

# CouchDB overview

Advanced Databases course, ULB

by *Irina Kameniar and Alexey Grigorev*

2013

# Table of Content

[Introduction](#)

[CouchDB](#)

[Document-Oriented Storages and Documents](#)

[CouchDB Locally](#)

[HTTP REST API Overview](#)

[Generating a Database](#)

[Futon](#)

[CouchDB Core](#)

[Design Documents](#)

[Validation](#)

[Types](#)

[Queries and MapReduce](#)

[Views](#)

[Map](#)

[Querying Views](#)

[Reduce](#)

[Replication](#)

[Scalability](#)

[Eventual Consistency](#)

[Incremental Replication](#)

[Replication API](#)

[Concurrency](#)

[B-Tree storage engine](#)

[Conflicts Management](#)

[Versioning](#)

[Conflicts](#)

[Conflict Resolution](#)

[Choosing Winning Revision](#)

[Revision ID](#)

[Revision Tree](#)

[Example Application](#)

[Comparisons](#)

[CouchDB and Relational Databases](#)

[CouchDB and MongoDB](#)

[Conclusions](#)

[Sources](#)

# Introduction

Apache CouchDB is a document-oriented NoSQL database that uses JSON to store data and MapReduce using Java Script for querying. For access it provides HTTP api which means it can be accessed from nearly all programming languages. CouchDB implements Multi-Version Concurrency Control to avoid locking and uses optimistic approach for conflict resolution, leaving the users decide what to do with conflicts.

In this work we want to make an overview of CouchDB. In particular, we want to focus on

- CouchDB main features
- Its API, including querying API and MapReduce
- How it deals with concurrency issues, replication and conflict resolution

For the practical part of this work we want to illustrate its main features with examples as well as to create a simple toy application to better understand this technology.

This work is organized as follows. First, we make an overview of what CouchDB is, its core and API, what it is a document-oriented storage and then we describe the MapReduce approach for querying data. The next part is devoted to replication, conflict management and concurrency issues. And finally, we will describe how to create a simple application in CouchDB.

## CouchDB

CouchDB's key features are:

- HTTP-based REST Api
- Distributed, scalable and fault-tolerant
- Document-oriented storage: the data is self-contained
  - i.e. it contains everything it needs - like real-world document
  - not a relational model (with rigid schemata and normal forms)
  - Traditional (Relational) DBs require you to model the data up-front
- Schema-free design allows to aggregate data after some fact has happened

## Document-Oriented Storages and Documents

A document is the central data structure in CouchDB, and it uses JSON to store documents. Each document has an id, which must be unique per database. Usually the best ids are UUIDs (Universal Unique ID - random string with extremely low collision probability<sup>1</sup>), but generally it can be anything

- A good example of a document is a file for a word processor or a user profile.

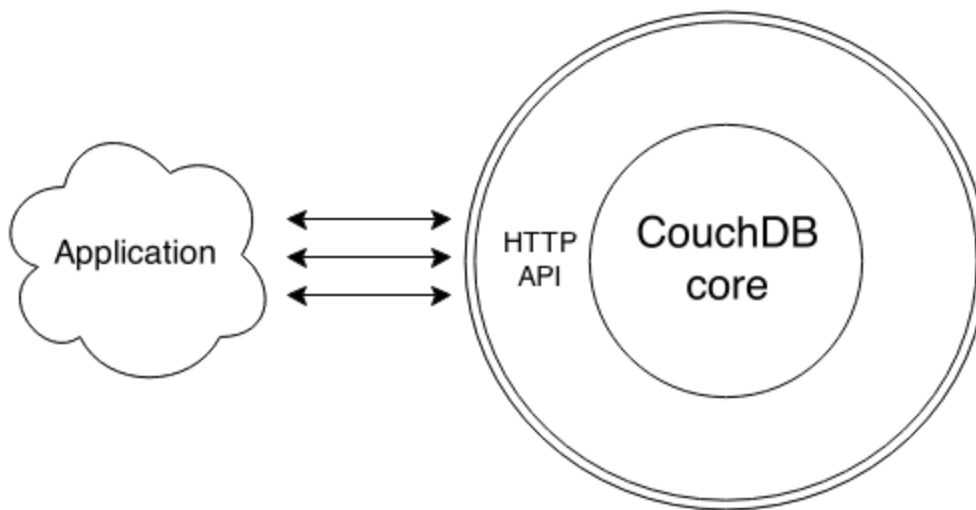
---

<sup>1</sup> [http://en.wikipedia.org/wiki/Universally\\_unique\\_identifier](http://en.wikipedia.org/wiki/Universally_unique_identifier)

- This sort of data you want to denormalize as much as possible
- Usually you want to fetch in one request as much data as it makes sense to display
- If we need to join some records, we want to precompute as much as possible and store related data together so it's retrieved at the same time. For that there's notion of *virtual documents* for that

## CouchDB Locally

How CouchDB works on a single machine



It consists of two components:

- HTTP REST API
- CouchDB core

## HTTP REST API Overview

- CouchDB provides RESTful HTTP Api to interact with it (on wikipedia there is a good overview of what is REST<sup>2</sup>)
- When it's installed with default settings, it can be accessed via <http://localhost:5984/>

To check if it's running, using **curl**<sup>3</sup> send a GET request to this address :

```
> curl -X GET http://localhost:5984/
```

The database replies with the following: (if you see that, everything works)

```
{
  "couchdb": "Welcome",
  "uuid": "2af023889ce22a70de68547c956e273a",
}
```

<sup>2</sup> [http://en.wikipedia.org/wiki/Representational\\_state\\_transfer](http://en.wikipedia.org/wiki/Representational_state_transfer)

<sup>3</sup> **curl** is an unix utility for sending HTTP requests, <http://curl.haxx.se/>

```

    "version": "1.4.0",
    "vendor": {
      "version": "1.4.0",
      "name": "The Apache Software Foundation"
    }
  }
}

```

(here and henceforth formatted for better readability)

To get the list of all available databases, use command "\_all\_dbs"

```
curl -X GET http://localhost:5984/_all_dbs
```

To create a new database, issue a **PUT** request to database you want to create

```
curl -X PUT http://localhost:5984/new_database
```

When an operation is successful, it replies with

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

## Adding

To add new document, we issue a PUT request to url/{database\_name}/{document\_id}

Since the schema is not rigid, we may put there everything we want, for example

```

> curl -X PUT http://localhost:5984/new_database/super_toaster -d
'{"title":"toaster","price":"10$"}'
{"ok":true,"id":"super_toaster","rev":"1-8f71d392bd5139ba142eb87ea52096d7"}

```

it returns id of the newly added plus revision id.

To retrieve this document use the same url

```

> curl -X GET http://localhost:5984/new_database/super_toaster
{
  "_id": "super_toaster",
  "_rev": "1-8f71d392bd5139ba142eb87ea52096d7",
  "title": "toaster",
  "price": "10$"
}

```

Note that we don't have to specify the id in the document, CouchDB takes care of adding it itself. Mechanisms behind versioning and revisions will be discussed below.

