CouchDB Best Practices

Collect best practices around the CouchDB universe.

Basics

Apache CouchDB™ is a database that uses JSON for documents, JavaScript for MapReduce indexes, and regular HTTP for its API.
Website: couchdb.apache.org
Docs: docs.couchdb.org
Blog: blog.couchdb.org
Old Wiki: wiki.apache.org
New Wiki: cwiki.apache.org
User List: user@couchdb.apache.org
Github: apache/couchdb
IRC: irc.freenode.net/couchdb

Reserved Properties, IDs and Revisions

Any top-level fields within a JSON document containing a name that starts with a `_` prefix are reserved for use by CouchDB itself. If you don't specify an `_id` CouchDB will create one for you. You

can configure in `uuids/algorithm` which algorithm will be used: `random`, `sequential` or `utc_random`.
Document revisions are used for optimistic concurrency control. If you try to update a document using an old revision the update will be in conflict. These conflicts should be resolved by your client, usually by requesting the newest version of the document, modifying and trying the update again.

Bootstrap

Bootstrap means in our case the preparation of the database in order to make it ready to run your project. In this process we configure CouchDB from a file which lives inside your project root, create the needed databases and deploy map functions and more, conveniently also from plain JavaScript files (or Erlang or Haskell or Ruby or whatever language you prefer for your views).
In order to make this process as seemless as possible we have created CouchDB Bootstrap. You can install it with npm:

```
$ npm install --global couchdb-bootstrap
```

CouchDB Bootstrap combines a set of small tools, which can be used independently. Each of those tools come has a similar API and is shipped with a CLI:

- couchdb-compile - Handle sources: fs mapping / JSON / CommonJS
- couchdb-configure - Configure CouchDB
- couchdb-ensure - Create database unless exists
- couchdb-secure - Secure databases: write security object
- couchdb-push - Push documents: users, replications, design docs and normal documents

Its best you install altogether right now:

```
$ npm install --global couchdb-bootstrap couchdb-
compile couchdb-configure couchdb-ensure couchdb-push
couchdb-secure
```

First we setup a couchdb folder inside your project:

```
couchdb
```

```
├── ...
└── ...
```

This folder is organized just like CouchDBs URLS: we have a `/_config.json` file (or folder, more about that below) and `_users`, `_replicator` and custom databases.

CouchDB Compile

When we talk about compilation we mean the transformation of a folder into a CouchDB JSON document. Map functions for example need to be stored as strings in design documents. Design documents are just like normal documents with an id that is prefixed with `_design/`:

```
{
  "_id": "_design/myapp",
  "views": {
    "by-date": {
      "map": "function(doc) { if ('date' in doc)
emit(doc.date, null) }"
    }
  }
}
```

You do not want to deal with the JSON encoding and want your view functions code checked into Git as *files*. Thats why the CouchDB Filesystem Mapping was invented. The idea is simple: every key in the JSON document corresponds to a directory if the value is an object, otherwise to a file, which contents will make the value:

```
myapp
├── _id             # contains `_design/myapp`
└── views
    └── by-date
        └── map.js # contains `function(doc) { if
('date' in doc) emit(doc.date, null) }`
```

We use Couchdb Compile and run `couchdb-compile` on this directory to get a JSON from those files:

```
[myapp]$ couchdb-compile
{
```

```
  "_id": "_design/myapp",
  "views": {
    "by-date": {
      "map": "function(doc) {\n  if ('date' in doc)
emit(doc.date, null)\n}"
    }
  }
}
```

JSON Files

Sometimes you don't want this excessive deep nested directory tree. You can abbreviate by using a JSON file. CouchDB Compile will use the filename as key (without the `.json` extension) and take the contents of that as value. Lets place an `package.json` file in the `myapp` directory with the following content:

```
{
  "name": "My Shiny Application",
  "description": "My Shiny Application helps everybody
to discover your brilliancy."
}
```

