Bases de données documentaires et distribuées

Version Janvier 2025

Philippe Rigaux

janv. 09, 2025

Table des matières

1 Introduction **3**
- 1.1 Sujet du cours . . . 4
- 1.2 Contenu et objectifs du cours . . . 5
- 1.3 Organisation . . . 6

2 Préliminaires : Docker **7**
- 2.1 Introduction à Docker . . . 9
- 2.2 Docker en pratique . . . 11
- 2.3 Nos systèmes NoSQL . . . 17
- 2.4 Exercices . . . 27

3 Modélisation de bases NoSQL **31**
- 3.1 S1 : documents structurés . . . 32
- 3.2 S2. Modélisation des collections . . . 41
- 3.3 Exercices . . . 52

4 Recherche exacte **61**
- 4.1 S1 : HTTP, REST, et CouchDB . . . 62
- 4.2 S2 : ElasticSearch . . . 71
- 4.3 S3 : le langage d'interrogation de MongoDB . . . 87

5 Etude de cas : Cassandra **95**
- 5.1 S1 : Cassandra, une base relationnelle étendue . . . 95
- 5.2 S2 : requêtes Cassandra . . . 104
- 5.3 S3 : étude de cas : conception d'un schéma . . . 110
- 5.4 Exercices . . . 116

6 Cassandra - Travaux Pratiques **119**
- 6.1 Partie 1 : Approche relationnelle . . . 120
- 6.2 Partie 2 : modélisation spécifique NoSQL . . . 126

7 Recherche approchée **129**
- 7.1 S1 : introduction à la recherche d'information . . . 129

7.2 S2 : L'analyse de documents . . . 137
7.3 S3 : recherche avec classement . . . 140
7.4 S4 : recherche plein texte . . . 143
7.5 Exercices . . . 149

8 ElasticSearch - Travaux pratiques 159
8.1 S1 : recherche plein texte avec ElasticSearch . . . 160
8.2 S2 : Agrégats . . . 169
8.3 S3 : Le classement dans Elasticsearch . . . 173

9 Traitements par lot 177
9.1 S1 : MapReduce démystifié . . . 178
9.2 S2 : MapReduce et CouchDB . . . 187
9.3 S3 : Spécification de traitements distribués avec Pig . . . 191
9.4 Exercices . . . 197

10 Pig : Travaux pratiques 205
10.1 Première partie : analyse de flux multiples . . . 205
10.2 Deuxième partie : analyse de requêtes . . . 208

11 Le *cloud*, une nouvelle machine de calcul 215
11.1 S1 : *cloud* et données massives . . . 216
11.2 S2 : La scalabilité . . . 227
11.3 3 : Calculs distribués . . . 230
11.4 Exercices . . . 237

12 Systèmes NoSQL : la réplication 245
12.1 S1 : réplication et reprise sur panne . . . 245
12.2 S2 : ElasticSearch . . . 255
12.3 S4 : Cassandra . . . 262
12.4 S4 : réplication dans MongoDB . . . 270
12.5 Exercices . . . 274

13 Systèmes NoSQL : le partitionnement 279
13.1 S1 : les bases . . . 279
13.2 S2 : partitionnement par intervalle . . . 287
13.3 S3 : partitionnement par hachage . . . 295
13.4 Exercices . . . 308

14 Etude de cas : Apache Spark 313
14.1 S1 : Introduction à Spark . . . 313
14.2 S2 : Mise en pratique . . . 320
14.3 S3 : Traitement de données structurées avec Cassandra et Spark . . . 327
14.4 Exercices . . . 333

15 Annales des examens 337
15.1 Examen du 3 février 2015 . . . 337
15.2 Examen du 14 avril 2015 . . . 339
15.3 Examen du 15 juin 2015 . . . 340

15.4 Examen du 1er juillet 2016 (FOD) . . . 342
15.5 Examen du 1er février 2017 (Présentiel) . . . 346
15.6 Examen du 6 février 2018 (Présentiel) . . . 349
15.7 Examen du 30 juin 2020 . . . 355
15.8 Examen du 5 septembre 2020 . . . 360
15.9 Examen du 28 juin 2023 . . . 361
15.10 Examen du 22 juin 2024 . . . 368

16 Indices and tables **373**

Tout le matériel proposé ici sert de support au cours « Bases de données documentaires et distribuées » proposé par le département d'informatique du Cnam. Le code du cours est NFE204 (voir le site http://deptinfo.cnam.fr/new/spip.php?rubrique146 pour des informations pratiques). Il est donné en

- Cours présentiel (premier semestre, mardi soir)
- Cours à distance (second semestre, avec supports audiovisuels)

Par ailleurs, le document que vous commencez à lire fait partie de l'ensemble des supports d'apprentissage proposés sur le site http://www.bdpedia.fr. Reportez-vous à ce site pour plus d'explications.

Ce cours fait partie d'un ensemble d'enseignements consacrés à l'analyse de données massives, permettant éventuellement d'obtenir un Certificat de Spécialisation au Cnam. Vous êtes invités à consulter :

- Le site du certificat : http://donneesmassives.cnam.fr/
- La fiche du certificat : http://formation.cnam.fr/rechercher-par-discipline/certificat-de-specialisation-analyste-de-donnees-massives-669531.kjsp
- La présentation du cours RCP216 sur la fouille de données distribuée http://cedric.cnam.fr/vertigo/Cours/RCP216/preambule.html
- La présentation du projet de synthèse (UASB03) qui conclut le Certificat de données massives, http://cedric.cnam.fr/vertigo/Cours/UASB03/uasb03.html

CHAPITRE 1

Introduction

Supports complémentaires :
— Diapositives: Présentation du cours

Les bases relationnelles sont adaptées à des informations bien structurées, décomposables en unités simples (chaînes de caractères, numériques), et représentables sous forme de tableaux. Beaucoup de données ne satisfont pas ces critères : leur structure est complexe, variable, et elles ne se décomposent par aisément en attributs élémentaires. Comment représenter le contenu d'un livre par exemple ? d'une image ou d'une vidéo ? d'une partition musicale ?

Les bases relationnelles répondent à cette question en multipliant le nombre de tables, et de lignes dans ces tables, pour représenter ce qui constitue conceptuellement une même « entité ». Cette décomposition en fragment « plats » (les lignes) est la fameuse *normalisation* (relationnelle) qui impose, pour reconstituer l'information complète, d'effectuer une ou plusieurs jointures assemblant les lignes stockées indépendamment les unes des autres.

Note : Ce cours suppose une connaissance solide des bases de données relationnelles. Si ce n'est pas le cas, vous risquez d'avoir des lacunes et des difficultés à assimiler les nouvelles connaissances présentées. Je vous recommande au préalable de consulter les cours suivants :
— le cours Bases relationnelles, modèles et langages, pour tout savoir sur la conception d'une base relationnelle et le langage SQL.
— le cours Systèmes relationnels, pour les aspects systèmes : indexation, optimisation, concurrence d'accès.

Cette approche, qui a fait ses preuves, ne convient cependant pas dans certains cas. Les données de nature essentiellement textuelle par exemple (livre, documentation) se représentent mal en relationnel ; c'est vrai aussi

de certains objets dont la stucture est très flexible ; enfin, *l'échange de données* dans un environnement distribué se prête mal à une représentation éclatée en plusieurs constituants élémentaires qu'il faut ré-assembler pour qu'ils prennent sens. Toutes ces raisons mènent à des modes de représentation plus riches permettant la réunion, en une seule structure, de toutes les informations relatives à un même objet conceptuel. C'est ce que nous appellerons *document*, dans une acception élargie un peu abusive mais bien pratique.

1.1 Sujet du cours

Dans tout ce qui suit nous désignons donc par le terme générique de *document* toute paire *(i, v)* où *i* est l'identifiant du document et *v* une *valeur structurée* contenant les informations caractérisant le document. Nous reviendrons plus précisément sur ces notions dans le cours.

La gestion d'ensembles de documents selon les principes des bases de données, avec notamment des outils de recherche avancés, relève des *bases documentaires*. Le volume important de ces bases amène souvent à les gérer dans un système distribué constitué de fermes de serveurs allouées à la demande dans une infrastructure de type « cloud ». L'usage est maintenant établi d'appeler ces systèmes « NoSQL » pour souligner leurs différences avec les systèmes relationnels. Le fait qu'ils ne suivent pas le modèle relationnel est d'ailleurs à peu près leur seul point commun. De manière générale, et avec de grandes variantes quand on se penche sur les détails, ils partagent également :

— la représentation des données sous forme d'unités d'information indépendantes les unes des unes, (ce que nous appelons justement *document*) organisées en *collections* ;
— des méthodes d'accès aux collections basées soit sur des langages non standardisés, soit sur des recherches par similarité qui relèvent de la *recherche d'information* ;
— la capacité à *passer à l'échelle* (expression énigmatique que nous essaierons de clarifier) par ajout de ressources matérielles, donnant ces fameux environnements distribués et extensibles,
— et enfin des techniques de distribution de calculs permettant de traiter des collections massives dans des délais raisonnables.

Tous ces aspects, centrés sur les documents de nature textuelle (ce qui exclut les documents multimédia comme les images ou vidéos), constituent le cœur de notre sujet. Il couvre en particulier :

— les modèles de données pour documents structurés (et, concrètemet, la syntaxe JSON), conception, bases de documents structurés (MongoDb, CouchDB, Cassandra, etc.).
— les techniques de recherche et d'interrogation, exacte ou approchée
— les traitements par lot de données massives
— la gestion de grandes collections dans des environnements distribués
— les traitements à grande échelle : Hadoop, MapReduce et Spark.

La *représentation* des données s'appuie sur un *modèle*. Dans le cas du relationnel, ce sont des tables (pour le dire simplement), et nous supposerons que vous connaissez l'essentiel. Dans le cas des documents, les structures sont plus complexes : tableaux, ensembles, agrégats, imbrication, références. Nous étudions essentiellement la notion de *document structuré* et son format de représentation le plus courant, JSON.

Disposer de données, mêmes correctement représentées, sans pouvoir rien en faire n'a que peu d'intérêt. Les *opérations* sur les données sont les créations, mises à jour, destruction, et surtout *recherche*, selon des critères plus ou moins complexes. Les bases relationnelles ont SQL, nous verrons que la recherche dans des grandes bases documentaires se fait soit par des recherches exactes (avec des langages propriétaires) soit par des recherches approchées (mis en œuvre par des moteurs de recherche).

Enfin, à tort ou à raison, les nouveaux systèmes de gestion de données, orientés vers ce que nous appelons, au sens large, des « documents », sont maintenant considérés comme des outils de choix pour passer à l'échelle

de très grandes masses de données (le « Big data »). Ces nouveaux systèmes, collectivement (et vaguement) désignés par le mot-valise « NoSQL » ont essentiellement en commun de pouvoir constituer à peu de frais des systèmes distribués, scalables, aptes à stocker et traiter des collections à très grande échelle. Une partie significative du cours est consacrée à ces systèmes, à leurs principes, et à l'inspection en détail de quelques exemples représentatifs.

1.2 Contenu et objectifs du cours

Le cours vise à vous transmettre, *dans un contexte pratique*, deux types de connaissances.

— **Connaissances fondamentales :**

1. *Modélisation de documents structurés* : structures, sérialisation, formats (JSON) ; les schémas de bases documentaires ; les échanges de documents sur le Web et notamment *l'Open Data*
2. *Techniques d'interrogation et de recherche* : recherche exacte et recherche approchée
3. *Stockage, gestion, et passage à l'échelle par distribution*. L'essentiel sur les systèmes distribués, le partitionnement, la réplication, la reprise sur panne ; le cas des systèmes NoSQL.
4. *Traitement de données massives* : Hadoop et MapReduce, et une introduction à Spark

— **Connaissances pratiques :**

1. Des systèmes « NoSQL » orientés « documents » ; (MongoDB, CouchDB, Cassandra)
2. Des moteurs de recherche (ElasticSearch)
3. L'étude, en pratique, de deux systèmes distribués : ElasticSearch, Cassandra.
4. La combinaison des moteurs de stockage et des moteurs de traitement distribué : Hadoop, Spark et Flink.

Les connaissances préalables pour bien suivre ce cours sont essentiellement une bonne compréhension des bases relationnelles, soit au moins la conception d'un schéma, SQL, ce qu'est un index et des notions de base sur les transactions.

Pour les aspects pratiques, il est souhaitable également d'avoir une aisance minimale dans un environnement de développement. Il s'agit d'éditer un fichier, de lancer une commande, de ne pas paniquer devant un nouvel outil, de savoir résoudre un problème avec un minimum de tenacité. Aucun développement n'est à effectuer, mais des exemples de code sont fournis et doivent être mis en œuvre pour une bonne compréhension.

Le cours vise à vous transmettre des connaissances génériques, indépendantes d'un système particulier. Il s'appuie cependant sur la mise en pratique. Vous avez donc besoin d'un ordinateur pour travailler. Si vous êtes au Cnam tout est fourni, sinon un ordinateur portable raisonnablement récent et puissant (8 GOs en mémoire RAM au minimum) suffit. Tous les logiciels utilisés sont libres de droits, et leur installation est décrite avec le système Docker.

1.3 Organisation

Le cours est découpé en *chapitres*, couvrant un sujet bien déterminé, et en *sessions*. J'essaie de structurer les sessions pour qu'elles demandent environ 2 heures de travail personnel (bien sûr, cela dépend également de vous). Pour assimiler une session vous pouvez combiner les ressources suivantes :

- La lecture du support en ligne : celui que vous avez sous les yeux, également disponible en PDF ou en ePub.
- Le suivi du cours, en vidéo ou en présentiel.
- La réponse au quiz pour valider votre compréhension
- La réalisation des exercices proposés en fin de session.
- Enfin, **optionnellement**, la reproduction des manipulations vues dans chaque session. N'y passez pas des heures :il vaut mieux comprendre les principes que de résoudre des problèmes techniques peu instructifs.

La réalisation des exercices en revanche est essentielle pour vérifier que vous maîtrisez le contenu. Pour les inscrits au cours, ils sont proposés sous forme de devoirs à rendre et à faire évaluer avant de poursuivre.

Vous devez assimiler le contenu des sessions *dans l'ordre où elles sont proposées*. Commencez par lire le support, jusqu'à ce que les principes vous paraissent clairs. Répondez alors au quiz de la session. Essayez de reproduire les exemples de code : ils sont testés et doivent donc fonctionner, sous réserve d'un changement de version introduisant une incompatibilité. Le cas échéant, cherchez à résoudre les problèmes par vous-mêmes : c'est le meilleur moyen de comprendre, mais n'y passez pas tout votre temps. Finissez enfin par les exercices. Les solutions sont dévoilées au fur et à mesure de l'avancement du cours, mais si vous ne savez pas faire un exercice, c'est sans doute que le cours est mal assimilé et il est plus profitable d'approfondir en relisant à nouveau que de simplement copier une solution.

Enfin, vous êtes totalement encouragés à explorer par vous-mêmes de nouvelles pistes. Certaines sont proposées dans les exercices.

CHAPITRE 2

Préliminaires : Docker

Supports complémentaires

— Diapositives: introduction à Docker
— Vidéo de la session consacrée à Docker

La plupart des systèmes étudiés dans ce cours peuvent s'installer et s'exécuter avec l'environnement *Docker* (http://www.docker.com). Docker permet d'émuler un système distribué de serveurs.

Un *serveur* est une entité qui fournit un service (!). Concrètement :

— un *serveur (machine)* est un ordinateur, tournant sous un système d'exploitation, et connecté en permanence au réseau via des *ports* ; un serveur machine est identifiable sur le réseau par son adresse IP.
— un *serveur (logiciel)*, ou *service*, est un processus exécuté en tâche de fond d'un serveur machine qui communique avec des *clients (logiciels)* via un port particulier.
— un *système distribué* est constitué de plusieurs services qui communiquent les uns avec les autres ; ces services peuvent ou non être répartis sur plusieurs serveurs-machine.
— un *client (logiciel)* est un programme qui communique avec un service ;
— une *machine virtuelle* est un programme qui simule, sur une machine hôte, un autre ordinateur.

Exemple.

Un serveur web est un processus (Apache par exemple) qui communique sur le port 80 d'un serveur machine. Si ce serveur machine a pour IP 163.12.9.10, alors tout client web (Firefox par exemple) peut s'adresser au serveur web à l'adresse 163.12.9.10 :80.

La Fig. 2.1 illustre ces concepts de base, que nous utiliserons maintenant intensivement sans plus d'explication. Elle montre dans une machine physique (le « système hôte ») deux machines virtuelles. Chacune de ces machines dispose d'une adresse IP qui lui est propre, et propose des services en écoute sur certains ports. Un

serveur MongoDB est présent par exemple sur chacune des deux machines, en écoute sur le port par défaut 27017, la différentiation des serveurs se faisant donc dans ce cas par l'adresse IP du serveur qui les héberge.

Inversement, on peut avoir deux serveurs identiques sur une même machine, mais sur des ports différents. C'est ce qu'illustre la présence de deux serveurs ElasticSearch sur la seconde machine virtuelle, sur les ports respectifs 9200 et 9201.

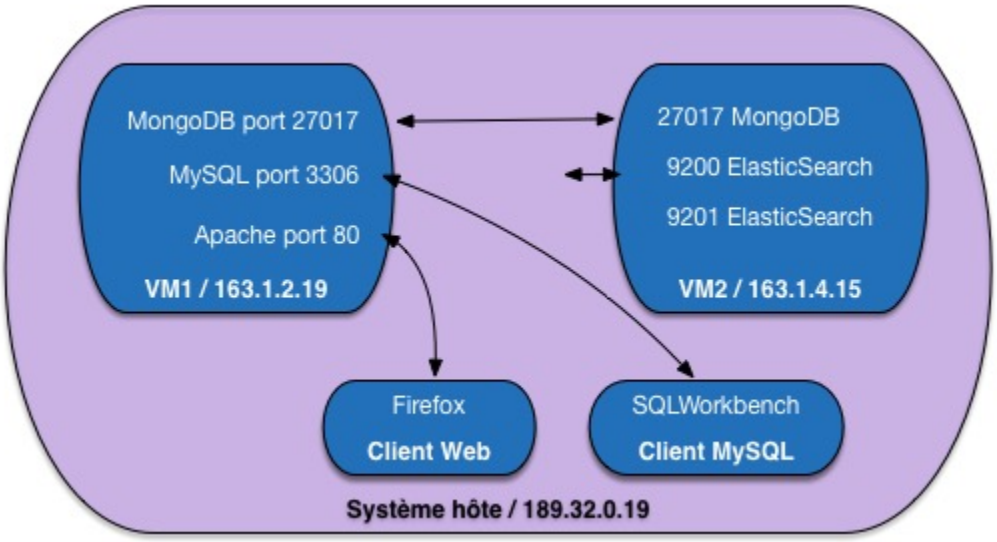


Fig. 2.1 – Un exemple de système distribué, avec serveurs virtuels et clients

Avant de donner quelques explications plus élaborées, il vous suffit de considérer que Docker permet d'installer et d'exécuter très facilement, sur votre ordinateur personnel, et avec une consommation de ressources (mémoire et disque) très faible, ce que nous appellerons pour l'instant des « pseudos-serveurs » en attendant d'être plus précis. Docker offre deux très grands avantages.

- il propose des pseudo-serveurs pré-configurés, prêts à l'emploi (les « images »), qui s'installent en quelques clics ;
- il est tout aussi facile d'installer *plusieurs* pseudos-serveurs communiquant les uns avec les autres et d'obtenir donc un système distribué complet, sur un simple portable (doté d'une puissance raisonnable).

Docker permet de transformer un simple ordinateur personnel en *data center* ! Bien entendu, il n'en a pas la puissance mais pour tester et expérimenter, c'est extrêmement pratique.

Installation : Docker existe sous tous les systèmes, dont Windows. Pour Windows et Mac OS, un installateur *Docker Desktop* est fourni à https://www.docker.com/products/docker-desktop. Il contient tous les composants nécessaires à l'utilisation de Docker.

Note : Merci de me signaler des compléments qui seraient utiles à intégrer dans le présent document, pour les environnements différents de Mac OS X, et notamment Windows.

2.1 Introduction à Docker

Essayons de comprendre ce qu'est Docker avant d'aller plus loin. Vous connaissez peut-être déjà la notion de *machine virtuelle* (VM). Elle consiste à simuler par un composant logiciel, sur une machine physique, un ordinateur auquel on alloue une partie des ressources (mémoire, CPU). Partant d'une machine dotée par exemple de 4 disques et 256 GO de mémoire, on peut créer 4 VMs indépendantes avec chacune 1 disque et 64 GO de RAM. Ces VMs peuvent être totalement différentes les unes des autres. On peut en avoir une sous le système Windows, une autre sous le système Linux, etc.

L'intérêt des VMs est principalement la souplesse et l'optimisation de l'utilisation des ressources matérielles. L'organisation en VMs rend plus facile la réaffectation, le changement du dimensionnement, et améliore le taux d'utilisation des dispositifs physiques (disque, mémoire, réseau, etc.).

Les VMs ont aussi l'inconvénient d'être assez gourmandes en ressource, puisqu'il faut, à chaque fois, faire tourner un système d'exploitation complet, avec tout ce que cela implique, en terme d'emprise mémoire notamment.

Docker propose une solution beaucoup plus légère, basée sur la capacité du système Linux (généralisée à Mac OS et Windows) à créer des espaces isolés auxquels on affecte une partie des ressources de la machine-hôte. Ces espaces, ou *containers* partitionnent en quelque sorte le système-hôte en sous-systèmes étanches, au sein desquels le nommage (des processus, des utilisateurs, des ports réseaux) est purement local. On peut par exemple faire tourner un processus *apache* sur le port 80 dans le conteneur A, un autre processus *apache* sur le port 80 dans le conteneur B, sans conflit ni confusion. Tous les nommages sont en quelque sorte interprétés par rapport à un container donné (notion *d'espace de nom*).

Les conteneurs sont beaucoup plus légers en consommation de ressources que les VMs, puisqu'ils s'exécutent au sein d'un unique système d'exploitation. Docker exploite cette spécificité pour proposer un mode de virtualisation (que nous avons appelé « pseudo-serveur » en préambule) léger et flexible.

2.1.1 Docker et ses conteneurs

Docker (ou, très précisément, le *docker engine*) est un programme qui va nous permettre de créer des conteneurs et d'y installer des environnements prêts à l'emploi, les *images*. Un peu de vocabulaire : dans tout ce qui suit,

- Le *système hôte* est le système d'exploitation principal gérant votre machine ; c'est par exemple Windows, ou Mac OS.
- *Docker engine* ou *moteur docker* est le programme qui gère les conteneurs.
- Un *conteneur* est une partie autonome du système hôte, se comportant comme une machine indépendante.
- Le *client Docker* est l'utilitaire grâce auquel on transmet au moteur les commandes de gestion de ces conteneurs. Il peut s'agir soit de la ligne de commande (*Docker CLI*) ou du *dashboard* intégré au *Docker desktop*.

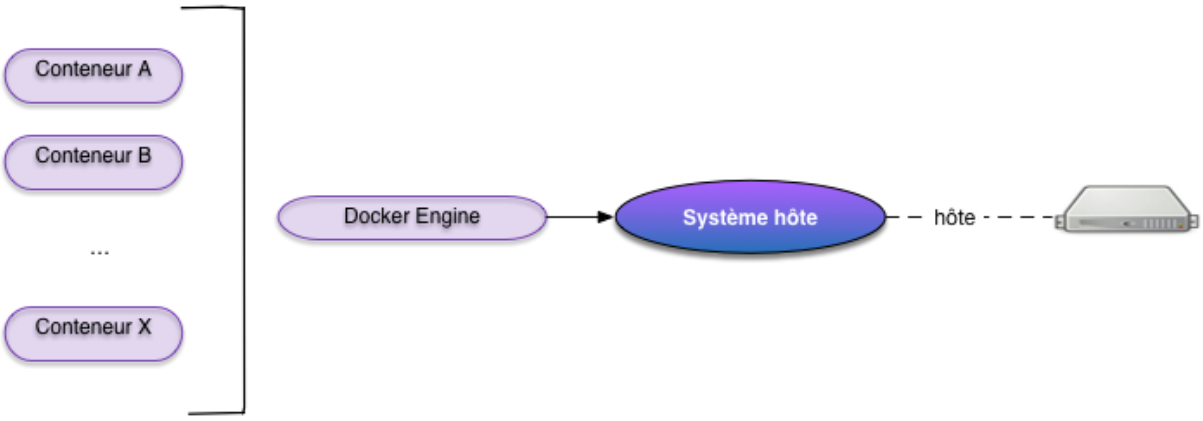


Fig. 2.2 – Le système hôte, le *docker engine* et les conteneurs

2.1.2 Les images Docker

Un conteneur Docker peut donc être vu comme un sous-système autonome, mobilisant très peu de ressources car l'essentiel des tâches système est délégué au système dans lequel il est instancié. On dispose donc virtuellement d'un moyen de multiplier à peu de frais des pseudo-machines dans lesquelles on pourait installer « à la main » des logiciels divers et variés.

Docker va un peu plus loin en proposant des installations pré-configurées, empaquetées de manière à pouvoir être placées très facilement dans un conteneur. On les appelle des *images*. On peut ainsi trouver des images avec pré-configuration de serveurs de données (Oracle, Postgres, MySQL), serveurs Web (Apache, njinx), serveurs NoSQL (mongodb, cassandra), moteurs de recherche (ElasticSearch, Solr). L'installation d'une image se fait très simplement, et soulage considérablement des tâches parfois pénibles d'installation directe.

Une image se place dans un conteneur. On peut placer la même image dans plusieurs conteneurs et obtenir ainsi un système distribué. Examinons la Fig. 2.3 montrant une configuration complète. Nous avons tous les composants à l'œuvre, essayons de bien comprendre.

- Le système hôte exécute le *Docker Engine*, un processus qui gère les images et instancie les conteneurs.
- Docker a téléchargé (nous verrons comment plus tard) les images de plusieurs systèmes de gestion de données : MySQL, MongoDB (un système NoSQL que nous étudierons), et Cassandra.
- Ces images ont été *instanciées* dans des conteneurs A, B et C. L'instanciation consiste à installer l'image dans le conteneur et à l'exécuter. Nous avons donc deux conteneurs avec l'image MySQL, et un troisième avec l'image Cassandra.

Chacun de ces conteneurs dispose de sa propre adresse IP. En supposant que les ports par défaut sont utilisés, le premier serveur MySQL est donc accessible à l'adresse IPb, sur le port 3306, le second à l'adresse IPc, sur le même port.

Important : Le *docker engine* implante un système automatique de « renvoi » qui « publie » le service d'un conteneur sur le port correspondant du système hôte. Le premier conteneur MySQL par exemple est *aussi* accessible sur le port 3306 de la machine hôte. Pour le second, ce n'est pas possible car le port est déjà

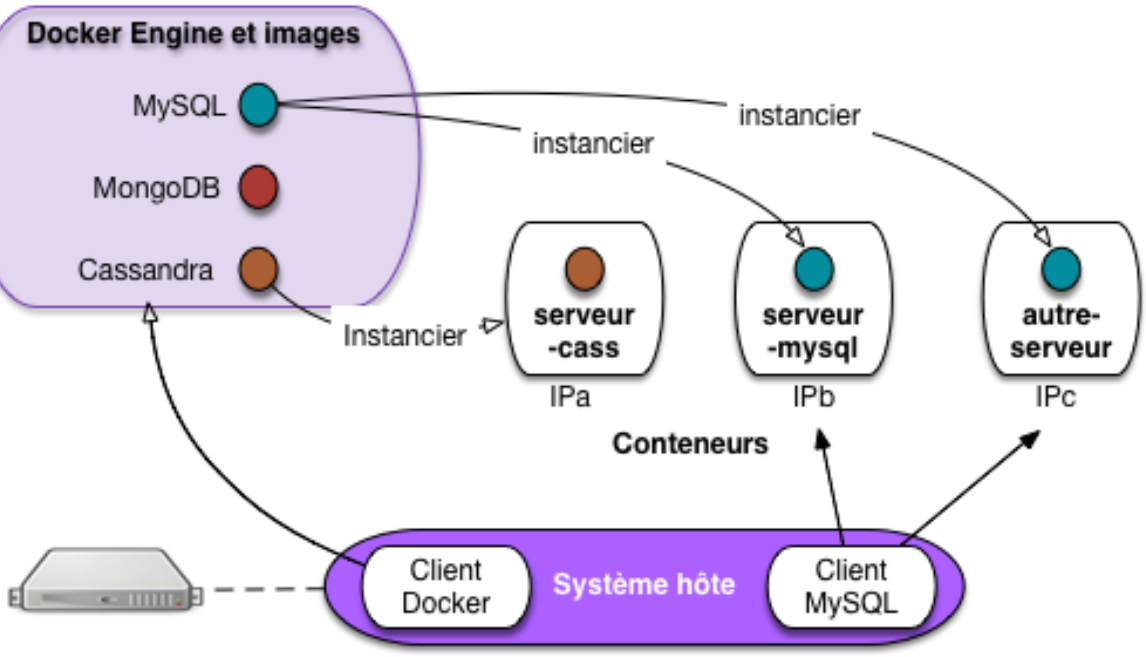


Fig. 2.3 – Les images docker, constituant un pseudo système distribué

occupé, et il faut donc configurer manuellement ce renvoi : nous verrons comment le faire.

L'ensemble constitue donc un système distribué virtuel, le tout s'exécutant sur la machine-hôte et gérable très facilement grâce aux utilitaires Docker. Nous avons par exemple dans chaque conteneur un serveur MySQL. Maintenant, on peut se connecter à ces serveurs à partir de la machine-hôte avec une application cliente (par exemple phpMyAdmin) et tester le système distribué, ce que nous ferons tout au long du cours.

On peut instancier l'image de MongoDB dans 4 conteneurs et obtenir un *cluster* MongoDB en quelques minutes. Evidemment, les performances globales ne dépasseront pas celle de l'unique machine hébergeant Docker. Mais pour du développement ou de l'expérimentation, c'est suffisant, et le gain en temps d'installation est considérable.

En résumé : avec Docker, on dispose d'une boîte à outils pour émuler des environnements complexes avec une très grande facilité.

2.2 Docker en pratique

Dans ce qui suit, je vais illustrer les commandes avec l'utilitaire de commandes en ligne en prenant l'exemple de ma machine Mac OS X. Ce ne doit pas être fondamentalement différent sous les autres environnements.

Note : Vous préférerez sans doute à juste titre utiliser un outil graphique comme le *Docker desktop*, décrit dans la prochaine section, mais avoir un aperçu de commandes transmises par ce dernier est toujours utile

pour comprendre ce qui se passe.

2.2.1 Lancement du Docker Desktop (Mac OS, Windows)

Sous Mac OS ou Windows, vous disposez du *Docker Desktop* qui vous permet de lancer la machine virtuelle et d'obtenir une interface graphique pour gérer votre *docker engine*. La Fig. 2.4 montre la fenêtre des paramètres.

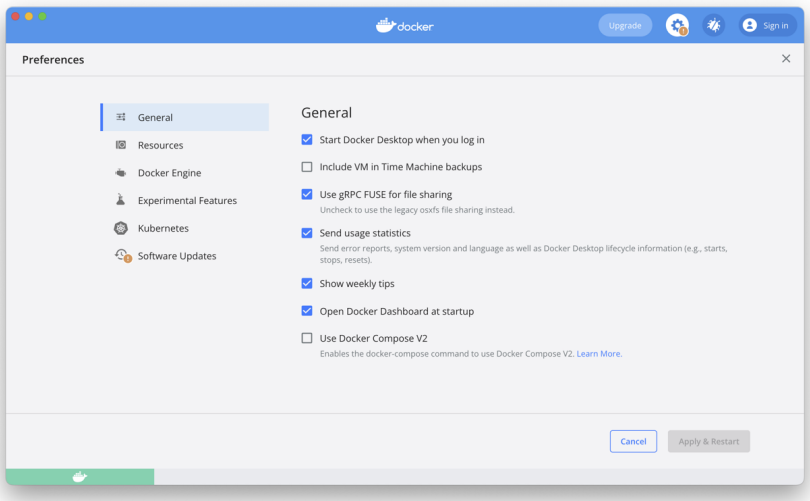


Fig. 2.4 – Le site d'hébergement des images Docker.

Pour communiquer avec le moteur Docker, on peut utiliser un programme client en ligne de commande nommé simplement `docker`. L'image la plus simple est un *Hello world*, on l'instancie avec la commande suivante :

```
docker run hello-world
```

Le `run` est la commande d'instanciation d'une nouvelle image dans un conteneur Docker. Voici ce que vous devriez obtenir à la première exécution.

```
docker run hello-world

   Unable to find image 'hello-world:latest' locally
   latest: Pulling from library/hello-world
   4276590986f6: Pull complete
   Status: Downloaded newer image for hello-world:latest

   Hello from Docker!
```

Décryptons à nouveau. La machine Docker a cherché dans son répertoire local pour savoir si l'image `hello-world` était déjà téléchargée. Ici, comme c'est notre première exécution, ce n'est pas le cas (mes-

sage `Unable to find image locally`). Docker va donc télécharger l'image et l'instancier. Le message `Hello from Docker.` s'affiche, et c'est tout.

L'utilité est plus que limitée, mais cela montre à toute petite échelle le fonctionnement général : on choisit une image, on applique `run` et Docker se charge du reste.

La liste des images est disponible avec la commande :

```
docker images
```

Voici le type d'affichage obtenu :

```
REPOSITORY                TAG                 IMAGE ID            CREATED
↪     SIZE
docker101tutorial         latest              be967614122c        4 days ago
↪     27.3MB
mysql                     latest              e1d7dc9731da        12 days ago
↪     544MB
python                    alpine              0f03316d4a27        13 days ago
↪     42.7MB
nginx                     alpine              6f715d38cfe0        5 weeks ago
↪     22.1MB
docker/getting-started    latest              1f32459ef038        2 months ago
↪     26.8MB
hello-world               latest              bf756fb1ae65        8 months ago
↪     13.3kB
```

Une image est instanciée dans un conteneur avec la commande `run`.

```
docker run --name 'nom-conteneur' <options>
```

Les options dépendent de l'image : voir les sections suivantes pour des exemples. La liste des conteneurs est disponible avec la commande :

```
docker ps -a
```

L'option `-a` permet de voir tous les conteneurs, quel que soit leur statut (en arrêt, ou en cours d'exécution). On obtient l'affichage suivant :

```
CONTAINER ID  IMAGE         COMMAND                  CREATED         STATUS
↪               PORTS  NAMES
d1c2291dc9f9  mysql:latest  "docker-entrypoint.s..."   16 minutes ago  Exited
↪(1) 9 minutes ago           mysql
ec5215871db3  hello-world   "/hello"                 19 minutes ago  Exited (0)
↪19 minutes ago          relaxed_mendeleev
```

Notez le premier champ, `CONTAINER ID` qui nous indique l'identifiant par lequel on peut transmettre des instructions au conteneur. Voici les plus utiles, en supposant que le conteneur est `d1c2291dc9f9`. Tout d'abord on peut l'arrêter avec la commande `stop`.

```
docker stop d1c2291dc9f9
```

Arrêter un conteneur ne signifie pas qu'il n'existe plus, mais qu'il n'est plus actif. On peut le relancer avec la commande `start`.

```
docker start mon-conteneur
```

Pour le supprimer, c'est la commande `docker rm`. Pour inspecter la configuration système/réseau d'un conteneur, Docker fournit la commande `inspect`.

```
docker inspect mon-conteneur
```

On obtient un large document JSON. Parmi toutes les informations données, l'adresse IP du coneneur est particulièrement intéressante. On l'obtient avec

```
docker inspect <container id> | grep "IPAddress"
```

2.2.2 Installons des serveurs Web

Testons Docker avec un des services les plus simples qui soient : un serveur web, Apache. La démarche générale pour une installation consiste à chercher l'image qui vous convient sur le site https://hub.docker.com qui donne accès au catalogue des images Docker fournies par la communauté des utilisateurs.

Faites une recherche avec le mot-clé « httpd » (correspondant aux images du serveur web Apache). Comme on pouvait s'y attendre, de nombreuses images sont disponibles. La plus standard s'appelle tout simplement `httpd` (Fig. 2.5).

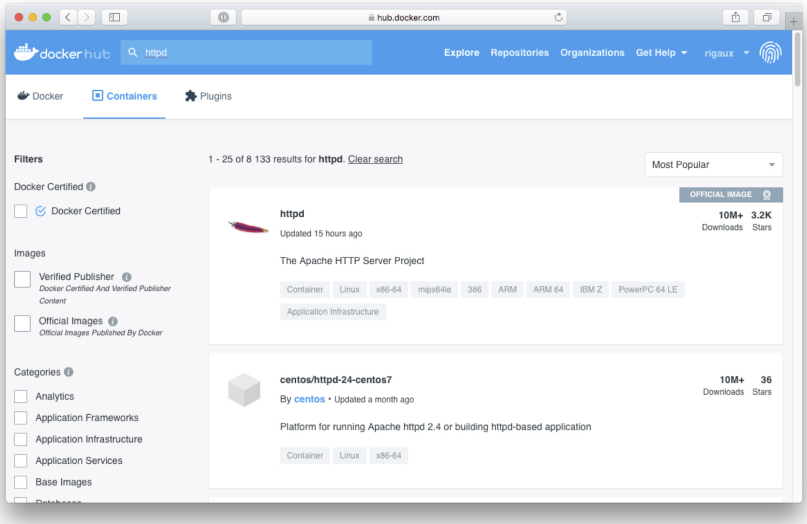


Fig. 2.5 – Les images de serveurs Apache.

Choisissez une image, et cliquez sur le bouton `Details` pour connaître les options d'installation. En prenant l'image standard, on obtient la page de documentation illustrée par la Fig. 2.6

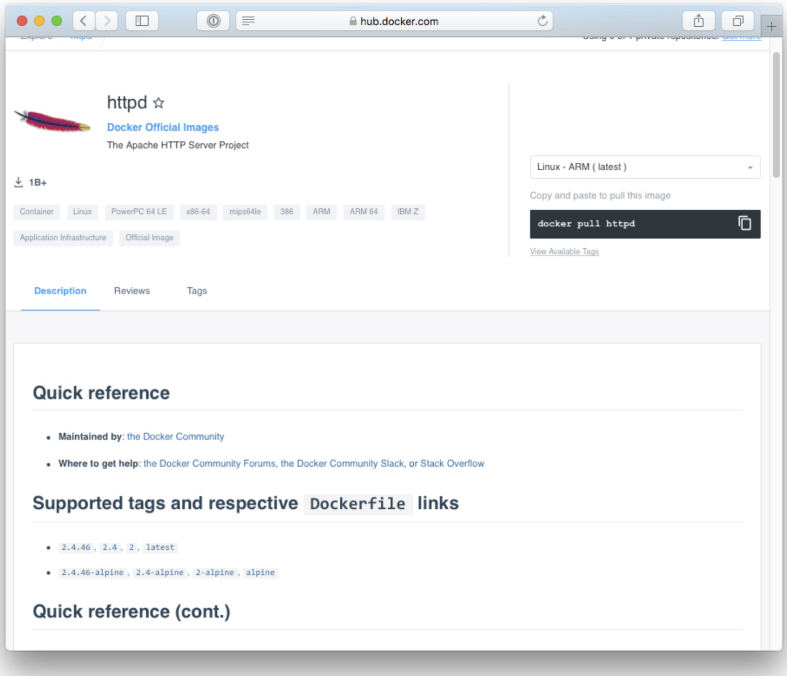


Fig. 2.6 – Documentation d'installation de l'image `httpd`

Vous avez deviné ce qui reste à faire. Installez l'image dans un conteneur sur votre machine avec la commande suivante :

```
docker run --name serveur-web1 --detach httpd:latest
```

Voici les options choisies :

- `name` est le nom du conteneur : il peut remplacer l'id du conteneur quand on veut l'arrêter / le relancer, etc.
- `--detach` (ou `-d`) indique que le conteneur est lancé en tâche de fond, ce qui évite de bloquer le terminal
- on indique enfin l'image à utiliser, ainsi que la version : prenez `latest` (ou ne précisez rien) sauf si vous avez de bonnes raisons de faire autrement.

La première fois, l'image doit être téléchargée, ce qui peut prendre un certain temps. Par la suite, le lancement du conteneur instanciant l'image est quasi instantané.

C'est tout ! Vous avez installé et lancé un serveur web. Vous pouvez le vérifier avec la commande suivante qui donne la liste des conteneurs en cours d'exécution.

```
docker ps
```

Vous devriez obtenir le résultat suivant. Notez le port sur lequel opn peut communiquer avec le service *sur le conteneur*, et le nom du conteneur.

```
CONTAINER ID  IMAGE         COMMAND              CREATED         STATUS         PORTS␣
↪  NAMES
d02a690e0253  httpd:latest "httpd-foreground" 3 minutes ago  Up 3 minutes  80/
↪tcp  serveur-web1
```

Le serveur web est donc accessible sur le port 80 du conteneur. Pour y accéder il faut donc connaître l'adresse IP de ce dernier (c'est possile : voir ci-dessous). Il existe une possibilité plus pratique : *renvoyer* le port du conteneur vers la machine-hôte. Docker fournit un mécanisme dit *de publication* pour indiquer sur quel port se met en écoute un conteneur. On spécifie simplement avec l'option `--publish` (ou `-p`) comment on associe un port du conteneur à un port du système hôte. Exemple :

```
docker run --name serveur-web2 --publish 81:80 --detach httpd:latest
```

Ou plus simplement

```
docker run --name serveur-web2 -p 81:80  -d httpd
```

L'option `-p` indique que le port 80 du conteneur est renvoyé sur le port 81 de la machine hôte.

2.2.3 Le tableau de bord (dashboard)

Plusieurs environnements graphiques existent pour interagir avec Docker. Le tableau de bord (*dashboard*) est l'interface « officielle » fournie avec l'outil *Docker desktop*, mais vous pouvez en tester d'autre si vous le souhaitez. En voici deux qui semblent intéressants.

— Portainer disponible à https://www.portainer.io/
— DockStation disponible https://dockstation.io/

Le *dashboard* facilite la gestion des conteneurs et des images et fournit un tableau de bord sur le système distribué virtuel (Fig. 2.7).

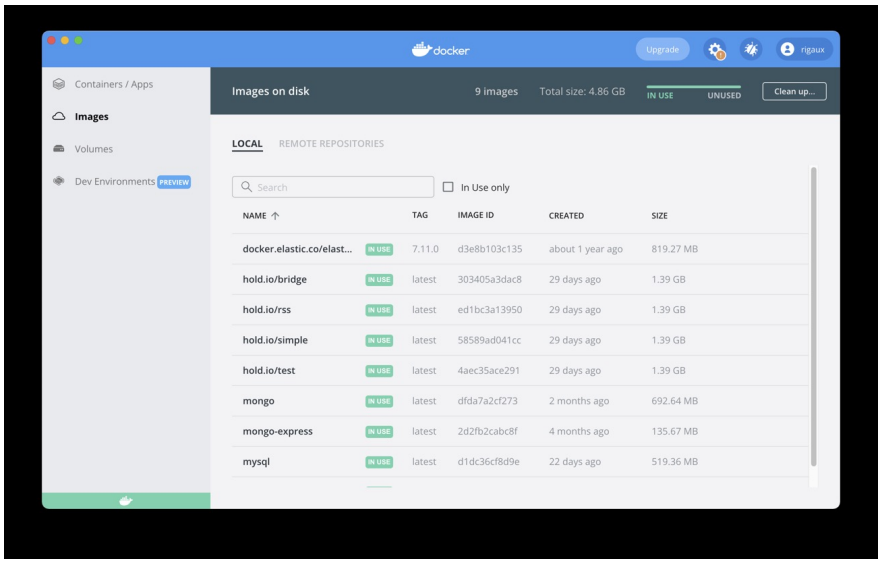


Fig. 2.7 – Le tableau de bord Docker

En cliquant sur le nom de l'un des conteneurs disponibles, on dispose de toutes les options associées. Un paramètre important est le renvoi du port de l'image instanciée dans le conteneur vers un port de la machine Docker. Ce renvoi permet d'accéder avec une application de la machine-hôte à l'instance de l'image comme si elle s'exécutait directement dans la machine Docker. Reportez-vous à la section précédente pour des explications complémentaires sur l'option `--publish`.

2.2.4 Un serveur c'est bien, mais où est le client ?

Une fois que l'on a installé des services dans des conteneurs, il faut disposer de programmes clients pour pouvoir dialoguer avec eux. Ces programmes clients sont en général à installer directement sur la machine hôte.

Dans le cas d'un serveur web, ou en général de tout service qui communique selon le protocole HTTP, un navigateur web fait parfaitement l'affaire. Avec votre navigateur préféré, essayer d'accéder aux adresses http://localhost:80 et http://localhost:81 : les services web Docker que vous venez d'installer devraient répondre par un message basique mais réconfortant.

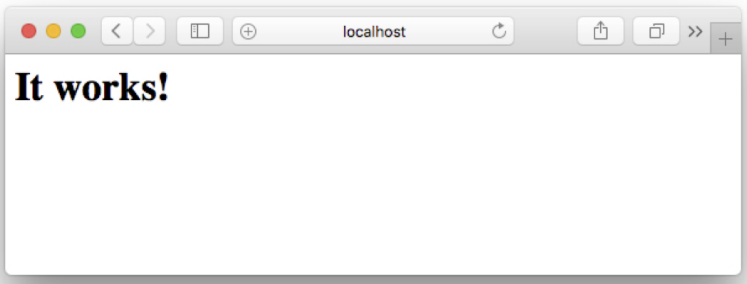


Fig. 2.8 – Accès au service avec le nom du serveur et le port

Ouf ! Prenez le temps de bien comprendre, car une fois ces mécanismes assimilés, nous serons libérés de tout souci pour créer nos systèmes distribués et les expérimenter par la suite. Et je vous rassure : l'ensemble est géré de manière plus conviviale avec le *dashboard* (ce qui ne dispense pas de comprendre ce qui se passe).

2.3 Nos systèmes NoSQL

Dans la suite de ce cours, nous seront amenés à expérimenter quelques systèmes NoSQL. J'en ai sélectionné un petit nombre, sur des critères de popularité, de représentativité des techniques étudiés, de variété. Tout ont en commun de s'installer facilement avec Docker et de disposer d'une interface cliente gratuite et raisonnablement simple.

Dans ce qui suit nous installons successivement CouchDB, MongoDB, Cassandra et ElasticSearch avec Docker. Pour chaque système nous installons également une application cliente, et nous chargeons un (petit) jeu de données que je fournis sur le site https://deptfod.cnam.fr/bd/tp/datasets/. Vous êtes invités à effectuer ces installations sur votre machine, de manière à être prêts aux expérimentations qui suivront.

2.3.1 CouchDB

CouchDB est un système NoSQL qui gère des collections de documents JSON. Vous pouvez installer CouchDB sur votre machine avec Docker, en exposant le port 5984 sur la machine hôte. Voici la commande d'installation.

```
docker run  -d --name my-couchdb -e COUCHDB_USER=admin \
    -e COUCHDB_PASSWORD=admin -p 5984:5984 couchdb:latest
```

Dans ce qui suit, on suppose que le serveur est accessible à l'adresse http://localhost:5984.

CouchDB est essentiellement un serveur Web étendu à la gestion de documents JSON. Comme tout serveur Web, il parle le HTTP, et on peut donc y accéder avec un navigateur qui fait office de client ! Pour des interactions HTTP plus complexes, nous utilliserons également l'utilitaire cURL en ligne de commande. S'il n'est pas déjà installé dans votre environnement (toutes plateformes), il est fortement conseillé de le faire dès maintenant : le site de référence est http://curl.haxx.se/.

Une première requête HTTP permet de vérifier la disponibilité de ce serveur. Entrez l'adresse suivante dans le navigateur : http://admin:admin@localhost:5984. Vous devriez obtenir un document similaire au suivant :

```
{
"couchdb": "Welcome",
"version": "3.3.3",
"git_sha": "40afbcfc7",
"uuid": "0f4f4743bf65f3c4cd61caf3e789c559",
"vendor": {"name": "The Apache Software Foundation"}
}
```

Vous pouvez obtenir le même résultat avec `cURL`.

```
curl -X GET http://admin:admin@localhost:5984
```

Note : Vous noterez qu'il faut indiquer dans l'URL le compte d'accès (admin/admin) juste avant le nom du serveur.

Un serveur CouchDB gère un ensemble de bases de données. Créer une nouvelle base se traduit, par la création d'une nouvelle ressource avec une requête HTTP `PUT`). Voici donc la commande avec cURL pour créer une base `films`.

```
curl -X PUT http://admin:admin@localhost:5984/films
{"ok":true}
```

Maintenant que la base est créée, et on peut obtenir sa représentation avec une requête `GET` (navigateur ou `cURL`).

```
curl -X GET http://admin:admin@localhost:5984/films
```

Cette requête renvoie un document JSON décrivant la nouvelle base.

```
{"update_seq":"0-g1AAAADfeJz6t",
 "db_name":"films",
 "sizes": {"file":17028,"external":0,"active":0},
 "purge_seq":0,
 "other":{"data_size":0},
 "doc_del_count":0,
 "doc_count":0,
 "disk_size":17028,
 "disk_format_version":6,
 "compact_running":false,
 "instance_start_time":"0"
}
```

Pour finir, nous allons insérer dans notre base CouchDB un ensemble de documents JSON représentant des films. Récupérez sur le site http://deptfod.cnam.fr/bd/tp/datasets/ le fichier `films_couchdb.json` au format spécifique d'insertion CouchDB. Il a la forme suivante (nous reparlerons de JSON bientôt) :

```
 {"docs":
  [
   {
    "_id": "movie:1",
    "title": "Vertigo",
    ...
   },
   {
    "_id": "movie:2",
    "title": "Alien",
    ...
   }
  ]
}
```

La commande d'insertion est alors la suivante :

```
curl -X POST  http://admin:admin@localhost:5984/films/_bulk_docs \
     -d @films_couchdb.json -H "Content-Type: application/json"
```

CouchDB fournit également une interface graphique intégrée disponible à l'URL relative `_utils` (donc à l'adresse complète http://localhost:5984/_utils dans notre cas). La Fig. 2.9 montre l'aspect de cette interface graphique, très pratique, avec la base que nous venons de créer.

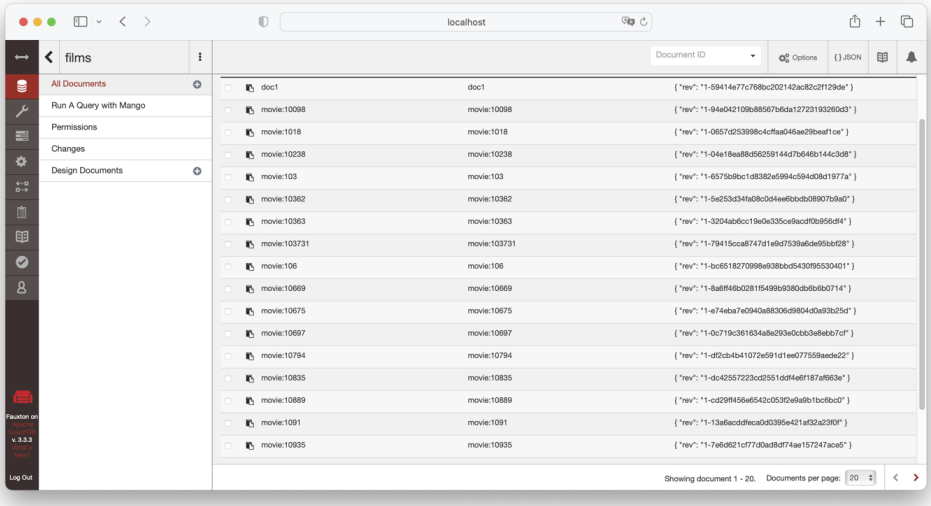


Fig. 2.9 – L'interface graphique (Fauxton) de CouchDB

2.3.2 Cassandra

Cassandra est un système de gestion de données à grande échelle conçu à l'origine (2007) par les ingénieurs de Facebook pour répondre à des problématiques liées au stockage et à l'utilisation de gros volumes de données. La communauté s'est tellement investie dans le projet Cassandra que, au final, ce dernier a complètement divergé de sa version originale. Facebook s'est alors résolu à accepter que le projet - en l'état - ne correspondait plus précisément à leurs besoins, et que reprendre le développement à leur compte ne rimerait à rien tant l'architecture avait évolué. Cassandra est donc resté porté par l'Apache Incubator. Aujourd'hui, c'est la société Datastax qui assure la distribution et le support de Cassandra qui reste un projet Open Source de la fondation Apache.

L'installation d'un conteneur Cassandra avec Docker se fait avec la commande suivante :

```
docker run --name mon-cassandra -p 3000:9042  -d cassandra:latest
```

On communique avec Cassandra via un langage, CQL, dont les commandes doivent être transmises au port 9042 du conteneur. Dans l'instruction ci-dessus, ce port est renvoyé sur le port 3000 du système hôte avec l'option `-p`.

Il vous faut un client sur la machine hôte. Des clients graphiques existent. Datastax propose des outils, dont le Datastax Studio qui est assez agréable à utiliser mais dont l'installation est lourde. Un « petit » utilitare graphique assez complet *DbVisualizer*, disponible en version gratuite à l'adresse https://www.dbvis.com/. Il permet classiquement de créer des connexions, d'inspecter un schéma et de transmettre des requêtes.

DbVisualizer peut être utilisé avec beaucoup de bases de données. Selon le système choisi, il faut télécharger des connecteurs spécifiques (*drivers*). Cela se fait de manière assez intuitive via DbVis lui-même. Dans le cas de Cassandra, il faut prendre le *driver* `Cassandra DataStax`.

La Fig. 2.10 montre l'interface, après création d'un *keyspace*, de tables et de données. Le *keyspace* est le nom que Cassandra donne à une base de données. Sous DbVisualizer, les *keyspaces* apparaissent à gauche de la fenêtre principale (voir figure Fig. 2.10). Un clic bouton droit permet d'ouvrir un formulaire de création

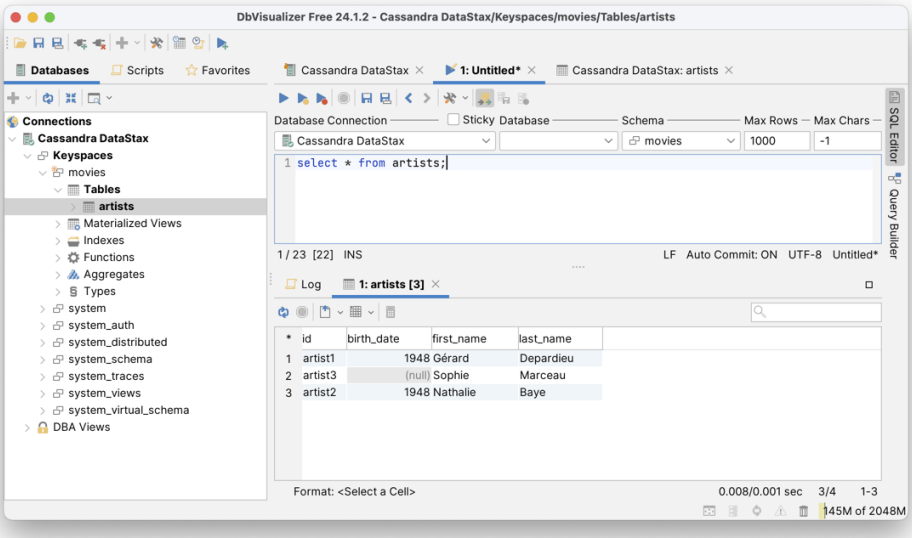


Fig. 2.10 – Le client *DbVisualizer* accédant à Cassandra

d'un *keyspace*.

Note : Cassandra est fait pour fonctionner dans un environnement distribué. Pour créer un *keyspace*, il faut donc préciser la stratégie de réplication à adopter. Nous verrons plus en détail après comment tout ceci fonctionne.

Cassandra est assez proche en apparence d'un système relationnel, avec création de tables, et insertion dans ces tables. Cassandra va au-delà de la norme relationnelle en permettant des données *dénormalisées* dans lesquelles certaines valeurs sont complexes (dictionnaires, ensembles, etc.). Il nous faut au préalable définir le *type* `artist` de la manière suivante :

```
create type artist (id text,
                    last_name text,
                    first_name text,
                    birth_date int,
                    role text);
```

On crée alors une table utilisant ce type : le metteur en scène est une instance du type `artist`, les acteurs un ensemble d'instance. Cela donne le schéma suivant

```
create table movies (id text,
                title text,
                year int,
                genre text,
                country text,
                director frozen<artist>,
```

(suite sur la page suivante)

(suite de la page précédente)

```
            actors set< frozen<artist>>,
          primary key (id) );
```

Cela correspond en JSON à la structure suivante :

```
insert into movies JSON '{
        "id": "movie:11",
        "title": "Star Wars",
        "year": 1977,
        "genre": "Adventure",
        "country": "US",
        "director": {
                "id": "artist:1",
                "last_name": "Lucas",
                "first_name": "George",
                "birth_date": 1944
        },
        "actors": [
                {
                        "last_name": "Hamill",
                        "first_name": "Mark",
                        "birth_date": 1951
                },
                {
                        "last_name": "Ford",
                        "first_name": "Harrison",
                        "birth_date": 1942
                },
                {
                        "last_name": "Fisher",
                        "first_name": "Carrie",
                        "birth_date": 1956
                }
        ]
}')
```

Je vous laisse effectuer l'insertion de l'ensemble des films tels qu'ils sont fournis par le site http://deptfod.cnam.fr/bd/tp/datasets/cassandra, avec tous les acteurs d'un film. Il suffit de récupérer le fichier contenant l'ensemble des commandes d'insertion et de l'exécuter comme un script dans DbVisualizer. Nous nous en servirons pour l'interrogation CQL ensuite.

2.3.3 ElasticSearch

ElasticSearch est un moteur de recherche disponible sous licence libre (Apache). Il repose sur Lucene (nous verrons plus bas ce que cela signifie). Il a été développé à partir de 2004 et est aujourd'hui adossé à une entreprise, Elastic.co.

Commençons par l'installation avec Docker. Voici une commande qui devrait fonctionner sous tous les systèmes et mettre ElasticSearch en attente sur le port 9200.

```
docker run -d --name es1 -p 9200:9200  elasticsearch:8.13.0
```

Les versions d'ElasticSearch

ElasticSearch évolue rapidement. Pour éviter que les instructions qui suivent ne deviennent rapidement obsolètes, j'indique le numéro de version dans l'installation avec Docker (la 8.13.0, de mars 2024).

Quelques commandes supplémentaires sont nécesssaires. Un compte `elastic` est créé, on lui attribue un mot de passe avec la commande suivante :

```
docker exec -it es1 /usr/share/elasticsearch/bin/elasticsearch-reset-password -u␣
↪elastic
```

Notez le mot de passe et placez-le éventuellement dans une variable d'environnement

```
export ELASTIC_PASSWORD=<le_mot_de_passe>
```

Si vous voulez interagir avec `curl`, il faut récupérer un certificat d'authentification SSL.

```
docker cp es1:/usr/share/elasticsearch/config/certs/http_ca.crt .
```

Toutes les interactions avec un serveur ElasticSearch passent par une interface HTTP basée sur JSON. Vous pouvez directement vous adresser au serveur HTTP en écoute sur https://localhost:9200. Avec `curl` vous pouvez donc faire :

```
curl --cacert http_ca.crt  -U elastic:mot_de_passe https://localhost:9200
```

Vous devriez obtenir un document JSON semblable à celui-ci :

```
 {
"name": "7a46670f6a9e",
"cluster_name": "docker-cluster",
"cluster_uuid": "89U3LpNhTh6Vr9UUOTZAjw",
"version": {
    "number": "8.13.0",
    "...": "...",
     "lucene_version": "9.1.1",
  },
```

(suite sur la page suivante)

(suite de la page précédente)

```
"tagline": "You Know, for Search"
}
```

Pour une inspection confortable du serveur et des index ElasticSearch, nous vous conseillons d'utiliser une interface d'administration : `ElasticVue`, disponible sous toutes les plateformes. Elle peut être téléchargée ici : https://elasticvue.com/.

En lançant l'application, on peut se connecter au serveur https://localhost:9200 et on obtient l'interface de la figure Fig. 2.11. La barre supérieure de l'interface propose des options `REST` et `SEARCH` qui vont nous intéresser dans un premier temps.

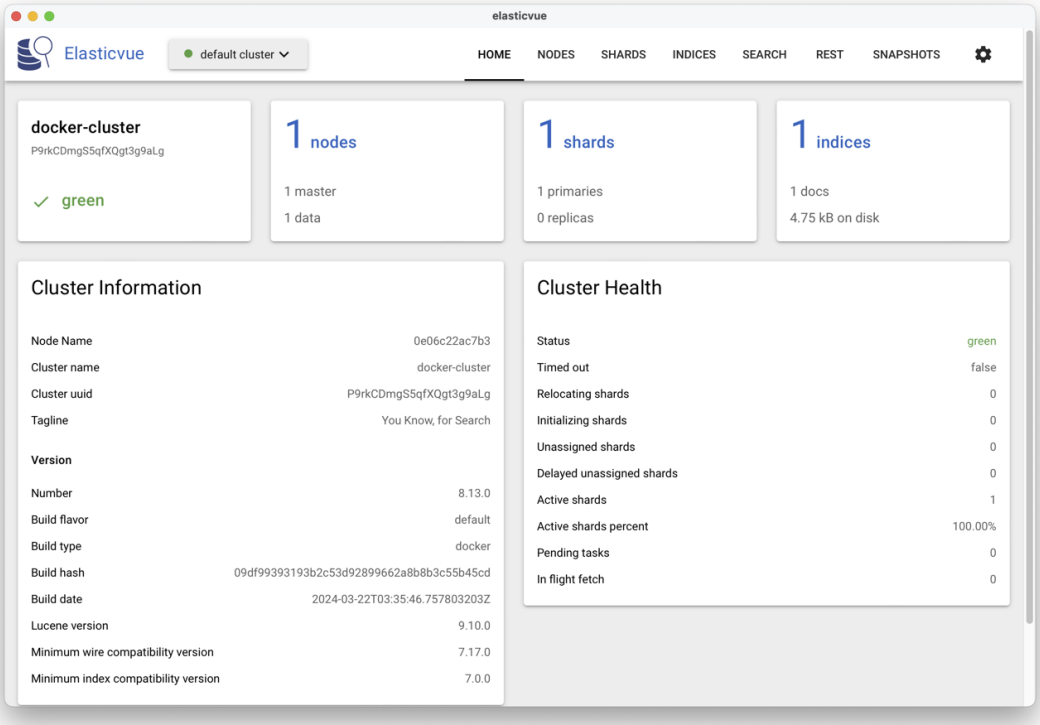


Fig. 2.11 – Le tableau de bord proposé par ElasticVue.

On ne parle pas de base de données dans ElasticSearch mais *d'index*. Pour créer un index `nfe204-1` on envoie un `PUT` à l'adresse https://localhost:9200/nfe204-1. C'est facile avec la fenêtre REST d'ElasticVue : voir Fig. 2.12.

Le `PUT` crée un index à l'URL indiquée. Pour créer notre base de données, nous allons utiliser une autre interface d'Elasticsearch, appelée `bulk` (*en grosses quantités*), qui permet comme son nom l'indique d'indexer de nombreux documents en une seule commande.

Récupérez notre collection de films, au format JSON adapté à l'insertion en masse dans ElasticSearch, sur le site http://deptfod.cnam.fr/bd/tp/datasets/. Le fichier se nomme `films_esearch.json`.

Vous pouvez l'ouvrir pour voir le format. Chaque document « film » (sur une seule ligne) est précédé d'un petit document JSON qui précise l'index (`movies`), et l'identifiant (`1`).

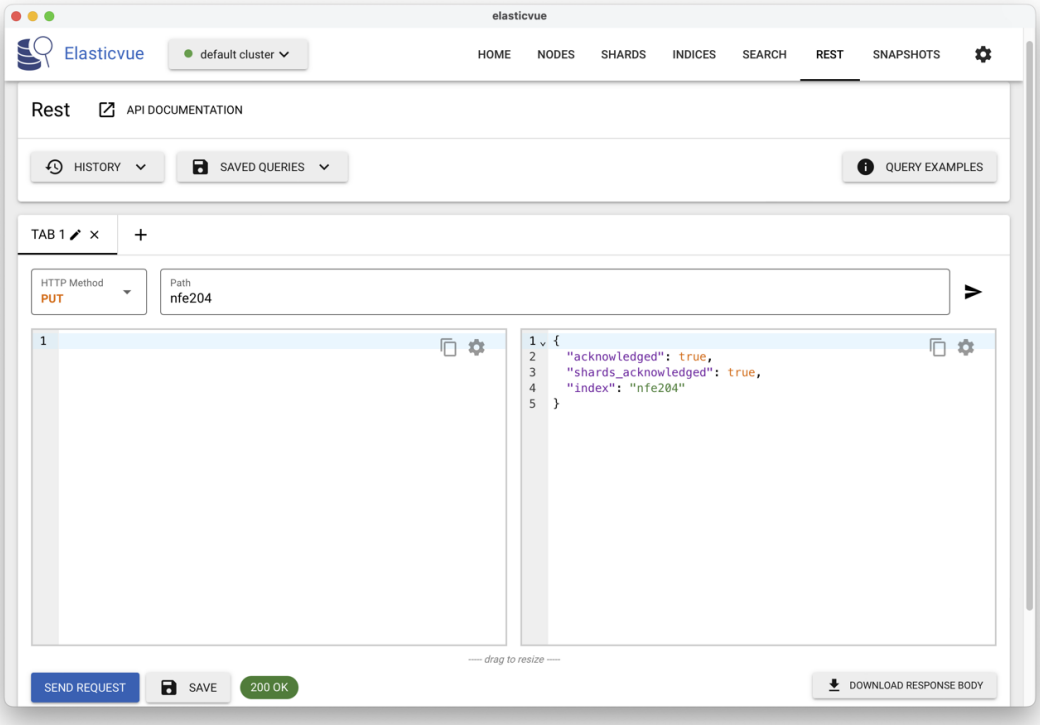


Fig. 2.12 – Utilisation d'ElasticVue pour transmettre des commandes REST

```
{"index":{"_index": "nfe204", "_id": "movie"}}
```

On trouve ensuite les documents JSON proprement dits. **Attention il ne doit pas y avoir de retour à la ligne dans le codage JSON**, ce qui est le cas dans le document que nous fournissons.

```
{"title": "Mars Attacks!", "summary": "...", "...": "..."}
```

Ensuite, importez les documents dans Elasticsearch en le transmettant par un `POST` à l'URL https://localhost:9200/_bulk/

Avec les paramètres spécifiés dans le fichier `films_esearch.json`, vous devriez retrouver un index `nfe204` maintenant présent dans l'interface, contenant les données sur les films.

2.3.4 MongoDB

Installons maintenant MongoDB, un des systèmes NoSQL les plus populaires. L'installation Docker se fait avec la commande suivante et instancie un conteneur accessible à localhost :30001.

```
docker run --name mon-mongo -p 30001:27017 -d mongo
```

MongoDB fonctionne en mode classique client/serveur. Le serveur `mongod` est en attente sur le port 27017 dans son conteneur, et peut être redirigé vers un port de la machine Docker, comme le port 30001 dans la

commande précédente.

En ce qui concerne les *applications* clientes, nous avons en gros deux possibilités : l'interpréteur de commande `mongo` (qui suppose d'avoir installé MongoDB sur la machine hôte) ou une application graphique plus agréable à utiliser. Parmi ces dernières, `Studio3T` (http://studio3.com) me semble le meilleur client graphique du moment ; il existe une version gratuite, pour des utilisations non commerciales, qui ne vous expose qu'à quelques courriels de relance de la part des auteurs du système (vous pouvez en profiter pour les remercier gentiment).

Studio3T propose un interpréteur de commande intelligent (autocomplétion, exécution de scripts placés dans des fichiers), des fonctionnalités d'import et d'export. La Fig. 2.13 montre l'interface en action.

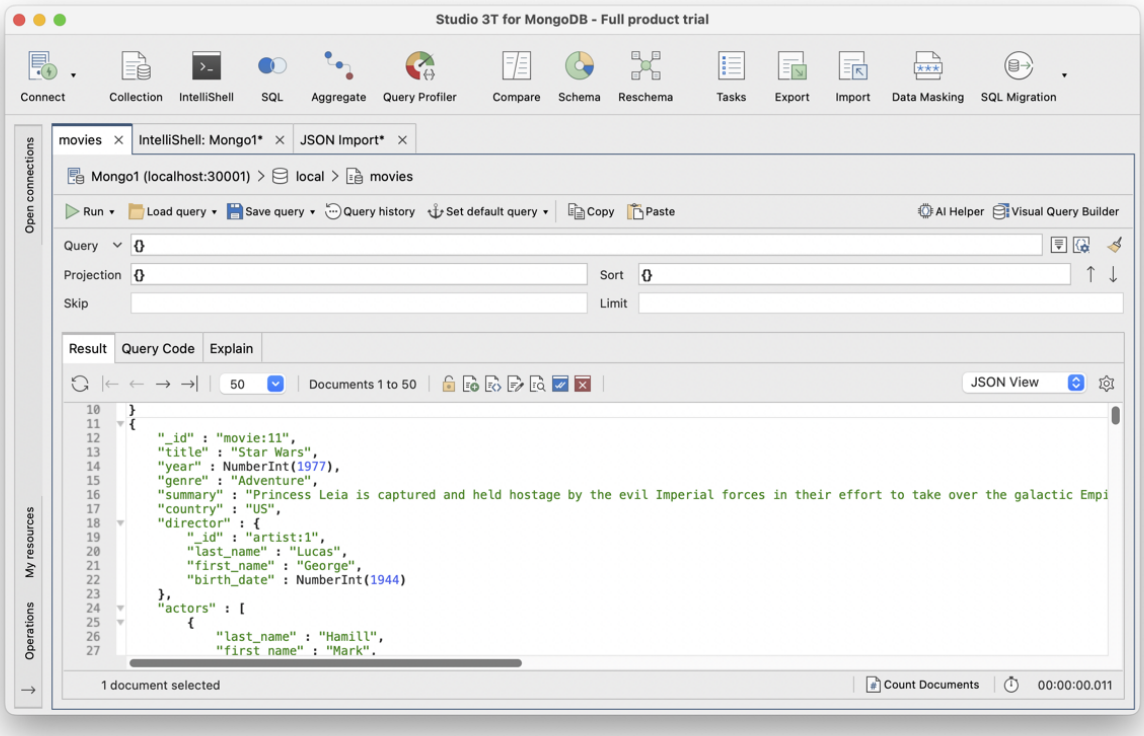


Fig. 2.13 – L'interface de Studio3T

Créons maintenant notre base des films, constituée de documents JSON ayant la forme suivante :

```
{
  "_id": "movie:100",
  "title": "The Social network",
  "summary": "On a fall night in 2003, Harvard undergrad and
     programming genius Mark Zuckerberg sits down at his
     computer and heatedly begins working on a new idea. (...)",
  "year": 2010,
```

(suite sur la page suivante)

(suite de la page précédente)

```
  "director": {"last_name": "Fincher",
               "first_name": "David"},
  "actors": [
    {"first_name": "Jesse", "last_name": "Eisenberg"},
    {"first_name": "Rooney", "last_name": "Mara"}
   ]
}
```

Comme il serait fastidieux de les insérer un par un, nous allons utiliser Studio3T. Un fichier conforme au format attendu est disponible parmi les jeux de données de http://deptfod.cnam.fr/bd/tp/datasets/. Vous pouvez le télécharger et l'utiliser pour insérer directement les films dans la base avec l'utilitaire d'import de Studio3T.

2.3.5 Quiz

Quelles phrases suivantes sont exactes à propos de la notion de serveur :

A) Un serveur est lancé à chaque fois qu'on fait appel à lui

B) Un serveur est nécessairement connecté à un port réseau

C) Il ne peut y avoir qu'un seul serveur d'un type donné (par exemple un serveur Web) sur une machine

Quelles phrases suivantes sont exactes à propos de Docker :

A) La machine Docker est un serveur

B) La machine Docker est un client

C) La machine Docker s'exécute dans un conteneur

D) La machine Docker s'exécute dans un système hôte

Après avoir installé un serveur dans un conteneur Docker, que reste-t-il à faire :

A) Il faut configurer le serveur

B) Il faut lancer le serveur

C) Il faut installer au moins un programme client dans le conteneur Docker

D) Il faut installer au moins un programme client sur la machine hôte

2.4 Exercices

Dans ces exercices vous devez mettre en action les principes de Docker vus ci-dessus, et vous êtes également invités à découvrir l'outil `docker compose` qui nous permet de configurer une fois pour toutes un environnement distribué constitué de plusieurs serveurs.

Exercice Ex-S1-1 : testons notre compréhension

Savez-vous répondre aux questions suivantes ?

— Qu'est-ce qu'une machine Docker, où se trouve-t-elle, quel est son rôle ?

— À quoi sert le terminal Docker, et qu'est-ce qui caractérise un tel terminal ?
— Qu'est-ce que la machine hôte ? Doit-elle forcément tourner sous Linux ?
— Une instance d'image est-elle placée dans un conteneur, dans la machine hôte ou dans la machine Docker ?
— Peut-on instancier une image dans plusieurs conteneurs ?

Si vous ne savez pas répondre, cela vaut la peine de relire ce qui précède, ou des ressources complémentaires sur le Web. Vous serez plus à l'aise par la suite si vous avez une idée claire de l'architecture et de ses concepts-clé.

Exercice Ex-S1-2 : premiers pas Docker

Maintenant, effectuez les opérations ci-dessus sur *votre* machine. Installez Docker, lancez le conteneur `hello-world`, affichez la liste des conteneurs, supprimez le conteneur `hello-world`.

Exercice Ex-S2-1 : installez MySQL

— Après installation de Docker, créez un conteneur avec la dernière version de MySQL. Vous pouvez utiliser la ligne de commande ou le *dashboard*.
Installez également un client MySQL sur votre machine (par exemple le *MySQL Workbench* accessible à https://www.mysql.com/products/workbench/) et connectez-vous à votre conteneur Docker.
— Au lieu de lancer toujours la même ligne de commande, on peut créer un fichier de configuration (dans un format qui s'appelle YAML) et l'exécuter avec l'utilitaire `docker-compose`. Voici un exemple de base : sauvegardez-le dans un fichier `mysql-compose.yml`.

```yaml
services:
  mysql1:
    image: mysql:latest
    ports:
      - "6603:3306"
    environment:
      - "MYSQL_ALLOW_EMPTY_PASSWORD=1"
```

Vous pouvez alors lancer la création de votre conteneur avec la commande :

```
docker compose -f mysql-compose.yml up
```

Voici quelques exercices à faire :
— Testez que vous pouvez bien accéder à votre conteneur avec le client MySQL (quel est le port d'accès défini dans la configuration Yaml ?)
— Configurez votre conteneur MySQL pour définir un compte d'accès `root` avec un mot de passe et donnez à la base le nom `nfe204` (aide : lisez la documentation associée au conteneur MySQL, laquelle se trouve ici : https://hub.docker.com/_/mysql)
— Configurez un ensemble de trois conteneurs MySQL. Vérifiez que vous pouvez vous connecter à chacun.
— Arrêtez/redémarrez le conteneur (cherchez la commande `docker-compose`), enfin supprimez-le.

Exercice Ex-S2-1 : installez MongoDB

Même exercice, mais cette fois avec MongoDB, un système NoSQL très utilisé. Les principes sont les mêmes : vous récupérez une image de MongoDB après voir fouillé sur http://hub.docker.com, vous l'instanciez, vous configurez le port d'accès, et vous testez l'accès avec un client à partir de la machine-hôte.

Pour MongoDB, voici les deux principaux clients disponibles gratuitement :

— Studio 3T (*free edition*), un des plus anciens, disponible à https://studio3t.com/

— Compass, disponible à https://www.mongodb.com/products/tools/compass, le client graphique « officiel » de MongoDB

Créez un fichier de configuration pour `docker-compose` également, afin d'instancier trois conteneurs.

CHAPITRE 3

Modélisation de bases NoSQL

Ce chapitre est consacré à la notion de *document* qui est à la base de la représentation des données dans l'ensemble du cours. Cette notion est volontairement choisie assez générale pour couvrir la large palette des situations rencontrées : une valeur atomique (un entier, une chaîne de caractères) est un document ; une paire clé-valeur est un document ; un tableau de valeurs est un document ; un agrégat de paires clé-valeur est un document ; et de manière générale, toute composition des possibilités précédentes (un tableau d'agrégats de paires clé-valeur par exemple) est un document.

Nos documents sont caractérisés par l'existence d'une *structure*, et on parlera donc de *documents structurés*. Cette structure peut aller du très simple au très compliqué, ce qui permet de représenter de manière autonome des informations arbitrairement complexes.

Deux formats sont maintenant bien établis pour représenter les documents structurés : XML et JSON. Le premier est très complet mais très lourd, le second a juste les qualités inverses. Ces formats sont, entre autres, conçus pour que le codage des documents soit adapté aux échanges dans un environnement distribué. Un document en JSON ou XML peut être transféré par réseau entre deux machines sans perte d'information et sans problème de codage/décodage.

Il s'ensuit que les documents structurés sont à la base des systèmes distribués visant à des traitements à très grande échelle, autrement dit le « NoSQL » pour faire bref. Plusieurs de ces systèmes utilisent directement XML et surtout JSON, mais le modèle utilisé par d'autres est le plus souvent, à la syntaxe près, tout à fait équivalent. **Il est important d'être capable de comprendre le modèle des documents structurés indépendamment d'un codage particulier**. Ce chapitre se concentre sur le codage JSON. XML, beaucoup plus riche, est un peu trop complexe pour les systèmes NoSQL.

Il est donc tout à fait intéressant d'étudier la construction de documents structurés comme base de la représentation des données. Une question très importante dans cette perspective est celle de la modélisation préalable de collections de documents. Cette modélisation est une étape essentielle dans la construction de bases relationnelles, et assez négligée pour les bases NoSQL où on semble parfois considérer qu'il suffit d'accumuler des données sans se soucier de leur forme. Ce chapitre aborde donc la question, ne serait-ce que

pour vous sensibiliser : *construire une collection de documents comme une décharge de données est une très mauvaise idée et se paye très cher à terme*.

3.1 S1 : documents structurés

Supports complémentaires

— Diapositives: documents structurés et JSON
— Vidéo sur les documents structurés
— Vidéo sur le codage JSON

3.1.1 Modèle des documents structurés

Le modèle des documents structurés repose sur quelques notions de base que nous définissons précisément pour commencer.

Définition (Valeur atomique)

Une *valeur atomique* est une instance de l'un des types de base usuels : entiers, flottants, chaînes de caractères.

Les types peuvent varier selon les systèmes mais la caractéristique première d'une valeur atomique est d'être *non décomposable* en sous-unités ayant un sens pour les applications qui les manipulent. De ce point de vue, une date n'est pas atomique puisqu'on pourrait la décomposer en jour/mois/an, sous-unités qui ont chacune un sens bien défini.

La signification d'une valeur est donnée par son association à un *identifiant*. Dans le modèle, les identifiants sont simplement des chaînes de caractère. On obtient des *paires clé - valeur*.

Définition (Paire clé - valeur)

Une paire clé - valeur est une paire (i, v) où i est une clé et v une valeur.

Pour l'instant nous ne connaissons que les valeurs atomiques mais la définition des paires clé-valeur s'étend aux valeurs structurées que nous pouvons maintenant définir.

Définition (Valeur structurée)

La définition est récursive

— Si v est une valeur atomique, v est une valeur structurée.
— Si $v_1, \cdots, v_n$ sont des valeurs structurées, alors la *liste* $[v_1, \cdots, v_n]$ est une valeur structurée.
— Si $p_1, \cdots, p_n$ sont des paires clé-valeur *dont les clés sont distinctes deux à deux*, alors le *dictionnaire* (ou *objet*) $p_1, \cdots, p_n$ est une valeur structurée.

Les listes (ou tableaux) et les dictionnaires (ou objets) sont les structures qui, appliquées récursivement, permettent de construire des valeurs structurées.

La définition des documents s'ensuit.

Définition (Document)

Tout dictionnaire est un document.

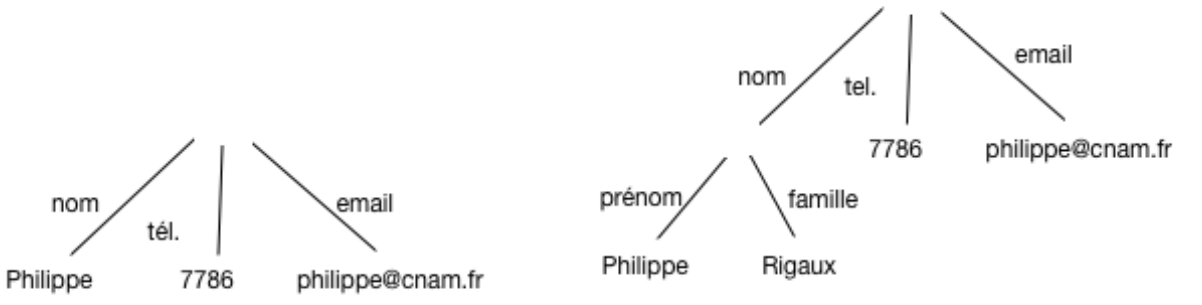
Une *collection* est un ensemble de documents. On ajoutera souvent, pour les documents appartenant à une collection, une contrainte *d'identification* : chaque document doit contenir une paire clé-valeur dont la clé est conventionnellement `id`, et dont la valeur est unique au sein de la collection. Cette valeur sert d'identifiant de recherche pour trouver rapidement un document dans une collection.

Ce modèle permet de représenter des informations plus ou moins complexes en satisfaisant les besoins suivants :

— *Flexibilité* : la structure s'adapte à des variations plus ou moins importantes ; prenons un document représentant un livre ou une documentation technique : on peut avoir (ou non) des annexes, des notes de bas de pages, tout un ensemble d'éléments éditoriaux qu'il faut pouvoir assembler souplement. L'imbrication libre des listes et des dictionnaires le permet.

— *Autonomie* : quand deux systèmes échangent un document, toutes les informations doivent être incluses dans la représentation ; en particulier, les données doivent être *auto-décrites* : le contenu vient avec sa propre description. C'est ce que permet la construction clé-valeur dans laquelle chaque valeur, atomique ou complexe, est qualifiée par par sa clé.

La construction récursive d'un document structuré implique une représentation sous forme d'un *arbre* dans lequel on représente à la fois le *contenu* (les valeurs) et la structure (les noms des clés et l'imbrication des constructeurs élémentaires). La Fig. 3.1 montre deux arbres correspondant à la représentation d'une personne. Les noms sont sur les arêtes, les valeurs sur les feuilles.



a. Arbre représentant un agrégat **b. Arbre représentant une composition d'agrégats**

Fig. 3.1 – Représentation arborescente (arêtes étiquetées par les clés)

Cette représentation associe bien une *structure* (l'arbre) et le *contenu* (le texte dans les feuilles). Une autre possibilité est de représenter à la fois la structure et les valeurs comme des nœuds. C'est ce que fait XML

(Fig. 3.2).

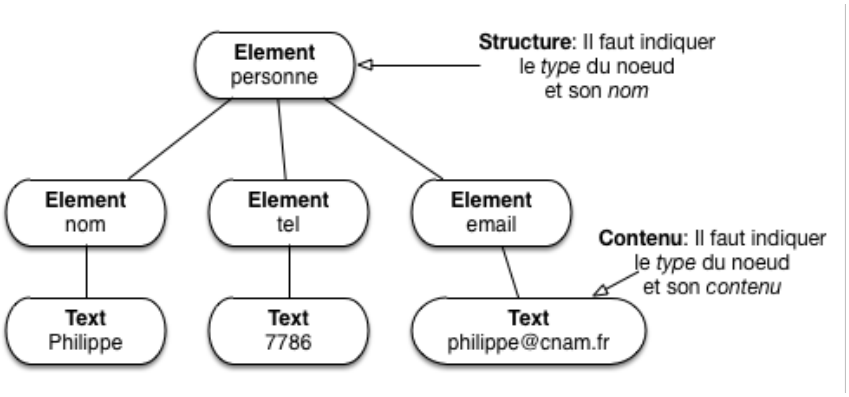


Fig. 3.2 – Représentation arborescente (clés représentées par des nœuds)

Important : les termes varient pour désigner ce que nous appelons *document* ; on pourra parler *d'objet* (JSON), *d'élément* (XML), de *dictionnaire* (Python), de *tableau associatif* (PHP), de *hash map* (Java), etc. D'une manière générale ne vous laissez pas troubler par la terminologie variable, et ne lui accordez pas plus d'importance qu'elle n'en mérite.

3.1.2 Sérialisation des documents structurés

La *sérialisation* désigne la capacité à coder un document sous la forme d'une séquence d'octets qui peut « voyager » sans dégradation sur le réseau, une propriété essentielle dans le cadre d'un système distribué.

Comme vu précédemment, les documents structurés sont des arbres dont chaque partie est auto-décrite. On peut sérialiser un arbre de plusieurs manières, et plusieurs choix sont possibles pour le codage des paires clé-valeur. Les principaux codages sont JSON, XML, et YAML. Nous allons nous contenter du plus léger, JSON, largement majoritaire dans les bases NoSQL. Mais pour bien comprendre qu'il ne s'agit que d'une convention pour sérialiser un arbre, voici un brève comparaison avec XML.

Commençons par la structure de base : les *paires (clé, valeur)*. En voici un exemple, codé en JSON.

```
"nom": "philippe"
```

Et le même, codé en XML.

```
<nom>philippe</nom>
```

Voici un second exemple JSON, montrant un document (qui, rappelons-le, est un dictionnaire).

```
{"nom": "Philippe Rigaux", "tél": 2157786,  "email": "philippe@cnam.fr"}
```

La représentation équivalente en XML est donnée ci-dessous.

```xml
<personne>
  <nom>Philippe Rigaux</nom>
  <tel>2157786</tel>
  <email>philippe@cnam.fr</email>
</personne>
```

On constate tout de suite que le codage XML est beaucoup plus bavard que celui de JSON. XML présente de plus des attributs inclus dans les balises ouvrantes dont l'interprétation est ambigue et qui viennent compliquer inutilement les choix de sérialisation. JSON est un choix clair et raisonnable.

Nous avons parlé de la nécessité de *composer* des structures comme condition essentielle pour obtenir une puissance de représentation suffisante. Sur la base des paires (clé, valeur) et des agrégats vus ci-dessus, une extension immédiate par composition consiste à considérer qu'un *dictionnaire est une valeur*. On peut alors créer une paire clé-valeur dans laquelle la valeur est un dictionnaire, et imbriquer les dictionnaires les uns dans les autres, comme le montre l'exemple ci-dessous.

```json
{
 "nom": {
         "prénom": "Philippe",
         "famille": "Rigaux"
        },
 "tél": 2157786,
 "email": "philippe@cnam.fr"
}
```

Une *liste* est une valeur constituée d'une séquence de valeurs. Les listes sont sérialisées en JSON (où on les appelle *tableaux*) avec des crochets ouvrant/fermant.

```json
[2157786, 2498762]
```

Une liste est une valeur (cf. les définitions précédentes), et on peut donc l'associer à une clé dans un document. Cela donne la forme sérialisée suivante :

```json
{nom: "Philippe", "téls": [2157786, 2498762] }
```

XML en revanche ne connaît pas explicitement la notion de tableau. Tout est uniformément représenté par balisage. Ici on peut introduire une balise `tels` englobant les items de la liste.

```xml
<personne>
  <nom>philippe</nom>
  <tels>
    <tel>2157786</tel>
    <tel>2498762</tel>
  </tels>
</personne>
```

Un des inconvénients de XML est qu'il existe plusieurs manières de représenter les mêmes données, ce qui donne lieu à des réflexions et débats inutiles. Un langage comme JSON propose un ensemble minimal et

suffisant de structures, représentées avec concision. La puissance de XML ne vient pas de sa syntaxe mais de la richesse des normes et outils associés.

