Rapport à propos du système NoSQL
CouchDB

Arthur BREUNEVAL, Nicolas GILLE et Grégoire POMMIER

Version 1.0, 17 février 2018

Table des Matières

Remerciements

Nous tenions a remercier Mme. SOUALMIA pour avoir respectivement assurée les cours et les TP de Big Data et d’avoir répondu à toutes nos questions aux cours de l’année scolaires 2017 / 2018.

Procédure d'installation

Pour installer **CouchDB** sur la machine de test [1: [computer-configuration]], il suffit d'effectuer les commandes suivante : `dnf install couchdb`.
Une fois celle-ci faite, lancer la commande suivante : `# couchdb` pour lancer le système *CouchDB*, vous obtiendrez alors le message suivant :

```
# couchdb
Apache CouchDB 1.7.1 (LogLevel=info) is logging to /var/log/couchdb/couch.log.
Apache CouchDB has started. Time to relax.
```

Cela nous indique donc que le système à démarrer et que nous sommes désormais prêt à l'utiliser.

Architecture de CouchDB

Présentation du système CouchDB

Une base NoSQL

CouchDB est une base faisant partie des bases NoSQL (Not Only SQL) les plus utilisées à l'heure actuelle. En effet, on y retrouve les caractèristiques typique d'une base NoSQL, comme on pourra le voir dans le schéma ci-dessous présentant le **Théorème de Brewer** ou **Théorème du CAP**, que nous ne redéfinirons pas ici.

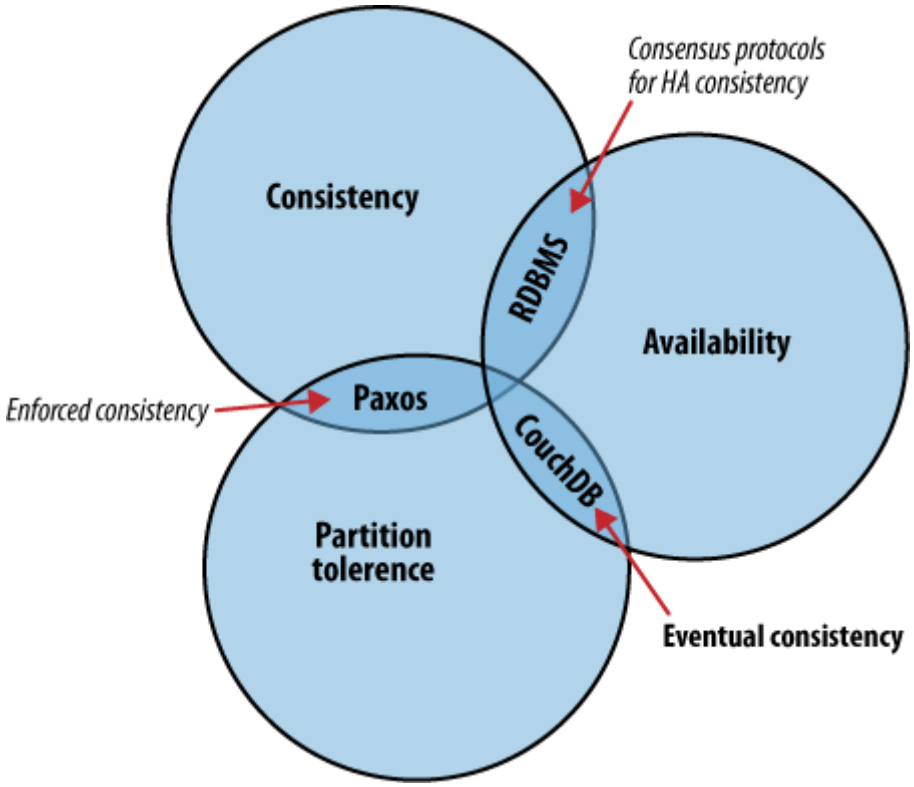

Figure 1. Théorème de Brewer

CouchDB fait parti des systèmes ayant pour but de répondre d'avantage à la **Disponibilité (Availability) et la Tolérance au partitionnement (Partition Tolerance)** des données, celui-ci réponds à des problèmatiques basées davantage sur l'accessibilité constante aux services ainsi que son partionnement sur un nombre important de serveurs répartis en divers point géographique.

