Quick Start Guide to CouchDB

By Bhaskar S
12/10/2010

Of late, there has been a lot of buzz around the term **NoSQL**. The term **NoSQL** refers to the new breed of Database Management Systems that are Document-centric rather that the traditional Table-centric Relational Database Management Systems. The reasons behind the **NoSQL** movement is perhaps due to the flexibility (rigid schema based structure) and scalability (Internet-scale data) challenges with the traditional Relational Database Management Systems, which is especially true with the spurt of new generation of social networking web applications that are document-centric and Internet-scale.

Enter **CouchDB** !!! **CouchDB** is one of the new breed of Document-oriented Database Management Systems. With Document-oriented databases, there is no rigid schema or table structure to adhere to. With **CouchDB**, data is stored as a collection of JSON document(s). A JSON document is a collection of name-value pairs, where the value could be simple (integer, string, etc) or complex (arrays, structures, attachments, etc). With JSON document(s), both the data (key-value pairs) as well as the structure of the data can evolve over time making it well suited for a schema-less data representation.

CouchDB includes a built-in HTTP engine to allow access to the JSON document(s) via REST style web service. This is unlike the traditional Relational Database Management Systems which use the Structured Query Language (SQL) to access data from the table(s).

Replication is one of the most important and powerful features of **CouchDB**. In fact, the term "**Couch**" stands for "**C**luster **O**f **U**nreliable **C**ommodity **H**ardware". This means that any number of hosts with an independent copy of the same database can be run as a cluster for scalability and reliability reasons. **CouchDB**'s replication works on the principle of Eventual Consistency where updated documents are incrementally replicated over a period of time.

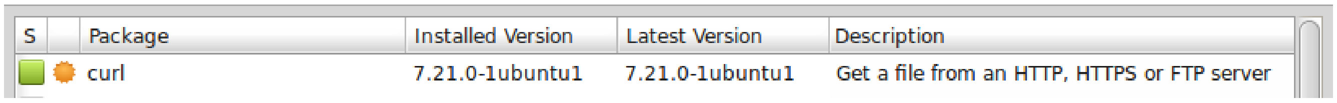
Now, lets get our hands dirty by exploring **CouchDB**.

Setup:

We will be exploring **CouchDB** in a Ubuntu based Linux desktop as it is much easier to install the necessary software.

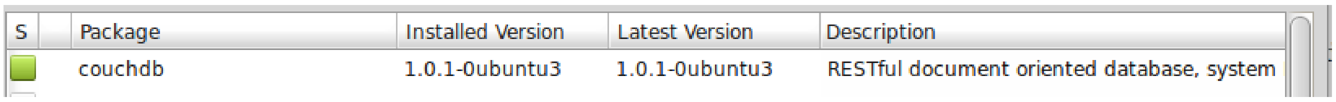
Go ahead and fire up the Synaptic Package Manager and install the packages: **curl** and **couchdb**.

Here is the screenshot for the installed package **curl**:



S		Package	Installed Version	Latest Version	Description
🟩	●	curl	7.21.0-1ubuntu1	7.21.0-1ubuntu1	Get a file from an HTTP, HTTPS or FTP server

Here is the screenshot for the installed package **couchdb**:



S		Package	Installed Version	Latest Version	Description
🟩		couchdb	1.0.1-0ubuntu3	1.0.1-0ubuntu3	RESTful document oriented database, system

Once the installation completes successfully, an instance of **CouchDB** is automatically started and running on the localhost at port 5984.

CouchDB Basics:

To check if **CouchDB** is running, type the following command using curl:

```
curl http://127.0.0.1:5984
```

The response from **CouchDB** should look something like the following:

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

Every response from **CouchDB** is in JSON format.

The newly installed **CouchDB** database system has no user defined database in it. To list all the databases in **CouchDB**, type the following command:

```
curl http://127.0.0.1:5984/_all_dbs
```

The response from **CouchDB** should look something like the following:

```
["_users"]
```

CouchDB stores all the user related authentication details in a special database called **_users**.

For our tests in this article, we will use the example of a DVD Lending Library where the database represents the DVD library and the DVDs will be represented as JSON documents in the library.

To create a DVD Library database called "**dvd-library**" in **CouchDB**, type the following command:

```
curl -X PUT http://127.0.0.1:5984/dvd-library
```

The response from **CouchDB** should look something like the following:

```
{"ok":true}
```

This response indicates that the database called "**dvd-library**" was created successfully.

Let us again list all the databases in **CouchDB**:

```
curl http://127.0.0.1:5984/_all_dbs
```

The response from **CouchDB** should look something like the following:

```
["_users","dvd-library"]
```

We now have an empty database called "**dvd-library**".

Remember that data in **CouchDB** is represented as self-contained JSON documents(s). Document(s) in a **CouchDB** database are stored in B-Tree storage. To store any data in B-Tree requires an unique key to identify what is being stored. Hence, to store a JSON document in CouchDB requires an unique ID.

To add a DVD represented as a JSON document with the unique ID “**wall-e**” to the database “**dvd-library**”, type the following command:

```
curl -X PUT http://127.0.0.1:5984/dvd-library/wall-e -d '{"Name":"Wall-E", "Format":"NTSC", "Studio":"Disney", "Year":2008, "Rating":"G"}'
```

The response from **CouchDB** should look something like the following:

```
{"ok":true,"id":"wall-e","rev":"1-ef5a5afcce16f1ecebee8a10a98e8050"}
```

This response indicates that the document with ID “**wall-e**” was successfully added to the database called “**dvd-library**”.