Enfin, la sérialisation (JSON ou XML) est conçu pour permettre des transferts sur le réseau sqns détérioration du contenu, ce qui est évidemment essentiel dans le contexte d'un système distribué où les données sont sans cesse échangées.

3.1.3 Abrégé de la syntaxe JSON

Résumons maintenant la syntaxe de JSON qui remplace, il faut bien le dire, tout à fait avantageusement XML dans la plupart des cas à l'exception sans doute de documents « rédigés » contenant beaucoup de texte : rapports, livres, documentation, etc. JSON est concis, simple dans sa définition, et très facile à associer à un langage de programmation (les structures d'un document JSON se transposent directement en structures du langage de programmation, valeurs, listes et objets).

Note : JSON est l'acronyme de *JavaScript Object Notation*. Comme cette expression le suggère, il a été initialement créé pour la sérialisation et l'échange d'objets Javascript entre deux applications. Le scénario le plus courant est sans doute celui des applications Ajax dans lesquelles le serveur (Web) et le client (navigateur) échangent des informations codées en JSON. Cela dit, JSON est un format texte indépendant du langage de programmation utilisé pour le manipuler, et se trouve maintenant utilisé dans des contextes très éloignés des applications Web.

C'est le format de données principal que nous aurons à manipuler. Il est utilisé comme modèle de données natif dans des systèmes NoSQL comme MongoDB, CouchDB, CouchBase, RethinkDB, et comme format d'échange sur le Web par d'innombrables applications, notamment celles basées sur l'architecture REST que nous verrons bientôt.

La syntaxe est très simple et a déjà été en grande partie introduite précédemment. Elle est présentée ci-dessous, mais vous pouvez aussi vous référer à des sites comme http://www.json.org/.

La structure de base est la paire (clé, valeur) (*key-value pair*).

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

Les valeurs atomiques sont :

- les chaînes de caractères (entourées par les classiques apostrophes doubles anglais (droits)),
- les nombres (entiers, flottants)
- les valeurs booléennes (`true` ou `false`).

Voici une paire (clé, valeur) où la valeur est un entier (NB : pas d'apostrophes).

```
"year": 2010
```

Et une autre avec un Booléen (toujours pas d'apostrophes).

```
"oscar": false
```

Les valeurs complexes sont soit des dictionnaires (qu'on appelle plutôt objets en JSON) soit des listes (séquences de valeurs). Un *objet* est un ensemble de paires clé-valeur dans lequel chaque clé ne peut apparaître

qu'une fois au plus.

```
{"last_name": "Fincher", "first_name": "David", "oscar": true}
```

Un objet est une *valeur* complexe et peut être utilisé comme valeur dans une paire clé-valeur avec la syntaxe suivante.

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

Une *liste* (*array*) est une séquence de valeurs dont les types peuvent varier : Javascript est un langage non typé et les tableaux peuvent contenir des éléments hétérogènes, même si ce n'est sans doute pas recommandé. Une liste est une valeur complexe, utilisable dans une paire clé-valeur.

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

La liste suivante est valide, bien que contenant des valeurs hétérogènes.

```
"bricabrac": ["Eisenberg", 1948, {"prenom", "Philippe", "nom": "Rigaux"}, true,␣
↪[1, 2, 3]]
```

Ici, on peut commencer à réfléchir : imaginez que vous écriviez une application qui doit traiter un document come celui ci-dessus. Vous savez que `bricabrac` est une liste (du moins vous le supposez), mais vous ne savez pas du tout à priori quelles valeurs elle contient. Pendant le parcours de la liste, vous allez donc devoir multiplier les tests pour savoir si vous avez affaire à un entier, à une chaîne de caractères, ou même à une valeur complexe, liste, ou objet. Bref, vous devez, *dans votre application*, effectuer le « nettoyage » et les contrôles qui n'ont pas été faits au moment de la constitution du document. Ce point est un aspect très négatif de la production incontrolée de documents (faiblement) structurés, et de l'absence de contraintes (et de schéma) qui est l'une des caractéristiques (négatives) commune aux système NoSQL. Il est développé dans la prochaine section.

L'imbrication est sans limite : on peut avoir des tableaux de tableaux, des tableaux d'objets contenant eux-mêmes des tableaux, etc. Pour représenter un *document* avec JSON, nous adopterons simplement la contrainte que le constructeur de plus haut niveau soit un objet (encore une fois, en JSON, *document* et *objet* sont synonymes).

```
{
  "title": "The Social network",
  "summary": "On a fall night in 2003, Harvard undergrad and \n
     programming genius Mark Zuckerberg sits down at his \n
     computer and heatedly begins working on a new idea. (...)",
  "year": 2010,
  "director": {"last_name": "Fincher",
               "first_name": "David"},
  "actors": [
```

(suite sur la page suivante)

(suite de la page précédente)

```
    {"first_name": "Jesse", "last_name": "Eisenberg"},
    {"first_name": "Rooney", "last_name": "Mara"}
  ]
}
```

3.1.4 Quiz

3.1.5 Mise en pratique (optionnel)

Voici quelques propositions d'explorations pratiques des environnements de documents JSON.

MEP MEP-S1-1 : validation d'un document JSON

Comment savoir qu'un document JSON est bien formé (c'est-à-dire syntaxiquement correct) ? Il existe des validateurs en ligne, bien utiles pour détecter les fautes.

Essayez par exemple http://jsonlint.com/ : copiez-collez les documents JSON donnés précédemment dans le validateur et vérifiez qu'ils sont correct (ou pas…).

Savez-vous quel est le jeu de caractères utilisé pour JSON ? Cherchez sur le Web. Savez-vous comment on peut représenter de longues chaînes de caractères (comme le résumé du film) ? Cherchez (aide : regardez en particulier comment gérer les sauts de ligne).

Le document suivant contient (beaucoup) d'erreurs, à vous de les corriger. Cherchez-les visuellement, puis aidez-vous du validateur.

```
  {
    "title": "Taxi driver",
    "year": 1976,
    "genre": "drama",
    "summary": 'Vétéran de la Guerre du Vietnam, Travis Bickle est chauffeur de
           taxi dans la ville de New York. La violence quotidienne l'affecte peu␣
↪à peu.',
    "country": "USA",
   "director":     {
     "last_name": "Scorcese",
     first_name: "Martin",
     "birth_date": "1962"
    },
   "actors": [
     {
     first_name: "Jodie",
     "last_name": "Foster",
     "birth_date": null,
     "role": "1962"
     }
```

(suite sur la page suivante)

(suite de la page précédente)

```
    {
    first_name: "Robert",
    "last_name": "De Niro",
    "birth_date": "1943",
    "role": "Travis Bickle ",
    }
  }

Au-delà des documents *bien formés*, on peut aussi contrôler qu'un document est␣
↪*valide*
par rapport à une spécification (un schéma).  Voir les exercices sur les schémas␣
↪JSON ci-dessous.
```

MEP MEP-S1-2 : récupérer des jeux de données

Nous allons récupérer aux formats JSON le contenu d'une base de données relationnelle pour disposer de documents à structure forte. Pour cela, rendez-vous sur le site http://deptfod.cnam.fr/bd/tp/datasets/. Sur ce site vous trouverez des fichiers en différents formats qui nous utiliserons dans d'autres mises en pratique.

MEP MEP-S1-3 : JSON et l'Open Data

L'Open Data désigne le mouvement de mise à disposition des données afin de favoriser leur diffusion et la construction d'applications. Les données sont fournies au format JSON ! Regardez les sites suivants, récupérez quelques documents, commencez à imaginer quel applications vous pourriez construire.

— https://www.data.gouv.fr/fr/
— http://data.enseignementsup-recherche.gouv.fr/

MEP MEP-S1-4 : produire un jeu de documents JSON volumineux

Pour tester des systèmes avec un jeu de données de taille paramétrable, nous pouvons utiliser des générateurs de données. Voici quelqes possibilités qu'il vous est suggéré d'explorer.

— le site http://generatedata.com/ est très paramétrable mais ne permet malheureusement pas (aux dernières nouvelles) d'engendrer des documents imbriqués ; à étudier quand même pour produire des tableaux volumineux ;
— https://github.com/10gen-labs/ipsum est un générateur de documents JSON spécifiquement conçu pour fournir des jeux de test à MongoDB, un système NoSQL que nous allons étudier. Une version adaptée à python3 de cet outil est disponible sur notre site http://b3d.bdpedia.fr/files/ipsum-master.zip

Ipsum produit des documents JSON conformes à un schéma (http://json-schema.org). Un script Python (vous devez avoir un interpréteur Python installé sur votre machine) prend ce schéma en entrée et produit un nombre paramétrable de documents. Voici un exemple d'utilisation.

```
python ./pygenipsum.py --count 1000000 schema.jsch > bd.json
```

Lisez le fichier `README` pour en savoir plus. Vous êtes invités à vous inspirer des documents JSON représentant nos films pour créer un schéma et engendrer une base de films avec quelques millions de documents. Pour notre base `movies`, vous pouvez récupérer le schéma JSON des documents. (Suggestion : allez jeter un œil à http://www.jsonschema.net/).

```
{
    "type":"object",
    "$schema": "http://json-schema.org/draft-03/schema",
    "properties":{
         "_id": {"type":"string", "ipsum": "id"},
         "actors": {
                 "type":"array",
                 "items":
                         {
                                 "type":"object",
                                 "required":false,
                                  "minItems": 2,
                  "maxItems": 6,
                                 "properties":{
                                         "_id": {"type":"string", "ipsum": "id"},
                                         "birth_date": {"type":"string", "format
↪": "date"},
                                         "first_name": {"type":"string", "ipsum
↪": "fname"},
                                         "last_name": {"type":"string", "ipsum":
↪"fname"},
                                         "role": {"type":"string"}
                                 }
                         }
         },
         "genre": {"type":"string",
                  "enum": [ "Drame", "Comédie", "Action", "Guerre", "Science-
↪Fiction"]},
         "summary": {"type":"string", "required":false},
         "title": {"type":"string"},
         "year": {"type":"integer"},
         "country": {"type":"string",  "enum": [ "USA", "FR", "IT"]},
         "director": {
                 "type":"object",
                 "properties":{
                         "_id":{"type":"string"},
                         "birth_date": {"type":"string"},
                         "first_name":  {"type":"string", "ipsum": "fname"},
                         "last_name":{"type":"string", "ipsum": "fname"}
```

(suite sur la page suivante)

(suite de la page précédente)

```
                    }
                }
            }
        }
```

3.2 S2. Modélisation des collections

Supports complémentaires

- Diapositives: modélisation de bases documentaires
- Vidéo sur la modélisation relationnelle
- Vidéo sur la modélisation basée sur les documents structurés

Nous abordons maintenant une question très importante dans le cadre de la mise en œuvre d'une grande base de données constituée de documents : comment *modéliser* ces documents pour satisfaire les besoins de l'application ? Et plus précisément :

- quelle est la structure de ces documents ?
- quelles sont les contraintes qui portent sur le contenu des documents ?

Cette question est bien connue dans le contexte des bases de données relationnelles, et nous allons commencer par rappeler la méthode bien établie. Pour les bases NoSQL, il n'existe pas de méthodologie équivalente. Une bonne (ou mauvaise) raison est d'ailleurs qu'il n'existe pas de modèle normalisé, et que la modélisation doit s'adapter aux caractéristiques de chaque système.

Note : Certains semblent considérer que la question ne se pose pas et qu'on peut entasser les données dans la base, n'importe comment, et voir plus tard ce que l'on peut en faire. C'est un(e absence de) choix porteur de redoutables conséquences pour la suite. La dernière partie de cette section donne mon avis à ce sujet.

Je vais donc extrapoler la méthodologie de conception relationnelle pour étudier ce que l'on peut obtenir avec un modèle de documents structurés.

3.2.1 Conception d'une base relationnelle

Note : Cette partie reprend de manière abrégée le contenu du chapitre « Conception d'une base de données » dans le support de cours Bases de données relationnelles. La lecture complète de ce chapitre est conseillée pour aller plus loin.

Voyons comment on pourrait modéliser notre base de films avec leurs réalisateurs et leurs acteurs. La démarche consiste à :

- déterminer les « entités » (film, réalisateurs, acteurs) pertinentes pour l'application ;

— définir une méthode *d'identification* de chaque entité ; en pratique on recourt à la définition d'un *identifiant* artificiel (il n'a aucun rôle descriptif) qui permet d'une part de s'assurer qu'une même « entité » est représentée une seule fois, d'autre part de référencer une entité par son identifiant.
— préserver le lien entre les entités.

Voici une illustration informelle de la méthode, dans le contexte d'une base relationnelle où l'on suit une démarche fondée sur des règles de *normalisation*. Nous reprendrons ensuite une approche plus générale basée sur la notation Entité/association.

Commençons par les deux premières étapes. On va d'abord distinguer deux types d'entités : les films et les réalisateurs. On en déduit deux tables, celle des films et celle des réalisateurs.

Note : Comment distingue-t-on des entités et modélise-t-on correctement un domaine ? Il n'y a pas de méthode magique : c'est du métier, de l'expérience, de la pratique, des erreurs, ...

Ensuite, on va ajouter à chaque table un attribut spécial, l'identifiant, désigné par `id`, dont la valeur est simplement un compteur auto-incrémenté. On obtient le résultat suivant.

id	titre	année
1	Alien	1979
2	Vertigo	1958
3	Psychose	1960
4	Kagemusha	1980
5	Volte-face	1997
6	Pulp Fiction	1995
7	Titanic	1997
8	Sacrifice	1986

La table des films.

id	nom	prénom	année
101	Scott	Ridley	1943
102	Hitchcock	Alfred	1899
103	Kurosawa	Akira	1910
104	Woo	John	1946
105	Tarantino	Quentin	1963
106	Cameron	James	1954
107	Tarkovski	Andrei	1932

La table des réalisateurs

Un souci constant dans ce type de modélisation est d'éviter toute redondance. Chaque film, et chaque information relative à un film, ne doit être représentée qu'une fois. La redondance dans une base de données est susceptible de soulever de gros problèmes, et notamment des *incohérences* (on met à jour une des versions et pas les autres, et on se sait plus laquelle est correcte).

Il reste à représenter le lien entre les films et les metteurs en scène, sans introduire de redondance. Maintenant que nous avons défini les identifiants, il existe un moyen simple pour indiquer quel metteur en scène a réalisé

un film : associer l'identifiant du metteur en scène au film. L'identifiant sert alors de *référence* à l'entité. On ajoute un attribut `idRéalisateur` dans la table *Film*, et on obtient la représentation suivante.

id	titre	année	idRéalisateur
1	Alien	1979	101
2	Vertigo	1958	102
3	Psychose	1960	102
4	Kagemusha	1980	103
5	Volte-face	1997	104
6	Pulp Fiction	1995	105
7	Titanic	1997	106
8	Sacrifice	1986	107

Cette représentation est correcte. La redondance est réduite au minimum puisque seule l'identifiant du metteur en scène a été déplacé dans une autre table. Pour peu que l'on s'assure que cet identifiant ne change *jamais*, cette redondance n'induit aucun effet négatif.

Cette représentation normalisée évite des inconvénients qu'il est bon d'avoir en tête :

— *pas de redondance*, donc toute mise à jour affecte l'unique représentation, sans risque d'introduction d'incohérences ;

— *pas de dépendance forte induisant des anomalies de mise à jour* : on peut par exemple détruire un film sans affecter les informations sur le réalisateur, ce qui ne serait pas le cas s'ils étaient associés dans la même table (ou dans un même document : voir plus loin).

Ce gain dans la qualité du schéma n'a pas pour contrepartie une perte d'information. Il est en effet facile de voir qu'elle peut être reconstituée intégralement. En prenant un film, on obtient l'identifiant de son metteur en scène, et cet identifiant permet de trouver *l'unique* ligne dans la table des réalisateurs qui contient toutes les informations sur ce metteur en scène. Ce processus de reconstruction de l'information, dispersée dans plusieurs tables, peut s'exprimer avec les opérations relationnelles, et notamment la *jointure*.

Il reste à appliquer une méthode systématique visant à aboutir au résultat ci-dessus, et ce même dans des cas beaucoup plus complexes. Celle universellement adoptée (avec des variantes) s'appuie sur les notions *d'entité* et *d'association*. En voici une présentation très résumée.

La méthode permet de distinguer les *entités* qui constituent la base de données, et les *associations* entre ces entités. Un schéma E/A décrit l'application visée, c'est-à-dire une *abstraction* d'un domaine d'étude, pertinente relativement aux objectifs visés. Rappelons qu'une abstraction consiste à choisir certains aspects de la réalité perçue (et donc à éliminer les autres). Cette sélection se fait en fonction de certains *besoins*, qui doivent être précisément définis, et rélève d'une démarche d'analyse qui n'est pas abordée ici.

Par exemple, pour notre base de données *Films*, on n'a pas besoin de stocker dans la base de données l'intégralité des informations relatives à un internaute, ou à un film. Seules comptent celles qui sont importantes pour l'application. Voici le schéma décrivant cette base de données *Films* (Fig. 3.3). On distingue

— des *entités*, représentées par des rectangles, ici *Film*, *Artiste*, *Internaute* et *Pays* ;

— des *associations* entre entités représentées par des liens entre ces rectangles. Ici on a représenté par exemple le fait qu'un artiste *joue* dans des films, qu'un internaute *note* des films, etc.

Chaque entité est caractérisée par un ensemble d'attributs, parmi lesquels un ou plusieurs forment l'identifiant unique (en gras). Nous l'avons appelé **id** pour *Film* et *Artiste*, **code** pour le pays. Le nom de l'attribut-identifiant est peu important, même si la convention **id** est très répandue.

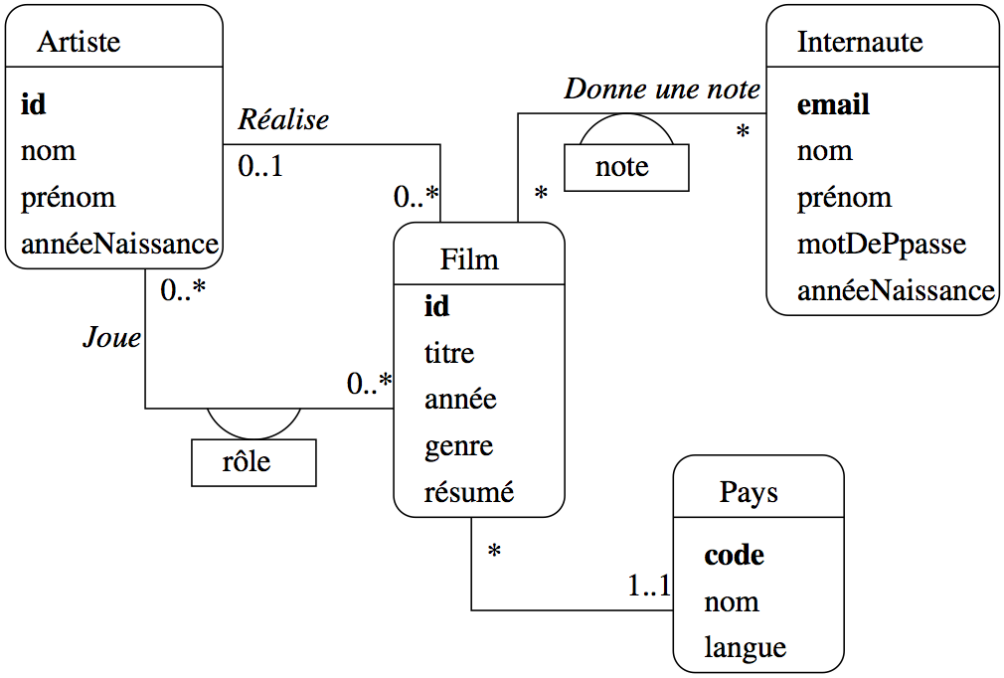


Fig. 3.3 – Le schéma E/A des films

Les associations sont caractérisées par des *cardinalités*. La notation 0..* sur le lien *Réalise*, du côté de l'entité *Film*, signifie qu'un artiste peut réaliser plusieurs films, ou aucun. La notation 0..1 du côté *Artiste* signifie en revanche qu'un film ne peut être réalisé que par au plus un artiste. En revanche dans l'association *Donne une note*, un internaute peut noter plusieurs films, et un film peut être noté par plusieurs internautes, ce qui justifie l'a présence de 0..* aux deux extrêmités de l'association.

Outre les propriétés déjà évoquées (simplicité, clarté de lecture), évidentes sur ce schéma, on peut noter aussi que la modélisation conceptuelle est totalement indépendante de tout choix d'implantation. Le schéma de la Fig. 3.3 ne spécifie aucun système en particulier. Il n'est pas non plus question de type ou de structure de données, d'algorithme, de langage, etc. En principe, il s'agit donc de la partie la plus stable d'une application. Le fait de se débarrasser à ce stade de la plupart des considérations techniques permet de se concentrer sur l'essentiel : que veut-on stocker dans la base ?

Schémas relationnels

La transposition d'une modélisation entité/association s'effectue sous la forme d'un *schéma relationnel*. Un tel schéma énonce la structure et les contraintes portant sur les données. À partir de la modélisation précédente, par exemple, on obtient les tables *Film*, *Artiste* et *Role* suivantes :

```
create table Artiste  (idArtiste integer not null,
                       nom varchar (30) not null,
                       prenom varchar (30) not null,
                       anneeNaiss integer,
                       primary key (idArtiste),
                       unique (nom, prenom))
```

(suite sur la page suivante)

(suite de la page précédente)

```
create table Film  (idFilm integer not null,
                    titre    varchar (50) not null,
                    annee    integer not null,
                    idRealisateur    integer not null,
                    genre varchar (20) not null,
                    resume      varchar(255),
                    codePays    varchar (4),
                    primary key (idFilm),
                    foreign key (idRealisateur) references Artiste);

create table Role (idFilm  integer not null,
                   idActeur Iinteger not null,
                   nomRole  varchar(30),
                   primary key (idActeur,idFilm),
                   foreign key (idFilm) references Film,
                   foreign key (idActeur) references Artiste);
```

Le schéma impose des contraintes sur le contenu de la base. On a par exemple spécifié qu'on ne doit pas trouver deux artistes avec la même paire de valeurs (prénom, nom). La contrainte `not null` indique qu'une valeur doit toujours être présente. Une contrainte très importante est la contrainte d'intégrité référentielle (`foreign key`) : elle garantit par exemple que la valeur de `idRéalisateur` correspond bien à une clé primaire de la table `Artiste`. En d'autres termes : un film fait référence, grâce à `idRéalisateur`, à un artiste qui est représenté dans la base. *Le système garantit que ces contraintes sont respectées*.

Voici un exemple de contenu pour la table *Artiste*.

Tableau 3.1 – Table des artistes

id	nom	prénom
11	Travolta	John
27	Willis	Bruce
37	Tarantino	Quentin
167	De Niro	Robert
168	Grier	Pam

On peut remarquer que le schéma et la base sont représentés séparément, contrairement aux documents structurés où chaque valeur est associée à une clé qui indique sa signification. Ici, le placement d'une valeur dans un colonne spécifique suffit.

Voici un exemple pour la table des films, illustrant la notion de clé étrangère.

Tableau 3.2 – Table des films

id	titre	année	idRéal
17	Pulp Fiction	1994	37
57	Jackie Brown	1997	37

Une valeur de la colonne `idReal`, une clé étrangère, est *impérativement* la valeur d'une clé primaire existante dans la table *Artiste*. Cette contrainte forte est vérifiée par le système relationnel et garantit que la base est

saine. Il est impossible de faire référence à un metteur en scène qui n'existe pas.

Dans une base relationnelle (bien conçue) les données sont cohérentes et cela apporte une garantie forte aux applications qui les manipulent : pas besoin de vérifier par exemple, quand on lit le film 17, que l'artiste avec l'identifiant 37 existe bien : c'est garanti par le schéma.

En contrepartie, la distribution des données dans plusieurs tables rend le contenu de chacune incomplet. Le système de référencement par clé étrangère en particulier ne donne aucune indication directe sur l'entité référencée, d'où des tables au contenu succinct et non interprétable. Voici la table *Role*.

Tableau 3.3 – Table des rôles

idFilm	idArtiste	rôle
17	11	Vincent Vega
17	27	Butch Coolidge
17	37	Jimmy Dimmick

En la regardant, on ne sait pas grand chose : il faut aller voir par exemple, pour le premier rôle, que le film 17 est *Pulp Fiction*, et l'artiste 11, John Travolta. En d'autres termes, il faut effectuer une opération rapprochant des données réparties dans plusieurs tables. Un système relationnel nous fournit cette opération : c'est la *jointure*. Voici comment on reconstituerait l'information sur le rôle « Vincent Vega » en SQL.

```
select titre, nom, prénom, role
from Film, Artiste, Role
where role='Vincent Vega'
and Film.id = Role.idFilm
and Artiste.id = Role.idActeur
```

La représentation des informations relatives à une même « entité » (un film) dans plusieurs tables a une autre conséquence qui motive (parfois) le recours à une représentation par document structuré. Il faut de fait effectuer *plusieurs écritures* pour une même entité, et donc appliquer une *transaction* pour garantir la cohérence des mises à jour. On peut considérer que ces précautions et contrôles divers pénalisent les performances (pour des raisons claires : assurer la cohérence de la base).

Note : Pour la notion de transaction, reportez-vous au chapitre introductif de http://sys.bdpedia.fr.

Ce qu'il faut retenir

En résumé, les caractéristiques d'une modélisation relationnelle sont

— Un objectif de *normalisation* qui vise à éviter à la fois toute redondance et toute perte d'information ;

1. la redondance est évitée en découpant les données avec une granularité fine, et en les stockant indépendamment les unes des autres ;
2. la perte d'information est évitée en utilisant un système de référencement basé sur les clés primaires et clés étrangères.

— Les données sont contraintes par un *schéma* qui impose des règles sur le contenu de la base

— Il n'y a aucune hiérarchie dans la représentation des entités ; une entité comme *Pays*, qui peut être considérée comme secondaire, a droit à sa table dédiée, tout comme l'entité *Film* qui peut être considérée comme essentielle ; *on ne pré-suppose pas en relationnel, l'importance respective des entités représentées* ;
— La distribution des données dans plusieurs tables est compensée par la capacité de SQL à effectuer des *jointures* qui exploitent le plus souvent le système de référencement (clé primaire, clé étrangère) pour associer des lignes stockées séparément.
— Plusieurs écritures transactionnelles peuvent être nécessaires pour créer une seule entité.

Ce modèle est cohérent. Il fonctionne très bien, depuis très longtemps, au moins pour des données fortement structurées comme celles que nous étudions ici. Il permet de construire des bases pérennes, conçues en grand partie indépendamment des besoins ponctuels d'une application, représentant un domaine d'une manière suffisamment générique pour satisfaire tous les types d'accès, mêmes s'ils n'étaient pas envisagés au départ.

Voyons maintenant ce qu'il en est avec un modèle de document structuré.

3.2.2 Conception NoSQL avec documents structurés

En relationnel, on a des lignes (des nuplets pour être précis) et des tables (des relations). Dans le contexte du NoSQL, on va parler de *documents* et de *collections* (de documents).

Documents et collections

Notons pour commencer que la représentation arborescente est très puissante, plus puissante que la représentation offerte par la structure tabulaire du relationnel. Dans un nuplet relationnel, on ne trouve que des valeurs dites atomiques, non décomposables. Il ne peut y avoir qu'un seul genre pour un film. Si ce n'est pas le cas, il faut (processus de normalisation) créer une table des genres et la lier à la table des films (je vous laisse trouver le schéma correspondant, à titre d'exercice). Cette nécessité de distribuer les données dans plusieurs tables est une lourdeur souvent reprochée à la modélisation relationnelle.

Avec un document structuré, il est très facile de représenter les genres comme un tableau de valeurs, ce qui rompt la première règle de normalisation.

```
{
  "title": "Pulp fiction",
  "year": "1994",
  "genre": ["Action", "Policier", "Comédie"]
  "country": "USA"
}
```

Par ailleurs, il est également facile de représenter une table par une collection de documents structurés. Voici la table des artistes en notation JSON.

```
 [
  artiste:  {"id": 11, "nom": "Travolta", "prenom": "John"},
  artiste:  {"id": 27, "nom": "Willis", "prenom": "Bruce"},
  artiste:  {"id": 37, "nom": "Tarantino", "prenom": "Quentin"},
  artiste:  {"id": 167, "nom": "De Niro", "prenom": "Robert"},
```

(suite sur la page suivante)

(suite de la page précédente)

```
  artiste:  {"id": 168, "nom": "Grier", "prenom": "Pam"}
]
```

On pourrait donc « encoder » une base relationnelle sous la forme de documents structurés, et chaque document pourrait être plus complexe structurellement qu'une ligne dans une table relationnelle.

D'un autre côté, une telle représentation, pour des données *régulières*, n'est pas du tout efficace à cause de la redondance de l'auto-description : à chaque fois on répète le nom des clés, alors qu'on pourrait les factoriser sous forme de schéma et les représenter indépendamment (ce que fait un système relationnel, voir ci-dessus).

L'auto-description n'est valable qu'en cas de variation dans la structure, ou éventuellement pour coder l'information de manière autonome en vue d'un échange. Une représentation arborescente XML / JSON est donc plus appropriée pour des données de structure *complexe* et surtout *flexible*.

Le pouvoir de l'imbrication des structures

Dans une modélisation relationnelle, nous avons dû séparer les films et les artistes dans deux tables distinctes, et lier chaque film à son metteur en scène par une clé étrangère. Grâce à l'imbrication des structures, il est possible avec un document structuré de représenter l'information de la manière suivante :

```
{
  "title": "Pulp fiction",
  "year": "1994",
  "genre": "Action",
  "country": "USA",
  "director": {
    "last_name": "Tarantino",
    "first_name": "Quentin",
    "birth_date": "1963"
  }
}
```

On a *imbriqué* un objet dans un autre, ce qui ouvre la voie à la représentation d'une entité par un unique document complet.

Important : Notez que nous n'avons plus besoin du système de référencement par clés primaires / clés étrangères, remplacé par l'imbrication qui associe *physiquement* les entités *film* et *artiste*.

Prenons l'exemple du film « Pulp Fiction » et son metteur en scène et ses acteurs. En relationnel, pour reconstituer l'ensemble du film « Pulp Fiction », il faut suivre les références entre clés primaires et clés étrangères. C'est ce qui permet de voir que Tarantino (clé = 37) est réalisateur de Pulp Fiction (clé étrangère `idRéal` dans la table Film, avec la valeur 37) et joue également un rôle (clé étrangère `idArtiste` dans la table Rôle).

Tout peut être représenté par un unique document structuré, en tirant parti de l'imbrication d'objets dans des tableaux.

```
{
  "title": "Pulp fiction",
  "year": "1994",
  "genre": "Action",
  "country": "USA",
  "director": {
    "last_name": "Tarantino",
    "first_name": "Quentin",
    "birth_date": "1963" },
  "actors": [
    {"first_name": "John",
     "last_name": "Travolta",
     "birth_date": "1954",
     "role": "Vincent Vega" },
    {"first_name": "Bruce",
     "last_name": "Willis",
     "birth_date": "1955",
     "role": "Butch Coolidge" },
    {"first_name": "Quentin",
     "last_name": "Tarantino",
     "birth_date": "1963",
     "role": "Jimmy Dimmick"}
   ]
  }
```

Nous obtenons une unité d'information autonome représentant l'ensemble des informations relatives à un film (on pourrait bien entendu en ajouter encore d'autres, sur le même principe). Ce rassemblement offre des avantages forts dans une perspective de performance pour des collections à très grande échelle.

- **Plus besoin de jointure** : il est inutile de faire des jointures pour reconstituer l'information puisqu'elle n'est plus dispersée, comme en relationnel, dans plusieurs tables.
- **Plus besoin de transaction (?)** : une écriture (du document) suffit ; pour créer toutes les données du film « Pulp fiction » ci-dessus, il faudrait écrire 1 fois dans la table *Film*, 3 fois dans la table *Artiste* ; 3 fois dans la table *Role*.
  De même, une lecture suffit pour récupérer l'ensemble des informations.
- **Adaptation à la distribution**. Si les documents sont autonomes, il est très facile des les déplacer pour les répartir au mieux dans un système distribué ; l'absence de lien avec d'autres documents donne la possibilité d'organiser librement la collection.

Cela semble séduisant… De plus, les transactions et les jointures sont deux mécanismes assez compliqués à mettre en œuvre dans un environnement distribué. Ne pas avoir à les implanter simplifie considérablement la création de systèmes NoSQL, d'où la prolifération à laquelle nous assistons. Tout système sachant faire des *put()* et des *get()* peut prétendre à l'appellation !

Mais il y a bien entendu des inconvénients.

Les inconvénients

En observant bien le document ci-dessus, on réalise rapidement qu'il introduit cependant deux problèmes importants.

- *Hiérarchisation des accès* : la représentation des films et des artistes n'est pas symétrique ; les films apparaissent près de la racine des documents, les artistes sont enfouis dans les profondeurs ; l'accès aux films est donc privilégié (on ne peut pas accéder aux artistes sans passer par eux) ce qui peut ou non convenir à l'application.
- *Perte d'autonomie des entités*. Il n'est plus possible de représenter les informations sur un metteur en scène si on ne connaît pas au moins un film ; inversement, en supprimant un film (e.g., Pulp Fiction), on risque de supprimer définitivement les données sur un artiste (e.g., Tarantino).
- *Redondance* : la même information doit être représentée plusieurs fois, ce qui est tout à fait fâcheux. Quentin Tarantino est représenté deux fois, et en fait il sera représenté autant de fois qu'il a tourné de films (ou fait l'acteur quelque part).

En extrapolant un peu, il est clair que la contrepartie d'un document *autonome* contenant toutes les informations qui lui sont liées est l'absence de *partage* de sous-parties potentiellement communes à plusieurs documents (ici, les artistes). On aboutit donc à une redondance qui mène immanquablement à des incohérences diverses.

Par ailleurs, on privilégie, en modélisant les données comme des documents, une certaine perspective de la base de données (ici, les films), ce qui n'est pas le cas en relationnel où toutes les informations sont au même niveau. Avec la représentation ci-dessus par exemple, comment connaître tous les films tournés par Tarantino ? Il n'y a pas vraiment d'autre solution que de lire tous les documents, c'est compliqué et surtout coûteux.

Ce sont des inconvénients *majeurs*, qui risquent à terme de rendre la base de données inexploitable. Il faut bien les prendre en compte avant de se lancer dans l'aventure du NoSQL. D'autant que …

Et le schéma ?

Les systèmes NoSQL (à quelques exceptions près, cf. Cassandra) ne proposent pas de schéma, ou en tout cas rien d'équivalent aux schémas relationnels. Il existe un gain apparent : on peut tout de suite, sans effectuer la moindre démarche de modélisation, commencer à insérer des documents. Rapidement la structure de ces documents change, et on ne sait plus trop ce qu'on a mis dans la base qui devient une véritable poubelle de données.

Si on veut éviter cela, c'est au niveau de l'application effectuant des insertions qu'il faut effectuer la vérification des contraintes qu'un système relationnel peut nativement prendre en charge. Il faut également, pour toute application exploitant les données, effectuer des contrôles puisqu'il n'y a pas de garantie de cohérence ou de complétude.

L'absence de schéma est (à mon avis) un autre inconvénient fort des systèmes NoSQL.

3.2.3 Ma conclusion : *Relationnel ou NoSQL ?*

Note : Ce qui suit constitue un ensemble de conclusions que je tire personnellement des arguments qui précèdent. Je ne cherche pas à polémiquer, mais à éviter de gros soucis à beaucoup d'enthousiastes qui penseraient découvrir une innovation mirifique dans le NoSQL. Contre-arguments et débats sont les bienvenus !

La (ma) conclusion de ce qui précède est que les systèmes NoSQL sont beaucoup moins puissants, fonctionnellement parlant, qu'un système relationnel. Ils présentent quelques caractéristiques potentiellement avantageuses dans *certaines* situations, essentiellement liés à leur capacité à passer à l'échelle comme système distribué. Ils ne devraient donc être utilisés que dans des situations très précises, et rarement rencontrées. Résumons les inconvénients :

- *Un modèle de données puissant, mais menant à des représentations asymétriques des informations.* Certaines applications seront privilégiées, et d'autres pénalisées. Une base de données est (de mon point de vue) beaucoup plus pérenne que les applications qui l'exploitent, et il est dangereux de concevoir une base pour une application initiale, et de s'apercevoir qu'elle est inadaptée ensuite.
- *Pas de jointure, pas de langage de requêtes et en tout cas non normalisé.* Cela implique une chute potentielle extrêmement forte de la productivité. Êtes-vous prêts à écrire un programme à chaque fois qu'il faut effectuer une mise à jour, même minime ?
- *Pas de schéma, pas de contrôle sur les données*. Ne transformez pas votre base en déchèterie de documents ! La garantie de ce que l'on va trouver dans la base évite d'avoir à multiplier les tests dans les applications.
- *Pas de transactions.* Une transaction assure la cohérence des données (cf. le support en ligne http://sys.bdpedia.fr). Êtes-vous prêts à baser un site de commerce électronique sur un système NoSQL qui permettra de livrer des produits sans garantir que vous avez été payé ?

D'une manière générale, ce qu'un système NoSQL ne fait pas par rapport à un système relationnel doit être pris en charge par les applications (contrôle de cohérence, opérations de recherche complexes, vérification du format des documents). C'est potentiellement une grosse surcharge de travail et un risque (comment garantir que les contrôles ou tests sont correctement implantés ?).

Alors, quand peut-on recourir un système NoSQL ? Il existe des niches, celles qui présentent une ou plusieurs des caractéristiques suivantes :

- *Des données très spécifiques, peu ou faiblement structurées.* graphes, séries temporelles, données textuelles et multimédia. Les systèmes relationnels se veulent généralistes, et peuvent donc être moins adaptés à des données d'un type très particulier.
- *Peu de mises à jour, beaucoup de lectures*. C'est le cas des applications de type analytique par exemple : on écrit une fois, et ensuite on lit et relit pour analyser. Dans ce cas, la plupart des inconvénients ci-dessus disparaissent ou sont minorés.
- *De très gros volumes*. Un système relationnel peut souffrir pour calculer efficament des jointures pour de très gros volumes (ordre de grandeur : des données dépassant les capacités d'un unique ordinateur, soit quelques TéraOctets à ce jour). Dans ce cas on peut vouloir dénormaliser, recourir à un système NoSQL, et assumer les dangers qui en résultent.
- *De forts besoins en temps réel*. Si on veut obtenir des informations en quelques ms, même sur de très grandes bases, certains systèmes NoSQL peuvent être mieux adaptés.

Voilà ! Un cas typique et justifié d'application est celui de l'accumulation de données dans l'optique de construire des modèles statistiques. On accumule des données sur le comportement des utilisateurs pour

construire un modèle de recommandation par exemple. La base est alors une sorte d'entrepôt de données, avec des insertions constantes et aucune mise à jour des données existantes.

NoSQL = Not Only SQL. En dehors de ces niches, je pense très sincèrement que dans la plupart des cas le relationnel reste un meilleur choix et fournit des fonctionnalités beaucoup plus riches pour construire des applications. Le reste du cours vous permettra d'apprécier plus en profondeur la technicité de certains arguments. Après ce sera à vous de juger.

3.2.4 Quiz

3.3 Exercices

Exercice Ex-S2-1 : document = graphe

Représenter sous forme de graphe le film complet « Pulp Fiction » donné précédemment.

Correction

La Fig. 3.4 montre la forme arborescente dans la variante où les étiquettes sont sur les arêtes. Les sous-graphes pour Bruce Willis et Quentin Tarantino (en tant qu'acteur) ne sont pas développés.

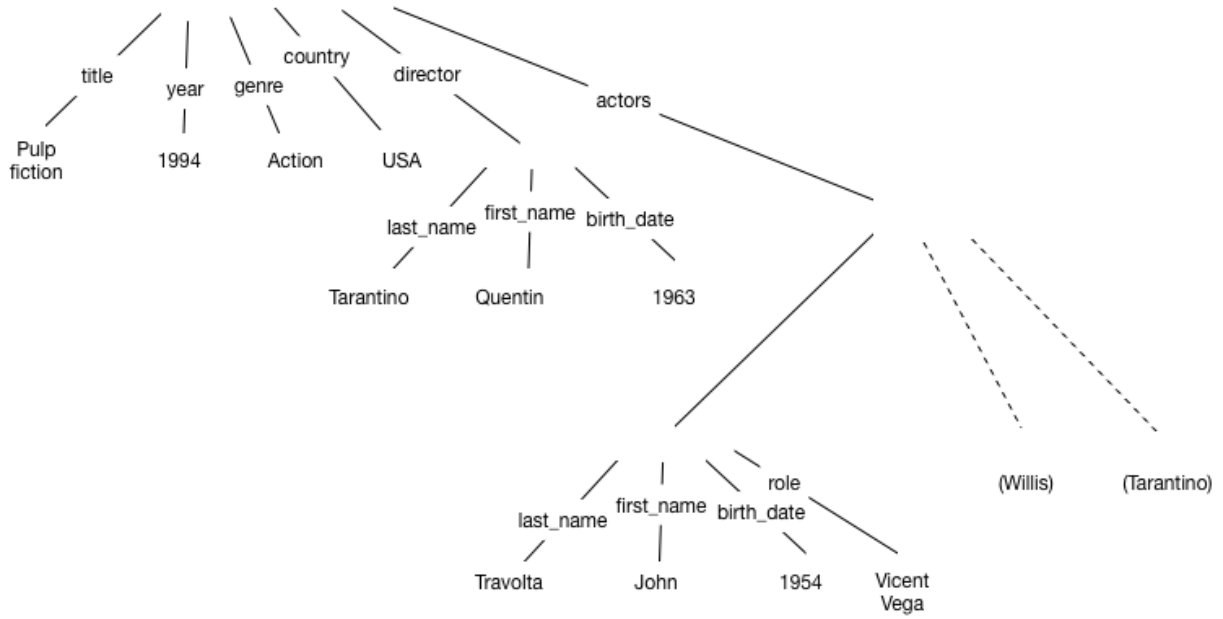


Fig. 3.4 – Représentation arborescente du film Pulp Fiction

La représentation avec les étiquettes sur les arêtes correspond à l'encodage JSON. XML s'appuie sur une représentation plus compliquée dans laquelle les étiquettes sont des nœuds intermédiaires. Cette différence

explique en grande partie l'abandon de XML comme langage de modélisation de données. Les sous-graphes pour Bruce Willis et Quentin Tarantino (en tant qu'acteur) ne sont pas développés.

Exercice Ex-S2-2 : Privilégions les artistes

Reprendre la petite base des films (les 3 tables données ci-dessus) et donner un document structuré donnant toutes les informations disponibles sur Quentin Tarantino. On veut donc représenter un document centré sur les artistes et pas sur les films.

Correction

Voici une représentation possible. Cette fois c'est la représentation des films qui est redondante.

```
{
  "_id": "37",
   "first_name": "Quentin",
  "last_name": "Tarantino",
  "films_dirigés" : [
    {
     "title": "Pulp fiction",
     "year": "1994",
     "actors": [
         {"artist:11", "role": "Vincent Vega" },
         {"artist:27", "role": "Butch Coolidge"}
       ]
     },
    {
     "title": "Jacky Brown",
     ...
    },
     ...
   ],
  "films_joués": [
    {
     "title": "Pulp fiction",
     ...
    },
    {
     "title": "Reservoir Dogs",
     ...
     },
     ...
  ]
}
```

Cette représentation convient pour des tâches d'analyse, en considérant qu'un document est créé une fois pour

toutes et jamais modifié. Mais elle est inexploitable pour une base dans laquelle on effectue des mises à jour fréquentes (bases dites « transactionnelles ») à cause de la difficulté à préserver la cohérence des données.

Exercice Ex-S2-3 : Comprendre la notion de document structuré

Vous gérez un site de commerce électronique et vous attendez des dizaines de millions d'utilisateurs (ou plus). Vous vous demandez quelle base de données utiliser : relationnel ou NoSQL ?

Les deux tables suivantes représentent la modélisation relationnelle pour les utilisateurs et les visites de pages (que vous enregistrez bien sûr pour analyser le comportement de vos utilisateurs).

Tableau 3.4 – Table des utilisateurs

id	email	nom
1	s@cnam.fr	Serge
2	b@cnam.fr	Benoît

Tableau 3.5 – Table des visites

idUtil	page	nbVisites
1	http://cnam.fr/A	2
2	http://cnam.fr/A	1
1	http://cnam.fr/B	1

Proposez une représentation de ces informations sous forme de document structuré
- — en privilégiant l'accès par les utilisateurs ;
- — en privilégiant l'accès par les pages visitées.

Correction

Voici une représentation possible, centrée utilisateurs.

```
[
{
  "_id": "1",
  "email": "s@cnam.fr",
  "nom": "Serge",
  "visites" : [
    {
      "page": "http://cnam.fr/A",
      "nbVisites": 2
    },
    {
      "page": "http://cnam.fr/B",
      "nbVisites": 1
    }
```

(suite sur la page suivante)

(suite de la page précédente)

```
  ]
},
{
 "_id": "2",
 "email": "b@cnam.fr",
 "nom": "Benoît",
 "visites" : [
   {
     "page": "http://cnam.fr/A",
     "nbVisites": 2
   }
  ]
 }
]
```

La représentation centrée sur les pages s'en déduit aisément.

Exercice Ex-S2-4 : extrait de l'examen du 16 juin 2016

Le service informatique du Cnam a décidé de représenter ses données sous forme de documents structurés pour faciliter les processus analytiques. Voici un exemple de documents centrés sur les étudiants et incluant les Unités d'Enseignement (UE) suivies par chacuns.

```
{
  "_id": 978,
  "nom": "Jean Dujardin",
  "UE": [{"id": "ue:11", "titre": "Java", "note": 12},
        {"id": "ue:27", "titre": "Bases de données", "note": 17},
        {"id": "ue:37",  "titre": "Réseaux", "note": 14}
        ]
}
{
  "_id": 476,
   "nom": "Vanessa Paradis",
   "UE": [{"id":  "ue:13",  "titre": "Méthodologie", "note": 17,
          {"id": "ue:27",  "titre": "Bases de données", "note": 10},
          {"id":  "ue:76",   "titre": "Conduite projet", "note": 11}
         ]
}
```

— Sachant que ces documents sont produits à partir d'une base relationnelle, reconstituez le schéma de cette base et indiquez le contenu des tables correspondant aux documents ci-dessus.

Correction

Il s'agit d'une sorte de rétro-ingéniere à partir de documents structurés dont la forme aparaît extrêmement régulière. On trouve, dans chaque document, une description de personnes (étudiants) au premier niveau, avec un ensemble imbriqué (le tableau de UEs).

Ces documents devraient vous rappeler quelque chose : les films et les acteurs, avec les rôles joués par les acteurs. Ici, on a des étudiants (premier type d'entité), des UEs (deuxième type d'entité) et une association entre les deux : les étudiants sont inscrits à des UEs, et obtiennent une note. Le petit exemple donné montre bien qu'un étudiant peut suivre plusieurs UEs, et inversement, on remarque qu'une même UE (la 27) est suivie par plusieurs étudiants.

Conclusion : il s'agit d'une classique assocation plusieurs-à-plusieurs, qui se représente en relationnel avec 3 tables : `Etudiant`, `UE` et `Inscription`. Remarquez bien que la note ne peut être placée ni dans la table `Etudiant` ni dans la table `UE`, mais seulement dans la table `Inscription`.

Tableau 3.6 – Table des étudiants

id	nom
978	Jean Dujardin
476	Vanessa Paradis

Tableau 3.7 – Table des UEs

id	titre
11	Java
13	Méthodologie
27	Bases de données
37	Réseaux
76	Conduite de projets

Il nous faut finalement une table des inscriptions.

Tableau 3.8 – Table des inscriptions

idEtudiant	idUE	note
978	11	12
978	27	17
978	37	14
476	13	17
476	27	10
476	76	11

Et voilà. La représentation relationnelle est entièrement à plat, ce qui a l'avantage de donner une vision parfaitement symétrique, non centrée sur une entité particulière. L'inconvénient est la distribution des données dans plusieurs tables : il faut faire des jointures.

— Proposez une autre représentation des mêmes données, centrée cette fois, non plus sur les étudiants, mais sur les UEs.
Avec les documents structurés, on choisit de privilégier certaines entités, celles qui sont proches de la racine de l'arbre. En centrant sur les UEs, on obtient le même contenu, mais avec une représentation très différente.

Correction

```
{
  "_id": "ue:11",
  "titre": "Java",
  "etudiants": [
                {"id": 978, "nom": "Jean Dujardin", "note": 12}
               ]
}
{
  "_id": "ue:13",
  "titre": "Méthodologie",
  "etudiants": [
          {"id": 476, "nom": "Vanessa Paradis", "note": 17}
         ]
}
{
  "_id": "ue:27",
  "titre": "Java",
  "etudiants": [
         {"id": 978, "nom": "Jean Dujardin", "note": 17},
         {"id": 476, "nom": "Vanessa Paradis", "note": 10}
     ]
}
{
  "_id": "ue:37",
  "titre": "Réseaux",
  "etudiants": [
         {"id": 978, "nom": "Jean Dujardin", "note": 14}
       ]
}
{
   "_id": "ue:76",
   "titre": "Conduite projet",
   "etudiants": [
          {"id": 476, "nom": "Vanessa Paradis", "note": 11}
        ]
}
```

Exercice Ex-S5-5 : passer du relationnel aux documents complexes

Vous trouverez la description d'une base relationnelle dans le chapitre de mon cours sur SQL http://sql.bdpedia.fr/relationnel.html#la-base-des-voyageurs. Elle décrit des voyageurs séjournant dans des logements. Notre but est de transformer cette base en une collection de documents JSON.

— Proposez un document JSON représentant *toutes* les informations disponibles sur un des logements, par exemple *U Pinzutu*. On devrait donc y trouver les activités proposées.

— Proposez un document JSON représentant *toutes* les informations disponibles sur un voyageur, par exemple Phileas Fogg.
— Proposez un schéma JSON pour des documents représentant les logements et leurs activités mais pas les séjours.
— Vérifiez la validité syntaxique et insérez les documents dans MongoDb en effectuant une validation avec le schéma.

Correction

Voici un document JSON représentant un logement. Notez que l'on pourrait aussi ajouter la liste des séjours (ça devient rapidement laborieux).

```
{
   "code":"pi",
    "nom":" U Pinzutu",
    "capacité":10,
    "type":"Gîte",
    "lieu":"Corse",
    "activités":[
        {
        "codeActivité":"Voile",
        "description":"Pratique du dériveur et du catamaran"
        },
        {
        "codeActivité":"Plongée",
        "description":"Baptèmes et préparation des brevets"
        }
    ]
}
```

Le schéma pour ce type de document est le suivant (on peut ajouter toutes sortes de contraintes, descriptions, etc.)

```
{
   "bsonType":"object",
    "required":["code","nom","capacité","lieu"],
    "properties":{
        "code":{ "bsonType":"string"},
        "nom":{"bsonType":"string"},
        "capacité":{"bsonType":"int"},
        "type":{"enum":["Gîte","Hôtel","Auberge"]},
        "lieu":{ "bsonType":"string"},
         "activités": {
            "bsonType":"array",
            "items": {
                "bsontype": "object",
                "required":[ "codeActivité"],
                "properties":{
                    "codeActivité":{"bsonType":"string"}
```

(suite sur la page suivante)

(suite de la page précédente)

```
                }
            }
        }
    }
}
```

3.3.1 Pour aller plus loin (optionnel)

Exercice Ex-S5-1 : des schémas pour valider les documents JSON

Il est facile de transformer MongoDB en une poubelle de données en insérant n'importe quel document. Depuis la version 3.2, MongoDB offre la possibilité d'associer un schéma à une collection et de contrôler que les documents insérés sont conformes au schéma.

La documentation est ici : https://docs.mongodb.com/manual/core/schema-validation

À vous de jouer : définissez le schéma de la collection des films, et appliquez la validation au moment de l'insertion. Vous pouvez commencer avec une collection simple, celle des artistes, pour vous familiariser avec cette notion de schéma.

Exercice Ex-S3-2 : modélisation d'une base Cassandra

Maintenant, vous allez modéliser une base Cassandra pour stocker les informations sur le métro parisien. Voici deux fichiers JSON :

— http://b3d.bdpedia.fr/files/metro-lines.json, les lignes de métro
— http://b3d.bdpedia.fr/files/metro-stops.json, tous les arrêts de métro

Proposez un modèle Cassandra, créez la ou les table(s) nécessaires, essayez d'insérer quelques données, voire toutes les données (ce qui suppose d'écrire un petit programme pour les mettre au bon format).

Correction

```
CREATE KEYSPACE IF NOT EXISTS Metros
 WITH REPLICATION = { 'class' : 'SimpleStrategy', 'replication_factor': 3 };
```

```
create table lines(color text, name text, number text,
                route_name text, primary key(color));

create type line(line text, position int);

create table stops(
         description text,
```

(suite sur la page suivante)

(suite de la page précédente)

```
        latitude float,
        lines set<frozen<line>>,
        longitude float,
        name text,
        primary key(description)
    );
```

```
insert into lines JSON '
      {
        "color": "#F58F53",
        "name": "Tramway 3A",
        "number": "3A",
        "route_name": "PONT GARIGLIANO - HOP G.POMPIDOU <-> PORTE DE VINCENNES"
      }';


  insert into stops json '
    {
    "description": "Jean Jaurès (23 boulevard) - 92012",
    "latitude": 48.84228,
    "lines": [{"line": "ligne-10", "position": 2}],
                "longitude": 2.238863,
                 "name": "Boulogne-Jean-Jaurès"
      }
';
```

CHAPITRE 4

Recherche exacte

Dans ce chapitre nous étudions quelques méthodes de *recherche exacte*, au sens de « recherche dont le contenu est défini de manière complète et univoque par la requête ». Cette notion de recherche exacte s'oppose à celle de recherche *approchée* que nous étudierons plus tard.

Etant donnée une recherche exacte *q1* et un document *d*, on peut dire si *d* appartient ou non au résultat de *q1*. À l'inverse, si *q2* est une recherche approchée, l'appartenance de *d* au résultat est plus ou moins forte. Conséquence, : tous les documents dans le résultat d'une recherche exacte sont aussi pertinents les uns que les autres, alors que pour une recherche exacte, on doit les classer par ordre de pertinence.

Nous commençons par la recherche basée sur les protocoles du Web. Ce dernier n'est pas vraiment une base de données mais c'est un système distribué de documents, et un cas-type de *Big Data* s'il en est. De plus, il s'agit d'une source d'information essentielle pour collecter des données, les agréger et les analyser.

Le Web s'appuie sur des protocoles bien connus (HTTP) qui ont été repris pour la définition de services (Web) dits REST. Nous utiliserons CouchDB pour illustrer l'organisation et la manipulation de documents basées sur REST.

Nous continuons ensuite notre exploration avec MongoDB et ElasticSearch. Le cas de Cassandra est étudié dans un chapitre à part, pour montrer comment la modélisation peut être influencée par les capacités du langage d'interrogation.

4.1 S1 : HTTP, REST, et CouchDB

Supports complémentaires
- Présentation: le Web, REST, illustration avec CouchDB
- Vidéo de la session REST + CouchDB

Le Web est la plus grande base de documents ayant jamais existé ! Même s'il est essentiellement constitué de documents très peu structurés et donc difficilement exploitables par une application informatique, les méthodes utilisées sont très instructives et se retrouvent dans des systèmes plus organisés. Dans cette section, l'accent est mis sur le protocole REST que nous retrouverons très fréquemment en étudiant les systèmes NoSQL. Les systèmes CouchDB et ElasticSearch qui s'appuient sur REST.

4.1.1 Web = ressources + URL + HTTP

Rappelons les principales caractéristiques du Web, vu comme un gigantesque système d'information orienté documents. Distinguons d'abord l'Internet, réseau connectant des machines, et le Web qui constitue une collection distribuée de *ressources* hébergés sur ces machines.

Le Web est (techniquement) essentiellement caractérisé par trois choses : la notion de *ressource*, l'adressage par des *URL* et le protocole HTTP.

Ressources

La notion de ressource est assez générale/générique. Elle désigne toute entité disposant d'une adresse sur le réseau, et fournissant des services. Un document est une forme de ressource : le service est dans ce cas le contenu du document lui-même. D'autres ressources fournissent des services au sens plus calculatoire du terme en effectuant sur demande des opérations produisant la représentation du résultat sous forme de ressource.

Il faut essentiellement voir une ressource comme un point adressable sur l'Internet avec lequel on peut échanger des messages. L'adressage se fait par une URL, l'échange par HTTP.

URLs

L'adresse d'une ressource est une URL, pour *Universal Resource Location*. C'est une chaîne de caractères qui encode toute l'information nécessaire pour trouver la ressource et lui envoyer des messages.

Note : Certaines ressources n'ont pas d'adresse sur le réseau, mais sont quand même identifiables de manière pérenne et unique par des URI (*Universal Resource identifier*).

Cet encodage prend la forme d'une chaîne de caractères formée selon des règles précises illustrées par l'URL fictive suivante :

```
https://www.example.com:443/chemin/vers/doc?nom=b3d&type=json#fragment
```

Ici, `https` est le *protocole* qui indique la méthode d'accès à la ressource. Le seul protocole que nous verrons est HTTP (le `s` indique une variante de HTTP comprenant un encryptage des échanges). *L'hôte* (*hostname*) est `www.example.com`. Un des services du Web (le DNS) va convertir ce nom d'hôte en adresse IP, ce qui permettra d'identifier la machine serveur qui héberge la ressource.

Note : Quand on développe une application, on la teste souvent localement en utilisant sa propre machine de développement comme serveur. Le nom de l'hôte est alors `localhost`, qui correspond à l'IP `127.0.0.1`.

La machine serveur communique avec le réseau sur un ensemble de *ports*, chacun correspondant à l'un des services gérés par le serveur. Pour le service HTTP, le port est par défaut 80, mais on peut le préciser, comme sur l'exemple précédent, où il vaut 443. On trouve ensuite le *chemin d'accès à la ressource*, qui suit la syntaxe d'un chemin d'accès à un fichier dans un système de fichiers. Dans les sites simples, « statiques », ce chemin correspond de fait à un emplacement physique vers le fichier contenant la ressource. Dans des applications plus sophistiquées, les chemins sont virtuels et conçus pour refléter l'organisation logique des ressources offertes par l'application.

Après le point d'interrogation, on trouve la liste des paramètres (*query string*) éventuellement transmis à la ressource. Enfin, le fragment désigne une sous-partie du contenu de la ressource. Ces éléments sont optionnels.

Le protocole HTTP

HTTP, pour *HyperText Transfer Protocol*, est un protocole extrêmement simple, basé sur TCP/IP, initialement conçu pour échanger des documents hypertextes. HTTP définit le format des requêtes et des réponses. Voici par exemple une requête envoyée à un serveur Web :

```
GET /myResource HTTP/1.1
Host: www.example.com
```

Elle demande une ressource nommée `myResource` au serveur `www.example.com`. Voici une possible réponse à cette requête :

```
HTTP/1.1 200 OK
Content-Type: text/html; charset=UTF-8

<html>
  <head><title>myResource</title></head>
  <body><p>Bonjour à tous!</p></body>
</html>
```

Un message HTTP est constitué de deux parties : l'entête et le corps, séparées par une ligne blanche. La réponse montre que l'entête contient des informations qualifiant le message. La première ligne par exemple indique qu'il s'agit d'un message codé selon la norme 1.1 de HTTP, et que le serveur a pu correctement

répondre à la requête (code de retour 200). La seconde ligne de l'entête indique que le corps du message est un document HTML encodé en UTF-8.

Le programme client qui reçoit cette réponse traite le corps du message en fonction des informations contenues dans l'entête. Si le code HTTP est 200 par exemple, il procède à l'affichage. Un code 404 indique une ressource manquante, une code 500 indique une erreur sévère au niveau du serveur. Voir http://en.wikipedia.org/wiki/List_of_HTTP_status_codes pour une liste complète.

Le Web est initialement conçu comme un système d'échange de documents hypertextes se référençant les uns les autres, codés avec le langage HTML (ou XHTML dans le meilleur des cas). Ces documents s'affichent dans des navigateurs et sont donc conçus pour des utilisateurs humains. En revanche, ils sont très difficiles à traiter par des applications en raison de leur manque de structure et de l'abondance d'instructions relatives à *l'affichage* et pas à la description du *contenu*. Examinez une page HTML provenant de n'importe quel site un peu soigné et vous verrez que la part du contenu est négligeable par rapport à tous les CSS, javascript, images et instructions de mise en forme.

Une évolution importante du Web a donc consisté à étendre la notion de ressource à des *services* recevant et émettant des documents structurés transmis dans le corps du message HTTP. Vous connaissez déjà les formats utilisés pour représenter cette structure : JSON et XML, ce dernier étant clairement de moins en moins apprécié.

C'est cet aspect sur lequel nous allons nous concentrer : le Web des services est véritablement une forme de très grande base de documents structurés, présentant quelques fonctionnalités (rudimentaires) de gestion de données comparable aux opérations d'un SGBD classique. Les services basés sur l'architecture REST, présentée ci-dessous, sont la forme la plus courante rencontrée dans ce contexte.

4.1.2 L'architecture REST

REST est une forme de service Web dont le parti pris est de s'appuyer sur HTTP, ses opérations, la notion de ressource et l'adressage par URL. REST est donc très proche du Web, la principale distinction étant que REST est orienté vers l'appel à des services à base d'échanges par documents structurés, et se prête donc à des échanges de données entre applications. La Fig. 4.1 donne une vision des éléments essentiels d'une architecture REST.

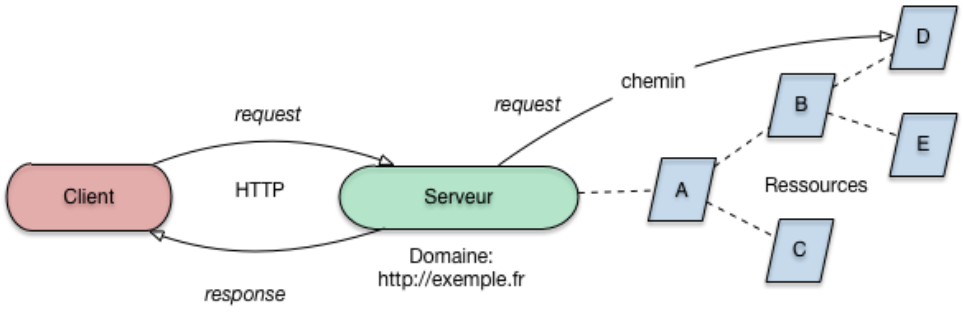


Fig. 4.1 – Architecture REST : client, serveur, ressources, URL (domaine + chemin)

Avec HTTP, il est possible d'envoyer quatre principaux types de messages, ou *méthodes*, à une ressource web :

— `GET` est une *lecture* de la ressource (ou plus précisément de sa représentation publique) ;

— `PUT` est la *création* d'une ressource ;
— `POST` est l'envoi d'un message à une ressource *existante* ;
— `DELETE` la destruction d'une ressource.

REST s'appuie sur un usage strict (le plus possible) de ces quatre méthodes. Ceux qui ont déjà pratiqué la programmation Web admettront par exemple qu'un développeur ne se pose pas toujours nettement la question, en créant un formulaire, de la méthode `GET` ou `POST` à employer. De plus le `PUT` (qui n'est pas connu des formulaires Web) est ignoré et le `DELETE` jamais utilisé.

La définition d'un service REST se doit d'être plus rigoureuse.

— le `GET`, en tant que lecture, ne doit *jamais* modifier l'état de la ressource (pas « d'effet de bord ») ; autrement dit, en l'absence d'autres opérations, des messages `GET` envoyés répétitivement à une même ressource ramèneront toujours le même document, et n'auront aucun effet sur l'environnement de la ressource ;
— le `PUT` est une création, et l'URL a laquelle un message `PUT` est transmis ne doit pas exister au préalable ; dans une interprétation un peu plus souple, le `PUT` crée ou *remplace* la ressource éventuellement existante par la nouvelle ressource transmise par le message ;
— inversement, `POST` doit s'adresser à une ressource existante associée à l'URL désignée par le message ; cette méthode correspond à l'envoi d'un message à la ressource (vue comme un service) pour exécuter une action, avec potentiellement un changement d'état (par exemple la création d'une nouvelle ressource).

Les messages sont transmis en HTTP (voir ci-dessus) ce qui offre, entre autres avantages, de ne pas avoir à redéfinir un nouveau protocole. Le contenu du message est une information codée en XML ou en JSON (le plus souvent), soit ce que nous avons appelé jusqu'à présent un *document*.

— quand le client émet une requête REST, le document contient les paramètres d'accès au service (par exemple les valeurs de la ressource à créer, ou bien le code de la requête à effectuer) ;
— quand la ressource répond au client, le document contient l'information constituant le résultat du service ; en cas d'erreur ou d'anomalie (droit d'accès insuffisant par exemple) un code d'erreur HTTP peut être utilisé.

Important : En toute rigueur, il faut bien distinguer la ressource et le document qui représente une information produite par la ressource.

On peut faire appel à un service REST avec n'importe quel client HTTP, et notamment avec votre navigateur préféré : copiez l'URL dans la fenêtre de navigation et consultez le résultat. Le navigateur a cependant l'inconvénient, avec cette méthode, de ne transmettre que des messages `GET`. Un outil plus général, s'utilisant en ligne de commande, est cURL. S'il n'est pas déjà installé dans votre environnement, il est fortement conseillé de le faire dès maintenant : le site de référence est http://curl.haxx.se/.

Voici quelques exemples d'utilisation de cURL pour parler le HTTP avec un service REST. Notre base de films est obtenue par appel à l'API REST de *themoviedb.org*. Voici la requête qui cherche le film `tt10404944`.

```
https://api.themoviedb.org/3/find/tt10404944?api_
↪key=8f479aa1cb024ffe0d95da8c4ee56fe7&external_source=imdb_id
```

Ce qui retourne le document suivant.

```
{
"adult": false,
```

(suite sur la page suivante)

(suite de la page précédente)

```
"backdrop_path": "/vkIJ2QgcKMJRvi6pBW4Tr2kgLdy.jpg",
"id": 637534,
"title": "The Stronghold",
"original_language": "fr",
"original_title": "BAC Nord",
"overview": "A police brigade works in the dangerous northern neighborhoods of␣
↪Marseille, where the level of crime is higher than anywhere else in France.",
"poster_path": "/nLanxl7Xhfbd5s8FxPy8jWZw4rv.jpg",
"media_type": "movie",
"genre_ids": [53, 28, 80],
"popularity": 21.929,
"release_date": "2021-08-18",
"video": false,
"vote_average": 7.426,
"vote_count": 1147
}
```

Notez le placement de paramètres dans l'URL, et notamment une clé d'accès, souvent requise pour utiliser des services. Autre exemple, l'API REST de *Open Weather Map*, un service fournissant des informations météorologiques (créez une clé API et ajoutez-la à l'URL pour obtenir une réponse complète).

```
curl -X GET api.openweathermap.org/data/2.5/weather?q=Paris
```

Et on obtient la réponse suivante (qui varie en fonction de la météo, évidemment).

```
{
   "coord":{
     "lon":2.35,
     "lat":48.85
   },
   "weather":[
     {
      "id":800,
      "main":"Clear",
      "description":"Sky is Clear",
      "icon":"01d"
     }
   ],
   "base":"cmc stations",
   "main":{
    "temp":271.139,
    "temp_min":271.139,
    "temp_max":271.139,
    "pressure":1021.17,
    "sea_level":1034.14,
    "grnd_level":1021.17,
```

(suite sur la page suivante)

(suite de la page précédente)

```
    "humidity":87
   },
   "name":"Paris"
}
```

Même chose, mais en demandant une réponse codée en XML. Notez l'option `-v` qui permet d'afficher le détail des échanges de messages HTTP gérés par cURL.

```
curl -X GET -v api.openweathermap.org/data/2.5/weather?q=Paris&mode=xml
```

Nous verrons ultérieurement des exemples de `PUT` et de `POST` pour créer des ressources et leur envoyer des messages avec cURL.

Note : La méthode `GET` est utilisée par défaut par cURL, on peut donc l'omettre.

Nous nous en tenons là pour les principes essentiels de REST, qu'il faudrait compléter de nombreux détails mais qui nous suffiront à comprendre les interfaces (ou API) REST que nous allons rencontrer.

Important : Les méthodes d'accès aux documents sont représentatives des opérations de type *dictionnaire* : toutes les données ont une adresse, on peut accéder à la donnée par son adresse (`get`), insérer une donnée à une adresse (`put`), détruire la donnée à une adresse (`delete`). De nombreux systèmes NoSQL se contentent de ces opérations qui peuvent s'implanter très simplement et efficacement.

Pour être concret et rentrer au plus vite dans le cœur du sujet, nous présentons l'API de CouchDB qui est conçu comme un serveur de documents (JSON) basé sur REST.

4.1.3 L'API REST de CouchDB

CouchDB est essentiellement un serveur Web étendu à la gestion de documents JSON. Comme tout serveur Web, il parle le HTTP et manipule des ressources (Fig. 4.2).

Je vous revoie au chapitre *Préliminaires : Docker* pour l'installation de CouchDB, le chargement d'une base et l'interaction avec le serveur, soit via cUrl, soit via l'interface graphique disponible à http://admin:admin@localhost:5984/_utils

CouchDB adopte délibérément les principes et protocoles du Web. Une base de données et ses documents sont vus comme des *ressources* et on dialogue avec eux en HTTP, conformément au protocole REST.

Un serveur CouchDB gère un ensemble de bases de données. Créer une nouvelle base se traduit, avec l'interface REST, par la création d'une nouvelle ressource. Voici donc la commande avec cURL pour créer une base `films` (notez la méthode `PUT` pour créer une ressource).

```
curl -X PUT http://admin:admin@localhost:5984/films
{"ok":true}
```

Maintenant que la ressource est créée, et on peut obtenir sa représentation avec une requête `GET`.

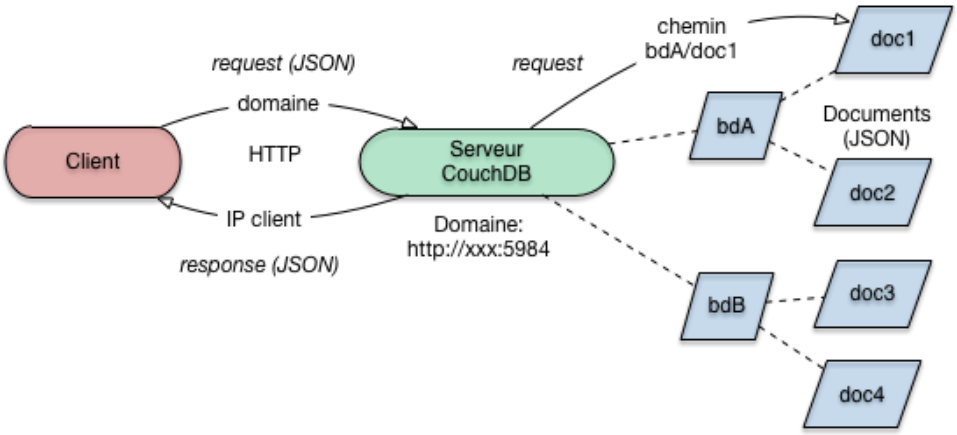


Fig. 4.2 – Architecture (simplifiée) de CouchDB

```
curl -X GET http://admin:admin@localhost:5984/films
```

Cette requête renvoie un document JSON décrivant la nouvelle base.

```
{"update_seq":"0-g1AAAADfeJz6t",
 "db_name":"films",
 "sizes": {"file":17028,"external":0,"active":0},
 "purge_seq":0,
 "other":{"data_size":0},
 "doc_del_count":0,
 "doc_count":0,
 "disk_size":17028,
 "disk_format_version":6,
 "compact_running":false,
 "instance_start_time":"0"
}
```

Vous commencez sans doute à saisir la logique des interactions. Les entités gérées par le serveur (des bases de données, des documents, voire des fonctions) sont transposées sous forme de ressources Web auxquelles sont associées des URLs correspondant, autant que possible, à l'organisation logique de ces entités.

Pour insérer un nouveau document dans la base `films`, on envoie donc un message `PUT` à l'URL qui va représenter le document. Cette URL est de la forme `http://localhost:5984/films/idDoc`, où `idDoc` désigne l'identifiant de la nouvelle ressource.

```
curl -X PUT http://admin:admin@localhost:5984/films/doc1 -d '{"key": "value"}'

  {"ok":true,"id":"doc1","rev":"1-25eca"}
```

Que faire si on veut insérer des documents placés dans des fichiers ? Vous avez dû récupérer dans nos jeux de données des documents représentant des films en JSON. Le document `film629.json` par exemple re-

présente le film *Usual Suspects*. Voici comment on l'insère dans la base en lui attribuant l'identifiant `us`.

```
curl -X PUT http://admin:admin@localhost:5984/films/us -d @film629.json -H
↪"Content-Type: application/json"
```

Cette commande cURL est un peu plus compliquée car il faut créer un message HTTP plus complexe que pour une simple lecture. On passe dans le corps du message HTTP le contenu du fichier `film629.json` avec l'option `-d` et le préfixe `@`, et on indique que le format du message est JSON avec l'option `-H`. Voici la réponse de CouchDB.

```
{
 "ok":true,
 "id":"us",
 "rev":"1-68d58b7e3904f702a75e0538d1c3015d"
}
```

Le nouveau document a un identifiant (celui que nous attribué par l'URL de la ressource) et un numéro de *révision*. L'identifiant doit être unique (pour une même base), et le numéro de révision indique le nombre de modifications apportées à la ressource depuis sa création.

Si vous essayez d'exécuter une seconde fois la création de la ressource, CouchDB proteste et renvoie un message d'erreur :

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

Un document existe déjà à cette URL.

Une autre façon d'insérer, intéressante pour illustrer les principes d'une API REST, et d'envoyer non pas un `PUT` pour créer une nouvelle ressource (ce qui impose de choisir l'identifiant) mais un `POST` à une ressource existante, ici la base de données, qui se charge alors de créer la nouvelle ressource représentant le film, et de lui attribuer un identifiant.

Voici cette seconde option à l'œuvre pour créer un nouveau document en déléguant la charge de la création à la ressource `films`.

```
curl -X POST http://admin:admin@localhost:5984/films -d @film629.json -H
↪"Content-Type: application/json"
```

Voici la réponse de CouchDB :

```
{
 "ok":true,
 "id":"movie:629",
 "rev":"1-68d58b7e3904f702a75e0538d1c3015d"
}
```

CouchDB a trouvé l'identifiant (conventionnellement nommé `id`) dans le document JSON et l'utilise. Si aucun identifiant n'est trouvé, une valeur arbitraire (une longue et obscure chaîne de caractère) est engendrée.

CouchDB est un système multi-versions : une nouvelle version du document est créée à chaque insertion. Chaque document est donc identifié par une paire (id, revision) : notez l'attribut `rev` dans le document ci-dessus. Dans certains cas, la version la plus récente est implicitement concernée par une requête. C'est le cas quand on veut obtenir la ressource avec un simple `GET`.

```
curl -X GET http://admin:admin@localhost:5984/films/us
```

En revanche, pour supprimer un document, il faut indiquer explicitement quelle est la version à détruire en précisant le numéro de révision.

```
curl -X DELETE http://admin:admin@localhost:5984/films/us?rev=1-
↪68d58b7e3904f702a75e0538d1c3015d
```

Nous en restons là pour l'instant. Cette courte session illustre assez bien l'utilisation d'une API REST pour gérer une collection de document à l'aide de quelques opérations basiques : création, recherche, mise à jour, et destruction, implantées par l'une des 4 méthodes HTTP. La notion de ressource, existante (et on lui envoie des messages avec `GET` ou `POST`) ou à créer (avec un message `PUT`), associée à une URL correspondant à la logique de l'organisation des données, est aussi à retenir.

Bien entendu il s'agit d'un langage très limité pour l'interrogation, puisqu'il n'offre que deux possibilités : accéder à un document si on connaît son identifiant, ou parcourir toute la collection. Nous verrons avec ElasticSearch une approche beaucoup plus puissante où REST est utilisé pour transmettre des requêtes dans un langage riche et complexe.

4.1.4 Quiz

4.1.5 Mise en pratique

Les propositions suivantes vous permettent de mettre en pratique les connaissances précédentes.

MEP MEP-S1-2 : Documents et services Web

Soyons concret : vous construisez une application qui, pour une raison X ou Y, a besoin de données météo sur une région ou une ville donnée. Comment faire ? la réponse est simple : trouver le service Web qui fournit ces données, et appeler ces services. Pour la première question, on peut par exemple prendrele site OpenweatherMap, dont les services sont décrits ici : http://openweathermap.org/api. Pour appeler ce service, comme vous pouvez le deviner, on passe par le protocole HTTP.

Application : utilisez les services de OpenWeatherMap pour récupérer les données météo pour Paris, Marseille, Lyon, ou toute ville de votre choix. Testez les formats JSON et XML.

MEP MEP-S1-3 : comprendre les services géographiques de Google.

Google fournit (librement, jusqu'à une certaine limite) des services dits de géolocalisation : récupérer une carte, calculer un itinéraire, etc. Vous devriez être en mesure de comprendre les explications don-

nées ici : https://developers.google.com/maps/documentation/webservices/?hl=FR (regardez en particulier les instructions pour traiter les documents JSON et XML retournés).

Pour vérifier que vous avez bien compris : créez un formulaire HTML avec deux champs dans lesquels on peut saisir les points de départ et d'arrivée d'un itinéraire (par exemple, Paris - Lyon). Quand on valide ce formulaire, afficher le JSON ou XML (mieux, donner le choix du format à l'utilisateur) retourné par le service Google (aide : il s'agit du service `directions`).

MEP MEP-S1-4 : explorer les services Web et l'Open data.

Le Web est une source immense de données structurées représentées en JSON ou en XML. Vous voulez connaître le programme d'une station de radio, accéder à une entre Wikipedia sous forme structurée, gérer des calendriers ? On trouve à peu près tout sous forme de services.

Explorez ces API pour commencer à vous faire une idée du type de projet qui vous intéresse. Récupérez des données et regardez leur format.

Autre source de données : les données publiques. Allez voir par exemple sur
- https://www.data.gouv.fr/fr/
- http://data.iledefrance.fr/page/accueil/
- http ://data.enseignementsup-recherche.gouv.fr/
- http://www.data.gov/ (Open data USA)
- La société OpendatSoft (http://www.opendatasoft.fr) propose non seulement des jeux de données, mais de nombreux outils d'analyse et de visualisation.

Essayer d'imaginer une application qui combine plusieurs sources de données.

4.2 S2 : ElasticSearch

Supports complémentaires
- Introduction à l'interrogation ElastiSearch
- Vidéo de la session Bases documentaires et moteur de recherche
- Vidéo de la session requêtes booléennes

Dans cette section, nous allons passer au concret en introduisant les moteurs de recherche. Nous allons utiliser ici Elastic Search. Nous indexerons nos premiers documents, et commencerons à faire nos premières requêtes.

Nous allons nous appuyer entièrement sur les choix par défaut d'ElasticSearch pour nous concentrer sur son utilisation. La construction d'un moteur de recherche en production demande un peu plus de soin, nous en verrons au chapitre suivant les étapes nécessaires.

4.2.1 Architecture du système d'information avec un moteur de recherche

Un moteur de recherche comme ElasticSearch est une application spécialisée dans la recherche efficace appliquée à des collections de documents, ces collections étant en général stockée avec un autre système NoSQL. Pourquoi ne pas utiliser directement le moteur de recherche comme gestionnaire des documents ? La réponse est qu'un système comme ElasticSearch est entièrement consacré à la recherche (donc à la *lecture*) la plus efficace possible de documents. Il s'appuie pour cela sur des structures compactes, compressées, optimisées (les index inversés). En revanche, ce n'est pas nécessairement un très bon outil pour les autres fonctionnalités d'une base de données. Le stockage par exemple n'est ni aussi robuste ni aussi stable, et il faut parfois *reconstruire* l'index à partir de la base originale (on parlera de *réindexer les documents*).

Un système comme ElasticSearch n'est pas non plus très peformant pour des données souvent modifiées. Pour des raisons qui tiennent à la structure de ces index, les mises à jour sont coûteuses et s'effectuent difficilement en temps réel. La notion de mise à jour vaut ici aussi bien pour le *contenu* des documents (modification de la valeur d'un champ) que pour leur *structure* (ajout ou suppression d'un champ par exemple).

La pratique la plus courante consiste donc à utiliser un système de recherche comme un *complément* d'un serveur de base de données (relationnelle ou documentaire) et à lui confier les tâches de recherche que le serveur BD ne sait pas accomplir (soit, en gros, les recherches non structurées). Dans le cas des bases NoSQL, l'absence fréquente de tout langage de requête fait du moteur de recherche associé un outil indispensable.

Même en cas de présence d'un langage d'interrogation fourni par le système NoSQL, le moteur de recherche reste un candidat tout à fait valide pour satisfaire les *recherches approchées* avec classement des documents. En résumé, à part les deux inconvénients (reconstruction depuis une source extérieure, support faible des mises à jour), les moteurs de recherche sont des composants puissants aptes à satisfaire efficacement les besoins d'un système documentaire.

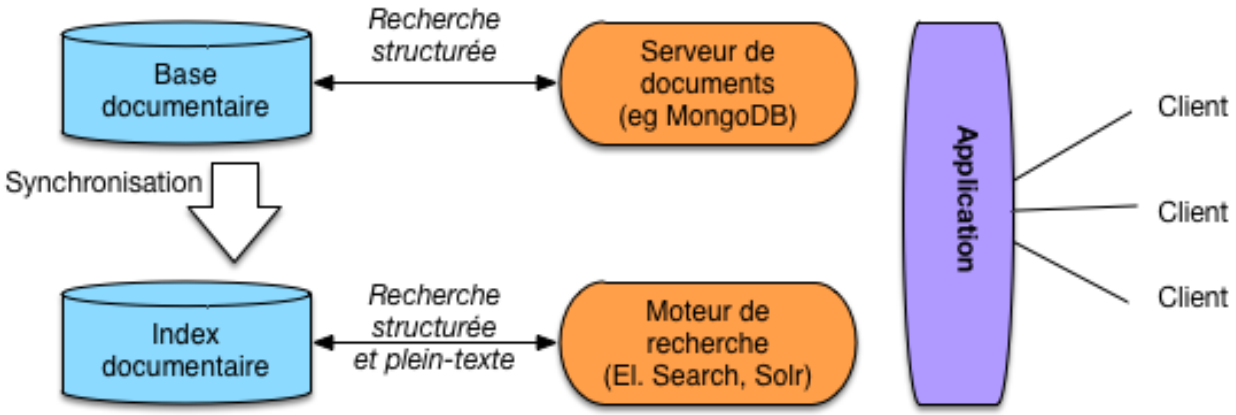


Fig. 4.3 – Architecture d'une application avec moteur de recherche.

La Fig. 4.3 montre une architecture typique, en prenant pour exemple une base de données MongoDB (mais cela s'applique à tout autre SGBD). Les *documents (applicatifs)* sont donc dans la base qui fournit (pas toujours) des fonctionnalités de recherche structurées. On peut indexer la collection des documents applicatifs en extrayant des « champs » formant des *documents (au sens d'ElasticSearch)* fournis à l'index qui se charge de les organiser pour satisfaire efficacement des requêtes. L'application peut alors soit s'adresser au serveur MongoDB, soit au moteur de recherche.

Un scénario typique est celui d'une recherche par mot-clé dans un site, scénario que nous appelons *recherche*

approchée et qui fait l'objet d'un prochain chapitre. En attendant voici une présentation limitée aux recherches exactes.

4.2.2 Interrogation

Une première méthode pour transmettre des recherches est de passer une expression en paramètre à l'URL à laquelle répond votre serveur ElasticSearch. La forme la plus simple d'expression est une liste de mots-clés. Voici quelques exemples d'URLs de recherche :

```
https://localhost:9200/nfe204/_search?q=alien
https://localhost:9200/nfe204/_search?q=alien,coppola
https://localhost:9200/nfe204/_search?q=alien,coppola,1994
```

Les dernières versions (à partir de la 8) d'ElasticSearch ont introduit des mesures de sécurité qui compliquent fortement l'accès direct au serveur en HTTPS. Mieux vaut donc utiliser ElasticVue avec la fenêtre `SEARCH` qui se charge se constituer l'URL de requête.

Une seconde méthode est de transmettre un document JSON décrivant la recherche. L'envoi d'un document suppose que l'on utilise la méthode `POST`. Voici par exemple un document avec une recherche sur trois mots-clé.

```
{
 "query": {
    "query_string" : {
       "query" : "alien,coppola,1994"
    }
  }
}
```

La Fig. 4.4 montre l'exécution avec l'interface ElasticVue.

On voit clairement (mais partiellement) le résultat, produit sous la forme d'un document JSON énumérant les documents trouvés dans un tableau `hits`. Notez que le document indexé lui-même est présent, dans le champ `_source`, correspondant à un comportement par défaut d'ElasticSearch : *la totalité des documents transmis à ElasticSearch y sont conservés sous leur forme « brute »*. La question de l'utilisation de *deux* systèmes qui semblent partiellement redondants se pose. Nous revenons sur cette question plus loin.

Exprimer une recherche revient donc à envoyer à ElasticSearch (utiliser la méthode `POST`) un document encodant la requête. Le langage de recherche proposé par ElasticSearch, dit « DSL » pour *Domain Specific Language*, est très riche. Pour vous donner juste un exemple, voici comme on prend les 5 premiers documents d'une requête, en excluant la source du résultat.

```
{
 "from": 0,
 "size": 5,
 "_source": false,
 "query": {
    "query_string" : {
```

(suite sur la page suivante)

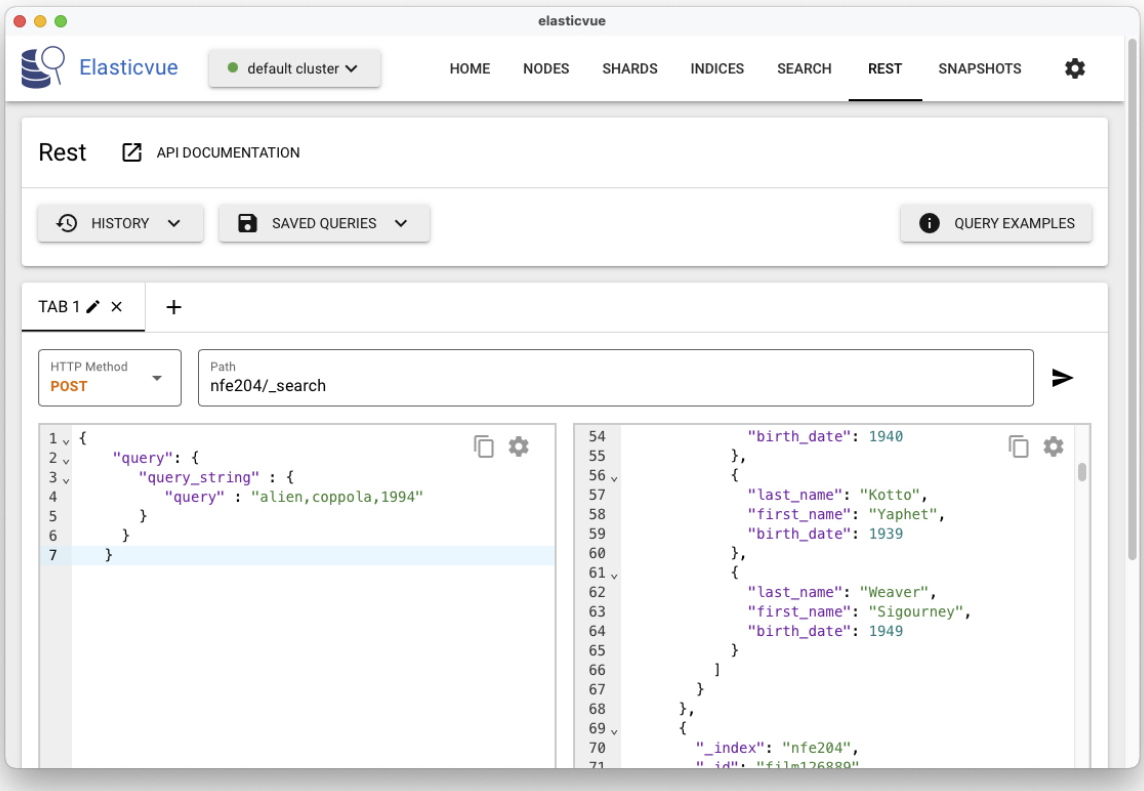


Fig. 4.4 – L'interface ElasticVue avec recherches structurées

(suite de la page précédente)

```
        "query" : "matrix,2000,jamais"
    }
  }
}
```

Nous allons pour l'instant nous contenter d'une variante du language, dite *Query String*, qui correspond, essentiellement, au langage de base du système d'indexation sous-jacent, Lucene (https://lucene.apache.org/). Toutes les expressions données ci-dessous peuvent être entrées comme valeur du champ `query` dans le document-recherche passé à l'interface `REST`.