## Generating a Database

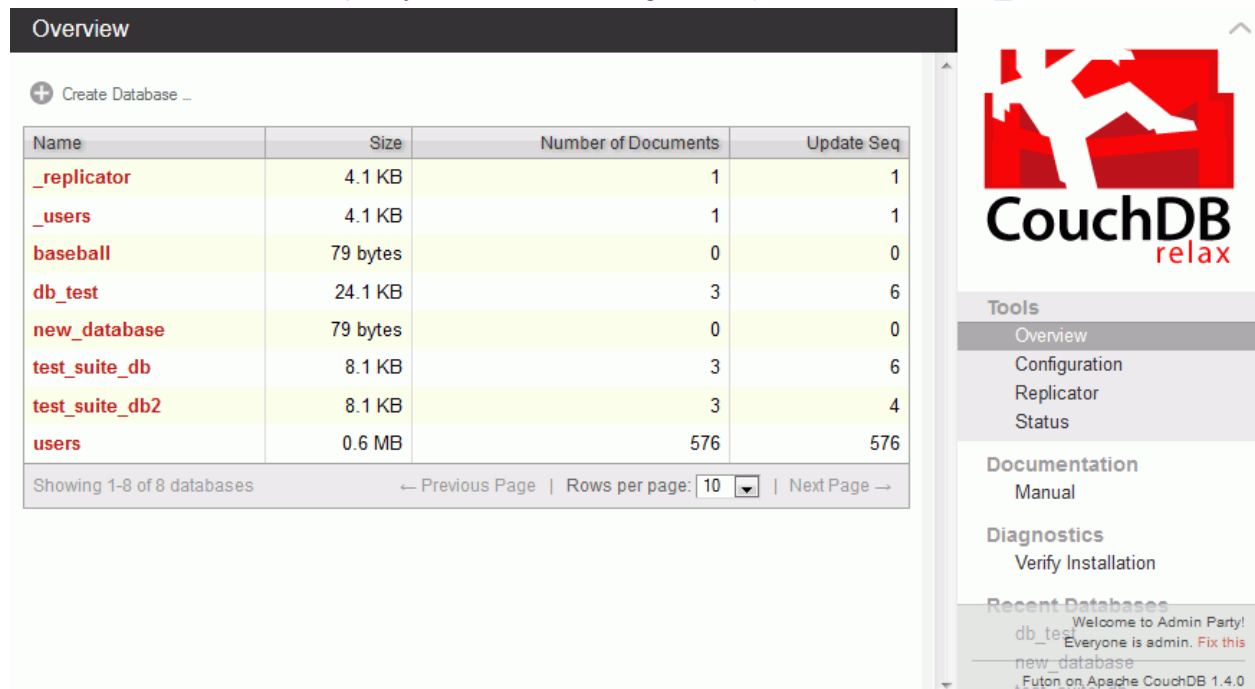
You can easily generate a lot of json data with <http://json-generator.appspot.com/> and It's easy to bulk post your data to CouchDB<sup>4</sup>. For doing that we have prepared 80k+ lines of JSON code (1500 documents) with user data to be inserted to the database (available at <http://goo.gl/jkcCim>)

To create this database execute the following:

```
# create a database "users"
curl -X PUT http://localhost:5984/users/
# download database data into "database.json"
wget http://goo.gl/jkcCim --no-check-certificate -O test-database.json
# use bulk post to add your data to CouchDB
curl -X POST http://localhost:5984/users/_bulk_docs -H
"Content-Type:application/json" -d @test-database.json
# at this point, CouchDB will answer with a list of newly added ids
```

## Futon

You don't have to interact with CouchDB only via HTTP requests: there is a web application for managing the database through a web browser, called Futon, which comes along with CouchDB. To access it, open your browser and go to [http://localhost:5984/\\_utils/](http://localhost:5984/_utils/)



Name	Size	Number of Documents	Update Seq
<a href="#">_replicator</a>	4.1 KB	1	1
<a href="#">_users</a>	4.1 KB	1	1
<a href="#">baseball</a>	79 bytes	0	0
<a href="#">db_test</a>	24.1 KB	3	6
<a href="#">new_database</a>	79 bytes	0	0
<a href="#">test_suite_db</a>	8.1 KB	3	6
<a href="#">test_suite_db2</a>	8.1 KB	3	4
<a href="#">users</a>	0.6 MB	576	576

Showing 1-8 of 8 databases    ← Previous Page | Rows per page: 10 | Next Page →

**CouchDB**  
relax

**Tools**

- Overview
- Configuration
- Replicator
- Status

**Documentation**

- Manual

**Diagnostics**

- Verify Installation

**Recent Databases**

- [db\\_test](#) Welcome to Admin Party! Everyone is admin. [Fix this](#)
- [new\\_database](#)
- [test\\_suite\\_db](#)

Futon on Apache CouchDB 1.4.0

With Futon you can create databases and explore existing ones

<sup>4</sup> <https://couchdb.readthedocs.org/en/latest/api/database.html#post-db-bulk-docs>

let's check what the users data we have put: click on "users"

Overview > users

+ New Document    Jump to:     View:     Stale views ☐

🔒 Security...    🗑️ Compact & Cleanup...    ✖ Delete Database...

Key ▲	Value
"5b8bed0d154256326d8e00bcc300180e" ID: 5b8bed0d154256326d8e00bcc300180e	{rev: "1-5e93a079072f9e9f53d17dd316ee722c"}
"5b8bed0d154256326d8e00bcc3001845" ID: 5b8bed0d154256326d8e00bcc3001845	{rev: "1-29775ff85f5b79ef4862b1affa12ab52"}
"5b8bed0d154256326d8e00bcc3001c22" ID: 5b8bed0d154256326d8e00bcc3001c22	{rev: "1-6c355345006e356032a4b0cbb30d425e"}
"5b8bed0d154256326d8e00bcc3002ac7" ID: 5b8bed0d154256326d8e00bcc3002ac7	{rev: "1-47f0d8028efe5b1c36f7fee0d34fa4ae"}
"5b8bed0d154256326d8e00bcc30035b9" ID: 5b8bed0d154256326d8e00bcc30035b9	{rev: "1-2c576a09ca35ff3b261428894b25386bb"}
"5b8bed0d154256326d8e00bcc3003a33" ID: 5b8bed0d154256326d8e00bcc3003a33	{rev: "1-61774b3ecf0c1cf2d9f11fb8b24f95f6"}
"5b8bed0d154256326d8e00bcc3004891" ID: 5b8bed0d154256326d8e00bcc3004891	{rev: "1-2eb6ddd1284eac7aad8547dc63d27885"}
"5b8bed0d154256326d8e00bcc30056f9" ID: 5b8bed0d154256326d8e00bcc30056f9	{rev: "1-53d59581ce017f3e6c2d66725dbd7c00"}
"5b8bed0d154256326d8e00bcc3005de7" ID: 5b8bed0d154256326d8e00bcc3005de7	{rev: "1-8f7ee52dca5a19816de7c8f739499711"}
"5b8bed0d154256326d8e00bcc3006b27" ID: 5b8bed0d154256326d8e00bcc3006b27	{rev: "1-f52107f209380b3c171dc654af098919"}

Showing 1-10 of 576 rows    ← Previous Page    Rows per page:     Next Page →

**CouchDB**  
relax

**Tools**  
Overview  
Configuration  
Replicator  
Status

**Documentation**  
Manual

**Diagnostics**  
Verify Installation

**Recent Databases**  
db\_test  
new\_database  
test\_suite\_db

**users** Welcome to Admin Party!  
Everyone is admin. [Fix this](#)

Futon on Apache CouchDB 1.4.0

To see what's inside a document, just click on it

There are two options:

- 1) to see formatted version of JSON

Overview > users > 005db5da-1fe9-47df-b947-cc941adfcfbb

Save Document Add Field Upload Attachment... Delete Document...