What is the field with the name “**rev**” in the response ? This field “**rev**” stands for Revision Number. In other words, it is the version number of the document. In the traditional Relational Database Management System, to update a row in a table means the database engine has to acquire a lock for that row to prevent others from updating the same row. Locking is expensive and limits the concurrency of the database. **CouchDB** takes a very different approach by avoiding locks. **CouchDB** uses Multi Version Concurrency Control (MVCC) to allow concurrent updates to a document in the database. This means that no in-place updates are done to a document in the database. Every update is copy-on-write operation (append only). When you want to update a document in the database, you get a copy of the latest version of the document from the database, make the necessary changes to the document and when you save the document to the database, it is appended with a newer revision number. In order to facilitate fast access to a document by revision number, CouchDB also indexes the document by revision number using a B-Tree (in addition to the index by unique ID).

The value of the attribute “**rev**” is specified as: **<N>-<md5>**, where **N** is the number of times the document was updated, followed by a dash '**-**', followed by the **MD5** hash of the document contents.

To fetch the DVD document for the unique ID “**wall-e**” from the database “**dvd-library**”, type the following command:

```
curl -X GET http://127.0.0.1:5984/dvd-library/wall-e
```

The response from **CouchDB** should look something like the following:

```
{"_id":"wall-e","_rev":"1-ef5a5afcce16f1ecebee8a10a98e8050","Name":"Wall-E", "Format":"NTSC", "Studio":"Disney","Year":2008,"Rating":"G"}
```

CouchDB uses two special attributes **_id** and **_rev** (same as “**rev**”) to represent the unique ID and the revision number of the document.

Now, let us go ahead and add few more DVD documents to the database “**dvd-library**” as follows:

```
curl -X PUT http://127.0.0.1:5984/dvd-library/up -d '{"Name":"UP", "Format":"NTSC", "Studio":"Disney", "Year":2009, "Rating":"PG"}'
```

```
curl -X PUT http://127.0.0.1:5984/dvd-library/despicable-me -d '{"Name":"Despicable Me", "Format":"NTSC", "Studio":"Universal", "Year":2010, "Rating":"PG"}'
```

```
curl -X PUT http://127.0.0.1:5984/dvd-library/toy-story-3 -d '{"Name":"Toy Story 3", "Format":"NTSC", "Studio":"Disney", "Year":2010, "Rating":"P"}'
```

```
curl -X PUT http://127.0.0.1:5984/dvd-library/shrek -d '{"Name":"Shrek", "Format":"NTSC", "Studio":"Dreamworks", "Year":2001, "Rating":"PG"}'
```

To query all the DVD documents from the database “**dvd-library**”, type the following command:

```
curl -X GET http://127.0.0.1:5984/dvd-library/_all_docs
```

The response from **CouchDB** should look something like the following:

```
{"total_rows":5,"offset":0,"rows":[
{"id":"despicable-me","key":"despicable-me","value":{"rev":"1-
0e34b2263cfb27e939f6fabd85d15826"}},
{"id":"shrek","key":"shrek","value":{"rev":"1-f3dd6b718c0b4ddeeb468d4d0fa3e56d"}},
{"id":"toy-story-3","key":"toy-story-3","value":{"rev":"1-dd8060ae88c7f1b85116b4d018cf40ba"}},
{"id":"up","key":"up","value":{"rev":"1-32c1377f760f8c273de2466cebe64bc6"}},
{"id":"wall-e","key":"wall-e","value":{"rev":"1-ef5a5afcce16f1ecebee8a10a98e8050"}}
]}
```

We have successfully retrieved all the 5 DVD documents from the database “**dvd-library**”.

Now, we want to lend the DVD with unique ID “**shrek**” to the customer “Mr. White”. When we lend a DVD, we update the DVD document to contain a new attribute called “**Borrower**” with the customer details such as the borrowers name and contact.

To update the DVD document for the unique ID “**shrek**” with the “**Borrower**” information in the database “**dvd-library**”, type the following command:

```
curl -X PUT http://127.0.0.1:5984/dvd-library/shrek -d '{"Name":"Shrek", "Format":"NTSC", "Studio":"Dreamworks", "Year":2001, "Rating":"PG", "Borrower":{"Name":"Mr. White", "Mobile":"123-456-7890"}}'
```

The response from **CouchDB** should look something like the following:

```
{"error":"conflict","reason":"Document update conflict."}
```

This response indicates that an error was encountered. What happened here ? Remember the earlier topic on the revision number “**rev**” ? In order to update the document, we need to specify the latest revision number of the document.