Termes

La notion de base est celle de *terme*. Un terme est soit un mot, soit une séquence de mots (une *phrase*) placée entre apostrophes. La recherche :

```
Princess Leia
```

retourne tous les documents contenant soit « Princess », soit « Leia ». La recherche

```
"Princess Leia"
```

ramène les documents contenant les deux mots côte à côte (vous devez utiliser \ » pour intégrer un guillemet double dans une requête).

```
{
 "query": {
    "query_string" : {
        "query" : "\"Princess Leia\""
    }
  }
}
```

Par défaut, la recherche s'effectue toujours sur tous les champs d'un document indexé (ou , plus précisément, sur un champ `_all` dans lequel ElasticSearch concatène toutes les chaînes de caractères). La syntaxe complète pour associer le champ et le terme est :

```
champ:terme
```

Par exemple, pour ne chercher le mot-clé `Alien` que dans les titres des films, on peut utiliser la syntaxe suivante :

```
{
 "query": {
    "query_string" : {
        "query" : "title:alien"
    }
```

(suite sur la page suivante)

(suite de la page précédente)

```
  }
}
```

Revenez au fichier JSON et à la structure de ses documents pour voir que les données de chaque film sont imbriquées sous un champ `fields`. Nous l'omettons dans la suite, pensez à l'ajouter.

Si on ne précise pas le champ, c'est celui par défaut qui est pris en compte. Les requêtes précédentes sont donc équivalentes à :

```
_all:"Princess Leia"
```

Les valeurs des termes (dans la requête) et le texte indexé sont tous deux soumis à des transformations que nous étudierons dans le chapitre suivant. Une transformation simple est de tout transcrire en minuscules. La requête :

```
_all:"PRINCESS LEIA"
```

devrait donc donner le même résultat, les majuscules étant converties en minuscules. La conception d'un index doit soigneusement indiquer les transformations à appliquer, car elles déterminent le résultat des recherches.

On peut spécifier un terme simple (pas une phrase) de manière incomplète

- le " ?" indique un caractère inconnue : `opti?al` désigne `optimal`, `optical`, etc.
- le "*" indique n'importe quelle séquence de caractères (`opti*` pour toute chaîne commençant par `opti`).

La valeur d'un terme peut-être indiquée de manière approximative en ajoutant le suffixe "-", si l'on n'est pas sûr de l'orthographe par exemple. Essayez de rechercher `optimal`, puis `optimal-`. La proximité des termes est établie par une distance dite « distance d'édition » (nombre d'opérations d'éditions permettant de passer d'une valeur - optimal - à une autre - optical).

Des recherches *par intervalle* sont possibles. Les crochets [] expriment des intervalles *bornes comprises*, les accolades {} des intervalles bornes non comprises. Voici comment on recherche tous les documents pour une année comprise entre 1990 et 2005 :

```
year:[1990 TO 2005]
```

Connecteurs booléens

Les critères de recherche peuvent être combinés avec les connecteurs Booléens `AND`, `OR` et `NOT`. Quelques exemples.

```
year:[1990 TO 2005] OR title:M*
year:[1990 TO 2005] AND NOT title:M*
```

Important : Attention à bien utiliser des majuscules pour les connecteurs Booléens.

Par défaut, un OR est appliqué, de sorte qu'une recherche sur plusieurs critères ramène l'union des résultats sur chaque critère pris individuellement.

Venons-en maintenant à l'opérateur « + ». Utilisé comme préfixe d'un nom de champ, il indique que la valeur du champ *doit* être égale au terme. La recherche suivante :

```
+year:2000  title:matrix
```

recherche les documents dont l'année est 2000 (obligatoire) **ou** dont le titre est `matrix` ou n'importe quel titre.

Quelle est alors la différence avec `+year:2000` ? La réponse tient dans le *classement* effectué par le moteur de recherche : les documents dont le titre est `matrix` seront mieux classés que les autres. C'est une illustration, parmi d'autres, de la différence entre « recherche d'information » et « interrogation de bases de données ». Dans le premier cas, on cherche les documents les plus « proches », les plus « pertinents », et on classe par pertinence.

Requêtes structurées

Nous pouvons donc transmettre des requêtes dans ElasticVue avec l'interface SEARCH, mais également en POST-ant un document JSON décrivant la requête.

Par exemple, votre première requête consiste à trouver les films « Star Wars » de la base. Le document JSON à POST-er est le suivant :

```
{
    "query": {
        "match": {
            "title": "Star Wars"
        }
    }
}
```

Elle est équivalente à la requête simplifiée, « à la Google », `_search?q=title:Star+Wars`. Le résultat consiste en un ensemble de documents JSON contenant le type, l'identifiant interne, le score et la *source*, cette dernière étant le document transmis pour indexation, avant transformations.

On peut demander à ne pas obtenir toute la source (option `'_source': false`), en sélectionnant en revanche certains champs qui nous intéressent particulièrement (l'équivalent donc du `select` en SQL). On ajoute pour cela un champ `fields`, ce qui donne, si on souhaite obtenir seulement le titre :

```
{
"query": {
        "match": {
                "title": "Star Wars"
        }
        },
        "fields": ["title"],
```

(suite sur la page suivante)

(suite de la page précédente)

```
        "_source": false
}
```

En utilisant ElasticVue, vous pouvez voir les titres et les scores (Fig. 4.5).

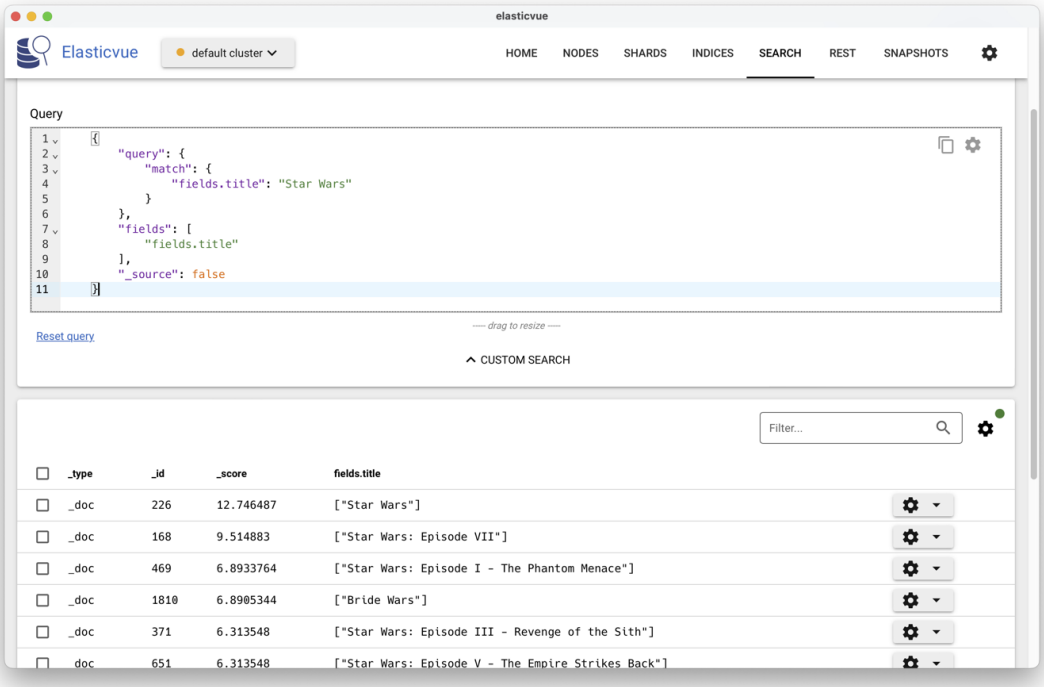


Fig. 4.5 – Affichage des résultats avec les scores.

Nous n'allons pas explorer ici toutes les possibilités du DSL, mais essentiellement les recherches exactes. Les recherches plein-texte seront étudiées ultérieurement On les appelle *Term-level queries* dans la documentation que vous trouverez ici : https://www.elastic.co/guide/en/elasticsearch/reference/current/term-level-queries.html.

Pour effectuer les recherches structurées, on introduit un opérateur dans l'objet `query`, exprimant une condition sur certains champs. On peut par exemple demander les documents dans lesquels un certain champs existe (ici, le champ `summary`) :

```
{
        "query": {
                "exists": {"field": "summary"}
        },
        "fields": ["title"],
        "_source": false
}
```

Important : Peut-on rechercher les documents pour lesquels un champ *n'existe pas* ? Oui bien sûr. Il faut

utiliser pour celà les opérateurs de combinaison booléenne. Voir plus loin.

Nous n'allons présenter que quelques-uns des opérateurs (reportez-vous à la documentation pour une présentation complète). L'opérateur `fuzzy` permet de chercher les mots *proches* (syntaxiquement). Très utile pour ne pas dépendre des fautes d'orthographe ! Recherchons le film *Matrice*.

```
{
  "query": {
     "fuzzy": {
        "title": {
                "value": "matrice"
        }
  }
        },
 "fields": ["title"],
 "_source": false
}
```

On obtient bien un résultat (je vous laisse vérifier). Dans le même esprit on peut faire des recherches par préfixe ou par expression régulière.

Voyons maintenant les recherches par intervalle : la requête suivante retourne tous les films entre 2010 et 2020.

```
{
        "query": {
            "range": {
              "year": {"gte": 2010, "lte": 2020}
        }
        },
        "fields": ["title", "year"],
        "_source": false
}
```

Il existe divers raffinements selon le type de donnée (et notamment pour les dates). Dans l'exemple ci-dessus, on effectue une recherche sur des entiers (l'année) ce qui est simple à exprimer et interpréter.

Qu'en est-il pour les champs de type `text`, les plus courants. Ici, on a deux interprétations possibles : recherche *exacte* et recherche *approchée*, cette dernière impliquant un classement.

La recherche exacte est exprimée avec l'opérateur `term`. Elle s'applique correctement pour des champs texte dont la valeur est codifiée, par exemple le genre du film.

```
{
  "query": {
         "term": {
        "genre": {"value": "drama"}
        }
        },
```

(suite sur la page suivante)

(suite de la page précédente)

```
        "fields": ["title"],
        "_source": false
}
```

Mais, notez que nous utilisons le mot `drama` sans majuscule, alors que dans le document indexé, c'est bien `Drama`. Explication : la recherche exacte porte sur la valeur des champ *après* les transformations décrites dans le chapitre *Recherche approchée* (et notamment, ici, le placement de tous les caractères en minuscules). Si on veut l'appliquer aux champs de type texte, il faut donc « deviner » les transformations effectuées, ce qui n'est pas toujours facile. Essayez par exemple de chercher *Star Wars*.

```
{
  "query": {
      "term": {
                "title": {"value": "star wars"}
                }
                },
        "fields": ["title"],
        "_source": false
}
```

Même en utilisant des minuscules on ne trouve rien car un texte contenant plusieurs mots est transformé en vecteur (une recherche avec `star` en revanche trouve bien des films, mais beaucoup plus que désiré) ;

Pour les recherches exactes sur du texte, ElasticSearch n'est pas très adapté, et en tout cas pas l'opérateur `term`. Je vous laisse étudier des opérateurs qui étendent `term` : `terms` et `terms_set`.

4.2.3 Quiz

4.2.4 Mise en pratique ElasticSearch

La documentation complète sur le DSL d'Elasticsearch se trouve en ligne à l'adresse : https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl.html. Vous aurez à vous y reporter dans ce qui suit.

Nous allons maintenant utiliser une base de films plus large que celle que vue précédemment. Récupérez le fichier suivant contenant environ 5000 films, au format JSON : http://b3d.bdpedia.fr/files/big-movies-elastic.json. Même s'il est assez volumineux, il reste possible de le copier/coller dans ElasticVue pour le transmettre à l'URL `_bulk` en mode `POST`. À défaut, la ligne de commande (pas très simple…) devrait fonctionner.

```
curl -XPOST --cacert http_ca.crt  -U elastic:mot_de_passe \
       https://localhost:9200/_bulk/ --data-binary @big-movies-elastic.json
```

Dans l'interface ElasticVue, vous devriez voir apparaître un index appelé *movies* contenant 4850 films.

Les documents ont la structure suivante (notez bien que toutes les données sont dans un champ imbriqué `fields`) :

```
{
  "fields": {
    "directors": [
      "Joseph Gordon-Levitt"
    ],
    "release_date": "2013-01-18T00:00:00Z",
    "rating": 7.4,
    "genres": [
      "Comedy",
      "Drama"
    ],
    "image_url": "http://ia.media-imdb.com/images/M/MVNTQ3OQ@@._V1_SX400_.jpg",
    "plot": "A New Jersey guy dedicated to his family, friends, and church,
     develops unrealistic expectations from watching porn and works to find
     happ iness and intimacy with his potential true love.",
    "title": "Don Jon",
    "rank": 1,
    "running_time_secs": 5400,
    "actors": [
      "Joseph Gordon-Levitt",
      "Scarlett Johansson",
      "Julianne Moore"
    ],
    "year": 2013
  },
  "id": "tt2229499",
  "type": "add"
}
```


Maintenant, proposez des requêtes pour les besoins d'information suivants.

— Films "Star Wars" dont le réalisateur (directors) est "George Lucas" (requête booléenne)

Correction

Dans toutes les solutions vous pouvez ajouter un champ `fields` pour n'afficher que quelques données.

```
{
  "query": {
    "bool": {
      "must": [
        {
          "match": {
            "fields.title": "Star Wars"
          }
        },
        {
```

(suite sur la page suivante)

(suite de la page précédente)

```
          "match": {
            "fields.directors": "George Lucas"
          }
        }
      ]
    }
  }
}
```

Pour une recherche exacte on utilise `match_phrase` :

```
{
  "query": {
    "bool": {
      "should": [
        {
          "match_phrase": {
            "fields.title": "Star Wars"
          }
        },
        {
          "match": {
            "fields.directors": "George Lucas"
          }
        }
      ]
    }
  }
}
```

ou de manière très compacte : `_search?q=fields.title:Star+Wars directors:George+Lucas`

— Films dans lesquels “Harrison Ford” a joué

Correction

```
{
  "query": {
    "match": {
      "fields.actors": "Harrison Ford"
    }
  }
}
```

ou `_search?q=fields.actors:Harrison+Ford`

— Films dans lesquels “Harrison Ford” a joué dont le résumé (plot) contient “Jones”.

Correction

```
{"query":{
  "bool": {
    "must": [
      { "match": { "fields.actors": "Harrison Ford" }},
      { "match": { "fields.plot": "Jones" }}
    ]
}}}
```

```
_search?q=fields.actors=Harrison+Ford fields.plot:Jones
```

— Films dans lesquels “Harrison Ford” a joué dont le résumé (plot) contient “Jones” mais sans le mot “Nazis”

Correction

```
{"query":{
  "bool": {
    "must": [
      { "match": { "fields.actors": "Harrison Ford" }},
      { "match": { "fields.plot": "Jones" }}
    ],
    "must_not" : { "match" : {"fields.plot":"Nazis"}}
}}}
```

```
_search?q=actors=Harrison+Ford plot:Jones -plot:Nazis
```

— Films de “James Cameron” dont le rang devrait être inférieur à 1000 (boolean + range query).

Correction

```
{"query":{
  "bool": {
    "must": [
      { "match": { "fields.directors": "James Cameron" }},
      { "range": { "fields.rank": {"lt":1000 }}}
    ]
}}}
```

— Films de “Quentin Tarantino” dont la note (rating) doit être supérieure à 5, sans être un film d’action ni un drame.

Correction

```
{
```

(suite sur la page suivante)

(suite de la page précédente)

```
  "query": {
    "bool": {
      "must": [
        {
          "match_phrase": {
            "fields.directors": "Quentin Tarantino"
          }
        },
        {
          "range": {
            "fields.rating": {
              "gte": 5
            }
          }
        }
      ],
      "must_not": [
        {
          "match": {
            "fields.genres": "Action"
          }
        },
        {
          "match": {
            "fields.genres": "Drama"
          }
        }
      ]
    }
  }
}
```

— Films de “J.J. Abrams” sortis (released) entre 2010 et 2015

Correction

```
{
  "query": {
    "bool":{
      "must": {"match": {"fields.directors": "J.J. Abrams"}},
      "filter": {
        "range": {
          "fields.release_date": { "from": "2010-01-01", "to": "2015-12-31"}
        }
      }
```

(suite sur la page suivante)

(suite de la page précédente)

```
    }
  }
}
```

Proposez maintenant les requêtes d'agrégation permettant d'obtenir les statistiques suivantes.

Important : Certaines de ces requêtes sont assez difficiles. Ne les faites que si vous êtes très motivés pour maîtriser ElasticSearch. La solution sera publiée.

— Donner la note (rating) moyenne des films.

Correction

```
{"size":0,
"aggs" : {
    "note_moyenne" : {
      "avg" : {"field" : "fields.rating"}
    }}}
```

— Donner la note (rating) moyenne, et le rang moyen des films de George Lucas.

Correction

```
{"size": 0,,
  "query" :{
    "match" : {"fields.directors": {"query": "George Lucas", "operator":
↪"and"}}
  }
 ,"aggs" : {
    "note_moyenne" : {
      "avg" : {"field" : "fields.rating"}
    },
    "rang_moyen" : {
      "avg" : {"field" : "fields.rank"}
    }
}}
```

— Donnez la note (rating) moyenne des films par année. Attention, il y a ici une imbrication d'agrégats (on obtient par exemple 456 films en 2013 avec un *rating* moyen de 5.97).

Correction

```
{"size": 0,
"aggs" : {
```

(suite sur la page suivante)

(suite de la page précédente)

```
    "group_year" : {
      "terms" : {
        "field" : "fields.year"
      },
      "aggs" : {
        "note_moyenne" : {
          "avg" : {"field" : "fields.rating"}
        }}
    }}}
```

— Donner la note (rating) minimum, maximum et moyenne des films par année.

Correction

```
{"size": 0,
"aggs" : {
    "group_year" : {
      "terms" : {
        "field" : "fields.year"
      },
      "aggs" : {
        "note_moyenne" : {"avg" : {"field" : "fields.rating"}},
        "note_min" : {"min" : {"field" : "fields.rating"}},
        "note_max" : {"max" : {"field" : "fields.rating"}}
      }
    }}}
```

— Donner le rang (rank) moyen des films par année et trier par ordre décroissant.

Correction

```
{"size": 0,
"aggs" : {
    "group_year" : {
      "terms" : {
        "field" : "fields.year",
        "order" : { "rating_moyen" : "desc" }
      },
      "aggs" : {
        "rating_moyen" : {
          "avg" : {"field" : "fields.rating"}
      }}
}}}
```

— Compter le nombre de films par tranche de note (0-1.9, 2-3.9, 4-5.9…). Indice : `group_range`.

Correction

```
{"size": 0,
"aggs" : {
    "group_range" : {
      "range" : {
        "field" : "fields.rating",
        "ranges" : [
          {"to" : 1.9},
          {"from" : 2, "to" : 3.9},
          {"from" : 4, "to" : 5.9},
          {"from" : 6, "to" : 7.9},
          {"from" : 8}
        ]
      }
    }}}
```

4.3 S3 : le langage d'interrogation de MongoDB

Important : Cette section est conservée pour vous permettre d'aller plus loin mais elle ne fait plus partie du contenu « officiel » du cours. Elle ne sera pas présentée et ne fera l'objet d'aucune question à l'examen.

Supports complémentaires

— Présentation: requêtes MongoDB
— Vidéo de démonstration du langage d'interrogation MongoDB

Précisons tout d'abord que le langage de requête sur des collections est spécifique à MongoDB. Essentiellement, c'est un langage de recherche dit « par motif » (*pattern*). Il consiste à interroger *une* collection en donnant un objet (le « motif/*pattern* », en JSON) dont chaque attribut est interprété comme une contrainte sur la structure des objets à rechercher. Voici des exemples, plus parlants que de longues explications. Nous travaillons sur la base contenant les films complets, sans référence (donc, celle nommée `movies` si vous avez suivi les instructions du chapitrte précédent).

L'apprentissage de ce langage n'est pas le sujet de cette session. Ce qui suit ne vise qu'à illustrer une approche délibérement différente de SQL pour tenter d'adapter l'interrogation de bases de données aux documents structurés. Une courte discussion est consacrée à l'opération de jointure, qui n'existe en pas en MongoDB mais qui peut être obtenue en programmant nous-mêmes l'algorithme.

4.3.1 Sélections

Commençons par la base : on veut parcourir toute une collection. On utilise alors `find()` dans argument.

```
db.movies.find ()
```

S'il y a des millions de documents, cela risque de prendre du temps… D'ailleurs, comment savoir combien de documents comprend le résultat ?

```
db.movies.countDocuments ()
```

Comme en SQL (étendu), les options `skip` et `limit` permettent de « paginer » le résultat. La requête suivante affiche 12 documents à partir du dixième inclus.

```
db.movies.find ().skip(9).limit(12)
```

Implicitement, cela suppose qu'il existe un ordre sur le parcours des documents. Par défaut, cet ordre est dicté par le stockage physique : MongoDB fournit les documents dans l'ordre où il les trouve (dans les fichiers). On peut trier explicitement, ce qui rend le résultat plus déterministe. La requête suivante trie les documents sur le titre du film, puis pagine le résultat.

```
db.movies.find ().sort({"title": 1}).skip(9).limit(12)
```

La spécification du tri repose sur un objet JSON, et ne prend en compte que les noms d'attribut sur lesquels s'effectue le tri. La valeur (ici, celle du titre) ne sert qu'à indiquer si on trie de manière ascendante (valeur 1) ou descendante (valeur -1).

Attention, trier n'est pas anodin. En particulier, tout tri implique que le système constitue l'intégralité du résultat au préalable, ce qui induit une latence (temps de réponse) potentiellement élevée. Sans tri, le système peut délivrer les documents au fur et à mesure qu'il les trouve.

Critères de recherche

Si on connaît l'identifiant, on effectue la recherche ainsi.

```
db.movies.find ({"_id": "movie:33"})
```

Une requête sur l'identifiant ramène (au plus) un seul document. Dans un tel cas, on peut utiliser `findOne`.

```
db.movies.findOne ({"_id": "movie:33"})
```

Cette fonction renvoie toujours *un* document (au plus), alors que la fonction `find` renvoie un *curseur* sur un ensemble de documents (même si c'est un singleton). La différence est surtout importante quand on utilise une API pour accéder à MongoDB avec un langage de programmation.

Sur le même modèle, on peut interroger n'importe quel attribut.

```
db.movies.find ({"title": "Alien"})
```

Ca marche bien pour des attributs atomiques (une seule valeur), mais comment faire pour interroger des objets ou des tableaux imbriqués ? On utilise dans ce cas des chemins, un peu à la XPath, mais avec une syntaxe plus « orienté-objet ». Voici comment on recherche les films de Quentin Tarantino.

```
db.movies.find ({"director.last_name": "Tarantino"})
```

Et pour les acteurs, qui sont eux-mêmes dans un tableau ? Ca fonctionne de la même manière.

```
db.movies.find ({"actors.last_name": "Tarantino"})
```

La requête s'interprète donc comme : « Tous les films dont *l'un* des acteurs se nomme Tarantino ».

Conformément aux principes du semi-structuré, on accepte sans protester la référence à des attributs ou des chemins qui n'existent pas. En fait, dire « ce chemin n'existe pas » n'a pas grand sens puisqu'il n'y a pas de schéma, pas de contrainte sur la structure des objets, et que donc tout chemin existe potentiellement : il suffit de le créer. La requête suivante ne ramène rien, mais ne génére pas d'erreur.

```
db.movies.find ({"actor.last_name": "Tarantino"})
```

Important : Contrairement à une base relationnelle, une base semi-structurée ne proteste pas quand on fait une faute de frappe sur des noms d'attributs.

Quelques raffinements permettent de dépasser la limite sur le prédicat *d'égalité* implicitement utilisé ici pour comparer les critères donnés et les objets de la base. Pour les chaînes de caractères, on peut introduire des expressions régulières. Tous les films dont le titre commence par `Re` ? Voici :

```
db.movies.find ({"title": /^Re/})
```

Pas d'apostrophes autour de l'expression régulière. On peut aussi effectuer des recherches par intervalle.

```
db.movies.find( {"year": { $gte: 2000, $lte: 2005 } })
```

Projections

Jusqu'à présent, les requêtes ramènent l'intégralité des objets satisfaisant les critères de recherche. On peut aussi faire des *projections*, en passant un second argument à la fonction *find()*.

```
db.movies.find ({"director.last_name": "Tarantino"}, {"title": true, "actors": 'j
↪'} )
```

Le second argument est un objet JSON dont les attributs sont ceux à conserver dans le résultat. Notez que seules les clés du document JSON sont prises en compte (et correspondent aux attributs à conserver). La valeur ne compte pas, pourvu qu'elle soit différente de 0 ou `null`.

Opérateurs ensemblistes

Les opérateurs du langage SQL `in`, `not in`, `any` et `all` se retrouvent dans le langage d'interrogation. La différence, notable, est que SQL applique ces opérateurs à des *relations* (elles-mêmes obtenues par des requêtes) alors que dans le cas de MongoDB, ce sont des tableaux JSON. MongoDB ne permet pas d'imbriquer des requêtes.

Voici un premier exemple : on cherche les films dans lesquels joue au moins un des artistes dans une liste (on suppose que l'on connaît l'identifiant).

```
db.movies.find({"actors._id": {$in: ["artist:34","artist:98","artist:1"]}})
```

Gardez cette recherche en mémoire : elle s'avèrera utile pour contourner l'absence de jointure en MongoDB. Le `in` exprime le fait que *l'une* des valeurs du premier tableau (`actors._id`) doit être égale à l'une des valeurs de l'autre. Il correspond implicitement, en SQL, à la clause `ANY`. Pour exprimer le fait que *toutes* les valeurs de premier tableau se retrouvent dans le second (en d'autres termes, une inclusion), on utilise la clause `all`.

```
db.movies.find({"director._id": {$all: ["artist:23","artist:147"]}})
```

Le `not in` correspond à l'opérateur `$nin`.

```
db.movies.find({"director._id": {$nin: ["artist:34","artist:98","artist:1"]}})
```

Comment trouver les films qui n'ont pas d'attribut `summary` ?

```
db.movies.find({"summary": {$exists: false}}, {"title": 1})
```

Opérateurs Booléens

Par défaut, quand on exprime plusieurs critères, c'est une conjonction (`and`) qui est appliquée. On peut l'indiquer explicitement. Voici la syntaxe (les films tournés avec Leonardo DiCaprio en 1997) :

```
db.movies.find({$and : [{"year": 1997}, {"actors.last_name": "DiCaprio"}]} )
```

L'opérateur `and` s'applique à un tableau de conditions. Bien entendu il existe un opérateur `or` avec la même syntaxe. Les films parus en 1997 *ou* avec Leonardo DiCaprio.

```
db.movies.find({$or : [{"year": 1997}, {"actors.last_name": "DiCaprio"}]} )
```

Voici pour l'essentiel en ce qui concerne les recherches portant sur *une* collection et consistant à sélectionner des documents. Grosso modo, on obtient la même expressivité que pour SQL dans ce cas. Que faire quand on doit croiser des informations présentes dans *plusieurs* collections ? En relationnel, on effectue des jointures. Avec Mongo, il faut bricoler.

4.3.2 Jointures

La jointure, au sens de : associer des objets *distincts*, provenant en général de *plusieurs* collections, pour appliquer des critères de recherche croisés, n'existe pas en MongoDB. C'est une limitation importante du point de vue de la gestion de données. On peut considérer qu'elle est cohérente avec une approche documentaire dans laquelle les documents s'appuient sur la dénormalisation et sont supposés indépendants les uns des autres. Cela étant, on peut imaginer toutes sortes de situations où une jointure est *quand même* nécessaire dans une aplication de traitement de données.

Voyons comment nous pouvons contourner le problème. Nous allons supposer pour les besoins de la cause que la collection des films ne contient que les identifiants des artistes impliquées, et qu'une seconde collection contient les informations sur ces artistes (vous pouvez charger cette dernière collection à partir d'un fichier disponible sur le site).

Une première approche est de créer une *vue* qui assemble deux collections dans une troisième, virtuel. Cela suppose qu'on accepte de créer une vue pour chaque jointure…

La création de vue est la suivante :

```
    db.createView( "full_movies", "movies", [
    {
      $lookup:
             {
            from: "artists",
            localField: "director._id",
            foreignField: "_id",
            as: "metteur_en_scene"
            }
    }]
)
```

On crée une collection-vue `full_movies` qui étend chaque document de la collection `movies``en y intégrant un champ ``metteur_en_scene`, lequel contient le document de la collection `artists` correspondant à l'identifiant `director._id`` (relisez encore une fois si ce n'est pas clair…).

On peut alors interroger la collection `full_movies`, qui implante à peu près l'équivalent d'une jointure externe en relationnel.

```
db.full_movies.find()
```

L'autre approche consiste à effectuer la jointure côté client, comme illustré sur la Fig. 4.6. Cela revient essentiellement à appliquer l'algorithme de jointures par boucle imbriquées en stockant des données temporaires dans des structures de données sur le client, et en effectuant des échanges réseaux entre le client et le serveur, ce qui dans l'ensemble est très inefficace.

Comme l'interpréteur `mongo` permet de programmer en Javascript, nous pouvons en fait illustrer la méthode assez simplement. Considérons la requête : « Donnez tous les films dont le directeur est Clint Eastwood ».

La première étape dans la jointure côté client consiste à chercher l'artiste Clint Eastwood et à le stocker dans l'espace mémoire du client (dans une variable, pour dire les choses simplement).

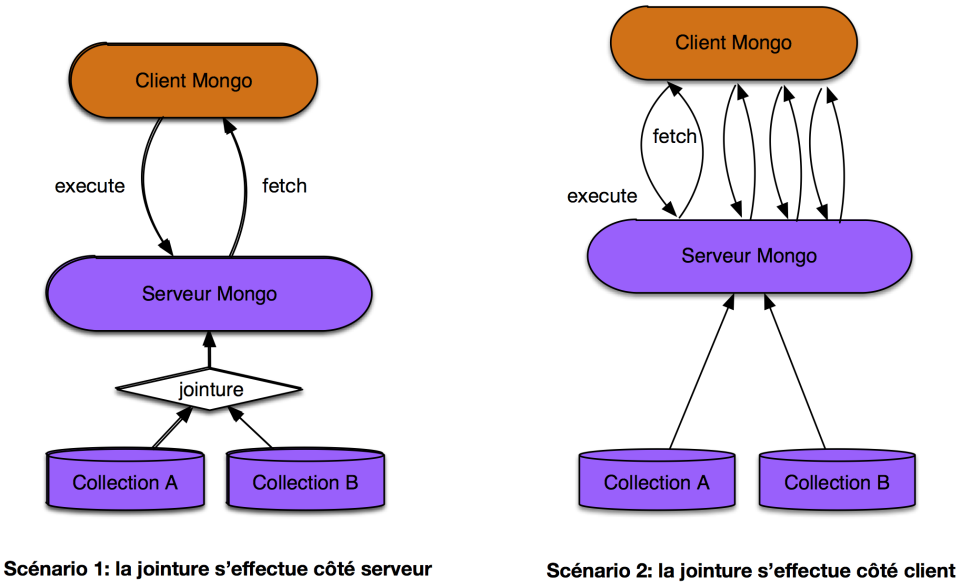


Fig. 4.6 – Jointure côté serveur et côté client

```
eastwood = db.artists.findOne({"first_name": "Clint", "last_name": "Eastwood"})
```

On dispose maintenant d'un objet `eastwood`. Une seconde requête va récupérer les films dirigés par cet artiste.

```
db.movies.find({"director._id": eastwood['_id']}, {"title": 1})
```

Voilà le principe. Voyons maintenant plus généralement comment on effectue l'équivalent des jointures en SQL. Prenons la requête suivante :

```
select m.titre, a.*  from Movie m, Artist a
where m.id_director = a.id
```

On veut donc les titres des films et le réalisateur. On va devoir coder, *du côté client*, un algorithme de jointure par boucles imbriquées. Le voici, sous le shell de MongoDB (et donc en programmation javascript).

```
var lesFilms = db.movies.find()
while (lesFilms.hasNext()) {
  var film = lesFilms.next();
  var mes = db.artists.findOne({"_id": film.director._id});
  printjson(film.title);
  printjson(mes);
}
```

On a donc une boucle, et une requête imbriquée, exécutée autant de fois qu'il y a de films. C'est exactement la méthode qui serait utilisée *par le serveur* si ce dernier implantait les jointures. L'exécuter du côté client induit un surcoût en programmation, et en échanges réseau entre le client et le serveur.

4.3.3 Mise en pratique

Voici quelques propositions d'exercices si vous souhaitez vous frotter concrètement à l'interrogation MongoDB. Les requêtes s'appliquent à la base des films.

- tous les titres ;
- tous les titres des films parus après 2000 ;
- le résumé de Spider-Man ;
- qui est le metteur en scène de *Gladiator* ?
- titre des films avec Kirsten Dunst ;
- quels films ont un résumé ?
- les films qui ne sont ni des drames ni des comédies.
- affichez les titres des films et les noms des acteurs.
- dans quels films Clint Eastwood est-il acteur mais pas réalisateur (aide : utilisez l'opérateur de comparaison `$ne`).
- **Difficile** : Comment chercher les films dont le metteur en scène est *aussi* un acteur ? Pas sûr que ce soit possible sans recourir à une auto-jointure, côté client…

Correction

- `db.movies.find({}, {"title": 1})`
- `db.movies.find({"year": {$gt: "2000"}}, {"title": 1, "year": 1})`
- `db.movies.find({"title": "Spider-Man"}, {"summary": 1})`
- `db.movies.find({"title": "Gladiator"}, {"director": 1})`
- `db.movies.find({"actors.last_name": "Dunst"}, {"title": 1})`
- `db.movies.find({"summary": {$exists: true}}, {"title": 1})`

  NB : cette fonction regarde si le champ existe, pas s'il est vide ou non. Dans la base, il existe des films avec un résumé ayant pour valeur null. Afin de ne récupérer que les films ayant réellement un résumé, on peut ajouter `$ne:null`

  ```
  db.movies.find({"summary": {$exists: true, $ne:null}}, {"title":
  1})
  ```

  donne les films dont le champ résumé existe et dont la valeur du champ est différente de null.
- `db.movies.find({"genre": {$nin: ["Drame", "Comédie"]}}, {"title": 1, "genre": 1})`
- `db.movies.find({}, {"title": 1, "actors.first_name": 1, "actors.last_name": 1})`
- `db.movies.find({"actors.last_name": "Eastwood", "director.last_name": {$ne: "Eastwood"}}, {"title": 1})`

CHAPITRE 5

Etude de cas : Cassandra

5.1 S1 : Cassandra, une base relationnelle étendue

Supports complémentaires

— Diapositives: le modèle de données Cassandra
— Vidéo sur le modèle de données Cassandra

Cassandra est un système de gestion de données à grande échelle conçu à l'origine (2007) par les ingénieurs de Facebook pour répondre à des problématiques liées au stockage et à l'utilisation de gros volumes de données. En 2008, ils essayèrent de le démocratiser en founissant une version stable, documentée, disponible sur Google Code. Cependant, Cassandra ne reçut pas un accueil particulièrement enthousiaste. Les ingénieurs de Facebook décidèrent donc en 2009 de faire porter Cassandra par l'Apache Incubator. En 2010, Cassandra était promu au rang de *top-level Apache Project*.

Apache a joué un rôle de premier plan dans l'attraction qu'a su créer Cassandra. La communauté s'est tellement investie dans le projet Cassandra que, au final, ce dernier a complètement divergé de sa version originale. Facebook s'est alors résolu à accepter que le projet - en l'état - ne correspondait plus précisément à leurs besoins, et que reprendre le développement à leur compte ne rimerait à rien tant l'architecture avait évolué. Cassandra est donc resté porté par l'Apache Incubator.

Aujourd'hui, c'est la société Datastax qui assure la distribution et le support de Cassandra qui reste un projet Open Source de la fondation Apache.

Cassandra a *beaucoup* évolué depuis l'origine, ce qui explique une terminologie assez erratique qui peut prêter à confusion. L'inspiration initiale est le système BigTable de Google, et l'évolution a ensuite plutôt porté Cassandra vers un modèle tendant vers le relationnel, avec quelques différences significatives, notamment sur les aspects internes. C'est un système NoSQL très utilisé, et sans doute un bon point de départ pour passer du relationnel à un système distribué.

5.1.1 Le modèle de données

Cassandra est un système qui s'est progressivement orienté vers un modèle relationnel étendu, avec typage fort et schéma contraint. Initialement, Cassandra était beaucoup plus permissif et permettait d'insérer à peu près n'importe quoi.

Note : Méfiez-vous des « informations » qui trainent encore sur le Web, où Cassandra est par exemple qualifié de « *column-store*, avec une confusion assez générale due en partie aux évolutions du système, et en partie au fait que certains se contentent de répéter ce qu'ils ont lu quelque part sans se donner la peine de vérifier ou même de comprendre.

Comme dans un système relationnel, une base de données Cassandra est constituée de *tables*. Chaque table a un nom et est constituée de colonnes. Toute ligne (*row*) de la table doit respecter le schéma de cette dernière. Si une table a 5 colonnes, alors à l'insertion d'une entrée, la donnée devra être composée de 5 valeurs respectant le typage. Une colonne peut avoir différents types,

— des types atomiques, comme par exemple *entier*, *texte*, *date* ;
— des types complexes (ensembles, listes, dictionnaires) ;
— des types construits et nommés.

Cela vous rappelle quelque chose ? Nous sommes effectivement proche d'un modèle de documents structurés de type JSON, avec imbrication de structures, mais avec un *schéma* qui assure le contrôle des données insérées. La gestion de la base est donc très contrainte et doit se faire en cohérence avec la structure de chaque table (son *schéma*). C'est une différence notable avec de nombreux systèmes NoSQL.

Important : Le vocabulaire encore utilisé par Cassandra est hérité d'un historique complexe et s'avère source de confusion. Ce manque d'uniformité et de cohérence dans la terminologie est malheureusement une conséquence de l'absence de normalisation des systèmes dits « No-SQL ». Dans tout ce qui suit, nous essayons de rester en phase avec les concepts (et leur nommage) présentés dans ce cours, d'établir le lien avec le vocabulaire Cassandra et si possible d'expliquer les raisons des écarts terminologiques. En particulier, nous allons utiliser *document* comme synonyme de *row* Cassandra, pour des raisons d'homogénéïté avec le reste de ce cours.

5.1.2 Paires clé/valeur (*columns*) et documents (*rows*)

La structure de base d'un document dans Cassandra est la paire *(clé, valeur)*, autrement dit la structure atomique de représentation des informations semi-structurées, à la base de XML ou JSON par exemple. Une valeur peut être atomique (entier, chaîne de caractères) ou complexe (dictionnaire, liste).

Vocabulaire

Dans Cassandra, cette structure est parfois appelée *colonne*, ce qui est difficilement explicable au premier abord (vous êtes d'accord qu'une paire-clé/valeur *n'est pas* une colonne ?). Il s'agit en fait d'un héritage de l'inspiration initiale de Cassandra, le système *BigTable* de Google dans lequel les données sont stockées en colonnes. Même si l'organisation finale de Cassandra a évolué, le vocabulaire est resté. Bilan : chaque fois

que vous lisez « colonne » dans le contexte Cassandra, comprenez « paire clé-valeur » et tout s'éclaircira.

Versions

Il existe une deuxième subtilité que nous allons laisser de côté pour l'instant : les valeurs dans une paire clé-valeur Cassandra sont associées à des versions. Au moment où l'on affecte une valeur à une clé, cette valeur est étiquetée par l'estampille temporelle courante, et il est possible de conserver, pour une même clé, la série temporelle des valeurs successives. Cassandra, à strictement parler, gère donc des *triplets* (clé, estampille, valeur). C'est un héritage de BigTable, que l'on retrouve encore dans HBase par exemple.

L'estampille a une utilité dans le fonctionnement interne de Cassandra, notamment lors des phases de réconciliation lorsque des fichiers ne sont plus synchronisés suite à la panne d'un nœud. Nous y reviendrons.

Un *document* dans Cassandra est un identifiant unique associé à un ensemble de paires *(clé, valeur)*. Il s'agit ni plus ni moins de la notion traditionnelle de dictionnaire que nous avons rencontrée dès le premier chapitre de ce cours et qu'il serait très facile de représenter en JSON par exemple.

Vocabulaire

Cassandra appelle *row* les documents, et *row key* l'identifiant unique. La notion de ligne (*row*) vient également de BigTable. Conceptuellement, il n'y a pas de différence avec les documents structurés que nous étudions depuis le début de ce cours.

Dans les versions initiales de Cassandra, le nombre de paires clé-valeur constituant un document (ligne) n'était pas limité. On pouvait donc imaginer avoir des documents contenant des milliers de paires, tous différents les uns des autres. Ce n'est plus possible dans les versions récentes, chaque document devant être conforme au schéma de la table dans laquelle il est inséré. Les concepteurs de Cassandra ont sans doute considéré qu'il était malsain de produire des fourre-tout de données, difficilement gérables. La Fig. 5.1 montre un document Cassandra sous la forme de ses paires clés-valeurs

5.1.3 Les tables (*column families*)

Les documents sont groupés dans des tables qui, sous Cassandra, sont parfois appelées des *column families* pour des raisons historiques.

Vocabulaire

La notion de *column family* vient là encore de Bigtable, où elle avait un sens précis qui a disparu ici (pourquoi appeler une collection une « famille de colonnes ? »). Transposez *column family* en *collection* et vous serez en territoire connu. Pour retrouver un modèle encore très proche de celui de BigTable, vous pouvez regarder le système HBase où les termes *column family* et *column* ont encore un sens fort.

Note : Il existe aussi des *super columns*, ainsi que des *super column families*. Ces structures apportent un réel niveau de complexité dans le modèle de données, et il n'est pas vraiment nécessaire d'en parler ici. Il se

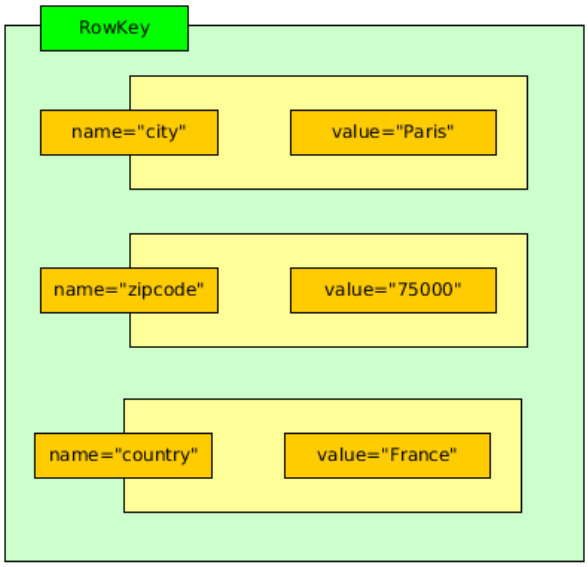


Fig. 5.1 – Structure d'un document dans Cassandra

peut d'ailleurs que ces notions peu utiles disparaissent à l'avenir.

La Fig. 5.2 illustre une table et 3 documents avec leur identifiant.

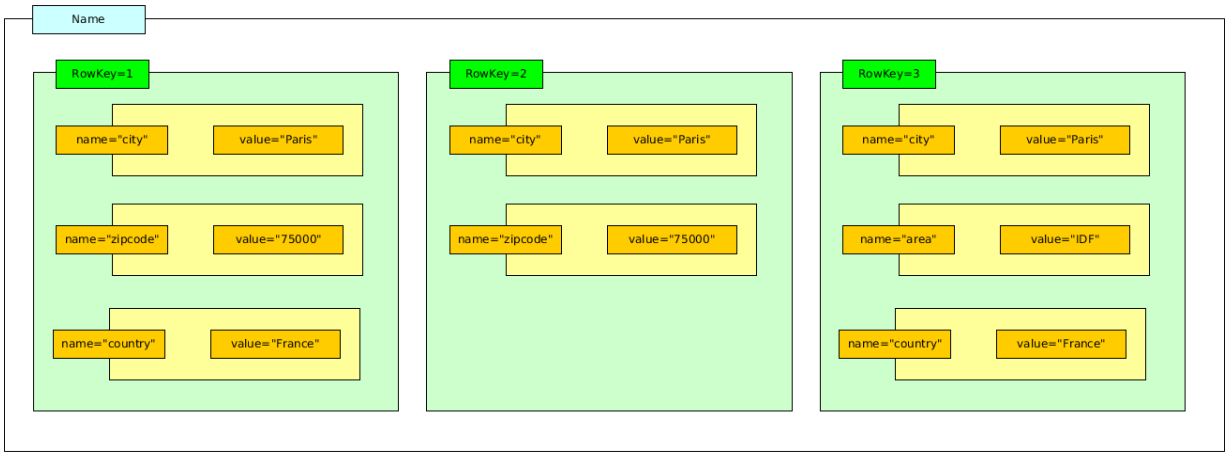


Fig. 5.2 – Une table (*column family*) contenant 3 documents (*rows*) dans Cassandra

5.1.4 Bases (*Keyspaces*)

Enfin le troisième niveau d'organisation dans Cassandra est le *keyspace*, qui contient un ensemble de tables (*column families*). C'est l'équivalent de la notion de base de données, ensemble de tables dans le modèle relationnel, ou ensemble de collections dans des systèmes comme MongoDB.

En résumé :

- Cassandra permet de stocker des tables *dénormalisées* dans lesquelles les valeurs ne sont pas nécessairement atomiques ; il s'appuie sur une plus grande diversité de types (pas uniquement des entiers et des chaînes de caractères, mais des types construits comme les listes ou les dictionnaires).
- La modélisation d'une architecture de données dans Cassandra est beaucoup plus ouverte qu'en relationnel ce qui rend notamment la modélisation plus difficile à évaluer, surtout à long terme.
- La dénormalisation (souvent considérée comme la bête noire à pourchasser dans un modèle relationnel) devient recommandée avec Cassandra, en restant conscient que ses inconvénients (notamment la duplication de l'information, et les incohérences possibles) doivent être envisagés sérieusement.
- En contrepartie des difficultés accrues de la modélisation, et surtout de l'impossibilté de garantir formellement la qualité d'un schéma grâce à des méthodes adaptées, Cassandra assure un passage à l'échelle par distribution basé sur des techniques de partitionnement et de réplication que nous détaillerons ultérieurement. C'est un système qui offre des performances jugées très satisfaisantes dans un environnement Big Data.

5.1.5 Créons notre base

À vous de vous retrousser les manches pour créer votre base Cassandra et y insérer nos films (ou toute autre jeu de données de votre choix). Les commandes de base sont données ci-dessous ; elles peuvent toutes être entrées directement dans client graphique comme DbVisualizer. Le *keyspace* ————-

Rappelons que *keyspace* est le nom que Cassandra donne à une base de données. Cassandra est fait pour fonctionner dans un environnement distribué. Pour créer un *keyspace*, il faut donc préciser la stratégie de réplication à adopter. Nous verrons plus en détail après comment tout ceci fonctionne. Voici la commande en ligne :

```
CREATE KEYSPACE IF NOT EXISTS Movies
       WITH REPLICATION = { 'class' : 'SimpleStrategy', 'replication_factor': 3 }
↪;
```

Sous DbVisualizer, les *keyspaces* apparaissent à gauche de la fenêtre principale. Un clic bouton droit permet d'ouvrir un formulaire de création d'un *keyspace* (voir Fig. 5.3).

Une fois le *keyspace* créé, essayez les commandes suivantes (sous `cqlsh` uniquement).

```
cqlsh > DESCRIBE keyspaces;
cqlsh > DESCRIBE KEYSPACE Movies;
```

Avec un client graphique, il est facile d'explorer un *keyspace*. Sous DbVisualizer, vous pouvez entrer les commandes dans une fenêtre « SQL commander ».

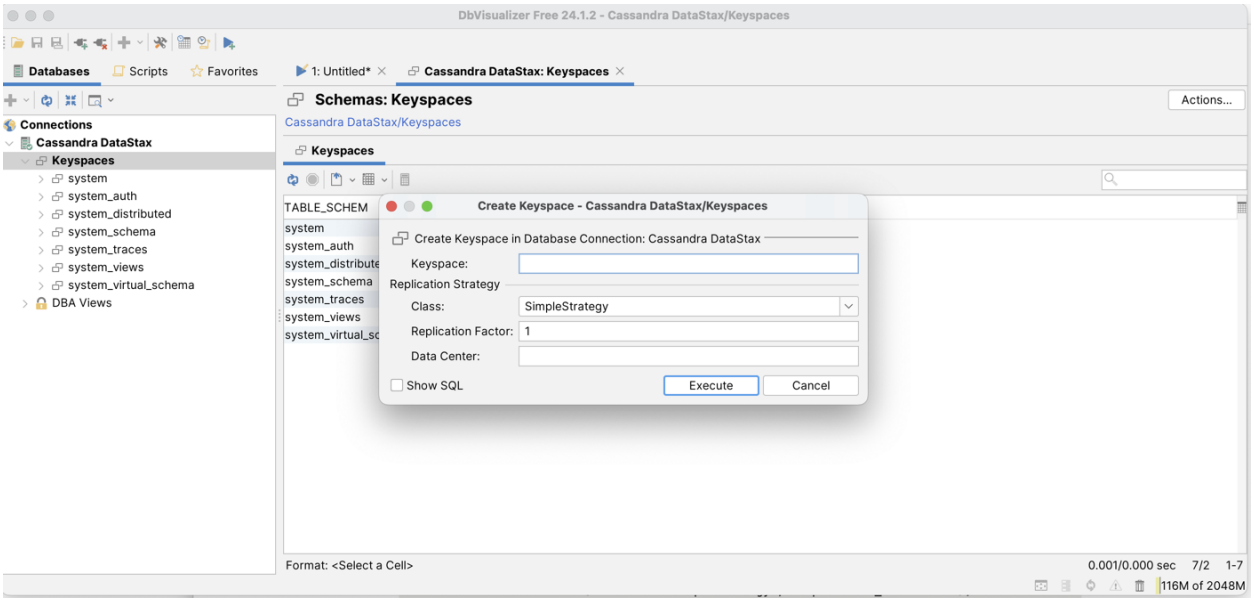


Fig. 5.3 – Utilisation de DbVisualizer – Création d'un *keyspace*

Important : Il peut être nécessaire de se reconnecter à Cassandra pour que le *keyspace* devienne visible.

Données relationnelles (à plat)

On peut traiter Cassandra comme une base relationnelle (en se plaçant du point de vue de la modélisation en tout cas). On crée alors des tables destinées à contenir des données « à plat », avec des types atomiques. Commençons par créer une table pour nos artistes.

```
create table artists (id text,
              last_name text, first_name text,
              birth_date int, primary key (id)
             );
```

Je vous renvoie à la documentation Cassandra pour la liste des types atomiques disponibles. Ce sont, à peu de chose près, ceux de SQL.

L'insertion de données suit elle aussi la syntaxe SQL. Insérons quelques artistes.

```
insert into artists (id, last_name, first_name, birth_date)
            values ('artist1', 'Depardieu', 'Gérard', 1948);
insert into artists (id, last_name, first_name, birth_date)
            values ('artist2', 'Baye', 'Nathalie', 1948);
insert into artists (id, last_name, first_name)
            values ('artist3', 'Marceau', 'Sophie');
```

On peut vérifier que l'insertion a bien fonctionné en sélectionnant les données.

```
select * from artists;

 id         | last_name    | first_name     | birth_date
------------+--------------+----------------------------
  'artist1' | Depardieu    | Gérard         | 1948
  'artist2' | Baye         | Nathalie       | 1948
  'artist3' | Marceau      | Sophie         | null
```

À la dernière insertion, nous avons délibérément omis de renseigner la colonne `birth_date`, et Cassandra accepte la commande sans retourner d'erreur. Cette flexibilité est l'un des aspects communs à tous les modèles s'appuyant sur une représentation semi-structurée.

Il est également possible d'insérer à partir d'un document JSON en ajoutant le mot-clé `JSON`.

```
insert into artists JSON '{
     "id": "a1",
     "last_name": "Coppola",
     "first_name": "Sofia",
     "birth_date": "1971"
 }';
```

La structure du document doit correspondre très précisément (types compris) au schéma de la table, sinon Cassandra rejette l'insertion.

Note : Vous pouvez récupérer sur le site http://deptfod.cnam.fr/bd/tp/datasets/ des commandes d'insertion Cassandra pour notre base de films.

Documents structurés (avec imbrication)

Cassandra va au-delà de la norme relationnelle en permettant des données *dénormalisées* dans lesquelles certaines valeurs sont complexes (dictionnaires, ensembles, etc.). C'est le principe de base que nous avons étudié pour la modélisation de document : en permettant l'imbrication on s'autorise la création de structures beaucoup plus riches, et potentiellement suffisantes pour représenter intégralement les informations relatives à une entité.

Note : Le concept de relationnel « étendu » à des types complexes est très ancien, et existe déjà dans des systèmes comme Postgres depuis longtemps.

Prenons le cas des films. En relationnel, on aurait la commande suivante :

```
create table movies (id text,
            title text,
            year int,
            genre text,
```

(suite sur la page suivante)

(suite de la page précédente)

```
            country text,
          primary key (id) );
```

Tous les champs sont de type atomique. Pour représenter le metteur en scène, objet complexe avec un nom, un prénom, etc., il faudrait associer (en relationnel) chaque ligne de la table *movies* à une ligne d'une *autre* table représentant les artistes.

Cassandra permet l'imbrication de la représentation d'un artiste dans la représentation d'un film ; une seule table suffit donc. C'est le principe de dénormalisation : on regroupe les données le plus possible dans des lignes pour éviter les jointures. Une valeur d'attribut peut correspondre :

— à un ensemble de valeurs (non ordonnées) : `SET`
— à une liste de valeurs : `LIST`
— à un dictionnaire : `MAP`
— à un nuplet : *TUPLE* `
— à une instance d'un type

Définissons le *type* `artist` de la manière suivante :

```
create type artist (id text,
                    last_name text,
                    first_name text,
                    birth_date int,
                    role text);
```

Et on peut alors créer la table `movies` en spécifiant que l'un des champs a pour type `artist`.

```
create table movies (id text,
                     title text,
                     year int,
                     genre text,
                     country text,
                     director frozen<artist>,
                     primary key (id) );
```

Notez le champ `director`, avec pour type `frozen<artist>` indiquant l'utilisation d'un type défini dans le schéma.

Note : L'utilisation de `frozen` semble obligatoire pour les types imbriqués. Les raisons sont peu claires pour moi. Il semble que `frozen` implique que toute modification de la valeur imbriquée doive se faire par remplacement complet, par opposition à une modification à une granularité plus fine affectant l'un des champs. Vous êtes invités à creuser la question si vous utilisez Cassandra.

Il devient alors possible d'insérer des documents structurés, comme celui de l'exemple ci-dessous. Ce qui montre l'équivalence entre le modèle Cassandra et les modèles des documents structurés que nous avons étudiés. Il est important de noter que les concepteurs de Cassandra ont décidé de se tourner vers un typage fort : tout document non conforme au schéma précédent est rejeté, ce qui garantit que la base de données est saine et respecte les contraintes.

```
INSERT INTO movies JSON '{
    "id": "movie:1",
    "title": "Vertigo",
    "year": 1958,
    "genre": "drama",
    "country": "USA",
    "director": {
        "id": "artist:3",
        "last_name": "Hitchcock",
        "first_name": "Alfred",
        "birth_date": "1899"
    }
}';
```

Sur le même principe, on peut ajouter un niveau d'imbrication pour représenter l'ensemble des acteurs d'un film. Le constructeur `set<...>` déclare un type *ensemble*. Voici un exemple parlant :

```
create table movies (id text,
              title text,
              year int,
              genre text,
              country text,
              director frozen<artist>,
              actors set< frozen<artist>>,
           primary key (id) );
```

Les acteurs sont donc une liste d'instances du type `artist`, ce qui correspond en JSON à la structure suivante :

```
insert into movies JSON '{
        "id": "movie:11",
        "title": "Star Wars",
        "year": 1977,
        "genre": "Adventure",
        "country": "US",
        "director": {
                "id": "artist:1",
                "last_name": "Lucas",
                "first_name": "George",
                "birth_date": 1944
        },
        "actors": [
                {
                        "last_name": "Hamill",
                        "first_name": "Mark",
                        "birth_date": 1951
                },
```

(suite sur la page suivante)

(suite de la page précédente)

```
                {
                        "last_name": "Ford",
                        "first_name": "Harrison",
                        "birth_date": 1942
                },
                {
                        "last_name": "Fisher",
                        "first_name": "Carrie",
                        "birth_date": 1956
                }
        ]
}';
```

Je vous laisse effectuer (si ce n'est déjà fait) l'insertion de l'ensemble des films tels qu'ils sont fournis par le site http://deptfod.cnam.fr/bd/tp/datasets/cassandra, avec tous les acteurs d'un film. Il suffit de récupérer le fichier contenant l'ensemble des commandes d'insertion et de l'exécuter comme un script. Nous nous en servirons pour l'interrogation CQL ensuite.

En résumé :

- Cassandra propose un modèle relationnel étendu, basé sur la capacité à imbriquer des types complexes dans la définition d'un schéma, et à sortir en conséquence de la première règle de normalisation (ce type de modèle est d'ailleurs appelé depuis longtemps N1NF pour *Non First Normal Form*) ;
- Cassandra a choisi d'imposer un typage fort : toute insertion doit être conforme au schéma ;
- L'imbrication des constructeurs de type, notamment les *dictionnaires* (nuplets) et les *ensembles* (set) rend le modèle comparable aux documents structurés JSON ou XML.

Il faut noter que les ensembles doivent rester de taille raisonnable sous peine de dégrader les performances. La conception d'un schéma Cassandra repose sur des principes très spécifiques sur lesquels nous revenons plus loin.

5.2 S2 : requêtes Cassandra

Supports complémentaires

- Diapositives: CQL et ses limitations (et pourquoi…)

Cassandra propose un langage, nommé CQL, inspiré de SQL, mais fortement restreint par l'absence de jointure. De plus, d'autres types de restrictions s'appliquent, motivées par l'hypothèse qu'une base Cassandra est nécessairement une base à très grande échelle, et que les seules requêtes raisonnables sont celles pour lequelles la structuration des données permet des temps de réponse acceptables.

Note : Cette session est une démonstration pratique ces capacités d'interrogation de Cassandra. Si vous souhaitez reproduire les manipulations, il vous faut un environnement constitué d'un serveur Cassandra, d'un client et de la base de données des films. En résumé, vous devriez avoir une table `movies` où chaque film contient des données imbriquées représentant le réalisateur du film et les acteurs.

5.2.1 CQL, un sous-ensemble de SQL

CQL ne permet d'interroger qu'une seule table. Cette (*très* forte) restriction mise à part (!), le langage est délibérement conçu comme un sous-ensemble de SQL et de sa construction `select from where`.

Note : Toute requête CQL doit se terminer par un ";"

Commençons par quelques exemples.

Sélectionnons tous les films.

```
select  * from movies;
```

Selon l'utilitaire que vous utilisez, vous devriez obtenir l'affichage des premiers films sous une forme ou sous une autre. Cassandra étant supposé gérer de très grandes bases de données, ces utilitaires vont souvent ajouter automatiquement une clause limitant le nombre de lignes retournées. Vous pouvez ajouter cette clause explicitement.

```
select  * from movies limit 20;
```

On peut obtenir le résultat encodé en JSON en ajoutant simplement le mot-clé `JSON`.

```
select JSON * from movies;
```

Bien entendu, le * peut être remplacé par la liste des attributs à conserver (projeter).

```
select title from movies;
```

Si une valeur v est un dictionnaire (objet en JSON), on peut accéder à l'un de ses composants c avec la notation $v.c$. Exemple pour le réalisateur du film.

```
select title, director.last_name from movies;
```

En revanche, quand la valeur est un ensemble ou une liste, on ne sait pas avec CQL accéder à son contenu. La tentative d'exécuter la requête :

```
select title, actors.last_name from movies;
```

devrait retourner une erreur. Il est vrai que l'on ne sait pas très bien à quoi devrait ressembler le résultat. Ou plus exactement on le sait, mais cela supposeait que Cassandra soit capable de créer des types à la volée. D'autres langages (notamment XQuery, mais également le langage de script Pig que nous étudierons en fin de cours) proposent des solutions au problème d'interrogation de collections imbriquées. Il se peut que CQL évolue un jour pour proposer quelque chose de semblable.

On peut, dans la clause `select`, appliquer des fonctions. Cassandra permet la définition de fonctions utilisateur, et leur application aux données grâce à CQL. Quelques fonctions prédéfinies sont également disponibles. Voici un exemple (sans intérêt autre qu'illustratif) de conversion de l'année du film en texte (c'est un entier à l'origine).

```
select cast(year as text) as yearText from movies ;
```

Notez le renommage de la colonne avec le mot-clé `as`. Tout cela est directement emprunté à SQL. On peut également compter le nombre de lignes dans la table.

```
select count(*) from movies ;
```

On peut effectuer des filtrages avec la clause `where`. Par exemple :

```
select  *  from movies where id='movie:33';
```

Remarque importante : le critère de sélection porte ici sur la *clé*. On peut généraliser à plusieurs valeurs avec la clause `in`.

```
select  * from movies
where id in ('movie:33', 'movie:44214', 'movie:29845');
```

Tentons maintenant une recherche sur un attribut non-clé.

```
select  * from movies
where title='Elle' ;
```

Vous devriez obtenir un rejet de cette requête avec le message suivant :

```
Unable to execute CQL script. Cannot execute this query as it might involve data
filtering and thus may have unpredictable performance. If you want
to execute this query despite the performance unpredictability,
use ALLOW FILTERING.
```

En revanche, en ajoutant l'option ALLOW FILTERING, on obtient le résultat.

```
select  * from movies
where title='Elle'
ALLOW FILTERING;
```

Nous avons atteint les limites de CQL en tant que clône de SQL.

5.2.2 Pourquoi CQL n'est pas SQL

Pourquoi un `where` sur un attribut non-clé est-il rejeté ? Pour une raison qui tient à l'organisation des données : Cassandra organise une table selon une structure qui permet très rapidement de trouver un document par sa clé. La recherche par clé est donc autorisée. Pour aller plus loin, il faut regarder plus en détail le schéma d'une table. La syntaxe complète est ci-dessous :

```
   create table Tname (
          part_key_1 type,
          part_key_2 type,
```

(suite sur la page suivante)

(suite de la page précédente)

```
          ... type,
          clust_key_1 type,
          ... type,
          att_1 type,
          att_2 type,
          ... type,
          primary key (
      (part_key_1, part_key2, ...),
      (clust_key_1, clust_key_2, ...)
  )
)
```

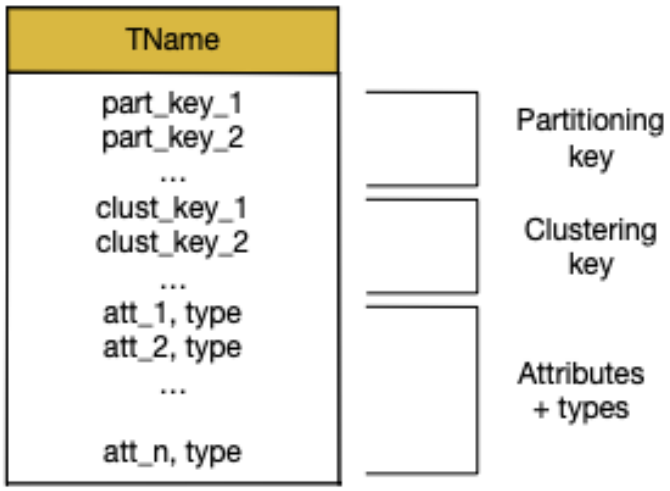


Fig. 5.4 – Le schéma d'une table, avec la structure d'une clé

Les attributs d'un schéma peuvent être divisés en deux parties : les *attributs de la clé* et les attributs non-clés. Mais les attributs de la clé eux-mêmes se divisent en deux (Fig. 5.4) :

— les attributs de *partitionnement* : ils déterminent le placement de la ligne sur un serveur
— les attributs de *regroupement* : ils servent à trier les lignes sur un même serveur.

Il faut ici anticiper un peu sur l'étude de Cassandra comme système distribué. Sans entrer dans les détails, une table Cassandra est censée être très volumineuse. Cassandra la découpe en *fragments* et place chaque fragment sur un des serveurs du système distribué (Fig. 5.5). Ce système a la forme d'un anneau mais nous allons laisser de côté cette caractéristique pour l'instant.

Le contenu d'un fragment est déterminé par la *clé de partitionnement*. Pour chaque ligne on applique en effet une fonction (de hachage) aux valeurs de la clé de *partitionnement*. La valeur retournée par cette fonction détermine le placement dans le système distribué. Conséquence : **une requête incluant comme critère une clé de partitionnement ne concerne qu'un seul serveur**.

De plus,d ans un fragment (sur un serveur) les lignes sont triées sur la clé primaire. Pour une même valeur de partitionnement, les lignes sont donc *consécutives* et ordonnées sur la clé de regroupement. Une requête sur

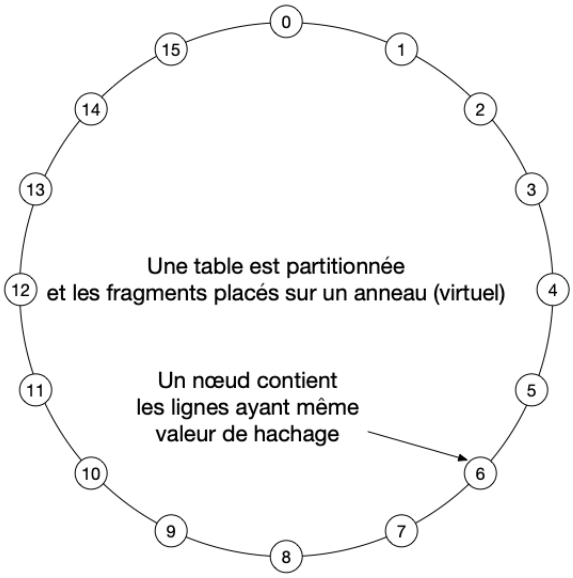


Fig. 5.5 – Cassandra est un système distribué : la *clé de partitionnement* détermine le serveur

une préfixe de la clé primaire (part. + regroupement) correspond à un parcours *séquentiel* sur un seul nœud, beaucoup plus efficace qu'une recherche aléatoire.

Prenons un exemple concret : nous considérons que l'ensemble imbriqué des acteurs d'un film est potentiellement trop grand pour l'imbriquer dans la table `movies`. On peut alors créer une table `roles` comme suit :

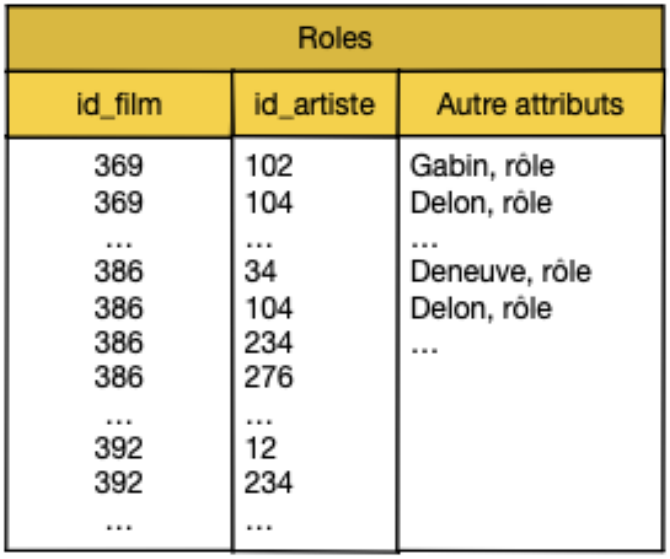
```
create table Roles (
    id_film int,
    id_artiste int,
    role text,
    artist frozen<artist>,
    primary key (id_film, id_artiste)
 )
```

La clé de partitionnement est donc l'identifiant du film, et la clé de regroupement l'identifiant de l'artiste. Tous les rôles d'un même film seront sur le même serveur. Ils seront de plus regroupés et triés par identifiant d'artiste.

Toute recherche sur un autre attribut (par exemple l'intitulé du rôle ou le nom de l'artiste) n'a d'autre solution que de parcourir séquentiellement toute la table en effectuant le test sur le critère de recherche à chaque fois.

Comme déjà indiqué, Cassandra est conçu pour de très grandes bases de données, et le rejet de ces requêtes séquentielle est une précaution. Le message indique clairement à l'utilisateur que sa requête est susceptible de prendre beaucoup de temps à s'exécuter. À l'usage on décrouvre tout un ensemble de restrictions (par rapport à SQL) qui s'expliquent par cette volonté d'éviter l'exécution d'une requête qui impliquerait un parcours de tout ou partie de la table. Voyons quelques exemples, avec explications.

Tentons une requête sur la clé primaire, mais avec un critère *d'inégalité*.



Roles		
id_film	id_artiste	Autre attributs
369	102	Gabin, rôle
369	104	Delon, rôle
...	...	...
386	34	Deneuve, rôle
386	104	Delon, rôle
386	234	...
386	276	
...	...	
392	12	
392	234	
...	...	

Fig. 5.6 – Cassandra est un système distribué : la *clé de partitionnement* détermine le serveur

```
select  * from movies
where id > '000000';
```

On obtient un rejet avec un message indiquant que seule l'égalité est autorisée sur la clé (et d'autres détails à éclaircir ultérieurement).

Peut-on trier les données avec la clause `order by` ? Essayons.

```
select  * from movies order by title;
```

Les deux requêtes sont rejetées. Le message nous dit (à peu près) que le tri est autorisé seulement quand on est assuré que les données à trier proviennent d'une seule partition. En (un peu plus) clair : Cassandra ne veut pas avoir à trier des données provenant de plusieurs serveurs, dans un environnement distribué avec répartition d'une table sur plusieurs nœuds.

Et voilà. Cassandra interdit tout usage de CQL qui amènerait à parcourir toute la base ou une partie non prédictible de la base pour constituer le résultat. Cette interdiction n'est cependant pas totale. Dans le cas de la clause `where`, l'utilisateur peut prendre explicitement ses responsabilités en ajoutant la clause `allow filtering`, comme nous l'avons montré ci-dessus. Si la table contient des milliards de ligne, il faudra certainement attendre longtemps et exploiter intensivement les ressources du système pour un résultat limité. À utiliser à bon escient donc.

Il faut penser que le coût d'évaluation de cette requête est proportionnel à la taille de la base. Cassandra tente de limiter les requêtes à celles dont le coût est proportionnel à la taille du résultat.

Note : Cette remarque explique pourquoi la requête `select * from movies;`, qui parcourt toute la base,

est autorisée.

À partir du moment où on autorise explicitement le filtrage, on peut combiner plusieurs critères de recherche, comme en SQL.

```
select  * from movies
where country='US' and year=2020 allow filtering;
```

Mais, si c'est pour faire du SQL, autant choisir une base relationnelle. Les restrictions de Cassandra doivent s'interpréter dans un contexte *Big Data* où l'accès aux données doit prendre en compte leur volumétrie (et notamment le fait que cette volumétrie impose une répartition des données dans un système distribué).

Une autre possibilité est de créer un index secondaire sur les attributs auxquels on souhaite appliquer des critères de recherche.

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

Cassandra autorise alors de requêtes avec la clause `where` portant sur les attributs indexés.

```
select * from movies where year = 2020;
```

En présence d'un index, il n'est plus nécessaire de parcourir toute la collection. Cette option est cependant à utiliser avec prudence. En premier lieu, un index peut être coûteux à maintenir. Mais surtout sa sélectivité n'est pas toujours assurée. Ici, par exemple, un index sur l'année est probablement une très mauvaise idée. On peut estimer qu'un film sur 100 a été tourné en 1992, et à l'échelle du *Big Data*, ça laisse beaucoup de films à trouver, même avec l'index, et une requête qui peut ne pas être performante du tout.

5.3 S3 : étude de cas : conception d'un schéma

Supports complémentaires
— Etude de cas Cassandra

De nombreux conseils sont disponibles pour la conception d'un schéma Cassandra. Cette conception est nécessairement différente de celle d'un schéma relationnel à cause de l'absence du système de clé étrangère et de l'opération de jointure. C'est la raison pour laquelle de nombreux design patterns sont proposés pour guider la mise en place d'une architecture de données dans Cassandra qui soit cohérente avec les besoins métiers, et la performance que peut offrir la base de données. Je présente dans ce qui suit une étude de cas concrète, menée pour une entreprise souhaitant archiver des dépôts de code logiciel de type GitHub. Les principes adoptés sont largement inspirés de la documentation officielle Cassandra que vous pouvez consulter à https://cassandra.apache.org/doc/stable/cassandra/data_modeling/index.html.

5.3.1 Les principes

En l'absence de normalisation comme en relationnel, plusieurs solutions sont possibles, présentant des caractéristiques différentes en terme de capacité à satisfaire certaines requêtes. Il en résulte que le schéma est construit en fonction d'un besoin, ce qui est problématique puisque d'une part il faut garantir que ce besoin est correctement exprimé, et que d'autre part le besoin eut évoluer ou d'autres peuvent apparaître, le schéma devenant du coup obsolète. Avec un système relationnel comme MySQL, le raisonnement est opposé : la disponibilité des jointures permet de se fixer comme but la normalisation du modèle de données afin de répondre à tous les cas d'usage possibles, éventuellement de manière non optimale.

L'argument des défenseurs de Cassandra est que cette approche est acceptable pour une application de type WORM (Write Once, Read many). C'est le cas pour l'application d'archivage que nous allons étudier.

Admettons que le besoin soit bien identifié. On doit alors scénariser la séquence des requêtes transmises par l'application (en l'absence de jointure, une seule requête ne peut suffire, on effectue une requête par table). On organise alors les tables pour que chaque requête s'exécute localement et séquentiellement, avec un critère d'accès portant sur la clé. *Le principe est que l'exécution d'une requête A fournit la valeur de la clé qui sert d'acès à la requête suivante.*

On peut créer ponctuellement des index ou des vues matérialisées. Au pire (mais est-ce évitable ?) on multiplie les organisations physiques et donc la redondance.

5.3.2 Les besoins

Nous voulons donc archiver des dépôts de données de type GitHub ou GitLab. La Fig. 5.7 montre notre modèle de données (simplifié). Nous avons donc des dépôts (*repository*), par exemple https://github.com/apache/cassandra, contenant le code d'un logiciel ou autre ressource. Pour archiver ce dépôt, on effectue périodiquement des visites (*snapshot*) et on capture le contenu du dépôt à la date de visite.

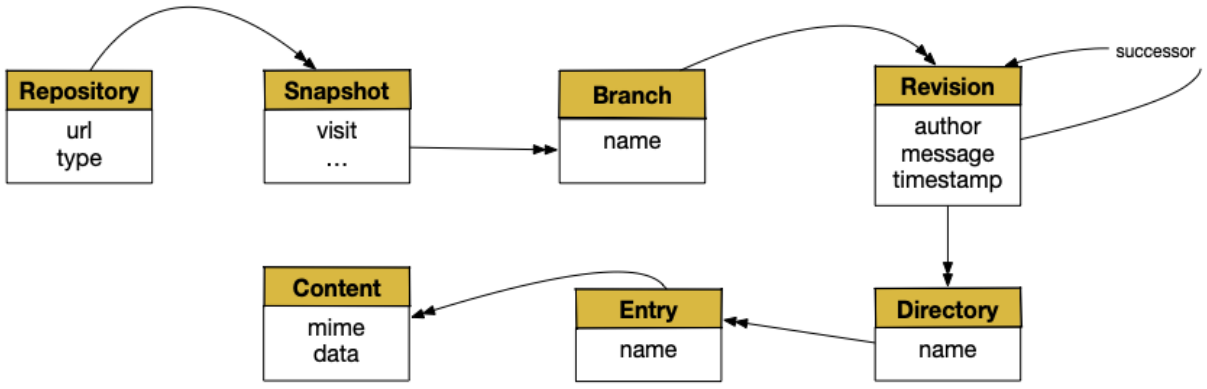


Fig. 5.7 – Notre modèle de données

La structure d'un dépôt logiciel est constituée de *branches* qui évoluent en parallèle les unes des autres, chaque évolution consituant une révision. Enfin, dans une révision, on stocke une arborescence de répertoire (*directory*) contenant des *entry* de plusieurs types (fichiers, ou liens, ou sous-répertoires, etc.). Dans le cas des fichiers, on stocke le contenu (*content*).

Le besoin consiste à explorer cette structure en partant d'un dépôt identifié par son URL. On peut vouloir connaître la liste des visites, choisir une visite et lister les branches, choisir une branche et consulter la liste des révisions. Enfin, ayant choisi une révision, on peut reconstituer l'arborescence de son contenu.

Remarquer que dans un modèle relationnel, chaque besoin correspondrait à une seule requête avec plus ou moins de jointure selon la profondeur d'exploration. Avec Cassandra, chaque besoin va correspondre à une séquence plus ou moins longue de requêtes mono-tables. Encore faut-il s'assurer que chaque requête va s'effectuer efficacement.

5.3.3 Schéma et requêtes

Prenons pour commencer l'hypothèse qu'on cherche les visites faites à un dépôt. Une première approche purement relationnelle, avec clé primaire et clé étrangère, ne fonctionne pas.

```
create table Repository
   (repo_id uuid,
    url varchar,
    description text,
    primary key (repo_id));

create table Visit
   (visit_id uuid,
    repo_id
    visit_ref int,
    date date,
    primary key (visit_id)
   );
```

Il faut deux requêtes (pas de jointure) pour trouver les visites faites au dépôt Cassandra. De plus aucune n'est indexée : une catastrophe dans un contexte de données massives.

```
select repo_id in oid
from Repository
where url ='github.com/apache/cassandra'

select * from Visit
where repo_id = oid
```

Cassandra refuse les deux puisqu'aucune requête n'utilise la clé comme critère ! Organisons les clés et chemins d'accès, en prenant comme principe que les critères de recherche doivent être dans la clé. La table `Repository` est identifiée par l'URL qui servira toujours de point d'accès.

```
create table Repository
 (url varchar,
  description text,
  primary key (url)
  );
```

On peut donc toujours trouver très efficacement un dépôt.

```
select * from Repository
where  url ='github.com/apache/cassandra';
```

Pour trouver les visites à une URL, on peut commencer à exploiter la structure des clés et le placement physique qu'elle induit.

```
create table Visit_by_repository
 (url_repo varchar,
  visit_ref int,
  date date,
  primary key (url_repo,
               visit_ref)
  );
```

Notez d'abord le nommage. On ne peut accéder (efficacement) aux visites que si on connaît l'URL qui nous intéresse. C'est le *point d'accès*. On nomme donc la table `Visit_by_repository` pour bien indiquer cette restriction. Toutes les visites à un dépôt seront sur un même serveur et stockées consécutivement. On peut alors effectuer *efficacement* les requêtes suivantes.

```
select * from repository
where  url ='github.com/apache/cassandra';

select * from Visit_by_repository
where  url ='github.com/apache/cassandra';

select * from Visit_by_repository
where  url ='github.com/apache/cassandra'
and visit_ref < 3;
```

Il existe des alternatives. On peut imbriquer les visites dans la table Repository en créant un type `Visit`.

```
create type Visit (
    number int,
    date_visit date)
```

Et en l'imbriquant dans `Repository`

```
create table Repository
   (url varchar,
    description text,
    visites list<Visit>,
    primary key (url)
    )
```

Cette option suppose un nombre limité de visites (l'ordre de grandeur étant assez flou, quelques dizaines semble-t-il ?). On dispose alors d'une requête localisée et unique pour trouver toutes les visites d'un dépôt.

```
select * from Repository
where  url ='github.com/apache/cassandra'
```

Cela permet des requêtes plus sophistiquées, comme les dépôts visités au moins 10 fois.

```
select * from Repository
where  url ='github.com/apache/cassandra'
and count(visit.number) >= 10
```

Introduisons maintenant les clichés *Snapshot* avec comme point d'accès l'URL d'un dépôt.

```
create table Snapshot_by_repository
   (url varchar,
    snapshot_date,
    snapshot_id uuid,
    repo Repository,
    branches map<string, uuid>,
    primary key (url, snapshot_date)  )
```

On a utilisé un structure de dictionnaire (Map) pour placer les branches, en supposant peu de branches par *snapshot*. On peut obtenir tous les clichés d'un dépôt.

```
select * from Snapshot_by_repository
where  url ='github.com/apache/cassandra'
```

Ou tous les clichés après une date et contenant une branche `master`.

```
select * from Snapshot_by_repository
where  url ='github.com/apache/cassandra'
and snapshot_date > '01-MARCH-2024'
and branches contains key 'master'
```

On obtient les identifiants des clichés et des branches. Maintenant, connaissant une branche, comment trouver les révisions ?

On continue la même démarche et on crée une table `Revision_by_branch`. On arrive ici à une situation épineuse : les révisions forment un graphe (à partir d'une même révision, chaque développeur peut en créer une nouvelle, et les révisions peuvent être réconciliées ensuite. On va donc représenter les parents d'une révision (en supposant qu'il n'y en a pas des centaines, ce qui semble raisonnable).

```
create table Revision_by_branch
   (branch_id uuid,
    revision_id uuid,
    parents list<uuid>,
    author varchar,
    message varchar,
    tstamp date,
    primary key (branch_id,
```

(suite sur la page suivante)

(suite de la page précédente)

```
            revision_id)
    );
```

Dans le cas d'un graphe, on ne peut parfois plus se contenter de faire une requête par table. Il faut accepter dans certains cas de faire une requête par nœud du graphe !

Commençons par les requêtes adaptées à la modélisation. On veut toutes les révisions d'une branche par Stefano.

```
select * from Revision_by_branch
where  branch_id ='bxyz'
and author = 'Stefano'
```

Tout va bien, on reste dans le cadre des requêtes efficaces. Prenons maintenant les révisions dont un parent est la révision "abcd".

```
select * from Revision_by_branch
where   branch_id ='bxyz'
and parents contains 'abcd'
```

Ca va encore, tant que l'on connait l'identifiant de la branche. Maintenant on veut aller en sens inverse et « remonter » vers les parents et ascendants de la révision "lklklk". On obtient la liste des parents avec la requête

```
select parents from Revision_by_branch
where   branch_id ='bxyz'
and revision_id = 'lklklk'
```

Il faut ensuite effectuer *une* requête par parent. Pour chaque valeur `id_du_parent`, on exécute :

```
select parents from Revision_by_branch
where   branch_id ='bxyz'
and revision_id = 'id_du_parent'
```

Soit une requête pour chaque objet, ce qui est inadapté à un système gérant des données massives. On a peut-être intérêt dans ce cas à charger une bonne fois le graphe dans l'application.

En résumé :

— La clé primaire détermine le stockage *et* les requêtes acceptables. La *clé de partitionnement* est le point d'accès, la *clé de regroupement* est l'identifiant relatif au point d'accès.

— Il faut parfois « retourner » une clé primaire si on souhaite un accès symétrique à une table existante. Ce serait le cas par exemple si on voulait trouver toutes les révisions d'un fichier *et* tous les fichiers d'une révision (cf. exercice). C'est automatisable avec une vue matérialisée mais entraine une forte redondance.

— L'imbrication d'ensembles ou de listes peut limiter le nombre d'étapes mais leur taille doit être restreinte, cf. exemple des branches.

La modélisation NoSQL, c'est du cas par cas, et les mêmes règles ne s'appliquent pas à tous les systèmes.

À titre d'exercice, je vous laisse compléter le schéma avec les tables permettant de stocker les répertoires, les entrées et leur contenu.

5.4 Exercices

Voici quelques manipulations et suggestions de recherches complémentaires.

Exercice MEP-S2-1 : expérimentez CQL

À vous de jouer : reproduisez les requêtes ci-dessus sur votre base Cassandra.

Exercice Ex-S3-1 : complétons le modèle GitHub

Notre schéma de l'étude de cas n'est pas fini. Nous voudrions pouvoir trouver tous les répertoires (*directory*) d'un dépôt pour une branche donnée.

Nous voulons également trouver toutes les entrées d'un répertoire donné.

Proposez les schémas des tables correspondant à ces besoins, et donnez les requêtes CQL correspondantes.

Question complémentaire, : comment faire pour obtenir les révisions d'un répertoire ?

Correction

Les répertoires forment une structure d'arbre, donc un graphe. Pas idéal avec Cassandra.

Voici une proposition. Il existe d'autres solutions car tout dépend des besoins et des volumétries anticipées.

La table suivante intègre les enfants et parents.

```
 create table Directory_by_revision
     (revision_id uuid,
      directory_id uuid,
      children list<uuid>,
      parent_id uuid,
      primary key (revision_id, directory_id)
)
```

On peut naviguer vers les parent ou vers les enfants, mais avec une requête à chaque fois. Il faut de plus supposer qu'il n'y a pas trop d'enfants (sous-répertoires) sinon il faut s'en remettre à la table des entrées ci-dessous en considérant qu'un sous-répertoire est une entrée d'un type particulier.

Maintenant, connaissant une *directory* pour une révision donnée, je modélise les entrées.

```
create table Entry_by_dir
     (revision_id uuid,
        directory_id uuid,
        entry_id uuid,
```

(suite sur la page suivante)

(suite de la page précédente)

```
        name varchar,
        access_rights varchar,
        primary key ( (revision_id, directory_id), entry_id)
```

Remarquez la clé de partitionnement composite. Toutes les entrées d'un même répertoire seront sur le même serveur, *mais* les répertoires d'une même révision pourront être sur des serveurs différents. Notez également qu'il y a sans doute trop d'entrées (fichiers) par répertoire pour les représenter par une liste imbriquée

On peut alors obtennir toutes les entrées d'une *directory* (pour une révision donnée).

```
select name from Entry_by_dir
where revision_id = 'a986678'
and directory_id = 'xyz'
```

On peut trouver une entrée dont le nom est `README.md` à condition de le faire dans un répertoire spécifique.

```
select content_ref from Entry_by_dir
where revision_id = 'a986678'
and directory_id = 'xyz'
and name = 'README.md'
```

À l'inverse si je veux avoir la liste des révisions d'une *directory* , je dois inverser la clé.

```
create table Revision_by_directory
     (directory_id uuid,
      revision_id uuid,
      children list<uuid>,
        primary key (revision_id,directory_id)
        )
```

On a une forte redondance si on veut les deux… Cette situation peut être gérée en créant `Revision_by_directory` comme une *vue matérialisée* (étude complémentaire à mener si cela vous intéresse).