Fields Source

Field	Value
<b>_id</b>	"005db5da-1fe9-47df-b947-cc941adfcfbb"
<b>_rev</b>	"1-dad88c6c6a0df7f0e09e1e2d0d145eeb"
<b>about</b>	"Enim ipsum quis nisi esse eiusmod. Est labore cupidatat fugiat proident. Adipisicing irure consectetur ea laborum cupidatat ullamco cillum a..."
<b>address</b>	"489 Lincoln Avenue, Carrsville, Alaska, 9716"
<b>age</b>	40
<b>balance</b>	"\$3,252.00"
<b>company</b>	"Kiosk"
<b>email</b>	"potterfarley@kiosk.com"
<b>friends</b>	<div>0</div> <div>1</div> <div>2</div> <div>3</div> <div>4</div> <div>5</div> <div>6</div> <div>7</div> <div>8</div> <div>9</div>



# CouchDB

relax

Tools
Overview
Configuration
Replicator
Status

Documentation
Manual

Diagnostics
Verify Installation

Recent Databases
db\_test
new\_database
test\_suite\_db

users Welcome to Admin Party!
Everyone is admin. [Fix this](#)

Futon on Apache CouchDB 1.4.0

2) or raw JSON data

Overview > users > 5b8bed0d154256326d8e00bcc300180e


Save Document Add Field Upload Attachment... Delete Document...

Fields Source

Source

```

{
  "_id": "5b8bed0d154256326d8e00bcc300180e",
  "_rev": "1-5e93a079072f53d17dd316ee722c",
  "id": 0,
  "guid": "f84e962e-11b1-4ecb-a45b-b7de2b896c87",
  "isActive": true,
  "balance": "$1,329.00",
  "picture": "http://placeholder.it/32x32",
  "age": 21,
  "name": "Ericka Witt",
  "gender": "female",
  "company": "Solaren",
  "email": "erickawitt@solaren.com",
  "phone": "+1 (889) 422-3015",
  "address": "339 Central Avenue, Tryon, Oregon, 9699",
  "about": "Laboris magna ad proident voluptate ipsum nulla velit. Ullamco labore sunt laboris officia. Anim ipsum ut ullamco cupidatat reprehenderit adipisicing enim culpa proident proident. Amet proident nulla amet enim.\r\n",
  "registered": "2006-08-20T23:40:31 -02:00",
  "latitude": 69.724401,
  "longitude": 111.340739,
  "tags": [
    "sunt",
    "nulla",
    "aute"
  ]
}
```



# CouchDB

relax

Tools
Overview
Configuration
Replicator
Status

Documentation
Manual

Diagnostics
Verify Installation

Recent Databases
db\_test
new\_database
test\_suite\_db

users Welcome to Admin Party!
Everyone is admin. [Fix this](#)

Futon on Apache CouchDB 1.4.0



# CouchDB Core

Main core components:

- B-Tree-based storage engine
- MapReduce for querying (MapReduce queries are called *views*)

## Design Documents

A *design document* is a special type of documents that contain application code. They also live inside the database, but they are highly structured. These documents are very similar to usual documents: they can be replicated, have id and revision id.

### Virtual Documents

We typically want to fetch all the data we want to display in one request, so it makes sense to store related records together, and if there is a need for joining, we want to pre-compute this. For that there's a technique called "virtual documents" which uses views to collate data together

A design document starts with a special prefix "\_design/".

A design document may contain:

- MapReduce queries: "views" field
- "show" and "list" functions to render responses in other formats rather than JSON: XML, HTML, whatever you want

## Validation

Validation is a powerful tool to ensure that only document you need/want end up in your database.

There is a function "validate\_doc\_update" in a design document. It is used to prevent invalid or unauthorized updates. For example: only authorized users can add blog posts

Validation functions as all other CouchDB functions must not have any side-effects, and they are run in isolation, it also can block invalid updates from other CouchDB instances during replication. This function is executed each time a document is added or updated. If it raises an exception, the update is canceled, otherwise - accepted.

Validation is optional, if there's no such function, every update will get accepted

A design document may contain only one validation function, but if you have several design documents, all the validation function will be executed on a write request. If at least one of them decides to reject, the update is rejected.

NB: order of execution is not defined, so you must not make any assumptions about it

```
function(newDoc, savedDoc, ctx) {
  // some logic
  if (/* validation */) {
    throw({unauthorized: 'some message'})
  }
}
```

## Types

Types are needed to ensure that documents have proper type - i.e. have all required fields. This is a common pattern: to assign document types to records. It's not the part of CouchDB and it's up to user to decide whether to include type fields or not. This is quite convenient to use in MapReduce queries

Consider the following validation query:

```
function(newDoc, oldDoc, ctx) {
  if (newDoc.type == "post") {
    // validate post
  }
  if (newDoc.type == "comment") {
    // validate comment
  }
}
```

## Queries and MapReduce

For Relational Databases you can issue any query, and as long as you data is structured correctly, you'll be able to get an answer.

However, documents aren't always as structured as relations in Relational Databases, and for that we need a different approach. For CouchDB this approach is MapReduce.

MapReduce<sup>5</sup> is a paradigm of parallel computation.

A user has to provide two functions that will operate on all data:

- Map - apply to each document and emit zero or more key/value pairs
- Reduce - apply reduce function to the result of Map function grouped by key
- A combination of Map and Reduce functions is called a *view*

These functions provide CouchDB with great flexibility: they can adapt to various document structures.

## Views

So a view is a combination of map and reduce functions

---

<sup>5</sup> <http://en.wikipedia.org/wiki/MapReduce>

Views allows for parallel and incremental computation of views (described below). Since MapReduce produces key-value pairs, the results are also stored in the B-Trees. View results are stored in a B-Tree (like documents), but in their own file.

Views can be used for:

- filtering documents - to find needed ones
- extracting data
- presenting data in specific order
- building indexes
- doing some calculations

View functions are stored inside "views" field of a design document. Once you create a view, you query it to get results.

## Map

Map is applied to each document and emits zero or more key/value pairs - *view rows*. A map function doesn't depend on any information outside of the document, which allows CouchDB views be generated incrementally and in parallel. Views are stored as rows that are sorted by key in a B-Tree, which makes range retrievals efficient. When writing a map function, your goal is to build an index that stores related data records under nearby keys.

### Writing a map function

Map functions must not have any side-effects. A map function can fail during the execution - CouchDB recovers from it easily. It doesn't mean your functions should fail systematically, so it is important to check if fields you want to use exist.

Keys in the result returned by a map function can be anything, including lists that compose several keys. Map function takes one parameter "doc", it refers to a document from the database. Emit function is to be used inside map, it takes two arguments: key and value, emit function can be called multiple times (including zero times). Emitted pairs are stored in the B-Tree

### Examples

For our generated database (see [Generating a database](#)) we want to retrieve all active users that are women with more than 3 friends

```
function(doc) {  
  if (doc.isActive && doc.gender == 'female' && doc.friends.length >= 3) {  
    emit(null, doc);  
  }  
}
```

This gives us unsorted output (it is sorted by document id, which gives us an impression that the result is not ordered)

Since the results are sorted by keys emitted by a map function, we to order the result on the last name of a user, we pass their name as the first argument of emit function

```
function(doc) {  
  if (doc.isActive && doc.gender == 'female' && doc.friends.length >= 3) {  
    var lastName = doc.name.split(" ")[1];  
    emit(lastName, doc);  
  }  
}
```

Note that using Java Script gives us a lot of flexibility and we can transform our output as we want.

### Incremental Computation of Map Results

A map function runs through all records when you first query the view. A call to emit creates an entry in the view results where everything is sorted by the key. Indexes for each document can be computed independently and in parallel. If a document is changed, the map function is run only once to recompute the key and values for this single documents. If a document is deleted, corresponding entries are marked *invalid* - and they don't show up in the results

### Querying Views

Consider the following view function:

```
function(doc) {  
  if (doc.isActive && doc.gender == 'female' && doc.friends.length >= 3) {  
    var lastName = doc.name.split(" ")[1];  
    emit(lastName, {"name": doc.name, "email": doc.email});  
  }  
}
```

It outputs names and emails of all active female users with at least 3 friends and sorts the result by their last names. We save this view in a design document "females" under name "byLastName".