When we compile the `myapp` directory again the `package.json` file gets included:

```
[myapp]$ couchdb-compile
{
  "_id": "_design/myapp",
  "views": {
    "by-date": {
      "map": "function(doc) {\n  if ('date' in doc)
emit(doc.date, null)\n}"
    }
  },
  "package": {
    "name": "My Shiny Application",
    "description": "My Shiny Application helps
everybody to discover your brilliancy."
  }
}
```

ID Derivation from Filename

Thats great. You might argue about the duplicated information found both inside the id and in the filename. Thats why CouchDB Compile derives the `_id` property from the filename in case the id is not included in the final JSON. So lets change our directory to look like this:

```
_design
└── myapp
    ├── package.json
    └── views
        └── by-date
            └── map.js
```

The result is the same, but this time we have omitted the `_id` file. Note that CouchDB Compile not only considers the filename but also checks the parent directory. If it is `_design` or `_local` and if so, prepends it, too.

CommonJS Modules (AKA Node Modules)

I hear everybody scream:

"*How can I test my view? The view code even is no valid JavaScript at all!*"

CommonJS modules to the rescue! Besides filesystem mapping and JSON CouchDB Compile also supports CommonJS modules. Consider the following files:

```
_design/
└── myapp
    ├── index.js
    ├── package.json
    └── views
        └── by-date.js
```

_design/myapp/index.js

Just like a normal node module we use an index.js file as an entry point to our document (you can also use an myapp.js file instead):

```
exports.package = require('./package.json')
exports.views = {
  'by-date': require('./views/by-date')
}
```

_design/myapp/views/by-date.js
Now everything is common js:

```
exports.map = function(doc) {
  if ('date' in doc) emit(doc.date, null)
}
```

Again, we get the same result as before when we run `couchdb-compile --index`. Remember that we need to enable this feature with the `--index` flag.
This is the most flexible way to structure your files and also gives us testability, because the views are now plain CommonJS modules.

Configure CouchDB

CouchDB not only exposes its configuration over HTTP, configuration settings can also be altered via HTTP.
For example, you might want to enable CORS. For this you must change a bunch of settings. Write the following JSON to a `_config.json` file (or grab it from couchdb-cors-config):

```
{
  "httpd": {
    "enable_cors": true
  },
  "cors": {
    "origins": "*",
    "credentials": true,
    "methods": [
      "GET",
      "PUT",
      "POST",
      "HEAD",
      "DELETE"
    ],
    "headers": [
      "accept",
      "authorization",
      "content-type",
      "origin",
```

```
      "referer",
      "x-csrf-token"
    ]
  }
}
```

We can now write that configuration with CouchDB Configure. The configuration will be used immediately - no restart required:

```
$ couchdb-configure http://localhost:5984 _config.json
{
  "httpd/enable_cors": {
    "ok": true,
    "value": "true"
  },
  "cors/origins": {
    "ok": true,
    "value": "*"
  },
  "cors/credentials": {
    "ok": true,
    "value": "true"
  },
  "cors/methods": {
    "ok": true,
    "value": "GET,PUT,POST,HEAD,DELETE"
  },
  "cors/headers": {
    "ok": true,
    "value": "accept,authorization,content-
type,origin,referer,x-csrf-token"
  }
}
```

CouchDB Configure uses CouchDB Compile under the hood so you have all the options to define your configuration as files, JSON or CommonJS module.

Create Databases

You can create a database with CouchDB Ensure:

```
$ couchdb-ensure http://localhost:5984/mydb
{
```

```
  "ok": true
}
```
If it does already exist nothing happens.

Secure a Database

CouchDB is *absolutely open by default*. That means everybody can write documents (except design documents) and everybody can read every document.
This can be changed on a per database basis by altering the security object:
The security object consists of two compulsory elements, admins and members, which are used to specify the list of users and/or roles that have admin and members rights to the database respectively:
`members`: they can read all types of documents from the DB, and they can write (and edit) documents to the DB except for design documents.
`admins`: they have all the privileges of members plus the privileges: write (and edit) design documents, add/remove database admins and members, set the database revisions limit and execute temporary views against the database. They can not create a database nor delete a database.
From the CouchDB documentation.
A simple security object looks like this:

```
{
  "admins": {
    "names": [],
    "roles": []
  },
  "members": {
    "names": [],
    "roles": [
      "myapp-user"
    ]
  }
}
```

Here, access is only permitted to database admins and users

with the role `myapp-user`.
Use CouchDB Secure to alter the security object:

```
$ couchdb-secure _security.json
{
  "ok": true
}
```

CouchDB Secure uses CouchDB Compile under the hood so you have all the options to define your security object as files, JSON or CommonJS module.