To get the latest revision number associated with the DVD document for the unique ID “**shrek**” from the database “**dvd-library**”, type the following command:

```
curl -X GET http://127.0.0.1:5984/dvd-library/shrek
```

The response from **CouchDB** should look something like the following:

```
{"_id":"shrek","_rev":"1-f3dd6b718c0b4ddeeb468d4d0fa3e56d", "Name":"Shrek", "Format":"NTSC",
"Studio":"Dreamworks","Year":2001,"Rating":"PG"}
```

From the response, we see the revision number to be "_rev":"1-f3dd6b718c0b4ddeeb468d4d0fa3e56d". Let us include this revision number to update the DVD document for the unique ID “**shrek**”, as follows:

```
curl -X PUT http://127.0.0.1:5984/dvd-library/shrek -d '{
"_rev":1-f3dd6b718c0b4ddeeb468d4d0fa3e56d", "Name": "Shrek", "Format": "NTSC", "Studio":
"Dreamworks", "Year":2001, "Rating":"PG", "Borrower":{"Name":"Mr. White", "Mobile":"123-456-
7890"}}'
```

The response from **CouchDB** should look something like the following:

```
{"ok":true,"id":"shrek","rev":"2-21c84195f868af9ff843be92f85173e6"}
```

The updated DVD document for “**shrek**” has been successfully saved as a new document with a new revision number in the database “**dvd-library**”.

To list all the revisions associated with the DVD document for the unique ID “**shrek**” from the database “**dvd-library**”, type the following command:

```
curl -X GET http://127.0.0.1:5984/dvd-library/shrek?revs=true
```

The response from **CouchDB** should look something like the following:

```
{"_id":"shrek","_rev":"2-21c84195f868af9ff843be92f85173e6", "Name":"Shrek", "Format":"NTSC",
"Studio":"Dreamworks","Year":2001,"Rating":"PG","Borrower":{"Name":"Mr. White", "Mobile":
"123-456-7890"},"_revisions":{"start":2,"ids":["21c84195f868af9ff843be92f85173e6",
"f3dd6b718c0b4ddeeb468d4d0fa3e56d"]}}
```

As we can see from the response, the various revisions of the document for the unique ID “**shrek**" is listed under the special attribute “**_revisions**”.

To fetch the first version of the document for the unique ID “**shrek**", type the following command:

```
curl -X GET http://127.0.0.1:5984/dvd-library/shrek?rev=1-f3dd6b718c0b4ddeeb468d4d0fa3e56d
```

The response from **CouchDB** should look something like the following:

```
{"_id":"shrek","_rev":"1-f3dd6b718c0b4ddeeb468d4d0fa3e56d", "Name":"Shrek", "Format":"NTSC",
"Studio":"Dreamworks","Year":2001,"Rating":"PG"}
```

Remember that the revision number is specified in the format: **<N>-<MD5>**.

CAUTION: Older versions of a document will be deleted when the CouchDB database is compacted. Also, if you enable replication to another node, only the latest revision of the document(s) is replicated.

So far in this article, we used the HTTP **PUT** method to add a new document to the **CouchDB** database "**dvd-library**". When we use the HTTP **PUT** method, we need to specify an unique ID to identify the document. However, it is possible to use the HTTP **POST** method to add a new document to the **CouchDB** database "**dvd-library**". When we use the HTTP **POST** method to add a new document to the database, **CouchDB** will automatically assign an unique ID for the document being added.

To add a dummy document to the database "**dvd-library**" using the HTTP **POST** method, type the following command:

```
curl -X POST http://127.0.0.1:5984/dvd-library -d '{"Name":"Dummy"}'
```

The response from **CouchDB** should look something like the following:

```
{"error":"bad_content_type","reason":"Content-Type must be application/json"}
```

This response indicates that an error was encountered. When we use the HTTP **POST** method, we need to explicitly specify the content type of the document as a JSON document.

To fix this, type the following command:

```
curl -X POST http://127.0.0.1:5984/dvd-library -H "Content-Type: application/json"
-d '{"Name":"Dummy"}'
```

The response from **CouchDB** should look something like the following:

```
{"ok":true,"id":"448b4ec73bd3af9934157bcf2b0009d5",
"rev":"1-0e4852d7b7849bb71fcc251e475611be"}
```

From the response, it is clear that the dummy document was successfully added to the database "**dvd-library**". One point to note here – in is case **CouchDB** automatically assigned an unique ID for the document "id":"448b4ec73bd3af9934157bcf2b0009d5". This is analogous to the auto-increment sequence column in the traditional Relational Database Systems except with a twist. Since **CouchDB** is built to run in a cluster of nodes that could potentially be distributed across the globe with the same database replica, the generated ID needs to be globally unique across the nodes. For this reason, **CouchDB** uses the 16-byte Universally Unique ID (**UUID**) to generate the unique ID, which is guaranteed to be unique across the nodes.

We could have used the UUID as the unique ID for our DVD documents instead of the DVD names. To get a UUID from CouchDB, type the following command:

```
curl -X GET http://127.0.0.1:5984/_uuids
```

The response from **CouchDB** should look something like the following:

```
{"uuids":["448b4ec73bd3af9934157bcf2b001801"]}
```

We could have then used the unique UUID 448b4ec73bd3af9934157bcf2b001801 with one of the DVD documents.