To query a view use the following url

```
> curl -X GET HOST/db/_design/{design_document}/_views/{view_name}
```

this returns all rows from the view.

For our example it is

```
> curl -X GET http://localhost:5984/users/_design/females/_view/byLastName
```

```

{
  "total_rows": 353,
  "offset": 0,
  "rows": [
    {
      "id": "d2a8c788-665a-4475-a123-56cef7a03dbe",
      "key": "Adams",
      "value": {
        "name": "Elvira Adams",
        "email": "elviraadams@centuria.com"
      }
    },
    /* remaining 352 rows are truncated */
  ]
}

```

You also can pass a *view parameter*

```
> curl -X GET
'HOST/db/_design/{design_document}/_views/{view_name}?key="{key}"'
```

where "{key}" is the key we used in emit call.

Example

```
> curl -X GET
'http://localhost:5984/users/_design/females/_view/byLastName?key="Burns"'
```

```

{
  "total_rows": 353,
  "offset": 41,
  "rows": [
    {
      "id": "ec8b0f81-4507-4a7e-8aa8-e9dbf9e57089",
      "key": "Burns",
      "value": {
        "name": "Elvia Burns",
        "email": "elviaburns@opticon.com"
      }
    },
    {
      "id": "ee6e6491-7e3b-48a7-8fda-241e571f1b4c",
      "key": "Burns",
      "value": {
        "name": "Mamie Burns",
        "email": "mamieburns@indexia.com"
      }
    },
  ]
}

```

```

    /* 1 record is not shown */
  ]
}

```

If we want to specify several keys, we then use the `keys` parameter.

However the value passed to this parameter has to be a properly URL-encoded JSON-string, which is not very convenient and readable. But there an alternative syntax for querying: we need to issue a PUT request with all needed parameters in the request body

```
> curl -X POST HOST/db/_design/{design_document}/_view/{view_name} -d '{...}'
```

For example, we want to see only users with last name “Burns”

```
$ curl -X POST http://localhost:5984/users/_design/females/_view/byLastName -d
'{"keys": ["Burns"]}'
```

Also we can ask to output the result of a key range

```
> curl -X GET 'HOST/db/_design/{design_document}/_views/{view_name}?
startkey="abc"&endkey="zzz"'
```

With our view, suppose we want to list all users within range “Adkins” and “Aquirre”

```
$ curl -X GET
'http://localhost:5984/users/_design/females/_view/byLastName?startkey="Adkins
"&endkey="Aguirre"'
```

```

{
  "total_rows": 353,
  "offset": 1,
  "rows": [
    {
      "id": "128a8c1c-8422-4cb4-8fd0-6b82503eb96d",
      "key": "Adkins",
      "value": {
        "name": "Mallory Adkins",
        "email": "malloryadkins@qaboos.com"
      }
    },
    /* 1 record is not shown */
    {
      "id": "0c55ee3b-1f22-40e3-a9f1-57b0b2f8742d",
      "key": "Aguirre",
      "value": {
        "name": "Stacie Aguirre",
        "email": "stacieaguirre@velocity.com"
      }
    }
  ]
}

```

```
]
}
```

Note that the range is inclusive, that is, the records with the “endkey” are also included in the result.

If we want to start from the beginning, we just don’t specify it at all or put null. The same applies to endkey: if we want to show the result till the end, we just don’t put it to a query.

It is also possible to output the result in the descending order

```
> curl -X GET HOST/db/_design/{design_document}/_views/{view_name}?
startkey="zzz"&endkey="aaa"&descending=true
```

But in this case we must put the start key as end key and the end key as start key.

```
> curl -X GET 'http://localhost:5984/users/_design/females/_view/byLastName?
startkey="Alford"&endkey="Aguirre"&descending=true'
{
  "total_rows": 353,
  "offset": 348,
  "rows": [
    {
      "id": "0c55ee3b-1f22-40e3-a9f1-57b0b2f8742d",
      "key": "Aguirre",
      "value": {
        "name": "Stacie Aguirre",
        "email": "stacieaguirre@velocity.com"
      }
    },
    /* 1 record is not shown */
    {
      "id": "128a8c1c-8422-4cb4-8fd0-6b82503eb96d",
      "key": "Adkins",
      "value": {
        "name": "Mallory Adkins",
        "email": "malloryadkins@qaboos.com"
      }
    }
  ]
}
```

And lastly, there are two important parameters for pagination: `limit` - number of items shown in the result, and `skip` - how many records we skip before starting to output the result.

Suppose that we show 10 items per page, and this is our limit. The value for skip we can calculate using the following formula:  $\text{skip} = (\text{page} - 1) * \text{limit}$

For page #2 skip is 10, and thus we have the following request:

```
> curl -X GET 'http://localhost:5984/users/_design/females/_view/byLastName?limit=10&skip=10'
```

For more details see [http://wiki.apache.org/couchdb/HTTP\\_view\\_API#Querying\\_Options](http://wiki.apache.org/couchdb/HTTP_view_API#Querying_Options)

## Reduce

After executing the map function, we run a series of reduce queries, one for each group returned from map. These functions operate on sorted rows emitted by map functions.

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

It takes three parameters, which we will describe below, and one of them - rereduce - is optional. So we might write this query:

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

In CouchDB reduce functions take advantage of B-Tree properties. The view result - pre-order traversal<sup>6</sup> of the tree. For every leaf node there's a chain of internal nodes reaching back to the root so reduce runs on every leaf node and then it runs on every intermediate node going up to the root. So, the end result of Reduce is caches and can be updated incrementally on changes to the data (first map results are recalculated, then reduce). When reduce runs on leaves, rereduce parameter is false, but for inner nodes it's true, it means it gets intermediate results from downstream nodes

So Reduce is also computed incrementally: we cache reduce results in the intermediate nodes of B-Tree complexity:

- generating a view takes  $O(n)$  time
- querying takes  $O(\log n)$  for lookup or  $O(\log n + k)$  for range queries

## Example

Suppose we want to calculate what is the average balance for all active female users with at least 3 friends. Here is our view:

---

<sup>6</sup> [http://en.wikipedia.org/wiki/Tree\\_traversal](http://en.wikipedia.org/wiki/Tree_traversal)



```
function(doc) {
  if (doc.isActive && doc.gender == 'female' && doc.friends.length >= 3) {
    var sum = doc.balance.replace(',', '').slice(1);
    emit(null, parseInt(sum));
  }
}

function(keys, values) {
  return sum(values) / values.length;
}
```

The result is only one value. It is also possible to calculate the average value per group. Say, we want to see the average salary per first letter of user's last name

```
function(doc) {
  if (doc.isActive && doc.gender == 'female' && doc.friends.length >= 3) {
    var sum = doc.balance.replace(',', '').slice(1);
    var lastName = doc.name.split(" ")[1];
    var firstLetter = lastName[0];
    emit(firstLetter , parseInt(sum));
  }
}

function(keys, values) {
  return sum(values) / values.length;
}
```

Now to query this view we need to issue the following GET request

```
> curl -X GET
http://localhost:5984/users/_design/females/_view/avg_sal_firstletter?group=true
```

Note that we also pass parameter group set to true to enable grouping (i.e. the reduce phase)

```
{
  "rows": [
    {"key": "A", "value": 2865.6666666666665},
    {"key": "B", "value": 2502.2370370370372},
    {"key": "C", "value": 2695.6999999999998},
    /* 15 records are not shown */
    {"key": "T", "value": 2146},
    {"key": "U", "value": 2746},
    {"key": "V", "value": 2176.5999999999999},
    {"key": "W", "value": 2561.4210526315787},
    {"key": "Y", "value": 1471.6666666666667},
```

```
{ "key": "Z", "value": 3480 }
}
```

If we didn't pass group attribute, it would use the default value (which is false), and the result of the reduce phase will not be grouped

```
> curl -X GET
http://localhost:5984/users/_design/females/_view/avg_sal_firstletter
{
  "rows": [
    { "key": null, "value": 2590.2145937493101 }
  ]
}
```

In this case the result is average of all values in the database.