> Tout nœud d'un système devrait pouvoir prendre des décisions en se basant uniquement sur l'état local. Si vous avez besoin de quelque chose, que vous êtes sous pression, que des problèmes se produisent, et que vous devez obtenir un accord, vous êtes perdu. Si vous vous préoccupez du passage à l'échelle, tout algorithme qui vous force à obtenir un accord deviendra inexorablement votre goulot d'étranglement. Soyez-en certain.
>
> — Werner Vogels, Directeur de la technologie et Vice-président d'Amazon

Si la disponibilité est la priorité, nous pouvons laisser un client écrire sur un nœud de la base de données sans attendre l'accord des autres nœuds. Si la base de données est capable de réconcilier ces opérations avec les autres nœuds, nous obtenons une sorte de « cohérence finale » en échange de la haute disponibilité. Étonnamment, c'est un compromis souvent acceptable pour les applications.

À la différence des bases de données relationnelles, où chaque action effectuée est nécessairement sujette à des contrôles d'intégrité, CouchDB facilite aisi la conception d'applications qui sacrifient la cohérence immédiate au profit de bien meilleures performances rendues possibles par une distribution simple des données.

Anatomie du requêtage du système

Pour bien comprendre le fonctionnement de CouchDB, il est nécessaire de voir son fonctionnement sur un point unique, à savoir une seule instance de base de données. Vous trouverez ci-dessous un schéma montrant le fonctionnement interne de l'API de recherche, d'ajout, de suppression et de modification des données présente sous forme de document **Json**.