Had we used the UUID as the ID for the DVD documents, we would have had to execute the above command for each of the DVD documents. Rather than getting one UUID at a time, we could get a set of UUIDs by issuing the following command:

```
curl -X GET http://127.0.0.1:5984/_uuids?count=3
```

The response from **CouchDB** should look something like the following:

```
{"uuids":["448b4ec73bd3af9934157bcf2b001b3b","448b4ec73bd3af9934157bcf2b001dd3",
"448b4ec73bd3af9934157bcf2b001f76"]}
```

To add a DVD document using the first UUID 448b4ec73bd3af9934157bcf2b001b3b as the unique ID, type the following command:

```
curl -X PUT http://127.0.0.1:5984/dvd-library/448b4ec73bd3af9934157bcf2b001b3b
-d '{"Name":"Cars", "Format":"NTSC", "Studio":"Disney", "Year":2006, "Rating":"G"}'
```

The response from **CouchDB** should look something like the following:

```
{"ok":true,"id":"448b4ec73bd3af9934157bcf2b001b3b",
"rev":"1-5c14a13b73c3f7bb57c41d04f31667bb"}
```

We have successfully added a new DVD document for “**Cars**”.

We now want to delete the dummy document we added to our database “**dvd-library**” earlier. As we know, CouchDB had automatically assigned the unique ID of 448b4ec73bd3af9934157bcf2b0009d5 to the dummy document. Let us first try to fetch it by issuing the following command:

```
curl -X GET http://127.0.0.1:5984/dvd-library/448b4ec73bd3af9934157bcf2b0009d5
```

The response from **CouchDB** should look something like the following:

```
{"_id":"448b4ec73bd3af9934157bcf2b0009d5","_rev":"1-0e4852d7b7849bb71fcc251e475611be",
"Name":"Dummy"}
```

The dummy document exists in the database “**dvd-library**”.

To delete the dummy document from the database “**dvd-library**”, type the following command:

```
curl -X DELETE http://127.0.0.1:5984/dvd-library/448b4ec73bd3af9934157bcf2b0009d5?rev=1-
0e4852d7b7849bb71fcc251e475611be
```

The response from **CouchDB** should look something like the following:

```
{"ok":true,"id":"448b4ec73bd3af9934157bcf2b0009d5",
"rev":"2-957a05fdf2ff22ca2d4cc00bdf731cee"}
```

We used the HTTP **DELETE** method on the current revision of the dummy document to invoke the delete operation. Remember any update (change or delete) is a copy-on-write operation in **CouchDB**. As a result the **CouchDB** creates a newer revision of the dummy document to indicate deletion. To verify type the following command:

```
curl -X GET http://127.0.0.1:5984/dvd-library/448b4ec73bd3af9934157bcf2b0009d5?rev=2-957a05fdf2ff22ca2d4cc00bdf731cee
```

The response from **CouchDB** should look something like the following:

```
{"_id":"448b4ec73bd3af9934157bcf2b0009d5",
"_rev":"2-957a05fdf2ff22ca2d4cc00bdf731cee","_deleted":true}
```

CouchDB has added a special attribute called "**_deleted**" to the newer version of the document and set it to **true** to indicate the deletion of the dummy document. Let us verify that the dummy document is indeed deleted by issuing the following command:

```
curl -X GET http://127.0.0.1:5984/dvd-library/448b4ec73bd3af9934157bcf2b0009d5
```

The response from **CouchDB** should look something like the following:

```
{"error":"not_found","reason":"deleted"}
```

The response from **CouchDB** confirms that the dummy document is indeed deleted from the database "**dvd-library**".

Congratulations !!! We have completed the basic CRUD (Create, Read, Update, Delete) like operations with the **CouchDB** database.

Lets us continue with some other interesting capabilities of **CouchDB**.

CouchDB Views:

With the traditional Relational Database System, we use the Structured Query Language (**SQL**) engine to query and report on data stored in relational table(s). **CouchDB** database on the other hand does not support **SQL**. Also, remember that **CouchDB** database does not stored data in table(s) but as JSON documents. So, then how do we query and report on data that is stored as JSON documents ?

The answer is **CouchDB Views**. **CouchDB Views** are the mechanism by which to query and report on the documents stored in the **CouchDB** database. Think of **Views** as Stored Procedures from the traditional Relational database world.

CouchDB includes a built-in HTTP engine to allow access to the data stored as JSON documents in the **CouchDB** database via REST style services. Interestingly, **CouchDB** also includes a built-in JavaScript engine. JavaScript language makes it very easy to work with JSON document(s). As a result, **CouchDB Views** are implemented using JavaScript functions that are executed inside **CouchDB**.

There are two types of **Views** – **Permanent Views** and **Temporary Views**.

Permanent Views are stored in the **CouchDB** database in special documents. Just as user-defined data is stored in **CouchDB** database as JSON document(s) with unique ID, the JavaScript function(s) that define the Permanent Views are also stored in JSON format in special documents called **Design Documents** with the unique ID "**_design/<name>**". More on this in a little bit.

On the other hand, **Temporary Views** are not stored in the **CouchDB** database. To execute a **Temporary View**, we have to use the HTTP POST method to send the JavaScript function for the **Temporary View** to the specific URI "**/<database>/_temp_view**" for execution. **Temporary Views** are useful during application development.

Lets look at some examples so that we can get a better understanding of **CouchDB Views**.

