate your *movies* database there. You should be able to verify that the content of *movies* can be found in the replica, as well as any subsequent change. Replication is basically useful for security purposes, as it represents a backup of the database (preferably on a remote server). It also serves as a basic service of the distribution options, presented next.

Distribution options

A first distribution strategy, called *master-slave*, is illustrated on Figure 20.4, left part (refer also to the introduction given in Chapter 14). It relies on a *Master* server and one or several *slaves* (for the sake of simplicity we illustrate the ideas with a 2-machines scenario, but the extension to any number of participants is straightforward). The master receives all write requests of the form $w(d)$ by Clients. A replication service at the Master's site monitors all

the writes and replicates them to the slave(s). Replication in COUCHDB is asynchronous: the Client does not have to wait for the completion of the write on the slave.

Read requests, on the other hand, can be served either by the Master or by the slave. This approach avoids inconsistencies, because writes are handled by a single process, and therefore implicitly serialized. On the other hand, it may happen that a Client issues a $w(d)$ to the Master, then a read $r(d)$ to the slave, and receives an outdated version of d because the replication has not yet been carried out. The system is said to be *eventually consistent* (see, again, Chapter 14).

Remark 20.1.1 *Recall that "Client" in our terminology refers to any software component in charge of communicating with the distributed storage system. It may take the form of a library incorporated in the client application, of a proxy that receives network requests, etc.*

A second strategy, called *master-master*, allows write operations to take place at any node of the distributed system. So, each server plays the role of a "Master", as defined above, and the replication now works both sides. In a cluster with n machines, each COUCHDB servers replicates its write request to the $n-1$ other nodes. This avoids the bottleneck of sending writes to a single machine, but raises consistency issues. It may happen that a same document d is modified concurrently on two distinct sites S_1 and S_2, thereby creating two *conflicting* versions.

Conflict management

When a replication is attempted from, say, S_1 to S_2, COUCHDB detects the conflict. The detection is based on a classical transaction protocol called Multi-Versions Concurrency Control (MVCC) that relies heavily on the revision numbers. The protocol is simple and easily understood from an example (summarized in Figure 20.5). Assume a document d with revision number r, denoted $d_{(r)}$. This document is replicated on S_1 and S_2, and each replica is going to be modified by two client transactions denoted respectively T_1 and T_2.

1. T_1: $d_{(r)}$ is modified on S_1 by the local COUCHDB server which assigns a new revision, r'; the current version becomes $d_{(r')}$
2. T_2: $d_{(r)}$ is modified on S_2 by the local COUCHDB server which assigns a new revision, r''; the current version becomes $d_{(r'')}$
3. now, the replication mechanism must be triggered; S_1 sends to S_2 a transaction request specifying that d evolves from revision r to revision r'; S_2 detects that its current revision is *not* r but r'' and concludes that there is an update conflict.

A conflict is also detected when S_2 attempts a replication of its own transaction to S_1. Basically, the protocol describes the modification of a document d by specifying the initial and final revisions, and each replica must check that it is able to execute the very same transaction, which is only possible if its own current revision is the initial one specified by the transaction. Else, the document has been modified meanwhile by another application and we are in presence of two conflicting versions.

What happens then? COUCHDB takes two actions. First, a current version is chosen with a deterministic algorithm that operates similarly for each replica. For instance, each local COUCHDB server chooses $d_{(r'')}$ as the current revision, both at S_1 and S_2. No communication

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

is required: the decision algorithm is guaranteed to make the same decision at each site. This can be seen as a serialization *a posteriori* of the concurrent transactions $T_1 \to T_2$, resulting in a revision sequence $d_{(r)} \to d_{(r')} \to d_{(r'')}$. Second, the conflict is recorded in both $d_{(r')}$ and $d_{(r'')}$ with a `_conflict` attribute added to the document.

COUCHDB does not attempt any automatic reconciliation, since the appropriate strategy is clearly application dependent. A specific module should be in charge on searching for conflicting documents versions (i.e, those featuring a `_conflicts` attribute) in order to apply an *ad hoc* reconciliation mechanism.