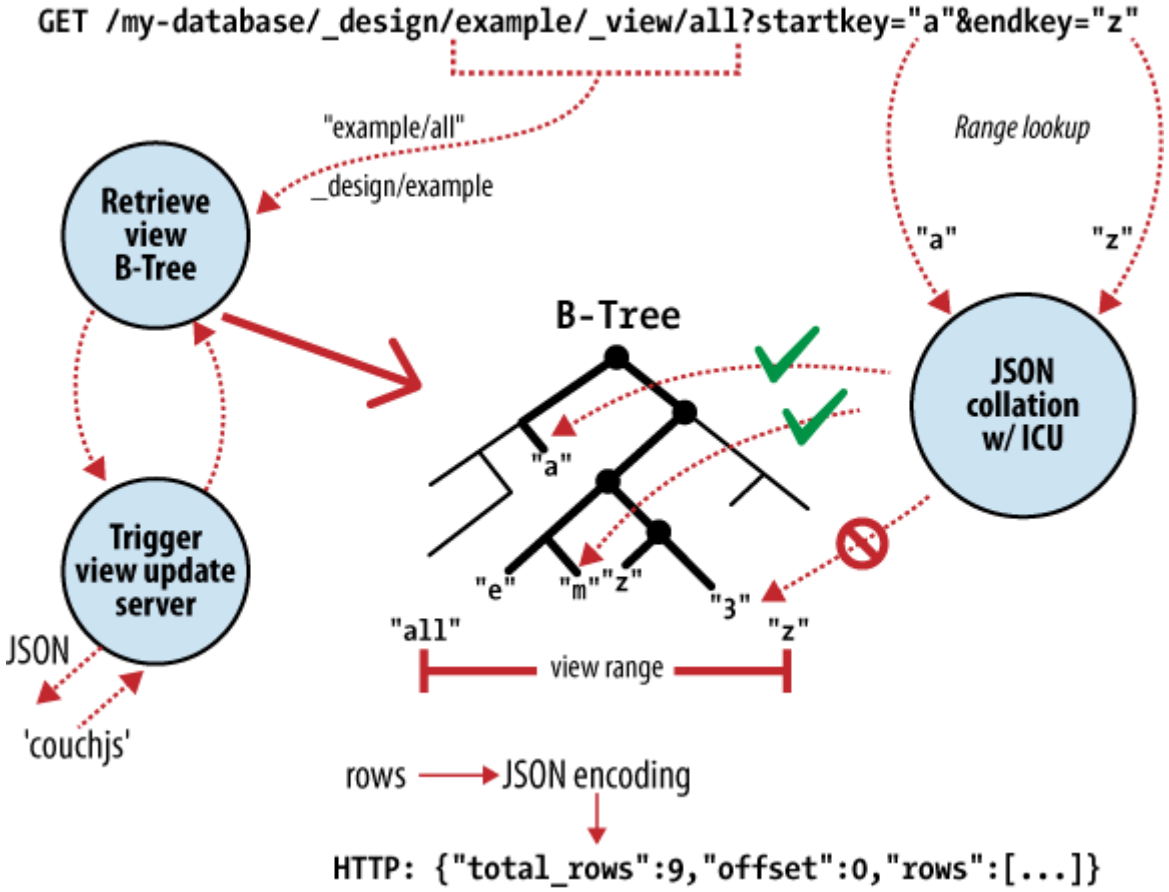

Figure 2. Anatomie d'une requête CouchDB

On observe que CouchDB utilise comme moteur de stockage en **B-Tree** aussi appelé **Arbre Equilibré**. Un arbre B est une structure ordonnée permettant la recherche, l'insertion et la suppression avec un temps de traitement logarithmique.

Couplé a ce moteur de recherche, CouchDB utilise aussi l'algorithme **MapReduce**, pour trouver les résultats d'une vue. Cette algorithme utilise deux fonctions, à savoir **subdiviser et agréger** qui sont appliquées sur chaque document indépendamment des autres. Cette application séparée de ces

étapes pour chaque document implique ainsi le calcul d'une vue peut être parallélisé et puisse être calculé de manière incrémental. Plus important encore, ces deux fonctions produisent un couple de clé/valeur, ce qui permet à CouchDB de stocker les résultats dans l'arbre B, trié par clé. Or, les recherches par clé, ou intervalle de clés, sont d'une rapidité redoutable dans un arbre B. Traduit en notation de complexité (O), cela donne respectivement O(log(N)) et O(log(N + K)).

Avec CouchDB, nous accédons aux documents et aux vues par clé ou intervalle de clés. C'est une correspondance directe avec les opérations sous-jacentes effectuées par le moteur de stockage en arbre B de CouchDB. En joignant les insertions et mises à jour de documents, cette correspondance directe explique que nous parlions de fine couche d'interface entourant le moteur de base de données pour décrire l'API.

Être capable d'accéder aux enregistrements uniquement à l'aide de leur clé est une contrainte très importante qui nous permet d'obtenir des gains de performance impressionnants. En plus de ces gains colossaux de rapidité, nous pouvons répartir les données sur plusieurs nœuds sans perdre la faculté de requêter chaque nœud indépendamment.

Accès concurrenciel aux documents

Dans tout système relationnel classique, si un ensemble d'utilisateur souhaite accéder aux données de manières simultanées, le SGBDR mets en place un **verrou**, qui implique qu'un seul utilisateur peut accéder à la donnée à un **instant t**, les autres étant mise en **attente** par le système.

Plutôt que de recourir aux verrous, CouchDB utilise un système de **Multi-Version Concurrency Control (MVCC)** pour gérer les accès concurrents à la base. Vous trouverez ci dessous une illustration présentant le **MVCC** fasse aux systèmes de verrous d'un SGBDR classique.