We want to query and list all the DVD documents in the database "**dvd-library**". We will use the **Temporary View** to illustrate our example. To list all the DVD documents, type the following command:

```
curl -X POST http://127.0.0.1:5984/dvd-library/_temp_view -H 'Content-Type: application/json' -d
'{"map":"function(doc) { emit(doc.Year, doc.Name + \", \" + doc.Studio);}" }'
```

NOTE: The **View** function is specified in JSON format after the '-d' option

The response from **CouchDB** should look something like the following:

```
{"total_rows":6,"offset":0,"rows":[
{"id":"shrek","key":2001,"value":"Shrek, Dreamworks"},
{"id":"448b4ec73bd3af9934157bcf2b001b3b","key":2006,"value":"Cars, Disney"},
{"id":"wall-e","key":2008,"value":"Wall-E, Disney"},
{"id":"up","key":2009,"value":"UP, Disney"},
{"id":"despicable-me","key":2010,"value":"Despicable Me, Universal"},
{"id":"toy-story-3","key":2010,"value":"Toy Story 3, Disney"}
]}
```

Execution of the **View** against the database "**dvd-library**" returned 6 rows. Note that the response from **CouchDB** is also in JSON format. As indicated earlier, the **View** function is implemented as a JavaScript function. The **View** function takes the whole document as the argument. When we query the **View**, **CouchDB** executes the **View** function against all the documents in the database. The following is the definition of the **View** function in JavaScript:

```
function(doc) { emit(doc._id, doc.Name + \", \" + doc.Studio); }
```

The "**emit()**" function is used to return the results of the **View**. The "**emit()**" function takes two arguments – the first is the Key and the second is the Value. The rows returned by the View are sorted by the Key of the "**emit()**" function. In our example, we used the DVD Year as the Key to the "**emit()**" function. Think of this as an equivalent of the query "SELECT YEAR, NAME, STUDIO" in SQL from the traditional Relational Database world.

Now, what if we want to query only the DVD documents with the Year 2010 ? Think of this as an equivalent of the query “SELECT YEAR, NAME, STUDIO WHERE YEAR = 2010” in SQL from the traditional Relational Database world. To achieve this results, type the following command:

```
curl -X POST http://127.0.0.1:5984/dvd-library/_temp_view?key=2010 -H 'Content-Type: application/json' -d '{"map":"function(doc) { emit(doc.Year, doc.Name + \", \" + doc.Studio);}" }'
```

<u>NOTE</u>: Look at the use of the **View** parameters to constrain the results, which is specified in the URI as HTTP query parameter “**key=2010**”. The **View** function definition did not change

The response from **CouchDB** should look something like the following:

```
{"total_rows":6,"offset":4,"rows":[
{"id":"despicable-me","key":2010,"value":"Despicable Me, Universal"},
{"id":"toy-story-3","key":2010,"value":"Toy Story 3, Disney"}
]}
```

Execution of the **View** against the database “**dvd-library**” returned only 2 rows matching the criteria. Even though we are emitting all the documents from the database, the **View** parameters are used to constrain the output results. The constraint can only be applied on the Key values used in the “**emit()**” function, which is the DVD Year in our example.

More interesting, what if we want to query only the DVD documents between the Year 2008 and 2009 ? Think of this as an equivalent of the query “SELECT YEAR, NAME, STUDIO WHERE YEAR >= 2008 AND YEAR <= 2009” in SQL from the traditional Relational Database world. To achieve this results, type the following command:

```
curl -X POST http://127.0.0.1:5984/dvd-library/_temp_view?startkey=2008\&endkey=2009 -H 'Content-Type: application/json' -d '{"map":"function(doc) { emit(doc.Year, doc.Name + \", \" + doc.Studio);}" }'
```

<u>NOTE</u>: Look at the use of the **View** parameters to constrain the results, which is specified in the URI as HTTP query parameters “**startkey=2008\&endkey=2009**” (we had to escape '**&**' using backslash '\' as it means run in the background in Linux shell). Again, the **View** function definition did not change

The response from **CouchDB** should look something like the following:

```
{"total_rows":6,"offset":2,"rows":[
{"id":"wall-e","key":2008,"value":"Wall-E, Disney"},
{"id":"up","key":2009,"value":"UP, Disney"}
]}
```

Execution of the **View** against the database “**dvd-library**” returned only 2 rows matching the criteria.

Thus far we have been exploring the capabilities of **CouchDB Views** using **Temporary Views**. As we indicated earlier, the **View** functions are not stored in the **CouchDB** database and have to be provided as an option to the “**curl**” command. If we use the **Permanent Views**, the **View** functions will be stored in the **CouchDB** database permanently in a special design document associated with an unique ID. They can then be invoked anytime by querying the **View** in the special design document associated with the unique ID. More importantly, if we use **Permanent Views**, the **View** results will be stored in a B-Tree the first time the **View** is executed so that the future access will be much more efficient and faster. This is why we use **Temporary Views** only during application development.

To store **Views** in **Permanent Views** in **CouchDB** database, we will first defined the **View** function(s) in a "**.json**" file. The following are the contents of the file "**$HOME/Documents/dvd_views.json**" that defines the **View** function to query and list all the DVD documents by the attribute **Year** in the database "**dvd-library**":

```
{
    "language": "javascript",

    "views": {
        "all_dvds_by_year": {
            "map": "function(doc) {
                emit(doc.Year, doc.Name + ', ' + doc.Studio);
            }"
        }
    }
}
```

To store the **View** definition in the JSON file "**$HOME/Documents/dvd_views.json**" as a special **Design Document** in the **Permanent View** under the unique ID "**_design/dvd-views**", type the following command:

```
curl -X PUT http://127.0.0.1:5984/dvd-library/_design/dvd-views
-d @$HOME/Documents/dvd_views.json
```

The response from **CouchDB** should look something like the following:

```
{"ok":true,"id":"_design/dvd-views","rev":"1-436c17f7f4afcb9e80b64356d8829687"}
```

The response from **CouchDB** indicates we have successfully stored the **View** definition.