Deploy Documents

CouchDB Push can be used to deploy documents, be it design documents, users, replications or ordinary documents to a CouchDB database. Under the hood CouchDB Compile is used, so the everything you have learned about compilation above is also valid here.

```
[myapp]$ couchdb-push http://localhost:5984/mydb
{
  "ok": true,
  "id": "_design/myapp",
  "rev": "1-e2dfeb85f19c981e311144a6105d7de8"
}
```

Complete Bootstrap

Now you have all in place to understand CouchDB Bootstrap. While CouchDB Push acts on document level, CouchDB Bootstrap acts on the server level. Here you can organize different databases, users, replications and configuration:

```
couchdb
├── _config.json
├── _replicator
│   ├── setup-alice.json
│   └── setup-bob.json
├── _users
│   ├── alice.json
│   └── bob.json
├── myapp-db
```

```
│   ├── _design
│   │   └── myapp.js
│   ├── _security.json
│   └── adoc.json
├── myapp-alice-db
│   └── _security.json
└── myapp-bob-db
    └── _security.json
```

The whole project can now be bootstrapped with

```
[couchdb]$ couchdb-bootstrap http://localhost:5984
```

Admin User

First thing to do is setup the user account

- Go to http://localhost:5984/_utils/ in your web browser
- Click on `Setup more admins` in the bottom right hand corner of the sidebar
- Enter a username + password (this will be the root admin of your CouchDB)
- Click on `Logout` which is also in bottom right had corner

Great, now you've configured your Admin User. However, you don't want to actually use this account to do things, so proceed!

Creating User

To create a non admin user, follow these steps:

- While still on `http://localhost:5984/_utils/` in your browser (and logged out)
- Click on `Signup` in the bottom right of the sidebar
- Enter username + password

Change Password

Since CouchDB 1.2 updating the user password has become much easier:

1. Request the user doc: `GET /_users/org.couchdb.user:a-`

`username`
2. Set a `password` property with the password
3. Save the doc

The User authentication plugin for PouchDB and CouchDB provides a `changePassword` function for your convenience.

Document Modeling

Embrace the Document ID

In CouchDB ids can be arbitrary strings. Make use of this!
The id cannot be changed afterwards, instead you can move a document via COPY and DELETE.
Before deciding on using a random value as doc `_id`, read the section When not to use map reduce
Use domain specific document ids where possible. With CouchDB it is best practice to use meaningful ids.
You can use the tiny to-id module to normalize names, titles and other properties for use as id.
When splitting documents into different subdocuments I often include the parent document id in the id. Use docuri to centralize document id knowledge.

A note about `/` in document ids

Slashes are totally fine in document ids and widely used.
However, a specific deployment setup (nginx proxy with subdirectory rewrite) might require you to not use slashes in document ids. Thats the reason we use colons here.
Note that you have to encode the document id, when used in an url, but you should do that anyway.

Use document ids to enforce uniqueness

The only way to enforce something is unique is to include it in the document id.

For example, in the *_users* database the username is part of the id:

```
{
  "_id": "org.couchdb.user:jo@die-tf.de",
  "_rev": "1-64db269b26ed399733beb259d1a31304",
  "name": "jo@die-tf.de",
  "type": "user"
}
```

Document Modeling to Avoid Conflicts

At the moment of writing, most of our data documents are modeled as 'one big document'. This is not according to CouchDB best practice. *Split data into many small documents* depending on how often and when data changes. That way, you can avoid conflicts and conflict resolution by just appending new documents to the database. Its often a good idea to have many small documents versus less big documents. As a guideline on how to model documents *think about when data changes*. Data that changes together, belongs in a document. Modelling documents this way a) avoids conflicts and b) keeps the number of revisions low, which improves replication performance and uses less storage.