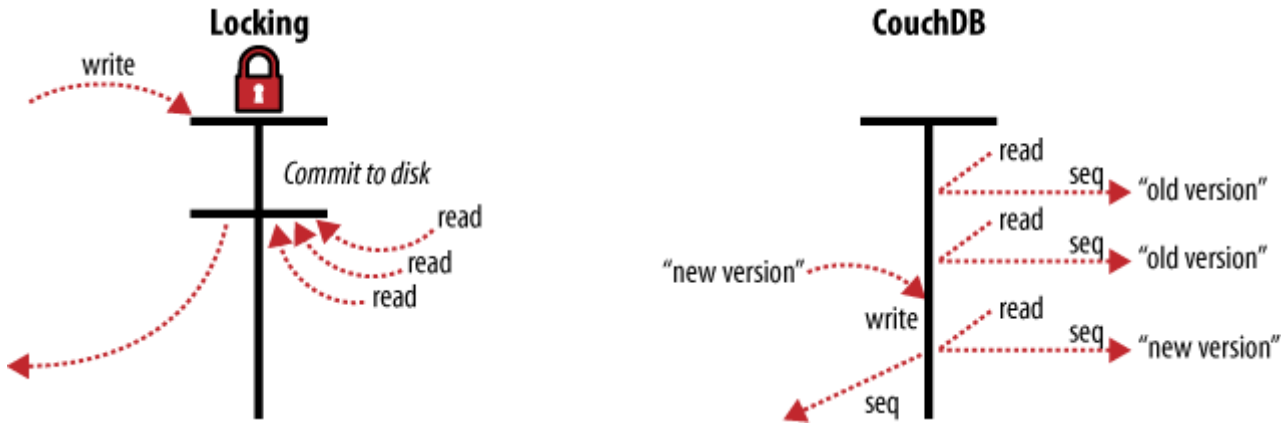

Figure 3. Principe du Multi-Version Concurrency Control

On remarque sur le schéma que les documents sont tous versionnées au fur et à mesure de leurs modifcations, comme on le retrouve dans un système de versionning de fichier, tels que **Subversion** ou **Git**. Ceci à pour avantage que si un utilisateur lambda utilise la version 1.0 d'un document et que dans le même temps un second utilisateur modifie les données de celui-ci. L'utilisateur 1 pourra toujours manipuler son document de manière transparante, sans avoir eu vent de la modification apporté par l'utilisateur 2 et inversement. Ainsi, une requête de lecture renverra toujours à la dernière version du document, et si une requête précédente avait requêté sur le document avec une version ultérieure, celle-ci reste utilisable sans problème.

Distributivité des données

Maintenir la cohérence au sein d'un unique nœud de base de données est relativement simple pour la plupart des bases. Les réels problèmes surviennent lorsqu'il s'agit de faire la même chose entre plusieurs serveurs. Si un client écrit sur le serveur A, comment s'assurer que c'est cohérent avec le serveur B, ou C, ou D ?
Pour les bases de données relationnelles, c'est un problème complexe avec des livres entiers qui traitent du sujet. Vous pourriez utiliser des topologies de réplication multimaître, maître-esclave, partitionner, fragmenter, disposer des caches d'écriture et toutes sortes de techniques compliquées.

Cohérence Distribuée

Puisque les actions que vous effectuez sur le système ce base uniquement sur les documents qu'ils contiennent, il n'est plus nécessaire de garder une communication constante entres les serveurs pour que les données soient toujours valide.

En effet, CouchDB a mis en place un système de **Réplication Incrémentale**, permettant une **cohérence finale** entre les bases de données. Ce processus consiste a copier l'ensemble des données entres les diverses bases de données mise en place dans le **cluster** de manières périodiques entres elles. Nous arrivons ainsi sur un système appelé **shared-nothing cluster**, signifiant que les noeuds sont indépendants et autosuffisant. Vous trouverez ci-dessous le principe de **réplication incrémentale**, sous forme schématique pour mieux saisir le principe.

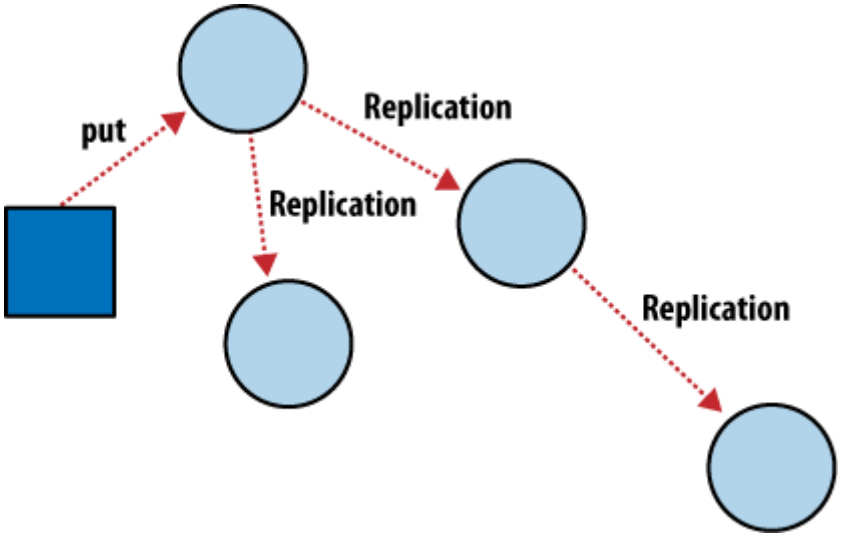

Figure 4. Réplication incrémental des noeuds

Cette réplication peut être lancer de manière complétement automatique par des tâche **cron** présente sur les serveurs, ou bien de manière manuelle par tout utilisateur ayant les droits de réplication sur le système.