Conflict management is easy to investigate. Run the following simple scenario: in your replicated, *backup*, database, edit and modify with Futon one of the movies (say, *The Social Network*). Then change (in a different way) the same movie in the original database *movies*. The continuous replication from *movies* to *backup* will generate a conflict in the latter. Conflicts can be reported with the following view:

```
function(doc) {
  if(doc._conflicts) {
    emit(doc._conflicts, null);
  }
}
```

It returns an array of the conflicting versions. A reconciliation-aware application should implement a module that monitors conflicts and determines the correct current version content, based on the specific application needs.

Shared-nothing architecture

The third replication option is an implementation of the shared-nothing architecture presented in Chapter 15, based on consistent hashing and data partition (often called "sharding"). A set of COUCHDB servers is (logically) assigned to a position on a ring, and documents are stored on the server that follows their hash value on the ring. The topology of the ring is replicated on each server, so that Client requests can be forwarded in one message to the relevant server. We do not further elaborate the design which closely follows that presented in Chapter 15, and tends to become a standard in the world of distributed storage system (see CASSANDRA, VOLDEMORT, MONGODB, and other "NoSQL" emerging platforms).

20.2 Putting COUCHDB into Practice!

We now propose several exercises and projects to further discover the features of COUCHDB that relate to the book scope, namely data representation, semi-structured data querying, and distribution features. Recall that you can create an account on our COUCHDB server and one or several database to play with the system.

20.2.1 Exercises

The following exercises apply to the *movies* database. You should first load the JSON documents available on our site. Then create and query MAPREDUCE views to obtain the required results. Views can be created with Futon, and searched with the following HTTP request:

```
/database/_design/application/_view/viewname?key=value
```

Many of these queries are similar to those suggested in the Chapter devoted to EXIST.

1. Give all `titles`.
2. Titles of the movies published after 2000.
3. Summary of "Spider-Man".
4. Who is the director of *Heat*?
5. Title of the movies featuring Kirsten Dunst.
6. What was the role of Clint Eastwood in *Unforgiven*?
7. Get the movies whose cast consists of exactly three actors?
8. Create a flat list of all the title-role pairs. (Hint: recall that you can emit several pairs in a MAP function.)
9. Get a movie given its title. (Hint: create a view where movies are indexed by their title, then query the view.)
10. Get the movies featuring an actor's name.
11. Get the title of movies published a given year or in a year range.
12. Show the movies where the director is also an actor.
13. Show the directors, along with the list of their films.
14. Show the actors, along with the list of directors of the film they played in.

Note: some of the above queries are *joins*. Expressing joins in MAPREDUCE is not the most natural operation but it can be achieved with a few tricks. Hint: recall that the result of the MAP phase is sorted on the key because it is stored in a Btree (and because this can be convenient if a subsequent REDUCE operation must be carried out). The order thereby defined on the MAP results helps to obtain the join result.

20.2.2 Project: build a distributed bibliographic database with COUCHDB

The proposed project consists in building a (simple) distributed bibliographic database based on a master-master architecture. Here are the full specifications:

1. there should be several (at least two!, up to the number of participants) COUCHDB instances (or "master"), storing a fully replicated collection of bibliographic entries; each update on one master should be replicated to all the other masters;
2. there should be a view that produces the Bibtex entry;
3. PDF files can be associated with entries (COUCHDB uses "attachments" to associate file in any format with a JSON document: see the documentation for details);

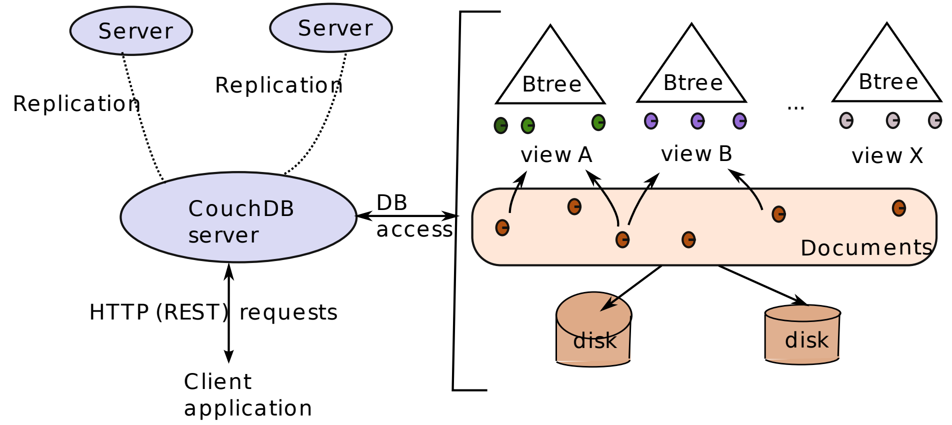
@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