Document Validations

To enforce a certain document schema use Validate document update functions.

On every document write (via PUT, POST, DELETE or `_bulk_docs`) the validation function, if present, gets called. Validation functions live in design documents in the special `validate_doc_update` property. You can have many validation functions, each in different design documents. If any of the validation function throws an error, the update gets rejected and the error message gets returned to the client. Note that validation is also triggered during replication causing documents to get rejected if they do not meet criteria defined in the target

database validations.
An example validation, which denies document deletion for non admins, look like this:

```
function(newDoc, oldDoc, userCtx, secObj) {
  if (newDoc._deleted &&
userCtx.roles.indexOf('_admin') === -1) {
    throw({
      forbidden: 'Only admins may delete user docs.'
    })
  }
}
```

The document to validate and the old document gets passed to the function, along with a User Context Object and Security Object

Data Migrations

Use pouchdb-migrate, a PouchDB plugin to help with migrations.
You can either run migration on startup (the plugin remembers if a replication has already been run, so the extra cost on startup is getting a single local document) or you can setup a daemon on the server.
Its also possible to run a migration manually from a dev computer.
Please take care of your migration function. It *must* recognize whether a doc already has been migrated, otherwise the migration is caught in a loop!

Per Document Access Control

There is no way to control access on a per document level.
A common solution is to have a database per user.
To create the dbs, you need to install a daemon. There are different projects:

- couchperuser (Erlang, donated to Apache CouchDB)
- couchdb-dbperuser-provisioning (Node)

An other way to achive per document access control, even on a

per field basis, is to use encryption. crypto-pouch and pouch-box might help.

Useful Meta Keys

In documents some information is generally useful. Rails, for example, inserts timestamps per default.:

- `created_at`: Date of creation
- `updated_at`: Date of last update
- `created_by`: Username of creator
- `updated_by`: Username of last updater
- `type`: Document type
- `version`: Document schema version

You can add validations, which takes care that those fields are set. Nonetheless you cannot guarantee that dates are correct - just that they are in the right format.

A Note on Dates

When saving dates, just store them as ISO 8601 strings, created by `.toISOString()` as well as by `JSON.stringify()`. That way the dates can be easily consumed by `new Date` or, who prefers, parsed by `Date.parse()`:

```
var date = new Date()
// Mon May 18 2015 16:50:52 GMT+0200 (CEST)
var datestring = date.toJSON()
// '2015-05-18T14:50:52.018Z'
new Date(datestring)
// Mon May 18 2015 16:50:52 GMT+0200 (CEST)
```

One To N Relations

Of course you can just use an array inside the document to store related data. This has a downside when the related data has to be updated, because conflicts can be created and this also introduces growing revisions which affects replication performance.

Its best to store related data in an extra document. You can use the id to link the documents together. That way you don't even need a view to fetch them together.

```
{
  "_id": "artist:tom-waits",
  "name": "Tom Waits"
}
[
  {
    "_id": "artist:tom-waits:album:closing-time",
    "title": "Closing Time"
  }
  {
    "_id": "artist:tom-waits:album:rain-dogs",
    "title": "Rain Dogs"
  }
  {
    "_id": "artist:tom-waits:album:real-gone",
    "title": "Real Gone"
  }
]
```

Now you can use the built-in `_all_docs` view to query the artist and all of its albums together, using start- and endkey:

```
{
  "include_docs": true,
  "startkey": "artist:tom-waits",
  "endkey": "artist:tom-waits:\ufff0"
}
```

N To N Relations

Similar to 1:N relations but use extra documents which describes each relation:

```
[
  {
    "_id": "person:avery-mcdonalid",
    "name": "Avery Mcdonalid"
  },
  {
    "_id": "person:troy-howell",
```

```
    "name": "Troy Howell"
  },
  {
    "_id": "person:tonya-payne",
    "name": "Tonya Payne"
  }
]
[
  {
    "_id": "friendship:person:avery-
mcdonalid:with:person:troy-howell",
    "since": "2015-05-13T08:57:53.786Z"
  },
  {
    "_id": "friendship:person:avery-
mcdonalid:with:person:tonya-payne",
    "since": "2015-03-02T07:21:01.123Z"
  }
]
```

Note how we used a deterministic order of friends in the id, we just sorted them alphabetically. Its important to be able to derivate the friendship document id from the constituting ids to make it easy to delete a relation.