To verify that the **View** definition is indeed permanently stored in the database "**dvd-library**", type the following command:

```
curl -X GET http://127.0.0.1:5984/dvd-library/_design/dvd-views
```

The response from **CouchDB** should look something like the following:

```
{"_id":"_design/dvd-views","_rev":"1-436c17f7f4afcb9e80b64356d8829687",
"language":"javascript","views":{"all_dvds_by_year":{"map":"function(doc)
{           emit(doc.Year, doc.Name + ', ' + doc.Studio);          }"}}}
```

The response from **CouchDB** indicates we have permanently stored the **View** definition that lists all the DVD documents in the database "**dvd-library**".

Now, to query and list all the DVD documents in the database "**dvd-library**", we query the **View** "**all_dvds_by_year**" by executing the following command:

```
curl -X GET http://127.0.0.1:5984/dvd-library/_design/dvd-views/_view/all_dvds_by_year
```

The response from **CouchDB** should look something like the following:

```
{"total_rows":6,"offset":0,"rows":[
{"id":"shrek","key":2001,"value":"Shrek, Dreamworks"},
{"id":"448b4ec73bd3af9934157bcf2b001b3b","key":2006,"value":"Cars, Disney"},
{"id":"wall-e","key":2008,"value":"Wall-E, Disney"},
{"id":"up","key":2009,"value":"UP, Disney"},
{"id":"despicable-me","key":2010,"value":"Despicable Me, Universal"},
{"id":"toy-story-3","key":2010,"value":"Toy Story 3, Disney"}
]}
```

NOTE: From the HTTP URI, it should be clear that the **Design Documents** are under "**_design**" and the **Views** are under "**_view**".

Now to query only the DVD documents from the Year 2010, execute the following command:

```
curl -X GET http://127.0.0.1:5984/dvd-library/_design/dvd-views/_view/all_dvds_by_year?key=2010
```

NOTE: Look at the use of the **View** parameters to constrain the results, which is specified in the URI as HTTP query parameter "**key=2010**"

The response from **CouchDB** should look something like the following:

```
{"total_rows":6,"offset":4,"rows":[
{"id":"despicable-me","key":2010,"value":"Despicable Me, Universal"},
{"id":"toy-story-3","key":2010,"value":"Toy Story 3, Disney"}
]}
```

We can have more View functions in the Permanent View "_design/dvd-views". Let us enhance the contents of the file "**$HOME/Documents/dvd_views.json**" and add an additional **View** function to list all the DVD documents by the attribute **Studio** in the database "**dvd-library**":

```
{
    "_rev": "1-436c17f7f4afcb9e80b64356d8829687",

    "language": "javascript",

    "views": {
        "all_dvds_by_year": {
            "map": "function(doc) {
                emit(doc.Year, doc.Name + ', ' + doc.Studio);
            }"
        },
        "all_dvds_by_studio": {
            "map": "function(doc) {
                emit(doc.Studio, doc.Name + ', ' + doc.Year);
            }"
        }
    }
}
```

NOTE: We need to specify the revision number "**_rev**" to update an existing document in CouchDB database

To store the updated **View** definition in the JSON file "**$HOME/Documents/dvd_views.json**", type the following command:

```
curl -X PUT http://127.0.0.1:5984/dvd-library/_design/dvd-views
-d @$HOME/Documents/dvd_views.json
```

The response from **CouchDB** should look something like the following:

```
{"ok":true,"id":"_design/dvd-views","rev":"2-9a89b7d93718f043d27adda5a152627e"}
```

The response from **CouchDB** indicates we have successfully updated the **View** definition.

Now to query only the DVD documents from the Studio "Disney", execute the following command:

```
curl -X GET http://127.0.0.1:5984/dvd-library/_design/dvd-views/_view/all_dvds_by_studio?
key=\"Disney\"
```

NOTE: Look at the use of the **View** parameters to constrain the results, which is specified in the URI as HTTP query parameter **key=\"Disney\"** (we had to escape the double quotes since the key is of type String)

The response from **CouchDB** should look something like the following:

```
{"total_rows":6,"offset":0,"rows":[
{"id":"448b4ec73bd3af9934157bcf2b001b3b","key":"Disney","value":"Cars, 2006"},
{"id":"toy-story-3","key":"Disney","value":"Toy Story 3, 2010"},
{"id":"up","key":"Disney","value":"UP, 2009"},
{"id":"wall-e","key":"Disney","value":"Wall-E, 2008"}
]}
```

What we have seen thus far is **Views** returning a list of rows. What about aggregate results ? For example, we want to know how many DVD documents we have in our "**dvd-library**" database. Think of this as an equivalent of the query "SELECT COUNT(*)" in SQL from the traditional Relational Database world. In all our examples above, we used the special attribute "**map**" to define the **View** function to return a list of rows. This is called the **map** function of the **View**. For a **View** function to return an aggregate result, we will use the special attribute "**reduce**". This is called the **reduce** function of the **View**.