4. several views should be created to allow the presentation (and search) of the bibliographic collection with respect to the following criteria: title, author, journal or publisher, year.
5. (advanced) COUCHDB manages a log of changes that records all the modifications affecting a database; use this log to create (with a view) a notification mechanism showing all the recently created entries of interest to a user (for instance: all the entries referring to a publication in JACM).

We provide a collection of JSON bibliographic entries extracted from the DBLP data sets as a starting point. The project could include the development of an interface to add / update / remove entries.

20.3 Further reading

The main source of information on COUCHDB is the Wiki available at `http://couchdb.apache.org`. The book [17], available on-line at `http://wiki.apache.org/couchdb/`, covers the main practical aspects of the system. The incremental maintenance of views built using a MAPREDUCE is inspired from the Sawzall language presented in [135].



@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

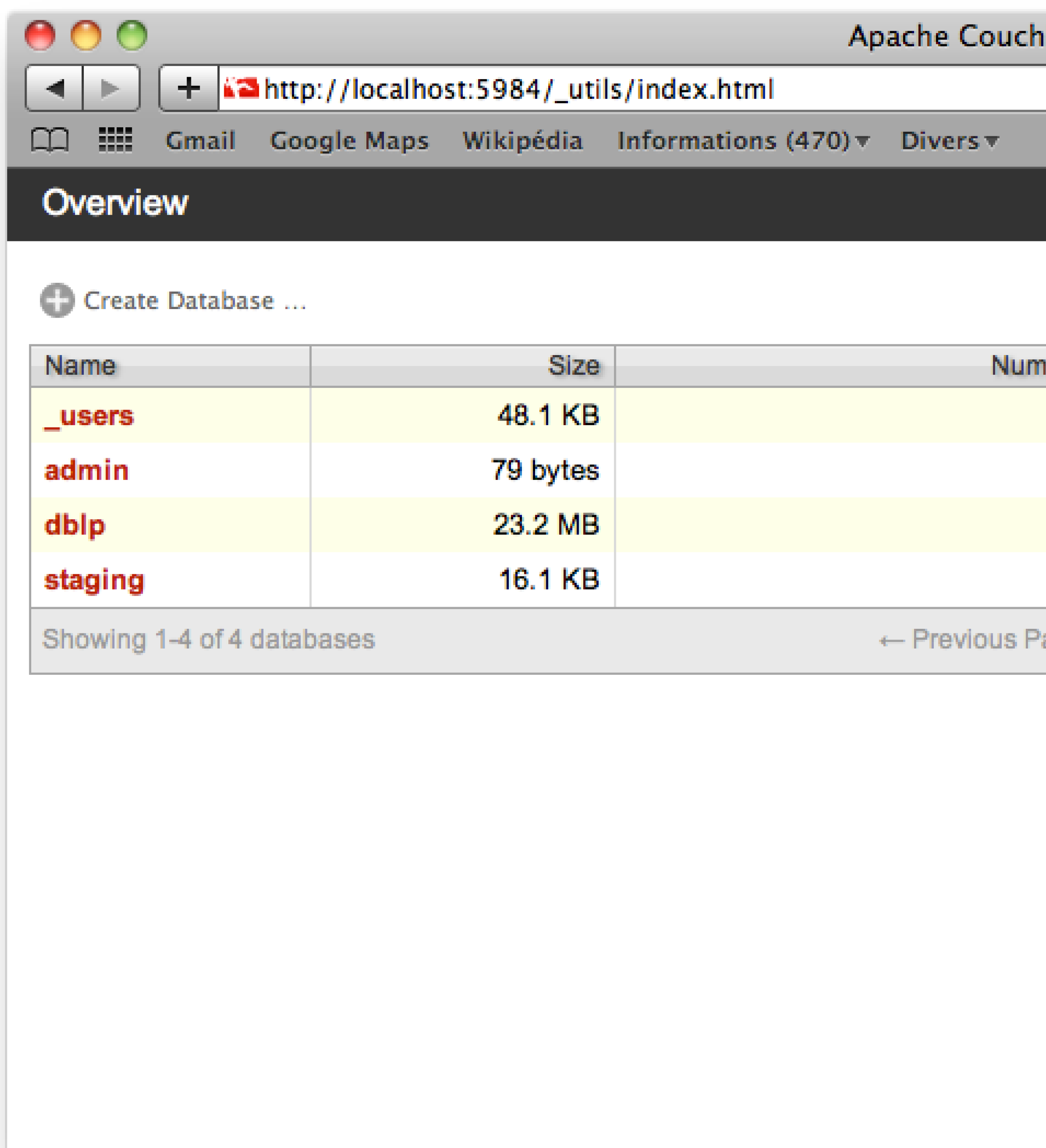


Figure 20.2: Futon, the admin interface of CouchDB

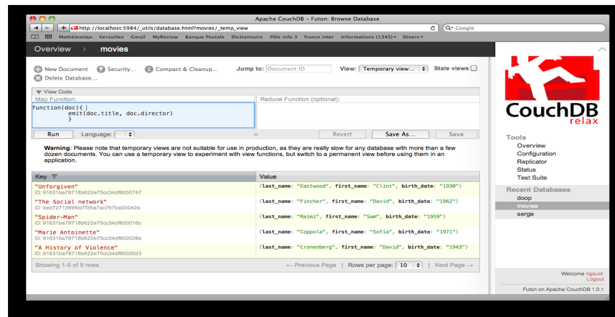
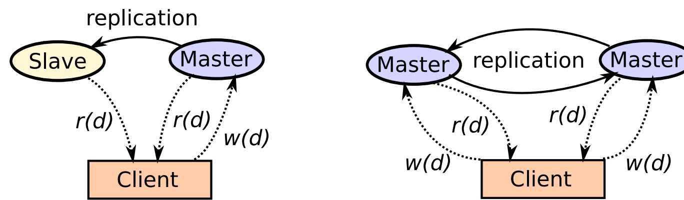
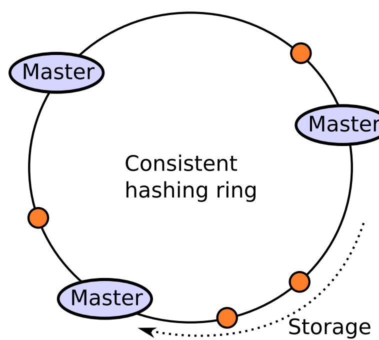


Figure 20.3: The view creation form in Futon



a - Master-slave arch.

b - Master-master arch.



c - Shared-nothing (Consistent hashing)

Figure 20.4: Distribution strategies with COUCHDB

@ Serge Abiteboul, Ioana Manolescu, Philippe Rigaux, Marie-Christine Rousset, Pierre Senellart, 2011; to be published by Cambridge University Press 2011.

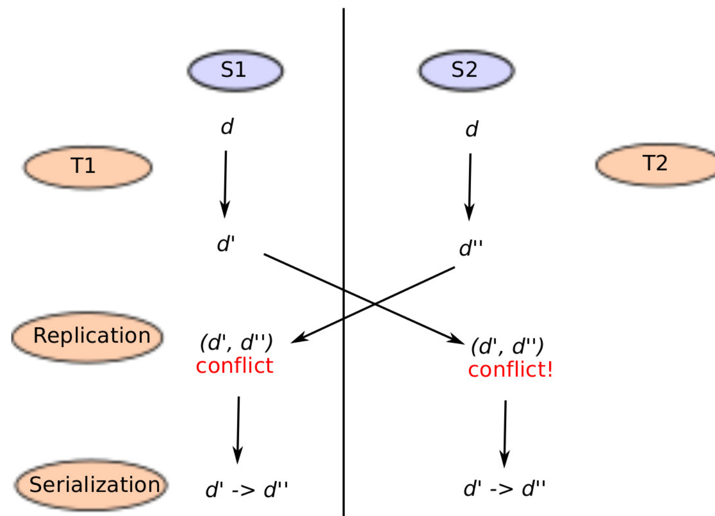


Figure 20.5: Multi-version concurrency in COUCHDB