For more information about views and its API, you may refer to CouchDB's wiki page [http://wiki.apache.org/couchdb/HTTP\\_view\\_API](http://wiki.apache.org/couchdb/HTTP_view_API) which gives a more detailed description.

## Replication

A replication is a mechanism that allows to synchronize two or more database instances.

Reasons for doing replication:

- reliability
- scaling
- load balancing

## Scalability

Three properties that you can scale

### Read Requests

Read: process an HTTP request, open a socket, find where data is stored - this takes processing time, responses to these requests may be cached. If your users generate more requests that you can handle, you need to add an additional server, make sure all users read same data.

### Write Requests

Same steps as for reads, but responses cannot be cached. If you have several servers - write must eventually occur on all of them

### Data (when amount grows too high)

- chunk data into manageable pieces

- put each on a separate server

**Replication** is the basis for all the 3 types.

## Eventual Consistency

Distributed systems operate over some network, and networks are often segmented. Eventual consistency means that data on all the servers will be consistent eventually, but the database (as a single unit) is always available. For CouchDB “availability” means being always available for writes: even if the current version of a document is in conflict, it still can accept new writes to the data. And it is up to the application how to resolve the conflicts at the later time. This concept is tightly related to [Conflict Resolution](#) which is discussed in the next section.

The opposite of the Eventual Consistency model is Strong Consistency. In this model the synchronization and replication between all database servers happen with each transaction. That is, a transaction is executed on all the servers and considered successful only when every server has replied that it is successfully committed. This approach is widely used for Relational Databases and has proven to be not scalable, and therefore many NoSQL solutions sacrifice Consistency in favor to better scalability. For more details please refer to Amazon’s Dynamo paper [DeCandia et al, 2007].

## Incremental Replication

Data is kept locally, no need for constant network access for communicating with other CouchDB instances. Synchronization happens whenever possible (when a network connection appears, etc).

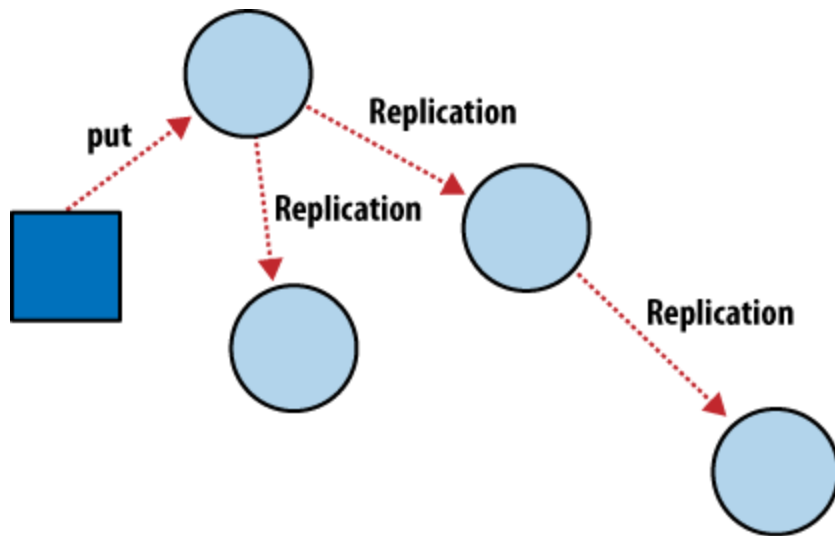
Replication in CouchDB works incrementally, only differences are replicated, not whole databases. If something during the replication goes wrong, when this is fixed, next time it starts from the same moment.

Note that replication is unidirectional (from source to target). If you want bidirectional replication, run it twice, swapping the source and the target for the second run.

## Incremental Replication

CouchDB achieves eventual consistency by Incremental Replication - this is the process when all document changes are copied periodically. This is called “Shared Nothing” cluster of databases with each node being independent and self-sufficient: there is no single point of connection in the system. Changes can be propagated in any way we like, and after replication each server can continue working independently.

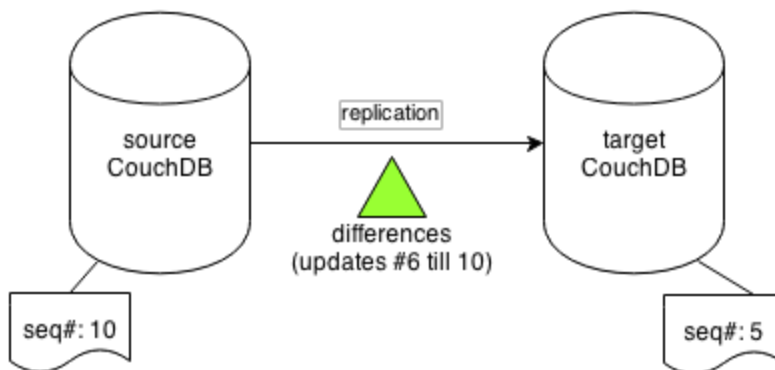
This is how it works



([figure source](#))

To scale the system we just add another server

Schematically we may show replication like that:



When the replication process is run first, it runs the comparison between the two servers, which returns a list of changed documents, this includes:

- new documents
- changed documents
- deleted documents

Documents that exist both on source and on target are not transferred (only differences will be moved).

Databases in CouchDB have a *sequence number*. It gets incremented when any change occurs and it remembers what change was associated with a particular sequence number. So calculating difference between source and target is efficient.

If replication process is interrupted, the target database may be left in an inconsistent state. But if you trigger the replication again, it will continue from the moment of interruption

## Replication API

To synchronize two databases we issue a simple PUT request where we specify

- the source of the updates
- the target

CouchDB will figure out what are the new documents and what are the new revisions that are not on the source but not yet on the target, and will transfer it to the target

```
> curl -X PUT http://localhost:5984/_replicate -d
'{"source":"users","target":"users_replica"}
```

The database replies with some statistics and tells if it was successful or not

**NB:** the request for replication will stay open till the replication process finishes, so it may take a while.

This is called *local replication* because both source and target are local. It is useful for backups, taking snapshots, etc. "Source" and "target" parameters are links, and they are relative to the server

There are *remote replications* as well:

- *push replication*: local source, remote target  
> curl -X PUT http://localhost:5984/\_replicate -d  
'{"source":"users","target":"http://localhost:5985/users"}'
- *pull replication*: remote source, local target  
> curl -X PUT http://localhost:5984/\_replicate -d  
'{"source":"http://localhost:5985/users","target":"users"}'
- and you can also run fully remote replication, with both source and target being remote  
> curl -X PUT http://localhost:5984/\_replicate -d  
'{"source":"http://localhost:5985/users","target":"http://localhost:5984/users"}'

A reply for a replication request may look like the following:

```
{
  "ok": true,
  "source_last_seq": 10,
  "session_id": "c7a2bbbf9e4af774de3049eb86eaa447",
  "history": [
    {
      "session_id": "c7a2bbbf9e4af774de3049eb86eaa447",
      "start_time": "Wed, 20 Nov 2013 19:30:46 GMT",
      "end_time": "Wed, 20 Nov 2013 19:30:47 GMT",
      "start_last_seq": 0,
      "end_last_seq": 1,
      "recorded_seq": 1,
      "missing_checked": 0,
      "missing_found": 1,
    }
  ]
}
```

```

    "docs_read": 1,
    "docs_written": 1,
    "doc_write_failures": 0,
  }
]
}