Once again, we will use the **Temporary View** to illustrate our aggregate example. To return a count of all the DVD documents, type the following command:

```
curl -X POST http://127.0.0.1:5984/dvd-library/_temp_view -H 'Content-Type: application/json' -d
'{"map":"function(doc) { emit(doc.Year, 1);}", "reduce":"function(keys, values) { return
sum(values); }" }'
```

NOTE: The **View** functions are specified in JSON format after the '-d' option

The response from **CouchDB** should look something like the following:

```
{"rows":[
{"key":null,"value":6}
]}
```

Execution of the **View** against the database "**dvd-library**" returned a single value of 6.

The following are the definitions for the **map** and **reduce View** functions in JavaScript:

```
{
    "map":"function(doc) { emit(doc.Year, 1);}",

    "reduce":"function(keys, values) { return sum(values); }"
}
```

The **map** function takes one argument – the document. On the other hand, the **reduce** function takes two arguments - the first is the List of Keys and the second is the List of Values. The List of Keys and Values are those emitted by the **map** function.

In the above example, the **map** function emits the Year attribute from the DVD document as the Key and the number 1 as the Value. The **reduce** function sums up the List of Values, which in this case is the sum of all the number 1 emitted by the **map** function for each of the Keys.

Next, we want to find how many DVD documents we have in "**dvd-library**" database grouped by Year. Think of this as an equivalent of the query "SELECT COUNT(*) … GROUP BY YEAR" in SQL from the traditional Relational Database world.

Once again, we will use the **Temporary View** to illustrate our "group by" aggregate example. To return a count of DVD documents by Year, type the following command:

```
curl -X POST http://127.0.0.1:5984/dvd-library/_temp_view?group=true -H 'Content-Type: application/json' -d '{"map":"function(doc) { emit(doc.Year, 1);}", "reduce":"function(keys, values) { return sum(values); }" }'
```

NOTE: Look at the use of **View** parameter "**group=true**" in the URI

The response from **CouchDB** should look something like the following:

```
{"rows":[
{"key":2001,"value":1},
{"key":2006,"value":1},
{"key":2008,"value":1},
{"key":2009,"value":1},
{"key":2010,"value":2}
]}
```

Execution of the **View** with the parameter "**group=true**" against the database "**dvd-library**" returned

DVD document counts grouped by Year.

To permanently store the "**reduce**" **View** function in the database "**dvd-library**", we will update the View functions in the Permanent View "_design/dvd-views" as follows:

```
{

    "_rev": "2-9a89b7d93718f043d27adda5a152627e",

    "language": "javascript",

    "views": {
        "all_dvds_by_year": {
            "map": "function(doc) {
                emit(doc.Year, doc.Name + ', ' + doc.Studio);
            }"
        },

        "all_dvds_by_studio": {
            "map": "function(doc) {
                emit(doc.Studio, doc.Name + ', ' + doc.Year);
            }"
        },

        "all_dvds_count_by_year": {
            "map": "function(doc) {
                emit(doc.Year, 1);
            }",

            "reduce": "function(keys, values) {
                return sum(values);
            }"
        },
    }
}
```

NOTE: We added a new **View** "**all_dvds_count_by_year**" for aggregation

To store the updated **View** definition in the JSON file "**$HOME/Documents/dvd_views.json**", type the following command:

```
curl -X PUT http://127.0.0.1:5984/dvd-library/_design/dvd-views
-d @$HOME/Documents/dvd_views.json
```

The response from **CouchDB** should look something like the following:

```
{"ok":true,"id":"_design/dvd-views","rev":"3-f98026b75157062cea383c6c49efebd5"}
```

The response from **CouchDB** indicates we have successfully updated the **View** definition.

Now to verify the aggregation View function "**all_dvds_count_by_year**", execute the following command:

```
curl -X GET http://127.0.0.1:5984/dvd-library/_design/dvd-views/_view/all_dvds_count_by_year
```

The response from **CouchDB** should look something like the following:

{"rows":[
{"key":null,"value":6}
]}

Execution of the **View** against the database "**dvd-library**" returned a single value of 6.

CouchDB Validation:

With the traditional Relational Database System, we use SQL Triggers to perform validation on data stored in relational table(s). This way only clean and consistent data gets stored in the relational table(s). The question then is how do we perform such validation on data that is stored as JSON documents in **CouchDB** ?

The answer is **CouchDB Validation** function.

Just like the **CouchDB Views**, the **CouchDB Validation** function is also implemented using JavaScript and is executed inside **CouchDB**. However, unlike **CouchDB Views**, the **CouchDB Validation** function must be permanently stored in the **CouchDB** database in a **Design Document** with the unique ID "**_design/<name>**". There can be only one **Validation** function per **Design Document** and is given a pre-defined name of "**validate_doc_update**". If you want multiple **Validation** functions, then you will have to define multiple **Design Documents**. When a document is either added or updated in the **CouchDB** database, it is validated against each **Validation** function in all the **Design Documents**. The order in which each of the **Validation** functions from the different **Design Documents** will be evaluated is arbitrary.

Lets look at an example so that we can get a better understanding of **CouchDB Validation** function**.**

From one of our earlier examples, we were able to store a dummy DVD document in our database "**dvd-library**" without any issues (refer to page 6). We want to store only valid DVD document(s) by ensuring that the DVD document contains all the attributes, namely, **Name**, **Format** (NTSC only), **Studio**, **Year** (must be greater than 1900 and less than 2010), and **Rating** (P, G, and PG only).