Now write a map function like this:

```
function(doc) {
  if (!doc._id.match(/^friendship:/)) return

  var ids = doc._id.match(/person:[^:]*/g)
  var one = ids[0]
  var two = ids[1]

  emit([one, two], { _id: two })
  emit([two, one], { _id: one })
}
```

You now can query the view for one person:

```
{
  "include_docs": true,
  "startkey": ["person:avery-mcdonalid"],
  "endkey": ["person:avery-mcdonalid", {}]
}
```

and get all friends of that person:


```
{
  "total_rows" : 4,
  "rows" : [
    {
      "doc" : {
        "name" : "Tonya Payne",
        "_rev" : "1-7aafe790e8f4f00220c699e966246421",
        "_id" : "person:tonya-payne"
      },
      "value" : {
        "_id" : "person:tonya-payne"
      },
      "id" : "friendship:person:avery-
mcdonalid:with:person:tonya-payne",
      "key" : [
        "person:avery-mcdonalid",
        "person:tonya-payne"
      ]
    },
    {
      "value" : {
        "_id" : "person:troy-howell"
      },
      "doc" : {
        "_rev" : "1-7e0328a9eb72a5544663c30052c67161",
        "_id" : "person:troy-howell",
        "name" : "Troy Howell"
      },
      "key" : [
        "person:avery-mcdonalid",
        "person:troy-howell"
      ],
      "id" : "friendship:person:avery-
mcdonalid:with:person:troy-howell"
    }
  ],
  "offset" : 0
}
```

Views

Do Not Emit Entire Docs

You can query a view with `include_docs=true`. Then in the view result every row has the whole doc included:

```
{
  "rows": [
    {
      "id": "mydoc",
      "key": "mydoc",
      "value": null,
      "doc": {
        "_id": "mydoc",
        "_rev": "1-asd",
        "foo": "bar"
      }
    }
  ]
}
```

This has no disadvantages performance wise, at least on PouchDB. For CouchDB it means additional lookups and prevents CouchDB from directly streaming the view result from disk. But this is negligible. So don't emit the whole doc unless you need the last bit of performance.

Linked Documents

Or How do I do SQL-like JOINs? Can I avoid them?
CouchDB (and PouchDB) supports linked documents. Use them to join two types of documents together, by simply adding an `_id` to the emitted value in the map function :

```
// join artist data to albums
function(doc) {
  if (doc._id.match('^album:*')) {
    emit(doc._id, null);
    emit(doc._id, { '_id' : doc.artistId });
  }
```

```
}
```
When you query the view with the id of an album:
```
{
  "include_docs": true,
  "key": "album:hunkydory"
}
```
And this is a result:
```
{
  "rows": [
    {
      "doc": {
        "_id": "album:hunkydory",
        "title": "Hunky Dory",
        "year": 1971,
        "artistId": "artist:david-bowie"
      },
      "id": "album:hunkydory",
      "key": "album:hunkydory"
    }
    {
      "doc": {
        "_id": "artist:david-bowie",
        "firstName": "David",
        "lastName": "Bowie"
      },
      "id": "artist:david-bowie",
      "key": "album:hunkydory",
      "value": {
        "_id": "artist:david-bowie"
      }
    }
  ]
}
```
Using linked documents can be a way to group together related data.

Built-In Reduce Functions

CouchDB has some built in reduce functions to accomplish common tasks. They are native and very performant. Choose

them wherever possible. To use a built in reduce, insert the name string instead of the function code, eg

```
{
  "views": {
    "myview": {
      "map": "function(doc) { emit(doc._id, 1) }",
      "reduce": "_stats"
    }
  }
}
```

`_sum`
Adds up the emitted values, which must be numbers.