```

Some details:

- source\_last\_seq - source sequence value that was considered by this replication
- each replication id request is assigned a "session\_id"
- replication "history": some statistics (how many documents were read, written, how many failures, how long it took, etc)

### Continuous Replication

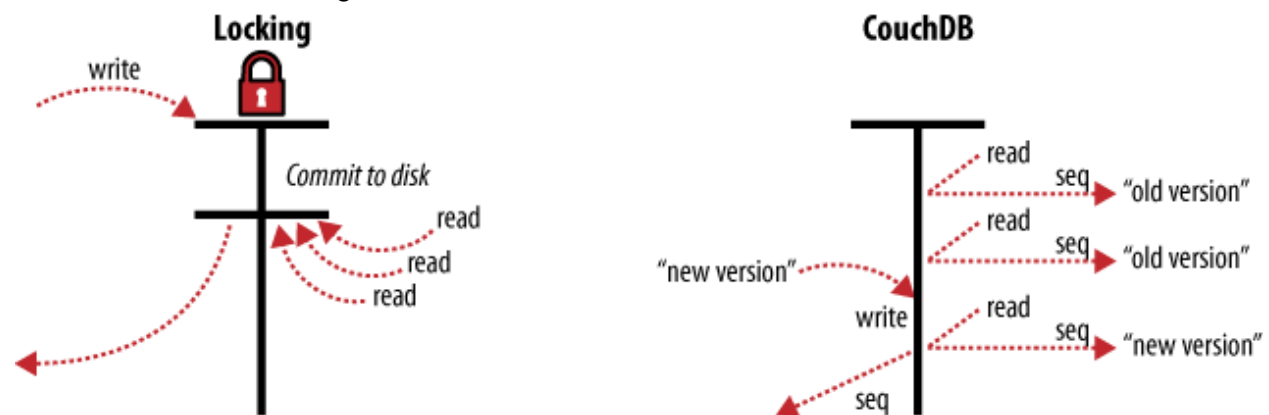
This was *manual replication* - replication that is triggered manually and happens just once. But it is possible to set up *continuous replication* - replication that will be triggered by some events (e.g. when the number of changes exceeds some threshold, etc) to do that add "continuous": true to the replication query. Note that you need to call this function every time your server restarts (it doesn't remember the replication settings from the previous runs).

In Futon replication is simple: just click on "Replication"

## Concurrency

In a typical relational database when we modify a table, we put a lock - and all other clients that want to access the table are queued.

This sequential execution of tasks wastes a lot of processor's power and time: under high load it may spend a lot of time trying to figure out whose turn is next. MVCC, Multi-Version Concurrency Control, is used to manage concurrent access to the data in CouchDB



([figure source](#))

This concurrency model allows CouchDB to run effectively even under high load, without worrying about queuing requests.

## B-Tree storage engine

B-Tree (CouchDB uses a variation of a B-Tree<sup>7</sup> called B+Tree<sup>8</sup>)

*B-Tree* is a sorted data structure that allows for searching, insertions and deletion in logarithmic time. Lookup is  $O(\log N)$  time, and range is  $O(\log N + K)$

This data structure is used everywhere, also for internal data: documents and views. Usage of this data structure imposes an important restriction: can access only by key.

Reason: to be make huge performance gains

In CouchDB the implementation is a little bit different from original B+Trees. It adds:

Support for MVCC

- reads and writes without locking the system
- writes do not block reads
- this is because of append-only design

Append-only design

- old versions are not deleted
- every time something is updated, a new node is created, and a new root as well but old reads still have references to the old root, so they are able to continue reading without being interrupted, i.e. have old consistent state
- data never lost and never corrupted

## Conflicts Management

### Versioning

All documents have versions (like in version control systems such as SVN)

If you want to change a document, you create a new one and save it over the old one. After doing that there will be two versions of the same document. Since a new version is just appended to the database, the read requests don't have to be suspended.

Once a new version is appended, all new requests are routed to this newer version

Updates in CouchDB

- load object
- change something
- save as a new revision

each revision is identified with a new "\_rev" value

if you want to update or delete a document, you must specify the revision you're updating. This is to ensure that you will not overwrite some other update

---

<sup>7</sup> <http://en.wikipedia.org/wiki/B-tree> and [http://www.scholarpedia.org/article/B-tree\\_and\\_UB-tree](http://www.scholarpedia.org/article/B-tree_and_UB-tree)

<sup>8</sup> [http://en.wikipedia.org/wiki/B%2B\\_tree](http://en.wikipedia.org/wiki/B%2B_tree)

suppose you want to update a document without providing the revision id:

```
> curl -X PUT http://localhost:5984/new_database/super_toaster -d
'{"title":"toaster","price":"15$"}'
```

CouchDB responses with an error:

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

So we add the revision id to the document we're updating:

```
> curl -X PUT http://localhost:5984/new_database/super_toaster -d
'{"title":"toaster","price":"15$","_rev":"1-8f71d392bd5139ba142eb87ea52096d7"}'
```

This time the database replies with "ok" and a new revision update:

```
{"ok":true,"id":"super_toaster","rev":"2-9c85d3c3324c3777a4665f00330b73b5"}
```

The same applies to deletes. Since a delete is also an update which just sets some special flag to "true", CouchDB also needs to know a revision that it deletes:

```
> curl -X DELETE http://localhost:5984/new_database/super_toaster
?rev=2-9c85d3c3324c3777a4665f00330b73b5
```

## Conflicts

A *conflicting* change is a change that occurs simultaneously in two or more replicas. This happens regularly in distributed databases.

So a *document conflict* means that now there are two latest revisions of the same document.

CouchDB can detect a conflicting change in a document and signals it with "\_conflict" flag set to true. When there are two revisions of the same file, CouchDB has to choose one *winning* revision - revision that will be stored and the latest revision. However the *losing* revisions aren't deleted - they are stored as well, but as previous revisions.

CouchDB doesn't attempt to reconcile the conflicting changes: it ensures that all conflicts are detected, but it's up to the application to deal with them. Essentially this is the same mechanism used by SVN<sup>9</sup> and other popular version control systems.

## Conflict Resolution

Replication from *A* to *B* (assuming triggered replication, not continuous) Direction  $A \rightarrow B$  (not  $B \rightarrow A$ )

---

<sup>9</sup> SVN is a version control system for managing source code, see [http://en.wikipedia.org/wiki/Apache\\_Subversion](http://en.wikipedia.org/wiki/Apache_Subversion) for more details



All other types of replication are reduced to these steps

1.  $A$ : create document  $d_1$
2. trigger replication  $A \rightarrow B$
3. now  $B$  also has  $d_1$
4. change  $d_1$  on  $B$  (CouchDB generates a new revision id)
5. change  $d_1$  on  $A$  (CouchDB also generates a new different revision id)
6. trigger replication  $A \rightarrow B$
7. CouchDB detects a conflict (two conflicting revisions of the same document)
8. Application resolves the conflict:
  - it tells CouchDB which revision to keep
  - another way: we merge two revisions, update the document and CouchDB will generate a new revision and mark the conflict resolved

To see if we have any conflicts we may use this simple view:

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

"\_conflict" is an array that contains all conflicting revisions

## Choosing Winning Revision

CouchDB uses a deterministic algorithm to ensure that each CouchDB instance will come up with the same winning and losing revision.

Note that your application should never depend on these details and should treat the results as an arbitrary choice rather than some deterministic algorithm.