Exercice Ex-S3-2 : déduplication

Enfin se pose la question du contenu des entrées (fichiers). Ici nous avons un problème de redondance : il s'agit de ressources souvent volumineuses, et qui ne changent souvent pas d'une révision à l'autre (et même d'une branche à l'autre). Comment éviter de dupliquer un contenu en le partageant entre des révisions et des branches ? Proposez une approche avec Cassandra.

Correction

Si on veut éviter la duplication des contenus, il faut les stocker avec leur identifiant propre, sans les placer en fonction d'une branche ou d'une révision

```
 create table Content
     (content_id uuid,
      raw_content bits,
      primary key (content_id)
)
```

Dans ce cas, on modifie la table des entrées pour référencer le contenu.

```
create table Entry_by_dir
     (revision_id uuid,
        directory_id uuid,
        entry_id uuid,
        name varchar,
        access_rights varchar,
        content_id uuid,
        primary key ( (revision_id, directory_id), entry_id)
```

On voit tout de suite le problème : en essayant d'éviter une déduplication on s'est condamné à effectuer *une* requête pour *chaque* contenu, ce qui risque d'être insupportable. Tout dépend du fait qu'on doive ou non accéder souvent aux contenus.

Exercice MEP-S2-3 : sujet d'étude, les vues matérialisées

Depuis la version 3, Cassandra propose un mécanisme de *vue matérialisé*. Etudiez la documentation à ce sujet, et montrez comment ce mécanisme peut permettre de répondre à des requêtes comme celle de l'exercice précédent.

CHAPITRE 6

Cassandra - Travaux Pratiques

Les exercices qui suivent sont à effectuer sur machine, avec Cassandra.

Après avoir lancé votre machine Cassandra (avec docker, voir chapitre *Modélisation de bases NoSQL*), vous aurez besoin d'une interface cliente pour y accéder. Pour cela, nous utiliserons **DbVisualizer**, voir le chapitre *Modélisation de bases NoSQL* pour des détails.

Si vous êtes à l'aise en ligne de commande, vous pourrez également accéder à Cassandra avec `cqlsh` qui se lance avec la commande suivante :

```
sudo docker exec -it mon-cassandra cqlsh
# ceci suppose que mon-cassandra est le nom de votre container
# Option -it pour disposer d'un terminal interactif persistant
# cqlsh pour lancer cette commande au démarrage
```

Le sujet des travaux pratiques est la mise en place d'une base de données représentant des restaurants, et des inspections de ces restaurants. Un échantillon de données est disponible ici :

— La base Restaurants au format csv (archive ZIP)

Note

Avant de vous lancer dans le travail proprement dit, vous êtes invités fortement à prendre le temps d'ouvrir cette archive zip et d'en examiner le contenu (au moins les en-têtes, pour avoir une première idée de la structure des données initiales).

Bien entendu, on supppose qu'à terme cette base contiendra tous les restaurants du monde, et toutes les inspections, ce qui justifie d'utiliser un système apte à gérer de grosses volumétries.

6.1 Partie 1 : Approche relationnelle

Nous allons étudier ici la création d'une base de données (appelée **Keyspace**), puis son interrogation. Cette première phase du TP consiste à créer la base comme si elle était relationnelle, et à effectuer des requêtes simples. Une fois les limites atteintes, nous utiliserons les spécificités de Cassandra pour aller plus loin.

6.1.1 Création de la base de données

Avant d'interroger la base de données, il nous la créer. Pour commencer :

```
CREATE KEYSPACE IF NOT EXISTS resto_NY
   WITH REPLICATION = { 'class' : 'SimpleStrategy', 'replication_factor
↪': 1};
```

Nous créons ainsi une base de données *resto_NY* pour laquelle le facteur de réplication est mis à 1, ce qui suffit dans un cadre centralisé.

Sous `csqlsh`, vous pouvez maintenant sélectionner la base de données pour vos prochaines requêtes.

```
USE resto_NY;
```

Bien entendu vous pouvez exécuter ces commandes via DbVisualizer.

Tables

Nous pouvons maintenant créer les tables (*Column Family* pour Cassandra) *Restaurant* et *Inspection* à partir du schéma suivant :

```
CREATE TABLE Restaurant (
  id INT, Name VARCHAR, borough VARCHAR, BuildingNum VARCHAR, Street␣
↪VARCHAR,
  ZipCode INT, Phone text, CuisineType VARCHAR,
  PRIMARY KEY ( id )
) ;

CREATE INDEX fk_Restaurant_cuisine ON Restaurant ( CuisineType ) ;

CREATE TABLE Inspection (
  idRestaurant INT, InspectionDate date, ViolationCode VARCHAR,
  ViolationDescription VARCHAR, CriticalFlag VARCHAR, Score INT, GRADE␣
↪VARCHAR,
  PRIMARY KEY ( idRestaurant, InspectionDate )
) ;

CREATE INDEX fk_Inspection_Restaurant ON Inspection ( Grade ) ;
```

Nous pouvons remarquer que chaque inspection est liée à un restaurant via l'identifiant de ce dernier.

Pour vérifier si les tables ont bien été créées (sous `cqlsh`).

```
DESC Restaurant;
DESC Inspection;
```

Nous pouvons voir le schéma des deux tables mais également des informations relatives au stockage dans la base *Cassandra*.

Import des données

Maintenant, nous pouvons importer les fichiers CSV pour remplir les *Column Family* :

1. Décompresser le fichier "restaurants.zip" (il contient le fichier "restaurants.csv" et "restaurants_inspections.csv")

Note : En mode console, sur le répertoire de téléchargement du fichier *restaurants.zip*, il suffit de mettre la commande :

```
unzip restaurants.zip
```

---

2. Importer un fichier CSV :

— Dans votre console (machine locale, pas docker), copier les fichiers sous « **Docker** » (container "Cassandra")

```
docker cp path-to-file/restaurants.csv    docker-container-ID:/
docker cp path-to-file/restaurants_inspections.csv    docker-
↪container-ID:/
```

Note : Le chemin « *path-to-file* » correspond à l'endroit où a été décompressé le fichier restaurants.zip

le docker-container-ID peut être récupéré grâce à la commande

```
docker ps
```

```
CONTAINER ID        IMAGE                       COMMAND                  ␣
↪    CREATED             STATUS                                       ␣
↪                                                                     ␣
↪                                                                     ␣
↪                                                 NAMES
b1fa2c7c255d        poklet/cassandra:latest     "/bin/sh -c start
↪"   6 minutes ago       Up 6 minutes
```

Ici, le container-ID est donc *b1fa2c7c255d*

---

3. Dans la console *cqlsh*, importer les fichiers "**restaurants.csv**" et "**restaurants_inspections.csv**"

```
use resto_NY ;
COPY Restaurant (id, name, borough, buildingnum, street,
                    zipcode, phone, cuisinetype)
   FROM '/restaurants.csv' WITH DELIMITER=',';
COPY Inspection (idrestaurant, inspectiondate, violationcode,
                    violationdescription, criticalflag, score,␣
↪grade)
   FROM '/restaurants_inspections.csv' WITH DELIMITER=',';
```

Note : Les fichiers sont copiés à la racine du container. Si vous changez le dossier de stockage, il faut bien sûr l'indiquer dans l'instruction précédente.

Vous pouvez vérifier l'existence des fichiers dans le container avec :

```
ls /*.csv
```

Pour vérifier le contenu des tables :

```
SELECT count(*) FROM Restaurant;
SELECT count(*) FROM Inspection;
```

Ce qui devrait vous indiquer environ 25 000 restaurants et 150 000 inspections. Nous sommes prêts !

6.1.2 Interrogation

Les requêtes qui suivent sont à exprimer avec **CQL** (pour *Cassandra Query Language*) qui est fortement inspirée de SQL. Vous trouverez la syntaxe complète ici :

<https://cassandra.apache.org/doc/latest/cql/dml.html#select>).

Notez que certaines requêtes seront refusées par CQL. Essayez de comprendre pourquoi, et trouvez un contournement. Il y en a essentiellement deux : créer un index ou utiliser `ALLOW FILTERING`.

Requêtes CQL simples

Pour la suite des exercices, exprimer en *CQL* les requêtes suivantes :

1. Liste de tous les restaurants
2. Liste des noms de restaurants
3. Nom et quartier (*borough*) du restaurant dont l'id est 41569764
4. Dates et grades des inspections de ce restaurant
5. Noms des restaurants de cuisine Française (*French*)
6. Noms des restaurants situés dans *BROOKLYN* (attribut *borough* ; si vous recevez une erreur en retour notez-la bien)

7. Grades et scores donnés pour une inspection pour le restaurant n° 41569764 avec un score d'au moins 10
8. Grades (non nuls) des inspections dont le score est supérieur à 30
9. Nombre de lignes retournées par la requête précédente

Correction

1. ```
   SELECT * FROM Restaurant;
   ```

   Note : Sous `cqlsh`, faire Ctrl+C pour annuler l'affichage de toutes les lignes

   ---

   ```
   SELECT * FROM Restaurant LIMIT 10;
   ```

   Note : Il est possible de ne montrer que 10 tuples

2. ```
   SELECT Name FROM Restaurant;
   ```

3. ```
   SELECT Name, Borough FROM Restaurant WHERE id=41569764;
   ```

4. ```
   SELECT InspectionDate, Grade FROM Inspection WHERE idRestaurant=41569764 ;
   ```

5. ```
   SELECT Name FROM Restaurant WHERE cuisineType = 'French' ;
   ```

6. ```
   SELECT Name FROM Restaurant WHERE borough='BROOKLYN' ;
   ```

   L'erreur suivante apparaît :

   ```
   Error from server: code=2200 [Invalid query] message="Cannot execute␣
   ↪this query
    as it might involve data filtering and thus may have unpredictable␣
   ↪performance.
    If you want to execute this query despite the performance␣
   ↪unpredictability,
     use ALLOW FILTERING"
   ```

   Il suffit d'ajouter *ALLOW FILTERING* à la fin de la requête pour pouvoir l'exécuter. La requête précédente fonctionnait grâce à l'index qui a été créé sur `cuisineType` et qui permet d'exécuter cette requête simplement.

   Note : Attention, dans le cadre de notre TP, cette requête est peu coûteuse. Ce n'est pas toujours le cas.

```
SELECT Name FROM Restaurant WHERE borough='BROOKLYN' ALLOW FILTERING ;
```

7. ```
   SELECT Grade, Score FROM Inspection WHERE idRestaurant=41569764 AND score >
   ↪= 10 ;
   ```

   Du fait de la présence de 2 critères dans la requête, dont un est indexé, Cassandra ne peut prédire la taille du résultat et envoi un message d'alerte :

   ```
   InvalidRequest: code=2200 [Invalid query] message=
      "Cannot execute this query as it might involve data filtering and␣
   ↪thus may have
      unpredictable performance. If you want to execute this query despite␣
   ↪the
      performance unpredictability, use ALLOW FILTERING"
   ```

   Il suffit d'ajouter *ALLOW FILTERING* à la fin de la requête pour pouvoir l'exécuter.

   Note : Attention, dans le cadre de notre TP, cette requête est peu coûteuse. Ce n'est pas toujours le cas.

   ```
   SELECT Grade, Score FROM Inspection WHERE idRestaurant=41569764
   AND score >= 10 ALLOW FILTERING ;
   ```

8. ```
   SELECT Grade FROM Inspection WHERE score > 30 AND Grade > '' ALLOW␣
   ↪FILTERING ;
   ```

   Note : L'opération « != » n'existe pas en CQL, ni la valeur "nulle". De fait, la valeur nulle pour un nombre est 0, et "" pour un texte. Ainsi, pour filtrer la valeur nulle, il faut faire « > "" ».

9. ```
   SELECT COUNT(*) FROM Inspection WHERE score > 30 AND Grade > '' ALLOW␣
   ↪FILTERING ;
   ```

   Note : L'opération affiche le résultat. Toutefois, des messages sont affichés pour montrer que de nombreuses lignes sont parcourues lors du traitement. Un message pour chaque "bucket" stocké dans la base Cassandra.

   ```

CQL Avancé

1. Pour la requête ci-dessous faites en sorte qu'elle soit exécutable sans *ALLOW FILTERING*.

```
SELECT Name FROM Restaurant WHERE borough='BROOKLYN' ;
```

2. Utilisons les deux indexes sur *Restaurant* (*borough* et *cuisineType*). Trouvez tous les noms de restaurants français de Brooklyn.
3. Utiliser la commande *TRACING ON* avant la d'exécuter à nouveau la requête pour identifier quel index a été utilisé.
4. On veut les noms des restaurants ayant au moins un grade "A" dans leurs inspections. Est-ce possible en CQL ?

Correction

1. Il faut pour cela créer un index sur "status" :

```
CREATE INDEX Restaurant_borough ON Restaurant ( borough ) ;
```

Note : Attention à ne pas exécuter la requête trop tôt après la création de l'index, le résultat ne sera pas cohérent (message d'erreur). Après quelques secondes la requête peut s'exécuter.

2.
```
SELECT Name FROM Restaurant WHERE borough = 'BROOKLYN'
AND cuisinType = 'French' ALLOW FILTERING ;
```

Note : Les deux indexes ne peuvent être utilisés conjointement, il faut donc filtrer normalement.

---

3. En utilisant la commande *TRACING ON*, nous pouvons constater l'ensemble de la *trace* de la requête (ou log d'exécution). Parmi toute la trace, la ligne suivante est disponible au début :

```
Index mean cardinalities are fk_restaurant_cuisine:1,restaurant_
↪borough:1.
Scanning with fk_restaurant_cuisine.
```

Cela nous permet de voir que l'index *fk_restaurant_cuisine* a été utilisé en priorité.

4. Naturellement, nous voudrions faire cela :

```
SELECT Name FROM Restaurant, Inspection
WHERE id = idRestaurant and Grade='A';
```

ou

```
SELECT Name FROM Restaurant
WHERE id IN (SELECT idRestaurant FROM Inspection WHERE Grade='A');
```

Sauf qu'il n'est pas possible de faire de jointures en CQL. Le langage ne donnera aucune possibilité à cette requête. Nous allons étudier la solution par la suite.

6.2 Partie 2 : modélisation spécifique NoSQL

La jointure n'est pas possible avec CQL, mais ce manque est partiellement compensé par la possibilité d'imbriquer les données pour créer des documents qui représentent, d'une certaine manière, le résultat pré-calculé de la jointure.

Cela suppose au préalable la détermination des requêtes à soumettre à la base puisque les données ne sont plus symétriques, et privilégient de fait certains types d'accès (cf. le chapitre *Etude de cas : Cassandra*). Si on ne souhaite pas enchaîner les requêtes (cf. étude de cas) il faut utiliser l'imbrication.

Notre besoin ici est de pouvoir sélectionner les restaurants en fonction de leur grade. On voudrait par exemple répondre à la question :

```
noms des restaurants ayant au obtenu moins un grade 'A' dans leurs inspections
```

À vous de définir la bonne modélisation et de vérifier qu'elle permet de satisfaire ce type de recherche.

Note : Pour importer un gros fichier de documents JSon, nous avons implémenté une application permettant de lire le document et de l'importer dans une base (présent dans le fichier `restaurants.zip`);

```
java -jar JSonFile2Cassandra [-host <host>] [-port <port>]
            [-keyspace <keyspace>] [-columnFamily <columnFamily>] [file]
```

Exemple :

```
   java -jar JSonFile2Cassandra.jar -host localhost -port 3000 \
-keyspace resto_NY -columnFamily InspectionRestaurant  \
-file InspectionsRestaurant.json
```

Voici les étapes à suivre.

1. Définir un schéma associant les restaurants et leurs inspections, en utilisant les types imbriqués, et créer la table. Le format des documents attendus est le suivant

```
{
        "idRestaurant": 40373938,
         "restaurant": {
        "name": "IHOP",
        "borough": "BRONX",
```

(suite sur la page suivante)

(suite de la page précédente)

```
        "buildingnum": "5655",
        "street": "BROADWAY",
        "zipcode": "10463",
        "phone": "7185494565",
        "cuisineType": "American"
        },
        "inspectionDate": "2016-08-16",
        "violationCode": "04L",
        "violationDescription": "Evidence of mice or live mice present in␣
↪facilitys food and/or non-food areas.",
        "criticalFlag": "Critical",
        "score": 15,
        "grade": "A"
}
```

Vous trouverez dons notre jeu de données un fichier `InspectionRestaurant.json` à importer.

Correction

1. On peut imbriquer les restaurants dans les inspections ou l'inverse. On va illustrer la première solution. Commençons par créer un type `Restaurant`.

```
CREATE TYPE Restaurant (
    Name VARCHAR, borough VARCHAR, BuildingNum VARCHAR, Street VARCHAR,
   ZipCode INT, Phone VARCHAR, CuisineType VARCHAR);

        Puis la table avec le restaurant imbriqué.

        .. code-block:: sql

CREATE TABLE InspectionRestaurant (
   idRestaurant INT, InspectionDate date, ViolationCode VARCHAR,
   ViolationDescription VARCHAR, CriticalFlag VARCHAR, Score INT,
   GRADE VARCHAR, Restaurant frozen<Restaurant>,
  PRIMARY KEY ( idRestaurant, InspectionDate )
 ) ;
```

---

2. Insérer le document-exemple ci-dessus dans la table pour vérifier que tout va bien.

Correction

Voici un exemple.

```
INSERT INTO InspectionRestaurant JSON ' {"idRestaurant":40373938,
   "restaurant": {"name":"IHOP", "borough":"BRONX", "buildingnum":"5655",
                  "street":"BROADWAY", "zipcode":"10463",
                  "phone":"7185494565", "cuisineType":"American"},
```

(suite sur la page suivante)

(suite de la page précédente)

```
  "inspectionDate":"2016-08-16",
  "violationCode":"04L",
  "violationDescription": "On voit des souris!.",
  "criticalFlag": "Critical",
  "score":15,
  "grade":"A"}';
```

3. Faire l'import avec l'utilitaire d'insertion de documents JSON.
4. Créer un index sur le *Grade* de la table **InspectionRestaurant**, puis trouver les restaurants ayant reçu le grade "A" au moins une fois.

Correction

```
create index InspectionRestaurant_grade ON InspectionRestaurant (␣
↪Grade ) ;

select Restaurant from InspectionRestaurant where grade = 'A';
```

Malheureusement, les *types* imbriqués, *map*, *set* et *list* ne peuvent être dissociés dans la clause *SELECT*. Il faut donc projeter l'ensemble des informations du restaurant. On pourra constater que le nom du restaurant apparaît autant de fois qu'il y a d'inspections. Ce qui rend cette requête un peu moins efficace car elle demande d'interroger plus de ressources. Aucun `distinct` ne peut être exécuté en dehors de la clé primaire. Bref, ce n'est pas vraiment satisfaisant.

5. Recréez la table, mais cette fois sans index, en utilisant la clé primaire pour permettre la recherche sur le grade. Insérez le document-exemple et vérifiez que la requête s'exécute sans `allow filtering`.

Correction

On supprime la table puis on la recrée.

```
CREATE TABLE InspectionRestaurant (
   idRestaurant INT, InspectionDate date, ViolationCode VARCHAR,
   ViolationDescription VARCHAR, CriticalFlag VARCHAR, Score INT,
   Grade VARCHAR, Restaurant frozen<Restaurant>,
  PRIMARY KEY ( Grade, InspectionDate )
 ) ;
```

Plus besoin de créer un index. Mais la table n'est utile que pour une seule requête : ce n'est pas terrible non plus....

CHAPITRE 7

Recherche approchée

Supports complémentaires :

— Un cours complet en ligne (en anglais) et accompagné d'un ouvrage de référence : http://www-nlp.stanford.edu/IR-book/. *Certaines parties du cours empruntent des exemples à ce livre*.

7.1 S1 : introduction à la recherche d'information

Supports complémentaires :

— Présentation: Introduction à la recherche d'information

— Vidéo de la session Introduction RI - principes

7.1.1 Qu'est ce que la recherche d'information

La *Recherche d'Information* (RI, *Information Retrieval, IR* en anglais) consiste à trouver des *documents* peu ou faiblement structurés, dans une grande *collection*, en fonction d'un *besoin d'information*. Le domaine d'application le plus connu est celui de la recherche « plein texte ». Étant donné une collection de documents constitués essentiellement de texte, comment trouver les plus pertinents en fonction d'un besoin exprimé par quelques mots-clés ? La RI développe des *modèles* pour interpréter les documents d'une part, le besoin d'information d'autre part, en vue de faire correspondre les deux, mais aussi des *techniques* pour calculer des réponses rapidement même en présence de collections très volumineuses. Enfin, des systèmes (appelés « moteurs de recherches ») fournissent des solutions sophistiquées prêtes à l'emploi.

La RI a pris une très grande importance, en raison notamment de l'émergence de vastes sources de documents que l'on peut agréger et exploiter (le Web bien sûr, mais aussi les systèmes d'information d'entreprise). Les

techniques utilisées ont également beaucoup progressé, avec des résultats spectaculaires. La RI est maintenant omniprésente dans beaucoup d'environnements informatiques, et notamment :

— la recherche sur le Web, utilisée quotidiennement par des milliards d'utilisateurs ;
— la recherche de messages dans votre boîte mail ;
— la recherche de fichiers sur votre ordinateur (*Spotlight*) ;
— la recherche de documents dans une base documentaire, publique ou privée.

Ce chapitre introduit ces différents aspects en se concentrant sur la recherche d'information appliquée à des collections de documents structurés, comprenant des parties textuelles importantes.

7.1.2 Précision et rappel

Comme le montre la définition assez imprécise donnée en préambule, la recherche d'information se donne un objectif à la fois ambitieux et relativement vague. Cela s'explique en partie par le contexte d'utilisation de ces systèmes. Avec une base de données classique, on connaît le schéma des données, leur organisation générale, avec des contraintes qui garantissent qu'elles ont une certaine régularité. En RI, les données, ou « documents », sont souvent hétérogènes, de provenances diverses, présentent des irrégularités et des variations dues à l'absence de contrainte et de validation au moment de leur création. De plus, un système de RI est souvent utilisé par des utilisateurs non-experts. À la difficulté d'interpréter le contenu des documents et de le décrire s'ajoute celle de comprendre le « besoin », exprimé souvent de manière très partielle.

D'une certaine manière, la RI vise essentiellement à prendre en compte ces difficultés pour proposer des réponses les plus pertinentes possibles. La notion de *pertinence* est centrale ici : un document est pertinent s'il satisfait le besoin exprimé. Quand on utilise SQL, on considère que la réponse est toujours exacte car définie de manière mathématique (en fonction de l'état de la base). En RI, on ne peut jamais considérer qu'un résultat, constitué d'un ensemble de documents, est exact, mais on mesure son *degré de pertinence* (c'est-à-dire les erreurs du système de recherche) en distinguant :

— les **faux positifs** : ce sont les documents *non pertinents* inclus dans le résultat ; ils ont été sélectionnés à tort. En anglais, on parle de *false positive*.
— les **faux négatifs** : ce sont les documents *pertinents* qui *ne sont pas* inclus dans le résultat. En anglais, on parle de *false negative*.

Les documents *pertinents* inclus dans le résultat sont appelés les **vrais positifs**, les documents *non pertinents* non inclus dans le résultat sont appelés les **vrais négatifs**. On le voit à ce qui vient d'être dit : on emploie le terme *positif* pour désigner ce qui a été ramené dans le résultat de recherche. À l'inverse, les documents qui sont *négatifs* ont été laissés de côté par le moteur de recherche au moment de faire correspondre des documents avec un besoin d'information. Et l'on désigne par *vrai* ce qui a été bien classé (dans le résultat ou hors du résultat).

Deux indicateurs formels, basés sur ces notions, sont couramment employés pour mesurer la qualité d'un système de RI.

La précision

La *précision* mesure la proportion des vrais positifs dans le résultat *r*. Si on note respectivement $t_p(r)$ et $f_p(r)$ le nombre de vrais et de faux positifs dans *r* (de taille $|r|$), alors

$$\text{précision} = \frac{t_p(r)}{t_p(r) + f_p(r)} = \frac{t_p(r)}{|r|}$$

Une précision de 1 correspond à l'absence totale de faux positifs. Une précision nulle indique un résultat ne contenant aucun document pertinent.

Le rappel

Le rappel mesure la proportion des documents pertinents qui sont inclus dans le résultat. Si on note $f_n(r)$ le nombre de documents faussement négatifs, alors le rappel est :

$$\frac{t_p(r)}{t_p(r) + f_n(r)}$$

Un rappel de 1 signifie que tous les documents pertinents apparaissent dans le résultat. Un rappel de 0 signifie qu'il n'y a aucun document pertinent dans le résultat.

La Fig. 7.1 illustre (en anglais) ces concepts pour bien distinguer les ensembles dont on parle pour chaque requête de recherche et les mesures associées.

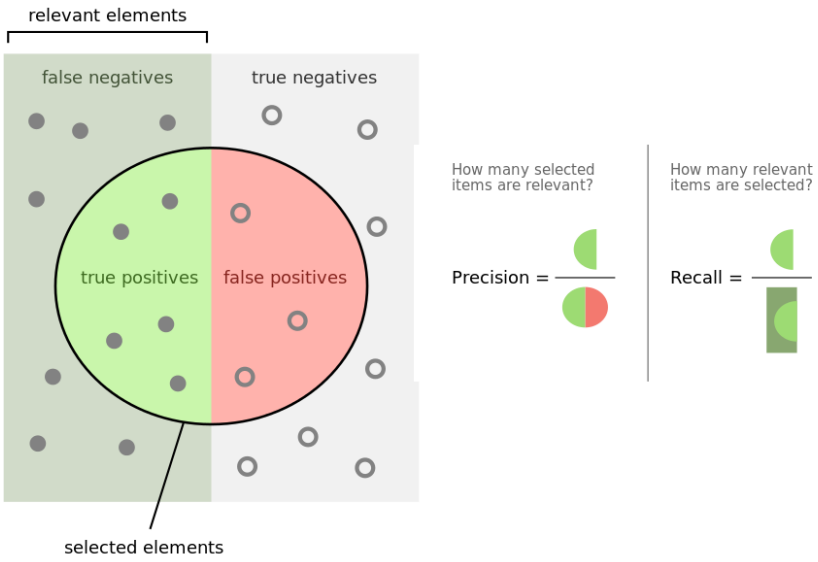


Fig. 7.1 – Vrai, faux, positifs, négatifs.

Ces deux indicateurs sont très difficiles à optimiser simultanément. Pour augmenter le rappel, il suffit d'ajouter plus de documents dans le résultat, au détriment de la précision. À l'extrême, un résultat qui contient *toute* la collection interrogée a un rappel de 1, et une précision qui tend vers 0. Inversement, si on fait le choix de ne garder dans le résultat que les documents dont on est sûr de la pertinence, la précision s'améliore mais on augmente le risque de faux négatifs (c'est-à-dire de ne pas garder des documents pertinents), et donc d'un rappel dégradé.

L'évaluation d'un système de RI est une tâche complexe et fragile car elle repose sur des enquêtes impliquant des utilisateurs. Reportez-vous à l'ouvrage de référence cité au début du chapitre (et au matériel de cours associé) pour en savoir plus.

7.1.3 La recherche plein texte

Commençons par étudier une méthode simple de recherche pour trouver des documents répondant à une recherche dite « plein texte » constituée d'un ensemble de mots-clés. On va donner à cette recherche une définition assez restrictive pour l'instant : il s'agit de trouver tous les documents contenant *tous* les mots-clés. Prenons pour exemple l'ensemble (modeste) de documents ci-dessous.

Exemple Exemple-S1-1 : des petits documents pour comprendre

- d_1 : `Le loup est dans la bergerie.`
- d_2 : `Le loup et le trois petits cochons`
- d_3 : `Les moutons sont dans la bergerie.`
- d_4 : `Spider Cochon, Spider Cochon, il peut marcher au plafond.`
- d_5 : `Un loup a mangé un mouton, les autres loups sont restés dans la bergerie.`
- d_6 : `Il y a trois moutons dans le pré, et un mouton dans la gueule du loup.`
- d_7 : `Le cochon est à 12 le Kg, le mouton à 10 E/Kg`
- d_8 : `Les trois petits loups et le grand méchant cochon`

Et ainsi de suite. Supposons que l'on recherche **tous les documents parlant de loups, de moutons mais pas de bergerie** (c'est le besoin). Une solution simple consiste à parcourir tous les documents et à tester la présence des mots-clés. Ce n'est pas très satisfaisant car :

- c'est potentiellement long, pas sur notre exemple bien sûr, mais en présence d'un ensemble volumineux de documents ;
- le critère « pas de bergerie » n'est pas facile à traiter ;
- les autres types de recherche (« le mot "loup" doit être *près* du mot "mouton" ») sont difficiles ;
- *quid* si je veux *classer* par pertinence les documents trouvés ?

On peut faire mieux en créant une structure compacte sur laquelle peuvent s'effectuer les opérations de recherche. Les données sont organisées dans une matrice (dite d'incidence) qui représente l'occurrence (ou non) de chaque mot dans chaque document. On peut, au choix, représenter les mots en colonnes et les documents en ligne, ou l'inverse. La première solution semble plus naturelle (chaque fois que j'insère un nouveau document, j'ajoute une ligne dans la matrice). Nous verrons plus loin que la seconde représentation, appelée *matrice inversée*, est en fait beaucoup plus appropriée pour une recherche efficace.

Terme, vocabulaire.

Nous utilisons progressivement la notion de *terme* (*token* en anglais) qui est un peu différente de celle de « mot ». Le *vocabulaire*, parfois appelé *dictionnaire*, est l'ensemble des termes sur lesquels on peut poser une requête. Ces notions seront développées plus loin.

Commençons par montrer une matrice d'incidence avec les documents en ligne. On se limite au vocabulaire suivant : {« loup », « mouton », « cochon », « bergerie », « pré », « gueule »}.

Tableau 7.1 – La matrice d'incidence

	loup	mouton	cochon	bergerie	pré	gueule
d_1	1	0	0	1	0	0
d_2	1	0	1	0	0	0
d_3	0	1	0	1	0	0
d_4	0	0	1	0	0	0
d_5	1	1	0	1	0	0
d_6	1	1	0	0	1	1
d_7	0	1	1	0	0	0
d_8	1	0	1	0	0	0

Cette structure est parfois utilisée dans les bases de données sous le nom *d'index bitmap*. Elle permet de répondre à notre besoin de la manière suivante :

- On prend les *vecteurs d'incidence* de chaque terme contenu dans la requête, soit les colonnes dans notre représentation :
  - Loup : 11001101
  - Mouton : 00101110
  - Bergerie : 01010011
- On fait un `et` (logique) sur les vecteurs de *Loup* et *Mouton* et on obtient 00001100
- Puis on fait un `et` du résultat avec le *complément* du vecteur de *Bergerie* (01010111)
- On obtient 00000100, d'où on déduit que la réponse est limitée au document d_6, puisque la 6e position est la seule où il y a un "1".

7.1.4 Les index inversés

Si on imagine maintenant des données à grande échelle, on s'aperçoit que cette approche un peu naïve soulève quelques problèmes. Posons par exemple les hypothèses suivantes :

- Un million de documents, mille mots chacun en moyenne (ordre de grandeur d'une encyclopédie en ligne bien connue)
- Disons 6 octets par mot, soit 6 Go (pas très gros en fait !)
- Disons 500 000 termes *distincts* (ordre de grandeur du nombre de mots dans une langue comme l'anglais)

La matrice a 1 000 000 de lignes, 500 000 colonnes, donc 500×10^9 bits, soit 62 GO. Elle ne tient pas en mémoire de nombreuses machines, ce qui va beaucoup compliquer les choses.... Comment faire mieux ?

Une première remarque est que nous avons besoin des vecteurs pour les *termes* (mots) pour effectuer des opérations logiques (`and`, `or`, `not`) et il est donc préférable *d'inverser* la matrice pour disposer les termes en ligne (la représentation est une question de convention : ce qui est important c'est qu'au niveau de la structure de données, le vecteur d'incidence associé à un *terme* soit stocké contigument en mémoire et donc accessible simplement et rapidement). On obtient la structure de la Fig. 7.2 pour notre petit ensemble de documents.

Seconde remarque : la matrice d'incidence est *creuse*. En effet, sur chaque ligne, il y a au plus 1000 « 1 » (cas extrême où tous les termes du document sont distincts). Donc, dans l'ensemble de la matrice, il n'y a au plus que 10^9 positions avec des 1, soit un sur 500. Il est donc tout à fait inutile de représenter les cellules avec des 0. On obtient une structure appelée *index inversé* qui est essentiellement l'ensemble des lignes de la matrice d'incidence, dans laquelle on ne représente que les cellules correspondant à *au moins* une occurrence du terme (en ligne) dans le document (en colonne).

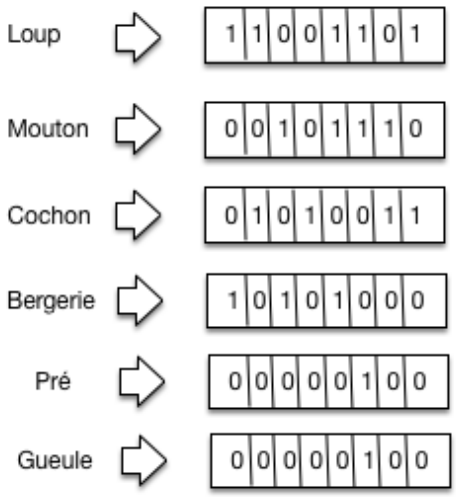


Fig. 7.2 – Inversion de la matrice

Important : Comme on ne représente plus l'ensemble des colonnes pour chaque ligne, il faut indiquer, pour chaque cellule, à quelle colonne elle correspond. Pour cela, on place dans les cellules *l'identifiant* du document (*docId*). **De plus chaque liste est triée sur l'identifiant du document**.

La Fig. 7.3 montre un exemple d'index inversé pour trois termes, pour une collection importante de documents consacrés à nos animaux familiers. À chaque terme est associé une liste inverse.

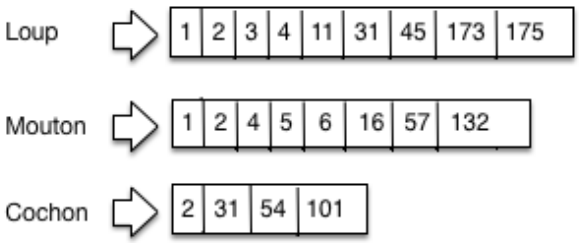


Fig. 7.3 – Un index inversé

On parle donc de cochon dans les documents 2, 31, 54 et 101 (notez que les documents sont ordonnés — très important). Il est clair que la taille des listes inverses varie en fonction de la fréquence d'un terme dans la collection.

La structure d'index inversé est utilisée dans *tous* les moteurs de recherche. Elle présente d'excellentes propriétés pour une recherche efficace, avec en particulier des possibilités importantes de compression des listes associées à chaque terme.

Vocabulaire

Le *dictionnaire* (*dictionary*) est l'ensemble des termes de l'index inversé ; le *répertoire* est la structure qui

associe chaque terme à l'adresse de la liste inversée (*posting list*) associée au terme. Enfin on ne parle plus de cellule (la matrice a disparu) mais *d'entrée* pour désigner un élément d'une liste inverse.

En principe, le répertoire est toujours en mémoire, ce qui permet de trouver très rapidement les listes impliquées dans la recherche. Les listes inverses sont, autant que possible, en mémoire, sinon elles sont compressées et stockées dans des fichiers (contigus) sur le disque.

7.1.5 Opérations de recherche

Étant donné cette structure, recherchons les documents parlant de loup et de mouton. On ne peut plus faire un `et` sur des tableaux de bits de taille fixe puisque nous avons maintenant des listes de taille variable. L'algorithme employé est une *fusion* (»*merge* ») de liste triées. C'est une technique très efficace qui consiste à parcourir en parallèle et séquentiellement des listes, en une seule fois. Le parcours unique est permis par le tri des listes sur un même critère (l'identifiant du document).

La Fig. 7.4 montre comment on traite la requête `loup et mouton`. On maintient deux curseurs, positionnés au départ au début de chaque liste. L'algorithme compare les valeurs des docId contenues dans les cellules pointées par les deux curseurs. On compare ces deux valeurs, puis :

- (choix A) si elles sont égales, on a trouvé un document : on place son identifiant dans le résultat, à gauche ; *on avance les deux curseurs d'un cran*
- (choix B) sinon, on avance le curseur pointant sur la cellule dont la valeur est la plus petite, jusqu'à atteindre ou dépasser la valeur la plus grande.

La Fig. 7.4 se lit de gauche à droite, de haut en bas. On applique tout d'abord deux fois le choix A : les documents 1 et 2 parlent de loup et de mouton. À la troisième étape, le curseur du haut (loup) pointe sur le document 3, le pointeur du bas (mouton) sur le document 4. On applique le choix B en déplaçant le curseur du haut jusqu'à la valeur 4.

Et ainsi de suite. Il est clair *qu'un seul parcours suffit*. La recherche est linéaire, et l'efficacité est garantie par un parcours séquentiel d'une structure (la liste) très compacte. De plus, il n'est pas nécessaire d'attendre la fin du parcours de toutes les listes pour commencer à obtenir le résultat. L'algorithme est résumé ci-dessous.

```
// Fusion de deux listes l1 et l2
function Intersect($l1, $l2)
{
  $résultat = [];
  // Début de la fusion des listes
  while ($l1 != null and $l2 != null) {
    if ($l1.docId == $l2.docId) {
      // On a trouvé un document contenant les deux termes
      $résultat += $l1.docId;
      // Avançons sur les deux listes
      $l1 = $l1.next; $l2 = $l2.next;
    }
    else if ($l1.docId < $l2.docId) {
      // Avançons sur l1
      $l1 = $l1.next;
    }
```

(suite sur la page suivante)

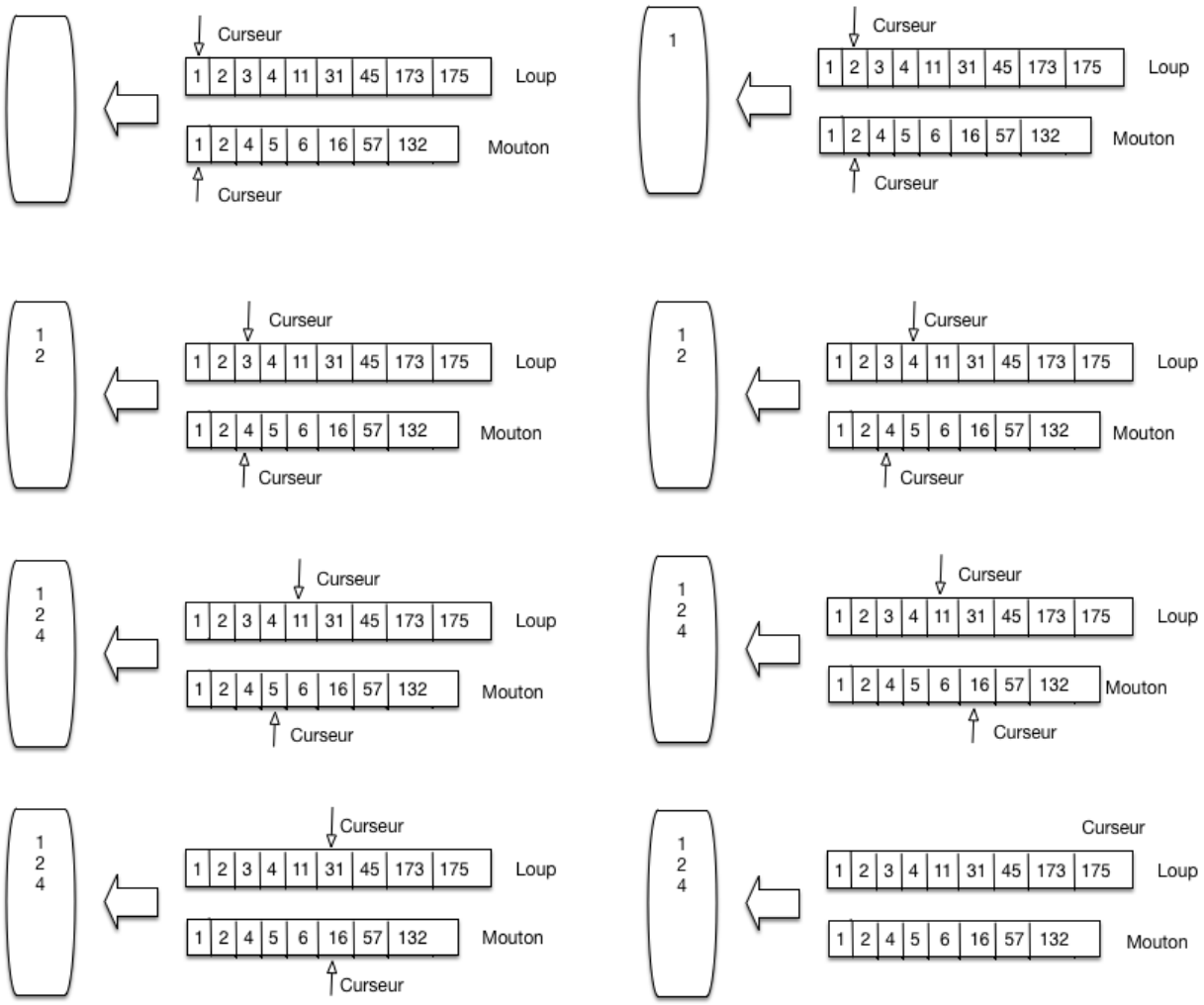


Fig. 7.4 – Parcours linéaire pour la fusion de listes triées

(suite de la page précédente)

```
    else {
       // Avançons sur l2
      $l2 = $l2.next;
    }
  }
}
```

C'est l'algorithme de base de la recherche d'information. Dans la version présentée ici, on satisfait des requêtes dites *booléennes* : l'appartenance d'un document au résultat est binaire, et il n'y a aucun classement par pertinence.

À partir de cette technique élémentaire, on peut commencer à raffiner, pour aboutir aux techniques sophistiquées visant à capturer au mieux le besoin de l'utilisateur, à trouver les documents qui satisfont ce besoin et à les classer par pertinence. Pour en arriver là, tout un ensemble d'étapes que nous avons ignorées dans la présentation abrégée qui précède sont nécessaires. Nous les reprenons dans ce qui suit.

7.1.6 Quiz

7.2 S2 : L'analyse de documents

Supports complémentaires :

- Présentation: Recherche d'information - analyse de documents
- Vidéo : Pré-traitement des documents textuels

En présence d'un document textuel un tant soit peu complexe, on ne peut pas se contenter de découper plus ou moins arbitrairement en mots sans se poser quelques questions et appliquer un pré-traitement du texte. Les effets de ce pré-traitement doivent être compris et maîtrisés : ils influent directement sur la précision et le rappel. Quelques exemples simples pour s'en convaincre :

- si on cherche les documents contenant le mot « loup », on s'attend généralement à trouver ceux contenant « loups », « Loup », « louve » ; il faut donc, quand on conserve un document dans Elasticsearch, qu'il soit en mesure de mettre ces différentes formes dans *le même index inversé* ;
- si on ne normalise pas (on conserve les majuscules et les pluriels), on va dégrader le rappel, puisqu'un utilisateur saisissant le mot-clef « loup » ne trouvera pas les documents dans lesquels ce terme apparaît *seulement* sous la forme « Loup » ou « loups » ;
- on le comprend immédiatement avec le cas de « loup / louve », il faut une connaissance experte de la langue pour décider que « louve » et « loup » doivent être associés, ce qui requiert une transformation qui dépend de la langue et nécessite une analyse approfondie du contenu ;
- inversement, si on normalise (retrait des accents, par exemple) « cote », « côte », « côté », on va unifier des mots dont le sens est différent, et on va diminuer la précision.

En fonction des caractéristiques des documents traités, des utilisateurs de notre système de recherche, il faudra trouver un bon équilibre, aucune solution n'étant parfaite. C'est de l'art et du réglage…

L'analyse se compose de plusieurs phases :

- *Tokenization* : découpage du texte en « termes ».

— *Normalisation* : identification de toutes les variantes d'écritures d'un même terme et choix d'une règle de normalisation (que faire des majuscules ? acronymes ? apostrophes ? accents ?).
— *Stemming* (« racinisation ») : rendre la racine des mots pour éviter le biais des variations autour d'un même sens (auditer, auditeur, audition, etc.)
— *Stop words* (« mots vides »), comment éliminer les mots très courants qui ne rendent pas compte de la signification propre du document ?

Ce qui suit est une brève introduction, essentiellement destinée à comprendre les outils prêts à l'emploi que nous utiliserons ensuite. Remarquons en particulier que les étapes ci-dessus sont parfois décomposées en sous-étapes plus fines avec des algorithmes spécifiques (par exemple, un pour les accents, un autre pour les majuscules). L'ordre que nous donnons ci-dessus est un exemple, il peut y avoir de légères variations. Enfin, notez bien que **le texte transformé dans une étape sert de texte d'entrée à la transformation suivante** (nous y reviendrons dans la partie pratique).

7.2.1 Tokenisation et normalisation

Un *tokenizer* prend en entrée un texte (une chaîne de caractères) et produit une séquence de *tokens*. Il effectue donc un traitement purement lexical, consistant typiquement à éliminer les espaces blancs, la ponctuation, les liaisons, etc., et à identifier les « mots ». Des transformations peuvent également intervenir (suppression des accents par exemple, ou normalisation des acronymes - U.S.A. devient USA).

La tokenization est très fortement dépendante de la langue. La première chose à faire est d'identifier cette dernière. En première approche on peut examiner le jeu de caractères (Fig. 7.5).

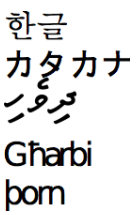


Fig. 7.5 – Quelques jeux de caractères … exotiques

Il s'agit respectivement du : Coréen, Japonais, Maldives, Malte, Islandais. Ce n'est évidemment pas suffisant pour distinguer des langues utilisant le même jeu de caractères. Une extension simple est d'identifier les *séquences de caractères fréquents*, (*n*-grams). Des bibliothèques fonctionnelles font ça très bien (e.g., Tika, http://tika.apache.org)

Une fois la langue identifiée, on divise le texte en *tokens* (« mots »). Ce n'est pas du tout aussi facile qu'on le dirait !

— Dans certaines langues (Chinois, Japonais), les mots *ne sont pas* séparés par des espaces.
— Certaines langues s'écrivent de droite à gauche, de haut en bas.
— Que faire (et de manière *cohérente*) des acronymes, élisions, nombres, unités, URL, email, etc.
— Que faire des *mots composés* : les séparer en *tokens* ou les regrouper en un seul ? Par exemple :
  — Anglais : *hostname*, *host-name* et *host name*, …
  — Français : Le Mans, aujourd'hui, pomme de terre, …
  — Allemand : *Levensversicherungsgesellschaftsangestellter* (employé d'une société d'assurance vie).

Pour les majuscules et la ponctuation, une solution simple est de normaliser systématiquement (minuscules, pas de ponctuation). Ce qui donnerait le résultat suivant pour notre petit jeu de données.

— d_1 : `le loup est dans la bergerie`
— d_2 : `le loup et les trois petits cochons`
— d_3 : `les moutons sont dans la bergerie`
— d_4 : `spider cochon spider cochon il peut marcher au plafond`
— d_5 : `un loup a mangé un mouton les autres loups sont restés dans la bergerie`
— d_6 : `il y a trois moutons dans le pré et un mouton dans la gueule du loup`
— d_7 : `le cochon est à 12 euros le kilo le mouton à 10 euros kilo`
— d_8 : `les trois petits loups et le grand méchant cochon`

7.2.2 Stemming (racine), lemmatisation

La racinisation consiste à *confondre* toutes les formes d'un même mot, ou de mots apparentés, en une seule *racine*. Le *stemming morphologique* retire les pluriels, marque de genre, conjugaisons, modes, etc. Le *stemming lexical* fond les termes proches lexicalement : « politique, politicien, police (?) » ou « université, universel, univers (?) ». Ici, le choix influe clairement sur la précision et le rappel (plus d'unification favorise le rappel au détriment de la précision).

La racinisation est très dépendante de la langue et peut nécessiter une analyse linguistique complexe. En anglais, *geese* est le pluriel de *goose*, *mice* de *mouse* ; les formes masculin / féminin en français n'ont parfois rien à voir (« loup / louve ») mais aussi (« cheval / jument » : parle-t-on de la même chose ?) Quelques exemples célèbres montrent les difficultés d'interprétation :

— « Les poules du couvent couvent » : où est le verbe, où est le substantif ?
— « La petite brise la glace » : idem.

Voici un résultat possible de la racinisation pour nos documents.

— d_1 : `le loup etre dans la bergerie`
— d_2 : `le loup et les trois petit cochon`
— d_3 : `les mouton etre dans la bergerie`
— d_4 : `spider cochon spider cochon il pouvoir marcher au plafond`
— d_5 : `un loup avoir manger un mouton les autre loup etre rester dans la bergerie`
— d_6 : `il y avoir trois mouton dans le pre et un mouton dans la gueule du loup`
— d_7 : `le cochon etre a 12 euro le kilo le mouton a 10 euro kilo`
— d_8 : `les trois petit loup et le grand mechant cochon`

Il existe des procédures spécialisées pour chaque langue. En anglais, l'algorithme Snowball de Martin Porter fait référence et est toujours développé aujourd'hui. Il a connu des déclinaisons dans de nombreuses langues, dont le français, par un travail collaboratif.

7.2.3 Mots vides et autres filtres

Un des filtres les plus courants consiste à retire les mots porteurs d'une information faible (»*stop words* » ou « mots vides ») afin de limiter le stockage.

— Les articles : *le*, *le*, *ce*, etc.
— Les verbes « fonctionnels » : *être*, *avoir*, *faire*, etc.
— Les conjonctions : *et*, *ou*, etc.
— et ainsi de suite.

Le choix est délicat car, d'une part, ne pas supprimer les mots vides augmente l'espace de stockage nécessaire (et ce d'autant plus que la liste associée à un mot très fréquent est très longue), d'autre part les éliminer peut diminuer la pertinence des recherches (« pomme de terre », « Let it be », « Stade de France »).

Parmi les autres filtres, citons en vrac :

— *Majuscules / minuscules*. On peut tout mettre en minuscules, mais on perd alors la distinction nom propre / nom commu,, par exemple Lyonnaise des Eaux, Société Générale, Windows, etc.
— *Acronymes*. CAT = *cat* ou *Caterpillar Inc.* ? M.A.A.F ou MAAF ou Mutuelle … ?
— *Dates, chiffres*. Monday 24, August, 1572 – 24/08/1572 – 24 août 1572 ; 10000 ou 10,000.00 ou 10,000.00

Dans tous les cas, les même règles de transformation s'appliquent aux documents ET à la requête. Voici, au final, pour chaque document la liste des *tokens* après application de quelques règles simples.

— d_1 : `loup etre bergerie`
— d_2 : `loup trois petit cochon`
— d_3 : `mouton etre bergerie`
— d_4 : `spider cochon spider cochon pouvoir marcher plafond`
— d_5 : `loup avoir manger mouton autre loup etre rester bergerie`
— d_6 : `avoir trois mouton pre mouton gueule loup`
— d_7 : `cochon etre douze euro kilo mouton dix euro kilo`
— d_8 : `trois petit loup grand mechant cochon`

7.2.4 Quiz

7.3 S3 : recherche avec classement

Supports complémentaires :

— Présentation: Recherche avec classement
— Vidéo de la session « Principes de la recherche avec classement »

Le mode d'interrogation classique en base de données consiste à exprimer des critères de recherche et à produire en sortie les données qui satisfont *exactement* ces critères. En d'autres termes, dans le cas d'une base documentaire, on peut déterminer qu'un document est ou n'est pas dans le résultat. Cette décision univoque correspond au modèle de requêtes Booléennes présenté précécemment.

7.3.1 Notions de base : espace métrique, distance et similarité

Dans le cas typique d'un document textuel, il est illusoire de vouloir effectuer une recherche exacte en tentant de produire comme critère la chaîne de caractères complète du document, voire même une sous-chaîne. La même remarque s'applique à des objets complexes ou multimédia (images, vidéos, etc.). La logique est alors plutôt d'interpréter la requête comme un *besoin*, et d'identifier les documents les plus « proches » du besoin. En recherche d'information (*RI*), on raisonne donc plutôt en terme de *pertinence* pour décider si un document fait ou non partie du résultat d'une recherche. La formalisation de ces notions de besoin et de pertinence est au centre des méthodes de RI.

En particulier, la formalisation de la pertinence consiste à en donner une expression *quantitative*. Pour cela, l'approche classique consiste

— à définir un espace métrique E doté d'une fonction de *distance* m_E,

— à définir une fonction f de l'espace des documents vers E ; cette fonction s'applique également à la requête q, vue comme un document ;
— enfin, on mesure la pertinence (ou *similarité*) entre deux documents d_1 et d_2 comme l'inverse de la distance entre $f(d_1)$ et $f(d_2)$.

$$sim(d, q) = \frac{1}{m_E(f(d_1), f(d_2))}$$

On obtient une mesure de la similarité, ou *score*, mesurant la proximité de deux documents (ou, plus précisément, des vecteurs représentant ces documents). Il reste à interpréter la requête q comme un document et à évaluer $sim(f(d), f(q))$ pour chaque document d afin d'évaluer la pertinence d'un document vis-à-vis du besoin exprimé par la requête.

Avec cette approche, contrairement aux requêtes Booléennes, on ne peut souvent plus dire de manière stricte qu'un document d n'appartient pas au résultat d'une recherche. Il est plus correct de dire que d est plus ou moins *pertinent*. Cela rend les résultats beaucoup plus riches, et offre à l'utilisateur la possibilité d'éviter le « tout ou rien » de l'approche Booléenne.

Note : Reportez-vous au chapitre *Recherche approchée* pour une discussion introductive sur les notions de faux et vrais positifs, de rappel et de précision.

La contrepartie de cette flexibilité est l'abondance des candidats potentiels et la nécessité de les *classer* en fonction de leur pertinence/score. Le rôle d'un moteur de recherche consiste donc (conceptuellement), pour chaque requête q, à calculer le score $s_i = sim(q, d_i)$ pour chaque document d_i de la collection, à trier tous les documents par ordre décroissant des scores et à présenter ce classement à l'utilisateur.

Important : Il faut ajouter une contrainte de temps : le résultat doit être disponible en quelques dizièmes de secondes, même dans le cas de collections comprenant des millions, des centaines de millions ou des milliards de documents (cas du Web). La performance de la recherche s'appuie sur les structures d'index inversés et des optimisations fines qui dépassent le cadre de ce cours : reportez-vous, par exemple, au livre en ligne mentionné en début de chapitre.

En pratique, le calcul du score pour *tous* les documents n'est bien sûr pas faisable (ni souhaitable d'ailleurs), et le moteur de recherche dispose de structures de données qui vont lui permettre de déterminer rapidement les documents ayant le meilleur score. Ces documents (disons les 10 ou 20 premiers, typiquement) sont présentés à l'utilisateur, et le reste de la liste est calculé à la demande si besoin est. Dans le cas d'une interface interactive (et si le classement est réellement pertinent vis-à-vis du besoin), il est rare qu'un utilisateur aille au-delà de la seconde, voire même de la première page.

Vocabulaire. En résumé, voici les points à retenir.

1. on effectue des calculs dans un espace métrique, le plus souvent un espace vectoriel ;
2. pour chaque document, on produit un objet de l'espace métrique, appelé *descripteur*, qui a le plus souvent la forme d'un *vecteur* (*features vector*) ;
3. on applique le même traitement à la requête q pour obtenir un descripteur v_q ;
4. le *score* est une mesure de la pertinence d'un document d_i par rapport au *besoin* exprimé par la requête q ;
5. le calcul du score s'appuie sur la mesure de la *distance* entre le descripteur de d_i et celui de q.

Ces principes étant posés, voyons une application concrète (quoique simplifiée pour l'instant, et peu satisfaisante en pratique) au cas de la recherche plein texte.

7.3.2 Application à la recherche plein texte

Important : La méthode présentée ci-dessous n'est qu'une première approche, à la fois très simplifiée et présentant de sévères défauts par rapport à la méthode générale que nous présenterons ensuite.

Pour commencer, on suppose connu l'ensemble $V = \{t_1, t_2, \cdots, t_n\}$ de tous les termes utilisables pour la rédaction d'un document et on définit *E* comme l'espace de tous les vecteurs constitués de *n* coordonnées valant soit 0, soit 1 (soit, en notation mathématique, $E = \{0, 1\}^n$). Ce sont nos descripteurs.

Par exemple, on considère que le vocabulaire est {« papa », « maman », « gateau », « chocolat », « haut », « bas »}. Nos vecteurs sont donc constitués de 6 coordonnées valant soit 0, soit 1. Il faut alors définir la fonction f qui associe un document *d* à son descripteur (vecteur) $v = f(d)$. Voici cette définition :

$$v[i] = \begin{cases} 1 \text{ si } d \text{ contient le terme } t_i \\ 0 \text{ sinon} \end{cases}$$

C'est exactement la représentation que nous avons adoptée jusqu'à présent. À chaque document on associe une séquence (un vecteur) de 1 ou de 0 selon que le terme t_i est présent ou non dans le document.

Prenons un premier exemple. Le document d_{maman} :

```
"maman est en haut, qui fait du gateau"
```

sera représenté par le descripteur/vecteur $[0, 1, 1, 0, 1, 0]$. Je vous laisse calculer le vecteur de ce second document d_{papa} :

```
"papa est en bas, qui fait du chocolat"
```

Note : Remarquez que l'on choisit délibérément d'ignorer certains mots considérés comme peu représentatifs du contenu du document. Ce sont les *stop words* (mots inutiles) comme « est », « en », « qui », « fait », etc.

Note : Remarquez également que *l'ordre des mots* dans le document est ignoré par cette représentation qui considère un texte comme un « sac de mots » (*bag of words*). Si on prend un document contenant les deux phrases ci-dessus, on ne sait plus distinguer si papa est en haut ou en bas, ou si maman fait du gateau ou du chocolat.

Maintenant, contrairement à la recherche Booléenne dans laquelle on vérifiait que, pour chaque terme requis, la position correspondante dans le vecteur d'un document était à 1, on va appliquer une fonction de distance sur les vecteurs afin d'obtenir une valeur entre 0 et 1 mesurant la pertinence. Un candidat naturel est la distance Euclidienne dont nous rappelons la définition, pour deux vecteurs v_1 et v_2.

$$E(v_1, v_2) = \sqrt{(v_1^1 - v_2^1)^2 + (v_1^2 - v_2^2)^2 + \cdots + (v_1^n - v_2^n)^2}$$

Et la similarité est l'inverse de la distance.

$$sim(v_1, v_2) = \begin{cases} \infty \text{ si } v_1^i = v_2^i \text{ pour tout } i \\ \frac{1}{E(v_1, v_2)} \text{ sinon} \end{cases}$$

On obtient une mesure de la similarité, ou *score*, mesurant la proximité de deux documents (ou, plus précisément, des vecteurs représentant ces documents).

Il reste à interpréter la requête comme un document et à évaluer $sim(f(d), f(q))$ pour chaque document d et la requête q pour évaluer la pertinence d'un document vis-à-vis du besoin exprimé par la requête. La requête q par exemple :

```
"maman haut chocolat"
```

est donc transformée en un vecteur $v_q = [0, 1, 0, 1, 1, 0]$. Pour le document d_{maman}, on obtient un score de $sim(v_q, d_{maman}) = \frac{1}{\sqrt{2}}$. À vous de calculer $sim(v_q, d_{papa})$ et de vérifier que ce score est moins elevé, ce qui correspond à notre intuition. Notez quand même :

- que « chocolat », un des mots-clés de q, n'apparaît pas dans le document d_{maman}, malgré tout classé en tête ;
- qu'un seul terme est commun entre d_{papa} et q, et que le document est quand même (bien) classé ;
- qu'un document comme « `bébé mange sa soupe` » obtiendrait un score non nul (lequel ?) et serait donc lui aussi classé (si on ne met pas de borne à la valeur du score).

Une différence concrète très sensible (illustrée ci-dessus) avec les requêtes Booléennes est qu'il n'est pas nécessaire qu'un document contienne *tous* les termes de la requête pour que son score soit différent de 0.

Les limites de l'approche présentée jusqu'ici sont explorées dans des exercices. La méthode beaucoup plus robuste détaillée dans la prochaine section montrera aussi, par contraste, coment des facteurs comme la taille des documents, la taille du vocabulaire, le nombre d'occurrences d'un terme dans un document et la rareté de ce terme influent sur la précision du classement.

7.3.3 Quiz

7.4 S4 : recherche plein texte

Supports complémentaires :

- Présentation: Recherche plein texte
- Vidéo de la session classement dans la recherche plein texte

Nous reprenons maintenant un approche plus solide pour la recherche plein texte, qui pour l'essentiel s'appuie sur les principes précédents, mais corrige les gros inconvénients que vous avez dû découvrir en complétant les exercices.

La méthode présentée dans ce qui suit est maintenant bien établie et utilisée, à quelques raffinements près, comme approche de base par tous les moteurs de recherche. Résumons (une nouvelle fois) :

- les documents (textuels) sont vus comme des *sacs de mots*, l'ordre entre les mots étant ignoré ; on ne fera pas de différence entre un document qui dit que le mouton est dans la gueule du loup et un autre qui prétend que le loup est dans la gueule du mouton (?) ;
- quand on parle de « mots », il faut bien comprendre : les termes obtenus par application d'un processus de simplification / normalisation lexicale déjà étudié ;

— un descripteur est associé à chaque document, dans un espace doté d'une fonction de distance qui permet d'estimer la similarité entre deux documents ;
— enfin, la *requête* elle-même est vue comme un document, et placée donc dans le même espace ; on considère donc ici les requêtes exprimées comme une liste de mots, sans aucune construction syntaxique complémentaire.

Ceci posé, nous nous concentrons sur la fonction de similarité.

Note : « mot » et « terme » sont utilisés comme des synonymes à partir de maintenant.

7.4.1 Le poids des mots

Dans l'approche très simplifiée présentée ci-dessus, nous avons traité les mots uniformément, selon une approche Booléenne : 1 si le mot est présent dans le document, 0 sinon.

Pour obtenir des résultats de meilleure qualité, on va prendre en compte les degrés de pertinence et d'information portés par un terme, selon deux principes :

1. plus un terme est présent dans un document, plus il est représentatif du contenu du document ;
2. moins un terme est présent dans une collection, et plus une occurrence de terme est significative.

De plus, on va tenter d'éliminer le biais lié à la longueur variable des documents. Il est clair que plus un document est long, et plus il contiendra de mots et de répétitions d'un même mot. Si on n'introduit pas un élément correctif, la longueur des documents a donc un impact fort sur le résultat d'une recherche et d'un classement, ce qui n'est pas forcément souhaitable.

En tenant compte de ces facteurs, on aboutit à affecter un *poids* à chaque mot dans un document, et à représenter ce dernier comme un vecteur de paires *(mot, poids)*, ce qui peut être considéré comme une représentation compacte du contenu du document. La méthode devenue classique pour déterminer le poids est de combiner la *fréquence des termes* et la *fréquence inverse (des termes) dans les documents*, ce que l'on abrège par *tf* (*term frequency*) et *idf* (*inverse document frequency*).

La fréquence des termes

La *fréquence d'un terme t* dans un document *d* est le nombre d'occurrences de *t* dans *d*.

$$\mathrm{tf}(t, d) = n_{t,d}$$

où $n_{t,d}$ est le nombre d'occurrences de *t"* dans *d*. On représente donc un document par la liste des termes associés à leur fréquence. Si on prend une collection de documents, dans laquelle certains termes apparaissent dans plusieurs documents (ce qui est le cas normal), on peut représenter les *tf* par une matrice semblable à la matrice d'incidence déjà vue dans la cas Booléen. Celle ci-dessous correspond à une collection de trois documents, avec un vocabulaire constitué de 4 termes.

terme	d1	d2	d3
voiture	27	15	24
marais	3	20	0
serpent	0	25	29
baleine	14	0	17
total	*44*	*60*	*70*

Normalisation des *tf*

Il y a donc 44 termes dans le document *d1*, 60 dans le *d2* et 70 dans le *d3*. Il est clair qu'il est difficile de comparer dans l'absolu des fréquences de terme pour des documents de longueur très différentes, car la probabilité qu'un terme apparaisse souvent augment avec la taille du document.

Pour s'affranchir de l'effet induit par la taille (qui amènerait à classer systématiquement en tête les documents longs), on *normalise* donc les valeurs des *tf*. Une méthode simple est, par exemple, de diviser chaque *tf* par le nombre total de termes dans le document, ce qui donnerait la matrice suivante :

terme	d1	d2	d3
voiture	27/44	15/60	24/70
marais	3/44	20/60	0
serpent	0	25/60	29/70
baleine	14/44	0	17/70

Un calcul un peu plus sophistiqué consiste à considérer l'ensemble une colonne de la matrice d'incidence comme un vecteur dans un espace multidimensionel. Dans notre exemple l'espace est de dimension 4, chaque axe correspondant à l'un des termes. Le vecteur de *d1* est (27, 3, 0, 14), celui de *d2* (15,20, 25,0), etc. Pour normaliser ces vecteurs, on va diviser leurs coordonnées par leur norme euclidienne. Rappel : la norme d'un vecteur $v = (x_1, x_2, \cdots, x_n)$ est

$$||v|| = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}$$

La norme du vecteur *d1* est donc :

$$\sqrt{27^2 + 3^2 + 14^2} = 30,56$$

Celle de *d2* :

$$\sqrt{15^2 + 20^2 + 25^2} = 35,35$$

Celle de *d3* :

$$\sqrt{24^2 + 29^2 + 17^2} = 41,3$$

On voit que le résultat est assez différent de la simple somme des *tf*. L'interprétation des descripteurs de documents comme des vecteurs est à la base d'un calcul de similarité basé sur les cosinus, que nous détaillons ci-dessous.

La fréquence inverse dans les documents

La fréquence inverse d'un terme dans les documents (*inverse document frequency*, ou *idf*) mesure l'importance d'un terme par rapport à une collection *D* de documents. Un terme qui apparaît rarement peut être considéré comme plus caractéristique d'un document qu'un autre, très commun. On retrouve l'idée des mots inutiles, avec un raffinement consistant à mesurer le degré d'utilité.

L'idf d'un terme *t* est obtenu en divisant le nombre total de documents par le nombre de documents contenant au moins une occurrence de *t*. De plus, on prend le logarithme de cette fraction pour conserver cette valeur dans un intervalle comparable à celui du *tf*.

$$\mathrm{idf}(t) = \log \frac{|D|}{\left|\left\{d' \in D \,|\, n_{t,d'} > 0\right\}\right|}$$

Notez que si on ne prenait pas le logarithme, la valeur de l'idf pourrait devenir très grande, et rendrait négligeable l'autre composante du poids d'un terme. La base du logarithme est 10 en général, mais quelle que soit la base, vous noterez que l'idf est nul dans le cas d'un terme apparaissant dans *tous* les documents (c'est clair ? sinon réfléchissez !).

Reprenons notre matrice ci-dessus en supposant que la collection se limite aux trois documents. Alors

— l'idf de « voiture » est 0, car il apparaît dans tous les documents. Intuitivement, « voiture » est (pour la collection étudiée) tellement courant qu'il n'apporte rien comme critère de recherche.
— l'idf de « marais », « serpent » et « baleine » est $log(3/2)$

Dans un cas plus réaliste, un terme qui apparaît 100 fois dans une collection d'un million de documents aura un idf de $log_{10}(1000000/100) = log_{10}(10000) = 4$ (en base 10). Un terme qui n'apparaît que 10 fois aura un idf de $log_{10}(1000000/10) = log_{10}(100000) = 5$. Une valeur d'idf plus élevée indique le terme est relativement plus important car plus rare.

Le poids tf.idf

On peut combiner le tf (normalisé ou non) et l'idf pour obtenir le poids tf.idf d'un terme *t* dans un document. C'est simplement le produit des deux valeurs précédentes :

$$\mathrm{tf.idf}(t, d) = n_{t,d} \cdot \log \frac{|D|}{\left|\left\{d' \in D \,|\, n_{t,d'} > 0\right\}\right|}$$

À chaque document *d* nous associons un vecteur v_d dont chaque composante $v_d[i]$ contient le tf.idf du terme t_i pour *d*.

Si le tf n'est pas normalisé, les valeurs des tf.idf seront d'autant plus élevées que le document est long. En terme de stockage (et pour anticiper un peu sur la structure des index inversés), il est préférable de stocker

— l'idf à part, dans une structure indexée par le terme,
— la norme des vecteurs à part, dans une structure indexéee par les documents
— et enfin de placer dans chaque cellule la valeur du tf.

On peut alors effectuer le produit tf.idf et la division par la norme au moment du calcul de la distance.

7.4.2 La similarité cosinus

Nous avons donc des vecteurs représentant les documents. La requête est elle aussi représentée par un vecteur dans lequel les coefficients des mots sont à 1. Comment calculer la distance entre ces vecteurs ? Si on prend comme mesure la norme de la différence entre deux vecteurs comme nous l'avons fait initialement, des anomalies sévères apparaissent car deux documents peuvent avoir des contenus semblables mais des tailles très différentes. La distance Euclidienne n'est donc pas un bon candidat.

On pourrait mesure la distance euclidienne entre les vecteurs normalisés. Une mesure plus adaptée en pratique est la *similarité cosinus*. Commençons par quelques rappels, en commençant par la formule du *produit scalaire* de deux vecteurs.

$$v_1.v_2 = ||v_1|| \times ||v_2|| \times cos\theta = \sum_{i=1}^{n} v_1[i] \times v_2[i]$$

où θ désigne l'angle entre les deux vecteurs et $||v||$ la norme d'un vecteur v (sa longueur Euclidienne).

On en déduit donc que le cosinus de l'angle entre deux vecteurs satisfait :

$$cos\theta = \frac{\sum_{i=1}^{n} v_1[i] \times v_2[i]}{||v_1|| \times ||v_2||}$$

Quel est l'intérêt de prendre ce cosinus comme mesure de similarité ? L'idée est que l'on compare la *direction* de deux vecteurs, indépendamment de leurs longueurs. La Fig. 7.6 montre la représentation des vecteurs pour nos documents de l'exercice *Ex-S3-1*. Les vecteurs en ligne pleine sont les vecteurs unitaires, normalisés, les lignes pointillées montrant les vecteurs complets. Pour des raisons d'illustration, l'espace est réduit à deux dimensions correspondant aux deux termes, « loup » et « bergerie ». Il faut imaginer un espace vectoriel de dimension *n*, *n* étant le nombre de termes dans la collection, et donc potentiellement très grand.

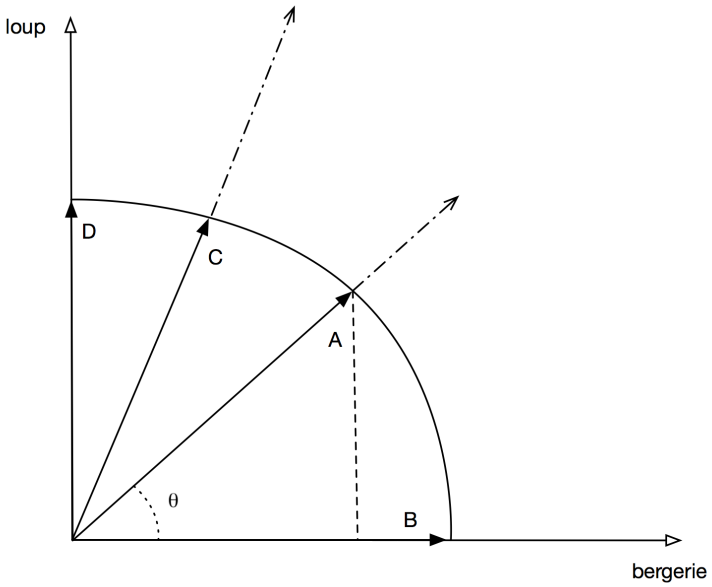


Fig. 7.6 – Illustration de la similarité cosinus

Les documents (B) et (D) contiennent respectivement une occurrence de « bergerie » et une de « loup » : ils sont alignés avec les axes respectifs.

Le document (A) contient une occurrence de « loup » et une de « bergerie » et fait donc un angle de 45 degrés avec l'abcisse : le cosinus de cet angle, égal à $\sqrt{2}/2$, représente la similarité entre A et B.

En ce qui concerne C, « loup » est mentionné deux fois et « bergerie » une, d'où un angle plus important avec l'abcisse.

Le fait d'avoir comme dénominateur dans la formule le produit des normes revient à normaliser le calcul en ne considérant que des vecteurs de longueur unitaire. La mesure satisfait aussi des conditions satisfaisantes intuitivement :

— l'angle entre deux vecteurs de même direction est 0, le cosinus vaut 1 ;
— l'angle entre deux vecteurs orthogonaux, donc « indépendants » (aucun terme en commun), est 90 degrés, le cosinus vaut 0 ;
— toutes les autres valeurs possibles (dans la mesure où les coefficients de nos vecteurs sont positifs) varient continuement entre 0 et 1 avec la variation de l'angle entre 0 et 90 degrés.

Dernier atout : la similarité cosinus est très simple à calculer, et très rapide pour des vecteurs comprenant de nombreuses composantes à 0, ce qui est le cas pour la représentation des documents.

Note : La similarité cosinus n'est pas une distance au sens strict du terme (l'inégalité triangulaire n'est pas respectée), mais ses propriétés en font un excellent candidat.

Passons à la pratique sur notre exemple de trois documents représentés par la matrice donnée précédemment. On va ignorer l'idf pour faire simple, et se contenter de prendre en compte le tf.

Commençons par une requête simplissime : « voiture ». Cette requête est représentée dans l'espace de dimension 4 de notre vocabulaire par le vecteur (1, 0, 0, 0), dont la norme est 1. On pourrait croire qu'il suffit de prendre le classement des *tf* du terme concerné, sans se lancer dans des calculs compliqués, auquel cas *d1* arriverait en tête juste devant *d3*. *Erreur !* Ce qui compte ce n'est pas la fréquence d'un terme, mais sa proportion par rapport aux autres. Il faut appliquer le calcul cosinus rigoureusement.

Calculons donc les cosinus. Pour d1, le cosinus vaut est le produit scalaire des vecteurs (27, 3, 0, 14) et (1, 0, 0, 0), divisé par le produit de la norme de ces deux vecteurs :

$$\frac{27 \times 1 + 3 \times 0 + 0 \times 0 + 14 \times 0}{1 \times 30,56} = 0,88$$

Pour les autres documents :

— Pour d2, le cosinus vaut : $\frac{15+0+0+0}{1 \times 35,35} = 0,424$
— Pour d3, le cosinus vaut : $\frac{24+0+0+0}{1,41 \times 41,3} = 0,58$

Le classement est d1, d3, d2, et on voit que *d1* l'emporte assez nettement sur *d3* alors que le nombre d'occurrences du terme « voiture » est à peu près le même dans les deux cas. Explication : *d1* parle *essentiellement* de voiture, le second terme le plus important, « baleine », ayant moins d'occurrences. Dans *d3* au contraire, « serpent » est le terme principal, « voiture » arrivant en second. Le document *d1* est donc plus pertinent pour la requête et doit être classé en premier.

Prenons un second exemple, « voiture » et « baleine ». Remarquons d'abord que les coefficients de la requête sont (1, 0, 0, 1) et sa norme $\sqrt{1+1} = 1,41$. Voici les calculs cosinus :

— Pour d1, le cosinus vaut : $\frac{27+14}{1,41 \times 30,56} = 0,95$
— Pour d2, le cosinus vaut : $\frac{15+0}{1,41 \times 35,35} = 0,30$
— Pour d3, le cosinus vaut : $\frac{24+17}{1,41 \times 41,3} = 0,70$

L'ordre est donc d1, d3, d2. Le document d3 présente un meilleur équilibre entre les composantes `voiture` et `baleine`, mais, contrairement à d1, il a une autre composante forte pour `serpent` ce qui diminue sa similarité.

7.4.3 Quiz

7.5 Exercices

Exercice Ex-S1-1 : précision et rappel, calculons

Voici une matrice donnant les faux positifs et faux négatifs, vrais positifs et vrais négatifs pour une recherche.

Tableau 7.2 – Table de contingence

	Pertinent	Non pertinent
Ramené (positif)	50	10
Non ramené (négatif)	30	120

Quelques questions :

— Donnez la précision et le rappel.
— Quelle méthode stupide donne toujours un rappel maximal ?
— Quelles sont les valeurs possibles pour la précision si je tire un seul document au hasard ?
— Si j'effectue un tirage aléatoire, que penser de l'évolution de la précision et du rappel en fonction de la taille du tirage ?

Correction

— Précision : 50 / 60
— Rappel : 50 / 80
— Ramener tous les documents
— La précision est soit 1, soit 0.
— Avec un tirage aléatoire, la relation entre espérance du rappel (plus précisément de l'espérance du rappel) et nombre de résultats est monotone croissante. La proportion pertinents / non pertinents reste constante en espérance. Evidemment, plus l'échantillon est restreint, plus cette hypothèse se fragilise, avec donc une variance qui augmente.

Exercice Ex-S1-2 : précision et rappel, réfléchissons.

Soit un système qui affiche systématiquement 20 documents, ni plus ni moins, pour toutes les recherches. Indiquez quel est la précision et le rappel dans les cas suivants :

— Mon besoin de recherche correspond à un document unique de la collection, il est affiché parmi les 20.
— Ma base contient 100 documents, je veux tous les obtenir, le système m'en renvoie 20.
— Je fais une recherche dont je sais qu'elle devrait me ramener 30 documents. Parmi les 20 que renvoie le système, je n'en retrouve que 10 parmi ceux attendus.

Correction

— p = 1/20, r = 1/1 = 1
— p = 20/20 = 1 ; r = 20/100 = 0.2
— p = 10/20 = 0.5 ; r = 10/30 = 1/3

Exercice Ex-S1-3 : proposons une autre mesure

Voici une autre mesure de qualité, que nous appellerons *exactitude* comme traduction de *accuracy* (exactitude). Elle se définit comme la fraction du résultat qui est correcte (en comptant les vrais positifs et vrais négatifs comme corrects). Donc,

$$accuracy = \frac{t_n + t_p}{t_n + t_p + f_n + f_p}.$$

Où p et n désignent les négatifs et positifs, t et f les vrais et faux, respectivement.

Cette mesure ne convient pas en recherche d'information. Pourquoi ? Imaginez un système où seule une infime partie des documents sont pertinents, quelle que soit la recherche.

— Quelle serait l'exactitude dans ce cas ?
— Quelle méthode simple et stupide donnerait une exactitude proche de la perfection ?

Correction

— L'exactitude tend vers 1
— Il suffit de renvoyer un résultat vide, quelle que soit la requête.

Exercice Ex-S1-4 : matrice d'incidence

Faire un fichier Excel (ou l'équivalent) qui calcule la taille d'une collection, la taille de la matrice d'incidence, et la densité de 1 dans cette matrice, en fonction des paramètres suivants :

— n, le nombre de documents,
— d, le nombre de termes distincts,
— m, le nombre moyen de mots dans un document,
— w, la taille moyenne d'un mot (en octets).

Exercice Ex-S1-5 : algorithme de négation

On sait faire une fusion pour une recherche de type ET. La recherche de type OU est évidente (?). Montrer qu'un algorithme similaire existe pour une recherche de type NOT (« les documents qui parlent de loup mais pas de mouton »).

Exercice Ex-S3-1 : premiers pas vers la recherche plein texte

Voici quelques documents textuels (brefs !).

— A : `Le loup est dans la bergerie.`
— B : `Les moutons sont dans la bergerie.`
— C : `Un loup a mangé un mouton, les autres loups sont restés dans la bergerie.`
— D : `Il y a trois moutons dans le pré, et un mouton dans la gueule du loup.`

Prenons le vocabulaire suivant : {« loup », « mouton », « bergerie », « pré », « gueule »}.

— Construisez la fonction qui associe chaque document à un vecteur dans $\{0,1\}^5$. Vous pouvez représenter cette fonction sous forme d'une matrice d'incidence.
— Calculer le score de chaque document par la distance Euclidienne pour les recherches suivantes, et en déduire le classsement :
  — q_1. « loup et pré »
  — q_2. « loup et mouton »
  — q_3. « bergerie »
  — q_4. « gueule du loup »

Correction

Les vecteurs des documents

— $v(A) = [1,0,1,0,0]$
— $v(B) = [0,1,1,0,0]$
— $v(C) = [1,1,1,0,0]$
— $v(D) = [1,1,0,1,1]$

Les vecteurs des requêtes

— $v(q_1) = [1,0,0,1,0]$
— $v(q_2) = [1,1,0,0,0]$
— $v(q_3) = [0,0,1,0,0]$
— $v(q_4) = [1,0,0,0,1]$

En ce qui concerne les requêtes :

— $E(q_1,A) = \sqrt{2}; E(q_1,B) = \sqrt{4}; E(q_1,C) = \sqrt{3}; E(q_1,D) = \sqrt{2}$. A et D sont les plus pertinents, suivi de C et enfin B. Notez que A de contient pas le mot « pré ». Pourquoi obtient-il le même score que D ?
— $E(q_2,A) = \sqrt{2}; E(q_2,B) = \sqrt{2}; E(q_2,C) = \sqrt{1}; E(q_2,D) = \sqrt{2}$.
— $E(q_3,A) = \sqrt{1}; E(q_3,B) = \sqrt{1}; E(q_3,C) = \sqrt{2}; E(q_3,D) = \sqrt{5}$.
— $E(q_4,A) = \sqrt{2}; E(q_4,B) = \sqrt{4}; E(q_4,C) = \sqrt{3}; E(q_4,D) = \sqrt{2}$. C'est donc D qui l'emporte, mais à égalité avec A, ce ne correspond pas vraiment à l'intuition.

Exercice Ex-S3-2 : à propos de la fonction de distance

Supposons que l'on prenne comme distance non pas la distance Euclidienne mais le carré de cette distance. Est-ce que cela change le classement ? Qu'est-ce que cela vous inspire ?

Correction

On peut effectuer un calcul *simplifié* de la distance, tant que l'ordre est respecté. Ce qui nous intéresse en fait, ce n'est pas le score proprement dit, mais l'ordre des scores.

Exercice Ex-S3-3 : critique de la distance Euclidienne

La distance que nous avons utilisée mesure la *différence* entre la requête et un document, par comparaison des termes un à un. Cela induit des inconvénients qu'il est assez facile de mettre en évidence.

Supposons maintenant que le vocabulaire a une taille très grande. On fait une recherche avec 1 mot-clé.

— quel est le score pour un document qui ne contient 99 termes et pas ce mot-clé ?

— quel est le score pour un document qui contient 101 termes *et* le mot-clé ?

Conclusion ? Le classement obtenu sera-t-il satisfaisant ? Trouvez un cas où un document est bien classé même s'il ne contient pas le mot-clé !

Correction

Distance de $\sqrt{100} = 10$ dans le premier cas ; de $\sqrt{100}$ dans le second également. Ils seront classés au même niveau, ce qui ne va pas du tout !

Un document avec 50 termes sera mieux classé que n'importe quel document de 100 termes contenant ou non le mot-clé. Cela nous met sur la piste de ce qui ne va pas : la mesure que nous utilisons est fortement dépendante de la taille du document, au détriment de sa pertinence.

Exercice Ex-S3-4 : critique de l'hypothèse d'uniformité des termes

Enfin, dans notre approche très simplifiée, tous les termes ont la même importance. Calculez le classement pour la requête :

— q_5. « bergerie et gueule »

et tentez d'expliquer le résultat. Est-il satisfaisant ? Quel est le biais (pensez au raisonnement sur la longueur du document dans l'exercice précédent).

Correction

Vecteur de la requête : $v(q_5) = [0, 0, 1, 0, 1]]$

Calcul des classements :

— $E(q_5, A) = \sqrt{2}; E(q_5, B) = \sqrt{2}; E(q_5, C) = \sqrt{3}; E(q_2, D) = \sqrt{4}.$

Tous les documents sont mieux classés que D, alors que « gueule » est un terme plus discriminant.

Exercice Ex-S4-1 : à propos de la requête

Je soumets une requête $t_1, t_2, \cdots, t_n$. Quel est le poids de chaque terme dans le vecteur représentant cette requête ? La normalisation de ce vecteur est elle importante pour le classement (justifier) ?

Exercice Ex-S4-2 : encore des voitures, des serpents et des baleines

Reprenons notre exemple des documents *d1*, *d2* et *d3*, calculez le classement pour les requêtes suivantes :

— serpent
— voiture et serpent

Exercice Ex-S4-3 : pesons le loup, le mouton et la bergerie

Nous reprenons nos documents de l'exemple *Ex-S3-1*. Le vocabulaire est toujours le suivant : {« loup », « mouton », « bergerie », « pré », « gueule »}.

— Donnez, pour chaque document, le tf de chaque terme.
— Donnez les idf des termes (ne pas prendre le logarithme, pour simplifier).
— En déduire la matrice d'incidence montrant l'idf pour chaque terme, le nombre de termes pour chaque document, et le tf pour chaque cellule.

Correction

Fréquence des termes par document (tf).

— Document A : loup (1), bergerie (1)
— Document B : mouton (1), bergerie (1)
— Document C : loup (2), mouton (1), bergerie (1)
— Document D : loup (1), mouton (2), pré (1), gueule (1)

Fréquence inverse des termes dans les documents (idf) : loup (4/3), mouton (4/3), bergerie (4/3), pré (4), gueule (4).

Tableau 7.3 – La matrice d'incidence

	loup (4/3)	mouton (4/3)	bergerie (4/3)	pré (4)	gueule (4)
A	1	0	1	0	0
B	0	1	1	0	0
C	2	1	1	0	0
D	1	2	0	1	1

Exercice Ex-S4-4 : interrogeons et classons

Reprendre les requêtes suivantes :

— q_1. « loup et pré »
— q_2. « loup et mouton »
— q_3. « bergerie »
— q_4. « gueule du loup »

et calculer le classement avec la distance cosinus, en ne prenant en compte que le vecteur des tf.

Correction

1. Les normes :
   — A : $\sqrt{1+1} = \sqrt{2}$
   — B : $\sqrt{1+1} = \sqrt{2}$
   — C : $\sqrt{2^2+1+1} = \sqrt{6}$
   — D : $\sqrt{2^2+1+1+1} = \sqrt{7}$

1. Classement (on ne normalise pas la requête car cela n'influe pas sur le classement, et du coup on a des cosinus supérieurs à 1) :

   — loup et pré :
     — A : $\frac{1+0+0+0+0}{\sqrt{2}} = 0,707$
     — B : $\frac{0}{\sqrt{2}} = 0$
     — C : $\frac{2+0+0+0+0}{\sqrt{6}} = 0,816$
     — D : $\frac{1+0+0+1+0}{\sqrt{7}} = 0,755$

     Seul D contient les deux termes, *mais* il en contient aussi beaucoup d'autres. Il est donc dominé par C, qui parle fortement de loup mais par de pré, et contient (un peu) moins de termes. A est bien classé, même s'il ne contient pas « pré », car (intuitivement)à il parle de « loup » de manière forte, ne contenant que très peu de termes.

   — **loup et mouton :**
     — A : $\frac{1+0+0+0+0}{\sqrt{2}} = 0,707$
     — B : $\frac{0+1+0+0+0}{\sqrt{2}} = 0,707$
     — C : $\frac{2+1+0+0+0}{\sqrt{6}} = 1,22$
     — D : $\frac{1+2+0+0+0}{\sqrt{7}} = 1,13$

     C et D sont à peu près à égalité, comme on peut le prévoir intuitivement : ils parlent des deux termes de manière un peu déséquilibrée, par rapport à la requête. C l'emporte car sa norme est plus petite, il est donc plus « concentré » sur les termes de la requête.