`_count`
Counts the number of emitted values.

`_stats`
Calculates some numerical statistics on your emitted values, which must be numbers. For example:

```
{
  "sum": 80,
  "count": 20,
  "min": 1,
  "max": 100,
  "sumsqr": 30
}
```

Debugging Views

View debugging can be a pain when you're restricted to Futon or even Fauxton. By using couchdb-view-tester you can write view code in your preferred editor and watch the results in real time.

Testing Views

Use couchdb-ddoc-test, a simple CouchDB design doc testing tool.

```
var DDocTest = require('couchdb-ddoc-test')
var test = new DDocTest({
    fixture: {a: 1},
      src: 'path/to/map.js'
})
var result = test.runMap()

assert.equals(result, fixture)
```

Deploying Views

You want to keep your view code in VCS. Now you need a way to install the view function. You can use the Python Couchapp Tool. Its a CLI which reads a directory, compiles documents from it and saves the documents in a CouchDB. Since this is a common task a quasi standard for the directory layout has been emerged, which is supported by a range of compatible tools. For integration in JavaScript projects using a tool written in JavaScript avoids additional dependencies. I wrote couch-compile for that purpose. It reads a directory, JSON file or node module and converts it into a CouchDB document, which can be deployed to a CouchDB database with couch-push. There is also a handy Grunt task, grunt-couch which integrates both projects into the Grunt toolchain.

Modularize View Code

While you work with views at some point you might want to share code between views or simply break your code into smaller modules. For that purpose CouchDB has support for CommonJS modules, which you know from node. CouchDB implements CommonJS 1.1.1.
Take this example:

```
var person = function(doc) {
  if (!doc._id.match(/^person:/)) return
  return {
    name: doc.firstname + ' ' + doc.lastname,
    createdAt: new Date(doc.created_at)
```

```
  }
}

var ddoc = {
  _id: '_design/person',
  views: {
    lib: {
      person: "module.exports = " + person.toString()
    },
    'people-by-name': {
      map: function(doc) {
        var person = require('views/lib/person')(doc)
        if (!person) return
        emit(person.name, null)
      }.toString(),
      reduce: '_count'
    },
    'people-by-created-at': {
      map: function(doc) {
        var person = require('views/lib/person')(doc)
        if (!person) return
        emit(person.createdAt.getTime(), null)
      }.toString(),
      reduce: '_count'
    }
  }
}
```

CommonJS modules can also be used for shows, lists and validate_doc_update functions. Note that the `person` module is inside the `views` object. This is needed for CouchDB in order to detect changes on the view code. Reduce functions *can NOT* use modules.
Read more about CommonJS modules in CouchDB.

View Collation

View Collation basically just means the concept to query data by ranges, thus using `startkey` and `endkey`. In CouchDB keys does not necessarily be strings, they can be arbitrary JSON values. If

thats the case we talk about *complex keys*.
Making use of View Collation enables us to query related documents together. Its the CouchDB way of doing *joins*. See Linked Documents.
Complex Keys order in the following matter:

```
1  null, false, true
2  1, 2...
3  a, A, b...\ufff0
4  ["a"], ["b"], ["b","c"]
5  {}, {a:1}, {a:2}, {b:1}
```

Read more about this topic in CouchDB "Joins" by Christopher Lenz and in the CouchDB docs about View Collation

Group Level

Consider the follwing documents

```
[
  { "_id": "1", "date": "2014-05-01T00:00:00.000Z",
"temperature": -10.00 },
  { "_id": "2", "date": "2015-01-01T00:00:00.000Z",
"temperature":  -7.00 },
  { "_id": "3", "date": "2015-02-01T00:00:00.000Z",
"temperature":  10.00 },
  { "_id": "4", "date": "2015-02-02T00:00:00.000Z",
"temperature":  11.00 },
  { "_id": "5", "date": "2015-02-02T01:00:00.000Z",
"temperature":  12.00 },
  { "_id": "6", "date": "2015-02-02T01:01:00.000Z",
"temperature":  12.20 },
  { "_id": "7", "date": "2015-02-02T01:01:01.000Z",
"temperature":  12.21 }
]
```