### Algorithm:

- Each revision has a revision history: a list with all previous revision IDs.
- A version with longest revision history wins
- If the length is the same, "\_rev" values are compared in ASCII sort order

for example, "2-de..." wins over "2-7e..."

If we don't agree with CouchDB automatic choice, we delete one revision and keep another

```
> curl -X DELETE $HOST/database/document_id?rev=2-de...
```

This returns a new revision (remember that a delete is also an update)

Next, we put the values we want to keep back to the database, specifying the revision we like

```
> curl -X PUT $HOST/database/document_id -d '{..., "_rev": "2-7e..."}'
```

It also responds with a new revision ID.

This way we resolve the conflict

Now we need to replicate  $B \rightarrow A$ , so both instances are synchronized.

## Revision ID

Let's have a look at a typical revision ID:

3-dad88c6c6a0df7f0e09e1e2d0d145eeb

3 - an integer, the current version number, it gets incremented with each update

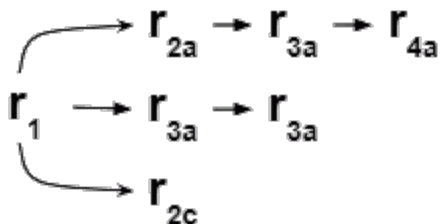
dad88c6c6a0df7f0e09e1e2d0d145eeb - md5 hash over a set of properties: JSON body, attachments, "\_delete" flag

It means that:

- updates on the same document on different instances create their own independent revision IDs.
- for two different documents with same data the right part of the revision ID will be the same:

```
> curl -X PUT $HOST/db/a -d '{"a":1}'  
{"ok":true,"id":"a","rev":"1-23202479633c2b380f79507a776743d5"}  
> curl -X PUT $HOST/db/b -d '{"a":1}'  
{"ok":true,"id":"b","rev":"1-23202479633c2b380f79507a776743d5"}
```

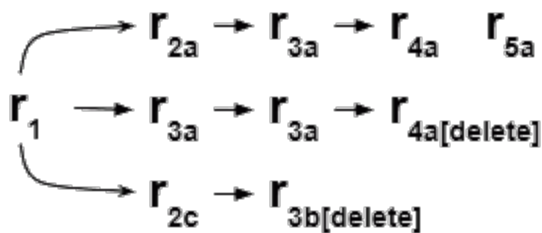
## Revision Tree



When there is a conflict, the history branches into a tree. Each branch can extend its own history independently. The last documents in the tree (i.e. leaves) are the set of conflicting revisions, in this case these are  $r_{4a}, r_{3b}, r_{2c}$

The way to resolve conflict:

- combine all the conflicting revisions ( $r_{4a}, r_{3b}, r_{2c}$ ) into a single document
- replace one of them, say  $r_{4a}$  with the new document. That will give us a new revision  $r_{5a}$
- delete all the remaining of leaves:  $r_{3b}, r_{2c}$



Note that when we delete a record, another revision is added to the revision tree, and the deleted record still exists, but as "deleted" node. It will be still possible to retrieve this record, but it will be marked with "\_deleted" flag set to true

Also afterwards during compaction data from non-leaf nodes will be removed

- leaving only a chunk of metadata called *tombstone* plus
- the list of historical "\_revs" is still retained in case in future you meet old database replicas

There also is a mechanism for "pruning" the revision tree to prevent it from growing too large

## Example Application

As a showcase we have chosen to implement a small blog on top of CouchDB.

Our schema is following:

- Article(author, title, content, created, tags);

Field	Value
<b>_id</b>	"000000da-b421-41c7-9b51-1740f4c6ef84"
<b>_rev</b>	"1-4b2c2093cf752a2cb5cd3a778a79dec5"
<b>author</b>	"Logan"
<b>content</b>	"Consequat exercitation in ex reprehenderit occaecat. Id id in fugiat mollit cillum cupidatat officia Lorem ad in eiusmod est veniam. Enim in..."
<b>created</b>	"2000-04-23T11:15:54 -02:00"
<b>tags</b>	0 "deserunt" 1 "aliquip"
<b>title</b>	"nisi magna non labore non qui"
<b>type</b>	"article"

- Comment(author, content, created, article\_id);

Field	Value
<b>_id</b>	"000bbef3-551d-4218-bc66-3bf8ecf5f482"
<b>_rev</b>	"1-7e2234287bc2dd4f8233ee388da40021"
<b>article_id</b>	"32fff6d6-1b5c-471b-a374-e836a26f0412"
<b>author</b>	"Browning"
<b>content</b>	"Irure exercitation dolore do reprehenderit aliqua ad nisi consectetur magna Lorem. Quis laboris anim nostrud commodo adipisicing ad. Magna s..."
<b>created</b>	"2003-08-05T12:54:13 -02:00"
<b>type</b>	"comment"

For generating a database we used the same approach as described in [Generating a database](#). That is:

- We created 2000 blog post articles each with 5-7 comments
- We created "blog" database
- Using bulk update uploaded all the blog posts to the database

However this time we created two types of documents: articles and comments for these articles. To distinguish between types we used the approach described in [Types](#): to article documents we assigned tag "article", and to comment documents - type "comment"

To retrieve all articles sorted by date of creation (descending)

```
function(doc) {
  if (doc.type == "article") {
    var dateTimeStr = doc.created.split(' ')[0]
    var date = Date.parse(dateTimeStr);
    var copy = eval(uneval(doc));
    copy.content = copy.content.slice(0, 100) + '...';
    emit(-date, copy);
  }
}
```

We save it as "blog/\_design/queries/\_view/articles" view

Thus the formula for the url for querying this view is

/blog/\_design/queries/\_view/articles?limit={limit}&skip={page \* limit}

## Blog

[New post](#)

[Read more](#)

ad aute est veniam adipisicing cillum

Date: 2013-10-17T06:26:57 -02:00

Author: Marissa

Aliquip nisi elit ea consectetur irure ipsum et. Commoda mollit ad adipisicing nostrud nulla do null...

[Read more](#)

qui aliqua ullamco ullamco veniam adipisicing

Date: 2013-10-16T08:37:00 -02:00

Author: Baldwin

Anim adipisicing nostrud aute occaecat officia culpa id tempor. Deserunt sit adipisicing laborum lab...

To retrieve some certain article we might just directly query it by the id. However we also want to show related comments, and therefore we need to join articles with comments

To do that we use technique called “View Collation”:

```
function(doc) {  
  if (doc.type == "article") {  
    emit([doc._id, 'a'], doc);  
  } else if (doc.type == "comment") {  
    emit([doc.article_id, 'b'], doc);  
  }  
}
```

That is, as for key we output the article key - the key we join on - and artificially created “tags” to signify different entities. Since in views all data is sorted by keys, we can just issue a range query (see [Map section](#) and [Querying Views](#)) to obtain the needed result.

In this case we save this view as “blog/\_design/queries/\_view/fullArticle” and the formula for calculating url for querying is as follows:

```
start = ["{id}", "a"]  
end = ["{id}", "b"]  
url = /blog/_design/queries/_view/fullArticle?startkey={start}&endkey={end}
```

That query will return us a sequence where an article is followed by its comments.