   — **bergerie :**
     — A : $\frac{1}{\sqrt{2}} = 0,707$
     — B : $\frac{1}{\sqrt{2}} = 0,707$
     — C : $\frac{1}{\sqrt{6}} = 0,4$
     — D : $\frac{0}{\sqrt{7}}$

     A et B arrivent à égalité : il parlent tous les deux une fois de bergerie, et contiennent exactement un autre terme.

   — **gueule du loup :**
     — A : $\frac{1+0+0+0+0}{\sqrt{2}} = 0,707$
     — B : $\frac{0}{\sqrt{2}} = 0$
     — C : $\frac{2+0+0+0+0}{\sqrt{6}} = 0,816$
     — D : $\frac{1+0+0+0+1}{\sqrt{7}} = 0,75$

     Même analyse que pour la première requête

Exercice Ex-S4-5 : comparons les loups et les moutons

— Reprenez une nouvelle fois les documents de l'exercice *Ex-S1-1*. Vous devriez avoir la matrice des tf.idf calculée dans l'exercice *Ex-S4-3*.
  — classez les documents B, C, D par similarité cosinus décroissante avec A ;
  — calculez la similarité cosinus entre chaque paire de documents ; peut-on identifier 2 groupes évidents ?

Exercice Ex-S4-6 : un exemple complet

Voici trois recettes.

— Panna cotta (pc) : Mettre la crème, le sucre et la vanille dans une casserole et faire frémir. Ajouter les 3 feuilles de gélatine préalablement trempées dans l'eau froide. Bien remuer et verser la crème dans des coupelles. Laisser refroidir quelques heures.
— Crème brulée (cb) : Faire bouillir le lait, ajouter la crème et le sucre hors du feu. Ajouter les jaunes d'œufs, mettre au four au bain marie et laisser cuire doucement à 180C environ 10 minutes. Laisser refroidir puis mettre dessus du sucre roux et le brûler avec un petit chalumeau.
— Mousse au chocolat (mc) : Faire ramollir le chocolat dans une terrine. Incorporer les jaunes et le sucre. Puis, battre les blancs en neige ferme et les ajouter délicatement au mélange à l'aide d'une spatule. Mettre au frais 1 heure ou 2 minimum.

À vous de jouer pour la création de l'index et les calculs de classement.

— On prend pour vocabulaire les mots suivants : crème, sucre, œuf, gélatine. Tous les autres mots sont ignorés. Donnez la matrice d'incidence avec l'idf de chaque terme, et le tf de chaque paire (terme, document).
— Donnez les normes de vecteurs représentant chaque document.
— Donner les résultats classés par combinaison tf (on ignore l'idf) pour les requêtes suivantes
  — crème et sucre
  — crème et œuf
  — œuf et gélatine
— Même chose mais en tenant compte de l'idf (sans appliquer le logarithme).
— Commentez le résultat de la dernière requête. Est-il correct intuitivement ? Que penser de l'indexation du terme "œuf", est-elle réprésentative du contenu des recettes ?

Correction

1. Matrice d'incidence

   Crème apparaît dans 2 documents sur 3, idf=3/2 ; sucre dans 3 sur 3 (idf=1), œuf dans 1 sur 3 (idf=3) comme gélatine. On obtient :

	pc	cb	mc
crème (3/2)	2	1	0
sucre (1)	1	2	1
œuf (3)	0	1	0
gélatine (3)	1	0	0

1. Les normes :
   — pc : $\sqrt{2^2 + 1 + 1} = \sqrt{6}$

— cb : $\sqrt{1+2^2+1} = \sqrt{6}$
— mc : $\sqrt{1} = 1$

2. Classement (on ne normalise pas la requête car cela n'influe pas sur le classement, et du coup on a des cosinus supérieurs à 1) :
— crème et sucre :
— pc : $\frac{2+1}{\sqrt{6}}$
— cb : $\frac{1+2}{\sqrt{6}}$
— mc : $\frac{1}{1}$
— crème et œuf :
— pc : $\frac{2}{\sqrt{6}}$
— cb : $\frac{1+1}{\sqrt{6}}$
— mc : $\frac{0}{1}$
— œuf et gélatine :
— pc : $\frac{1}{\sqrt{6}}$
— cb : $\frac{1}{\sqrt{6}}$
— mc : $\frac{0}{1}$
3. Etudions maintenant l'impact de l'idf. En prenant le logarithme, on obtient :
— idf(crème) $= log(1.5) = 0,176$
— idf(sucre) $= log(1) = 0$
— idf(œuf) $= log(3) = 0,477$
— idf(gélatine) $= log(3) = 0,477$
On remarque que l'IDF de "sucre" vaut zéro, ce qui revient à dire que ce terme ne sert à rien dans une requête. Cela reflête le fait que "sucre" apparaît dans *tous* les documents et que son pouvoir de discrimination est nul.
Ce qui nous donne la matrice des tf.idf suivante :

	pc	cb	mc
crème	0,352	0,176	0
sucre	0	0	0
œuf	0	0,477	0
gélatine	0,477	0	0

Un inconvénient immédiat apparaît : le document `mc` a tous ses coefficients à zéro et n'apparaîtra dans aucun résultat, même pour les requêtes sur le sucre. Pour pallier ce défaut, on peut prendre pour l'idf non pas le log mais $1 + log$. Celà fait partie des très nombreux petits réglages qui font la qualité d'un moteur de recherche.
Les normes des vecteurs :
— pc : $\sqrt{0,352^2 + 0,477^2} = 0,593$
— cb : $\sqrt{0,176^2 + 0,477^2} = 0,508$
La requête « œuf et gélatine » est représentée par le vecteur (0,477, 0,477). La similarité cosinus (toujours en ignorant la norme de la requête qui est constante et ne change pas le classement) est donc :
— pc : $\frac{0,477^2}{0,593}$
— cb : $\frac{0,477^2}{0,508}$
C'est donc la crème brulée qui l'emporte. Pourquoi ? Parce que le terme « œuf » commun entre la requête et le document est prépondérant dans la représentation vectorielle de ce dernier. En prenant en

compte l'idf, il semble en effet que le document `cb` parle essentiellement d'œuf, alors que le document `pc` parle à peu près à parts égales d'œuf et de crème. Le document le plus spécifiquement liée à la requête vient donc en premier.

La pauvre mousse au chocolat n'apparaît pas dans le résultat, alors qu'il faut vraiment des œufs pour la préparer.

4. Interprétation : clairement, il faut des œufs dans la mousse au chocolat, mais le mot "œuf" lui-même n'apparaît pas, ce qui fausse les résultats. Le tf est à 0 pour le document mc, et l'idf de "œuf" est surévalué (parce que le nombre de documents lui-même est ridiculement petit).

   Notre moteur de recherche souffre donc des limitations d'une approche purement lexicale basée uniquement sur le tf.idf. Il mérite donc des améliorations, la plus séduiasante étant la consrtruction de réseaux de termes de type *word to vec* pour tenter d'identifier les concepts à apparier, plutôt que les termes.

Exercice Ex-S4-7 : et si on calculait autrement ?

Chaque document est représenté comme un vecteur dans un espace *n*-dimensionnel. avec des coefficients *normalisés* tf.idf.

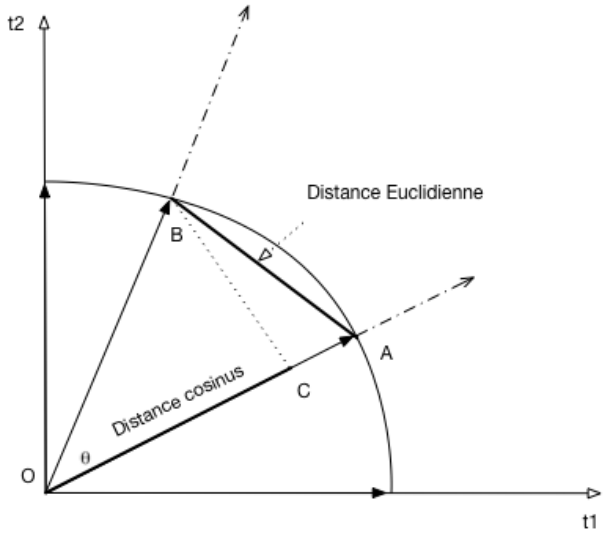


Fig. 7.7 – Calcul basé sur la distance Euclidienne

Revenons à notre idée initiale de calculer la similarité basée sur la distance Euclidienne entre les deux points A et B :

$$E(A,B) = \sqrt{(a_1 - b_1)^2 + (a_2 - b_2)^2 + \cdots + (a_n - b_n)^2}$$

La figure Fig. 7.7 montre la distance Euclidienne, et la distance cosinus entre les deux points A et B. Montrez qu'un classement par ordre décroissant de la similarité cosinus et identique au classement par ordre croissant de la distance Euclidienne ! (Aide : montrez que la distance augmente quand le cosinus diminue. Un peu de Pythagore peut aider).

Correction

Notons que : OBC et CBA sont deux triangles rectangles, par définition du cosinus comme projection orthogonale. Par ailleurs, on ne considère pour les calculs de similarité que les points situés dans le quadrant des coordonnées positives, et le cosinus est donc une valeur positive entre 0 et 1.

Enfin, on sait que A et B sont sur le cercle trigonométrique. Donc :

- $||OB|| = 1$
- $||OA|| = 1 = ||OC|| + ||CA||$

C'est parti pour Pythagore, comme au collège. Nous avons :

- $||AB||^2 = ||BC||^2 + ||AC||^2$
- $||OB||^2 = 1 = ||OC||^2 + ||BC||^2$, et donc $||BC||^2 = 1 - ||OC||^2$

Ce qui nous donne :

- $||AB||^2 = 1 - ||OC||^2 + ||AC||^2 = 1 - ||OC||^2 + (1 - ||OC||)^2$
- $||AB||^2 = 2(1 - ||OC||)$

AB est la distance euclidienne, OC la distance cosinus comprise entre 0 et 1. L'une augmente (de 0 à $\sqrt{2}$) quand l'autre diminue (de 1 à 0). C.Q.F.D.

La distance cosinus est plus efficace à calculer car il suffit de considérer les indices des deux vecteurs pour lesquels les coordonnéess sont toutes les deux non nulles, alors que pour la distance euclidienne on doit prendre en compte ceux pour lesquelles *au moins* une des coordonnées est non nulle.

CHAPITRE 8

ElasticSearch - Travaux pratiques

Ce chapitre est consacré l'interrogation d'une base Elasticsearch, en utilisant le DSL (Domain Specific Language) dédié à ce moteur de recherche. Il est conçu dans une optique de mise en pratique : vous devriez disposer d'un serveur installé avec Docker et de l'application ElasticVue (reportez-vous au chapitre *Recherche approchée*).

Nous allons utiliser une base de films plus large que celle que l'on a vue en cours. Récupérez un fichier contenant environ 5000 films, au format JSON : http://b3d.bdpedia.fr/files/big-movies-elastic.json.

Vous pouvez importer avec ElasticVue en faisant des copier/coller. Autre possibilité : appeler directement le service `_bulk` avec `curl`. Dans le dossier où vous avez récupéré le fichier, lancez la commande de chargement dans ElasticSearch suivante (ajoutez login et mot de passe) :

```
curl -s --cacert http_ca.crt  -U elastic:mot_de_passe https -X POST http://
↪localhost:9200/_bulk/ --data-binary @big-movies-elastic.json
```

Dans l'interface Elasticvue, vous devriez voir apparaître un index appelé *movies* contenant 5000 films. Les documents ont la structure suivante :

```
{
  "fields": {
    "directors": [
      "Joseph Gordon-Levitt"
    ],
    "release_date": "2013-01-18T00:00:00Z",
    "rating": 7.4,
    "genres": [
      "Comedy",
      "Drama"
    ],
```

(suite sur la page suivante)

(suite de la page précédente)


```
    "image_url": "http://ia.media-imdb.com/images/M/MVNTQ3OQ@@._V1_SX400_.jpg",
    "plot": "A New Jersey guy dedicated to his family, friends, and church,
      develops unrealistic expectations from watching porn and works to find
      happ iness and intimacy with his potential true love.",
    "title": "Don Jon",
    "rank": 1,
    "running_time_secs": 5400,
    "actors": [
      "Joseph Gordon-Levitt",
      "Scarlett Johansson",
      "Julianne Moore"
    ],
    "year": 2013
  },
  "id": "tt2229499",
  "type": "add"
}
```


8.1 S1 : recherche plein texte avec ElasticSearch

Le DSL est un langage extrêmement riche. Nous avons déjà vu les recherches « exactes » dans le chapitre *Recherche exacte*. Nous nous concentrons sur les recherches plein-texte (avec classement donc) et quelques opérations de combinaison. La documentation officielle est ici : https://www.elastic.co/guide/en/elasticsearch/reference/current/full-text-queries.html.

8.1.1 Les recherches plein-texte

On s'intéresse ici aux recherches portant sur des champs textuels analysés (et ayant donc fait l'objet de transformations, cf. le chapitre *Recherche approchée*). La correspondance entre le texte indexé et celui de la requête est exprimée par un *score*. Reprenons la requête qui cherche les occurrences de *Star Wars* avec l'opérateur `match`

```
{
"query": {
        "match": {
                "title": "Star Wars"
        }
        },
        "fields": [
        "title", "summary"
        ],
        "_source": false
}
```

Cette fois, contrairement à ce qui se passait avec `term`, les transformations sont appliquées au document *et* à la requête, et le résultat correspond aux principes de la recherche plein texte avec classement. Vous pouvez constater que la requête est triée sur le score (attribut `_score` dans chaque élément du résultat). Prenez le temps de comprendre (au moins intuitivement) le rapport entre le titre du film et son classement.

Il faut bien réaliser que chaque terme est pris en compte *individuellement*. Cherchons par exemple les résumés de film qui contiennent `Roman Empire`.

```
{
"query": {
        "match": {
                "summary": "Roman Empire"
        }
        },
        "fields": [
        "title", "summary"
        ],
        "_source": false
}
```

On obtient des résumés qui contiennent `Roman` et `Empire`, `Roman` tout seul et `Empire` tout seul : la différence est reflétée dans le score, et donc dans le classement.

Remplacez `match` par `match_phrase`, comparez les résultats et reportez-vous à la documentation pour comprendre.

Enfin il est possible d'effectuer une recherche plein-texte moins « structurée » avec l'opérateur `query_string` dont le paramètre est une liste de mots-clé enrichis de connecteur booléens et d'options. Cette liste est l'équivalent (en plus puissant) de l'approche habituelle avec les applications de recherche, consistant à mettre en vrac les termes principaux de la recherche. Voici un premier exemple.

```
{
        "query": {
        "query_string": {
                "query": "(DiCaprio) OR (Deneuve)"
        }
        },
        "fields": ["title"],
        "_source": false
}
```

Et un second exemple un peu plus élaboré :

```
{
        "query": {
                "query_string": {
          "query": "year:[1990 TO 2010] AND director.last_name:Tarantino"
        }
        },
```

(suite sur la page suivante)

(suite de la page précédente)

```
        "fields": ["title"],
        "_source": false
}
```

Reportez-vous à la documentation pour les très nombreuses options (qui reprennent en fait celles du langage du système d'indexation sous-jacent, Lucene).

8.1.2 Combinaison de recherches

On peut combiner plusieurs recherches en paramétrant la manière dont les critères de recherche et les scores se combinent. Nous allons nous limiter ci-dessus aux combinaisons booléennes. La documentation (https://www.elastic.co/guide/en/elasticsearch/reference/current/query-dsl-bool-query.html) donne des informations plus complètes.

Les recherches booléennes sont exprimées par des objets (au sens JSON) `bool` dans l'object `query`. Cet objet `bool` peut lui-même avoir plusieurs sous-clauses : `must`, `should`, `must_not` et `filter`. Voyons quelques exemples.

La clause `must`

L'objet `must` est un tableau de recherches plein-texte ou exactes, et s'interprète comme un `ET` logique (ou, en termes ensemblistes, comme une intersection des résultats). Voici un exemple d'une requête des films dont le titre est proche de Star Wars *et* qui sont parus entre 1970 et 2000. On combine un `match` et un `range`.

```
{
  "query": {
        "bool": {
        "must": [
                {
                        "match": {"title": "Star Wars"}
                },
                {
                         "range": {"year": {"gte": 1970,"lte": 2000} }
            }
         ]
                }
        }
}
```

Et oui, la syntaxe devient un peu compliquée… Si on utilise `should` à la place de `must`, on obtient une interprétation disjonctive `OR` (et donc une *union* des résultats). À vous de vérifier (vous pouvez aussi trouver un exemple plus intéressant, comme les films tournés soit par Q. Tarantino soit par J. Woo.).

Enfin le `must_not` correspond au `NOT`. On peut donc construire des expressions booléennes complexes, au prix d'un syntaxe il est vrai assez lourde. Un exemple : les films avec Bruce Willis, sauf ceux tournés par Q. Tarantino.

```
{
        "query": {
        "bool": {
                "must":
                        {
                  "match": {"actors.last_name": "Willis"}
                         },
                "must_not":
                        {
                        "match": {"director.last_name": "Tarantino"}
                        }
                         }
                }
        }
```

Je vous laisse étudier la clause `filter` qui sert principalement à exclure des documents du résultat sans infllluer sur le score.

Vous pouvez limiter la quantité d'informations qui se trouvent dans le champ `_source` de chacun des résultats comme ceci :

```
{
  "_source": {
    "includes": [
      "*.title",
      "*.directors"
    ],
    "excludes": [
      "*.actors*",
      "*.genres"
    ]
  },
  "query": {
    "match": {
      "fields.title": "Star Wars"
    }
  }
}
```

8.1.3 À vous de jouer

Cette section suppose un investissement de votre part pour comprendre et expérimenter le langage d'Elastic Search. *Elle est optionnelle dans le cadre de l'UE NFE204* pour laquelle on ne vous demandra pas de connaître la syntaxe d'un système particulier. À faire donc uniquement si vous souhaitez approfondir le sujet.

Proposez des requêtes pour les besoins d'informations suivants (vous pouvez aussi proposer des variantes « exactes ») :

— Films "Star Wars" dont le réalisateur (directors) est "George Lucas" (requête booléenne)

Correction

```
{
  "query": {
    "bool": {
      "should": [
        {
          "match": {
            "fields.title": "Star Wars"
          }
        },
        {
          "match": {
            "fields.directors": "George Lucas"
          }
        }
      ]
    }
  }
}
```

Pour une recherche exacte :

```
{
  "query": {
    "bool": {
      "should": [
        {
          "match_phrase": {
            "fields.title": "Star Wars"
          }
        },
        {
          "match": {
            "fields.directors": "George Lucas"
          }
        }
      ]
```

(suite sur la page suivante)

(suite de la page précédente)

```
    }
  }
}
```

Variante :

```
{
  "_source": {
    "includes": [ "*.title" ],
    "excludes": [ "*.actors*", "*.genres", "*.directors" ]
  },
  "query": {
    "bool": {
      "should": [
        {
          "match": {
            "fields.title": {
              "query": "Star Wars",
              "operator": "and"
            }
          }
        },
        {
          "match": {
            "fields.directors": {
              "query": "Georges Lucas",
              "operator": "and"
            }
          }
        }
      ]
    }
  }
}
```

ou `_search?q=fields.title:Star+Wars directors:George+Lucas`

— Films dans lesquels “Harrison Ford” a joué

Correction

```
{
  "query": {
    "match": {
      "fields.actors": "Harrison Ford"
    }
  }
```

(suite sur la page suivante)

(suite de la page précédente)

```
}
```

ou `_search?q=fields.actors:Harrison+Ford`

— Films dans lesquels “Harrison Ford” a joué dont le résumé (plot) contient “Jones”.

Correction

```
{"query":{
  "bool": {
    "should": [
      { "match": { "fields.actors": "Harrison Ford" }},
      { "match": { "fields.plot": "Jones" }}
    ]
}}}
```

`_search?q=fields.actors=Harrison+Ford fields.plot:Jones`

— Films dans lesquels “Harrison Ford” a joué dont le résumé (plot) contient “Jones” mais sans le mot “Nazis”

Correction

```
{"query":{
  "bool": {
    "should": [
      { "match": { "fields.actors": "Harrison Ford" }},
      { "match": { "fields.plot": "Jones" }}
    ],
    "must_not" : { "match" : {"fields.plot":"Nazis"}}
}}}
```

`_search?q=actors=Harrison+Ford plot:Jones -plot:Nazis`

— Films de “James Cameron” dont le rang devrait être inférieur à 1000 (boolean + range query).

Correction

```
{"query":{
  "bool": {
    "should": [
      { "match": { "fields.directors": "James Cameron" }},
      { "range": { "fields.rank": {"lt":1000 }}}
    ]
}}}
```

— Films de “James Cameron” dont le rang **doit** être inférieur à 400 (réponse exacte : 2)

Correction

```
{
  "query": {
    "bool": {
      "must": [{
          "match_phrase": {
            "fields.directors": "James Cameron"
          }
        },
        {
          "range": {
            "fields.rank": {
              "lt": 400
            }
          }
        }
      ]
    }
  }
}
```

— Films de “Quentin Tarantino” dont la note (rating) doit être supérieure à 5, sans être un film d’action ni un drame.

Correction

```
{
  "_source": {
    "includes": [
      "*.title"
    ],
    "excludes": [
      "*.actors*"
    ]
  },
  "query": {
    "bool": {
      "must": [
        {
          "match_phrase": {
            "fields.directors": "Quentin Tarantino"
          }
        },
        {
          "range": {
```

(suite sur la page suivante)

(suite de la page précédente)

```
            "fields.rating": {
              "gte": 5
            }
          }
        }
      ],
      "must_not": [
        {
          "match": {
            "fields.genres": "Action"
          }
        },
        {
          "match": {
            "fields.genres": "Drama"
          }
        }
      ]
    }
  }
}
```

— Films de “J.J. Abrams” sortis (released) entre 2010 et 2015

Correction

```
{
  "query": {
    "bool":{
      "must": {"match": {"fields.directors": "J.J. Abrams"}},
      "filter": {
        "range": {
          "fields.release_date": { "from": "2010-01-01", "to": "2015-12-31"}
        }
      }
    }
  }
}
```

8.2 S2 : Agrégats

Elasticsearch permet également d'effectuer des agrégations, dans l'esprit du `group by` de SQL.

8.2.1 Syntaxe

Les agrégats fonctionnent avec deux concepts, les *buckets* (seaux, en français) qui sont les catégories que vous allez créer, et les *metrics* (indicateurs, en français), qui sont les statistiques que vous allez calculer sur les *buckets*.

Si l'on compare à une requête SQL très simple :

```
select count(color)
from table
group by color
```

`count(color)` est la métrique, `group by color` crée les groupes (*buckets*).

Une agrégation est la combinaison d'un *bucket* (au moins) et d'une *metric* (au moins). On peut, pour des requêtes complexes, imbriquer des *buckets* dans d'autres *buckets*. La syntaxe est, comme précédemment, très modulaire.

Un exemple, avec le nombre de films par année :

```
{
  "size": 0,
  "aggs" : { "nb_par_annee" : {
    "terms" : {"field" : "year"}
      }
  }
}
```

Le paramètre `aggs` permet à Elasticsearch de savoir qu'on travaille sur des agrégations. Le paramètre `size:0` permet de ne pas afficher les résultats de recherche de la requête. `nb_par_annee` est le nom que l'on donne à notre agrégat. Les buckets sont créés avec le `terms` qui ici indique que l'on va créer un groupe par valeur différente du champ `year`. La métrique sera automatique ici, ce sera simplement la somme de chaque catégorie.

Le résultat d'une agrégation apparaît dans un champ `aggregations` dans le résultat. Voici un extrait de ce dernier. Remarquez que le tableau `hits` est vide (car `size:0`). Le tableau `buckets` en revanche contient ce que nous cherchons (ouf).

```
  {
          "hits": {
          "total": {
                  "value": 326,
                  "relation": "eq"
          },
          "max_score": null,
```

(suite sur la page suivante)

(suite de la page précédente)

```
          "hits": []
          },
          "aggregations": {
          "nb_par_annee": {
          "buckets": [
                  {"key": 2017,"doc_count": 12},
                  {"key": 2019,"doc_count": 11},
                  {"key": 2005,"doc_count": 9},
          ]
   }
}
```

On peut appliquer une agrégation sur le résultat d'une requête, comme par exemple ci-dessous où on ne prend que les films du genre « western ».

```
        {
          "query": {
        "term": {
                "genre": {"value": "western"}
        }
        },
        "aggs": {
        "nb_par_annee": {
                "terms": {
                        "field": "year"
                }
         }
        }
}
```

8.2.2 À vous de jouer

Proposez maintenant les requêtes permettant d'obtenir les statistiques suivantes :

— Donner la note (rating) moyenne des films.

Correction

```
{"size":0,
"aggs" : {
    "note_moyenne" : {
      "avg" : {"field" : "fields.rating"}
    }}}
```

— Donner la note (rating) moyenne, et le rang moyen des films de George Lucas (cliquer sur (-) à côté de « hits » dans l'interface pour masquer les résultats et consulter les valeurs calculées)

Correction

```
{"query" :{
    "match" : {"fields.directors": {"query": "George Lucas", "operator":
↪"and"}}
  }
 ,"aggs" : {
    "note_moyenne" : {
      "avg" : {"field" : "fields.rating"}
    },
    "rang_moyen" : {
      "avg" : {"field" : "fields.rank"}
    }
}}
```

— Donnez la note (rating) moyenne des films par année. Attention, il y a ici une imbrication d'agrégats (on obtient par exemple 456 films en 2013 avec un *rating* moyen de 5.97).

Correction

```
{"aggs" : {
    "group_year" : {
      "terms" : {
        "field" : "fields.year"
      },
      "aggs" : {
        "note_moyenne" : {
          "avg" : {"field" : "fields.rating"}
        }}
    }}}
```

— Donner la note (rating) minimum, maximum et moyenne des films par année.

Correction

```
{"aggs" : {
    "group_year" : {
      "terms" : {
        "field" : "fields.year"
      },
      "aggs" : {
        "note_moyenne" : {"avg" : {"field" : "fields.rating"}},
        "note_min" : {"min" : {"field" : "fields.rating"}},
        "note_max" : {"max" : {"field" : "fields.rating"}}
      }
    }}}
```

— Donner le rang (rank) moyen des films par année et trier par ordre décroissant.

Correction

```
{"aggs" : {
    "group_year" : {
      "terms" : {
        "field" : "fields.year",
        "order" : { "rating_moyen" : "desc" }
      },
      "aggs" : {
        "rating_moyen" : {
          "avg" : {"field" : "fields.rating"}
      }}
}}}
```

— Compter le nombre de films par tranche de note (0-1.9, 2-3.9, 4-5.9...). Indice : `group_range`.

Correction

```
{"aggs" : {
    "group_range" : {
      "range" : {
        "field" : "fields.rating",
        "ranges" : [
          {"to" : 1.9},
          {"from" : 2, "to" : 3.9},
          {"from" : 4, "to" : 5.9},
          {"from" : 6, "to" : 7.9},
          {"from" : 8}
        ]
      }
    }}}
```

— Donner le nombre d'occurrences de chaque genre de film.

Correction
Cette opération n'est pas possible car « genres » est une liste de valeur. Il faudrait pour cela créer un mapping particulier sur les données (autre que automatique) pour pouvoir faire l'agrégation. (Nous ferons cela dans la section suivante). Même chose si l'on cherchait à connaître les occurrences des termes utilisés.

8.3 S3 : Le classement dans Elasticsearch

Etudions maintenant le classement effectué par ElasticSearch et la façon dont on peut le contrôler voire le modifier.

8.3.1 Le score

Le score d'un document est calculé en fonction d'une requête au moyen d'une fonction dite *Practical Scoring Function* reprise du système d'indexation sous-jacent Lucene. La dernière trace de documentation de cette fonction semble être ici (me dire si vous trouvez plus récent) : https://www.elastic.co/guide/en/elasticsearch/guide/current/practical-scoring-function.html>. On retrouve les notions de *tf*, *idf* et *normalisation* présentées dans le chapitre *Recherche approchée*, mais avec des formules (légèrement) différentes des versions canoniques : chaque système fait sa petite cuisine pour essayer d'arriver au meilleur résultat.

En lisant les explications sur la *Practical Scoring Function*, on constate donc que pour chaque terme :

- la valeur de `idf` est $log(1 + \frac{N-n}{n})$, *N* étant le nombre total de documents, *n* le nombre de documents contenant le terme
- le `tf` est la fréquence normalisée de manière simplifiée par rapport à un calcul cosinus exact, l'idée étant toujours de ne pas surestimer les longs documents.
- le terme est multiplié par un facteur dite de *boost*

Pour étudier cela concrètement, prenons un exemple. Nous pouvons observer avec l'API `_explain` d'Elasticsearch le calcul du score pour un film donné et pour une requête donnée. Prenons, par exemple, dans l'index *movies*, les films dont le titre contient `life`, comme ci-dessous :

```
{
  "query": {
    "match": {
      "fields.title": "life"
    }
  }
}
```

Chaque film a son propre score, qui dépend de l'index, de la requête, et de la représentation du film dans le document indexé. Pour obtenir le détail de ce score, on transmet la requête non pas à l'API `_search` mais à l'URL `movies/_explain/<id_doc>`. Par exemple, le score du film des Monty Python, « Life of Brian » (La vie de Brian, en français), dont l'identifiant est 2232, est obtenu avec l'URL `movies/_explain/2232` comme le montre la Fig. 8.1.

Dans la fenêtre droite, le résultat contient un objet `explanation` qui détaille les paramètres du classement. Il est notamment indiqué qu'il est obtenu par la formule $boost \times idf \times tf$, ce qui devrait vous rappeler la méthode présentée dans le chapitre *Recherche approchée*. En détaillant, on voit que Elasticsearch utilise une fonction de score qui utilise 3 facteurs.

- *boost* vaut 2,2 (pour le *boosting*, voir ci-dessous)
- *idf* vaut 5,167, valeur obtenue en considérant que 28 films sur les 4 999 contiennent le terme *life*, et $ln(1 + (4999 - 28 + 0,5)/(28 + 0,5)) = 5,167$.
- enfin *tf* vaut 0,434, calcul basé sur une fréquence de 1 (*life* apparaît une fois dans le titre), la présence de 3 termes dans le titre et des facteurs de normalisation dont la taille moyenne d'un titre (2,7). Je vous laisse tenter d'éclaircir le détail de ces calculs.

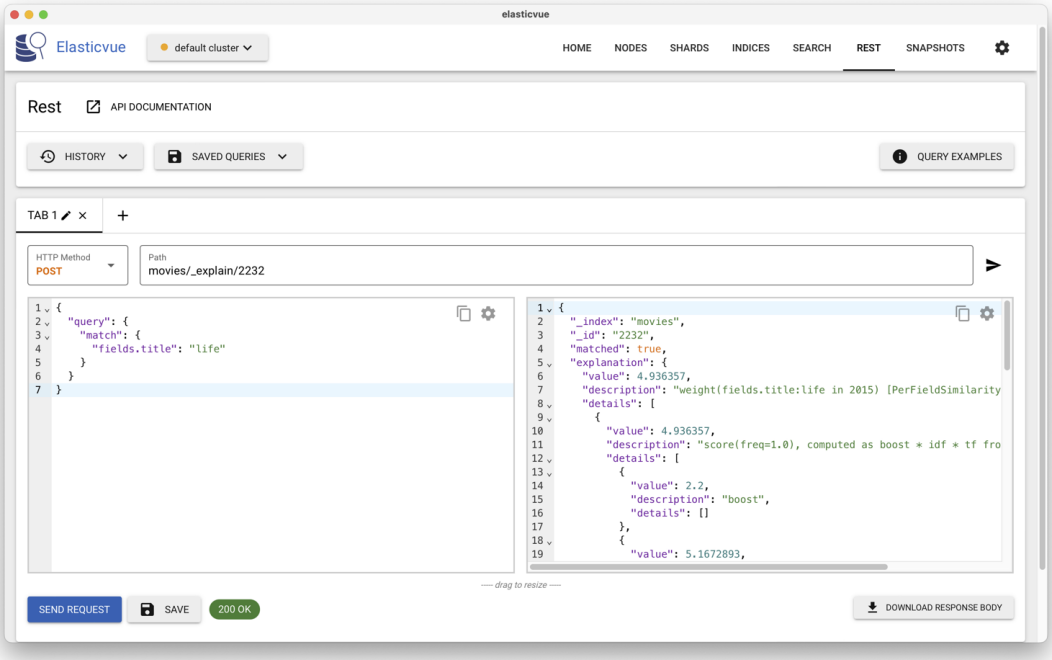


Fig. 8.1 – Obtenir l'explication d'un classement avec ElasticVue

8.3.2 Le boosting

Quand on effectue des recherches sur plus d'un champ, il peut rapidement devenir pertinent de donner davantage de poids à l'un ou l'autre de ces champs, de façon à améliorer les résultats de recherche. Par exemple, il peut être tentant d'indiquer qu'une correspondance (*match*) dans le titre d'un document vaut 2 fois plus qu'une correspondance dans n'importe quel autre champ. C'est ce que l'on appelle en anglais le *boosting*, cela autorise la modification du score calculé par Elasticsearch en vue de rendre les résultats plus pertinents (pour les utilisateurs d'un système donné).

La valeur du *boost* peut-être spécifiquement introduite dans la requête. Voici comment on *booste* d'un facteur de 2 la requête précédente (ce qui a peu d'intérêt puisqu'il y a un seul terme, mais nous verrons ensuite comment *booster* chaque terme individuellement).

```
 {
"query": {
"match": {
        "fields.title": {
          "query": "life",
          "boost": 2
        }
}
}
}
```

Le résultat du `_explain` montre que le *boost* pris en compte dans le calcul du score a doublé par rapport à

la version précédente.

Note : Pourquoi ElasticSearch affiche-t-il des valeurs de *boost* qui semblent supérieures aux valeurs d'entrée ? Parce que la valeur affichée tient compte de facteurs de normalisation du terme. Il est difficile de détailler les calculs, mais l'important est la valeur relative qui a effectivement doublé.

Saisissez la commande suivante et observez la position d'American Graffiti (réalisé par G. Lucas) dans le classement, avec et sans l'option « boost ». Que se passe-t-il ?

```
 {
"query": {
 "bool": {
  "should": [
        {
                "match": {
                "fields.title": {
                "query": "Star Wars",
                "boost": 4
                }
                }
        },
        {
                "match": {
        "fields.directors": {
                "query": "George Lucas"
                        }
                }
        }
     ]
}
        },
        "fields": ["fields.title"],
        "_source": false
}
```

Avec le boosting, American Graffiti est 9e, derrière Bride Wars, mieux classé car le boosting favorise la correspondance avec (au moins) un des mots du titre. Sans le boosting, American Graffiti arrive en cinquième position.

Si on peut associer du boosting positif à certaines valeurs de certains champs, on peut rejeter vers le bas du classement des documents qui contiennent certaines valeurs pour d'autres champs. On peut combiner boosting positif et boosting négatif (évidemment pour des champs différents).

CHAPITRE 9

Traitements par lot

Nous avons étudié jusqu'à présent les recherches, exactes ou approchées, permettant de retrouver des informations en temps réel, ou du moins avec un délai de réponse compatible avec une utilisation interactive. Un autre scénario pour la gestion de très grandes collections est celui d'un *traitement par lot* qui consiste à extraire des informations à partir d'un sous-ensemble important de la collection voire même de la collection entière. C'est le cas par exemple pour la construction d'indicateurs statistiques, pour l'apprentissage de modèles IA, ou la production d'une collection dérivée, typiquement un index.

Pour ce type de scénario on ne s'attend pas à des temps de réponses de quelques secondes, au vu de la taille des données considérées. Les temps de parcours et de calcul peuvent prendre des minutes, des heures ou même des jours. Les problématiques sont plutôt de contrôler le temps d'exécution en parallélisant les calculs, d'une part, et de s'assurer de la terminaison de ces calculs même si des pannes surviennent, d'autre part.

Nous étudions dans ce chapitre la première problématique avec les modèles de *chaîne de traitement* (par traduction de « *data processing pipelines* » ou, simplement, *workflow*). Le principe général est de soumettre tous les documents d'une collection à une séquence d'opérateurs, comme par exemple :

- un *filtrage*, en ne gardant le document que s'il satisfait certains critères ;
- une *restructuration*, en changeant la forme du document ;
- une *annotation*, par ajout au document de propriétés calculées ;
- un *regroupement* avec d'autres documents sur certains critères ;
- des *opérations d'agrégation* sur des groupes de documents.

Les opérateurs qui nous intéressent sont dits *opérateurs de second ordre*. Contrairement aux opérateurs classiques qui s'appliquent directement à des données, un opérateur de second ordre prend des fonctions en paramètre et applique ces fonctions à des données au cours d'un traitement immuable (par exemple un parcours séquentiel). Pour le traitement de données massives, on s'intéresse plus particulièrement aux opérateurs *parallélisables* qui permettent une forme d'industrialisation des calculs en les distribuant sur plusieurs machines.

Nous allons développer longuement ces notions un peu complexes. Pour commencer nous présentons deux

des opérateurs qui sont à l'origine des systèmes de traitement de données massives, Map et Reduce. Nous introduirons ensuite d'autres opérateurs avec un langage simple d'utilisation, Pig latin. Les futurs chapitres viseront à expliquer la distribution des données et des calculs en vue d'obtenir une parallélisation efficace, ce que nous résumerons avec une étude de Spark, sans doute le principal système distribué à l'heure actuelle.

Nous nous en tenons donc pour l'instant (à l'exception d'une présentation intuitive dans la première session) au contexte centralisé (un seul serveur), ce qui permet de se familiariser avec les concepts et la pratique dans un cadre simple.

9.1 S1 : MapReduce démystifié

Supports complémentaires :
- Présentation: MapReduce expliqué avec les mains
- Vidéo présentant MapReduce

Commençons par expliquer que MapReduce, en tant que modèle de calcul, ce n'est pas grand chose ! Cette section propose une découverte « avec les mains », en étudiant comment cuisiner quelques recettes simples avec un robot MapReduce. La fin de la section récapitule en termes plus informatiques.

9.1.1 Un jus de pomme MapReduce

Ecartons-nous de l'informatique pour quelques moments. Cela nous laisse un peu de temps pour faire un bon jus de pomme. Vous savez faire du jus de pomme ? C'est simple :
- l'épluchage : il faut éplucher les pommes une par une ;
- le pressage : on met toutes les pommes dans un pressoir, et on récupère le jus.

Le processus est résumé sur la Fig. 9.1. Observons le cuisinier. Il a un tas de pomme à gauche, les prend une par une, épluche chaque pomme et place la pomme épluchée dans un tas à droite. Quand toutes les pommes sont épluchées (et pas avant !), on peut commencer la seconde phase.

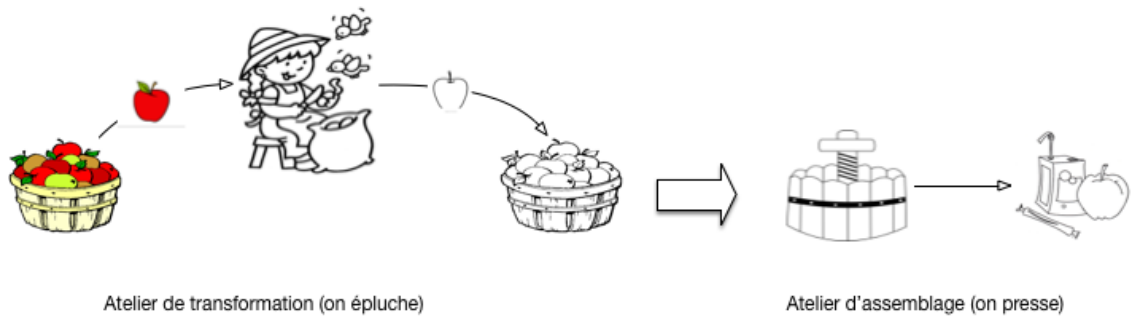


Fig. 9.1 – Les deux phases de la confection de jus de pomme

Comme notre but est de commencer à le formaliser en un modèle que nous appellerons à la fin *MapReduce*, nous distinguons précisément la frontière entre les deux phases, et les tâches effectuées de chaque côté.
- À gauche, nous avons donc *l'atelier de transformation* : il consiste en un agent, l'éplucheur, qui prend une pomme dans son panier à gauche, produit une pomme épluchée dans un second panier à droite, et répète la même action jusqu'à ce que le panier de gauche soit vide.

— À droite nous avons *l'atelier d'assemblage* : on lui confie un tas de pommes épluchées et il produit du jus de pomme.

Nous pouvons déjà tirer deux leçons sur les caractéristiques essentielles de notre processus élémentaire. La première porte sur l'atelier de transformation qui applique une opération individuelle à chaque produit.

Leçon 1 : l'atelier de transformation est centré sur les pommes

Dans l'atelier de transformation, les pommes sont épluchées *individuellement* et *dans n'importe quel ordre*.

L'éplucheur ne sait pas combien de pommes il a à éplucher, il se contente de piocher dans le panier tant que ce dernier n'est pas vide. De même, il ne sait pas ce que deviennent les pommes épluchées, il se contente de les transmettre au processus général.

La seconde leçon porte sur l'atelier d'assemblage qui, au contraire, applique une transformation aux produits *regroupés* : ici, des tas de pommes.

Leçon 2 : l'atelier d'assemblage est centré sur les tas de pommes

Dans l'atelier d'assemblage, on applique des transformations à des *ensembles* de pommes.

Tout celà est assez élémentaire, voyons si nous pouvons faire mieux en introduisant une première variante. Au lieu de cuire des pommes entières, on préfère les couper au préalable en quartiers. La phase d'épluchage devient une phase d'épluchage/découpage.

Cela ne change pas grand chose, comme le montre la Fig. 9.2. Au lieu d'avoir deux tas identiques à gauche et à droite avec des pommes, le cuisinier a un tas avec p pommes à gauche et un autre tas avec $4p$ quartiers de pommes à droite.

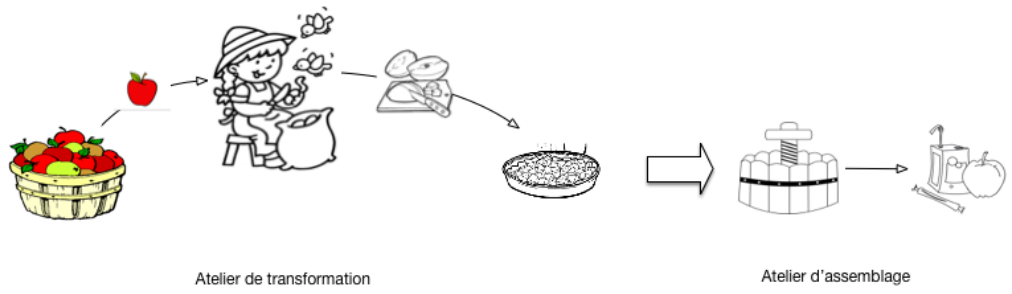


Fig. 9.2 – Avec des quartiers de pomme

Cela nous permet quand même de tirer une troisième leçon.

Leçon 3 : la transformation peut modifer le nombre et la nature des produits

La première phase n'est pas limitée à une transformation un pour un des produits consommés. Elle peut prendre en entrée des produits d'une certaine nature (des pommes), sortir des produits d'une autre nature (des quartiers de pomme épluchées), et il peut n'y avoir aucun rapport fixe entre le nombre de produits en

sortie et le nombre de produits en entrées (on peut jeter des pommes pourries, couper une petite pomme en 4 ou en 6, une grosse pomme en 8, etc.)

Nous avons notre premier processus MapReduce, et nous avons identifié ses caractéristiques principales. Il devient possible de montrer comment passer à grande échelle dans la production de jus de pomme, sans changer le processus.

9.1.2 Beaucoup de jus de pomme

Votre jus de pomme est très bon et vous devez en produire beaucoup : une seule personne ne suffit plus à la tâche. Heureusement la méthode employée se généralise facilement à une brigade de *n* éplucheurs.

— répartissez votre tas de pommes en *n* sous-tas, affectés chacun à un éplucheur ;
— chaque éplucheur effectue la tâche d'épluchage/découpage comme précédemment ;
— regroupez les quartiers de pomme et pressez-les.

Il se peut qu'un pressoir ne suffise plus : dans ce cas affectez *c* pressoirs et répartissez équitablement les quartiers dans chacun. Petit inconvénient : vous obtenez plusieurs fûts de jus de pomme, un par pressoir, avec une qualité éventuellement variable. Ce n'est sans doute pas très grave.

Important : Notez que cela ne marche que grâce aux caractéristiques identifiées par la leçon N° 1 ci-dessus. Si l'ordre d'épluchage était important par exemple, ce ne serait pas si simple car il faudrait faire attention à ce que l'on confie à chaque éplucheur ; *idem* si l'épluchage d'une pomme dépendait de l'épluchage des autres.

La Fig. 9.3 montre la nouvelle organisation de vos deux ateliers. Dans l'atelier de transformation, vous avez *n* éplucheurs qui, chacun, font *exactement* la même chose qu'avant : ils produisent des tas de quartiers de pomme. Dans l'atelier d'assemblage, vous avez *r* pressoirs : un au minimum, 2, 3 ou plus selon les besoins. Attention : il n'y aucune raison d'imposer comme contrainte que le nombre de pressoirs soit égal au nombre de éplucheurs. Vous pourriez avoir un très gros pressoir qui suffit à occuper 10 éplucheurs.

Vous avez parallélisé votre production de jus de pomme. Remarque essentielle : vous n'avez pas besoin de recruter des éplucheurs avec des compétences supérieures à celles de votre atelier artisanal du début. Chaque éplucheur épluche ses pommes et n'a pas besoin de se soucier de ce que font les autres, à quel rythme ils travaillent, etc. Vous n'avez pas non plus besoin d'un matériel nouveau et radicalement plus cher. En d'autres termes vous avez défini un processus qui passe sans douleur à un stade **industriel** où seulent comptent l'organisation et l'affecation de ressources, pas l'augmetation des compétences.

Vous pouvez même prétendre que la rentabilité économique est préservée. Si un éplucheur épluche 50 Kgs de pomme par jour, 10 éplucheurs (avec le matériel correspondant) produiront 500 Kgs par jour ! C'est aussi une question de matériel à affecter au processus : il est clair que si vous n'avez qu'un seul économe (le couteau qui sert à éplucher) ça ne marche pas. Mais si la production de jus de pomme est rentable avec un éplucheur, elle le sera aussi pour 10 avec le matériel correspondant. Nous dirons que le processus est *scalable*, et cela vaut une quatrième leçon.

Leçon 4 : parallélisation et scalabilité (linéaire)

La production de jus de pomme est parallélisable et proportionnelle aux ressources (humaines et matérielles)

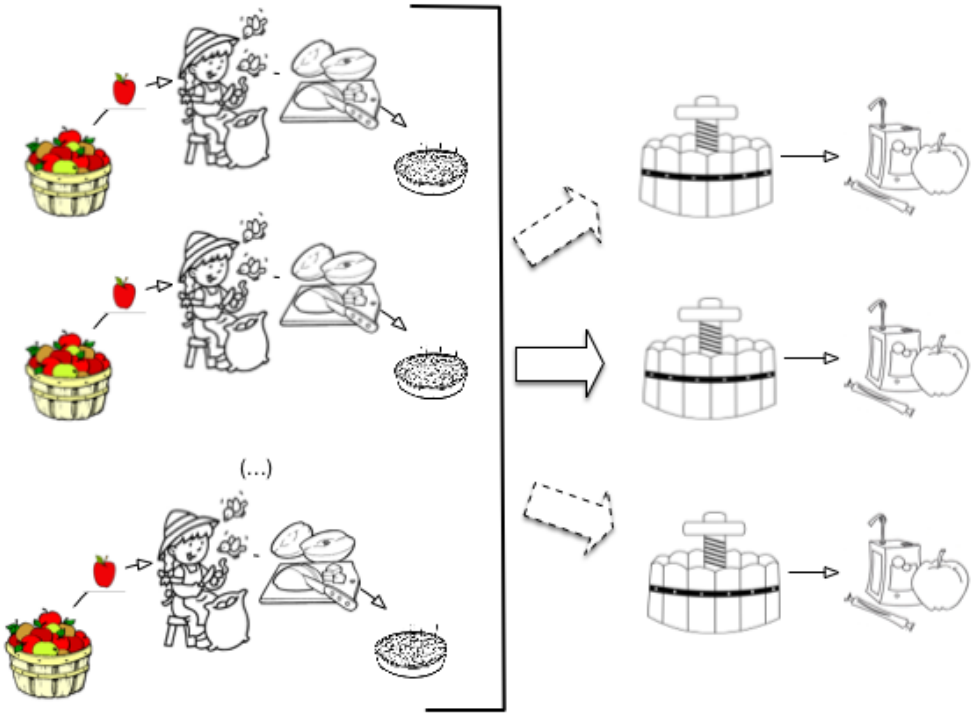


Fig. 9.3 – Parallélisation de la production de jus de pomme

affectées.

Votre processus a une seconde caractéristique importante (qui résulte de la remarque déjà faire que les épluchеurs sont indépendants les uns des autres). Si un épluchеur éternue à répétition sur son tas de pomme, s'il épluche mal ou si les quartiers de pomme à la fin de l'épluchage tombent par terre, cela ne remet pas en cause l'ensemble de la production mais seulement la petite partie qui lui était affectée. Il suffit de recommencer cette partie-là. De même, si un pressoir est mal réglé, cela n'affecte pas le jus de pomme préparé dans les autres et les dégâts restent *locaux*.

Leçon 5 : le processus est robuste

Une défaillance affectant la production de jus de pomme n'a qu'un effet *local* et ne remet pas en cause *l'ensemble* de la production.

Les leçons 4 et 5 sont les deux propriétés essentielles de MapReduce, modèle de traitement qui se prête bien à la distribution des tâches. Si on pense en terme de puissance ou d'expressivité, cela reste quand même très limité. Peut-on faire mieux que du jus de pomme ? Oui, en adoptant la petite généralisation suivante.

9.1.3 Jus de fruits MapReduce

Pourquoi se limiter au jus de pomme ? Si vous avez une brigade d'éplucheurs de premier plan et des pressoirs efficaces, vous pouvez aussi envisager de produire du jus d'orange, du jus d'ananas, et ainsi de suite. Le processus consistant en une double phase de transformation *individuelle* des ingrédients, puis d'élaboration collective convient tout à fait, à une adaptation près : comme on ne peut pas presser ensemble des oranges et les pommes, *il faut ajouter une étape initiale de tri/regroupement* dans l'atelier d'assemblage.

En revanche, pendant la première phase, on peut soumettre un tas indifférencé de pommes/oranges/ananas à un même éplucheur. L'absence de spécialisation garantit ici une meilleure utilisation de votre brigade, une meilleure adaptation aux commandes, une meilleure réactivité aux incidents (pannes, blessures, cf. exercices).

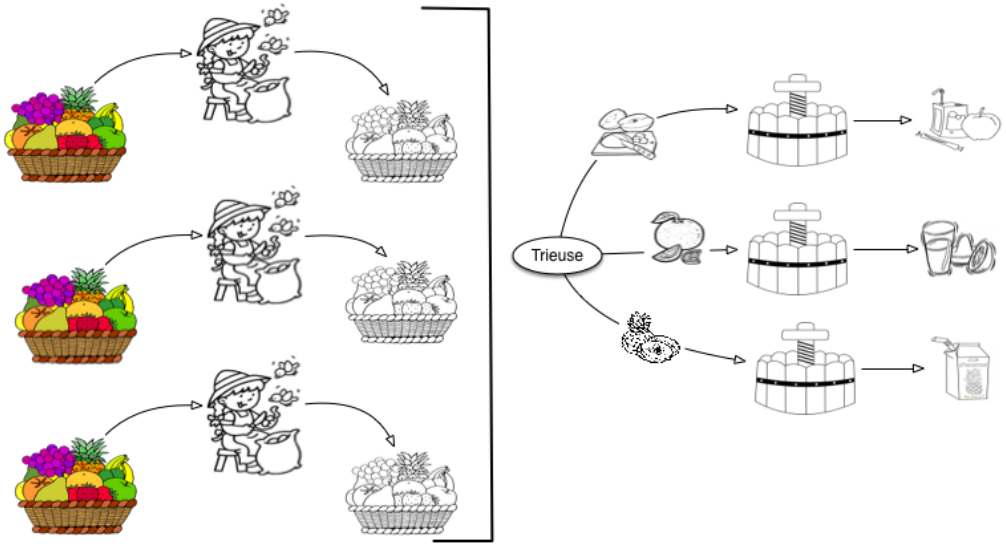


Fig. 9.4 – Production de jus de fruits en parallèle

La Fig. 9.4 montre une configuration de vos ateliers de production de jus de fruit, avec 4 ateliers de transformation, et 1 atelier d'assemblage.

Résumons : chaque éplucheur a à sa gauche un tas de fruits (pommes, oranges, ananas). Il épluche chaque ingrédient, un par un, et les transmet à l'atelier d'assemblage. Cet atelier d'assemblage comporte maintenant une trieuse qui envoie chaque fruit vers un tas homogène, transmis ensuite à un pressoir dédié. *Le processus reste parallélisable, avec les mêmes propriétés de scalabilité que précedemment.* Nous avons simplement besoin d'une opération supplémentaire.

Une question non triviale en générale est celle du critère de tri et de regroupement. Dans le cas des pommes, oranges et ananas, on peut supposer que l'opérateur fait facilement la distinction visuellement. Pour des cas plus subtils (vous distinguez une variété de pomme *Reinette* d'une *Jonagold* ?) il nous faut une méthode plus robuste. Les produits fournis par l'atelier d'assemblage doivent être *étiquetés* au préalable par l'opérateur de l'atelier de transformation.

Leçon 6 : phase de tri / regroupement, étiquetage

Si les produits doivent être traitées par catégorie, il faut ajouter une phase de tri / regroupement au début de l'atelier d'assemblage. Le tri s'appuie sur une *étiquette* associée à chaque produit en entrée, indiquant le

groupe d'appartenance.

Et finalement, comment faire si nous mettons en place *plusieurs* ateliers d'assemblage ? Deux choix sont possibles :

— *spécialiser* chaque atelier à une ou plusieurs catégories de fruits ;
— *ne pas spécialiser* les ateliers, et simplement répliquer l'organisation de la Fig. 9.4 où un atelier d'assemblage sait presser tous les types de fruits.

Les deux choix se défendent sans doute (cf. exercices), mais dans le modèle MapReduce, c'est la spécialisation (choix 1) qui s'impose, pour des raisons qui tiennent aux propriétés des méthodes d'agrégation de données, pas toujours aussi simple que de mélanger deux jus d'oranges.

Dans une configuration avec plusieurs ateliers d'assemblage, chacun est donc spécialisé pour traiter une ou plusieurs catégories. Bien entendu, il faut s'assurer que chaque catégorie est prise en charge par un atelier. C'est le rôle d'une nouvelle machine, le *répartiteur*, illustré par la Fig. 9.5. Nous avons deux ateliers d'assemblage, le premier prenant en charge les pommes et les oranges, et le second les ananas.

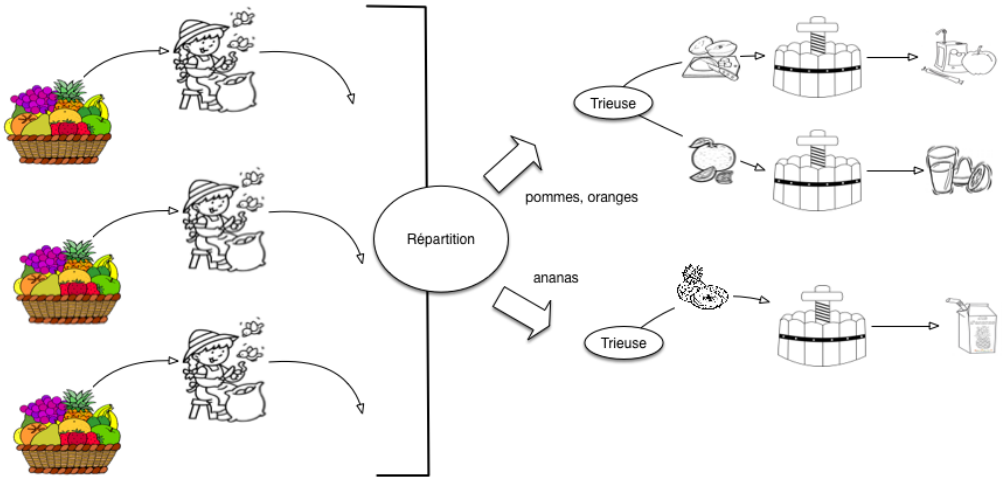


Fig. 9.5 – Production en parallèle, avec répartition vers des ateliers d'assemblage spécialisés

C'est fini ! Cette fois nous avons une métaphore complète d'un processus MapReduce dans un contexte Cloud/Big Data. Tirons une dernière leçon avant de le reformuler en termes abstraits/informatiques.

Leçon 7 : répartition vers les ateliers d'assemblage

Si nous avons plusieurs ateliers d'assemblage, il faut mettre en place une opération de répartition qui envoie chaque type de fruit vers l'atelier spécialisé. Cette opération doit garantir que chaque type de fruit a son atelier.

On peut envisager de nombreuses variantes qui ne remettent pas en cause le modèle global d'exécution et de traitement. Une réflexion sur ces variantes est proposée en exercice.

9.1.4 Le modèle MapReduce

Il est temps de prendre un peu de hauteur (?) pour caractériser le modèle MapReduce en termes informatiques.

Important : Pour l'instant, nous nous concentrons uniquement sur la compréhension de ce que *spécifie* un traitement MapReduce, et pas sur la manière dont ce traitement est *exécuté*. Nous savons par ce qui précède qu'il est possible de le paralléliser, mais il est également tout à fait autorisé de l'exécuter sur une seule machine en deux étapes. C'est le scénario que nous adoptons pour l'instant, jusqu'au moment où nous aborderons les calculs distribués.

Le principe de MapReduce est ancien et provient de la programmation fonctionnelle. Il se résume ainsi : étant donné une collection *d'items*, on applique à chaque item un processus de transformation individuelle (phase dite « de Map ») qui produit des *valeurs intermédiaires* étiquetées. Ces valeurs intermédiaires sont regroupées par étiquette et soumises à une fonction d'assemblage (on parlera plus volontiers d'agrégation en informatique) appliquée à chaque groupe (phase dite « de Reduce »). La phase de Map correspond à notre atelier de transformation, la phase de Reduce à notre atelier d'assemblage.

Reprenons le modèle dans le détail.

Notion d'item en entrée (document)

Un *item d'entrée* est une valeur quelconque apte à être soumise à la fonction de transformation. Dans tout ce qui suit, nos items d'entrée seront des *documents structurés*.

Dans notre exemple culinaire, les items d'entrées sont les fruits « bruts » : pommes, oranges, ananas, etc. La transformation appliquée aux items est représentée par une fonction de Map.

Notion de fonction de Map

La fonction de Map, notée F_{map} est appliquée à chaque item de la collection, et produit zéro, une ou plusieurs valeurs dites « intermédiaires », placées dans un *accumulateur*.

Dans notre exemple, F_{map} est l'épluchage. Pour un même fruit, on produit plusieurs valeurs (les quartiers), voire aucune si le fruit est pourri. L'accumulateur est le tas à droite du cuisinier.

Il est souvent nécessaire de *partitionner* les valeurs produites par le *map* en plusieurs groupes. Il suffit de modifier la fonction F_{map} pour qu'elle émette non plus une valeur, mais associe chaque valeur au groupe auquel elle appartient. F_{map} produit, pour chaque item, une paire *(k, v)*, où *k* est l'identifiant du groupe et *v* la valeur à placer dans le groupe. L'identifiant du groupe est déterminé à partir de l'item traité (c'est ce que nous avons informellement appelé « étiquette » dans la présentation de l'exemple.)

Dans le modèle MapReduce, on appelle *paire intermédiaire* les données produites par la phase de Map.

Notion de paire intermédiaire

Une paire intermédiaire est produite par la fonction de Map ; elle est de la forme *(k, v)* où *k* est l'identifiant

(ou *clé*) d'un groupe et *v* la valeur extraite de l'item d'entrée par F_{map}.

Pour notre exemple culinaire, il y a trois groupes, et donc trois identifiants possibles : `pomme`, `orange`, `ananas`.

À l'issue de la phase de Map, on dispose donc d'un ensemble de paires intermédiaires. Chaque paire étant caractérisée par l'identifiant d'un groupe, on peut constituer les groupes par regroupement sur la valeur de l'identifiant. On obtient des *groupes intermédiaires*.

Notion de groupe intermédiaire

Un *groupe intermédiaire* est l'ensemble des valeurs intermédiaires associées à une même valeur de clé.

Nous aurons donc le groupe des quartiers de pomme, le groupe des quartiers d'orange, et le groupe des rondelles d'ananas. On entre alors dans la seconde phase, dite de Reduce. La transformation appliquée à chaque groupe est définie par la fonction de Reduce.

Notion de fonction de Reduce

La fonction de Reduce, notée F_{red}, est appliquée à chaque groupe intermédiaire et produit une valeur finale. L'ensemble des valeurs finales (une pour chaque groupe) constitue le résultat du traitement MapReduce.

Résumons avec la Fig. 9.6, et plaçons-nous maintenant dans le cadre de nos bases documentaires. Nous avons une collection de documents $d_1, d_2, \cdots, d_n$. La fonction F_{map} produit des paires intermédiaires sous la forme de documents d_i^j, dont l'identifiant (j) désigne le groupe d'appartenance. Notez les doubles flêches : un document en entrée peut engendrer plusieurs documents en sortie du *map*. F_{map} place chaque d_i^j dans un groupe $G_j, j \in [1, k]$. Quand la phase de *map* est terminée (et pas avant !), on peut passer à la phase de *reduce* qui applique successivement F_{red} aux documents de chaque groupe. On obtient, pour chaque groupe, une valeur (un document de manière générale) v_j.

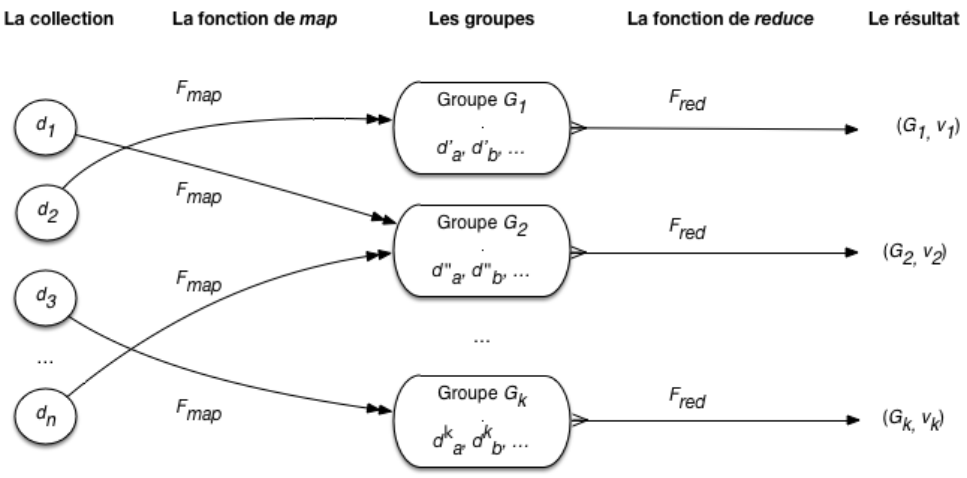


Fig. 9.6 – MapReduce (en centralisé)

Voici pour la théorie. En complément, notons dès maintenant que ce mécanisme a quelques propriétés intéressantes :

— il est *générique* et s'applique à de nombreux problèmes,
— il se *parallélise* très facilement ;
— il est assez tolérant aux pannes dans une contexte distribué.

La première propriété doit être fortement relativisée : on ne fait quand même pas grand chose, en termes d'algorithmique, avec MapReduce et il ne faut *surtout* par surestimer la capacité de ce modèle de calcul à prendre en charge des traitements complexes.

Cette limitation est compensée par la parallélisation et la tolérance aux pannes. Pour ces derniers aspects, il est important que chaque document soit traité *indépendamment* du contexte. Concrètement, l'application de F_{map} à un document *d* doit donner un résultat qui ne dépend ni de la position de *d* dans la collection, ni de ce qui s'est passé avant ou après dans la chaîne de traitement, ni de la machine ou du système, etc. En pratique, cela signifie que la fonction F_{map} ne conserve pas un *état* qui pourrait varier et influer sur le résultat produit quand elle est appliquée à *d*.

Si cette propriété n'était pas respectée, on ne pourrait pas paralléliser et conserver un résultat invariant. Si F_{map} dépendait par exemple du système d'exploitation, de l'heure, ou de n'importe quelle variable extérieure au document traité (un *état*), l'exécution en parallèle aboutirait à des résultats non déterministes.

9.1.5 Concevoir un traitement MapReduce

Quelques conseils pour finir sur la conception d'un traitement MapReduce. En un mot : c'est très simple, à condition de se poser les bonnes questions et de faire preuve d'un minimum de rigueur.

Les questions à se poser

Questions pour la phase de Map :

— Il faut être clair sur la nature des documents que l'on traite en entrée. D'où viennent-ils, quelle est leur structure ? Un traitement MapReduce s'applique à un flux de documents, on ne peut rien faire si on ne sait pas en quoi ils consistent.
— Quels sont les groupes que je veux constituer ? Combien y en a-t-il ? Comment les identifier (valeur de clé identifiant un groupe, l'étiquette) ? Comment déterminer le ou les étiquettes d'un ou de plusieurs groupes dans un document en entrée ?
— Quelles sont les valeurs intermédiaires que je veux produire à partir d'un document et placer dans des groupes ? Comment les produire à partir d'un document ?

Ces questions sont nécessaires (et pratiquement suffisantes) pour savoir ce que doit faire la fonction de Map. Pour la fonction de Reduce, c'est encore plus simple :

— Quelle est la nature de l'agrégation qui va prendre un groupe et produire une valeur finale ?

Et c'est tout, il reste à traduire en termes de programmation.

Un peu de rigueur

Un traitement MapReduce se spécifie sous la forme de deux fonctions

- La fonction de Map prend *toujours* en entrée *un* (un seul) document ; elle produit *toujours* des paires *(k, v)* où *k* est l'étiquette (la clé) du groupe et *v* la valeur intermédiaire.
- La fonction de Reduce prend *toujours* en entrée une paire *(k, list(v))*, où *k* est l'étiquette (la clé) du groupe et *list(v)* la liste des valeurs du groupe.

Spécifier un traitement, c'est donc *toujours* définir deux fonctions avec les caractéristiques ci-dessus. Le corps de chaque fonction doit indiquer, respectivement :

- comment on produit des paires *(k, v)* à partir d'un document (fonction de Map, transformation) ;
- comment on on produit une valeur agrégée *V* à partir d'uune paire *(k, list(v))*.

Pratiques, réflechissez, vérifiez que vous avez bien compris !

9.1.6 Quiz

9.2 S2 : MapReduce et CouchDB

Supports complémentaires

- Vidéo présentant la programmation MapReduce avec CouchDB

CouchDB propose un moteur d'exécution MapReduce, avec des fonctions javascript, mais dans un but un peu particulier : celui de créer des collections structurées dérivées par application d'un traitement MapReduce à une collection stockée. De telles collections sont appelées *vues* dans CouchDB. Leur contenu est matérialisé et maintenu incrémentalement.

Cette section introduit la notion de vue CouchDB, mais se concentre surtout sur la définition des fonctions de Map et de Reduce qui est rendue très facile grâce à l'interface graphique de CouchDB. Vous devriez avoir importé la collection des films dans CouchDB dans une base de données *movies*. C'est celle que nous utilisons, comme d'habitude, pour les exemples.

9.2.1 La notion de vue dans CouchDB

Comme dans beaucoup de systèmes NoSQL, une collection CouchDB n'a pas de schéma et les documents peuvent donc avoir n'importe quelle structure, ce qui ne va pas sans soulever des problèmes potentiels pour les applications. CouchDB répond à ce problème de deux manières : par des fonctions dites de validation, et par la possibilité de créer des *vues*.

Une vue dans CouchDB est une collection dont les éléments ont la forme (clé, document). Ces éléments sont *calculés* par un traitement MapReduce, stockés (on parle donc de matérialisation, contrairement à ce qui se fait en relationnel), et maintenus en phase avec la collection de départ. Les vues permettent de structurer et d'organiser un contenu.

La définition d'une vue est stockée dans CouchDB sous la forme de documents JSON dits « documents de conception » (*design documents*). Pour tester une nouvelle définition, on peut aussi créer des vues temporaires : c'est ce qui nous intéresse directement, car nous allons pouvoir tester le MapReduce de CouchDB grâce à l'interface de définition de ces vues temporaires

Accédez à l'interface d'administration de CouchDB à l'URL `_utils`, puis choisissez la base des films (que vous devez avoir chargé en expérimentant Docker). Vous devriez avoir l'affichage de la Fig. 9.7.

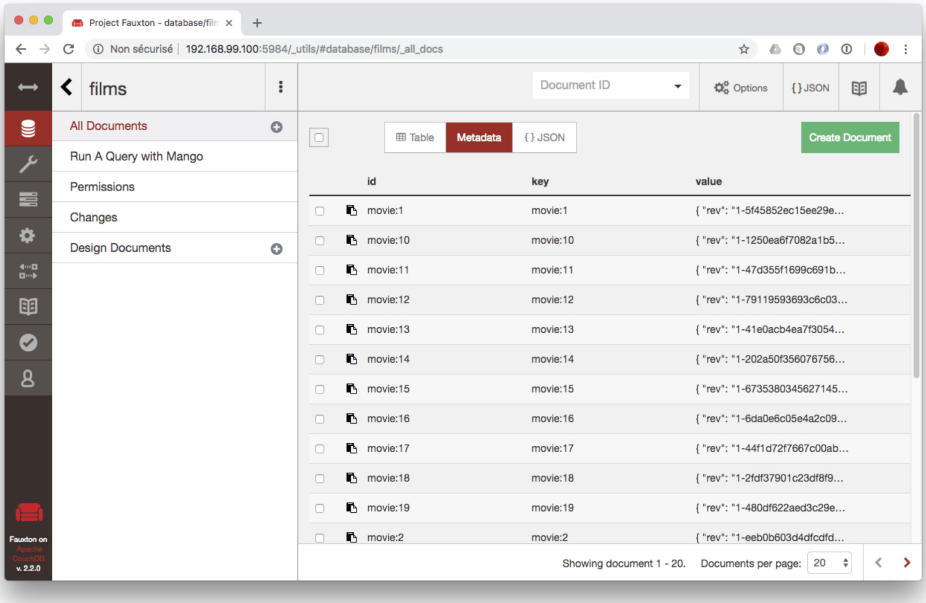


Fig. 9.7 – La collections *films* vues par l'interface de CouchDB

Dans le menu de gauche, choissisez l'option « New view » dans le menu « Design documents ». Dans le menu déroulant « Reduce », choisissez l'option « Custom » pour indiquer que vous souhaitez définir votre propre fonction de Reduce.

On obtient un formulaire avec deux fenêtres pour, respectivement, les fonctions de Map et de Reduce (Fig. 9.8).

Avant de passer aux fonctions MapReduce, donnez un nom à votre vue dans le champ en haut à droite.

9.2.2 Les fonctions Map et Reduce

Avec CouchDB, la fonction de Map est obligatoire, contrairement à la fonction de Reduce. La fonction Map par défaut ne fait rien d'autre qu'émettre les documents de la base, avec une clé `null`.

```
function(doc) {
  emit(null, doc);
}
```

Toute fonction de Map prend un document en entrée, et appelle la fonction `emit` pour produire des clés intermédiaires. Produisons une première fonction de Map qui produit une vue dont la clé est le titre du document, et la valeur le metteur en scène.

```
function(doc)
{
```

(suite sur la page suivante)

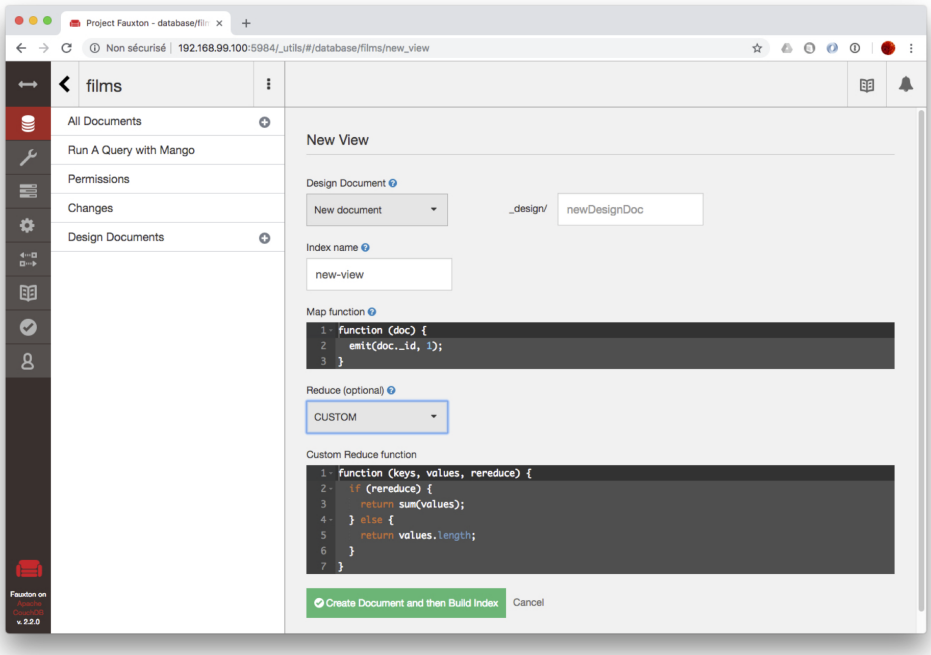


Fig. 9.8 – Définition des vues temporaires.

(suite de la page précédente)

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

À vous de tester. Vous devriez (en actionnant le bouton « Create and build index ») obtenir l'affichage de la Fig. 9.9.

Important : Pour être sûr d'activer la fonction de Reduce, cochez la case « Reduce » dans l'interface de CouchDB. Cette case se trouve dans la fenêtre des options (montrée sur la Fig. 9.9).

Un deuxième exemple : la vue produit la liste des acteurs (c'est la clé), chacun associé au film dans lequel il joue (c'est la valeur). C'est un cas où la fonction de Map émet plusieurs paires intermédiaires.

```
 function(doc)
 {
   for  (i = 0; i < doc.actors.length; i++) {
      emit({"prénom": doc.actors[i].first_name, "nom": doc.actors[i].last_name},␣
↪doc.title) ;
   }
}
```

On peut remarquer que la clé peut être un document composite. À vous de tester cette nouvelle vue. Vous remarquerez sans doute que les documents de la vue sont *triés* sur la clé. C'est un effect indirect de l'organisation sous forme d'arbre-B, qui repose sur l'ordre des clés indexées. Pour MapReduce, le fait que les

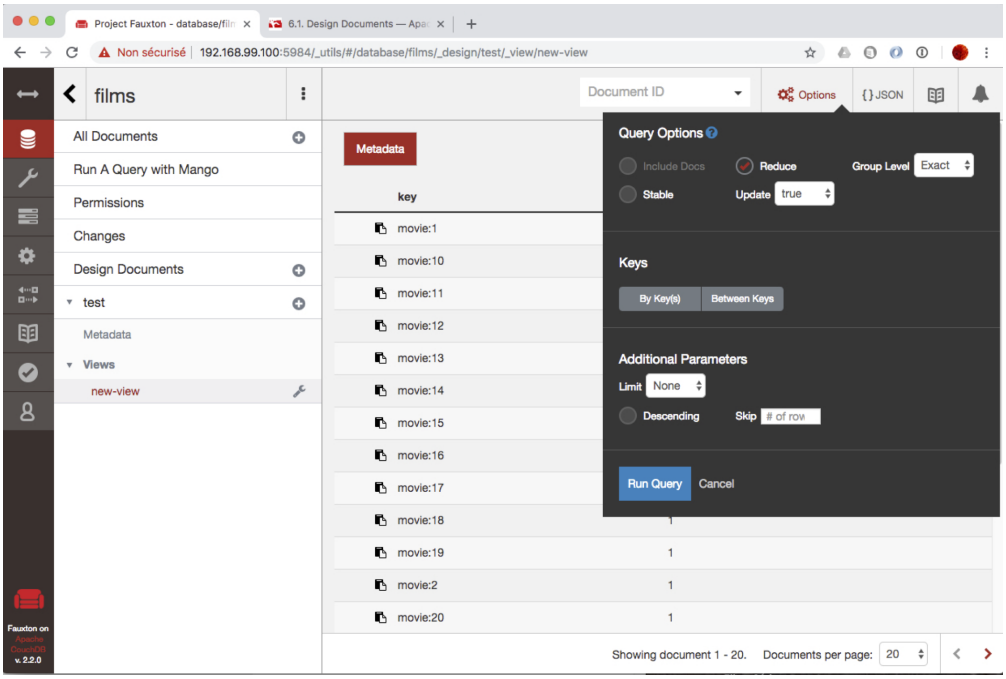


Fig. 9.9 – Définition et test d'une vue

paires intermédiaires soient triées facilite les regroupements sur la clé puisque les documents à regrouper sont consécutifs dans la liste.

Vérifiez : cherchez dans la liste des documents de la vue Bruce Willis par exemple (ou Adam Driver qui est plus près de la première page). Vous remarquerez qu'il apparaît pour chaque film dans lequel il joue un rôle, et que toutes ces occurrences sont en séquence dans la liste. Pour effectuer ce regroupement, on applique une fonction de Reduce. La voici :

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

À vous de continuer en expérimentant l'interface et en étudiant le résultat intermédiaire (sans appliquer de fonction Reduce) puis le résultat final.

9.2.3 Mise en pratique

Exercice MEP-S2-1 : quelques programmes MapReduce à produire

Outre les commandes de découverte de CouchDB décrites précédemment, voici quelques programmes à produire

— Donnez, pour chaque année, le nombre de films parus cette année-là. Puis donnez pour chaque année la liste des titres de ces films.
— Donnez, pour chaque metteur en scène, la liste des films qu'il a réalisés.

9.3 S3 : Spécification de traitements distribués avec Pig

Supports complémentaires
- Diapositives: Le langage Pig latin
- Vidéo de la session Pig latin

MapReduce est un système initialement orienté vers les développeurs qui doivent concevoir et implanter la composition de plusieurs *jobs* pour des algorithmes complexes qui ne peuvent s'exécuter en une seule phase. Cette caractéristique rend les systèmes MapReduce difficilement accessibles à des non-programmeurs. De plus, disposer seulement de deux opérateurs assez rudimentaires n'est pas très satisfaisant pour des traitements complexes.

Très tôt après l'apparition des premières versions de systèmes MapReduce comme Hadoop sont apparus des propositions visant d'une part à définir des langages de plus haut niveau permettant de spécifier des opérations complexes sur les données en limitant la programmation, et d'autre d'étendre la collection des opérateurs disponibles. L'initiative est souvent venue de communautés familières des bases de données et désirant retrouver la simplicité et la « déclarativité » du langage SQL, transposées dans le domaine des chaînes de traitements pour données massives.

Avant d'étudier un système complet avec Spark, nous commençons ici avec le langage Pig latin, une des premières tentatives du genre, une des plus simples, et surtout l'une des plus représentatives des opérateurs de manipulation de données qu'il est possible d'exécuter sous forme de chaînes de traitement conservant la scalabilité et la gestion des pannes.

Pig latin (initialement développé par un laboratoire Yahoo !) est un projet Apache disponible à http://pig.apache.org. Il n'évolue plus depuis longtemps mais la version finale est suffisante pour tester l'effet des opérateurs. Vous avez (au moins) deux possibilités pour l'installation.
- Utilisez la machine Docker https://hub.docker.com/r/hakanserce/apache-pig/
- Ou récupérez la dernière version sous la forme d'une archive compressée et décompressez-la quelque part, dans un répertoire que nous appellerons `pigdir`.

Nous utiliserons directement l'interpréteur de scripts (nommé `grunt`) qui se lance avec :

```
<pigdir>/bin/pig -x local
```

L'option `local` indique que l'on teste les scripts en local, ce qui permet de les mettre au point sur de petits jeux de données avant de passer à une exécution distribuée à grande échelle dans un *framework* MapReduce.

Cet interpréteur affiche beaucoup de messages, ce qui devient rapidement désagréable. Pour s'en débarasser, créer un fichier `nolog.conf` avec la ligne :

```
log4j.rootLogger=fatal
```

Et lancez Pig en indiquant que la configuration des *log* est dans ce fichier :

```
<pigdir>/bin/pig -x local -4 nolog.conf
```

9.3.1 Une session illustrative

Pig applique des *opérateurs* à des *flots de données semi-structurées*. Le flot initial (en entrée) est constituée par lecture d'une source de données quelconque contenant des documents qu'il faut structurer selon le modèle de Pig, à peu de choses près comparable à ce que proposent XML ou JSON.

Dans un contexte réel, il faut implanter un chargeur de données depuis la source. Nous allons nous contenter de prendre un des formats par défaut, soit un fichier texte dont chaque ligne représente un document, et dont les champs sont séparés par des tabulations. Nos documents sont des entrées bibliographiques d'articles scientifiques que vous pouvez récupérer à http://b3d.bdpedia.fr/files/journal-small.txt. En voici un échantillon.

```
2005    VLDB J. Model-based approximate querying in sensor networks.
1997    VLDB J. Dictionary-Based Order-Preserving String Compression.
2003        SIGMOD Record   Time management for new faculty.
2001    VLDB J. E-Services - Guest editorial.
2003        SIGMOD Record   Exposing undergraduate students to system internals.
1998    VLDB J. Integrating Reliable Memory in Databases.
1996    VLDB J. Query Processing and Optimization in Oracle Rdb
1996    VLDB J. A Complete Temporal Relational Algebra.
1994        SIGMOD Record   Data Modelling in the Large.
2002        SIGMOD Record   Data Mining: Concepts and Techniques - Book Review.
...
```

Voici à titre d'exemple introductif un programme Pig complet qui calcule le nombre moyen de publications par an dans la revue SIGMOD Record.

```
-- Chargement des documents de journal-small.txt
articles = load 'journal-small.txt'
    as (year: chararray, journal:chararray, title: chararray) ;
sr_articles = filter articles BY journal=='SIGMOD Record';
year_groups = group sr_articles by year;
count_by_year = foreach year_groups generate group, COUNT(sr_articles.title);
dump count_by_year;
```

Quand on l'exécute sur notre fichier-exemple, on obtient le résultat suivant :

```
(1977,1)
(1981,7)
(1982,3)
(1983,1)
(1986,1)
...
```

Un programme Pig est essentiellement une séquence d'opérations, chacune prenant en entrée une collection de documents (les collections sont nommées *bag* dans Pig latin, et les documents sont nommés *tuple*) et produisant en sortie une autre collection. La séquence définit une chaîne de traitements transformant progressivement les documents.

Fig. 9.10 – Un exemple de *workflow* (chaîne de traitements) avec Pig

Il est intéressant de décomposer, étape par étape, cette chaîne de traitement pour inspecter les collections intermédiaires produites par chaque opérateur.

Chargement. L'opérateur `load` crée une collection initiale `articles` par chargement du fichier. On indique le *schéma* de cette collection pour interpréter le contenu de chaque ligne. Les deux commandes suivantes permettent d'inspecter respectivement le schéma d'une collection et un échantillon de son contenu.

```
grunt> describe articles;
articles: {year: chararray,journal: chararray,title: chararray}

grunt> illustrate articles;
-------------------------------------------------------------------------------
| articles | year: chararray | journal: chararray | title: chararray      |
-------------------------------------------------------------------------------
|          | 2003            | SIGMOD Record      | Call for Book Reviews.|
-------------------------------------------------------------------------------
```

Pour l'instant, nous sommes dans un contexte simple où une collection peut être vue comme une table relationnelle. Chaque ligne/document ne contient que des données élémentaires.

Filtrage. L'opération de filtrage avec `filter` opère comme une clause `where` en SQL. On peut exprimer avec Pig des combinaisons Booléennes de critères sur les attributs des documents. Dans notre exemple le critère porte sur le titre du journal.

Regroupement. On regroupe maintenant les tuples/documents par année avec la commande `group by`. À chaque année on associe donc l'ensemble des articles parus cette année-là, sous la forme d'un ensemble imbriqué. Examinons la représentation de Pig :

```
grunt> year_groups = GROUP sr_articles BY year;

grunt> describe year_groups;
  year_groups: {group: chararray,
    sr_articles: {year: chararray,journal: chararray,title:chararray}}

grunt> illustrate year_groups;
 group: 1990
 sr_articles:
  {
   (1990, SIGMOD Record, An SQL-Based Query Language For Networks of Relations.),
   (1990, SIGMOD Record, New Hope on Data Models and Types.)
  }
```

Le schéma de la collection `year_group`, obtenu avec `describe`, comprend donc un attribut nommé `group`

correspondant à la valeur de la clé de regroupement (ici, l'année) et une collection imbriquée nommée d'après la collection-source du regroupement (ici, `sr_articles`) et contenant tous les documents partageant la même valeur pour la clé de regroupement.

L'extrait de la collection obtenu avec `illustrate` montre le cas de l'année 1990.

À la syntaxe près, nous sommes dans le domaine familier des documents semi-structurés. Si on compare avec JSON par exemple, les objets sont notés par des parenthèses et pas par des accolades, et les ensembles par des accolades et pas par des crochets. Une différence plus essentielle avec une approche semi-structurée de type JSON ou XML est que le schéma est *distinct* de la représentation des documents : à partir d'une collection dont le schéma est connu, l'interpréteur de Pig infère le schéma des collections calculées par les opérateurs. Il n'est donc pas nécessaire d'inclure le schéma avec le contenu de chaque document.

Le modèle de données de Pig comprend trois types de valeurs :

— Les *valeurs atomiques* (chaînes de caractères, entiers, etc.).
— Les *collections* (*bags* pour Pig) dont les valeurs peuvent être hétérogènes.
— Les *documents* (*tuples* pour Pig), équivalent des objets en JSON : des ensembles de paires (clé, valeur).

On peut construire des structures arbitrairement complexes par imbrication de ces différents types. Comme dans tout modèle semi-structuré, il existe très peu de contraintes sur le contenu et la structure. Dans une même collection peuvent ainsi cohabiter des documents de structure très différente.

Application de fonctions. Souvenez-vous de la notion *d'opérateur de second ordre* : ils prennent en argument des fonctions à appliquer à une collection. Ces fonctions servent à annoter, restructurer ou enrichir le contenu des documents passant dans le flux. Ici, la collection finale `avg_nb` est obtenue en appliquant une fonction standard `count()`. Dans le cas général, on applique des fonctions applicatives intégrées au contexte d'exécution Pig : *ces fonctions utilisateurs* (User Defined Functions ou UDF) sont appliquées par les opérateurs de second ordre d'un langage comme Pig à de grandes collections, en mode distribué, et avec gestion des pannes.

9.3.2 Les opérateurs

La table ci-dessous donne la liste des principaux opérateurs du langage Pig. Tous s'appliquent à une ou deux collections en entrée et produisent une collection en sortie.