And the map function:

```
function(doc) {
  var date = new Date(doc.date)

  emit([
    date.getFullYear(),
    date.getMonth(),
```

```
    date.getDay(),
    date.getHours(),
    date.getMinutes(),
    date.getSeconds()
  ], doc.temperature)
}
```

And the build in reduce function `_stats`.
When you not query the view you get the stats over all entries:

```
{
  "rows" : [
    {
      "value" : {
        "max" : 12.21,
        "sumsqr" : 811.9241,
        "count" : 7,
        "sum" : 40.41,
        "min" : -10
      },
      "key" : null
    }
  ]
}
```

`group_level=1` gives you

```
{
   "rows" : [
      {
         "key" : [
            2014
         ],
         "value" : {
            "sumsqr" : 100,
            "count" : 1,
            "max" : -10,
            "min" : -10,
            "sum" : -10
         }
      },
      {
         "key" : [
            2015
```

```
         ],
         "value" : {
            "count" : 6,
            "sumsqr" : 711.9241,
            "min" : -7,
            "max" : 12.21,
            "sum" : 50.41
         }
      }
   ]
}
```


Upto `group_level=7` (which is the same as `group=true`)


```
{
   "rows" : [
      {
         "key" : [
            2014,
            4,
            4,
            2,
            0,
            0
         ],
         "value" : {
            "max" : -10,
            "count" : 1,
            "sum" : -10,
            "min" : -10,
            "sumsqr" : 100
         }
      },
      ...
      {
         "value" : {
            "min" : 12.21,
            "sum" : 12.21,
            "sumsqr" : 149.0841,
            "max" : 12.21,
            "count" : 1
         },
         "key" : [
```

```
            2015,
            1,
            1,
            2,
            1,
            1
         ]
      }
   ]
}
```

Combining this with key or range queries you can get all sort of fine grained stats.

Naming Conventions For Views

Naming convention for views, starting from the basic case of no reduce functions. Views are couples of arbitrary functions, and as such it is impossible to express their whole variety with a name, so I am just trying to cover the most common cases.

No Reduce

In the case of no reduce function, usually views are just meant to sort documents by a set of properties. An idea in this case is to name them like this:

The main grouping parameter for the name is the “object” as it (roughly) exists in the application space. For example: `person`, `contact`, `address` etc.:

```
<object>-
```

What follows is a condition for a subset of objects, e.g. `persons-with-address`:

```
<object>-with-<property>
<object>-with-no-<property>
```

Then, there is a sort option, e.g. `persons-with-address-by-createddate`:

```
<option>-with-<property>-by-<sortfield>
```

`with` and `by` clauses are both optional.

Finally, there is an optional suffix that describes whether the view

emits the full document as a value, e.g. `persons-with-address-fulldoc`:
`<object>-with-<property>-fulldoc`
When a property is nested, just replace the dot `.` with a dash `-`. Convert cases to lowercase. Convert underscore separated to no separation.

Examples

All people documents:
`people`
All cases sorted by doc.lastModifiedDate:
`cases-by-lastmodifieddate`
All addresses that have a city property sortedby `doc.createdDate`:
`address-with-city-by-createddate`

With a reduce function

In this case, i would use the same convention, with a prefix expressing the reduce. The reduce part could be structured as follows:
`[count|sum|stats]-[on-<property>-]<map part>`

Examples

`stats-on-patient-age-by-case-status`
The case above refers to built-in reduce functions, which should cover the wide majority of uses.

Replication

Filtered Replication

Filtered replication is a great way limit the amount of data synchronized on a device.
Filters can be by id (`_doc_ids`), by filter function, or by view (`_view`).
Be aware that replication filters other than `_doc_ids` are very

slow, because they run on *every* document. Consider writing those filter functions in Erlang.
When using filtered replication think about deleted documents, and whether they pass the filter. One way is to filter by id (this might be an argument for keeping `type` in the id).
Or deletion can be implemented as an update with `_deleted: true`. That way data is still there and can be used in the filter.