## Blog

[New post](#)

test post

Date: 2013-12-30T01:55:31.035Z

Author: test

test content

[Edit](#)

COMMENTS:

Alex

this is my comment

Bob

comment #2

[Back](#)

To edit posted article we delete current one, having ID and Revision ID, and replace it with edited article, ID remains the same:

EDIT YOUR ARTICLE:

Title:

Author:

Aliqua dolore laboris dolore deserunt quis magna fugiat cillum pariatur incididunt ea. Ipsum ipsum ea non tempor esse adipisicing Lorem aliquip. Aliquip anim esse culpa minim id esse laborum excepteur labore. Quis elit officia exercitation ut aute aute incididunt adipisicing duis. Veniam dolor ad ullamco labore excepteur do voluptate nulla. Nostrud elit proident velit eiusmod Lorem ea. Excepteur eu dolore proident consectetur velit consectetur exercitation. Do dolor sint ex exercitation pariatur dolor culpa occaecat. Cupidatat nulla ex ex proident proident ad tempor anim nisi sint consequat fugiat. Do pariatur consectetur proident nostrud esse. Sunt adipisicing irure qui aliquip. Commodo aliquip do culpa occaecat nisi incididunt cillum duis. Deserunt proident aliquip amet sunt incididunt. Aliqua nulla veniam in eiusmod deserunt. Et eu enim elit reprehenderit quis reprehenderit labore ipsum dolor. Adipisicing adipisicing incididunt minim eiusmod irure ipsum

Tags:

```
var url = '/blog/' + currentID + "?rev=" + currentRev;
$.ajax({
  url: url,
  type: 'DELETE',
  success: function (data) {
    var urlid = '/_uuids';
```

```

$.ajax({
    url: urlid,
    dataType: 'json',
    success: function (data) {
        var uuid = data.uuids[0];
        var url2 = '/blog/' + currentID;
        var article = $("#editform").serializeObject();
        article.type = 'article';
        article.article_id = currentID;
        article.created = new Date();
        article.tags = article.tags.split(',');
        $.ajax({
            url: url2,
            type: 'PUT',
            data: JSON.stringify(article),
            dataType: 'json',
            success: function (data) {
                alert("Article was updated");
                location.reload(); }
        });
    }
});
}
});
}
});

```

To add new comment we dynamically create form and insert it to database via ajax request, using current article ID:

LEAVE YOUR COMMENT:

Author:

comment #3

The page at 127.0.0.1:5984 says:

Comment was added

OK

```
var article = $("#commentform").serializeObject();
article.type = 'comment';
article.article_id = currentID;
article.created = new Date();
$.ajax({
  url: url,
  type: 'PUT',
  data: JSON.stringify(article),
  dataType: 'json',
  success: function (data) {
    alert("Comment was added");
  }
});
```

For adding new post we created new page with content form:



## New post

---

Title:

Author:

Enter text here...

Enter text here...

Send post

To insert new post to our database we used the following request:

```
var article = $("#usrform").serializeObject();
article.type = 'article';
article.created = new Date();
article.tags = article.tags.split(',');
$.ajax({
    url: url,
    type: 'PUT',
    data: JSON.stringify(article),
    dataType: 'json',
    success: function (data) {
        alert("Post was added");
    }
});
```

To get values from the forms elements we used function ***serializeObject*** :

```
$.fn.serializeObject = function() {
    var o = {};
    var a = this.serializeArray();
    $.each(a, function() {
        if (o[this.name] !== undefined) {
            if (!o[this.name].push) {
```

```

        o[this.name] = [o[this.name]];
    }
    o[this.name].push(this.value || '');
} else {
    o[this.name] = this.value || '';
}
});
return o;
};

```

The blog application itself is also a CouchDB document, which stores (as attachments) all html, css and javascript files that are used for displaying the results. This document id is “app” and it contains the main index file. Attachments can easily be accessed via an Internet browser, and in our application we use this: we put all this code into index.html file and upload it as an attachment.

To access the application use <http://localhost:5984/blog/app/index.html>.

## Comparisons

### CouchDB and Relational Databases

In comparison with traditional relational databases we may conclude that CouchDB has the following advantages and disadvantages.

#### Advantages

- The "schema" of data can be changed at any time.
- The document-oriented way of storing data is probably the closest to relational databases and the concept is easy to grasp.
- Another advantage of document-oriented storages is self-containedness, which, when used properly, may lead to joins-free design and easy object mapping
- Data will be indexed exactly for your queries, so you will get results in constant time.
- CouchDB's concurrency model allows users to get consistent answers immediately without being queued. Traditional RDBMS systems do not offer such options.
- Replication is very easy with CouchDB and it helps with detecting conflicts. On the other hand, it is very hard to achieve that ease in relational databases because of their Strong Concurrency model.

#### Disadvantages

- The schema-free design may easily become a disadvantage if the data is not kept tidy.
- We need to create views for each and every query, i.e. ad-hoc like queries (such as concatenating dynamic WHERE's and SORT's in SQL) queries are not easily available.
- The MapReduce paradigm for querying is not the simplest one to grasp quickly for people who are used to SQL.
- You will either have redundant data, or you will end up implementing join and sort logic yourself on "client-side" (e.g. sorting a many-to-many relationship on multiple fields). And implementing joins is not always straightforward.
- NoSQL in general and Document-Oriented databases in particular are not very mature technologies as Relational Databases. The latter have more than 30 years of history and development behind them.

## **CouchDB and MongoDB**

MongoDB and CouchDB are both document-oriented databases. Aside from both storing documents though, it turns out that they don't share much in common.

### **Protocol**

MongoDB uses a custom binary protocol. CouchDB uses HTTP REST. There's undeniably something nice about CouchDB's approach. Practically anything can be a CouchDB client, which is why not having a shell really isn't a big deal: your normal shell is Couch's shell.

On the flip side, whether you use CouchDB or MongoDB, most of what you do will use a library, which completely abstracts the underlying protocol. Yes, CouchDB's approach leads to nice and impressive documentation/examples, but it doesn't really change how you code. Ultimately, Mongo's approach is more flexible since you can build an HTTP REST interface on top of its binary protocol.

### **Organization**

In CouchDB, you have the concept of a database most people are familiar with, which contains documents. MongoDB has the same concept of a database, but rather than containing documents directly, a database contains one or more collection which contain your documents. In other words, MongoDB has one extra layer of containers.

Since we are dealing with schemaless documents, there's nothing that prevents from using a single collection in MongoDB and achieving the same thing.

Given that you can simulate either approach with either engine. Between the two, the CouchDB approach seems wrong. It isn't just that collections help organize things beyond documents, like indexes (or views in Couch), sharding and provide additional administrative flexibility (backup tools are collection-aware, for example). And it isn't that the collection approach is more efficient,

resulting in less wasted space and cpu cycles. It's that the collection-per-entity (or table-per-entity) just maps well with how most of applications are laid out.

## Conclusions

In this work we made an overview of Apache CouchDB: what it is, how to use it, how it addresses the issues of concurrency, replication and conflict management. Additionally, we created a simple application to show how to use CouchDB, and finally, compared it to traditional relational databases and to another document-oriented database MongoDB.

We conclude that it is best to use CouchDB in partitioned and high-load environment with its Eventual Consistency model and lock-free Multi-Version Concurrency Control approach. But in other environments it might be more beneficial to use MongoDB for the reasons discussed in the comparison section.

However we also see that the main problem of CouchDB, MongoDB and other NoSQL solutions is their lack of maturity. Traditional relational databases have decades of development behind them and we think that there should be a good reason not to use it. If an application does not need to operate over a highly-partitioned network, it is better to use a relational database with its guarantees for consistency, rely on a data schema and be able to use SQL for ad-hoc querying.

## Sources

- [CouchDB The Definitive Guide by Anderson, Lehnardt and Slater](#)
- [CouchDB documentation: Replication and Conflict Model](#)
- [Introduction to CouchDB views](#)
- [Example of a Simple CouchDB Application](#)
- [NoSQL: Is It Time to Ditch Your Relational Database?](#)
- DeCandia et al, 2007, Dynamo: Amazon's Highly Available Key-value Store [[pdf](#)]