Opérateur	Description
`foreach`	Applique une expression à chaque document de la collection
`filter`	Filtre les documents de la collection
`order`	Ordonne la collection
`distinct`	Elimine lse doublons
`cogroup`	Associe deux groupes partageant une clé
`cross`	Produit cartésien de deux collections
`join`	Jointure de deux collections
`union`	Union de deux collections

Voici quelques exemples pour illustrer les aspects essentiels du langage, basés sur le fichier http://b3d.bdpedia.fr/files/webdam-books.txt. Chaque ligne contient l'année de parution d'un livre, le titre et un auteur.

```
1995        Foundations of Databases Abiteboul
1995        Foundations of Databases Hull
1995        Foundations of Databases Vianu
2012        Web Data Management Abiteboul
2012    Web Data Management Manolescu
2012        Web Data Management Rigaux
2012        Web Data Management Rousset
2012        Web Data Management Senellart
```

Le premier exemple ci-dessous montre une combinaison de `group` et de `foreach` permettant d'obtenir une collection avec un document par livre et un ensemble imbriqué contenant la liste des auteurs.

```
-- Chargement de la collection
books = load 'webdam-books.txt'
    as (year: int, title: chararray, author: chararray) ;
group_auth = group books by title;
authors = foreach group_auth generate group, books.author;
dump authors;
```

L'opérateur `foreach` applique une expression aux attributs de chaque document. Encore une fois, *Pig est conçu pour que ces expressions puissent contenir des fonctions externes*, ou UDF (*User Defined Functions*), ce qui permet d'appliquer n'importe quel type d'extraction ou d'annotation.

L'ensemble résultat est le suivant :

```
(Foundations of Databases,
   {(Abiteboul),(Hull),(Vianu)})
(Web Data Management,
   {(Abiteboul),(Manolescu),(Rigaux),(Rousset),(Senellart)})
```

L'opérateur `flatten` sert à « aplatir » un ensemble imbriqué.

```
-- On prend la collection group_auth et on l'aplatit
flattened = foreach group_auth generate group ,flatten(books.author);
```

On obtient :

```
(Foundations of Databases,Abiteboul)
(Foundations of Databases,Hull)
(Foundations of Databases,Vianu)
(Web Data Management,Abiteboul)
(Web Data Management,Manolescu)
(Web Data Management,Rigaux)
(Web Data Management,Rousset)
(Web Data Management,Senellart)
```

L'opérateur `cogroup` prend deux collections en entrée, crée pour chacune des groupes partageant une même valeur de clé, et associe les groupes des deux collections qui partagent la même clé. C'est un peu compliqué en apparence ; regardons la Fig. 9.11. Nous avons une collection A avec des documents *d* dont la clé de

regroupement vaut a ou b, et une collection B avec des documents *d"*. Le `cogroup` commence par rassembler, séparément dans A et B, les documents partageant la même valeur de clé. Puis, dans une seconde phase, les groupes de documents provenant des deux collections sont assemblés, toujours sur la valeur partagée de la clé.

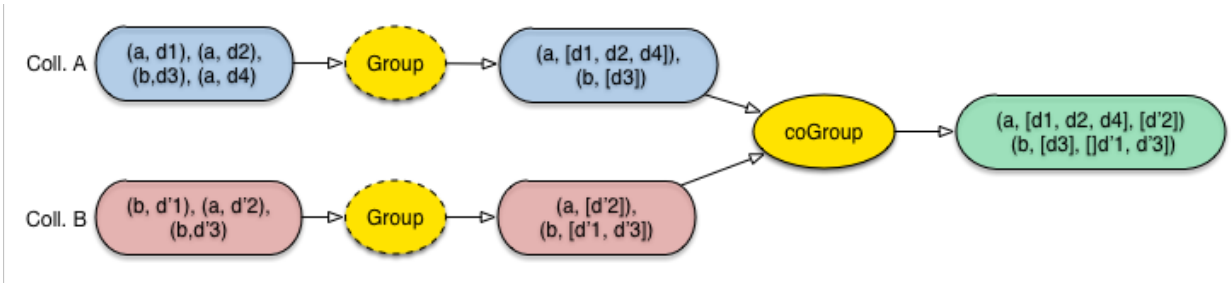


Fig. 9.11 – L'opérateur `cogroup` de Pig.

Prenons une seconde collection, contenant des éditeurs (fichier http://b3d.bdpedia.fr/files/webdam-publishers.txt) :

```
Fundations of Databases     Addison-Wesley  USA
Fundations of Databases     Vuibert France
Web Data Management         Cambridge University Press      USA
```

On peut associer les auteurs et les éditeurs de chaque livre de la manière suivante.

```
--- Chargement de la collection
publishers = load 'webdam-publishers.txt'
  as (title: chararray, publisher: chararray) ;
cogrouped = cogroup flattened by group, publishers by title;
```

Le résultat (restreint au premier livre) est le suivant.

```
(Foundations of Databases,
  { (Foundations of Databases,Abiteboul),
    (Foundations of Databases,Hull),
    (Foundations of Databases,Vianu)
  },
  {(Foundations of Databases,Addison-Wesley),
   (Foundations of Databases,Vuibert)
  }
)
```

Je vous laisse exécuter la commande par vous-même pour prendre connaissance du document complet. Il contient un document pour chaque livre avec trois attributs. Le premier est la valeur de la clé de regroupement (le titre du livre). Le second est l'ensemble des documents de la première collection correspondant à la clé, le troisième l'ensemble des documents de la seconde collection correspondant à la clé.

Il s'agit d'une forme de jointure qui regroupe, en un seul document, tous les documents des deux collections en entrée qui peuvent être appariés. On peut aussi exprimer la jointure ainsi :

```
-- Jointure entre la collection 'flattened' et  'publishers'
joined = join flattened by group, publishers by title;
```

On obtient cependant une structure différente de celle du `cogroup`, tout à fait semblable à celle d'une jointure avec SQL, dans laquelle les informations ont été « aplaties ».

```
(Foundations of Databases,Abiteboul,Fundations of Databases,Addison-Wesley)
(Foundations of Databases,Abiteboul,Fundations of Databases,Vuibert)
(Foundations of Databases,Hull,Fundations of Databases,Addison-Wesley)
(Foundations of Databases,Hull,Fundations of Databases,Vuibert)
(Foundations of Databases,Vianu,Fundations of Databases,Addison-Wesley)
(Foundations of Databases,Vianu,Fundations of Databases,Vuibert)
```

La comparaison entre `cogroup` et `join` montre la flexibilité apportée par un modèle semi-structuré et sa capacité à représenter des ensembles imbriqués. Une jointure relationnelle doit produire des tuples « plats », sans imbrication, alors que le `cogroup` autorise la production d'un état intermédiaire où toutes les données liées sont associées dans un même document, ce qui peut être très utile dans un contexte analytique.

Voici un dernier exemple montrant comment associer à chaque livre le nombre de ses auteurs.

```
books = load 'webdam-books.txt'
    as (year: int, title: chararray, author: chararray) ;
group_auth = group books by title;
authors = foreach group_auth generate group, COUNT(books.author);
dump authors;
```

9.4 Exercices

Commençons par un exercice-type commenté.

Exercice Ex-Exo-type : exercice-type MapReduce

Enoncé.

L'énoncé est le suivant (il provient d'un examen des annales). Un système d'observation spatiale capte des signaux en provenance de planètes situées dans de lointaines galaxies. Ces signaux sont stockés dans une collection *Signaux* de la forme *(idPlanète, date, contenu)*.

Le but est de déterminer si ces signaux peuvent être émis par une intelligence extra-terrestre. Pour cela les scientifiques ont mis au point les fonctions suivantes :

1. *Fonction de structure* : $f_S(c)$: $Bool$, prend un contenu en entrée, et renvoie `true` si le contenu présente une certaine structure, `false` sinon.
2. *Fonction de détecteur d'Aliens* : $f_D(< c >)$: $real$, prend une liste de contenus *structurés* en entrée, et renvoie un indicateur entre 0 et 1 indiquant la probabilité que ces contenus soient écrits en langage extra-terrrestre, et donc la *présence d'Aliens* !

Bien entendu, il y a beaucoup de signaux : c'est du Big Data. Le but est de produire une spécification MapReduce qui produit, pour chaque planète, l'indicateur de présence d'Aliens par analyse des contenus provenant de la planète.

Correction.

D'abord il faut se mettre en tête la forme des documents de la collection. En JSON ça ressemblerait à ça :

```
{"idPlanète" : "Moebius 756",
  "date": "24/02/2067",
  "contenu": "Xioinpoi <ubnnio 3980nklkn"
  }
```

Nous savons que la fonction de Map reçoit un document de ce type, et doit émettre 0, 1 ou plusieurs paires clé-valeur. *La première question à se poser c'est : quels sont les groupes que je dois constituer* et quelle est la clé (« l'étiquette ») qui caractérise ces groupes. Rappelons que MapReduce c'est avant tout un moyen de regrouper des données (ou des quartiers de pomme, ou d'ananas, etc.).

Ici on a un groupe par planète. La clé du groupe est évidemment l'identifiant de la planète. Le squelette de notre fonction de Map est donc :

```
function fMap(doc) {
     emit(doc.idPlanète, qqChose);
}
```

Ici c'est écrit en Javascript mais n'importe quel pseudo-code équivalent (ou du Java, ou du Scala, ou même du PHP…) fait l'affaire.

Quelle est la valeur à émettre ? Ici il faut penser que la fonction de Reduce recevra une liste de ces valeurs et devra produire une valeur agrégée. Dans notre énoncé la valeur agrégée est un indicateur de présence d'Aliens, et cet indicateur est produit sur une liste de contenus structurés.

La fonction de Map doit donc émettre un contenu structuré. Comme tous ne le sont pas, on va appliquer la fonction $f_S(c) : Bool$ (si elle est citée dans l'énoncé, c'est qu'elle sert à quelque chose). Ce qui donne

```
function fMap(doc) {
    if (fS(doc.contenu) == True) {
        emit(doc.idPlanète, doc.contenu)
    }
}
```

Notez bien que la fonction ne peut et ne doit accéder qu'aux informations du document (sauf cas de variables globale qui serait précisée).

Il ne reste plus qu'à écrire la fonction de Reduce. Elle reçoit *toujours* la clé d'un groupe et la liste des valeurs affectées à ce groupe. C'est l'occasion d'utiliser la seconde fonction de l'énoncé :

```
    function fReduce(idPlanète, contenusStruct) {
        return (idPlanète, fD(contenusStruct))
    }
}
```

C'est tout (pour cette fois). L'exemple est assez trivial, mais les mêmes principes s'appliquent toujours.

Exercice Ex-S1-1 : production de jus de fruits : les variantes

Proposons des variantes à notre processus de production de jus de fruit tel qu'il est résumé par la Fig. 9.5. Pour chaque variante envisagée réfléchissez à ses avantages / inconvénients et exposez vos arguments.

- peut-on trier les fruits à la fin de l'atelier de transformation, plutôt qu'au début de l'atelier d'assemblage ?
- et si on triait les fruits *avant* de les soumettre à l'atelier de transformation ?
- dans l'atelier d'assemblage, peut-on avoir un seul pressoir, ou faut-il autant de pressoirs que de types de fruits ?
- peut-on toujours se contenter d'un seul atelier d'assemblage ?
- discuter de la spécialisation des ateliers d'assemblage : MapReduce affecte chaque groupe à un seul atelier ; pourrait-on produire du jus d'orange dans chaque atelier ? Avantages ? Inconvénients ?

Correction

- On peut trier à la fin de la transformation, mais ça ne suffit pas : il faut également fusionner les listes triées provenant d'ateliers de transformation distinct avant l'assemblage. Il est beaucoup plus efficace de fusionner des listes triées que des listes non triées, donc on peut considérer que c'est une option à considérer.
- Rien ne dit que l'atelier de transformation va produire des fruits en sortie, même s'il en a en entrée. Donc ça ne sert à rien de trier (sauf cas particulier).
- On peut avoir plusieurs pressoirs. En fait, en général on ne peut pas prévoir le nombre de groupes, et donc le nombre de pressoirs. Un même tâche effectue donc plusieurs assemblages, en séquence.
- Oui, on peut toujours se contenter, fonctionnellement, d'un seul atelier d'assemblage. Il faut prendre garde à ce que ça ne devienne pas un goulot d'étranglement.
- Dans un contexte de gestion de données, il est impératif d'agréger tous les documents d'un même groupe ensemble. Beaucoup de calcul statistiques ne sont pas additifs (la somme des moyennes et la moyenne des sommes diffèrent) et on ne peut donc pas manipuler des agrégations partielles.

Exercice Ex-S1-2 : commençons à parler informatique

Vous avez un ensemble de documents textuels, et vous voulez connaître la fréquence d'utilisation de chaque mot. Si, par exemple, le mot « confiture » apparaît 1 fois dans le document A, deux fois dans le document B et 1 fois dans le document C, vous voulez obtenir (confiture, 4).

Quel est le processus Map Reduce qui prend en entrée les documents et fournit en sortie les paires (mot, fréquence) ? Décrivez-le avec des petits dessins si vous voulez.

NB : ce genre de calcul est à la base de nombreux algorithmes d'analyse, et sert par exemple à construire des moteurs de recherche.

Correction

Supposons que le format des documents est le suivant

```
{"texte" : "Bla farine bla confiture bla sucre bla confiture ..."}
```

Il faut découper chaque texte en mots, associés à leur nombre d'occurrences dans le texte. Par exemple, (confiture, 2) si le mot apparaît deux fois. L'important est de comprendre que le critère de regroupement (l'étiquette) est le mot. Pour chaque mot on « émet » la paire (mot, nbOccurrences).

```
function fonctionMap (doc)
{
  // On parcourt les mots du texte
  for (mot in doc.texte) {
     emit (mot, 1)
  }
}
```

À noter : on émet plusieurs paires (clé, valeur) par document, autant qu'il y a de mots dans le document. Pour un même mot *m*, on émettra donc autant de fois *(m, 1)* qu'il y a d'occurrences de *m*.

Noter aussi que dans une spécification de haut niveau, on n'a pas besoin d'indiquer comment on découpe un texte en mot. Ca se fera au moment de l'implantation.

La phase d'agrégation se charge de cumuler les occurrences d'un même mot sur l'ensemble des documents.

```
function fonctionReduce (mot, occurrences)
{
   return (mot, count (occurrences))
}
```

Exercice Ex-S1-3 : continuons l'examen du 16 juin 2016

Reprenons les documents représentant les inscriptions des étudiants à des UEs. Voici deux exemples.

```
{
  "_id": 978,
  "nom": "Jean Dujardin",
  "UE": [{"id": "ue:11", "titre": "Java", "note": 12},
        {"id": "ue:27", "titre": "Bases de données", "note": 17},
        {"id": "ue:37",  "titre": "Réseaux", "note": 14}
        ]
}

{
  "_id": 476,
   "nom": "Vanessa Paradis",
   "UE": [{"id":  "ue:13",  "titre": "Méthodologie", "note": 17,
          {"id": "ue:27",  "titre": "Bases de données", "note": 10},
```

(suite sur la page suivante)

(suite de la page précédente)

```
          {"id":  "ue:76",   "titre": "Conduite projet", "note": 11}
        ]
}
```

Spécifiez le calcul du nombre d'étudiants par UE, en MapReduce, en prenant en entrée des documents construits sur le modèle ci-dessus.

Correction

Une fonction de Map produit des paires (clé, valeur). La première question à se poser c'est : quelle est la clé que je choisis de produire ? Rappelons que la clé est une sorte d'étiquette que l'on pose sur chaque valeur et qui va permettre de les regrouper.

Ici, on veut regrouper par UE pour pouvoir compter tous les étudiants inscrits. On va donc émettre une paire intermédiaire pour chaque UE mentionnée dans un document en entrée. Voici le pseudo-code.

```
function fonctionMap ($doc) # doc est un document centré␣
↪étudiant
{
  # On parcourt les UEs du tableau UEs
  for    [$ue in $doc.UE] do
     emit ($ue.id, $doc.nom)
  done
 }
```

Quand on traite le premier document de notre exemple, on obtient donc trois paires intermédiaires :

```
{"ue:11", "Jean Dujardin"}
{"ue:27", "Jean Dujardin"}
{"ue:37", "Jean Dujardin"}
```

Et quand on traite le second document, on obtient :

```
{"ue:13", "Vanessa Paradis"}
{"ue:27", "Vanessa Paradis"}
{"ue:76", "Vanessa Paradis"}
```

Toutes ces paires sont alors transmises à « l'atelier d'assemblage » qui les regroupe sur la clé. Voici la liste des groupes (un par UE).

```
{"ue:11", ["Jean Dujardin"]}
{"ue:13", ["Vanessa Paradis"]}
{"ue:27", ["Jean Dujardin", "Vanessa Paradis"]}
{"ue:37", ["Jean Dujardin"]}
{"ue:76", ["Vanessa Paradis"]}
```

Il reste à appliquer la fonction de Reduce à chaque groupe.

```
function fonctionReduce ($clé, $tableau)
{
  return ($clé, count($tableau)
}
```

Et voilà.

Exercice Ex-S1-4 : algèbre linéaire distribuée

Nous disposons d'une matrice *M* de dimension $N \times N$ représentant les liens entres les N pages du Web, chaque lien étant qualifié par un facteur d'importance (ou « poids »). La matrice est représentée par une collection C dans laquelle chaque document est de la forme {« id » : &23, « lig » : *i*, « col » : *j*, « poids » : m_{ij}}, et représente un lien entre la page P_i et la page P_j de poids m_{ij}

Exemple : voici une matrice M avec $N = 4$. La première cellule de la seconde ligne est donc représentée par un document {« id » : &t5x, « lig » : 2, « col » : 1, « poids » : 7}

$$M = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 7 & 6 & 5 & 4 \\ 6 & 7 & 8 & 9 \\ 3 & 3 & 3 & 3 \end{bmatrix}$$

Questions

— Chaque ligne L_i de M peut être vue comme un vecteur décrivant la page P_i. Spécifiez le traitement MapReduce qui calcule la norme de ces vecteurs à partir des documents de la collection C (rappel : la norme d'un vecteur $V(x_1, x_2, \cdots, x_n)$ est $\sqrt{x_1^2 + x_2^2 + \cdots + x_n^2}$).

— Nous voulons calculer le produit de la matrice avec un vecteur $V(v_1, v_2, \cdots v_N)$ de dimension N. Le résultat est un autre vecteur W tel que :

$$w_i = \Sigma_{j=1}^{N} m_{ij} \times v_j$$

On suppose pour le moment que V tient en mémoire RAM et est accessible comme variable statique par toutes les fonctions de Map ou de Reduce. Spécifiez le traitement MapReduce qui implante ce calcul.

Correction

On connaît le format des documents. Le critère de regroupement est l'identifiant de la ligne. La valeur à émettre est le carré du poids

```
function fonctionMap (doc)
{
  emit (doc.lig, doc.poids * doc.poids)
}
```

La fonction de Reduce recevra donc les carrés des poids d'une même ligne. Il restera à les cumuler pour obtenir la norme.

```
function fonctionReduce (lig, listePoids)
{
   return (lig, sqrt (sum (listePoids)))
}
```

Le produit de la matrice et du vecteur est presque identique. La fonction de Map multiplie chaque poids d'une ligne de la matrice par la coordonnées correspondante du vecteur V (supposé accessible comme variable globale).

```
function fonctionMap (doc)
{
  emit (doc.lig, doc.poids * V[doc.col])
}
```

La fonction de Reduce est la même qu'auparavant.

Exercice Ex-S1-5 : un peu plus difficile

On considère des documents représentant des articles ou ouvrages de recherche, avec la liste de leurs auteurs. Voici le format d'après un exemple.

```
{
  "type": "Book",
  "title": "Bases de données distribuées",
  "year": 2020,
  "publisher": "Cnam",
  "authors": ["R. Fournier-S'niehotta", "P. Rigaux", "N. Travers"]
}
```

On veut calculer, pour chaque trio d'auteurs (x, y, z), le nombre d'articles que ces trois auteurs ont co-signés. Précisions : si les auteurs d'un article sont un sur-ensemble de (x, y, z) (par ex. (x, u, v, y, w, z)) ça compte pour 1. L'ordre des auteurs ne compte pas : (z, y, x) est considéré comme une occurrence.

Comment faire ce calcul en MapReduce ?

NB : le résultat est appelé *support* de (x,y,z), et c'est une mesure utilisée, entre autres, dans la découverte de règles d'association.

Correction

Pour chaque publication, on prend la liste des auteurs, et on en extrait toutes les combinaisons possibles de 3 auteurs distincts. S'il y a 5 auteurs par exemple, il y a $(5 \times 4 \times 3)/(3 \times 2)$ combinaisons possibles (révisez votre combinatoire si nécessaire).

Pour chaque combinaison C, il faut émettre la paire $(C, 1)$. Le regroupement se fera sur la combinaison, et la somme des 1 donnera le nombre d'occurrences sur l'ensemble de la collection. Encore une fois, c'est le choix de la clé qui est essentiel.

Exercice Ex-S1-6 : classons les fruits

On reçoit d'un fournisseur une livraison de fruits (pommes, ananas, oranges, etc). On décide d'effectuer un test qualité en leur donnant 0 ou 1 pour les critères suivants (trop mûr, tâché, déformé, etc).

Pour chaque espèce de fruit, on veut calculer la proportion de fruits trop mûrs. Comment faire avec MapReduce ?

On veut faire le même calcul pour tous les critères. Comment faire ?

NB : ce genre d'indicateur permet de construire des classeurs pour reconnaitre automatiquement l'espèce d'un fruit (enfin, pour produire un indicateur de probabilité d'appartenance à une espèce à une classe, avec choix de la classe la plus probable).

Correction

Pour chaque fruit, on émet (espèce, 1) ou (espèce, 0) selon que le fruit est trop mûr ou pas. Ensuite, la phase d'agrégation regroupe les paires pour la même espèce, et on calcule la fraction du nombre de 1 sur le nombre total du groupe.

Si on a trois critères, on émet (espèce, [i, j, k]) en généralisant le principe.

CHAPITRE 10

Pig : Travaux pratiques

10.1 Première partie : analyse de flux multiples

Cet exercice vise à découvrir Pig de manière un peu plus approfondie. Nous nos plaçons dans la situation d'un système recevant deux flux de données distincts et effectuant des opérations de rapprochement, sélection et agrégation, dans l'optique de la préparation d'une étude statistique ou analytique.

Le jeu de données proposé est notre base de films, mais il est assez facile de transposer ce qui suit à d'autres applications. Vous trouverez sur le site http://deptfod.cnam.fr/bd/tp/datasets/ deux fichiers au format d'import Pig : la liste des films et la liste des artistes. Le format est JSON, avec la particularité qu'un fichier contient une liste d'objets, et que chaque objet est stocké sur une seule ligne.

Voici une ligne du fichier pour les films.

```
{ "_id": "movie:1", "title": "Vertigo", "year": 1958, "genre":
"drama", "summary": "...", "country": "USA", "director": { "_id": "artist:3"},
 "actors": [{ "_id": "artist:15", "role": "John
             Ferguson" }, { "_id": "artist:16", "role": "Madeleine Elster"
     }]
 }
```

Et une ligne du fichier pour les artistes.

```
{ "_id": "artist:15", "last_name": "Stewart", "first_name":
"James", "birth_date": "1908" }
```

Comme vous le voyez, le metteur en scène est complétement intégré au document des films, alors que les acteurs ne sont que référencés par leur identifiant.

Nommez les deux fichiers respectivement `artists-pig.json` et `movies-pig.json`. Voici les commandes de chargement dans l'interpréteur Pig.

```
artists = LOAD 'artists-pig.json'
      USING JsonLoader('id: chararray, firstName:chararray,
         lastName:chararray, birth:chararray');

movies = LOAD 'movies-pig.json'
  USING JsonLoader('id:chararray, title:chararray, year:chararray,
                    genre:chararray, summary:chararray, country:chararray,
                    director: (id:chararray,lastName:chararray,
                           firstName:chararray,birthDate:chararray),
                    actors: {(id:chararray, role:chararray)}'
                        );
```

Comme vous le voyez on indique le schéma des données contenues dans le fichier pour que Pig puisse créer sa collection.

Allons-y pour des programmes Pig traitant ces données.

1. Créez une collection `mUSA_annee` groupant les films américains par année (code du pays : US). Vous devriez obtenir le format suivant.

   ```
   (2003, {(The Matrix reloaded),(Lost in Translation),
             (Kill Bill),(The Matrix Revolutions)})
   ```

   Correction

   ```
   moviesUSA = filter movies BY country=='US';
   mUSA_group = group moviesUSA by year;
   mUSA_annee = foreach mUSA_group generate group, moviesUSA.title;
   ```

2. Créez une collection `mUSA_director` groupant les films américains par metteur en scène. Vous devriez obtenir des documents du type suivant :

   ```
   ((artist:181,Coppola,Francis Ford,1940),
       {(Le parrain III),(Le parrain II),(Le parrain)})
   ```

   Correction

   ```
   mUSA_group2 = group moviesUSA by director;
   mUSA_director = foreach mUSA_group2 generate group, moviesUSA.title;
   ```

3. Créez une collection `mUSA_acteurs` contenant des triplets (idFilm, idActeur, role). Chaque film apparaît donc dans autant de documents qu'il y a d'acteurs. Vous devriez obtenir par exemple :

```
(movie:54,artist:137,Sonny Corleone)
(movie:54,artist:176,Michael Corleone)
(movie:54,artist:182,Don Vito Corleone)
```

Aide : il faut « aplatir » la collection `actors` imbriquée dans chaque film.

Correction

```
mUSA_actors = foreach movies generate id, flatten(actors);
```

4. Maintenant, créez une collection `moviesActors` associant l'identifiant du film à la description complète de l'acteur. Ce qui devrait donner par exemple :

```
(movie:54,artist:176,Michael Corleone,artist:176,Pacino,Al,1940)
```

Aide : c'est une jointure bien sûr. Consultez le schéma de `mUSA_actors` pour connaître le nom des colonnes.

Correction

```
moviesActors = join mUSA_actors by actors::id, artists by id;
```

5. Et pour finir, créez une collection `fullMovies` associant la description complète du film à la description complète de tous les acteurs.
   Aide : soit une jointure entre `moviesActors` et `movies`, puis un regroupement par film, ce qui un contenu correct mais très compliqué (essayez), soit (mieux) un `cogroup` entre `moviesUSA` et `moviesActors`. Voici un exemple du résultat dans le second cas.

```
(movie:33,{(movie:33,Psychose,1960,Thriller,USA,(artist:3,Hitchcock,Alfred,
↪1899))},
          {(movie:33,artist:90,Marion Crane,artist:90,Leigh,Janet,1927),
           (movie:33,artist:89,Lila Crane,artist:89,Miles,Vera,1929),
           (movie:33,artist:88,Bates,artist:88,Perkins,Anthony,1932)})
```

Correction

Avec `join` :

```
joinMovies = join movies by id, moviesActors by mUSA_actors::id;
fullMovies = group joinMovies by movies::id;
```

Avec `cogroup` :

```
fullMovies2 = cogroup moviesUSA by id, moviesActors by mUSA_actors::id;
```

6. Créer une collection `ActeursRealisateurs` donnant pour chaque artiste la liste des films où il/elle a joué (éventuellement vide), et des films qu'il/elle a dirigé. On peut se contenter d'afficher l'identifiant de l'artiste, ce qui donnerait :

```
(artist:24,{},{(movie:10,Blade Runner,artist:24,Deckard),
               (movie:34,Le retour du Jedi,artist:24,Han Solo)})
```

Ou effectuer une jointure supplémentaire pour obtenir le nom et le prénom.

```
(artist:24,Ford,Harrison,1942,artist:24,{},
      {(movie:10,Blade Runner,artist:24,Deckard),
       (movie:34,Le retour du Jedi,artist:24,Han Solo)})
```

Correction

```
acteurs = foreach movies generate id, title, flatten(actors);
IdActeursRealisateurs = cogroup movies by director.id, acteurs by␣
↪actors::id;
ActeursRealisateurs = join artists by id, IdActeursRealisateurs by group;
```

10.2 Deuxième partie : analyse de requêtes

Note : Cet exercice est basé sur le tutoriel fourni avec Pig, voir https://cwiki.apache.org/confluence/display/pig/PigTutorial

L'objectif est d'aller significativement plus loin avec Pig, en procédant à de l'analyse de données. On va utiliser un fichier de logs du moteur de recherche Excite. C'était un portail de recherche très utilisé avant l'an 2000 et l'arrivée de Google. Nos données sont donc les requêtes qui ont été soumises à ce moteur pendant une journée, en 1997.

Normalement, vous trouverez deux versions du fichier de log, dans votre dossier `pigdir/tutorial/data`. Vous pouvez commencer par regarder la version courte, `excite-small.log`, par exemple avec `less`. Elle comporte 4501 lignes, la version longue approche le million. Chaque ligne est au format : `user time query` : un identifiant de l'utilisateur, un horodatage du moment où la requête a été reçue par le serveur, et les mots-clefs de celle-ci. La version longue est dans un fichier zippé, elle peut s'extraire avec la commande suivante :

```
bzip2 -d excite.log.bz2
```

10.2.1 Aparté sur l'utilisation de Pig

L'interprêteur de Pig peut s'utiliser en local ou avec Hadoop comme on l'a vu en cours. Dans chacun de ces deux cas, vous pouvez utiliser un mode interactif ou un mode avec script. Dans le cadre de ce TP, je vous conseille de coupler les deux : dans une fenêtre de terminal, vous utilisez le mode interactif pour écrire des commandes, les tester, les corriger, etc. À côté, conservez celles qui fonctionnent dans un fichier de script (par convention donnez lui l'extension `.pig`), de façon à pouvoir facilement en copier-coller certaines en cas de redémarrage de la session avec le mode interactif.

D'autre part, en mode interactif, vous pouvez écrire dans des fichiers les sorties de vos commandes, mais il sera souvent plus utile d'employer `dump` après avoir exécuté une commande et stocké le résultat. Attention, avec de grosses collections ce dump peut prendre longtemps (`dump raw` ci-dessous afficherait 1 million de lignes…).

10.2.2 Première analyse

Dans cette partie, nous allons chercher à obtenir quelles sont les requêtes les plus fréquentes à certaines heures de la journée.

On va cette fois utiliser des fonctions Java écrites pour l'occasion, fournies dans quelques fichiers du dossier `pigdir/tutorial/src/org/apache/pig/tutorial`.

Afin de pouvoir les utiliser, il vous faut le fichier tutorial.jar.

Il est probable qu'il vous faille exécuter la commande suivante pour que Pig puisse démarrer :

```
export JAVA_HOME="/usr"
```

Vous pouvez ensuite démarrer votre session Pig avec la commande suivante (si vous êtes dans `pigdir`) :

```
./bin/pig -x local
```

Nous devons commencer par une instruction permettant à Pig de savoir qu'on va utiliser des fonctions Java définies extérieurement :

```
REGISTER ./tutorial.jar;
```

Ensuite, on charge le fichier à utiliser, dans un « bag ». Vous pouvez utiliser `excite` (1 million de lignes), ou `excite-small` (4500).

```
raw = LOAD './tutorial/data/excite.log'
     USING PigStorage('\t') AS (user, time, query);
```

Nettoyage des données

On procède ensuite, en deux étapes, au nettoyage de nos données, étape toujours importante dans l'analyse de données réelles. L'anonymisation (retrait de données personnelles), la normalisation des encodages de caractères, le retrait des requêtes vides sont souvent nécessaires. Ici, on se contente d'enlever les requêtes vides et celles qui contiennent des URL.

```
clean1 = FILTER raw BY org.apache.pig.tutorial.NonURLDetector(query);
```

Regardez le code Java présent dans le fichier `NonURLDetector.java` du dossier `tutorial/src/org/apache/pig/tutorial/` :

```
public class NonURLDetector extends FilterFunc {

 private Pattern _urlPattern = Pattern.compile("^[\"]?(http[:|;])|(https[:|;
↪])|(www\\.)");

 public Boolean exec(Tuple arg0) throws IOException {
  if (arg0 == null || arg0.size() == 0)
    return false;

   String query;
   try{
    query = (String)arg0.get(0);
    if(query == null)
      return false;
    query = query.trim();
   } catch(Exception e){
     System.err.println("NonURLDetector: failed to process input; error - " + e.
↪getMessage());
     return false;
   }

   if (query.equals("")) {
    return false;
   }
   Matcher m = _urlPattern.matcher(query);
   if (m.find()) {
    return false;
   }
   return true;
 }
}
```

Cette classe reçoit des chaînes de caractères, et renvoie `false` si :

- la requête est vide (`arg0.size() == 0`, `arg0 == null`, `query.equals("")`)
- la requête trouve le motif d'expression régulière qui correspond à une URL : `^[\"]?(http[:|;])|(https[:|;])|(www\\.)`

Vous pouvez voir quelles sont les chaînes ignorées en utilisant l'opérateur `NOT` dans la commande précédente :

```
clean0 = LIMIT (FILTER raw BY NOT org.apache.pig.tutorial.NonURLDetector(query))␣
↪100;
dump clean0;
```

Attention, le *dump* va ici écrire beaucoup de lignes si l'on ne le limite pas, d'où la syntaxe ci-dessus qui permet d'avoir un aperçu, avec seulement 100 lignes. Cependant, ce n'est pas exhaustif, et ne permet pas vraiment de voir si l'on exclut bien toutes les requêtes que l'on souhaite.

Reprenons et poursuivons le nettoyage des données, en passant toutes les chaînes de caractères en bas de casse (minuscules).

```
clean2 = FOREACH clean1 GENERATE user, time, org.apache.pig.tutorial.
↪ToLower(query) as query;
```

Extraction de l'heure

Ensuite, on s'intéresse à l'extraction de l'heure. En effet, nos données ne concernent qu'une seule journée, on a donc seulement besoin de cette information (je vous rappelle que l'on cherche à distinguer les mots et groupes de mots fréquents à certaines heures de la journée).

On va passer à nouveau par du code Java, cette fois dans le fichier `ExtractHour.java` du dossier `tutorial/src/org/apache/pig/tutorial/` :

```
houred = FOREACH clean2 GENERATE user, org.apache.pig.tutorial.ExtractHour(time)␣
↪as hour, query;
```

La seule partie importante du code java est la suivante :

```
return timestamp.substring(6, 8);
```

L'horodatage utilisé dans notre fichier est de la forme `AAMMJJHHMMSS` : `970916161309` correspond au 16 septembre 1997, à 16h13m09s. Ainsi, le code retourne la sous-chaîne de cet horodatage entre les positions 6 et 8 (6 inclus, 8 exclu), c'est-à-dire le `HH` recherché.

Récupération des nGrams

Maintenant, on va chercher les n-grams, c'est-à-dire les séquences de n mots contenus dans nos requêtes. On va se restreindre à n=2. Pour la requête `une bien jolie requête`, on va extraire les mots et groupes de mots suivants :

- une
- bien
- jolie
- requête
- une bien
- bien jolie
- jolie requête

Pour cela, c'est le code Java de `NGramGenerator.java` qui se charge du traitement.

```
ngramed1 = FOREACH houred GENERATE user, hour,
     flatten(org.apache.pig.tutorial.NGramGenerator(query)) as ngram;
```

Je vous conseille ici de regarder des `dump` sur des ngrams particuliers, pour bien comprendre ce que font chacune des commandes qui suivent. Exemple, avec `demi` :

```
ng = filter ngramed1 by ngram=='demi';
dump ng;
```

Vous obtenez par exemple :

```
(BD64F5DBA403D401,18,demi)
(F18FA4825A88A1E1,10,demi)
(1DE4083F198B3F0E,22,demi)
(1DE4083F198B3F0E,22,demi)
```

On peut ensuite enlever les n-grams utilisés plusieurs fois par le même utilisateur dans la même heure (ci-dessus, `1DE4083F198B3F0E` a utilisé `demi` deux fois entre 22h et 22h59) :

```
ngramed2 = DISTINCT ngramed1;
```

Agrégation par heure

Groupons ensuite pour avoir une collection par n-gram et par heure :

```
hour_frequency1 = GROUP ngramed2 BY (ngram, hour);
```

Comptons maintenant le nombre d'occurences de chaque n-gram dans chaque heure :

```
hour_frequency2 = FOREACH hour_frequency1 GENERATE flatten($0), COUNT($1) as␣
↪count;
```

Correction

hf = filter hour_frequency2 by ngram==”demi”; dump hf;

On regroupe par n-gram :

```
uniq_frequency1 = GROUP hour_frequency2 BY group::ngram;
```

Correction

uf = filter uniq_frequency1 by group==”demi”; dump uf;

Heures anormales pour un mot

On utilise le code Java de `ScoreGenerator.java` qui calcule, pour chaque n-gram, la moyenne (et l'écart-type) des utilisations par heure, et un score pour chaque heure (un score de `1.0` signifie que ce n-gram, dans cette heure-là, a été utilisé de sa moyenne + 1.0 * l'écart type). La seconde commande assigne des noms à nos champs :

```
uniq_frequency2 = FOREACH uniq_frequency1
      GENERATE flatten($0), flatten(org.apache.pig.tutorial.ScoreGenerator($1));
uniq_frequency3 = FOREACH uniq_frequency2
     GENERATE $1 as hour, $0 as ngram, $2 as score, $3 as count, $4 as mean;
```

On ne garde que les scores supérieurs à `2.0` :

```
filtered_uniq_frequency = FILTER uniq_frequency3 BY score > 2.0;
```

Enfin, on écrit dans un fichier :

```
STORE filtered_uniq_frequency INTO './res1' USING PigStorage();
```

Le résultat de l'analyse se trouve dans le fichier `./res1/part-r-00000`.

10.2.3 Aller plus loin

Tri

On peut trier ce fichier par heure et par n-gram avec la commande suivante (de façon à voir un peu mieux quelles requêtes apparaissent à quelle heure) :

```
sort -k2,1 -T. -S1G res1/part-r-00000 | column -ts $'\t' | less
```

Ce tri peut aussi être fait en Pig, avant d'écrire.

Visualisation

On peut aussi s'intéresser à des résultats intermédiaires et faire un peu de visualisation. Regardons simplement le nombre d'utilisateur différents qui ont tapé le mot `demi` à chaque heure de la journée :

```
hf = filter hour_frequency2 by ngram=='demi';
STORE hf INTO './demi' USING PigStorage();
gnuplot
plot "demi/part-r-00000" using 2:3 with lines"
```

Bien sûr, `gnuplot` est un outil parmi d'autres, vous pouvez utiliser `matplotlib` en python par exemple.

Deuxième analyse

À titre d'exercice, vous pouvez aussi reprendre les commandes ci-dessus et essayer de comparer, par exemple, les utilisations de n-grams à 00h et à 12h (indices : vous n'aurez pas besoin de ScoreGenerator et il faudra utiliser un join).

Correction

La solution se trouve dans le fichier `pigdir/tutorial/scripts/script1-local.pig`.

CHAPITRE 11

Le *cloud*, une nouvelle machine de calcul

Jusqu'à présent nous avons considéré le cas de la gestion de documents (ou plus généralement d'objets semi-structurés sérialisés) dans le contexte classique d'une unique machine tenant le rôle de serveur de données, et communiquant avec des applications clientes. Le serveur dispose d'un CPU, de mémoire RAM, d'un ou plusieurs disques pour la persistance. C'est une architecture classique, courante, facile à comprendre. Elle permet de se pencher sur des aspects importants comme la modélisation des données, leur indexation, la recherche.

La problématique. Nous envisageons maintenant la problématique de la *scalabilité* et les méthodes pour l'aborder et la résoudre. Pour parler en termes intuitifs (pour l'instant) la scalabilité est la capacité d'un système à s'adapter à une croissance non bornée de la taille des données (nous parlons de données, et de traitements sur les données, c'est restrictif et voulu car c'est le sujet du cours). Cette croissance, si nous ne lui envisageons pas de limite, finit *toujours* par dépasser les capacités d'une seule machine. Si c'est en mémoire RAM, cette capacité se mesure au mieux en TeraOctets (TOs) ; si c'est sur le disque, en dizaines de TéraOctets. Même si on améliore les composants physiques, toujours viendra le moment où le serveur individuel sera saturé.

Un moyen de gérer la scalabilité est d'ajouter des machines au système, et d'en faire donc un *système distribué*. Cela ne va pas sans redoutables complications car il faut s'assurer de la bonne coopération des machines pour assumer une tâche commune. Cette méthode est aussi celle qui est privilégiée aujourd'hui pour faire face au déluge de production des données numériques (et de leurs utilisateurs). Nous en proposons dans ce qui suit une présentation qui se veut suffisamment exhaustive pour couvrir les techniques les plus couramment utilisées, en les illustrant par quelques systèmes représentatifs.

Une nouvelle perspective. Un mot, avant de rentrer dans le gras, sur le titre du chapitre qui fait référence au *cloud*, considéré comme un outil d'allocation de machines à la demande. Reconnaissons tout de suite que c'est un peu réducteur car le *cloud* a d'autres aspects, et on peut y recourir pour disposer d'une seule machine sans envisager la dimension de la scalabilité. Il serait plus correct de parler de *grappe de serveurs* mais c'est plus long.

Donc assumons le terme, dans une perspective générale qui définit *le cloud comme une nouvelle machine*

de calcul à part entière, élastique et scalable, apte à prendre en charge des masses de données sans cesse croissantes. La première session va introduire cette perspective d'ensemble, et nous commencerons ensuite une investigation systématique de ses possibilités.

Note : Rappelons qu'un MégaOctet (MO), c'est 10^6 octets ; un GigaOctet (GO) c'est 10^9 octets soit 1 000 MOs ; un TéraOctet (TO) c'est 10^{12} octets (1 000 GOs) ; un PétaOctet (PO) c'est 10^{15} octets, 1 000 TOs. Le PO, c'est l'unité du volume de données gérés par les applications à l'échelle du Web. Pas besoin d'aller au-delà pour l'instant !

11.1 S1 : *cloud* et données massives

Supports complémentaires :

- Présentation: Cloud et données massives
- Vidéo de la session Cloud

Ce chapitre envisage le *cloud* comme une nouvelle machine de calcul en tant que telle, qui se distingue de celles que nous utilisons tous les jours par le fait qu'elle est constituée de composants autonomes (les serveurs) communiquant par réseau. Sur cette machine globale se déploient des logiciels dont la caractéristique commune est de tenter d'utiliser au mieux les ressources de calcul et de stockage disponible, de s'adapter à leur évolution (ajout/retrait de machines, ou pannes) et accessoirement de faciliter la tâche des utilisateurs, développeurs et administrateurs du système.

11.1.1 Vision générale

La Fig. 11.1 résume la vision sur laquelle nous allons nous appuyer pour l'étude des systèmes distribués.

Note : Je rappelle une dernière fois pour ne plus avoir à le préciser ensuite : la perspective adoptée ici est celle de la gestion de *données* massives. Les grilles de calcul par exemple n'entrent pas dans ce contexte. Autre rappel : ces données sont des unités d'information autonomes et identifiables que nous appelons *documents* pour faire simple.

La figure montre une organisation en couches allant du matériel à l'applicatif. Ces couches reprennent l'architecture classique d'un système orienté données : stockage et calcul au niveau bas, système de gestion de données au-dessus de la couche matérielle, interfaces d'accès au données ou de traitement analytique, et enfin applications s'appuyant sur ces interfaces pour la partie de leurs tâches relative aux accès et à la manipulation des données.

Qui dit architecture en couche dit *abstraction*. Idéalement, chaque couche prend en charge des problèmes techniques pour épargner à la couche supérieure d'avoir à s'en soucier. Un SGBD relationnel par exemple gère l'accès aux disques, la protection des données, la reprise sur panne, la concurrence d'accès, en isolant les applications de ces préoccupations qui leur compliqueraient considérablement la tâche. Les deux couches intermédiaires de notre architecture tiennent un rôle similaire.

Pour bien comprendre en quoi ce rôle a des aspects particuliers dans le cadre d'un système distribué à grande échelle, commençons par la couche matérielle qui va principalement nous occuper dans ce chapitre.

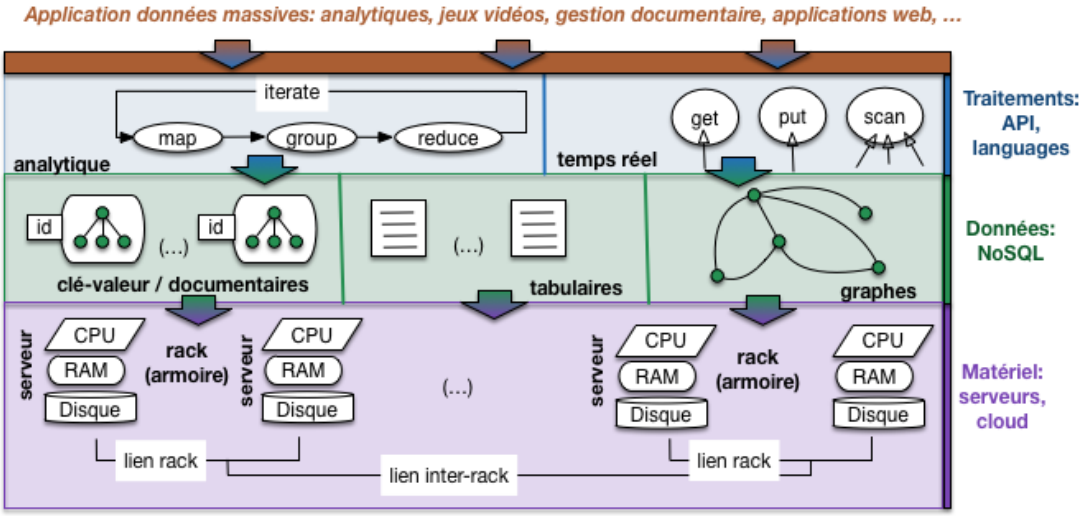


Fig. 11.1 – Perspective générale sur les systèmes distribués dans un *cloud*

La couche matérielle

Notre figure montre une couche matérielle constituée de serveurs reliés par un réseau. Chaque serveur dispose d'un processeur (souvent multi cœurs), d'une mémoire centrale à accès direct (RAM) et d'un ou plusieurs supports de stockage persistant, les disques.

Dans une grappe de serveur, les machines sont dotées de composants de base (soit essentiellement la carte mère, les disques et la connectique) et dépourvues de tous les petits accessoires des ordinateurs de bureau (clavier, souris, lecteur DVD, etc.). De plus, les composants utilisés sont souvent de qualité moyenne voire médiocre afin de limiter les coûts. On parle de *commodity server* dans le jargon du milieu. C'est un choix qui peut se résumer ainsi : *on préfère avoir beaucoup de serveurs de qualité faible que quelques serveurs de très bonne qualité*. Il a des conséquences très importantes, et notamment :

— le faible coût des serveurs permet d'en ajouter facilement à la demande, et d'obtenir la scalabilité souhaitée ;
— d'un autre côté la qualité médiocre des composants implique un taux de panne relativement élevé ; *c'est un facteur essentiel qui impacte toutes les couches du système distribué*.

Important : Dans les environnements de *cloud* proposés sous forme de service, les serveurs sont le plus souvent créés par *virtualisation* ce qui apporte plus de flexibilité pour la gestion du service mais impacte (modérément) les performances. Comme on ne peut pas parler de tout, on va ignorer cette option ici. Vous êtes maintenant familiarisés avec l'utilisation de Docker : imaginez ce que cela donne, en terme d'administration, si vous ne disposez pas d'une machine mais de quelques centaines ou milliers.

Les serveurs sont empilés dans des baies spéciales (*rack*) équipées pour leur fournir l'alimentation, la connexion réseau vers la grappe de serveur, la ventilation. La Fig. 11.2 montre un serveur et une baie typiques.

Enfin, les baies sont alignées les unes à côté des autres dans de grands hangars (les fameux *data centers*, ou grappes de serveurs), illustrés par la Fig. 11.3.

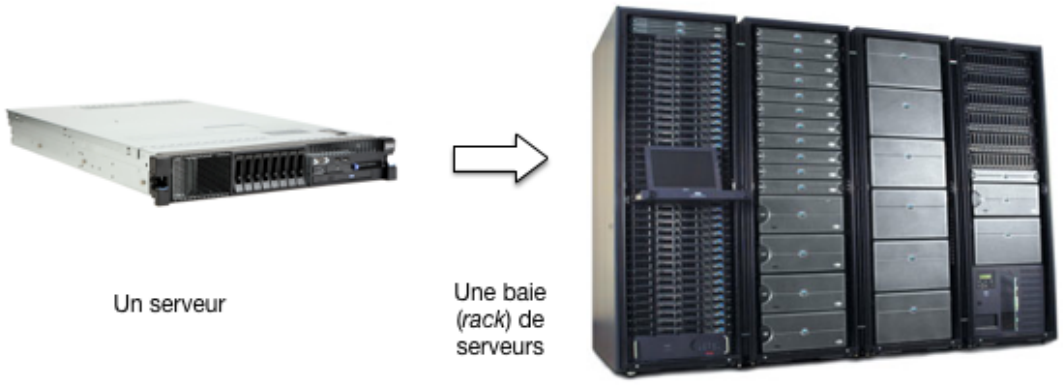


Fig. 11.2 – Un serveur et une baie de serveurs.

Fig. 11.3 – Une grappe de serveurs.

Les baies sont connectées les unes aux autres grâce à des routeurs, et la grappe de serveur est elle-même connectée à l'Internet par un troisième niveau de connexion (après ceux intra-baie, et inter-baies). On obtient une connectique dite « hiérarchique » qui joue un rôle dans la gestion des données massives.

Voici quelques ordres de grandeur pour se faire une idée de l'infrastructure matérielle d'un système à grande échelle :

— une baie contient quelques dizaines de serveurs, typiquement une quarantaine ;
— les grappes de serveurs peuvent atteindre des centaines de baies : 100 baies = env. 4000 serveurs = des PétaOctets de stockage.
— les grandes sociétés comme Google, Facebook, Amazon ont depuis longtemps dépassé le cap du million de serveurs : essayez d'en savoir plus si cela vous intéresse (pas si facile).

Pas besoin d'atteindre cette échelle pour commencer à faire du distribué : quelques machines (2 au minimum…) et on a déjà mis le pied dans la porte. Intervient alors la notion d'*élasticité* étroitement liée au *cloud* : étant donnée l'abondance de ressources, il est très facile d'allouer une ou plusieurs nouvelles machines à notre système. Si ce dernier est conçu de manière à être *scalable* (définition plus loin) il n'y a virtuellement pas de limite à l'accroissement de sa capacité matérielle.

Il faut souligner que dans l'infrastructure que nous présentons, la dépendance entre les serveurs est réduite au minimum. Ils ne partagent pas de mémoire ou de périphériques, et leur seul moyen de communiquer est l'échange de messages (*message passing*). Cela implique que si un serveur tombe en panne, l'impact reste local.

Gare à la panne

Une des conséquences importantes d'une infrastructure basée sur des serveurs à faible coût est la fréquence des *pannes* affectant le système. Ces pannes peuvent être matérielles, logicielles, ou liées au réseau. Elles sont temporaires (un composant ne répond plus pendant une période plus ou moins courte, puis réapparaît - typique des problèmes réseau) ou permanentes (un disque qui devient inutilisable).

La fréquence d'une panne est directement liée au nombre de composants. Si, pour prendre un exemple classique, un disque tombe en panne en moyenne tous les 10 ans, on aura affaire (en moyenne) à une panne par an avec 10 disques, et une panne par jour à partir de quelques milliers de disques ! Il ne s'agit que d'une moyenne : si tous les matériels ont été acquis au même moment, ils auront tendance à tomber en panne à peu près dans la même période.

L'ensemble du système distribué doit être conçu pour tolérer ces pannes et continuer à fonctionner, éventuellement en mode dégradé. La principale méthode pour faire face à des pannes est la *redondance* : on duplique par exemple systématiquement

— les données (sur des disques différents !) pour pallier les défaillances de disque ;
— les composants logiciels dont le rôle est vital et dont la défaillance impliquerait l'arrêt complet du système (*single point of failure*) sont également dupliqués, l'un d'eux assurant la tâche et le (ou les) autre(s) étant prêts à prendre la relève en cas de défaillance.

Par ailleurs, le système doit être équipé d'un mécanisme de détection des pannes pour pouvoir appliquer une méthode de reprise et assurer la disponibilité permanente. En résumé, les pannes et leur gestion automatisée constituent un des soucis majeurs dans les environnements *Cloud*.

11.1.2 Systèmes distribués

La couche logicielle qui exploite les ressources d'une ferme de serveur constitue un *système distribué*. Son rôle est de coordonner les actions de plusieurs ordinateurs connectés par un réseau, en vue d'effectuer une tâche commune. Les systèmes NoSQL ont esentiellement en commun d'être des systèmes distribués dont la tâche principale est la gestion de grandes masses de données.

Messages et protocole

Commençons par quelques caractéristiques communes et un peu de terminologie. Un système distribué est constitué de composants logiciels (des processus) que nous appellerons *nœuds*. En règle générale, on trouve un nœud sur chaque machine, mais ce n'est pas une obligation.

Ces nœuds sont interconnectés par réseau et communiquent par *échange de messages*. La connexion au réseau s'effectue par un port numéroté. Dans le cas du Web par exemple (le système distribué le plus connu et le plus fréquenté !), le port est en général le port 80. Rien n'empêche d'avoir plusieurs nœuds sur une même machine, mais il faut dans ce cas associer à chacun un numéro de port différent. La configuration du système distribué consiste à énumérer la liste des nœuds participants, référencés par l'adresse de la machine et le numéro de port.

Le format des messages obéit à un certain protocole que nous n'avons pas à connaître en général. Certains systèmes utilisent le protocole HTTP (par exemple CouchDB, ou Elastic Search pour son interface REST), ce qui présente l'avantage d'une très bonne intégration au Web et une normalisation de fait (des librairies HTTP existent dans tous les langages), mais l'inconvénient d'une certaine lourdeur. La plupart des protocoles d'échange sont spécifiques.

Clients, maîtres et esclaves

L'organisation du système distribué dépend des différents *types* de nœuds, de leur interconnexion et des relations possibles avec le nœud-client (l'application). Nous allons essentiellement rencontrer deux topologies, illustrées respectivement par la Fig. 11.4 et la Fig. 11.5.

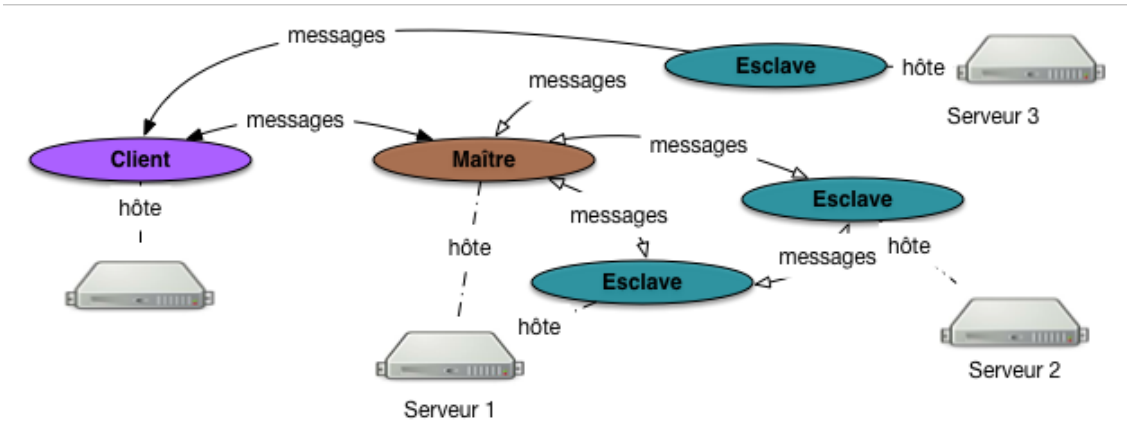


Fig. 11.4 – Une architecture avec maître-esclave

Dans la première, le système distribué est organisé selon une topologie *maître-esclave*. Un nœud particulier, le *maître*, tient un rôle central. Il est notamment chargé des tâches administratives du système

— ajouter un nœud, en supprimer un autre,
— surveiller la cohérence et la disponibilité du système,
— appliquer une méthode de reprise sur panne le cas échéant.

Il est aussi souvent chargé de communiquer avec l'application cliente (qui constitue un troisième type de nœud). Dans une telle architecture, le nœud-maître communique avec les nœuds-esclaves, qui eux-mêmes peuvent communiquer entre eux. Un client qui s'adresse au système distribué envoie ses requêtes au nœud-maître, et ce dernier peut le mettre en communication avec un ou plusieurs nœuds-esclaves. En revanche, un nœud-client ne peut pas transmettre une requête directement à un esclave, ce qui évite des problèmes de concurrence et de cohérence que nous aurons l'occasion de détailler sur des exemples pratiques.

Insistons sur le fait que les nœuds sont des composants *logiciels* (des processus qui tournent en tâche de fond) hébergés par une machine. Il n'y a pas forcément de lien un-à-un entre un nœud et une machine. La Fig. 11.4 montre par exemple que le maître et un des esclaves sont hébergés par le serveur 1. En fait, tous les nœuds peuvent être sur une même machine. Le système reste distribué, mais pas vraiment scalable !

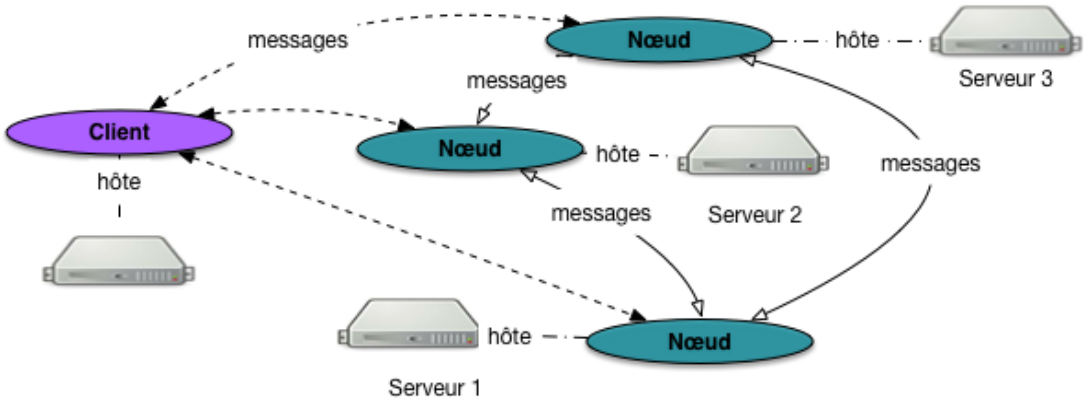


Fig. 11.5 – Une architecture multi-nœuds

Dans la seconde topologie (plus rare), il n'y a plus ni maître ni esclave, et on parlera en général de (multi-)nœuds, voire de (multi-)serveurs (au sens logiciel du terme). Dans ce cas (Fig. 11.5), le client peut indifféremment s'adresser à chaque nœud.

Cette topologie a la mérite d'éviter d'avoir un nœud particulier (le maître) dont l'existence est indispensable pour le fonctionnement global du système. On parle pour un tel nœud de *point individuel de défaillance* (*single point of failure*), situation que l'on cherche à éviter en principe. Cela rend l'ensemble du système plus fragile que si tous les nœuds ont des rôles équivalents. Cela dit, une architecture avec un maître est plus facile à mettre en œuvre en pratique et nous la rencontrerons plus souvent, associée à des dispositifs pour protéger les tâches du nœud-maître.

Gestion des pannes, ou *failover*

L'infrastructure à base de composants bon marché est soumise à des pannes fréquentes. Il est exclu de pallier ces pannes par la mobilisation permanente d'une armée d'ingénieurs système, et un caractère distinctif des systèmes distribués (et en particulier de ceux dits NoSQL) est d'être capable de fonctionner sans interruption en dépit des pannes, par application d'une méthode de reprise sur panne souvent désignée par le mot *failover*.

En règle générale, la reprise sur panne s'appuie sur deux mécanismes génériques :

— un des nœuds est chargé de surveiller en permanence l'ensemble des nœuds du système par envoi périodique de messages de contrôle (*heartbeat*) ; en cas d'interruption de la communication, on va considérer qu'une panne est intervenue ;
— la méthode de reprise s'appuie sur la redondance des services et la réplication des données : pour tout composant fautif, on doit pouvoir trouver un autre composant doté des mêmes capacités.

Signalons un cas épineux : celui d'un *partitionnement du réseau*. Dans un système avec n machines, on se retrouve avec deux sous-groupes qui ne communiquent plus, l'un constitué de m machines, l'autre des $n-m$ autres machines. Que faire dans ce cas ? Qui continue à fonctionner et comment ? Nous verrons en détail les méthodes de réplication et de reprise pour quelques systèmes représentatifs.

11.1.3 Les systèmes de stockage distribués, dits NoSQL

On peut considérer que les systèmes de gestion de données massives apparus depuis les années 2000, et collectivement regroupés sous le terme commode de « NoSQL », sont une réponse à la question : « *quel outil me permettrait de tirer parti de ma grappe de serveur pour mes données, en limitant au maximum les difficultés liés à la distribution et au parallélisme ?* »

Une solution tout à fait naturelle aurait été d'adopter les systèmes relationnels *distribués* qui existent depuis longtemps et ont fait leur preuve. Pour des raisons qui tiennent à la nature plus « documentaire » des données massives (voir le début de ce cours) et à la perception des lourdeurs de certains aspects des systèmes relationnels (jointures et transactions notamment), un autre choix s'est imposé. Il consiste à sacrifier certaines fonctionnalités (modèle, langage normalisé et puissament expressif, transactions) au profit de la capacité à se déployer dans un environnement distribué, à en tirer parti au mieux, à fournir une méthode de *failover* automatique, et à faire preuve d'élasticité pour monter en puissance par ajout de nouvelles ressources.

Note : Le terme « NoSQL » est censé signifier quelque chose comme « *Not Only SQL* », soit l'idée que les systèmes relationnels ne sont pas bons à tout faire, et que certaines niches (dont la gestion de données à très grande échelle) imposent des choix différents (en fait, des sacrifices sur les fonctionnalités avancées : modèle de données, interrogation, cohérence). Aucune personne sensée ne prétend *remplacer* les systèmes relationnels dans la très grande majorité des applications, mais tous les systèmes NoSQL prétendent faire mieux en matière de scalabilité horizontale.

Les systèmes dits « NoSQL » sont extraordinairement variés. Un de leurs points communs est d'être conçus explicitement pour une infrastructure *cloud* semblable à celle que nous avons décrite ci-dessus. On retrouve dans cette conception des principes récurrents que nous allons justement essayer de mettre en valeur dans tout ce qui suit. Résumons-les :

— capacité à exploiter de manière équilibrée un ensemble de machines en vue d'une tâche précise ;
— capacité à détecter les pannes et à s'y adapter automatiquement ;

— capacité à évoluer par ajout/suppression de nouvelles ressources matérielles, sans interruption de service.

Il n'est pas question ici d'énumérer tous les systèmes existants. Ce serait laborieux, répétitif et fragile puisqu'il en apparaît (et peut-être disparaît) tous les jours. Ce qu'il faut retenir essentiellement, c'est que :

— ces systèmes fournissent nativement une adaptation à une infrastructure distribuée ;
— les modèles de données sont principalement de nature documentaire (ceux que nous étudions principalement), graphe (réputés assez peu scalables) et … pas de modèle du tout : on stocke des chaînes d'octets !
— on peut distinguer les systèmes à orientation analytique (traitements longs appliqués à une partie significative des données à des fins statistiques) et temps réel (accès instantané, quelques millisecondes, à des unités d'informations/documents).

Malgré le côté foisonnant de la scène NoSQL, tous s'appuient sur quelques principes de base que nous allons étudier dans la suite du cours ; connaissant ces principes, il est plus facile de comprendre les systèmes. Une précision très importante : aucune normalisation dans le monde du NoSQL. Choisir un système, c'est se lier les mains avec un modèle de données, un stockage, et une interface spécifique.

Une dernière remarque pour finir : les systèmes NoSQL les plus sophistiqués (Cassandra par exemple) tendent lentement à évoluer vers une gestion de données structurées, équipée d'un langage d'interrogation qui rappelle furieusement SQL. Bref, on se dirige vers ce qui ressemble fortement à du relationnel distribué.

11.1.4 Les systèmes de calcul distribués

Enfin la dernière couche de notre architecture est constituée de systèmes de calcul distribués qui s'appuient en général sur un système de stockage NoSQL pour accéder aux données initiales. Le premier système de ce type est Hadoop, équipé d'un moteur de calcul MapReduce. Une alternative plus sophistiquée est apparue en 2008 avec Spark, qui propose d'une part des opérations plus complètes, d'autre part un système de gestion des pannes plus performant. D'autres systèmes plus ou moins équivalents existent, dont Flink qui est spécialisé pour le traitement de grand flux de données.

Tout système informatique repose sur un ensemble de mécanismes de stockage et d'accès à l'information, les *mémoires*. Ces mémoires se différencient par leur prix, leur rapidité, le mode d'accès aux données (séquentiel ou par adresse) et enfin leur durabilité.

La gestion des mémoires est une problématique fondamentale des SGBD : pour une révision, je vous renvoie au chapitre « stockage » de mon cours sur les aspects systèmes des bases de données. Les notions essentielles sont revues ci-dessous, et reprise dans le cadre d'une ferme de serveur où le réseau et la topologie du réseau jouent un rôle important.

11.1.5 La hiérarchie des mémoires

D'une manière générale, plus une mémoire est rapide, plus elle est chère et – conséquence directe – plus sa capacité est réduite. Dans un cadre centralisé, mono-serveur, les mémoires forment une hiérarchie classique illustrée par la Fig. 11.6, allant de la mémoire la plus petite mais la plus efficace à la mémoire la plus volumineuse mais la plus lente. Un bonne partie du travail d'un SGBD consiste à placer l'information requise par une application le plus haut possible dans la hiérarchie des mémoires : dans le cache du processeur idéalement ; dans la mémoire RAM si possible ; au pire il faut aller la chercher sur le disque, et là ça coûte très cher.

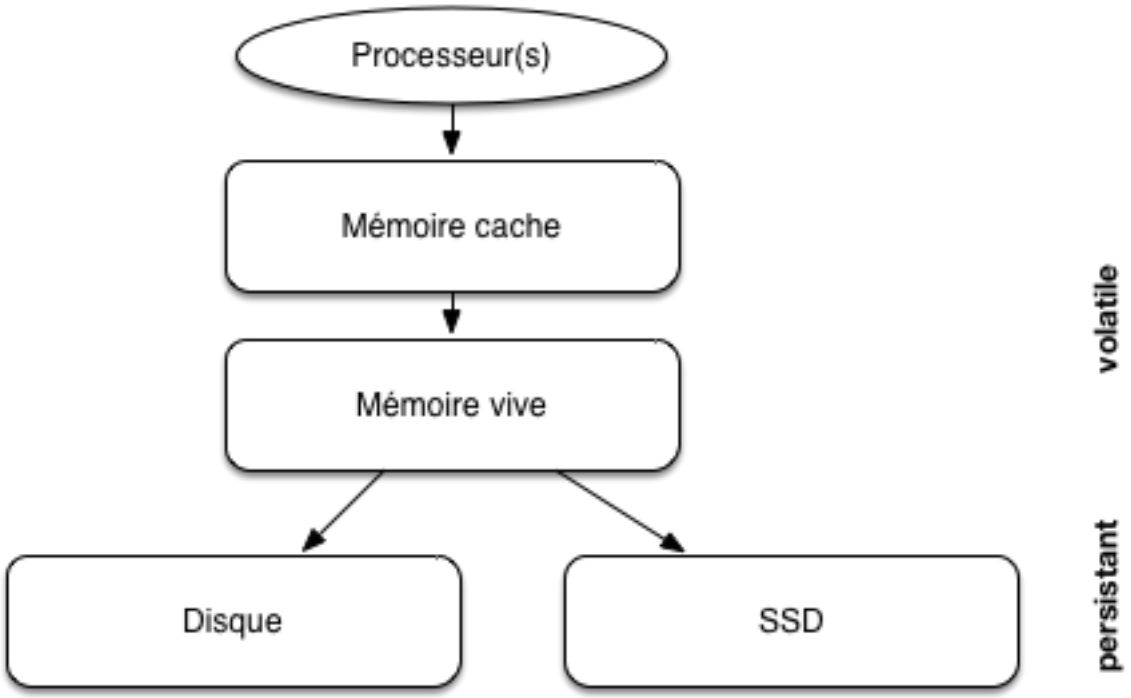


Fig. 11.6 – Hiérarchie des mémoires dans un serveur

La mémoire vive (que nous appellerons mémoire principale) et les disques (ou mémoire secondaire) sont les principaux niveaux à considérer pour des applications de bases de données. Une base de données est à peu près toujours stockée sur disque, pour les raisons de taille et de persistance, mais les données doivent impérativement être placées en mémoire vive pour être traitées.

Important : La mémoire RAM est une mémoire *volatile* dont le contenu est effacé lors d'une panne. Il est donc essentiel de synchroniser la mémoire avec le disque périodiquement (typiquement au moment d'un *commit*).

Si on considère maintenant un *cloud*, la hiérarchie se complète avec les liens réseaux connectant les serveurs. Du côté du réseau aussi on trouve une hiérarchie : les serveurs d'une même baie sont liés par un réseau rapide, mais les baies elles-mêmes sont liées par des routeurs qui limitent le débit des échanges.

La Fig. 11.7 illustre les mémoires et leur communication dans une ferme de serveur. On pourrait y ajouter un troisième niveau de communication réseau, celui entre deux fermes de serveurs distinctes.

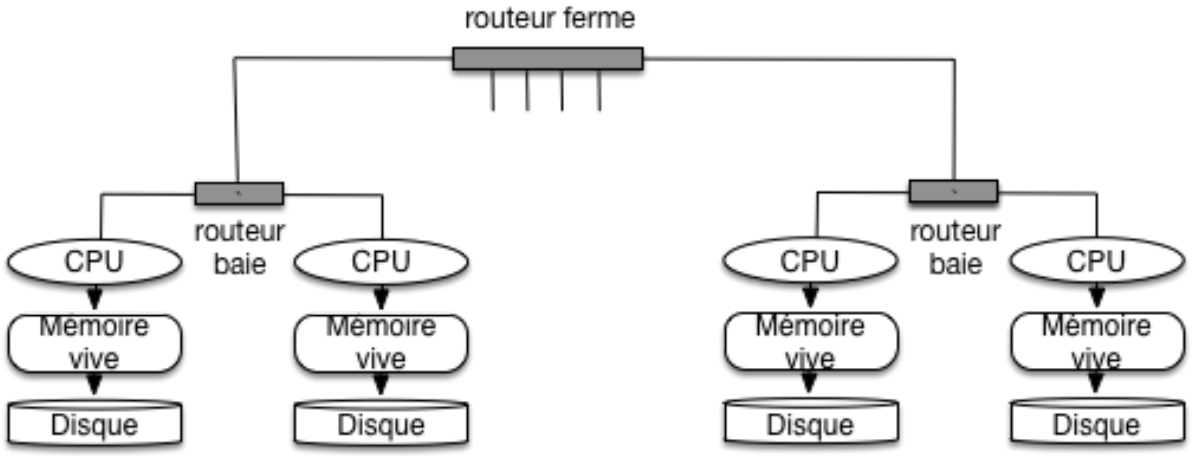


Fig. 11.7 – Hiérarchie des mémoires dans une ferme de serveurs

11.1.6 Performances

On peut évaluer (par des ordres de grandeur, car les variations sont importantes en fonction du matériel) les performances d'un système tel que celui de la Fig. 11.7 selon deux critères :

— *Temps d'accès*, ou *latence* : connaissant *l'adresse* d'un document, quel est le temps nécessaire pour aller à l'emplacement mémoire indiqué par cette adresse et obtenir le document ? On parle de lecture par clé ou encore *d'accès direct* pour cette opération ;
— *Débit* : quel est le volume de données lues par unité de temps dans le meilleur des cas ?

Important : Dans le cas d'un disque magnétique, le débit s'applique une lecture *séquentielle* respectant l'ordre de stockage des données sur le support. Si on effectue la lecture dans un ordre différent, il s'agit en fait d'une séquence d'accès directs, et c'est extrêmement lent.

Le temps d'un accès *direct* en mémoire vive est par exemple de l'ordre de 10 nanosecondes (10^{-8} sec.), de 0,1 millisecondes pour un SSD, et de l'ordre de 10 millisecondes (10^{-2} sec.) pour un disque. Cela représente un ratio approximatif de 1 000 000 (1 million !) entre les performances respectives de la mémoire centrale et du disque magnétique ! Un SSD est très approximativement 100 fois plus rapide (en accès direct) qu'un disque magnétique.

Tableau 11.1 – Performance des divers types de mémoire

Type mémoire	Taille	Temps d'accès aléatoire	Débit en accès séquentiel
Mémoire *cache* (Static RAM)	Quelques MOs	$\approx 10^{-8}$ (10 nanosec.)	Plusieurs dizaines de GOs par seconde
Mémoire principale (Dynamic RAM)	Quelques GOs	$\approx 10^{-8} - 10^{-7}$ (10-100 nanosec.)	Quelques GO par seconde
Disque magnétique	Quelques TOs	$\approx 10^{-2}$ (10 millisec.)	Env. 100 MOs par seconde.
SSD	Quelques TOs	$\approx 10^{-4}$ (0,1 millisec.)	Jusqu'à quelques GOs par seconde.

En ce qui concerne les composants réseau, la méthode la plus courante est d'utiliser des routeurs Ethernet 48 ports offrant un débit d'environ 10 GigaBits/sec. On utilise un routeur de ce type pour chaque baie, en connectant par exemple les 40 serveurs de la baie. À un second niveau, les routeurs-baie sont connectés par un routeur-ferme qui se charge de mettre en communication les serveurs de baies différentes (Fig. 11.7). Les 8 ports restant au niveau de chaque baie sont par exemple utilisés pour cette connexion globale. Cela introduit un facteur d'agrégation (*oversubscription*) puisque chaque port « global » doit gérer le débit de 5 serveurs de la baie.

Note : Vous avez compris la phrase qui précède ? Si non, réfléchissez.

	RAM locale	Disque local	RAM Baie	Disque baie	RAM cloud	Disque cloud
Latence (micro sec)	0.1	10 000	300	10 000	500	10 000
Débit (en MO/s)	10 000	100	125	100	25	20

Le tableau ci-dessus donne les ordres de grandeur pour la latence et le débit au sein de la hiérarchie de mémoire (il faudrait ajouter les SSDs, cf. les performances de ces derniers). Ce ne sont que des estimations : le débit de 20 MO/s par exemple est obtenu en considérant que le facteur d'agrégation au niveau de chaque baie est de 5 (soit 100 MO/s divisé par 5), et en supposant que le trafic est équitablement réparti entre les 5 serveurs partageant un même port. L'accès la RAM d'un serveur en dehors de la baie subit l'effet combiné du réseau (10 Gbits/s, soit 1,25 GO/s) et du facteur 5.

Il reste à donner une idée du coût. À ce jour (2024) voici ce qu'il en est pour une mémoire de 1 TO.

— RAM : environ 3 000 $
— SSD : environ 100 $
— Disque : environ 20 $

C'est sans doute moins cher si on achète en gros, mais cela donne une idée du rapport. Le SSD est très attractif mais reste encore presque 5 fois plus cher qu'un disque magnétique classique.

11.1.7 Quiz

11.2 S2 : La scalabilité

Supports complémentaires :
- Présentation: Scalabilité
- Vidéo de la session consacrée à la scalabilité

Il est temps de revenir sur la notion de scalabilité pour la définir précisément, et de donner quelques exemples pour comprendre en pratique ce que cela implique.

11.2.1 Une définition

Voici une définition assez générale, basée sur les notions (à expliciter) de « système », « performance » et « ressource ».

Définition (scalabilité).

Un système est *scalable* si ses *performances* sont proportionnelles aux *ressources* qui lui sont allouées.

Il s'agit d'une notion très stricte de la scalabilité, qu'il est difficile d'obtenir en pratique mais nous donne une idée précise du but à atteindre.

Explicitons maintenant les notions sur lesquelles repose la définition. Dans notre cas, la notion de *système* est assez bien identifiée : il s'agit de l'environnement distribué de gestion/traitement de données basé sur l'architecture de la Fig. 11.1.

Les *ressources* sont les composants matériels que l'on peut ajouter au système. Il s'agit essentiellement des serveurs, mais on peut aussi prendre en compte des dispositifs liés au réseau comme les routeurs. La consommation des ressources s'évalue essentiellement selon les unités de grandeur suivantes :
- la mémoire RAM alouée au système ;
- le temps de calcul ;
- la mémoire secondaire (disque)
- la bande passante (réseau).

La notion de *performance* est la plus flexible. Voici les deux acceptions principales que nous allons étudier.
- *débit* : c'est le nombre d'unités d'information (documents) que nous pouvons traiter par unité de temps, toutes choses égales par ailleurs ;
- *latence* : c'est le temps mis pour accéder à une unité d'information (document), en lecture et/ou en écriture.

Au lieu de mesurer le débit en documents/seconde, on regarde souvent le nombre d'octets (par exemple, 10 MO/s). C'est équivalent si on accepte que les documents ont une taille moyenne avec un écart-type pas trop élevé. En ce qui concerne la latence, on mesure souvent le nombre d'accès par seconde sur un nombre important d'opérations, afin de lisser les écarts, et on nomme cette mesure *transactions par seconde* (abrégé par tps).

11.2.2 Exemple : comptons les documents

Pour mettre en évidence la scalabilité, conformément à la définition ci-dessus, il faut quantifier les grandeurs *ressource* et *performance* et montrer que leur rapport est constant *pour un volume de données fixé*. Prenons un premier exemple : on veut compter le nombre de documents dans le système. Charge serveur effectue indépendamment son propre comptage. Il exécute pour cela une opération qui consiste à charger en mémoire centrale les données stockées sur des disques magnétiques. On mesure le débit de cette opération en MO/s (Fig. 11.8).

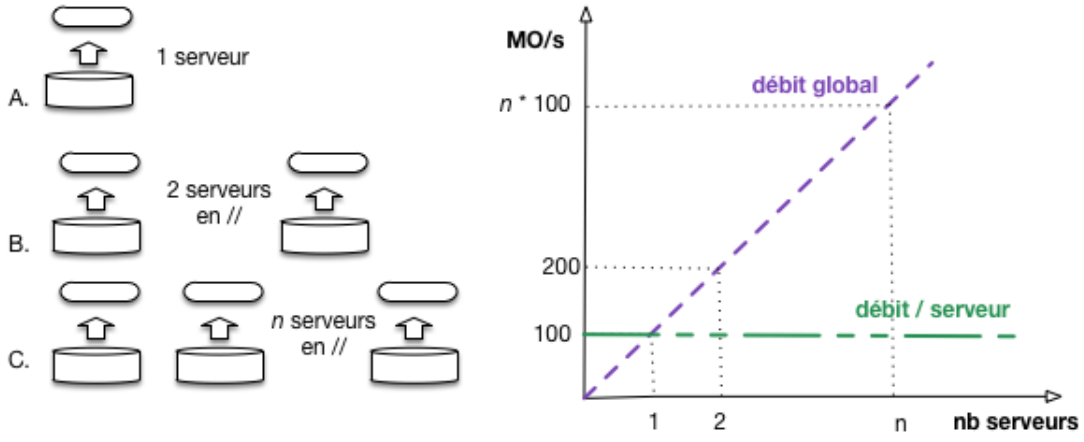


Fig. 11.8 – Mesure de scalabilité

Avec un seul serveur, on constate un débit de 100 MO/s. Avec deux serveurs, on peut répartir les données équitablement et effectuer les lectures en parallèle : on obtient un débit de 200 MO/s. Il n'est pas difficile d'extrapoler et de considérer que si on a *n* serveurs, le débit sera de *n* * 100 MO/s.

Attention : dans ce scénario on suppose que tous les accès sont de nature *locale*, et qu'il n'y a pas d'échange entre les serveurs. À la fin de l'opération on obtient le nombre de documents sur chaque serveur, et cette information (un entier sur 4 ou 8 octets) est suffisamment petite pour être transférée au client qui effectue l'agrégation.

Le traitement est scalable. La figure montre le débit global : c'est bien une droite exprimant la dépendance linéaire entre le nombre de ressources et la performance. La droite en vert montre que le débit par serveur est constant : c'est une condition nécessaire mais pas suffisante pour garantir la scalabilité au niveau du système global. Si toutes les données devaient par exemple transiter dans un unique tuyau de débit 1 GO/s, au bout de 10 serveurs en parallèle ce tuyau constituerait le goulot d'étranglement du système.

Comme le montre cet exemple simple, la scalabilité nécessite d'envisager le système de manière globale, en veillant à l'équilibre de tous ses composants.

11.2.3 Autre exemple : cherchons les doublons

Prenons un exemple un peu plus complexe. On veut chercher d'éventuels doublons dans notre collection de vidéos. Après d'intenses réflexions, le groupe d'ingénieur NFE204 décide d'implanter un algorithme en deux étapes.

— la première parcourt les vidéos et produit, pour chacune, une *signature* (cf. http://en.wikipedia.org/wiki/Digital_video_fingerprinting) ; cette première étape se fait localement ;
— la seconde étape regroupe les signatures ; si deux signatures (ou plus) égales sont trouvées, c'est un doublon !

La première étape est à peu de choses près le parcours en parallèle de tous les disques, associé à un traitement local pour extraire la signature (Fig. 11.9). Le résultat est une liste de paires *(id, signature)* où l'id de chaque document est associé à sa signature.

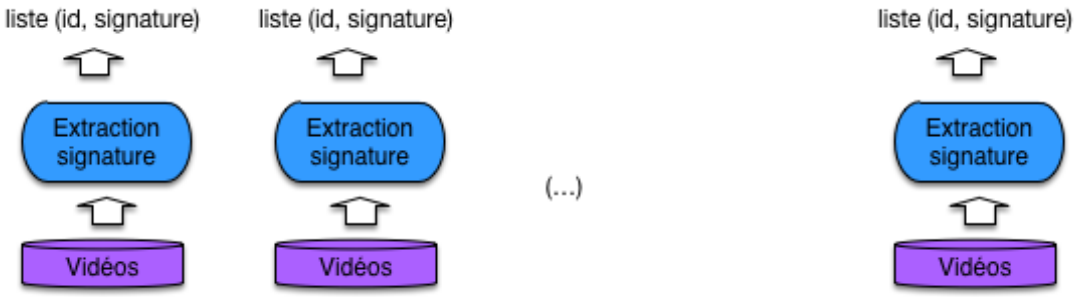


Fig. 11.9 – Extraction des signatures

La seconde étape va nécessiter des échanges réseau. On va associer chaque serveur à une partie des signatures, par hachage. Par exemple (très simple), si on a *n* serveurs, on va associer une signature *h* au serveur *mod(h, n)*. La Fig. 11.10 illustre le cas de 3 serveurs : les signatures *X1* et *X4* sont envoyées au serveur 1 puisque *mod(1,3)=mod(4,3)=1*.

On envoie alors chaque paire *(i, s)* issue de l'étape 1, constituée d'un identifiant *i* et d'une signature *s* au serveur associé à *s*. Si deux signatures sont égales, elles se retrouveront sur le même serveur. Il suffit donc d'effectuer, *localement*, une comparaison de signatures pour reporter les doublons. C'est le cas pour les documents i1 et i8 dans la Fig. 11.10.

Le traitement est-il scalable ? Oui en ce qui concerne la première étape, pour les raisons analysées dans le premier exemple : le traitement s'effectue localement sur chaque serveur, de manière indépendante.

La seconde étape implique un transfert réseau de l'ensemble des listes produites dans la première étape. Il faut donc tout d'abord comparer le coût du transfert réseau, celui de la lecture des données, et celui du traitement. Les données échangées (ici, des paires de valeurs) sont certainement beaucoup plus petites que les documents. Il est possible alors que le transfert réseau de ces listes soit 100 ou 1000 fois moins coûteux que le parcours des documents et l'extraction de la signature. Dans ce cas le coût prépondérant est celui de la première étape, et le traitement global est scalable.

Si le coût des transferts réseaux est comparable (ou supérieur) à celui des traitements, il faudrait, pour que la scalabilité soit stricte, qu'il soit possible d'améliorer le débit dans la grappe de serveur proportionnellement au nombre de serveurs. C'est typiquement difficile, à moins de recourir à du matériel de connectique très sophistiqué et donc très coûteux. Il est plus probable que le débit restera constant ou, pire se *dégradera* avec l'ajout de nouveaux serveurs. L'échange réseau devient le facteur entravant la scalabilité.

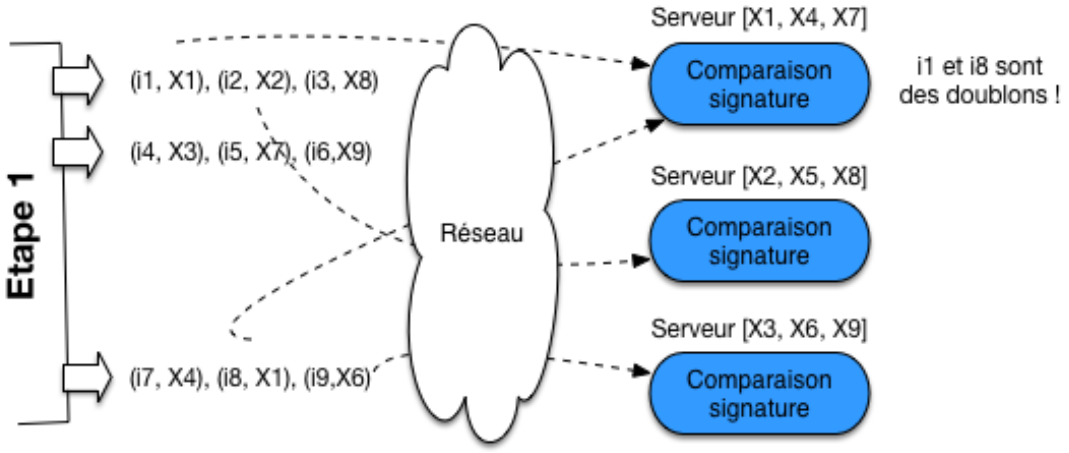


Fig. 11.10 – Transfert des signatures et détection des doublons.

11.2.4 Quelques conclusions

En pratique, il est difficile d'augmenter les performances du réseau proportionnellement aux ressources, et les transferts sont l'un des facteurs qui peuvent limiter la scalabilité en pratique. La situation la plus favorable est celle de notre premier exemple, où l'essentiel des traitements se fait *localement*, avec un résultat de taille négligeable qu'il est ensuite possible de transférer à une autre machine ou à l'application cliente. Plus généralement, il faut être attentif à limiter le plus possible la taille des données échangées entre les serveurs de manière à ce que le coût des transferts reste négligeable par rapport à celui des traitements.

Le fait d'être scalable (selon notre définition) ne veut pas dire que le temps de calcul est *acceptable*. Si un traitement prend 10 ans, il prendra encore 5 ans en doublant les ressources… La première chose à faire est de s'en apercevoir à l'avance en effectuant des tests et en mesurant le coût des éléments dans la chaîne de traitement. Ensuite, il faut voir si on peut optimiser.

Quand un traitement est complètement optimisé, et que la durée prévisible reste trop élevée, la question suivante est : suis-je prêt à allouer les ressources nécessaires ? Et là, si la réponse est non, il est temps de reconsidérer la définition du problème. La définition du BigData, ça pourrait être : toutes les données que je peux me permettre de stocker, et pour lesquelles il existe au moins un traitement assez intéressant au vu des ressources que je dois y consacrer. À méditer.

11.2.5 Quiz

11.3 3 : Calculs distribués

Supports complémentaires

— Diapositives: MapReduce et traitements à grande échelle
— Vidéo de la session MapReduce

Reportez-vous au chapitre *Recherche exacte* pour une présentation du *modèle* MapReduce d'exécution. Rappelons que MapReduce *n'est pas* un langage d'interrogation de données, mais un modèle d'exécution de *chaînes de traitement* dans lesquelles des données (massives) sont progressivement transformées et enrichies.

Pour être concrets, nous allons prendre l'exemple (classique) d'un traitement s'appliquant à une collection de documents textuels et déterminant la *fréquence des termes dans les documents* (indicateur TF, cf. *Recherche approchée*). Pour chaque terme présent dans la collection, on doit donc obtenir le nombre d'occurrences.