Résolution des conflits

En effet, supposons qu'un document soit modifier dans deux serveurs différents, avant que ceux-ci soient n'ai subit la réplication incrémentale, que se passerait-il lorsque la réplication sera mis en oeuvre ?

CouchDB est capable de résoudre ce genre de problème de la même façon que via un système de

versionning classique, c'es à dire en lui applicant un **flag** indiquant qu'il y a conflit entre deux versions de ce document.

Dans ce cas la, une version est marquée comme **gagnante** et l'autre comme **perdante**. La gagnante devient ainsi la dernière version stable connue du fichier, tandis que la perdante est stocké dans le système comme étant une version ultérieur à la version **gagnante** du document. Ce choix est ainsi répercuté sur l'ensemble des bases composant le système, respectant ainsi la cohérence de l'ensemble des données du système.

Il est aussi possible de gérer soit même le conflit, en fonction de la configuration que l'on a mis en place dans le système, soit en tentant de fusionner les deux documents, soit en revenant à la version antérieur du document.

Présentation des méthodes d'interaction avec le système

Fonctionnement de CouchDb

couchDb fonctionne via une architecture REST. L'idée est ici de pouvoir requêter le système de base de donnée directement via des requêtes **HTTP**, qui seront interprétés par le système afin d'effectuer les actions voulues par l'utilisateur est retourner les jeux de données demandées.
Pour cela, nous allons définir ensemble, pour les différentes actions possible, l'URL qui sera utilisé pour affecté le système de base de données. De plus, nous allons définir la méthode de requêtage HTTP à utiliser pour que l'action soit prise en compte. Définissons rapidement les différentes méthodes de requêtes HTTP : * GET : Permet de demander une ressource tels que un document ou une Vue. * POST : Permet d'ajouter un document aux sysyèmes. * PUT : Permet d'effectuer une modification sur le système tel que la modification d'une base de données, d'un document, ... * DELETE : Permet la suppression de ressources, de base de données, ...

Manipulation des bases de données

Création d'une base de données

Afin de créer une base de données, il faut envoyer une requête PUT sur l'url suivante `http://127.0.0.1:5894/{databae_name}` avec **database_name** le nom de le base de données que l'on souhaite créer. Par exemple, la commande suivante crée une base de donneés du nom de **big_data** et retourne le code HTTP correspond indiquant la réussite ou l'échec du processus.

```
# curl -X PUT http://127.0.0.1:5984/big_data
{"ok":true}
```

Le code `{"ok":true}` indique que la base de données a été créée correctement.

Listing de toutes les bases de données du système

Maintenant que nous avons créée une base de données, nous souhaitons voir l'ensemble des bases de données déjà présente dans le système. Pour cela, nous allons requêter en GET l'URL du système avec comme suffixe **_all_dbs** qui nous permettra de voir l'ensemble des bases en système.

```
$ curl -X GET http://127.0.0.1:5984/_all_dbs
["_replicator","_users","big_data"]
```

Nous remarquons qu'il existe déjà 2 base de données (_replicator et _users) ainsi que la nouvelle base de données créer précédement.

Récupération des informations relatives à une base

CouchDB propose de voir les informations relative a une base de données de la manière suivante : `$ http://127.0.0.1:5984/{database_name}`.
Les données récupérées indique entre autres le nom de la base, son nombre de documents, le nombre de documents supprimé, ...

```
$ curl -X GET http://127.0.0.1:5984/_replicator
{
  "db_name":"_replicator",
  "doc_count":1,
  "doc_del_count":0,
  "update_seq":1,
  "purge_seq":0,
  "compact_running":false,
  "disk_size":4194,
  "data_size":2006,
  "instance_start_time":"1518710799423032",
  "disk_format_version":6,
  "committed_update_seq":1
}
```

Suppression d'une base de donnée

Pour supprimer une base de données, il suffit de lancer la requête sur la base de données que l'on souhaite, via une requête DELETE, en précisant le nom de la base de données que l'on souhaite supprimer.

```
$ curl -X DELETE http://127.0.0.1:5984/big_data
{
   "ok" : true
}
```

Manipulation des documents

Ajout d'un document dans la base de données

Pour ajouter des données dans la base de données, il existe ainsi 2 solutions : en ligne de commande ou via l'interface grpahique. Nous verrons ici uniquement la façon de faire en ligne de commande.