Using Replication

There are two ways to start a replication: the `_replicator` database and the `_replicate` API endpoint. Use `_replicator` database when in doubt.

`_replicate` API endpoint

When you use Futon you use the replicator endpoint. To initiate a replication post a JSON:

```
{
  "source": "a-db",
  "target": "another-db",
  "continuous": true
}
```

The response includes a replication id (which can also be obtained from `_active_tasks`):

```
{
  "ok": true,
  "_local_id":
"0a81b645497e6270611ec3419767a584+continuous"
}
```

Having this id you can cancel a continuous replication by posting

```
{
  "replication_id":
"0a81b645497e6270611ec3419767a584+continuous",
  "cancel": true
}
```

to the `_replicate` endpoint.

`_replicator` Database

Replications created via the `_replicator` database are persisted and survive a server restart. Its just a normal database which means you have the default operations. Replications are initiated by creating replication documents:

```
{
  "_id": "initial-replication:a-db:another-db",
  "source": "a-db",
  "target": "another-db",
  "continuous": true
}
```

Look, you can have meaningful ids! Cancelling is straight forward - just delete the document. Read more about the `_replicator` database.

Conflicts

Conflict Handling

Some things need to and should be conflicts. CouchDB *conflicts are first class citizens*, (or at least should be treated as such). If 2 different users enter different addresses for the same person at the same time, that probably should create a conflict. Your best option is to have a conflict resolution daemon running on the server. While we don't have this at the moment, look at the pouch-resolve-conflicts plugin.

Deployment

CouchDB Behind A Proxy

Running CouchDB behind a proxy is recommended, e.g. to handle ssl termination.

Prefer subdomain over subdirectory. Nginx encodes urls on the

way through. So, for example, if you request `http://my.couch.behind.nginx.com/mydb/foo%2Fbar` it gets routed to CouchDB as `/mydb/foo/bar`, which is not what we want.
We can configure this mad behaviour away (by not appending a slash to the `proxy_pass` target). But there is no way to convince nginx not messing with the url when rewriting the proxy behind a subdirectory, e.g. `http://my.couch.behind.nginx.com/_couchdb/mydb/foo%2Fbar`

Misc

Debugging PouchDB

I often assign the database instance to window and then I run queries on it. Or you can replicate to a local CouchDB and debug your views there.
If you like PouchDB to be more verbose, enable debug output:

```
PouchDB.debug.enable('*')
```

PouchDB and AngularJS

To use PouchDB with AngularJS you should use angular-pouchdb, which wraps PouchDBs promises with Angulars `$qs`.
Debugging angular-pouchdb in a console can be done by first retrieving the injector and calling the `pouchDB` service as normal, e.g.:

```
var pouchDB =
angular.element(document.body).injector().get('pouchDB'
)
var db = pouchDB('mydb')
db.get('id').then()
```

Full Text Search

While basic full text search is possible just by using views, its not convenient and you should make use of a dedicated FTI.

Client Side

For PouchDB the situation is clear: Use PouchDB Quick Search. Its a PouchDB plugin based on lunr.js.

Server Side

On a CouchDB server you have options: Lucene (via couchdb-lucene) or ElasticSearch via The River. At eHealth Africa we use the latter.

Two Ways of Deleting Documents

There are two ways to delete a document: via DELETE or by updating the document with a `_deleted` property set to true:

```
{
  "_id": "mydoc",
  "_rev": "1-asd",
  "type": "person",
  "name": "David Foster Wallace",
  "_deleted": true
}
```

In either way the deleted document will stay in the database, even after compaction. (That way the deletion can be propagated to all replicas.) Using the manual variant allows you to keep data, which might be useful for filtered replication or other purposes. Otherwise all properties will get removed except the plain stub:

```
{
  "_id": "mydoc",
  "_rev": "2-def",
  "_deleted": true
}
```