To store the Validation function "**validate_doc_update**" in the database "**dvd-library**", we will update the View functions in the Permanent View "_design/dvd-views" as follows:

```
{

    "_rev": "2-9a89b7d93718f043d27adda5a152627e",

    "language": "javascript",

    "views": {
        "all_dvds_by_year": {
            "map": "function(doc) {
                emit(doc.Year, doc.Name + ', ' + doc.Studio);
            }"
        },

        "all_dvds_by_studio": {
            "map": "function(doc) {
                emit(doc.Studio, doc.Name + ', ' + doc.Year);
            }"
        },

        "all_dvds_count_by_year": {
            "map": "function(doc) {
                emit(doc.Year, 1);
            }",

            "reduce": "function(keys, values) {
                return sum(values);
            }"
        },
    },

    "validate_doc_update": "function(newDoc, oldDoc, userContext) {
        if (!newDoc.Name) {
            throw({forbidden:'No Name attribute defined'});
        }
        if (!newDoc.Format) {
            throw({forbidden:'No Format attribute defined'});
        }
        if (newDoc.Format != 'NTSC') {
            throw({forbidden:'Format attribute must have a value of NTSC'});
        }
        if (!newDoc.Studio) {
            throw({forbidden:'No Studio attribute defined'});
        }
        if (!newDoc.Year) {
            throw({forbidden:'No Year attribute defined'});
        }
        if (newDoc.Year < 1900 || newDoc.Year > 2010) {
            throw({forbidden:'Year attribute must have a value > 1900 and < 2010'});
        }
        if (!newDoc.Rating) {
            throw({forbidden:'No Rating attribute defined'});
        }
        if (newDoc.Rating != 'P' || newDoc.Rating != 'G' || newDoc.Rating != 'PG') {
            throw({forbidden:'Rating attribute must have a value of P or G or PG'});
        }
    }"
}
```

NOTE: We added a new **Validation** function "**validate_doc_update**"

To store the updated **View** definition in the JSON file "**$HOME/Documents/dvd_views.json**", type the following command:

```
curl -X PUT http://127.0.0.1:5984/dvd-library/_design/dvd-views
-d @$HOME/Documents/dvd_views.json
```

The response from **CouchDB** should look something like the following:

```
{"ok":true,"id":"_design/dvd-views","rev":"4-b8b75f0177c81beeed033909a09a16f1"}
```

The response from **CouchDB** indicates we have successfully updated the **View** definition.

Now, let us try to store the dummy document again using the following command:

```
curl -X POST http://127.0.0.1:5984/dvd-library -H "Content-Type: application/json" -d
'{"Name":"Dummy"}'
```

The response from **CouchDB** should look something like the following:

```
{"error":"forbidden","reason":"No Format attribute defined"}
```

The dummy document was not added to the database “**dvd-library**” as it failed the **Validation** function.

Now, let us try another example with all valid attributes but with an invalid **Rating** using the following command:

```
curl -X POST http://127.0.0.1:5984/dvd-library -H "Content-Type: application/json" -d
'{"Name":"Tangled", "Format":"NTSC", "Studio":"Disney","Year":2010, "Rating":"X"}'
```

The response from **CouchDB** should look something like the following:

```
{{"error":"forbidden","reason":"Rating attribute must have a value of P or G or PG"}
```

The DVD document with invalid **Rating** attribute was not added to the database “**dvd-library**” as it failed the **Validation** function.

CouchDB Compaction:

From the earlier examples, it is clear that **CouchDB** does not update or delete documents for efficiency reasons (refer to page 3). Over a period of time this behavior will result in more space being used by the database. To reclaim space as well as purge old and deleted documents from the **CouchDB** database, one needs to perform database **Compaction** at regular intervals.

To get the statistics of the database “**dvd-library**”, execute the following command:

```
curl -X GET http://127.0.0.1:5984/dvd-library
```

The response from **CouchDB** should look something like the following:

```
{"db_name":"dvd-library","doc_count":7,"doc_del_count":1,"update_seq":13,"purge_seq":0,
"compact_running":false,"disk_size":53337,"instance_start_time":"1293155186989564",
"disk_format_version":5,"committed_update_seq":13}
```

The response from **CouchDB** indicates that the current database size is 53337 bytes.

To compact the **CouchDB** database “**dvd-library**”, execute the following command:

```
curl -X POST http://127.0.0.1:5984/dvd-library/_compact -H "Content-Type: application/json"
```

The response from **CouchDB** should look something like the following:

```
{"ok":true}
```

The response from **CouchDB** indicates that the compaction operation was successful.

Now, let us again get the statistics of the database “**dvd-library**” by executing the following command:

```
curl -X GET http://127.0.0.1:5984/dvd-library
```

The response from **CouchDB** should look something like the following:

```
{"db_name":"dvd-library","doc_count":7,"doc_del_count":1,"update_seq":13,"purge_seq":0,
"compact_running":false,"disk_size":8281,"instance_start_time":"1293155186989564",
"disk_format_version":5,"committed_update_seq":13}
```

The response from **CouchDB** indicates that the current database size after compaction is 8281 bytes.

Conclusion:

We have only scratched the surface of some of the features and capabilities of **CouchDB** database. The purpose of this guide is to get one quickly started with **CouchDB** database.

Hope this guide is useful. Enjoy !!!