Ainsi, pour ajouter des données, il faut taper sur cette url via la méthode PUT : `http://127.0.0.1:5984/{database_name}/{id} -d '{ json_content }'`. On retrouve donc le nom de la base de données, l'identifiant du fichier devant être unique dans la base de données, puis l'option **-d** accompagné du fichier json que l'on souhaite envoyer en base de données.

Il est ainsi possible de définir un script permettant l'insertion d'un nombre important de données,

cependant, il m'a été impossible de le réaliser sur ma machine, celle-ci crashant au bout de quelques minutes d'insertion, sans que je puisse comprendre d'ou vienne le problème. J'ai donc due en ajouter un certain nombre à la main, ce qui m'a pris énormément de temps.

```
$ curl -X PUT http://127.0.0.1:5984/big_data/001 -d '{ ... }'
{
  "ok":true,
  "id":"001",
  "rev":"1-1c2fae390fa5475d9b809301bbf3f25e"
}
```

Mise à jour d'un document de la base

Pour mettre à jour un document, il suffit de requêter l'url suivante `http://127.0.0.1:5984/{database_name}/{id} -d '{ json_content }` via la méthode PUT en modifiant le contenu du document initial avec ces nouvelles valeurs. Pour identifier le document, on utilise l'**id** que l'on a utilisé pour son insertion, auquel on adjoint le nouveau contenu que l'on souhaite y intégrer.
Cela aura pour effet de modifier le champs **_rev** qui indique le nombre de modification que l'on a apporté aux documents, permettant ainsi de facilité le mécanisme de réplication des données.

Suppression d'un document de la base

Le mécanisme de suppression fonctionne de manière identique à la modification des données, sauf que cette fois-ci, l'on requête l'url via la méthode DELETE et non plus PUT, comme l'on faisait précédement pour la modification. De plus, nous n'ajoutons plus l'option **-d** ainsi que le contenu du fichier, car ici, nous supprimons uniquement le document de la base de données. Voici donc un exemple de requête de suppression du document ajouter précédement.

```
$ curl -X PUT http://127.0.0.1:5984/big_data/001
{
  "ok":true,
  "id":"001",
  "rev":"2-1c2fae390fa5475d9b809301bbf3f25e"
}
```

Requête d'un ensemble de documents

Nous nous baserons sur des documents stockant le prix d'articles de supermarché comme on en trouve dans divers magasins

Voici les différents fichiers json qui seront utilisés pour la suite de la présentation du fonctionnement de CouchDB.

apple_file.json

```
{
    "_id" : "bc2a41170621c326ec68382f846d5764",
    "_rev" : "2612672603",
    "item" : "apple",
    "prices" : {
        "Fresh Mart" : 1.59,
        "Price Max" : 5.99,
        "Apples Express" : 0.79
    }
}
```

citrus_file.json

```
{
    "_id" : "bc2a41170621c326ec68382f846d5764",
    "_rev" : "2612672603",
    "item" : "orange",
    "prices" : {
        "Fresh Mart" : 1.99,
        "Price Max" : 3.19,
        "Citrus Circus" : 1.09
    }
}
```

banana_file.json

```
{
    "_id" : "bc2a41170621c326ec68382f846d5764",
    "_rev" : "2612672603",
    "item" : "banana",
    "prices" : {
        "Fresh Mart" : 1.99,
        "Price Max" : 0.79,
        "Banana Montana" : 4.22
    }
}
```

Utilisation de la fonction de MapReduce

Voici un exemple de la fonction Map que l'on peut utiliser afin subdiviser les résultats des documents afin de les regrouper plus tard dans la fonction **Reduce**.

map.js

```
function(doc) {
    var store, price, value;
    if (doc.item && doc.prices) {
        for (store in doc.prices) {
            price = doc.prices[store];
            value = [doc.item, store];
            emit(price, value);
        }
    }
}
```

Ce script permet d'afficher la **Vue** finale de nos documents après requête.

view.js

```
function(doc) {
    var store, price, key;
    if (doc.item && doc.prices) {
        for (store in doc.prices) {
            price = doc.prices[store];
            key = [doc.item, price];
            emit(key, store);
        }
    }
}
```

Validation des documents

Il est possible d'écrire des fonctions de validations des documents reçus afin d'avoir un contrôle absolu sur les données que l'on souhaite stocker.
En effet, il peut être intéressant de ne stockers des données que sous un certain format, ou d'ajouter un certains nombre de champs, notamment le nom de l'auteur, la date de création ainsi que d'autres éléments supplémentaire.