11.3.1 Le principe de localité des données

Dans une approche classique de traitement de données stockées dans une base, on utilise une architecture client-serveur dans laquelle l'application cliente reçoit les données du serveur. Cette méthode est difficilement applicable en présence d'un très gros volume de données, et ce d'autant moins que les collections sont stockées dans un système distribué. En effet :

— le transfert par le réseau d'une large collection devient très pénalisant à grande échelle (disons, le TéraOctet) ;
— et surtout, la distribution des données est une opportunité pour effectuer les calculs *en parallèle* sur chaque machine de stockage, opportunité perdue si l'application cliente fait tout le calcul.

Ces deux arguments se résument dans un principe dit de *localité des données* (*data locality*). Il peut s'énoncer ainsi : les meilleures performances sont obtenues quand chaque fragment de la collection est traité *localement*, minimisant les besoins d'échanges réseaux entre les machines.

Note : Reportez-vous au chapitre *Le cloud, une nouvelle machine de calcul* pour une analyse quantitative montrant l'intérêt de ce principe.

L'application du principe de localité des données mène à une architecture dans laquelle, contrairement au client-serveur, les données ne sont pas transférées au programme client, mais le programme distribué à toutes les machines stockant des données (Fig. 11.11).

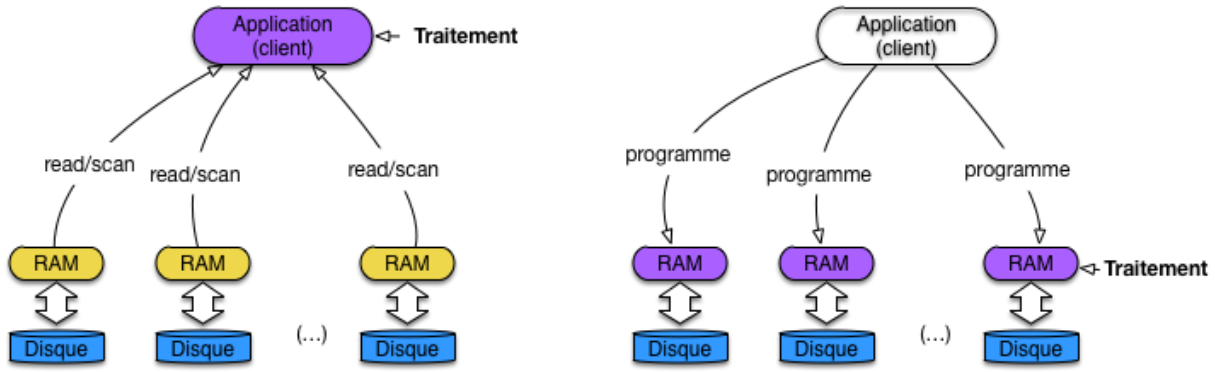


Fig. 11.11 – Principe de localité des données, par transfert des programmes

En revanche, demander à un développeur d'écrire une application distribuée basée sur ce principe constitue un défi technique de grande ampleur. Il faut en effet concevoir simultanément les tâches suivantes :

— implanter la *logique* de l'application, autrement dit le traitement particulier qui peut être plus ou moins complexe ;
— concevoir la parallélisation de cette application, sous la forme d'une exécution concurrente coordonnant plusieurs machines et assurant un accès à un partitionnement de la collection traitée ;
— et bien entendu, gérer la reprise sur panne dans un environnement qui, nous l'avons vu, est instable.

Un *framework* d'exécution distribuée comme MapReduce est justement dédié à la prise en charge des deux derniers aspects, spécifiques à la distribution dans un *cloud*, et ce de manière *générique*. Le *framework* définit un processus immuable d'accès et de traitement, et le programmeur implante la logique de l'application sous la forme de briques logicielles confiées au *framework* et appliquées par ce dernier dans le cadre du processus.

Avec MapReduce, le processus se déroule en deux phases, et les « briques logicielles » consistent en deux fonctions fournies par le développeur. La phase de Map traite chaque document individuellement et applique une fonction *map()* dont voici le pseudo-code pour notre application de calcul du TF.

```
function mapTF($id, $contenu)
{
  // $id: identifiant du document
  // $contenu: contenu textuel du document

  // On boucle sur tous les termes du contenu
  foreach  ($t in $contenu) {
    // Comptons les occurrences du terme dans le contenu
    $count = nbOcc ($t, $contenu);
    // On "émet" le terme et son nombre des occurrences
    emit ($t, $count);
  }
}
```

La phase de Reduce reçoit des valeurs groupées sur la clé et applique une agrégation de ces valeurs. Voici le pseudo-code pour notre application TF.

```
function reduceTF($t, $compteurs)
{
  // $t: un terme
  // $compteurs: la séquence des décomptes effectués localement par le Map
  $total = 0;

  // Boucles sur les compteurs et calcul du total
  foreach ($c in $compteurs) {
   $total = $total + $c;
  }

  // Et on produit le résultat
  return $total;
}
```

Dans ce cadre restreint, le *framework* prend en charge la distribution et la reprise sur panne.

Important : Ce processus en deux phases et très limité et ne permet pas d'exprimer des algorithmes complexes, ceux basés par exemple sur une itération menant progressivement au résultat. C'est l'objectif essentiel de modèles d'exécution plus puissants que nous présentons ultérieurement.

11.3.2 Exécution distribuée d'un traitement MapReduce

La Fig. 11.12 résume l'exécution d'un traitement (»*job* ») MapReduce avec un *framework* comme Hadoop. Le système d'exécution distribué fonctionne sur une architecture maître-esclave dans laquelle le maître (*JobTracker* dans Hadoop) se charge de recevoir la requête de l'application, la distribue sous forme de tâche à des nœuds (*TaskTracker* dans Hadoop) accédant aux fragments de la collection, et coordonne finalement le déroulement de l'exécution. Cette coordination inclut notamment la gestion des pannes.

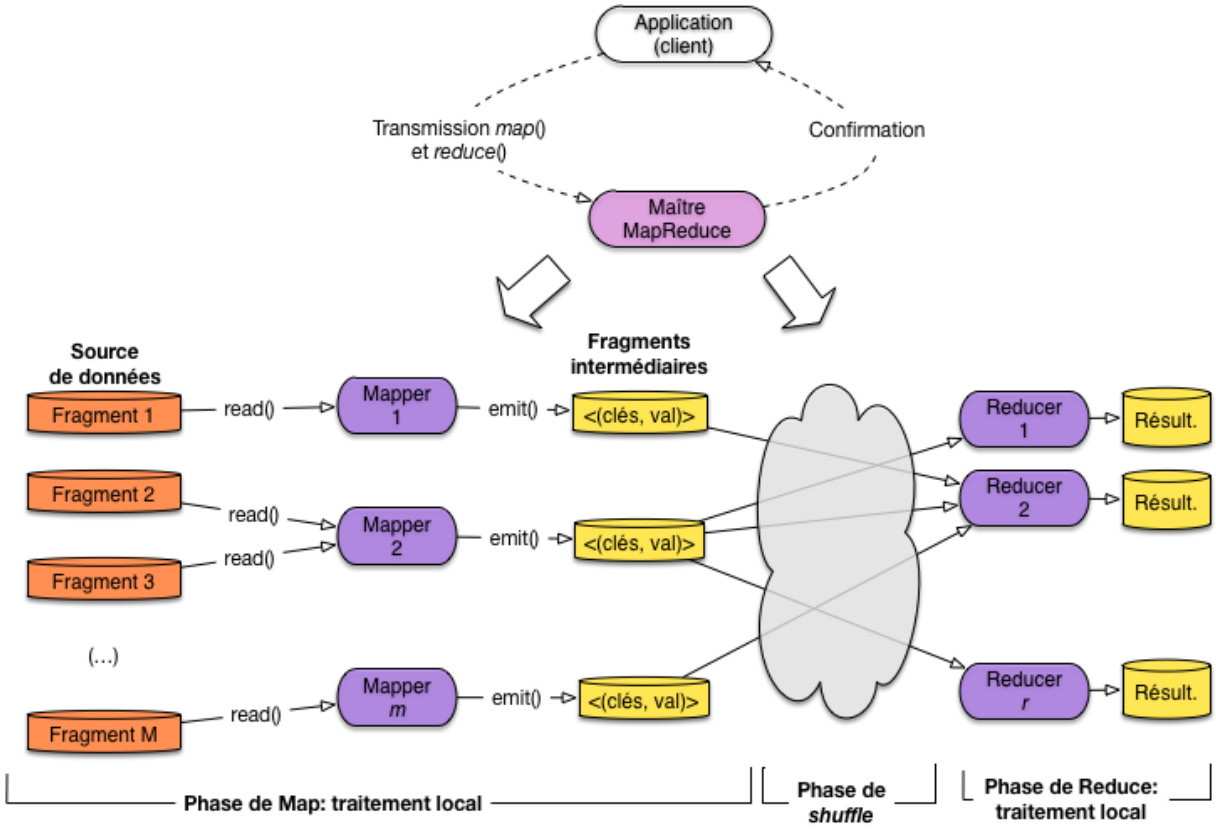


Fig. 11.12 – Exécution distribuée d'un traitement MapReduce

L'application cliente se connecte au maître, transmet les fonctions de Map et de Reduce, et soumet la demande d'exécution. Le client est alors libéré, en attente de la confirmation par le maître que le traitement est terminé (cela peut prendre des jours ...). Le framework fournit des outils pour surveiller le progrès de l'exécution pendant son déroulement.

Le traitement s'applique à une source de données partitionnée. Cette source peut être un simple système de fichiers distribués, un système relationnel, un système NoSQL type MongoDB ou HBase, voire même un

moteur de recherche comme Solr ou ElasticSearch.

Le Maître dispose de l'information sur le partitionnement des données (l'équivalent du contenu de la table de routage, présenté dans le chapitre sur le partitionnement) ou la récupère du serveur de données. Un nombre *M* de serveurs stockant tous les fragments concernés est alors impliqué dans le traitement. Idéalement, ces serveurs vont être chargés eux-mêmes du calcul pour respecter le principe de localité des données mentionné ci-dessus. Un système comme Hadoop fait de son mieux pour respecter ce principe.

La fonction de Map est transmise aux *M* serveurs et une tâche dite *Mapper* applique la fonction à un fragment. Si le serveur contient plusieurs fragments (ce qui est le cas normal) il faudra exécuter autant de tâches. Si le serveur est multi-cœurs, plusieurs fragments peuvent être traités en parallèle sur la même machine.

Exemple : le partitionnement des données pour l'application TF

Supposons par exemple que notre collection contienne 1 milliard de documents dont la taille moyenne est de 1000 octets. On découpe la collection en fragments de 64 MOs. Chaque fragment contient donc 64 000 documents. Il y a donc à peu près $\lceil 10^9/64,000 \rceil \approx 16,000$ fragments. Si on dispose de 16 machines, chacune devra traiter (en moyenne) 1000 fragments et donc exécuter mille tâches de *Mapper*.

Le parallélisme peut alors être interne à une machine, en fonction du nombre de *cores* dont elle dispose. Une machine *4 cores* pourra ainsi effectuer 4 tâches en parallèle en théorie.

Chaque *mapper* travaille, dans la mesure du possible, *localement* : le fragment est lu sur le disque *local*, document par document, et l'application de la fonction de Map « émet » des paires (clé, valeur) dites « intermédiaires » qui sont stockées sur le disque *local*. Il n'y a donc aucun échange réseau pendant la phase de Map (dans le cas idéal où la localité des données peut être complètement respectée).

Exemple : la phase de Map pour l'application TF

Supposons que chaque document contienne en moyenne 100 termes distincts. Chaque fragment contient 64 000 documents. Un *Mapper* va donc produire 6 400 000 paires *(t, c)* où *t* est un terme et *c* le nombre d'occurrences.

À l'issue de la phase de Map, le maître initialise la phase de Reduce en choisissant *R* machines disponibles. Il faut alors distribuer les paires intermédiaires à ces *R* machines. C'est une phase « cachée », dite de *shuffle*, qui constitue potentiellement le goulot d'étranglement de l'ensemble du processus car elle implique la lecture sur les disques des *Mappers* de toutes les paires intermédiaires, et leur envoi par réseau aux machines des *Reducers*.

Important : Vous noterez peut-être qu'une solution beaucoup plus efficace serait de transférer immédiatement par le réseau les paires intermédiaires des *Mappers* vers les *Reducers*. Il y a une explication à ce choix en apparence sous-optimal : c'est la reprise sur panne (voir plus loin).

Pour chaque paire intermédiaire, un simple algorithme de hachage permet de distribuer les clés équitablement sur les *R* machines chargées du Reduce.

Au niveau d'un *Reducer* R_i, que se passe-t-il ?

— Tout d'abord il faut récupérer *toutes* les paires intermédiaires produites par les *Mappers* et affectées à R_i.
— Il faut ensuite *trier* ces paires sur la clé pour regrouper les paires partageant la même clé. On obtient des paires *(k, [v])* où *k* est une clé, et *[v]* la liste des valeurs reçues par le *Reducer* pour cette clé.
— Enfin, chacune des paires *(k, [v])* est soumise à la fonction de Reduce.

Exemple : la phase de Reduce pour l'application TF

Supposons *R=10*. Chaque *Reducer* recevra donc en moyenne 640 000 paires *(t, c)* de chaque *Mapper*. Ces paires sont triées sur le terme *t*. Pour chaque terme on a donc la liste des nombres d'occurences trouvés dans chaque document par les *Mappers*. Au pire, si un terme est présent dans chaque document, le tableau *[v]* contient un million d'entiers.

Il reste, avec la fonction de Reduce, à faire le total de ces nombres d'occurences pour chaque terme.

Exemple : comptons les loups et le moutons

Vous souvenez-vous de ces quelques documents ?
— A : `Le loup est dans la bergerie.`
— B : `Les moutons sont dans la bergerie.`
— C : `Un loup a mangé un mouton, les autres loups sont restés dans la bergerie.`
— D : `Il y a trois moutons dans le pré, et un mouton dans la gueule du loup.`

Ils sont maintenant stockés dans un système partitionné sur 3 serveurs comme montré sur la Fig. 11.13. Nous appliquons notre traitement TF pour compter le nombre total d'occurrences de chaque terme (on va s'intéresser aux termes principaux).

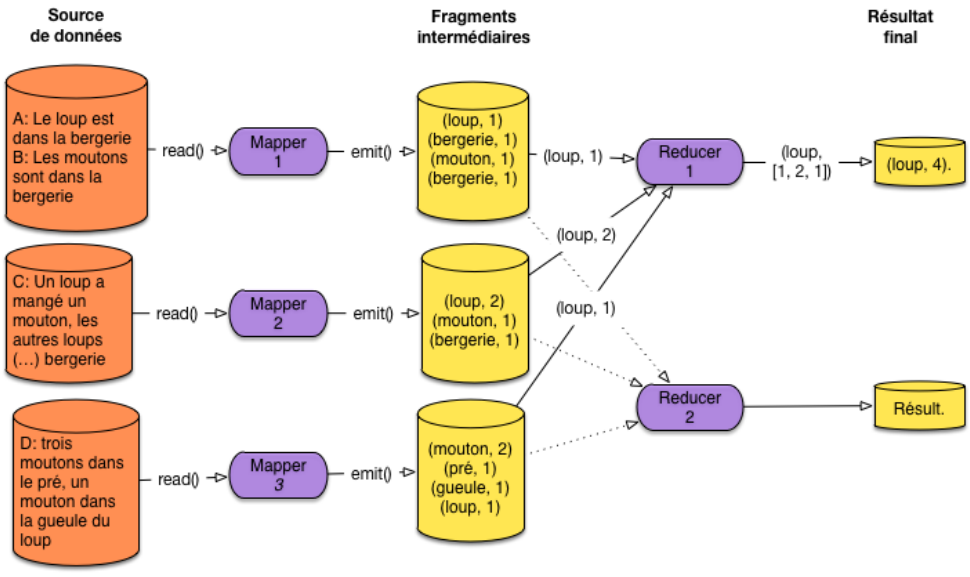


Fig. 11.13 – Un exemple minuscule mais concret

Nous avons trois *Mappers* qui produisent les données intermédiaires présentées sur la figure. Comprenez-vous pourquoi le terme *bergerie* apparaît deux fois pour le premier *Mapper* par exemple ?

La phase de Reduce, avec 2 *Reducers*, n'est illustrée que pour le terme *loup* donc on suppose qu'il est affecté au premier *Reducer*. Chaque *Mapper* transmet donc ses paires intermédiaires *(loup, ...)* à *R1* qui se charge de regrouper et d'appliquer la fonction de Reduce.

Quand tous les *Reducers* ont terminé, le résultat est disponible sur leur disque local. Le client peut alors le récupérer.

11.3.3 La reprise sur panne

Comment assurer la gestion des pannes pour une exécution MapReduce ? Dans la mesure où elle peut consister en centaines de tâches individuelles, il est inenvisageable de reprendre l'ensemble de l'exécution si l'une de ces tâches échoue, que ce soit en phase de Map ou en phase de Reduce. Le temps de tout recommencer, une nouvelle panne surviendrait, et le *job* ne finirait jamais.

Le modèle MapReduce a été conçu dès l'origine pour que la reprise sur panne puisse être gérée au niveau de chaque tâche individuelle, et que la coordination de l'ensemble soit également résiliente aux problèmes de machine ou de réseau.

Le Maître délègue les tâches aux machines et surveille la progression de l'exécution. Si une tâche semble interrompue, le Maître initie une action de reprise qui dépend de la phase.

Panne en phase de Reduce

Si la machine reste accessible et que la panne se résume à un échec du processus, ce dernier peut être relancé sur la même machine, et si possible sur les données locales déjà transférées par le *shuffle*. C'est le cas le plus favorable.

Dans un cas plus grave, avec perte des données par exemple, une reprise plus radicale consiste à choisir une autre machine, et à relancer la tâche en réinitialisant le transfert des paires intermédiaires depuis les machines chargées du Map. C'est possible car ces paires ont été écrites sur les disques locaux et restent donc disponibles. C'est une caractéristique très importante de l'exécution MapReduce : *l'écriture complète des fragments intermédiaires garantit la possibilité de reprise en cas de panne*.

Une méthode beaucoup plus efficace mais beaucoup moins robuste consisterait à ce que chaque *mapper* transfère immédiatement les paires intermédiaires, sans écriture sur le disque local, vers la machine chargée du Reduce. Mais en cas de panne de ce dernier, ces paires intermédiaires risqueraient de disparaître et on ne saurait plus effectuer la reprise sur panne (sauf à ré-exécuter l'ensemble du processus).

Cette caractéristique explique également la lenteur (désépérante) d'une exécution MapReduce, due en grande partie à la nécessité d'effectuer des écritures et lectures répétées sur disque, à chaque phase.

Panne en phase Map

En cas de panne pendant l'exécution d'une tâche de Map, on peut soit reprendre la tâche sur la même machine si c'est le processus qui a échoué, soit transférer la tâche à une autre machine. On tire ici parti de la *réplication* toujours présente dans les systèmes distribués : quel que soit le fragment stocké sur une machine, il existe un réplica de ce fragment sur une autre, et à partir de ce réplica une tâche équivalente peut être lancée.

Le cas le plus pénalisant est la panne d'une machine pendant la phase de transfert vers les Reducers. Il faut alors reprendre toutes les tâches initialement allouées à la machine, en utilisant la réplication.

Et le maître ?

Finalement, il reste à considérer le cas du Maître qui est un point individuel d'échec : en cas de panne, il faut tout recommencer.

L'argument des *frameworks* comme Hadoop est qu'il existe un Maître pour des dizaines de travailleurs, et qu'il est peu probable qu'une panne affecte directement le serveur hébergeant le nœud-Maître. Si cela arrive, on peut accepter de reprendre l'ensemble de l'exécution, ou prendre des mesures préventives en dupliquant toutes les données du Maître sur un nœud de secours.

11.3.4 Quiz

11.4 Exercices

Exercice Ex-S1-1 : quel *cloud* pour notre application ?

À partir de maintenant on va se placer dans le scénario d'une gestion (légale !) de vidéos à la demande. La collection est essentiellement constituée de vidéos d'une taille moyenne de 500 MO. À chaque vidéo on associe quelques méta données (de taille négligeable) comme son titre, ses auteurs, l'année de réalisation, etc. Au départ on a 10 000 vidéos, mais on espère rapidement en gérer 1 000 000 (un million).

Faites une petite étude économique pour déterminer quel serait le coût annuel d'une grappe de serveurs pour stocker ce million de vidéos. Regardez les offres de quelques fournisseurs : Amazon EC2, Microsoft Azure, Google Cloud Computing, OVH, Gandi, ...

À vous de choisir la configuration de vos serveurs. Vous avez un budget serré ! Et n'oubliez pas
- de prendre en compte la réplication pour faire face aux pannes inévitables.
- de prendre en compte le coût du traffic réseau pour installer vos vidéos et les transmettre à vos clients.

Vous avez le droit de faire un fichier Excel (ou équivalent) résumant les tailles et performance de vos composants, le coût d'exploitation, etc. Vous aurez un coût unitaire par vidéo, et en fonction du nombre de clients, vous pouvez même déterminer un tarif (merci de rémunérer les auteurs).

Correction

Un million de vidéos, cela fait 500 millions de MOs, soit 500 000 GOs, soit 500 TOs. Commençons par ne pas prendre en compte la réplication. Je regarde, par exemple, chez Gandi (https://www.gandi.net/hebergement/serveur/prix).

Je peux configurer un serveur avec 8 cores, 16 GOs de RAM et trois disques de 1 TO chacun, pour 335 Euros/mois.

Hum pour mes 500 TOs il me faut 170 serveurs, soit 57 KE/mois ! ! C'est un peu moins cher si je mets 4 voire 5 disques par serveur, mais là je risque la congestion. À étudier avec un ingénieur système/réseau expert.

Conclusion : ça me revient environ 6 cts/vidéo pour l'infrastructure.

Si on veut un taux de réplication de 2 (c'est le minimum, 3 c'est mieux), il faut doubler les chiffres.

Exercice Ex-S1-2 : combien de pannes ?

Essayons d'estimer le taux de panne dans mon système. On dispose des données suivantes :

- le taux annuel de panne (*Annual Failure Rate*) de mes disques est de 4% ;
- chaque serveur se plante 3 fois par an en moyenne ;
- je constate que chaque baie se trouve coupée du réseau 3 fois par mois, coupure d'une durée suffisante pour que je doive appliquer un *failover*.

Estimez le nombre moyen de disques perdus par an, de redémarrages de serveur par an/mois/jour, de pannes de réseau par mois/jour.

Correction

Sans réplication, j'ai plus de 500 disques de 1 TO. J'en ai donc en moyenne 20 disques par an qui vont se planter. Avec réplication 2, cela donne 40 disques par an. Je peux recourir à des disques de 2 ou 3 TO. J'aurai moins de pannes, mais plus de travail pour la reprise (qui consiste à créer une nouvelle réplication).

Sans réplication, j'ai en moyenne 170*3=510 redémarrage serveur par an, soit nettement plus d'un par jour. Avec réplication 2, j'ai trois redémarrages de serveur par jour.

Pour le réseau : considérons que j'ai 4 baies (40 serveurs par baie), sans réplication. J'ai 12 coupures réseau par mois, 2 fois plus avec une réplication 2.

Conclusion : tous les jours j'ai des pannes de plusieurs types : disque défaillant, coupure réseau, serveur qui redémarre. Il me faut une politique de *failover* robuste.

Exercice Ex-S1-3 : quelques calculs

Je dois exécuter mon traitement de recherche de doublons dans les vidéos, décrit dans la section sur la scalabilité. J'applique une fonction d'extraction de signature, f, et pour l'instant je ne considère que la première étape.

- Supposons pour commencer que le coût du traitement par f soit négligeable par rapport à la lecture des vidéos sur les disques. Combien de temps faut-il (dans le meilleur des cas) pour avoir parcouru toutes les vidéos dans votre *cloud* (reprenez la configuration choisie précédemment).
- Ma fonction f prend 1s pour chaque MO ; combien de temps faudra-t-il pour avoir testé toutes mes vidéos ?
- Jusqu'où faut-il que j'optimise ma fonction f pour que l'accès au disque redevienne prépondérant ?
- J'arrive à optimiser mon traitement : 2 millisecondes par MO. Quelle solution me reste-t-il ?

Correction

— Parcourir un disque de 1 TO, ça prend plus de deux heures. Si on a un processeur multi-core, on peut lire les disques d'un même serveur en parallèle.
— J'ai 1 000 000 MOs par disque à traiter .... Ca fait 11 jours. Bienvenu dans le Big Data
— Il faut traiter 100 MO/s, donc 100 KO par milliseconde, 1 MO par centièmes/sec.
— Doubler le nombre de serveurs, utiliser 500 GOs sur chaque serveur. On peut combiner ça avec la réplication.

Exercice Ex-S1-4 : et avec le réseau ?

Maintenant je considère la deuxième étape, et je suppose qu'une paire (id, signature) occupe en moyenne 100 octets. Calculer le temps de transfert, en supposant (1) que toutes les machines sont dans une même baie, avec un débit de 1 Gbits/s et (2) qu'elles sont dans des baies distinctes, avec un débit moyen de 200 Mbits/s.

Correction

Il faut donc transférer 1 million de paires (id, signature), soit 100 millions d'octets, un petit peu moins d'un Gigabit. Le temps de tranfert va de moins d'une seconde dans une baie, à quelques secondes en cas de baies distinctes.

Notez bien que les *mappers* envoient directement au *reducers*, sans passer par le maître. La modicité du temps de transfert par le réseau est due au fait que les paires intermédiaires sont petites, beaucoup plus petites que les documents initiaux. C'est la clé d'une bonne performance, et en tout cas de la scalabilité des traitements de type MapReduce.

Exercice Ex-mapred-1 : un `grep`, en MapReduce

On veut scanner des millards de fichiers et afficher tous ceux qui contiennent une chaîne de caractères c. Donnez la solution en MapReduce, en utilisant le formalisme de votre choix (de préférence un pseudo-code un peu structuré quand même).

Correction

Pas bien compliqué. Map : on charge les fichiers un par un (ou ligne par ligne s'ils sont vraiment très gros), on cherche c et si on la trouve on émet le nom du fichier et un indicateur quelconque. Soit, en code :

```
// Le pattern est supposé connu dans le contexte d'exécution
function MapGrep (fichier) {
  if (contains(fichier, pattern) {
       emit(fichier.nom, 1);
}
```

Reduce : on émet le nom du fichier, éventellement avec le nombre d'occurrences trouvées de c. Par exemple :

```
function ReduceGrep (nomFichier, [nb]) {
  return (nomFichier, sum(nb));
}
```

Exercice Ex-mapred-2 : un `rollup`, en MapReduce

Une grande surface enregistre tous ses tickets de caisse, indiquant les produits vendus, le prix et la date, ainsi que le client si ce dernier a une carte de fidélité.

Les produits sont classés selon une taxonomie comme illustré sur la Fig. 11.14, avec des niveaux de précision. Pour chaque produit on sait à quelle catégorie précise de N1 il appartient (par exemple, `chaussure`) ; pour chaque catégorie on connaît son parent.

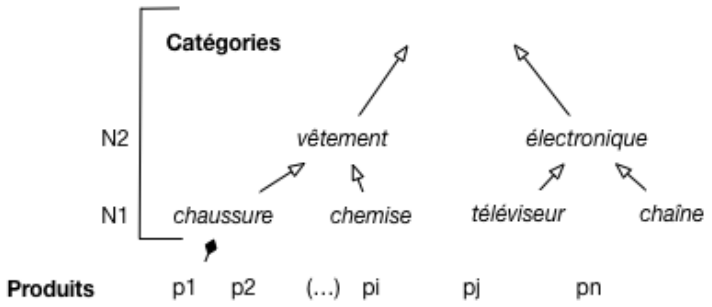


Fig. 11.14 – Les produits et leur classement.

Supposons que la collection `Tickets` contienne des documents de la forme (idTicket, idClient, idProduit, catégorie, date, prix). Comment obtenir en MapReduce le total des ventes à une date *d*, pour le niveau N2 ? On fait donc une agrégation de `Tickets` au niveau supérieur de la taxonomie.

Correction

Il nous faut donc une fonction *parent(n)* qui renvoie le parent d'un nœud *n* dans la taxonomie. On pourrait généraliser avec une fonction *ancêtre (n, niveau)* qui renvoie l'ancêtre à un niveau donné.

Cela acquis, il suffit d'appliquer la fonction *parent(n)* à chaque ticket pour pouvoir faire le regroupement voulu. La clé de regroupement est une paire constituée d'une date et du parent. Soit :

```
function MapRollup (ticket) {
       emit({ticket.date, parent(ticket.categorie)}, ticket.prix);
}
```

Reduce : direct.

```
function ReduceRollup ({date, categ}, [prix]) {
  return ({date, categ}, sum(prix));
}
```

Exercice Ex-mapred-3 : algèbre linéaire distribuée

Nous disposons du calcul d'algèbre linéaire du chapitre *Traitements par lot*. On a donc une matrice *M* de dimension $N \times N$ représentant les liens entres les N pages du Web, chaque lien étant qualifié par un facteur d'importance (ou « poids »). La matrice est représentée par une collection math :*C* dans laquelle chaque document est de la forme {« id » : &23, « lig » : *i*, « col » : *j*, « poids » : m_{ij}}, et représente un lien entre la page P_i et la page P_j de poids m_{ij}

Vous avez déjà vu le calcul de la norme des lignes de la matrice, et celui du produit de la matrice par un vecteur V. Prenons en compte maintenant la taille et la distribution.

Questions

- On estime qu'il y a environ $N = 10^{10}$ pages sur le Web, avec 15 liens par page en moyenne. Quelle est la taille de la collection C, en TO, en supposant que chaque document a une taille de 16 octets
- Nos serveurs ont 2 disques de 1 TO chacun et chaque document est répliqué 2 fois (donc trois versions en tout). Combien affectez-vous de serveurs au système de stokage ?
- Maintenant, on suppose que V ne tient plus dans la mémoire RAM d'une seule machine. Proposez une méthode de partitionnement de la collection C et de V qui permette d'effectuer le calcul distribué de $M \times V$ avec MapReduce sans jamais avoir à lire le vecteur sur le disque. Donnez le critère de partitionnement et la technique (par intervalle ou par hachage).
- Supposons qu'on puisse stocker *au plus* deux (2) coordonnées d'un vecteur dans la mémoire d'un serveur. Inspirez-vous de la Fig. 11.13 pour montrer le déroulement du traitement distribué précédent en choisissant le nombre minimal de serveurs permettant de conserver le vecteur en mémoire RAM.
  Pour illustrer le calcul, prenez la matrice 4×4 ci-dessous, et le vecteur $V = [4, 3, 2, 1]$.

$$M = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 7 & 6 & 5 & 4 \\ 6 & 7 & 8 & 9 \\ 3 & 3 & 3 & 3 \end{bmatrix}$$

- Expliquez pour finir comment calculer la similarité cosinus entre V et les lignes L_i de la matrice.

Correction

Il y a donc $N = 150 \times 10^9$ liens à placer dans la matrice, chaque lien étant représenté par un document de 16 octets. Soit $N = 2400 \times 10^9$ octets, ou 2,4 TO.

Avec trois copies de chaque lien, on arrive à 7,2 TO. Il faut donc au moins 4 serveurs pour pouvoir stocker la matrice (répliquée) sur disque.

Le vecteur V ne tient plus en mémoire RAM pour une seule machine. Il faut donc le découper en f fragments de manière à ce que chaque fragment contenant $\frac{N}{f}$ coordonnées tienne en RAM (on suppose qu'on a assez de machines, ce qui semble raisonnable). On a donc les fragments $V_1[0, \frac{N}{f}[$, $V_2[\frac{N}{f}, 2 \times \frac{N}{f}[$, etc.

Chaque fonction de MAP accède à l'un des fragments $V_i[(i-1) \times \frac{N}{f}, i \times \frac{N}{f}]$ du vecteur. Cette fonction peut donc se contenter d'accéder à la partie de la matrice qui doit être combinée à ce fragment : il s'agit évidemment des colonnes $[(i-1) \times \frac{N}{f}, i \times \frac{N}{f}[$. On va donc partitionner la matrice en f fragments, *verticalement*.

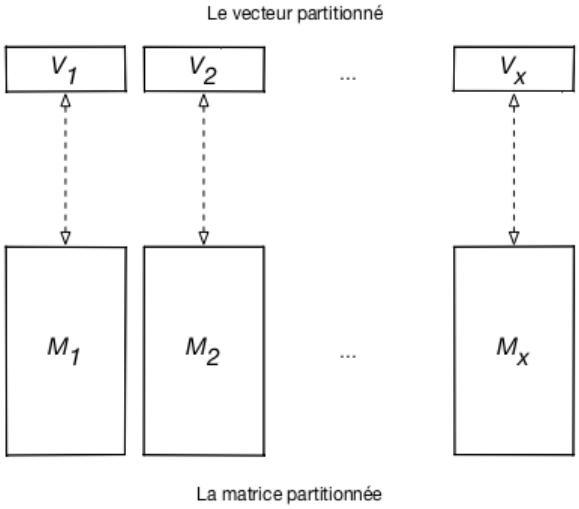


Fig. 11.15 – Partition verticale de la matrice

La Fig. 11.15 montre le partitionnement du vecteur et celui de la matrice. Le calcul MapReduce est alors *exactement* le même que celui déjà vu pour un calcul dans le cas où le vecteur tient en mémoire. La seule différence est qu'il s'applique aux fragments à apparier de la matrice et du vecteur.

La Fig. 11.16 illustre le calcul distribué avec un partitionnement minimal en deux fragments. Pour simplifier un peu la figure, on montre le calcul au niveau des lignes et pas des cellules élémentaires. Un premier serveur multiplie les deux premières colonnes de la matrice avec les deux premières coordonnées du vecteur. Le second serveur effectue le calcul complémentaire.

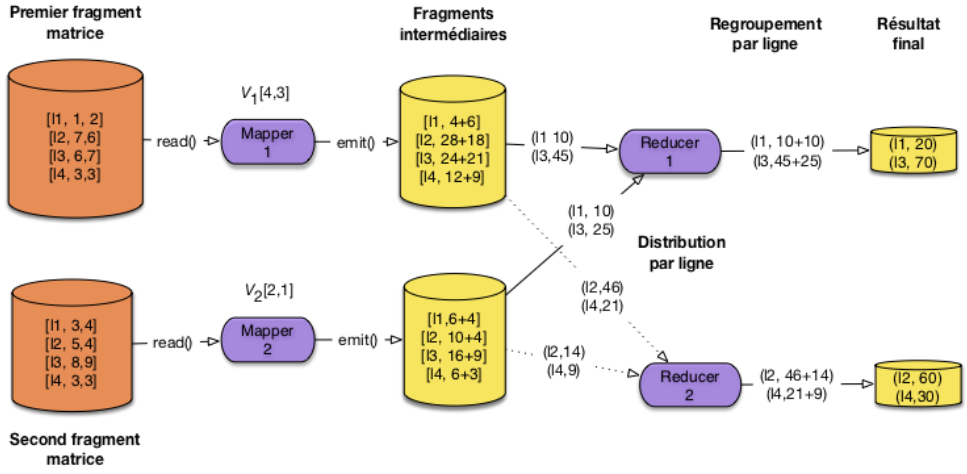


Fig. 11.16 – Illustration du calcul distribué

On a utilisé deux *reducers*, avec une distribution (*shuffle*) qui envoie tous les résultats intermédiaires des lignes 1 et 3 vers le premier *reducer*, et ceux des lignes 2 et 4 vers le second.

Pour finir, le calcul de la similarité cosinus est obtenu en combinant un premier calcul des normes des vecteurs, suivi du produit avec le vecteur du document-cible.

CHAPITRE 12

Systèmes NoSQL : la réplication

La réplication (des données) est une caractéristique commune aux systèmes NoSQL. Rappelons que ces systèmes s'exécutent dans un environnement sujet à des pannes fréquentes et répétées. Il est donc indispensable, pour assurer la sécurité des données, de les répliquer autant de fois que nécessaire pour disposer d'une solution de secours en cas de perte d'une machine.

Ce chapitre est entièrement consacré à la réplication, avec illustration pratique basée sur MongoDB, ElasticSearch et Cassandra.

12.1 S1 : réplication et reprise sur panne

Supports complémentaires

— Diapositives: réplication et reprise sur panne dans les systèmes NoSQL
— Vidéo sur les principes de réplication et de reprise sur panne

12.1.1 La réplication, pourquoi

Bien entendu, on pourrait penser à la solution traditionnelle consistant à effectuer des sauvegardes régulières, et on peut considérer la réplication comme une sorte de sauvegarde continue. Les systèmes NoSQL vont nettement plus loin, et utilisent la réplication pour atteindre plusieurs objectifs.

— **Disponibilité**. La réplication permet d'assurer la disponibilité constante du système. En cas de panne d'un serveur, d'un nœud ou d'un disque, la tâche effectuée par le composant défectueux peut être *immédiatement* prise en charge par un autre composant. Cette technique de reprise sur panne immédiate et automatique (*failover*) est un atout essentiel pour assurer la stabilité d'un système pouvant comprendre des milliers de nœuds, sans avoir à engloutir un budget monstrueux dans la surveillance et la maintenance.

— **Scalabilité (lecture)**. Si une donnée est disponible sur plusieurs machines, il devient possible de *distribuer* les requêtes (en lecture) sur ces machines. C'est le scénario typique pour la scalabilité des applications Web par exemple (voir le système memCached conçu spécifiquement pour les applications web dynamiques).
— **Scalabilité (écriture)**. Enfin, on peut penser à distribuer aussi les requêtes en écriture, mais là on se retrouve face à de délicats problèmes potentiels d'écritures concurrentes et de réconciliation.

Le niveau de réplication dépend notamment du budget qu'on est prêt à allouer à la sécurité des données. On peut considérer que 3 copies constituent un bon niveau de sécurité. Une stratégie possible est par exemple de placer un document sur un serveur dans une baie, une copie dans une autre baie pour qu'elle reste accessible en cas de coupure réseau, et une troisième dans un autre centre de données pour assurer la survie d'au moins une copie en cas d'accident grave (incendie, tremblement de terre). À défaut d'une solution aussi complète, deux copies constituent déjà une bonne protection. Il est bien clair que dès que la perte de l'une des copies est constatée, une nouvelle réplication doit être mise en route.

Note : Un peu de vocabulaire. On va parler
— de *copie* ou de *réplica* pour désigner la duplication d'un *même* document ;
— de *version* pour désigner les valeurs successives que prend un document au cours du temps.

Voyons maintenant comment s'effectue la réplication. L'application client, *C* demande au système l'écriture d'un document *d*. Cela signifie qu'il existe un des nœuds du système, disons N_m, qui constitue l'interlocuteur de *C*. N_m est typiquement le Maître dans une architecture Maître-Esclave, mais ce point est revu plus loin.

N_m va identifier, au sein du système, un nœud responsable du stockage de *d*, disons N_x. Le processus de réplication fonctionne alors comme suit :
— N_x écrit *localement* le document *d* ;
— N_x transmet la demande d'écriture à un ou plusieurs autres serveurs, N_y, N_z, qui a leur tour effectuent l'écriture.
— N_x, N_y, N_z renvoient un acquittement à N_m confirmant l'écriture.
— N_m renvoie un acquittement au client pour lui confirmer que *d* a bien été enregistré.

Note : Il existe bien sûr des variantes. Par exemple, N_m peut se charger de distribuer les trois requêtes d'écriture, au lieu de créer une chaîne de réplication. Ca ne change pas fondamentalement la problématique.

Dans un scénario standard (celui d'une base relationnelle par exemple), l'acquittement n'est donné au client que quand la donnée est *vraiment* enregistrée de manière permanente. Entre la demande d'écriture et la réception de l'acquittement, le client attend.

Rappelons que dans un contexte de persistance des données, une écriture est permanente quand la donnée est placée sur le disque. C'est ce que fait un SGBD quand on demande la validation par un `commit`. Une donnée placée en mémoire RAM sans être sur le disque n'est pas totalement sûre : en cas de panne elle disparaîtra et l'engagement de durabilité du `commit` ne sera pas respecté. Bien entendu, écrire sur le disque prend beaucoup plus de temps (quelques ms, au pire), ce qui bloque d'autant l'application client, prix à payer pour la sécurité.

Dans ces conditions, le fait d'effectuer des copies sur d'autres serveurs allonge encore le temps d'attente de l'application. Si on fait *n* copies, en écrivant à chaque fois sur le disque par sécurité, le temps d'attente du client à chaque écriture va dériver vers le dixième de seconde, ce que ne passe pas à l'échelle de mise à jour intensives.

12.1.2 La réplication, comment

Deux techniques sont utilisées pour limiter le temps d'attente, toutes deux affectant (un peu) la sécurité des opérations :

1. **écriture en mémoire RAM**, et fichier journal (*log*) ;
2. **réplication asynchrone**.

La première technique est très classique et utilisée par tous les SGBD du monde. Au lieu d'effectuer des écritures répétées sur le disque sans ordre pré-défini (accès dits « aléatoires ») qui imposent à chaque fois un déplacement de la tête de lecture et donc une latence de quelques millisecondes, on écrit *séquentiellement* dans un fichier de journalisation (*log*) et on place également la donnée en mémoire RAM (Fig. 12.1, A et B).

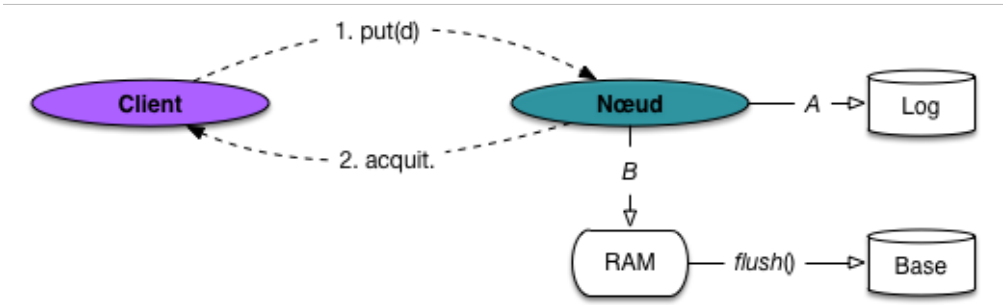


Fig. 12.1 – Ecriture avec journalisation

À terme, le contenu de la mémoire RAM, marqué comme contenant des données modifiées, sera écrit sur le disque dans les fichiers de la base de données (opération de *flush()*). La séquence est illustrée par la Fig. 12.1.

Cela permet de grouper les opérations d'écritures et donc de revenir à des entrées/sorties *séquentielles* sur le disque, aussi bien dans le fichier journal que dans la base principale.

Que se passe-t-il en cas de panne *avant* l'opération de *flush()* ? Dans ce cas les données modifiées n'ont pas été écrites dans la base, mais le journal (*log*) est, lui, préservé. La *reprise sur panne* consiste à ré-effectuer les opérations enregistrées dans le *log*.

Note : Cette description est très brève et laisse de côté beaucoup de détails importants (notez par exemple que le fichier *log* et la base devraient être sur des disques distincts). Reportez-vous à http://sys.bdpedia.fr pour en savoir plus.

Le scénario d'une réplication synchrone (avec deux copies) est alors illustré par la Fig. 12.2. Le nœud-coordinateur N_m distribue l'ensemble des demandes d'écritures à tous les nœuds participants *et attend leur acquittement* pour acquitter lui-même le client. On se condamne donc à être dépendant du nœud le plus lent à répondre. D'un autre côté le client qui reçoit l'acquittement est certain que les trois copies de *d* sont effectivement enregistrées de manière durable dans le système.

La seconde technique pour limiter le temps d'écriture est le recours à des écritures asynchrones. Contrairement au scénario de la Fig. 12.2, le serveur N_m va acquitter le client dès que l'un des participants a répondu (sur la figure Fig. 12.3 c'est le nœud N_y). Le client peut alors poursuivre son exécution.

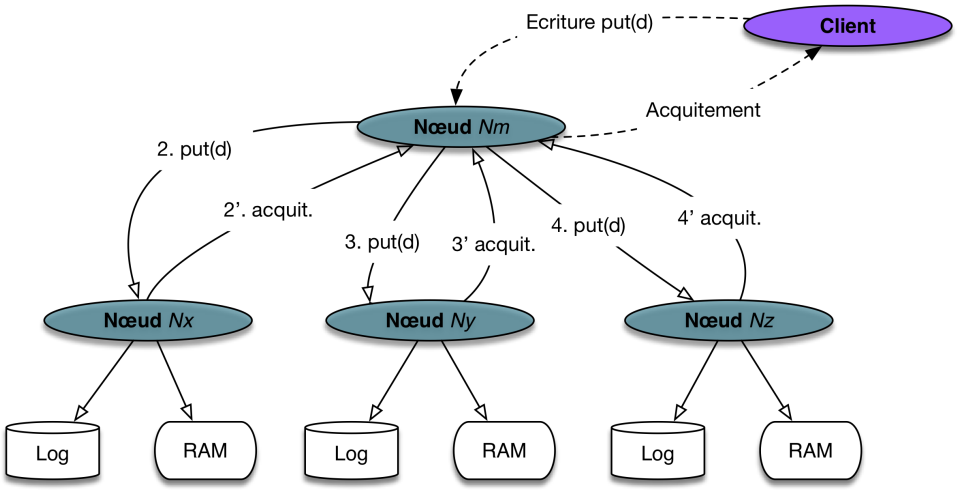


Fig. 12.2 – Réplication (avec écritures synchrones)

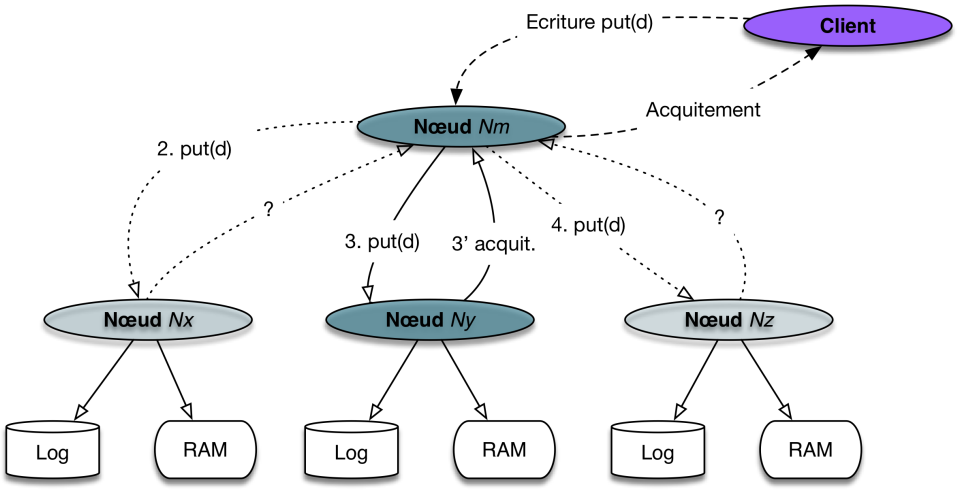


Fig. 12.3 – Réplication avec écritures asynchrones

Dans ce scénario, beaucoup plus rapide pour le client, deux phénomènes apparaissent :

— *le client reçoit un acquittement alors que la réplication n'est pas complète* ; il n'y a donc pas à ce stade de garantie complète de sécurité ;

— *le client poursuit son exécution alors que toutes les copies de d ne sont pas encore mises à jour* ; il se peut alors qu'une lecture renvoie une des versions antérieures de *d*.

Il y a donc un risque pour la cohérence des données. C'est un problème sérieux, caractéristique des systèmes distribués en général, du NoSQL en particulier.

Entre ces deux extrêmes, un système peut proposer un paramétrage permettant de régler l'équilibre entre sécurité (écritures synchrones) et rapidité (écritures asynchrones). Si on crée trois copies d'un document, on peut par exemple décider qu'un acquittement sera envoyé quand deux copies sont sur disque, pendant que la troisième s'effectue en mode asynchrone.

12.1.3 Cohérence des données

La cohérence est la capacité d'un système de gestion de données à refléter fidèlement les opérations d'une application. Un système est cohérent si toute opération (validée) est immédiatement visible et permanente. Si je fais une écriture de *d* suivie d'une lecture, je dois constater les modifications effectuées ; si je refais une lecture un peu plus tard, ces modifications doivent toujours être présentes.

La cohérence dans les systèmes répartis (NoSQL) dépend de deux facteurs : la topologie du système (maître-esclave ou multi-nœuds) et le caractère asynchrone ou non des écritures. Trois combinaisons sont possibles en pratique, illustrées par la Fig. 12.4.

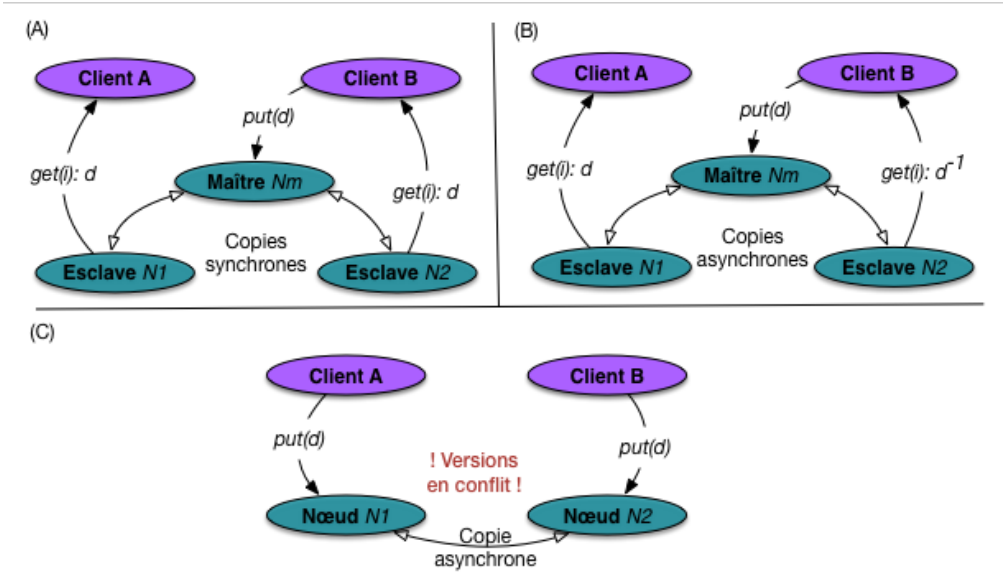


Fig. 12.4 – Réplication et cohérence des données

Premier cas (A, en haut à gauche) : la topologie est de type maître-esclave, et les écritures synchrones. Toutes les écritures se font par une requête adressée au nœud-maître qui se charge de les distribuer aux nœuds-esclaves. L'acquittement n'est envoyé au client que quand *toutes* les copies sont en phase.

Ce cas assure la cohérence *forte*, car toute lecture du document, quel que soit le nœud sur lequel elle est

effectuée, renvoie la même version, celle qui vient d'être mise à jour. Cette cohérence se fait au prix de l'attente que la synchronisation soit complète, et ce à chaque écriture.

Dans le second cas (B), la topologie est toujours de type maître-esclave, mais les écritures sont asynchrones. La cohérence n'est plus forte : il est possible d'écrire en s'adressant au nœud-maître, et de lire sur un nœud-esclave. Si la lecture s'effectue *avant* la synchronisation, il est possible que la version du document retournée soit non pas *d* mais d^{-1}, celle qui précède la mise à jour.

L'application client est alors confrontée à la situation, rare mais pertubante, d'une écriture sans effet apparent, au moins immédiat. *C'est le mode d'opération le plus courant des systèmes NoSQL, qui autorisent donc un décalage potentiel entre l'écriture et la lecture*. La garantie est cependant apportée que ce décalage est temporaire et que toutes les versions vont être synchronisées « à terme » (délai non précisé). On parle donc de *cohérence à terme* (*eventual consistency* en anglais).

Enfin, le dernier cas, (C), correspond à une topologie multi-nœuds, en mode asynchrone. Les écritures peuvent se faire sur n'importe quel nœud, ce qui améliore la scalabilité du système. L'inconvénient est que deux écritures concurrentes du même document peuvent s'effectuer en parallèle sur deux nœuds distincts. Au moment où la synchronisation s'effectue, le système va découvrir (au mieux) que les deux versions sont en conflit. Le conflit est reporté à l'application qui doit effectuer une *réconciliation* (il n'existe pas de mode automatique de réconciliation).

Note : La dernière combinaison envisageable, entre une topologie multi-nœuds et des écritures synchrones, mène à des inter-blocages et n'est pas praticable.

En résumé, trois niveaux de cohérence peuvent se recontrer dans les systèmes NoSQL :

1. **cohérence forte** : toutes les copies sont toujours en phase , le prix à payer étant un délai pour chaque écriture ;
2. **cohérence faible** : les copies ne sont pas forcément en phase, et rien ne garantit qu'elles le seront ; cette situation, trop problématique, a été abandonnée (à ma connaissance) ;
3. **cohérence à terme** : c'est le niveau de cohérence typique des systèmes NoSQL : les copies ne sont pas immédiatement en phase, mais le système garantit qu'elles le seront « à terme ».

Dans la cohérence à terme, il existe un risque, faible mais réel, de constater de temps en temps un décalage entre une écriture et une lecture. Il s'agit d'un « marqueur » typique des systèmes NoSQL par rapport aux systèmes relationnels, résultant d'un choix de conception privilégiant l'efficacité à la fiabilité stricte (voir aussi le théorème CAP, plus loin).

La tendance.

L'enthousiasme initial soulevé par les systèmes NoSQL a été refroidi par les problèmes de cohérence qu'ils permettent, et ce en comparaison de la garantie ACID apportée par les systèmes relationnels. On constate une tendance de ces systèmes à proposer des mécanismes transactionnels plus sûrs. Voir l'exercice sur les transactions distribubées en fin de chapitre.

12.1.4 Equilibrage entre cohérence et latence : le principe du quorum

De ce que nous avons vu jusqu'à présent, le choix apparaît binaire entre la *cohérence* obtenue par des écritures synchrones, et la réduction de la latence (temps d'attente) obtenue par des écritures asynchrones.

De nombreux systèmes proposent un paramétrage plus fin qui s'appuie sur trois paramètres
- *W* le nombre *d'écritures* synchrones avant acquittement au client
- *R* le nombre de *lectures* synchrones avant acquittement au client par renvoi de la copie la plus récente
- *RF* le facteur de réplication.

Les deux paramètres *W* et *R* ont une valeur comprise en 1 et *RF*. Une valeur élevée de *W* implique des écritures plus lentes mais plus sûres. Une valeur élevée de *R* implique des lectures plus lentes mais plus cohérentes.

Examinons la Fig. 12.5. Elle illustre les valeurs de paramètres *W=2*, *R=3* et *RF=4*. Les flêches représentent les opérations *synchrones*.

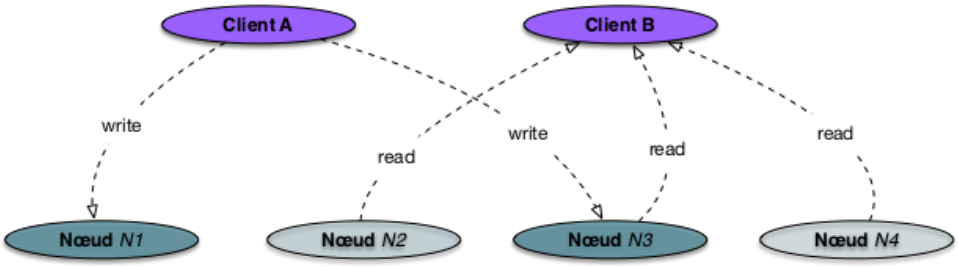


Fig. 12.5 – Paramétrage avec *W=2*, *R=3* et *RF=4*

Le client A effectue une écriture d'un document. La Fig. 12.5 montre que deux écritures synchrones ont été effectuées, sur les nœuds N_1 et N_3 (en vert foncé). Les deux autres copies, sur N_2 et N_4 sont en attente des écritures asynchrones et ne sont donc pas en phase.

Le client B effectue une lecture du même document. Comme *R=3*, il doit lire au moins 3 des 4 copies avant de répondre au client. En lisant (par exemple) sur N_2, N_3 et N_4, il va trouver la version la plus récente sur N_3 et on obtient donc une cohérence forte.

On peut établir la formule suivante qui guarantit la cohérence forte :

Critère de cohérence forte

La cohérence forte est assurée si $R + W > RF$.

L'intuition est qu'il existe dans ce cas un recouvrement entre les réplicas lus et les derniers réplicas écrits, de sorte qu'au moins une lecture va accéder à la dernière version. C'est ce qui est illustré par la Fig. 12.5, mais plusieurs autres situations sont possibles. En supposant *RF=4* :
- si *W=4* et *R=1* : on se satisfait d'une lecture, mais comme tous les écritures sont synchronisées, on est sûr qu'elle renvoie la dernière mise à jour.
- si *W=1* et *R=4*, le raisonnement réciproque amène à la même conclusion
- si *W=3* et *R=2*, on équilibre un peu mieux la latence entre écritures et lectures. En lisant 2 copies sur les 4 existantes, dont 3 sont synschrones, on est sûr d'obtenir la dernière version.

Le *quorum* est la valeur $\lfloor \frac{RF}{2} \rfloor + 1$, où $\lfloor x \rfloor$ désigne l'entier immédiatement inférieur à une valeur x. Le quorum est par exemple 3 pour *RF=4* ou *RF=5*. En fixant dans un système *W=QUORUM* et *R=QUORUM*, on est sûr d'être cohérent : c'est la configuration la plus flexible, car s'adaptant automatiquement au niveau de réplication.

12.1.5 Réplication et reprise sur panne

Voyons maintenant comment la réplication permet la reprise sur panne. Nous allons considérer la topologie maître-esclave, la plus courante. La situation de départ est illustrée par la Fig. 12.6. Tous les nœuds sont interconnectés et se surveillent les uns les autres par envoi périodique de courts messages dits *heartbeats*.

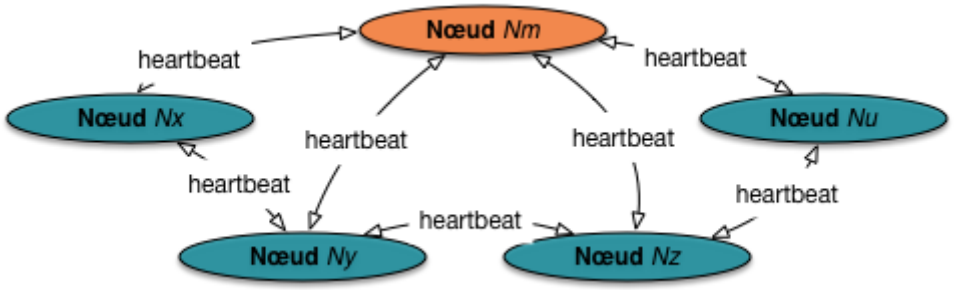


Fig. 12.6 – Surveillance par *heartbeat* dans un système distribué

Si l'un des nœuds-esclaves disparaît, la parade est assez simple : le nœud-maître va rediriger les requêtes des applications clientes vers les nœuds contenant des copies, et initier une nouvelle réplication pour revenir à une situation où le nombre de copies est au niveau requis. Si, par exemple, le nœud N_x disparaît, le maître N_m sait que l'esclave N_y contient une copie des données disparues (en reprenant les exemples de réplication donnés précédemment) et redirige les lectures vers N_y. Les copies placées sur N_y doivent également être répliquées sur un nouveau serveur.

Si le nœud-maître disparaît, les nœuds-esclaves doivent élire un nouveau maître (!) pour que ce nouveau maître soit opérationnel, il faut probablement qu'il récupère des données administratives (configuration de la grappe de serveur) qui elles-mêmes ont dû être répliquées au préalable pour être toujours disponibles. Les détails peuvent varier d'un système à l'autre (nous verrons des exemples) mais le principe et là aussi de s'appuyer sur la réplication.

Tout cela fonctionne, sous réserve que la condition suivante soit respectée : *toute décision est prise par une sous-grappe comprenant la majorité des participants*. Considérons le cas de la Fig. 12.7. Un partitionnement du réseau a séparé la grappe en deux sous-ensembles. Si on applique la méthode de reprise décrite précédemment, que va-t-il se passer ?

— le maître survivant va essayer de se débrouiller avec l'unique esclave qui lui reste ;
— les trois esclaves isolés vont élire un nouveau maître.

On risque de se retrouver avec deux grappes agissant indépendamment, et une situation à peu près ingérable (divergence des données, perturbation des applications clientes, etc.)

La règle (couramment établie dans les systèmes distribués bien avant le NoSQL) est *qu'un maître doit toujours régner sur la majorité des participants*. Pour élire un nouveau maître, un sous-groupe de nœuds doit donc atteindre le quorum de $\frac{n}{2} + 1$, où n est le nombre initial de nœuds. Dans l'exemple de la Fig. 12.7, le

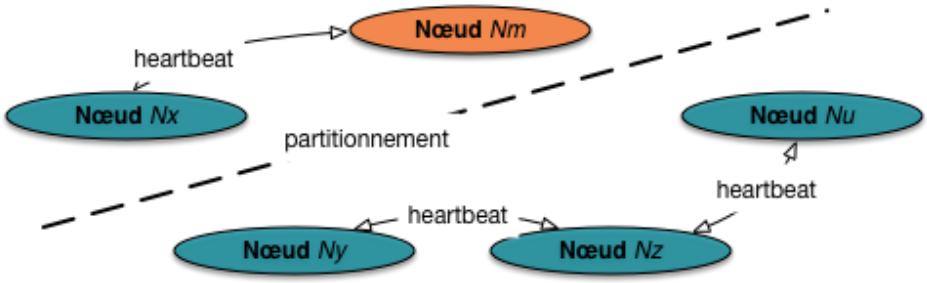


Fig. 12.7 – Un partitionnement réseau.

maître existant est rétrogradé au rang d'esclave, et un nouveau maître est élu dans le second sous-groupe, le seul qui continuera donc à fonctionner.

L'algorithme pour l'élection d'un maître relève des méthodes classiques en systèmes distribués. Voir par exemple l'algorithme Paxos.

12.1.6 Culture : le théorème CAP

Le moment est venu de citer une propriété (ou une incomplétude), énoncée par le *théorème CAP*. L'auteur (E. Brewer) ne l'a en fait pas présenté comme un « théorème » mais comme une simple conjecture, voire un constat pratique sans autre prétention. Il a ensuite pris le statut d'une vérité absolue. Examinons donc ce qu'il en est.

Le théorème CAP

Un système distribué orienté données ne peut satisfaire à chaque instant que deux des trois propriétés suivantes

- la cohérence (le « C ») : toute lecture d'une donnée accède à sa dernière version ;
- la disponibilité (le « A » pour *availability*) : toute requête reçoit une réponse, sans latence excessive ;
- la tolérance au partitionnement (le « P ») : le système continue de fonctionner même en cas de partionnement réseau.

Ce théorème est souvent cité sans trop réfléchir quand on parle des systèmes NoSQL. On peut le lire en effet comme une justification du choix de ces systèmes de privilégier la disponibilité et la tolérance au partitionnement (reprise sur panne), en sacrifiant partiellement la cohérence. Ce seraient donc des systèmes AP, alors que les systèmes relationnels seraient plutôt des systèmes CP. Cette interprétation un peu simpliste mérite qu'on aille voir un peu plus loin.

L'intuition derrière le théorème CAP est relativement simple : si une partition réseau intervient, il ne reste que deux choix possibles pour répondre à une requête : soit on répond avec les données (ou l'absence de données) dont on dispose, soit on met la requête en attente d'un rétablissement du réseau. Dans le premier cas on sacrifie la cohérence et on obtient un système de type AP, dans le second on sacrifie la disponibilité et on obtient un système de type CP.

Qu'en est-il alors de la troisième paire du triangle des propriétés, AC, « Cohérent et Disponible mais pas tolérant au Partitionnement » ? Cette troisième branche est en fait un peu problématique, car on ne sait pas ce que signifie précisément « tolérance au Partitionnement ». Comment peut-on rester cohérent et disponible avec une panne réseau ? Il faut bien sacrifier l'un des deux et on se retrouve donc devant un choix AP ou CP : le côté AC ressemble bien à une impasse.

Il existe donc une asymétrie dans l'interprétation du théorème CAP. On commence par une question : « *Existe-t-il un partitionnement réseau* ». Si oui on a le choix entre deux solutions : sacrifier la cohérence ou sacrifier la disponibilité.

Caractériser les systèmes NoSQL comme ceux qui feraient le choix de sacrifier la cohérence en cas de partitionnement apparaît alors comme réducteur. Si c'était le cas, en l'absence de partitionnement, ils auraient la cohérence *et* la disponibilité, or on s'aperçoit que ce n'est pas le cas : la cohérence est (partiellement) sacrifiée par le choix d'une stratégie de réplication asynchrone afin de favoriser un quatrième facteur, ignoré du théorème CAP : la latence (ou temps de réponse).

La Fig. 12.8 résume le raisonnement précédent. Il correspond au modèle PACELC (le "E" vient du "Else", représenté par un "Non" dans la figure) proposé dans cet article http://www.cs.umd.edu/~abadi/papers/abadi-pacelc.pdf que je vous encourage à lire attentivement.

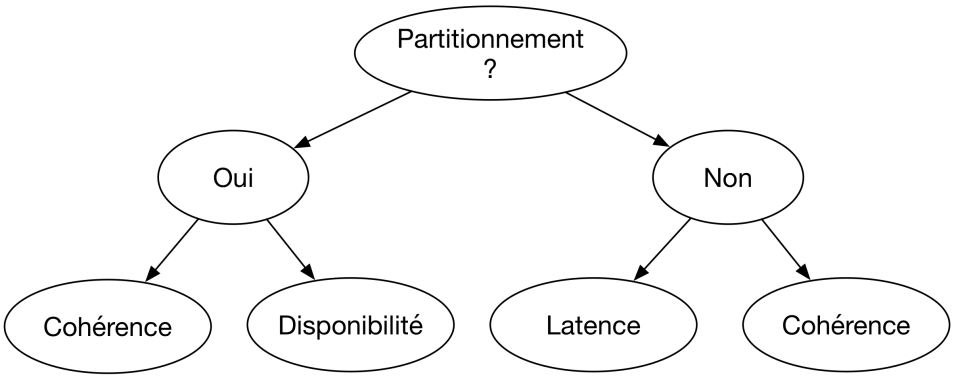


Fig. 12.8 – Le modèle PACELC, un CAP réorganisé et complété avec la latence

Tout cela nous ramène toujours au constat suivant : il faut toujours réfléchir indépendamment, lire attentivement les arguments et les peser, plutôt que d'avaler des slogans courts mais souvent mal compris. Qu'est-ce qu'un système NoSQL en prenant en compte ces considérations ? La définition devient un peu plus complexe : c'est un système de gestion de données distribuées qui doit être tolérant au partitionnement, mais qui même en l'absence de partitionnement peut accepter de compromettre la cohérence au profit de la latence.

Les trois systèmes que nous allons étudier dans ce qui suit gagneront à être interprétés dans cette optique explicative.

12.1.7 Quiz

12.2 S2 : ElasticSearch

Supports complémentaires
— Vidéo de démonstration d'une grappe ElasticSearch

ElasticSearch est un moteur de recherche distribué bâti sur les index Lucene, comme Solr. Contrairement à Solr, il a été conçu dès l'origine pour un déploiement dans une grappe de serveur et bénéficie en conséquence d'une facilité de configuration et d'installation incomparables.

12.2.1 Lancement du cluster

Vous avez déjà installé ElasticSearch avec Docker avec un unique serveur. Nous allons maintenant créer une grappe (un *cluster*) de plusieurs nœuds ElasticSearch et tester le comportement du système. Pour éviter de lancer beaucoup de commandes complexes, nous allons utiliser `docker-compose`, un utilitaire fourni avec le *Docker Desktop* qui permet de regrouper les commandes et les configurations. Nous vous fournissons plusieurs fichiers YAML de paramétrage correspondant aux essais successifs de configuration que nous allons effectuer. Il suffit de passer le nom du fichier à `docker-compose` de la manière suivante :

```
docker compose -f <nom-du-fichier.yml> up
```

Important : Il faut allouer au moins 4GB de mémoire RAM au *Docker Desktop* pour le système distribué que nous allons créer. Ouvrez votre interface *Docker Desktop* et indiquez bien 4GB pour l'option « Resources -> memory ».

Le premier fichier est dock-comp-es1.yml. Voici son contenu.

```
services:
  es01:
    image: docker.elastic.co/elasticsearch/elasticsearch:7.9.3
    container_name: es01
    environment:
      - node.name=es01
      - cluster.name=ma-grappe-es
      - discovery.seed_hosts=es01,es02
      - cluster.initial_master_nodes=es01,es02
    ports:
      - 9200:9200

  es02:
    image: docker.elastic.co/elasticsearch/elasticsearch:7.9.3
    container_name: es02
```

(suite sur la page suivante)

(suite de la page précédente)

```
    environment:
      - node.name=es02
      - cluster.name=ma-grappe-es
      - discovery.seed_hosts=es01,es02
    ports:
      - 9201:9200
```

On crée donc (pour commencer) deux nœuds Elastic Search, `es01` et `es02`. Ces deux nœuds sont placés dans une même grappe nommée `ma-grappe-es` (il va sans dire que les noms sont arbitraires et n'ont aucune signification propre). Le premier va être en écoute (pour les clients REST) sur le port 9200, le second sur le port 9201.

Le paramètre `discovery.seed_hosts` indique à chaque nœud les autres nœuds du *cluster* avec lesquels il doit se connecter et dialoguer. Au lancement, `es01` et `es02` vont donc pouvoir se connecter l'un à l'autre et échanger des informations.

Elastic Search fonctionne en mode Maître-esclave. Parmi les informations importantes se trouve la liste initiale des maîtres-candidats du *cluster*. Ici cette liste est donnée pour le nœud `es01` qui la communiquera ensuite à tous les autres nœuds. L'ensemble des nœuds ayant un statut de maître-candidat vont alors organiser une élection pour choisir le nœud-maitre du *cluster*, qui dans ElasticSearch se charge de la gestion de la grappe, et notamment de l'ajout/suppression de nouveaux nœuds et des reprises sur panne.

Dans le répertoire où vous avez placé ce fichier, exécutez la commande.

```
docker compose -f dock-comp-es1.yml up
```

L'utilitaire va créer les deux nœuds et les mettre en communication. Nous pouvons alors accéder à ce *cluster* avec ElasticVue.

12.2.2 Gérer un cluster avec ElasticVue

Pour une inspection confortable du serveur et des index ElasticSearch, nous allons reprendre ElasticVue. Il faut pour cela ajouter une connexion au *cluster* `ma-grappe-es` et indiquer au moins un serveur d'accès (par exemple http://localhost:9200). La Fig. 12.9 montre l'affichage ElasticVue.

En cliquant sur le menu *Nodes*, vous obtenez l'affiche de la Fig. 12.10. Regardez soigneusement les informations données, et cherchez par exemple quel est le maître. D'autres informations, comme *Master eligible*, *Data node*, *Ingest node* se comprennent aisément.

Remarquez qu'un nœud peut tenir plusieurs rôles (*master*, *data*, etc.). L'étude de ces rôles fait l'objet d'un exercice en fin de chapitre.

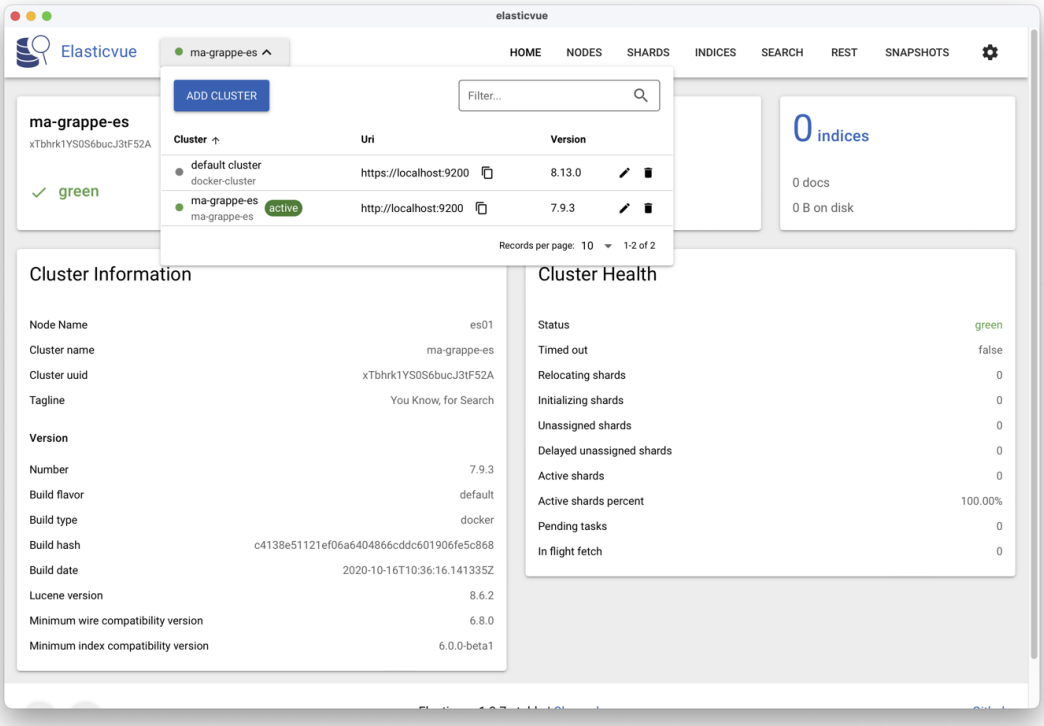


Fig. 12.9 – ElasticVue montrant les nœuds de notre grappe initiale ElasticSearch

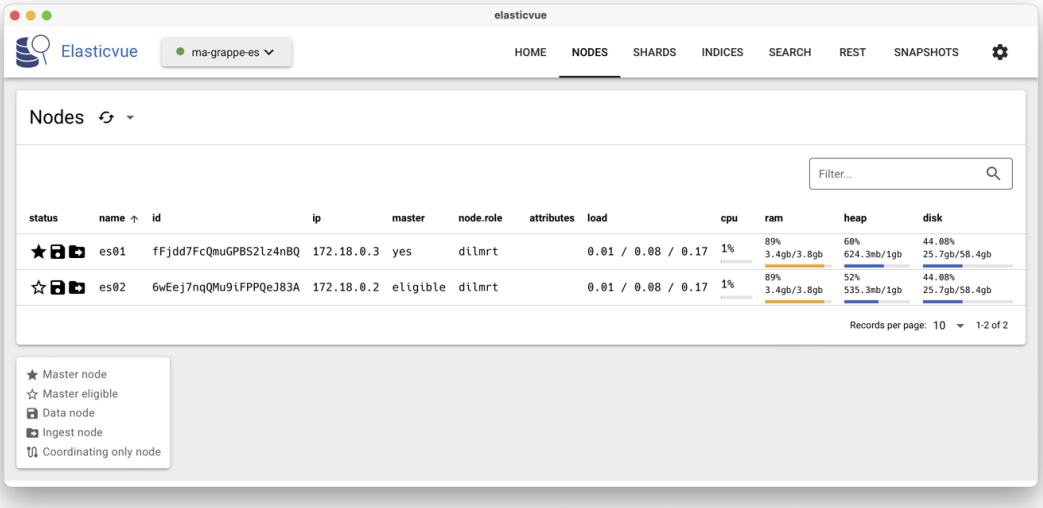


Fig. 12.10 – Evue montrant les nœuds de notre grappe initiale ElasticSearch

12.2.3 Le jeu de données

Pour charger des données, récupérez notre collection de films, au format JSON adapté à l'insertion en masse dans ElasticSearch, sur le site https://deptfod.cnam.fr/bd/tp/datasets/. Le fichier se nomme `films_esearch.json`. Ensuite, importez les documents dans le *cluster* en envoyant un `POST` à l'URL `_bulk` et le contenu du fichier, comme le montre la Fig. 12.11.

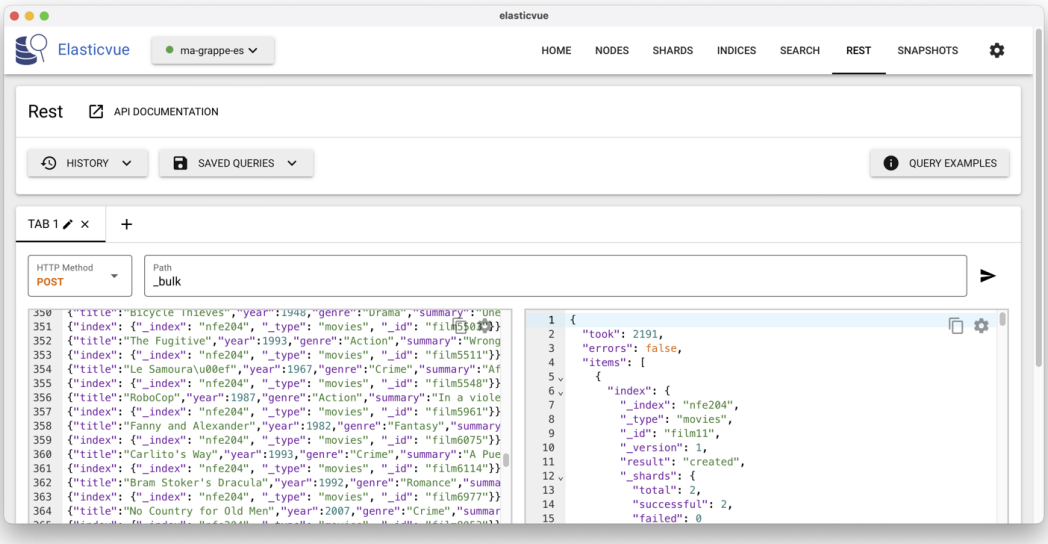


Fig. 12.11 – Insertion dans le cluster avec ElasticVue

En accédant au menu *Shards* dans ElasticVue, vous devriez alors obtenir l'affichage de la Fig. 12.12.

Que constate-t-on ? Les nœuds `es01` `es02` apparaissent associés à des rectangles verts qui représentent l'index ElasticSearch, nommé `nfe204`, stockant les quelques centaines de films de notre jeu de données. Remarquez que l'index apparaît en vert.

Un des rectangles est entouré d'un trait plein : c'est la *copie primaire de l'index*, celle sur laquelle s'effectuent les *écritures*, qui sont ensuite répliquées en mode asynchrone sur les autres copies.

Il existe une distinction dans ElasticSearch entre la notion de *master*, désignant le nœud responsable de la gestion du *cluster*, et la notion de *copie primaire* qui désigne le nœud stockant la copie sur laquelle s'effectuent en priorité les écritures. La copie primaire peut être sur un autre nœud que le *master*. Un système de *routage* permet de diriger une requête d'écriture vers la copie primaire, quel que soit le nœud de la grappe auquel la requête est adressée. Voir l'exercice en fin de chapitre sur ces notions.

En revanche, dans Elastic Search, chaque nœud peut répondre aux requêtes de *lectures*. Il est donc possible qu'une écriture ait lieu sur la copie primaire, puis une lecture sur la copie secondaire, avant réplication, donnant donc un résultat obsolète, ou *incohérent*. Dès que la réplication est achevée, la cohérence est rétablie. Dans un moteur de recherche, la cohérence à terme est considérée comme tout à fait acceptable.

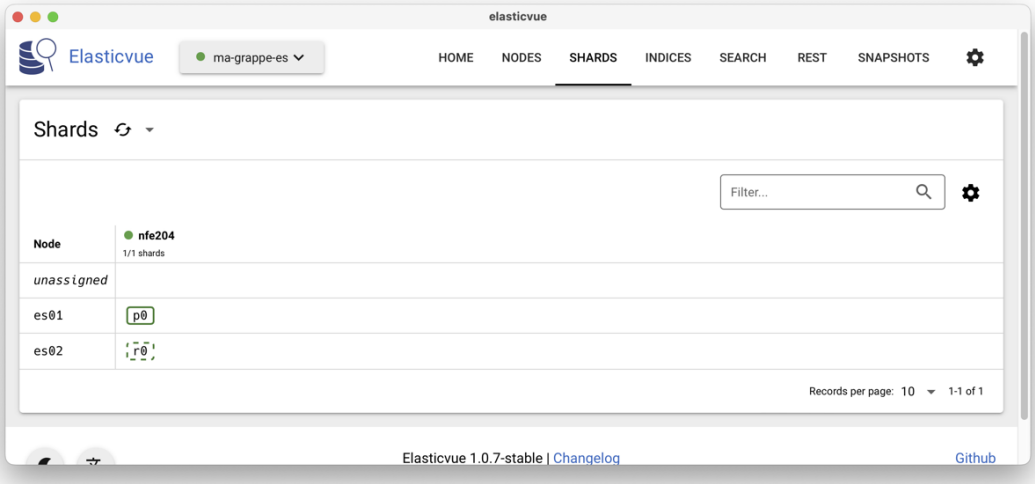


Fig. 12.12 – ElasticVue montrant la réplication

12.2.4 Changeons la réplication

Commençons quelques manipulations de notre index `nfe204`. Par défaut, ElasticSearch effectue une réplication de chaque document. Nous souhaitons en faire deux pour avoir trois copies au total, ce qui est considéré comme une sécurité suffisante. Pour modifier ce paramètre, envoyez en méthode `PUT` à l'URL `_settings` le document de paramétrage suivant :

```
{ "number_of_replicas": 2 }
```

La Fig. 12.13 montre ce que vous devez obtenir en réaffichant le menu `Shards`. L'index a trois copies, mais seulement deux nœuds. Un signe d'avertissement (le nom de l'index est en orange !) est apparu indiquant que l'une des copies manque d'un nœud pour être hébergée.

Il faut donc ajouter un nœud. Dans le fichier de configuration, on ajoute `es3` comme suit :

```
es03:
  image: docker.elastic.co/elasticsearch/elasticsearch:7.9.3
  container_name: es03
  environment:
    - node.name=es03
    - cluster.name=ma-grappe-es
    - discovery.seed_hosts=es01,es02
  ports:
    - 9202:9200
```

Le nœud s'appelle `es03`, il fait partie de la même grappe, et on lui indique qu'il peut se connecter aux nœuds `es01` ou `es02` pour récupérer la configuration actuelle de la grappe, et notamment son nœud-maitre.

On obtient un fichier dock-comp-es2.yml que vous pouvez récupérer. Arrêtez l'exécution du `docker compose` avec la commande `down`, et relancez-le avec le nouveau fichier.

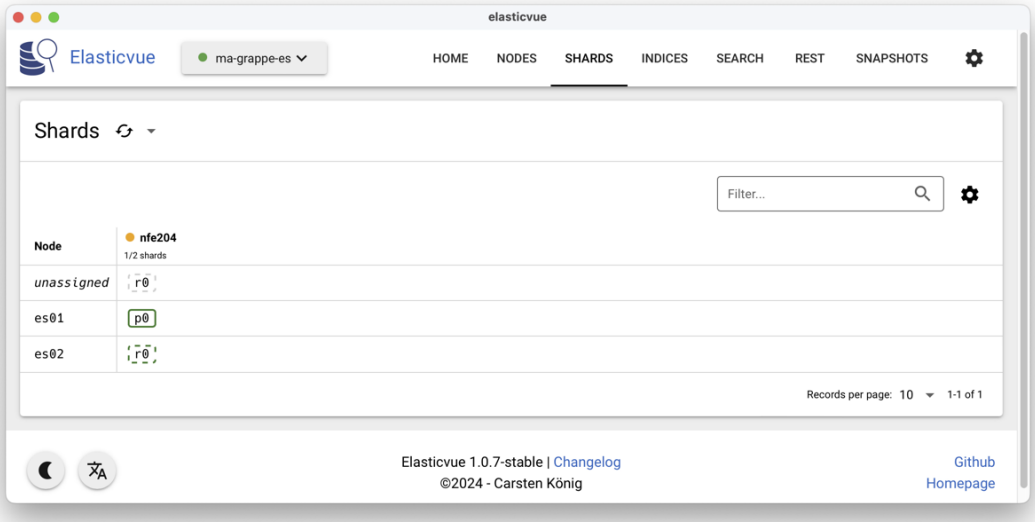


Fig. 12.13 – ElasticVue montrant l'index avec 3 copies mais seulement 2 nœuds.

```
docker compose -f dock-comp-es2.yml up
```

En consultant ElasticVue, vous devriez obtenir l'affichage de la Fig. 12.14, avec ses trois nœuds et son index en vert, dont la copie primaire stockée sur le maître `es03`. Tout va bien !

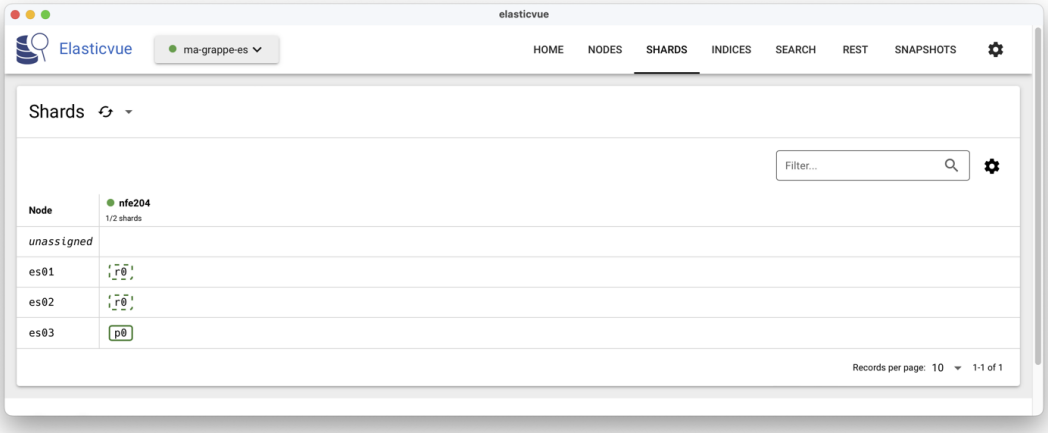


Fig. 12.14 – ElastiVue montrant l'index avec 3 copies sur 3 nœuds.

Important : En production, on ne procède évidemment pas à un arrêt et un redémarrage de l'ensemble des nœuds. Et d'ailleurs la configuration est plus complexe.

12.2.5 Reprise sur panne

Regardons plus précisément le fonctionnement de la réplication et de la reprise sur panne. ElasticSearch fonctionne en mode Maître-Esclave, avec reprise sur panne automatisée. Faites maintenant l'essai : interrompez le nœud-maître avec Docker. C'est facile avec le *Dashboard*, sinon en ligne de commande :

— Avec la commande `docker ps -a` cherchez l'indentifiant du nœud maître
— Arrêtez-le avec `docker stop <identifant>`

En supposant que le nœud maître était``es03``, vous obtenez alors l'affichage de la Fig. 12.15.

Important : Si vous avez arrêté `es01`, il faut se reconnecter à la grappe avec l'un des autres nœuds (donc, sur le port 9201 ou 9202).

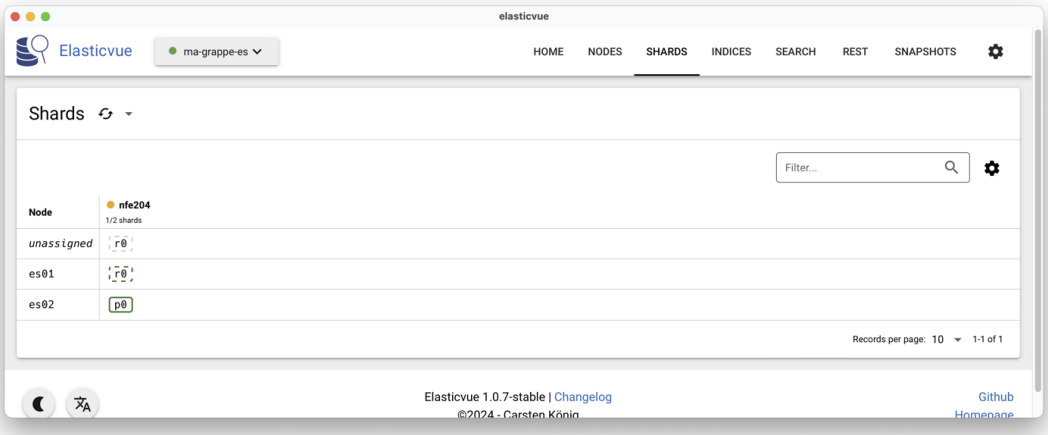


Fig. 12.15 – ElasticSearch montrant l'index après panne de `es03`

Bonne nouvelle : le nœud maître est maintenant `es01` et la copie primaire de l'index est sur `es02`. L'index peut donc continuer à fonctionner. Mais c'est en mode dégradé : un de ses replicas est marqué comme « Unassigned » : il faut redémarrer le nœud `es03` pour que la grappe retrouve un statut sain avec les trois copies sur trois nœuds différents.