Vous trouverez ci-dessous un exemple de script de validation de données que l'on applique a tout documents issues d'une requête **POST**, soit l'insertion d'un nouveau document.

validate.js

```
function(newDoc, oldDoc, userCtx) {
  function require(field, message) {
    message = message || "Document must have a " + field;
    if (!newDoc[field]) throw({forbidden : message});
  };

  if (newDoc.type == "post") {
    require("title");
    require("created_at");
    require("body");
    require("author");
  }
  if (newDoc.type == "comment") {
    require("name");
    require("created_at");
    require("comment", "You may not leave an empty comment");
  }
}
```

Utilisation d'un "Template"

En effet, la vue que l'on expose sur la partie précédente retourne le json brute, sans aucun traitement. Hors, il est fréquent que ces requêtes soient lancer depuis une application web.
De ce fait, CouchDB propose un mécanisme de template qu'il chargera lors de la requête d'une **vue** précise, que se soit pour ajouter, consulter ou modifier des données.

template.js

```
function(doc, req) {
  !json templates.edit
  !json blog
  !code vendor/couchapp/path.js
  !code vendor/couchapp/template.js

  // we only show html
  return template(templates.edit, {
    doc : doc,
    docid : toJSON((doc && doc._id) || null),
    blog : blog,
    assets : assetPath(),
    index : listPath('index','recent-posts',{descending:true,limit:8})
  });
}
```

Nous n'expliquerons pas dans les détails l'ensemble des attributs, car ceux-ci sont définis de manière plus complète dans la documentation de l'outil.

Utilisation de l’interface graphique

CouchDB possède en interne un service web graphique permettant d’intéragir avec le système directement depuis un navigateur web quelconque. Pour cela, il vous suffit de taper `http://127.0.0.1:5984/_utils/` dans votre navigateur favori afin d’accéder à la partie graphique de **CoucheDB**.

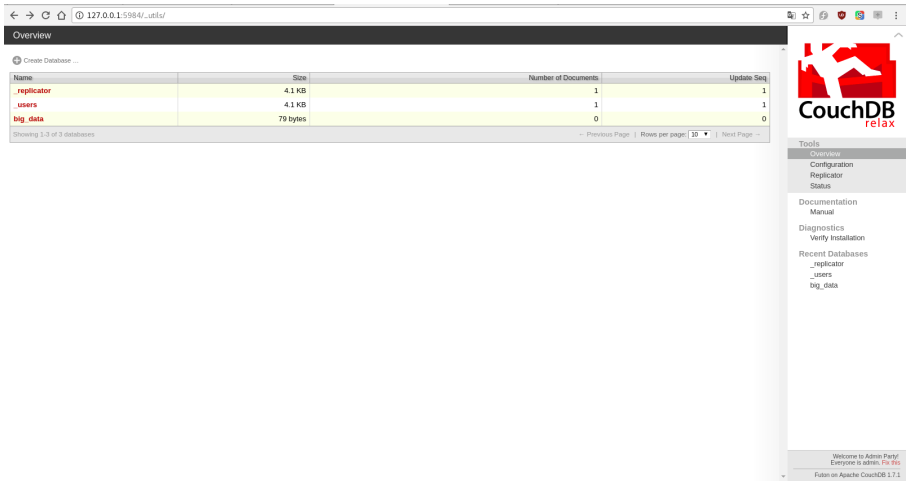

Figure 5. Interface graphique de CouchDB

Annexes

Annexe A: Configuration de la machine de test

Voici la configuration *hardware* de la machine sur lequel sont effectués l'ensemble des tests :

- OS : Fedora 27 / 64 Bits
- Processeur : Intel® Core™ i7-3610QM CPU @ 2.30GHz (8 Coeurs / 16 Threads).
- RAM : 6 Go.
- Carte Graphique : GeForce 630M 2GB dédiés.

Annexe B: Liens utiles

- CouchDB Apache, dernière consultation le 17 février 2018
- Guide CouchDB, dernière consultation le 17 février 2018

Annexe C: Auteurs

- Arthur BREUNEVEL
- Nicolas GILLE
- Grégoire POMMIER