Relancez le nœud `es03` avec la commande `docker start <identifiant>` : tout devrait rentrer dans l'ordre. Par rapport à la situation avant la panne, le maître a changé, et la copie primaire a également changé.

Vous avez tout compris ? Passez au quiz. Les exercices en fin de chapitre proposent également un approfondissent de plusieurs notions survolées ici.

12.2.6 Quiz

12.2.7 Mise en pratique

MEP Ex-MEP-ES1 : mise en pratique

Vous êtes invités à reproduire les commandes ci-dessus pour créer votre index ElasticSearch et tester la reprise sur panne. Testez d'abord sur une machine isolée, puis (si vous êtes en salle de TP) groupez-vous pour former des grappes de quelques serveurs, et testez les commandes de création (envoyez des insertions sur les différents nœuds) et de recherche (idem).

MEP Ex-MEP-ES2 : pour aller plus loin (optionnel)

Quelques questions intéressantes à creuser (en regardant la doc, en interrogeant Google). Ces questions peuvent former le point de départ d'une étude plus complète consacrée à ElasticSearch.

— Si j'envoie des commandes d'insertion à n'importe quel nœud, est-ce que cela fonctionne ? Cela signifie-t-il qu'ElasticSearch est en mode multinœuds et pas en mode maître-esclave ? Cherchez les mot-clés « *primary shard* » pour étudier la question.
— Comment exploiter la disponibilité des mêmes données sur plusieurs nœuds pour améliorer les performances ? Cherchez les mots-clés *ElasticSearch balancer* et faites des essais.
— La valeur par défaut du nombre de réplicas est 1 : cela signifie qu'il existe une copie primaire et un réplica, soit deux nœuds. Mais nous savons qu'en cas de partitionnement réseau nous risquons de nous retrouver avec deux maitres ? ! Etudiez la solution proposée par ElasticSearch.

12.3 S4 : Cassandra

Ressources complémentaires

— Diapositives: La réplication dans Cassandra.
— Vidéo à venir

Nous passons maintenant à une présentation de la réplication dans Cassandra. Un *cluster* Cassandra fonctionne en mode multi-nœuds. La notion de nœud maître et nœud esclave n'existe donc pas. Chaque nœud du cluster a le même rôle et la même importance, et jouit donc de la capacité de lecture et d'écriture dans le cluster. Un nœud ne sera donc jamais préféré à un autre pour être interrogé par le client.

Un client qui interroge Cassandra contacte un nœud au hasard parmi tous les nœuds du cluster. Ce nœud que l'on appellera *le coordinateur* va gérer la demande d'écriture.

Le facteur de réplication est le paramètre du *Keyspace* qui précise le nombre de réplicas qui seront utilisés. Le facteur de réplication par défaut est de 1, signifiant que la ressource (la ligne dans une table Cassandra) sera stockée sur un seul nœud ; 3 est la valeur du facteur de réplication considérée comme optimale pour assurer la disponibilité complète du système.

Le facteur de réplication est un indicateur qui précise le nombre final de copies du document dans le cluster. Si le facteur est de 3, il ne sera donc pas écrit 1 fois, puis répliqué 3 fois, mais écrit 1 fois, et répliqué 2 fois. Considérons pour l'instant que nous avons un cluster composé de 3 nœuds, et un facteur de réplication de 3. Comme expliqué précédemment, n'importe quel nœud peut recevoir la requête du client. Ce nœud, que l'on nommera *coordinateur*, va écrire localement la ressource, et rediriger la requête d'écriture vers les 2 autres nœuds suivant.

Note : Nous verrons dans le prochain chapitre que Cassandra effectue non seulement une réplication mais également un partitionnement des données, ce qui complique un peu le schéma présenté ci-dessus. Nous nous concentrons ici sur la réplication.

12.3.1 Ecriture et cohérence des données

Cassandra dispose de paramètres de configuration avancés qui servent à ajuster le compromis entre latence et cohérence. Tout ce qui suit est un excellent moyen de constater la mise en pratique des compromis nécessaires dans un système distribué entre disponibilité, latence, cohérence et tolérance aux pannes. Relire le début du chapitre (et la partie sur CAP et PACELC) si nécessaire.

Mécanisme d'écriture

Commençons par regarder de près le mécanisme d'écriture de cassandra (Fig. 12.16). Il s'appuie sur des concepts que nous connaissons déjà, avec quelques particularités qui sont reprises de l'architecture de Big-Table (d'où l'affirmation parfois rencontrée, mais rarement explicitée, que Cassandra emprunte pour sa conception à Dynamo ET à BigTable).

Une application cliente a donc pris contact avec un des serveurs, que nous appelons le coordinateur (Fig. 12.16).

Ce dernier reçoit une demande d'écriture d'un document *d*. Ce document est immédiatement écrit dans le fichier *log* (flèche A) et placé dans une structure en mémoire, appelée *memtable* (flèche B). À ce stade, on peut considérer que l'écriture est effectuée de manière durable.

Important : Notez sur la figure une zone mémoire, sur le coordinateur, nommée « Documents en attente ». Elle sert à stocker temporairement de demandes d'insertion qui n'ont pu être complètement satisfaites. Détails plus loin.

La figure donne quelques détails supplémentaires sur le stockage des documents sur disque. Quand la *memtable* est pleine, un *flush()* est effectué pour transférer son contenu sur disque, dans un fichier structuré sous forme de *SSTable*. Essentiellement, il s'agit d'une table dans laquelle les documents sont *triés*. L'intérêt du tri est de faciliter les recherches dans les SSTables. En effet :

— il est possible d'effectuer une recherche par dichotomie, ou de construire un index non dense (cf. http://sys.bdpedia.fr/arbreb.html pour des détails) ;
— il est facile et efficace de fusionner plusieurs *SSTables* en une seule pour contrôler la fragmentation.

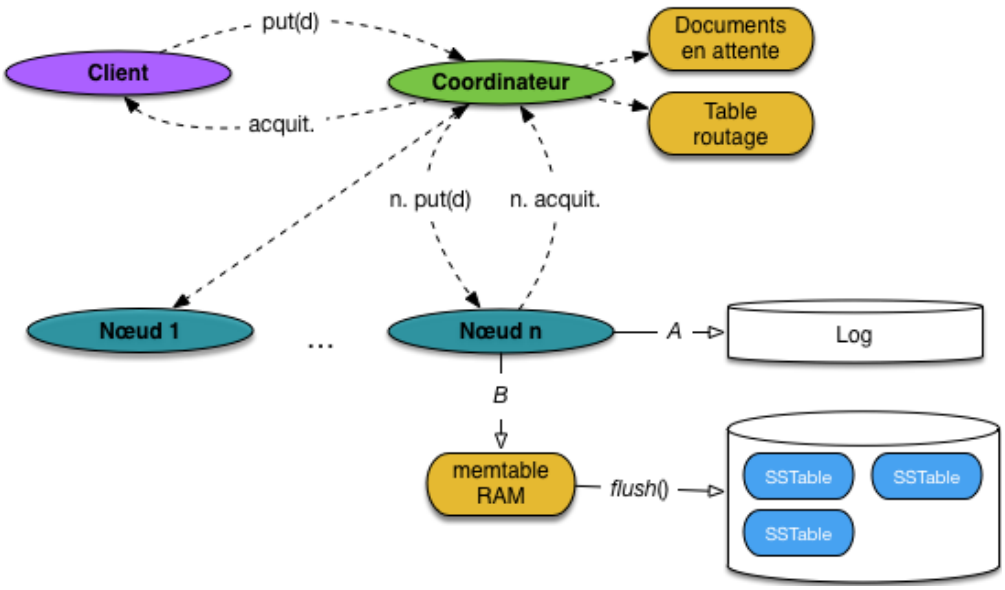


Fig. 12.16 – Le mécanisme d'écriture Cassandra

Ce mécanisme de maintien d'un stockage contenant des documents triés sur la clé est repris de BigTable, et est toujours utilisé dans son successeur, HBase. Vous pouvez vous reporter à la documentation de ce dernier, ou à l'article initial https://research.google.com/archive/bigtable.html pour en savoir plus.

Note : Le tri des documents (ou *rows*) Cassandra est effectué sur des attributs spécifiés sous forme de *compound key*. Faites une recherche dans la documentation officielle sur les termes *compound key* et *clustering* pour des détails.

Paramétrage de la cohérence (écritures)

Avec Cassandra, la cohérence des données en écriture est paramétrable. Ce paramétrage est nécessaire pour affiner la stratégie à adopter en cas de problèmes d'écriture.

Supposons par exemple que l'un des nœuds sur lesquels on doit écrire un document soit indisponible. Lorsque cela arrive, le coordinateur peut attendre que le nœud soit de nouveau actif avant d'écrire. Bien entendu, rien ne l'empêche d'écrire sur les autres réplicas. Lorsque le coordinateur attend la remise à disponibilité du nœud, il stocke alors la ressource localement, dans la zone que nous avons nommée « Documents en attente » sur la Fig. 12.16.

Si aucun nœud n'est disponible, le document n'est pas à proprement parler dans le cluster pendant cette attente. Il n'est donc pas disponible à la lecture. Ce cas de figure (extrême) s'appelle un *Hinted Handoff*.

La configuration du niveau de cohérence des écritures consiste à indiquer combien d'acquittements le coordinateur doit rececoir des nœuds de stockage avant d'acquitter à son tour le client. Voici les principales configurations possibles. Elles vont de celle qui maximise la disponibilité du système à celles qui maximisent la cohérence des données.

— **ANY** : Si tous les nœuds sont inactifs, alors le *Hinted Handoff* est appliqué : le document est écrit

dans une zone temporaire, en attente d'une nouvelle tentative d'écriture. La cohérence est minimale puisqu'aucune lecture ne peut accéder au document ! C'est la stratégie qui rend le système le plus disponible. En l'occurrence, c'est aussi la stratégie la plus dangereuse : si le nœud qui détient la zone temporaire tombe en panne, que se passe-t-il ?

— **ONE** (**TWO**, **THREE**) : La réponse au client sera assurée si la ressource a été écrite sur au moins 1 (ou 2, ou 3) réplicas.
— **QUORUM** : La réponse au client sera assurée si la ressource a été écrite sur un nombre de réplicas au moins égal à $\lfloor replication/2 \rfloor + 1$. Avec un facteur de réplication de 3, il faudra donc ($\lfloor 3/2 \rfloor + 1 = 2$ réponses). C'est un très bon compromis, car la règle du quorum va s'adapter au nombre de réplicas considéré
— **ALL** : La réponse au client sera assurée lorsque la ressource aura été écrite dans tous les réplicas. C'est la stratégie qui assure la meilleure cohérence des données, au prix de la disponibilité

Comme dans d'autres systèmes de bases de données de type NoSQL, la stratégie à adopter pour assurer la cohérence des données en écriture est souvent affaire de compromis. Plus on fait en sorte que le système soit disponible, plus on s'expose à des lectures incohérentes, retournant un version précédente du document, ou indiquant qu'il n'existe pas. À contrario, si on veut assurer la meilleure cohérence des données, alors il faut s'assurer pour chaque écriture que la ressource a été écrite partout, ce qui rend du coup le système beaucoup moins disponible.

12.3.2 Lecture et cohérence des données

Comme pour l'écriture de données, il existe des stratégies de cohérence de données en lecture. Certaines stratégies vont optimiser la réactivité du système, et donc sa disponibilité. D'autres vont mettre en avant la vérification de la cohérence des ressources, au détriment de la disponibilité.

Mécanisme de lecture

La lecture avec Cassandra est plus coûteuse que l'écriture. Sur chaque nœud, il faut en effet :

— chercher le document dans la *memtable* (en RAM)
— chercher également le document dans toutes les *SSTables*

Cela peut entraîner des accès disques, et donc une pénalité assez forte.

Paramétrage de la cohérence des lectures

La configuration consiste à spécifier le nombre de réplicas à obtenir avant de répondre au client. Dans tous les cas, une fois le nombre de réplicas obtenus, la version avec l'estampille la plus récente est renvoyée au client.

Quelques stratégies sont résumées ci-dessous :

— **ONE** (**TWO**, **THREE**) : Le coordinateur reçoit la réponse du premier réplica (ou de deux, ou de trois) et la renvoie au client. Cette stratégie assure une haute disponibilité, mais au risque de renvoyer un document qui n'est pas synchronisé avec les autres réplicas. Dans ce cas, la cohérence des données n'est pas assurée
— **QUORUM** : Le coordinateur reçoit la réponse de au moins $\lfloor replication/2 \rfloor + 1$ réplicas. C'est la stratégie qui représente le meilleur compromis

— **ALL** : Le coordinateur reçoit la réponse de tous les réplicas. Si un réplica ne répond pas, alors la requête sera en échec. C'est la stratégie qui assure la meilleure cohérence des données, mais au prix de la disponibilité du système

Pour la lecture aussi, la performance du système est affaire de compromis. Pour assurer une réponse qui reflète exactement les ressources stockées en base, il faut interroger plusieurs réplicas (voire tous), ce qui prend du temps. La disponibilité du système va donc être fortement dégradée. Si au contraire, on veut le système le plus disponible possible, alors il faut ne lire la ressource que sur 1 seul réplica, et la renvoyer directement au client. Il faudra dans ce cas accepter qu'il n'est pas impossible que le client reçoive une ressource non synchronisée, et donc fausse.

Lorsque en lecture la ressource n'est pas synchronisée entre les différents réplicas, le coordinateur détecte un conflit. Il déclenche alors deux actions :

— le réplica qui a l'estampille temporelle la plus récente parmi ceux reçu est considéré comme celui le plus à jour, et est donc retourné au client ;
— une procédure de réconciliation est lancée sur cette ressource particulière pour garantir que, au prochain appel, les données seront de nouveau synchronisée.

Il s'ensuit que pour assurer la cohérence des lectures, il faut *toujours* que la dernière version d'une ressource fasse partie des réplicas obtenus avant réponse au client. Une formule simple permet de savoir si c'est le cas. Si on note

— W le nombre de réplicas requis en écriture,
— R le nombre de réplicas requis en lecture,
— RF le facteur de réplication.

alors la cohérence est assurée si $R + W > RF$. Je vous laisse y réfléchir : l'intuition (si cela peut aider) est qu'il existe un recouvrement entre les réplicas lus et les derniers réplicas écrits, de sorte qu'au moins une lecture va accéder à la dernière version.

Par exemple,

— si *W=ALL* et *R=1* : on se satisfait d'une lecture, mais comme tous les écritures sont synchronisées, on est sûr qu'elle renvoie la dernière mise à jour.
— si *W=1* et *R=ALL*, le raisonnement réciproque amène à la même conclusion
— enfin, si *W=QUORUM* et *R=QUORUM*, on est sûr d'être cohérent : c'est la configuration la plus flexible, car s'adaptant automatiquement au niveau de réplication.

12.3.3 Mise en pratique

Voici un exemple de mise en pratique pour tester le fonctionnement d'un *cluster* Cassandra et quelques options. Pour aller plus lon, vous pouvez recourir à l'un des tutoriaux de Datastax, par exemple http://docs.datastax.com/en/cql/3.3/cql/cql_using/useTracing.html pour inspecter le fonctionnement des niveaux de cohérence.

Notre *cluster*

Créons maintenant un cluster Cassandra, avec 5 nœuds. Pour cela, nous créons un premier nœud qui nous servira de point d'accès (*seed* dans la terminologie Cassandra) pour en ajouter d'autres.

```
docker run -d -e "CASSANDRA_TOKEN=1" \
     --name cass1 -p 3000:9042 spotify/cassandra:cluster
```

Notez que nous indiquons explicitement le placement du serveur sur l'anneau. En production, il est préférable de recourir aux nœuds virtuels, comme expliqué précédemment. Cela demande un peu de configuration, et nous allons nous contenter d'une exploration simple ici.

Il nous faut l'adresse IP de ce premier serveur. La commande suivant extrait l'information `NetworkSettings.IPAddress` du document JSON renvoyé par l'instruction `inspect`.

```
docker inspect -f '{{.NetworkSettings.IPAddress}}' cass1
```

Vous obtenez une adresse. Par la suite on supppose qu'elle vaut 172.17.0.2.

Créons les autres serveurs, en indiquant le premier comme serveur-*seed*.

```
docker run -d -e "CASSANDRA_TOKEN=10" -e "CASSANDRA_SEEDS=172.17.0.2" \
    --name cass2 spotify/cassandra:cluster

docker run -d -e "CASSANDRA_TOKEN=100" -e "CASSANDRA_SEEDS=172.17.0.2" \
    --name cass3 spotify/cassandra:cluster

docker run -d -e "CASSANDRA_TOKEN=1000" -e "CASSANDRA_SEEDS=172.17.0.2" \
    --name cass4 spotify/cassandra:cluster

docker run -d -e "CASSANDRA_TOKEN=10000" -e "CASSANDRA_SEEDS=172.17.0.2" \
    --name cass5 spotify/cassandra:cluster
```

Nous venons de créer un cluster de 5 nœuds Cassandra, qui tournent tous en tâche de fond grâce à Docker.

Keyspace et données

Insérons maintenant des données. Vous pouvez utiliser le client *DevCenter*. À l'usage, il est peut être plus rapide de lancer directement l'interpréteur de commandes sur l'un des nœuds avec la commande :

```
docker exec -it cass1 /bin/bash
[docker]$ cqlsh 172.17.0.X
```

Créez un *keyspace*.

```
CREATE keyspace repli
         with replication = {'class':'SimpleStrategy', 'replication_factor':3};
USE repli;
```

Insérons un document.

```
CREATE TABLE data (id int, value text, PRIMARY KEY (id));
INSERT INTO data (id, value) VALUES (10, 'Premier document');
```

Nous venons de créer un *keyspace*, qui va répliquer les données sur 3 nœuds. Testons que le document inséré précedemment a bien été répliqué sur 2 nœuds.

```
docker exec -it cass1 /bin/bash
[docker]$ /usr/bin/nodetool cfstats -h 172.17.0.2 repli
```

Regardez pour chaque nœud la valeur de *Write Count*. Elle devrait être à 1 pour 3 nœuds consécutifs sur l'anneau, et 0 pour les autres. Vérifions maintenant qu'en se connectant à un nœud qui ne contient pas le document, on peut tout de même y accéder. Considérons par exemple que le nœud `cass1` ne contient pas le document.

```
docker exec -it cass1 /bin/bash
[docker]$ cqlsh 172.17.0.X
cqlsh > USE repli;
cqlsh:repli > SELECT * FROM data;
```

Cohérence des lectures

Pour étudier la cohérence des données en lecture, nous allons utiliser la ressource stockée, et stopper 2 nœuds Cassandra sur les 3. Pour ce faire, nous allons utiliser Docker. Considérons que la donnée est stockée sur les nœuds cass1, cass2 et cass3

```
docker pause cass2
docker pause cass3
docker exec -it cass1 /bin/bash
[docker]$ /usr/bin/nodetool ring
```

Vérifiez que les nœuds sont bien au statut *Down*.

Nous pouvons maintenant paramétrer le niveau de cohérence des données. Réalisons une requête de lecture. Le système est paramétré pour assurer la meilleure cohérence des données. On s'attend à ce que la requête plante car en mode `ALL`, Cassandra attend la réponse de tous les nœuds.

```
docker exec -it cass1 /bin/bash
[docker]$ cqlsh 172.17.0.X
cqlsh > use repli;
# devrait renvoyer Consistency level set to ALL.
cqlsh:repli > consistency all;
# devrait renvoyer Unable to complete request: one or more nodes were␣
↪unavailable.
cqlsh:repli > select * from data;
```

Comme attendu, la réponse renvoyée au client est une erreur. Testons maintenant le mode `ONE`, qui devrait normalement renvoyer la ressource du nœud le plus rapide. On s'attend à ce que la ressource du nœud 172.17.0.X soit renvoyée.

```
docker exec -it cass1 /bin/bash
[docker]$ cqlsh 172.17.0.X
cqlsh > use repli;
cqlsh:repli > consistency one;  # devrait renvoyer Consistency level set to ONE.
cqlsh:repli > select * from data;
```

Dans ce schéma, le système est très disponible, mais ne vérifie pas la cohérence des données. Pour preuve, il renvoie effectivement la ressource au client alors que tous les autres nœuds qui contiennent la ressource sont indisponibles (ils pourraient contenir une version pus récente). Enfin, testons la stratégie du quorum. Avec 2 nœuds sur 3 perdus, la requête devrait normalement renvoyer au client une erreur.

```
docker exec -it cass1 /bin/bash
[docker]$ cqlsh 172.17.0.X
cqlsh > use repli;
# devrait renvoyer Consistency level set to QUORUM.
cqlsh:repli > consistency quorum;
# devrait renvoyer Unable to complete request: one or more nodes were␣
↪unavailable.
cqlsh:repli > select * from data;
```

Le résultat obtenu est bien celui attendu. Moins de la moitié des réplicas est disponible, la requête renvoie donc une erreur. Réactivons un nœud, et re-testons.

```
docker unpause cass2
docker exec -it cass1 /bin/bash
[docker]$ nodetool ring
[docker]$ cqlsh 172.17.0.X
cqlsh > use repli;
# devrait renvoyer Consistency level set to QUORUM.
cqlsh:repli > consistency quorum;
cqlsh:repli > select * from data;
```

Lorsque le nœud est réactivé (via Docker), il faut tout de même quelques dizaines de secondes avant qu'il soit effectivement réintégré dans le cluster. Le plus important est que la règle du quorum soit validée, avec 2 nœuds sur 3 disponibles, Cassandra accepte de retourner au client une ressource.

12.3.4 Quiz

12.4 S4 : réplication dans MongoDB

Supports complémentaires

— Diapositives: distribution et réplication dans MongoDB
— Vidéo de la session

MongoDB présente un cas très représentatif de gestion de la réplication et des reprises sur panne. La section qui suit est conçue pour accompagner une mise en œuvre pratique afin d'expérimenter les concepts étudiés précédemment. Si vous disposez d'un environnement à plusieurs machines, c'est mieux ! Mais si vous n'avez que votre portable, cela suffit : les instructions données s'appliquent à ce dernier cas, et sont faciles à transposer à une véritable grappe de serveurs.

12.4.1 Les *replica set*

Une grappe de serveurs partageant des copies d'un même ensemble de documents est appelée un *replicat set* (RS) dans MongoDB. Dans un RS, un des nœuds joue le rôle de maître (on l'appelle *primary*) ; les autres (esclaves) sont appelés *secondaries*. Nous allons nous en tenir à la terminologie maître-esclave pour rester cohérent.

Note : Une version ancienne de MongoDB fonctionnait en mode dit « maître-esclave ». L'évolution de MongoDB (notamment avec la mise au point d'une méthode de *failover* automatique) a amené un changement de terminologie, mais les concepts restent identiques à ceux déjà présentés.

Un RS contient typiquement trois nœuds, un maître et deux esclaves. C'est un niveau de réplication suffisant pour assurer une sécurité presque totale des données. Dans une grappe MongoDB, on peut trouver plusieurs *replica sets*, chacun contenant un sous-ensemble d'une très grande collection : nous verrons cela quand nous étudierons le partitionnement. Pour l'instant, on s'en tient à un seul *replica set*.

La Fig. 12.17 montre le fonctionnement de MongoDB, à peu de chose près identique à celui décrit dans la section générale. Le maître est ici chargé du stockage de la donnée principale. L'écriture dans la base est paresseuse, et un journal des transactions est maintenu par le maître (c'est une collection spéciale nommée `opLog`). La réplication vers les deux esclaves se fait en mode asynchrone.

Deux niveaux de cohérence sont proposés par MongoDB. La *cohérence forte* est obtenue en imposant au client d'effectuer toujours les lectures via le maître. Dans un tel mode, les esclaves ne servent pas à répartir la charge, mais jouent le rôle restreint d'une sauvegarde/réplication continue, avec remplacement automatique du maître si celui-ci subit une panne. On ne constatera aucune différence dans les performances avec un système constitué d'un seul nœud.

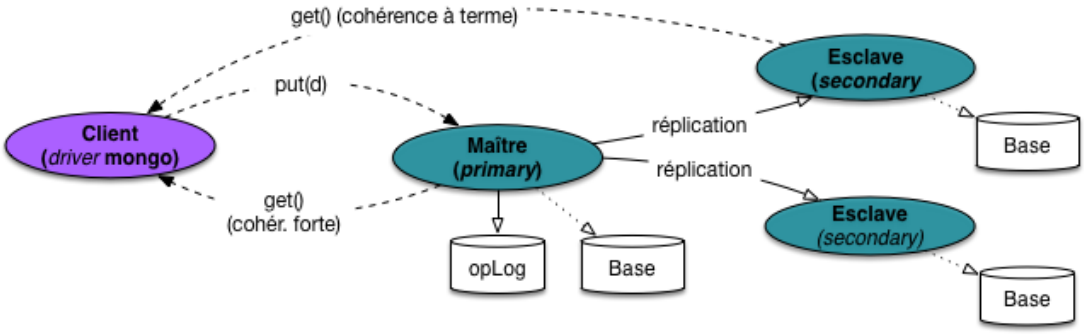


Fig. 12.17 – Un *replica set* dans MongoDB

Important : La cohérence forte est le mode par défaut dans MongoDB.

La cohérence à terme est obtenue en autorisant les clients (autrement dit, très concrètement, le *driver* MongoDB intégré à une application) à effectuer des *lectures* sur les esclaves. Dans ce cas on se retrouve exactement dans la situation déjà décrite dans le commentaire de la Fig. 12.4.

12.4.2 La reprise sur panne dans MongoDB

Tous les nœuds participant à un *replica set* échangent des messages de surveillance. La procédure de *failover* (reprise sur panne) est identique à celle décrite dans la section précédente, avec élection d'un nouveau maître par la majorité des nœuds survivants en cas de panne.

Pour assurer qu'une élection désigne toujours un maître, il faut que le nombre de votants soit impair. Pour éviter d'imposer l'ajout d'un nœud sur une machine supplémentaire, MongoDB permet de lancer un serveur *mongod* en mode « arbitre ». Un nœud-arbitre ne stocke pas de données et en général ne consomme aucune ressource. Il sert juste à atteindre le nombre impair de votants requis pour l'élection. Si on ne veut que deux copies d'un document, on définira donc un *replica set* avec un maître, un esclave et un arbitre (tous les trois sur des machines différentes).

12.4.3 À l'action : créons notre *replica set*

Passons aux choses concrètes. Nous allons créer un *replica set* avec trois nœuds avec Docker. Chaque nœud est un serveur `mongod` qui s'exécute dans un conteneur. Ces serveurs doivent pouvoir communiquer entre eux par le réseau. Après quelques essais, la solution qui me semble la plus simple est de lancer chaque serveur sur un port spécial, et de publier ce port sur la machine hôte.

Voici les commandes de création des conteneurs, nommés `mongo1`, `mongo2` et `mongo3` :

```
docker run --name mongo1 --net host mongo mongod --replSet mon-rs --port 30001
docker run --name mongo2 --net host mongo mongod --replSet mon-rs --port 30002
docker run --name mongo3 --net host mongo mongod --replSet mon-rs --port 30003
```

Pour bien comprendre :

— l'option `--net host` indique que les ports réseau des conteneurs sont publiés sur la machine-hôte de Docker ;
— l'option `replSet` indique que les serveurs *mongod* sont prêts à participer à un *replica set* nommé `mon-rs` (donnez-lui le nom que vous voulez).
— l'option `--port` indique le port sur lequel le serveur `mongod` est à l'écoute ; comme ce port est publié sur la machine hôte, en combinant l'IP de cette dernière et le port, on peut s'adresser à l'un des trois serveurs.

Vous pouvez lancer ces commandes dans un terminal configuré pour dialoguer avec Docker. Si vous utilisez Kitematic, lancez un terminal depuis l'interface à partir du menu `File`.

Une fois créés, les trois conteneurs sont visibles dans Kitematic, on peut les stopper ou les relancer.

Note : On suppose dans ce qui suit que l'IP de la machine-hôte est 192.168.99.100.

Tout est prêt, il reste à lancer les commandes pour connecter les nœuds les uns aux autres. Lancez un client `mongo` pour vous connecter au premier nœud.

```
mongo --host 192.168.99.100 --port 30001
```

Initialisez le *replica set*, et ajoutez-lui les autres nœuds.

```
rs.initiate()
rs.add ("192.168.99.100:30002")
rs.add ("192.168.99.100:30003")
```

Le *replica set* est maintenant en action ! Pour savoir quel nœud a été élu maître, vous pouvez utiliser la fonction `db.isMaster()`. Et pour tout savoir sur le *replica set* :

```
rs.status()
```

Regardez attentivement la description des trois participants au *replica set*. Qui est maître, qui est esclave, quelles autres informations obtient-on ?

On peut donc insérer des données, qui devraient alors être répliquées. Connectez-vous au maître (pourquoi ?) et insérer (par exemple) notre collection de films.

Note : Rappelons que pour importer la collection vous utiliser `mongoimport`

```
mongoimport -d nfe204 -c movies --file movies.json --jsonArray --host <hostIP> --
↪port <xxx>
```

Ou bien importer le fichier http://b3d.bdpedia.fr/files/movies-mongochef.json avec MongoChef.

Maintenant, on peut supposer que la réplication s'est effectuée. Vérifions : connectez-vous au maître et regardez le contenu de la collection `movies`.

```
use nfe204
db.movies.find()
```

Maintenant, faites la même chose avec l'un des esclaves. Vous obtiendrez sans doute un message d'erreur. Ré-essayez après avoir entré la commande `rs.slaveOk()`. À vous de comprendre ce qui se passe.

Vous pouvez faire quelques essais supplémentaires :

— insérer en vous adressant à un esclave,
— insérer un nouveau document avec un autre client (par exemple RoboMongo) et regarder quand la réplication est faite sur les esclaves,
— arrêter les serveurs, les relancer un par un, regarder sur la sortie console comment MongoDB cherche à reconstituer le *replica set*, le maître est-il toujours le même ?
— et ainsi de suite : en bref, vérifiez que vous êtes en mesure de comprendre ce qui se passe, au besoin en effectuant quelques recherche ciblées sur le Web.

12.4.4 Testons la reprise sur panne

Pour vérifier le comportement de la reprise sur panne nous allons (gentiment) nous débarasser de notre maître. Vous pouvez l'arrêter depuis Docker, ou entrer la commande suivante avec un client connecté au Maître.

```
use admin
db.shutdownServer()
```

Maintenant consultez les consoles pour regarder ce qui se passe. C'est un peu comme donner un coup de pied dans une fourmilière : tout le monde s'agite. Essayez de comprendre ce qui se passe. Qui devient le maître ? Vérifiez, puis redémarrez le premier nœud maître. Vérifiez qu'une nouvelle élection survient. Qui est encore le maître à la fin ?

12.4.5 Quiz

12.4.6 Mise en pratique

MEP Ex-MEP-1 : comprendre la documentation

Consultez la documentation en ligne MongoDB, et étudiez les points suivants

— qu'est-ce que la notion de *write concern*, à quoi cela sert-il ?
— qu'est-ce que la notion de *rollback* dans MongoDB, dans la cadre de la reprise sur panne ?
— expliquez la notion d'idempotence et son utilité pour le journal des transactions (aide : lire la documentation sur l'"*oplog*).

MEP Ex-MEP-2 : gérer une vraie grappe de serveurs

Cet exercice ne vaut que si vous avez une grappe de serveurs à votre disposition. Il est particulièrement conçu pour les exercices en direct du cours NFE204, en salle machine.

Définissez votre grappe de serveurs (par exemple, toutes les machines d'une même rangée forment un grappe).

Important : un *replica set* peut avoir *au plus* 7 nœuds votants. Si vous voulez avoir plus de 7 nœuds, certains doivent être configurés comme non-votants (chercher `non-voting-members` dans la documentation en ligne).

Définissez le nom de votre *replica set* (par exemple « rang7 »).

- créez le *replica set* avec tous les serveurs de la grappe ; identifiez le maître ;
- insérez des données dans une collection commune, surveillez la réplication ;
- tuez (gentiment) quelques-uns des esclaves, regardez ce qui se passe au niveau des connexions et échanges de messages (les serveurs impriment à la console) ;
- tuez le maître et regardez qui est nouvellement élu ;
- essayez de tuer au moins la moitié des participants au même moment et regardez ce qui se passe avec ce pauvre *replica set*.
- ajoutez un quatrième nœud, comment se passe l'élection ?
- ajoutez un nœud-arbitre, même question.

12.5 Exercices

Exercice Ex-rep-1 : comment fonctionne un site de commerce électronique ?

Prenons un site de commerce électronique à grande échelle, type Amazon. En interne, ce système s'appuie sur un système NoSQL avec cohérence à terme.

- Décrivez un scénario où vous choisissez un produit, sans le voir apparaître dans votre panier.
- Vous ré-affichez votre panier, le produit apparaît. Que s'est-t-il passé ?
- Vous supprimez un produit, en choisissez un autre, les deux apparaissent dans votre panier. Que s'est-t-il passé ?

Correction

Réponses :

- Le produit a été mis dans le panier sur un nœud *N*, puis la lecture du panier s'est effectuée sur un autre nœud *M*, avant synchronisation.
- La synchronisation s'est finie, au moins sur le nœud de lecture.
- Comme pour la première question, la version du panier qui apparaît est celle où le produit a été ajouté. On peut supposer que le système privilégie l'achat à la suppression…

Exercice Ex-rep-2 : le problème des deux armées

Pour bien comprendre la difficulté de construire des systèmes distribués fiables (et l'importance de la réplication), voici un premier problème classique, celui des deux armées. Un fort défendu par une armée verte

est encerclé par deux armées, la rouge au nord, la bleue au sud. Les généraux des armées bleue et rouge (appelons-les B et R) doivent se coordonner pour attaquer en convenant d'un jour et d'une heure précis : c'est la condition nécessaire et suffisante de la réussite. Il peuvent envoyer des messagers, mais il est possible que ces derniers soient interceptés par les défenseurs.

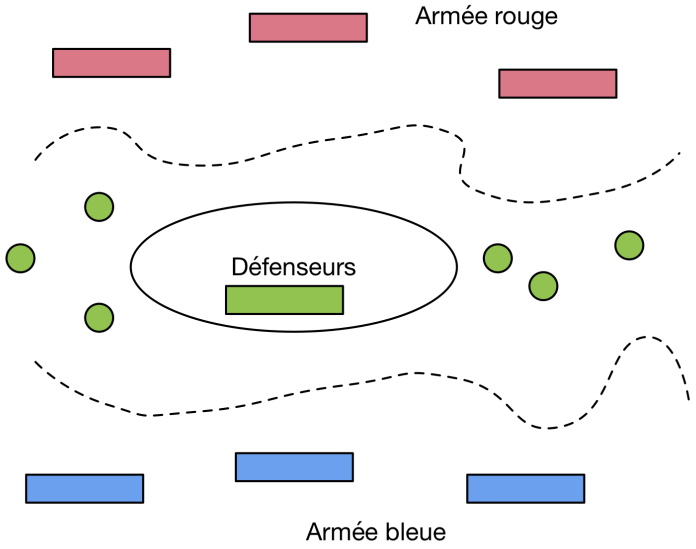


Fig. 12.18 – Les deux armées et le défenseur

Le problème est un classique des enseignements de systèmes distribués et vise à illustrer un cas de recherche de consensus en l'absence de communications totalement fiables. Il faut donc chercher à définir un protocole *tolérant aux pannes de communication*.

— Quelle stratégie permet à B et R de s'assurer qu'ils sont d'accord pour attaquer au même moment ? Raisonnez « par cas » en étudiant les différentes possibilités et essayez d'en trouver un dans lequel les deux généraux peuvent savoir de manière sûre qu'ils partagent la même information.
— (Plus difficile) Vous devriez arriver à vous faire une idée sur l'existence ou non d'un procole : démontrez cette intuition !

Correction

B décide d'attaquer le 14 à 6h du matin : il envoie un message à R. Mais tant que R n'a pas confirmé, B ne sait pas si le message est bien arrivé.

De son côté, si R reçoit le message, il peut envoyer une confirmation, mais il ne sera jamais sûr qu'elle est bien arrivée. La sagesse lui demande de ne pas attaquer le 14 à 6h du matin. Et ainsi de suite, car il faudrait que B confirme avoir reçu la confirmation, etc.

Un peu plus formellement : supposons qu'il existe un protocole avec une chaîne de messages $m_1 \cdots m_k$ telle que les deux généraux sont sûrs de partager la décision d'attaquer le jour J à l'heure H. Supposons que m_k ait été envoyé de R à B (l'hypothèse complémentaire mène au même résultat) et que ce message m_k ne soit pas arrivé. Pour R, cela ne change rien, et il va donc décider d'attaquer. Comme le protocole est supposé

correct, un consensus doit exister à cette étape m_{k-1} et B est censé décider attaquer également. (Si ce n'était pas le cas, on aurait trouvé une séquence de messages $m_1 \cdots m_{k-1}$ produite par le protocole menant à un non-consensus).

On peut donc supprimer le dernier message sans affecter le résultat. Les messages ne servent donc à rien : on est dans la situation initiale où les généraux ne peuvent décider.

Exercice Ex-rep-3 : le problème des époux trompés

Encore un petit problème de calcul distribué qui montre un autre type de raisonnement à priori insoluble mais qui (cette fois) trouve sa solution. Nous sommes au royaume des Amazones, chaque amazone a un époux, et certains sont infidèles (entendons-nous bien : le problème pourrait être exposé dans beaucoup de situations équivalentes, ou en transposant les rôles). Comme de juste, quand l'époux d'une amazone est infidèle, tout le monde le sait sauf elle.

Un jour la reine prend la parole : « je sais de source sûre qu'il existe des infidélités dans mon royaume ; je n'ai pas le droit de les révéler et aucune d'entre vous non plus, mais si vous êtes sûre que votre époux est infidèle, je vous ordonne de le sacrifier le soir à minuit ». Il y a 17 époux infidèles : le 17ème jour, à minuit, les 17 amazones trompées sacrifient leur époux.

Comment ont-elles fait ? À vous d'exposer le protocole et de montrer qu'il est correct (ce qui est préférable). Un peu d'aide : commencer à raisonner dans le cas où il y a un seul époux infidèle dans le royaume. Il reste ensuite à construire un raisonnement incrémental.

Question subsidiaire : le discours de la reine ne semble contenir aucune information. En quoi joue-t-il le rôle déclencheur ?

Correction

Réponses :

- Un seul époux infidèle : l'amazone concernée est la seule à ne pas connaître de cas infidélité ; elle déduit donc du discours de la reine que c'est elle qui est concernée et elle tue son époux.
- *k* époux infidèles : chacune des *k* amazones trompées connaît *k-1* cas d'infidélité. Or le jour $k-1$, personne n'est tué. Le k ième jour, chacune en déduit qu'elle est concernée et tue son époux.

Exercice Ex-rep-4 : une autre approche pour la cohérence forte

Supposons que l'on applique la règle du quorum au moment d'une lecture : sur les *R* versions que l'on récupère, on choisit celle qui apparaît en majorité.

Montrer que la règle $R+W > RF$ ne garantit plus la cohérence. Prendre par exemple RF=5, W=R=3 et donner un contre-exemple.

Correction

Facile : on écrit la dernière version sur les trois premiers nœuds N1, N2 et N3, puis on lit sur N3, N4 et N5 : la version ancienne sur N4 et N5 est en majorité et c'est elle qui est choisie.

Exercice Ex-rep-5 : allons plus loin avec Elastic Search

Cet exercice consiste à explorer la documentation Elastic Search pour répondre à certaines questions laissées en suspens dans la section consacrée à ce système.

Commencez par étudier les différents types de nœuds : https://www.elastic.co/guide/en/elasticsearch/reference/current/modules-node.html.

- Expliquez les différences entre les rôles *master* et *data*.
- Expliquez en quoi consiste un nœud coordinateur
- Peut-on avoir un nœud qui n'a que le rôle *master* ? Expliquez.
- Peut-on avoir un nœud qui n'a que le rôle *coordinateur* ? Expliquez.

Regardez ensuite la section https://www.elastic.co/guide/en/elasticsearch/reference/current/modules-discovery.html.

- Expliquez comment un nœud ajouté à un grappe trouve le *master* de la grappe
- Expliquez comment Elastic Search détermine un nouveau *master* en cas de panne affectant un nœud. Vous pouvez citer en référence des liens vers les parties de la documentation concernées.

Exercice Ex-rep-5 : parlons des transactions ACID distribuées

Les transactions considérées dans la session qui précède sont très éloignées de celles, dites ACID, en usage dans les SGBD relationnelles. Dans une transaction ACID, on ne regarde pas une opération, mais une *séquence* d'opérations censées s'effectuer de manière solidaire (« tout ou rien », c'est *l'atomicité*), marquée de manière définitive par des *commit* ou des *rollback* (c'est la *durabilité*), et pendant lesquelles les mises à jour effectuées par une transaction T sont invisibles des autres (c'est *l'isolation*).

En l'absence de propriétés ACID, il est bien difficile d'utiliser un système pour, par exemple, effectuer des virements bancaires ou réserver des billets d'avion.

Il existe un protocole pour effectuer des transactions ACID dans un système distribué : *le commit à deux phases* (TPC). Il arrivera (peut-être / sans doute) dans le systèmes NoSQL. En attendant il est instructif de se pencher sur son fonctionnement.

Note : Cet exercice est long et assez difficile, il s'apparente à un atelier d'approfondissement. **À faire de manière optionnelle si vous avez le temps et l'appétit pour des sujets avancés en gestion de données.**

Commencer par étudier le fonctionnement de l'algorithme, par exemple depuis les sources suivantes

- https://en.wikipedia.org/wiki/Two-phase_commit_protocol
- http://courses.cs.vt.edu/~cs5204/fall00/distributedDBMS/duckett/tpcp.html

Répondez aux questions suivantes

- Quelles sont les hypothèses requises sur le fonctionnement du système pour que le TPC fonctionne ?

— Expliquer pourquoi le coordinateur ne peut pas envoyer directement un ordre *commit* (ou *rollback*) à chaque participant (»*One phase commit* »).
— En fonction de ce que nous a apris l'exemple des deux armées, dressez la liste des problèmes qui peuvent se poser et empêcher la finalisation d'une transaction TPC.
— À l'issue de la première phase, quels engagements a pris chaque participant ?
— Qu'est-ce qui garantit qu'une transaction distribuée va finir par s'exécuter ?
— Quel est le plus grand inconvénient du TPC ?

Correction

Réponses :

— Chaque nœud est capable d'assurer des transactions ACID, sur support persistant avec un fichier journal. De plus on fait l'hypothèse de la stabilité de la topologie : aucun nœud ne disparaît définitivement, et tous finissent par répondre à toute requête. Ce sont des hypothèses fortes....
— Le *one phase commit* est très fragile : il suffit qu'un participant échoue à effectuer sa transaction locale et soit obligé d'effectuer un *rollback*. Si tous les autres ont validé, la contrainte de durabilité rend la situation irrémédiablement fautive.
— Après la première phase, chaque participant s'est déclaré prêt à effectuer soit un commit, soit un rollback de la transaction, et il maintient toutes les ressources verrouillées en attente de la décision du coordinateur. Notez que même en cas de panne d'un paticipant, suivi d'un redémarrage, la capacité d'effectuer commit ou rollback est préservée grâce au *log*.
— Voir les hypothèses : contrairement au cas des 2 armées, on suppose que tous les messages finissent par arriver. Le coordinateur va donc toujours envoyer une confirmation, et les participants toujours finir par envoyer un acquittement.
— Le TPC entraîne un blocage des ressources nécessaires sur chaque participant pendant toute la durée de la transaction. Si le réseau fonctionne tout va bien. En cas de panne du coordinateur et de défaillance (supposée temporaire) du réseau ça peut durer longtemps.

Aux dernières nouvelles (2020) aucun système NoSQL ne propose des transactions ACID distribuées, et ce ne sont pas des systèmes transactionnels au sens fort du terme. La fiabilité est en fait plutôt assurée par la réplication que par des protocoles algorithmiques compliqués. On pourrait dire (abruptement) qu'on préfère payer du stockage que de la complexité (et du temps de calcul).

CHAPITRE 13

Systèmes NoSQL : le partitionnement

La réplication est essentiellement destinée à pallier les pannes en dupliquant une collection sur plusieurs serveurs et en permettant donc qu'un serveur prenne la relève quand un autre vient à faillir. Le fait de disposer des *mêmes* données sur plusieurs serveurs par réplication ouvre également la voie à la distribution de la charge (en recherche, en insertion) et donc à la scalabilité. Ce n'est cependant pas une méthode appliquable à grande échelle car, sur ce que nous avons vu jusqu'ici, elle implique la copie de *toute* la collection sur *tous* les serveurs.

Le *partitionnement*, étudié dans ce chapitre, est la technique privilégiée pour obtenir une véritable scalabilité. Commençons par quelques rappels, que vous pouvez passer allègrement si vous êtes familier des notions de base en gestion de données.

13.1 S1 : les bases

Supports complémentaires :
- Diapositives: principes du partitionnement
- Vidéo sur les principes du partitionnement

13.1.1 Principes généraux

On considère une collection constituée de documents (au sens général du terme = valeur plus ou moins structurée) dotés d'un identifiant. Dans ce chapitre, on va essentiellement voir une collection comme un ensemble de paires *(i, d)*, où *i* est un identifiant et *d* le document associé.

Le principe du partitionnement s'énonce assez simplement : la collection est divisée en *fragments* formant une *partition* de l'ensemble des documents.

Vocabulaire : ensemble, fragment, élément

Un petit rappel pour commencer. Une *partition* d'un ensemble S est un ensemble $\{F_1, F_2, \cdots, F_n\}$ de *parties* de S, que nous appellerons *fragments*, tel que :

— $\bigcup_i F_i = S$
— $F_i \cap F_j = \emptyset$ pour tout $i, j, i \neq j$

Dit autrement : chaque élément de la collection S est contenu dans un et un seul fragment F_i.

Dans notre cas S est une collection, les éléments sont des documents, et les fragments sont des sous-ensembles de documents.

Note : On trouvera souvent la dénomination *shard* pour désigner un fragment, et *sharding* pour désigner le partitionnement.

Clé de partitionnement

Un partitionnement s'effectue toujours en fonction d'une *clé*, soit un ou plusieurs attributs dont la valeur sert de critère à l'affectation d'un document à un fragment. La première décision à prendre est donc le choix de la clé.

Un bon partitionnement répartit les documents en fragments de taille comparable. Cela suppose que la clé soit suffisamment discriminante pour permettre de diviser la collection avec une granularité très fine (si possible au niveau du document lui-même). Choisir par exemple un attribut dont la valeur est constante ou réduite à un petit nombre de choix, ne permet pas au système de séparer les documents, et on obtiendra un partitionnement de très faible qualité.

Idéalement, on choisira comme clé de partitionnement l'identifiant unique des documents. La granularité de division de la collection tombe alors au niveau du document élémentaire, ce qui laisse toute flexibilité pour décider comment affecter chaque document à un fragment. C'est l'hypothèse que nous adoptons dans ce qui suit.

Structures

Il existe deux grandes approches pour déterminer une partition en fonction d'une clé : *par intervalle* et *par hachage*.

— Dans le premier cas (par intervalle), on obtient un ensemble d'intervalles disjoints couvrant le domaine de valeurs de la clé ; à chaque intervalle correspond un fragment.
— Dans le second cas (par hachage), une fonction appliquée à la clé détermine le fragment d'affectation.

Que le partitionnement soit par hachage ou par intervalle, ces structures sont toujours au nombre de deux.

— la *structure de routage* établit la correspondance entre la valeur d'une clé et le fragment qui lui est associé (ou, très précisément, l'espace de stockage de ce fragment) ;
— la *structure de stockage* est un ensemble d'espaces de stockages séparés, contenant chacun un fragment.

Sans la structure de routage, rien ne fonctionne. Elle se doit de fournir un support très efficace à l'identification du fragment correspondant à une clé, et on cherche en général à faire en sorte qu'elle soit suffisamment compacte pour tenir en mémoire RAM. Les fragments sont, eux, nécessairement stockés séquentiellement sur disque (pour des raisons de persistance) et placés si possible en mémoire (Fig. 13.1)

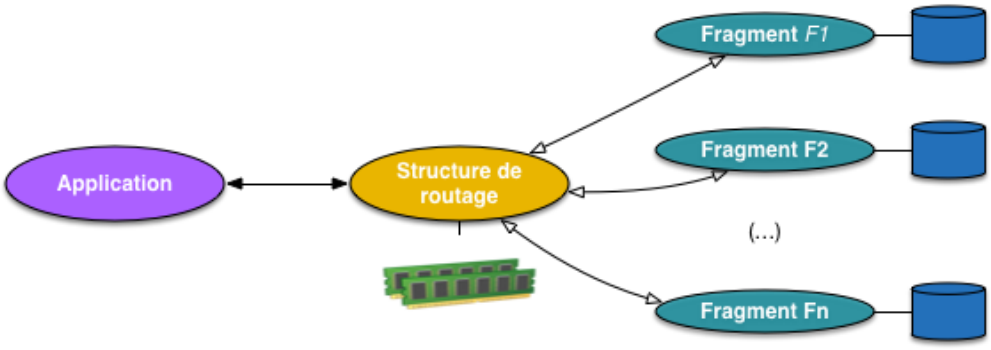


Fig. 13.1 – Vision générale des structures du partitionnement

La gestion de ces structures varie ensuite d'un système à l'autre, mais on retrouve quelques grand principes.

Dynamicité.

Un partitionnement doit être *dynamique* : en fonction de l'évolution de la taille de la collection et du nombre de ressources allouées à la structure, le nombre de fragments doit pouvoir évoluer. C'est important pour optimiser l'utilisation de l'espace disponible et obtenir les meilleurs performances. C'est aussi, techniquement, la propriété la plus difficile à satisfaire.

Opérations.

Les opérations disponibles dans une structure de partitionnement sont de type « dictionnaire ».

- *get(i) : d* renvoie le document dont l'identifiant est *i* ;
- *put(i, d)* insère le document *d* avec la clé *i* ;
- *delete(i)* recherche le document dont l'identifiant est *i* et le supprime ;
- *range(i, j) : [d]* renvoie l'ensemble des documents *d* dont l'identifiant est compris entre *i* et *j*.

Les trois premières opérations s'effectuent sur un seul fragment. La dernière peut impliquer plusieurs fragments, tous au pire.

Le fait de devoir parcourir toute la collection ne signifie pas que le partitionnement est inutile, au contraire. En effectuant le parcours *en parallèle* on diminue globalement par *N* le temps de traitement.

13.1.2 Et en distribué ?

Dans un système distribué, le principe du partitionnement se transpose assez directement de la présentation qui précède. La Fig. 13.2 montre une architecture assez générique dont nous verrons quelques variantes pratiques.

Un nœud particulier, le *routeur*, maintient la structure de *routage*, reçoit les requêtes de l'application et les redirige vers les nœuds en charge du *stockage*. Ces derniers stockent les fragments. On pourrait imaginer une équivalence stricte (un nœud = un fragment) mais pour des raisons de souplesse du système, un même nœud est en général en charge de plusieurs fragments.

Cette organisation s'additionne à celle gérant la réplication. Le routeur par exemple doit être synchronisé avec au moins un nœud-copie apte à le supléer en cas de défaillance ; de même, chaque nœud de stockage gère la réplication des fragments dont il a la charge et en informe le routeur pour que ce dernier puisse rediriger les requêtes en cas de besoin.

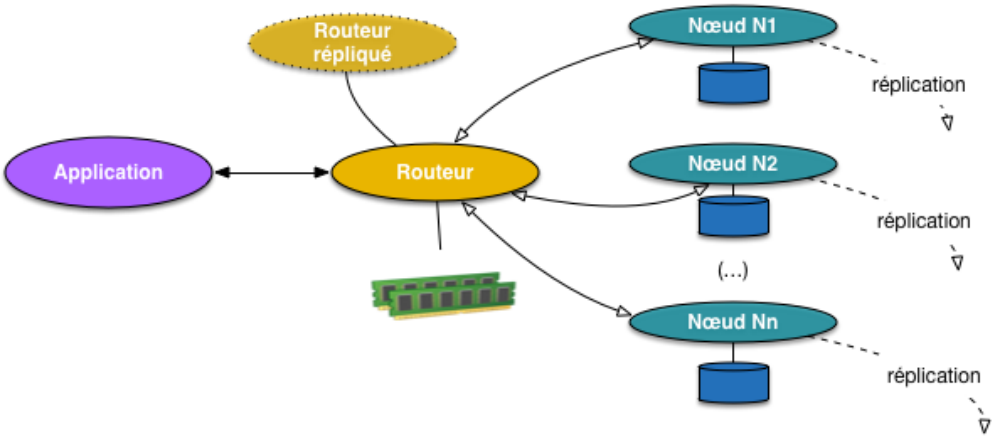


Fig. 13.2 – Partitionnement et systèmes distribués

Bien que cette figure s'applique à une grande majorité des systèmes pratiquant le partitionnement, il est malheureusement nécessaire de souligner que le vocabulaire varie constamment.

— le routeur est dénommé, selon les cas, *Master*, *Balancer*, *Primary*, *Router/Config server*, ...

— les fragments sont désignés par des termes comme *chunk*, *shard*, *tablet*, *region*, *bucket*,...

et ainsi de suite : il faut savoir s'adapter.

Note : La mauvaise habitude a été prise de parler de « partition » comme synonyme de « fragment ». On trouve des expressions comme « chaque partition de la collection », ce qui n'a aucun sens. J'espère que le lecteur de ce texte aura l'occasion d'employer un vocabulaire approprié.

Les méthodes de partitionnement, par intervalle ou par hachage, sont représentées par des systèmes de gestion de données importants

— par intervalle : HBase/BigTable, MongoDB, ...

— par hachage : Dynamo/S3/Voldemort, Cassandra, Riak, REDIS, memCached, ...

Les moteurs de recherche sont également dotés de fonctionnalités de partitionnement. ElasticSearch par exemple effectue une répartition par hachage, mais sans dynamicité, comme expliqué ci-dessous.

13.1.3 Etude de cas : ElasticSearch

Le modèle de partitionnement d'ElasticSearch est assez simple : on définit au départ le nombre de fragments, qui est immuable une fois l'index créé. Le partitionnement dans ElasticSearch n'est donc pas *dynamique*. Si la collection évolue beaucoup par rapport à la taille prévue initialement, il faut restructurer complètement l'index.

Ensuite, ElasticSearch se charge de distribuer ces fragments sur l'ensemble des nœuds disponibles, et copie *sur chaque nœud* la table de routage qui permet de diriger les requêtes basées sur la clé de partitionnement vers le ou les serveurs concernés. On peut donc interroger n'importe quel nœud d'une grappe ElasticSearch : la requête sera redirigée vers le nœud qui stocke le document cherché.

Important : ElasticSearch est un moteur de recherche et propose donc un langage de recherche bien plus

riche que le simple *get()* basé sur la clé de partitionnement. Le partitionnement est donc surtout un moyen de conserver des structures d'index de taille limitée sur lesquelles les opérations de recherche peuvent s'effectuer efficacement en parallèle.

Voici une brève présentation du partitionnement ElasticSearch, les exercices proposent des explorations complémentaires.

Lancement des serveurs

Commençons par créer une première grappe avec deux nœuds. Nous pouvons reprendre le fichier de configuration du chapitre *Systèmes NoSQL : la réplication*. Pour rappel, il se trouve ici : dock-comp-es1.yml.

```
docker compose -f dock-comp-es1.yml up
```

Note : Si vous en êtes restés à la configuration avec trois nœuds, il faut les supprimer (avec `docker rm`) avant la commande ci-dessus pour réinitiliaser proprement votre *cluster* ElasticSearch. De même si un index `nfe204` existe déjà, vous pouvez le supprimer (très facile avec ElasticVue)

Bien. Maintenant nous allons adopter une configuration avec 5 fragments et 1 réplica (donc, 2 copies de chaque document). Pour changer la configuration par défaut, il faut transmettre à ElasticSearch le fichier de configuration suivant.

```
{
  "index_patterns": "*",
  "settings": {
      "number_of_shards": "5",
      "number_of_replicas": "1"
    }
}
```

Vous pouvez le récupérer ici : es_shards_params.json. Pour des raisons que nous verrons ensuite, on ne peut pas changer le nombre de *shards après* la création d'un index. Le fichier de paramètres ci-dessus s'applique à tous les index à venir. On transmets ce fichier à l'URL `_template/index_defaults` avec un `PUT`, comme montré sur la Fig. 13.3.

Il reste à charger les données. Récupérez notre collection de films, au format JSON adapté à l'insertion en masse dans ElasticSearch, et importez-les comme nous l'avons déjà vu plusieurs fois.

L'interface ElasticVue devrait vous montrer l'équivalent de la Fig. 13.4.

Nous avons donc 5 fragments primaires (les `p`), répartis équitablement (à une unité près) sur nos deux serveurs, et répliqués chacun une fois (les `r`), soit 10 fragments au total. Avec deux serveurs, chaque fragment est stocké sur chaque serveur.

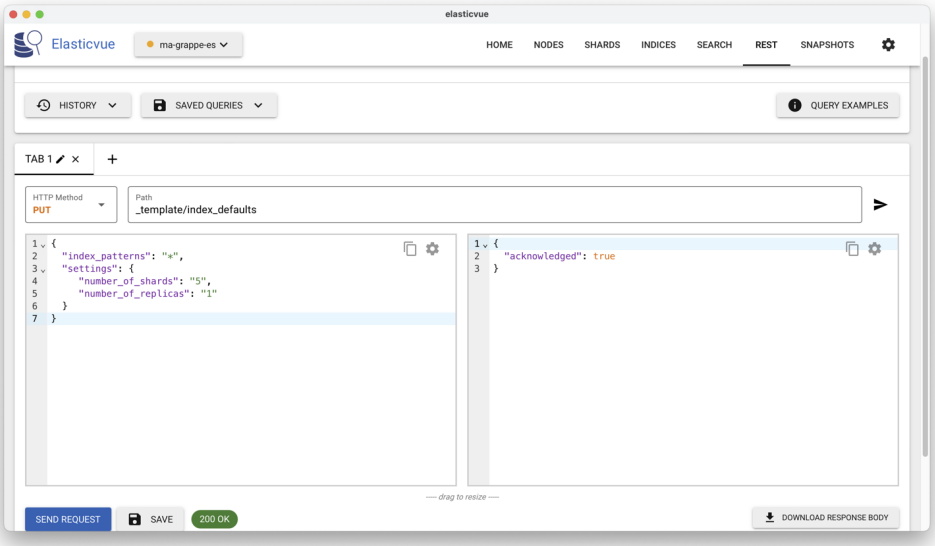


Fig. 13.3 – Commande de changement du nombre de fragments et de réplicas par défaut

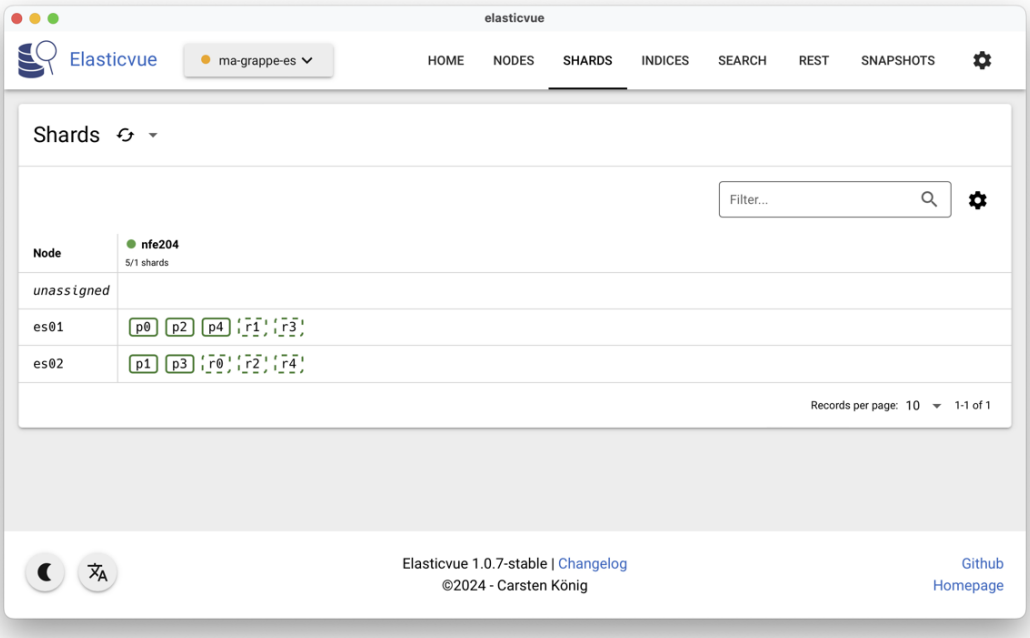


Fig. 13.4 – Un index ElasticSearch avec 5 fragments.

Ajout / suppression de nœuds

Maintenant, si nous ajoutons des serveurs, ElasticSearch va commencer à distribuer les fragments, diminuant d'autant la charge individuelle de chaque serveur. Nous reprenons le fichier dock-comp-es2.yml que vous pouvez récupérer. Arrêtez l'exécution du `docker-compose` en cours et relancez-le

```
docker compose -f dock-comp-es2.yml up
```

Avec trois serveurs, vous devriez obtenir un affichage semblable à celui de la Fig. 13.5. Je vous laisse analyser ce qui s'est passé.

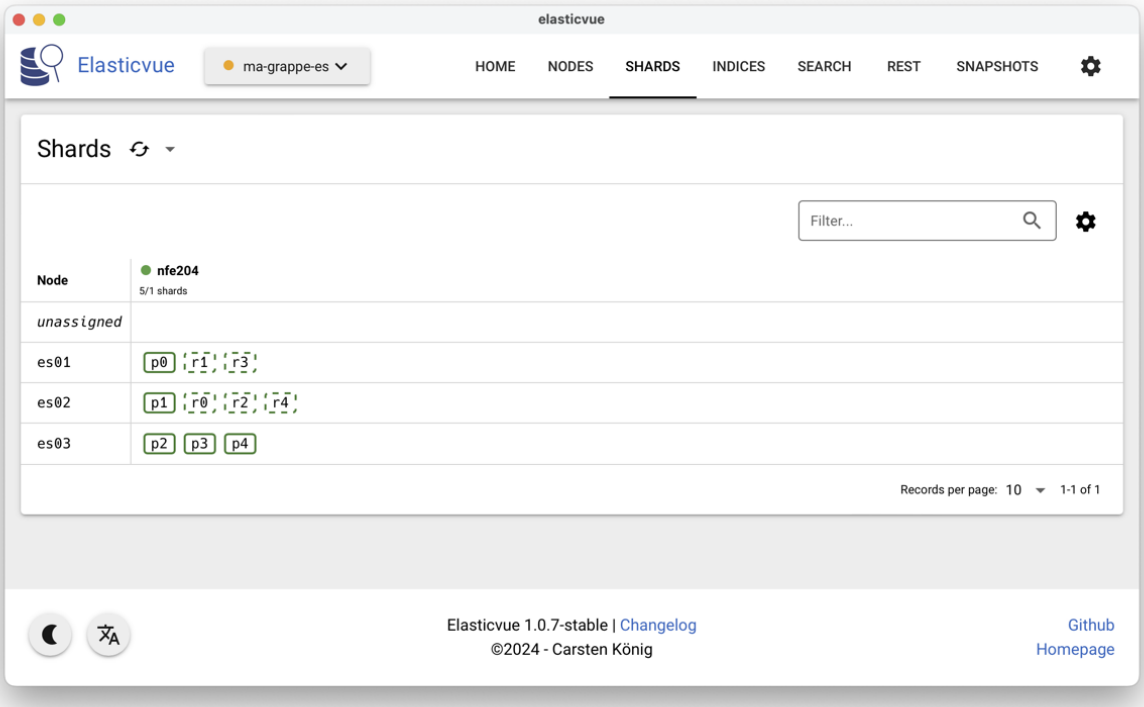


Fig. 13.5 – Distribution des fragments sur les serveurs.

On voit maintenant que la notion de « maître » est en fait raffinée dans ElasticSearch au niveau du fragment : chaque nœud est responsable (en tant que nœud de *stockage*) d'un sous-ensemble des *shards*, et est en contact (en tant que nœud de *routage*) avec les autres nœuds, dont ceux stockant les réplicas des fragments primaires. Une requête d'insertion est *toujours* redirigée par le nœud qui reçoit la requête vers le serveur stockant le fragment primaire dans lequel le nouveau document doit être placé. Une requête de lecture en revanche peut être satisfaite par n'importe quel nœud d'un *cluster* ElasticSearch, sans distinction du statut primaire/secondaire des fragments auxquels on accède.

Comment est déterminé le fragment dans lequel un document est placé ? ElasticSearch applique une méthode simple de distribution basée sur une clé (par défaut le champ `_id`) et sur le nombre de fragments.

```
fragment = hash(clé) modulo nb_fragments
```

La fonction *hash()* renvoie un entier, qui est divisé par le nombre de fragments. Le reste de cette division donne l'identifiant du fragment-cible. Avec 5 fragments, une clé hachée vers la valeur 8 sera placée dans le fragment 3, une clé hachée vers la valeur 101 sera placée dans le fragment 1, etc.

Important : Cette méthode simple a un inconvénient : si on décide de changer le nombre de fragments, tous les documents doivent être redistribués car le calcul du placement donne des résultats complètement différents. Plus de détails sur cette question dans la section consacrée au partitionnement par hachage.

Dans ElasticSearch, la table de routage est distribuée sur l'ensemble des nœuds qui sont donc chacun en mesure de router les requêtes d'insertion ou de recherche.

Vous pouvez continuer l'expérience en ajoutant d'autres nœuds, en constatant à chaque fois que les fragments (primaires ou réplicas) sont un peu plus distribués sur l'ensemble des ressources disponibles. Inversement, vous pouvez arrêter certains nœuds, et vérifier qu'ElasticSearch re-distribue automatiquement les fragments de manière à préserver le nombre de copies spécifié par la configuration (tant que le nombre de nœuds est au moins égal au nombre de copies).

13.1.4 Quiz

13.1.5 Mise en pratique

Exercice MEP-S1-1 : Créez une collection partitionnée ElasticSearch

Cet exercice consiste simplement à reproduire les commandes données ci-dessus.

Exercice MEP-S1-3 : Exploration d'ElasticSearch (atelier optionnel)

La présentation d'ElasticSearch doit être prise comme un point de départ pour l'exploration de ce système. Outre la reproduction des quelques manipulations données dans la section, voici quelques suggestions :

- Se pencher sur les questions habituelles : comment équilibrer la charge ; comment régler l'équilibre entre asynchronicité des écritures et sécurité ; comment est gérée la cohérence transactionnelle. Pour toutes ces questions, des ressources existent sur le Web qu'il faut apprendre à trouver, sélectionner et comprendre.
- Pour charger des données en masse, vous pouvez utiliser par exemple https://github.com/sematext/ActionGenerator.
- ElasticSearch propose un module original dit de *percolation*, le principe étant de déposer une requête permanente (ou « continue ») et d'être informé de tout nouveau document satisfaisant cette requête. Permet d'implanter un système de souscription-notification : à explorer.
- Kibana est un module analytique associé à ElasticSearch, équipé de très beaux modules de visualisation.

13.2 S2 : partitionnement par intervalle

Supports complémentaires :

— Diapositives: partitionnement par intervalle
— Vidéo de démonstration du partitionnement par intervalle

L'idée est simple : on considère le domaine de valeur de la clé de partition (par exemple, l'ensemble des entiers) et on le divise en n intervalles définissant n fragments. On suppose que le domaine est muni d'une relation d'ordre total qui sert à affecter sans équivoque un identifiant à un intervalle.

13.2.1 Structures et opérations

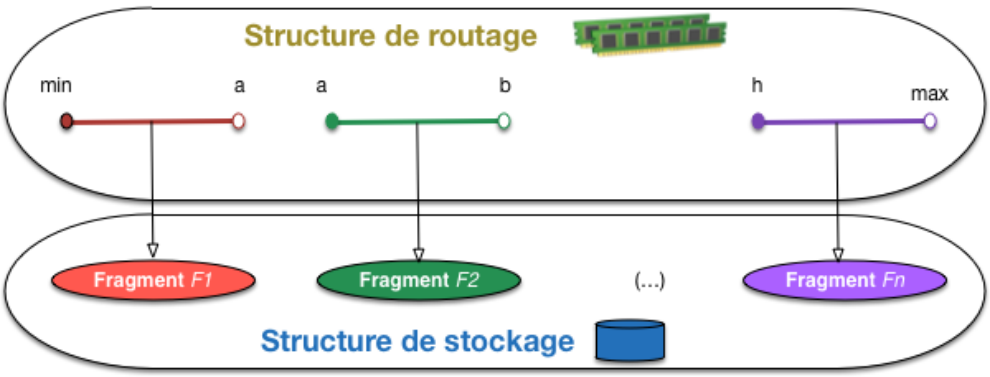


Fig. 13.6 – Partitionnement par intervalle

La Fig. 13.6 montre génériquement une structure de partionnement basée sur des intervalles. Le domaine de la clé est partitionné en intervalles semi-ouverts, dont la liste constitue la structure de routage. À chaque intervalle est associé un fragment dans la structure de stockage.

En pratique, on va déterminer la taille maximale d'un fragment de telle sorte qu'il puisse être lu très rapidement.

— dans un système orienté vers le temps réel, la taille d'un fragment est un (petit) multiple de celle d'un secteur sur le disque (512 octets) : 4 KO, 8 KO sont des tailles typiques ; le but est de pouvoir charger un fragment avec un accès disque et un temps de parcours négligeable, soit environ 10 ms pour le tout ;
— dans un système orienté vers l'analytique où il est fréquent de parcourir un fragment dans sa totalité, on choisira une taille plus grande pour minimiser le nombre des accès aléatoires au disque.

La structure de routage est constituée de paires *(I, a)* où I est la description d'un intervalle et a l'adresse du fragment correspondant. Un critère important pour choisir la taille des fragments est de s'assurer que leur nombre reste assez limité pour que la structure de routage tienne en mémoire. Faisons quelques calculs, en supposant une collection de 1 TO, et un taille de 20 octets pour une paire *(I, a)*.

— si la taille d'un fragment est de 4 KO (choix typique d'un SGBD relationnel), le routage décrit 250 millions de fragments, soit une taille de 5 GO ;
— si la taille d'un fragment est de 1 MO, il faudra 1 million de fragments, et seulement 20 MO pour la structure de routage.

Dans les deux cas, le routage tient en mémoire RAM (avec un serveur de capacité raisonnable). Le premier soulève quand même le problème de l'efficacité d'une recherche dans un tableau de 250 millions d'entrées.

On peut alors utiliser une structure plus sophistiquée, arborescente, la plus aboutie étant *l'arbre B* utilisé par *tous* les systèmes relationnels.

Le tableau des intervalles est donc assez compact pour être placé en mémoire, et chaque fragment est constitué d'un fichier séquentiel sur le disque. Les opérations s'implantent très facilement.

— *get(i)* : chercher dans le routage *(I, a)* tel que *I* contienne *i*, charger le fragment dont l'adresse est *a*, chercher le document en mémoire ;
— *put(i, d)* : chercher dans le routage *(I, a)* tel que *I* contienne *i*, insérer *d* dans le fragment dont l'adresse est *a* ;
— *delete(i)* : comme la recherche, avec effacement du document trouvé ;
— *range(i, j)* : chercher tous les intervalles dont l'intersection avec *[i, j]* est non vide, et parcourir les fragments correspondants.

Dynamicité

Comment obtient-on la dynamicité ? Et, accessoirement, comment assure-t-on une bonne répartition des documents dans les fragments ? La méthode consiste à augmenter le nombre de fragments en fonction de l'évolution de la taille de la collection.

— initialement, nous avons un seul fragment, couvrant la totalité du domaine de la clé ;
— quand ce fragment est plein, on effectue un *éclatement* (*split*) en deux parties égales correspondant à deux intervalles distincts ;
— on répète ce processus chaque fois que l'un des fragments est plein.

L'éclatement d'un fragment se comprend aisément sur la base d'un exemple illustré par la Fig. 13.7. On suppose ici que le nombre maximal de documents par fragment est de 8 (ce qui est bien entendu très loin de ce qui est réalisable en pratique). Seules les clés sont représentées sur la figure.

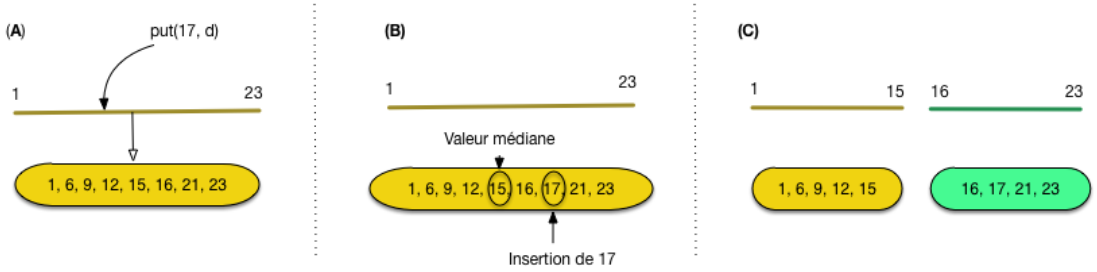


Fig. 13.7 – Eclatement d'un fragment

La situation initale (A) montre un unique fragment dont les clés couvrent l'intervalle *[1, 23]* (notez que le fragment est *trié* sur la clé). Une opération *put(17, d)* est soumise pour insérer un document avec l'identifiant 17. On se retrouve dans la situation de la part (B), avec un fragment en sur-capacité (9 documents) qui nécessite donc un éclatement.

Ce dernier s'effectue en prenant la valeur médiane des clés comme pivot de répartition. Tout ce qui se trouve à gauche (au sens large) de la valeur médiane (ici, 15) reste dans le fragment, tout ce qui se trouve à droite (au sens strict) est déplacé dans un nouveau fragment. Il faut donc créer un nouvel intervalle, ce qui nous place dans la situation finale de la partie (C) de la figure.

Cette procédure, très simple, présente de très bonnes propriétés :

— il est facile de voir que, par construction, les fragments sont équilibrés par cette répartition en deux parties égales ;
— l'utilisation de l'espace reste sous contrôle : au minimum la moitié est effectivement utilisée ;
— la croissance du routage reste faible : un intervalle supplémentaire pour chaque éclatement.

Simplicité, efficacité, robustesse : la procédure de croissance d'un partitionnement par intervalle est à la base de très nombreuses structures d'indexation, en centralisé ou distribué, et entre autres du fameux arbre-B mentionné précédemment.

Un effet indirect de cette méthode est que la collection est totalement ordonnée sur la clé : en interne au niveau de chaque fragment, et par l'ordre défini sur les fragments dans la structure de routage. C'est un facteur supplémentaire d'efficacité. Par exemple, une fois chargé en mémoire, la recherche d'un document dans un fragment peut se faire par dichotomie, avec une convergence très rapide.

13.2.2 Etude de cas : MongoDB

Dans MongoDB, le partitionnement est appelé *sharding* et correspond à quelques détails près à notre présentation générale.

Architecture

La Fig. 13.8 résume l'architecture d'un système MongoDB en action avec réplication et partitionnement. Les nœuds du système peuvent être classés en trois catégories.

— les *routeurs* (processus `mongos`) communiquent avec les applications clientes, se chargent de diriger les requêtes vers les serveurs de stockage concernés, et transmettent les résultats ;
— les *replica set* (processus `mongod`) ont déjà été présentés dans le chapitre *sysdistr* ; un *replica set* est en charge d'un ou plusieurs fragments (*shards*) et gère localement la reprise sur panne par réplication ;
— enfin un *replica set* dit « de configuration » est spécialement chargé de gérer les informations de routage. Il est constitué de *config servers* (processus `mongod` avec option `congifsrv`) qui stockent généralement la configuration complète du système : liste des *replica sets* (avec, pour chacun, le maître et les esclaves), liste des fragments et allocation des fragments à chaque *replica set*.

Les données des serveurs de configuration sont maintenues cohérentes par des protocoles transactionnels stricts. C'est ce qui permet d'avoir plusieurs routeurs : si l'un des routeurs décide d'un éclatement, la nouvelle configuration sera reportée dans les serveurs de configuration et immédiatement visible par tous les autres routeurs.

La scalabilité est apportée à deux niveaux. D'une part, la présence de plusieurs routeurs est destinée à équilibrer la charge de la communication avec les applications clientes ; d'autre part, le partitionnement permet de répartir la charge de traitement des données elles-mêmes (en lecture *et* en écriture). En particulier, le partitionnement favorise la présence des données en mémoire RAM, constituant ainsi une sorte de serveur de données virtuel doté d'une très grande mémoire principale. Idéalement, la taille de la grappe est telle que tous les documents « utiles » (soit, informellement, ceux utilisés couramment par les applications, par opposition aux documents auxquels on accède rarement) sont distribués dans la mémoire RAM de l'ensemble des serveurs.

Dans MongoDB le routage est basé (par défaut) sur un partitionnement par intervalles. Le domaine des identifiants de chaque collection est divisé par des éclatements successifs, associant à chaque fragment un intervalle de valeurs (voir les sections précédentes). La liste de tous les fragments et de leurs intervalles est maintenue

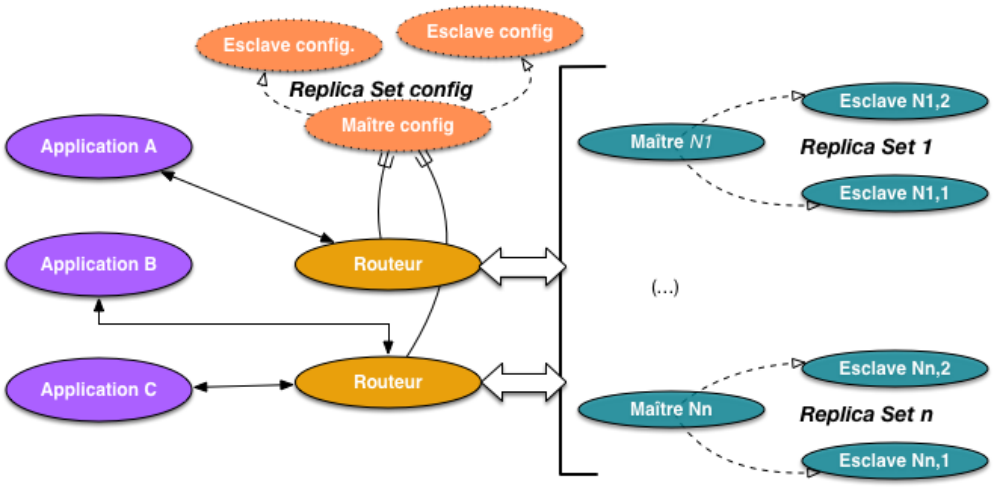


Fig. 13.8 – Architecture de MongoDB avec partitionnement

par les serveurs de configuration (qui fonctionnent en mode répliqué pour éviter la perte irrémédiable de ces données en cas de panne).

Note : Depuis la version 2.4, MongoDB propose également un partitionnement par hachage.

Chaque serveur gère un ou plusieurs fragments, et l'équilibrage du stockage se fait, après un éclatement, par déplacement de certains fragments. La Fig. 13.9 illustre le mécanisme avec un exemple simple. Initialement, nous avons deux serveurs stockant respectivement 2 fragments (F et G) et un (H). F est plein et MongoDB décide un éclatement produisant deux fragments F1 et F2, qui restent sur le même serveur.

Le processus d'équilibrage entre alors en jeu et détecte que la différence de charge entre le serveur N_i et le serveur N_j dépasse un certain seuil. Une migration de certains fragments (ici le fragment F2) est alors décidée pour aboutir à une meilleure répartition des données. Tout changement affectant un fragment (éclatement, déplacement) est immédiatement transmis aux serveurs de configuration.

La taille par défaut d'un fragment est de 64 MO. On peut donc avoir des centaines de fragments sur un même serveur. Cette granularité assez fine permet de bien équilibrer le stockage sur les différents serveurs.

Passons à la pratique. Comme d'habitude, nous utilisons Docker, en fixant la version de MongoDB pour ne pas être dépendant des changements dans les options de paramétrages apportés au fil des changements de version. Nous allons nous contenter du minimum pour mettre en place un partitionnement : un serveur de configuration, un routeur et deux serveurs de fragments chargés du stockage.

Note : Cette configuration n'est pas du tout robuste aux pannes et ne devrait pas être utilisée en production.

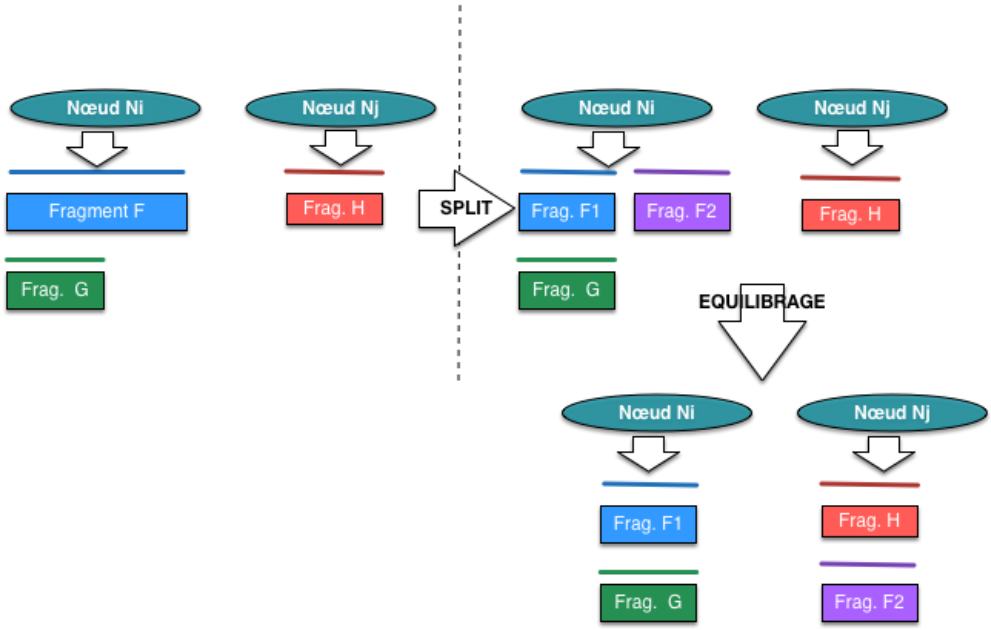


Fig. 13.9 – Gestion des fragments après partitionnement.

Configuration du système

Commençons par le serveur de configuration. C'est un nœud `mongod` dont la charge de travail est essentielle mais quantitativement minimale : il doit conserver la configuration du système distribué et la commmuniquer aux routeurs.

Voici la commande de lancement avec Docker.

```
docker run -d --name configsvr --net host mongo:3.2 \
   mongod --configsvr --replSet config-rs --port 29999
```

Les options :

- `-d` pour lancer le processus en tâche de fond (pas obligatoire).
- `--name` pour donner un nom au conteneur.
- `--net host` pour placer le conteneur dans l'espace réseau de la machine-hôte (on peut aussi sans doute rediriger le port par défaut de MongoDB, 27017).
- `--configsvr` pour spécifier qu'il s'agit d'un serveur de configuration.
- Enfin, le serveur est en écoute sur le port 29999.

Le serveur de configuration doit faire partie d'un *replica set*, en l'occurrence réduit à lui-même. Il faut quand même indiquer l'option `--replSet` au lancement, et initialiser le *replica set* en se connectant avec un client sur le port 29999 et en exécutant la commande déjà connue :

```
rs.initiate()
```

Ouf. Notre server de configuration est prêt, il est lui-même le `PRIMARY` (à vérifier avec `rs.status()`.

On continue avec le routeur, un processus `mongos` qui doit communiquer avec le serveur de configuration. On le lance sur le port 30000.

Note : Vous choisissez bien entendu les numéros de port comme vous l'entendez, l'essentiel étant qu'ils n'entrent pas en conflit avec des serveurs existant.

```
docker run -d --name router --net host mongo:3.2 mongos  --port 30000 \
     --configdb config-rs/$(hostname):29999
```

L'option `--configdb` indique au routeur quel est le *replica set* en charge des données de configuration. Notez qu'il faut spécifier le nom du *replica set* et l'un de nœuds (ici, notre serveur précédent, celui qui est en écoute sur le port 29999). Tout le monde suit ? Sinon relisez la partie sur l'architecture, ci-dessus.

Et finalement, lançons nos deux serveurs de stockage (qui devraient être, dans un système en production, des *replica sets* constitués de 3 nœuds).

```
docker run -d --name mongo1 --net host mongo:3.2 mongod --shardsvr --replSet rs1␣
↪--port 30001
docker run -d --name mongo2 --net host mongo:3.2 mongod --shardsvr --replSet rs2␣
↪--port 30002
```

Pour chacun, on utilise les options :

- `--shardsvr` pour spécifier qu'il s'agit d'un serveur de stockage de fragments.
- `--replSet` pour donner un nom au *replica set*.
- et bien sûr, on les lance sur des ports dédiés dans le réseau de la machine hôte.

Pour chacun, il faut également intitialiser le *replica set* en se connectant aux ports 30001 et 30002 avec un client et en lançant :

```
rs.initiate()
```

Notre système minimal est en place. Encore une fois, en production, il faudrait utiliser plusieurs serveurs pour chaque *replica set*, mais pas forcément un serveur par nœud. Reportez-vous à la documentation MongoDB ou à des experts si vous êtes un jour confrontés à cette tâche.

La commande `docker ps`, ou l'affichage de Kitematic, devrait vous donner la liste de vos quatre conteneurs. Jetez un œil à la sortie console (c'est très facile avec Kitematic) pour vérifier qu'il n'y a pas de message d'erreur.

Il reste à déclarer quels sont les serveurs de fragments. Cette déclaration se fait en se connectant au routeur, sur le port 30000 dans notre configuration. Une fois connecté au routeur, la commande `sh.addShard()` ajoute un *replica set* de fragments.

Voici donc les commandes, à effectuer avec un client connecté au port 30000 de la machine Docker.

```
sh.addShard("rs1/<votremachine>:30001")
sh.addShard("rs2/<votremachine>:30002")
```

Note : Bien entendu `votremachine` dans la commande précédente désigne l'IP ou le nom de votre ordinateur.

Votre configuration est terminée. La commande `sh.status()` devrait vous donner des informations sur le statut de votre système distribué. Vous devriez notamment obtenir la liste des *replica sets*.

```
{"shards": [
  {  "_id" : "rs1",  "host" : "rs1/192.168.99.100:30001" },
  {  "_id" : "rs2",  "host" : "rs2/192.168.99.100:30002" }
  ]
}
```

Si quelques chose ne marche pas, c'est très probablement que vous avez fait une erreur quelque part. J'ai testé et retesté les commandes qui précèdent mais, bien entendu, votre environnement est sans doute différent. C'est une bonne opportunité pour essayer de comprendre ce qui ne va pas, et du coup (une fois les problèmes résolus) pour approfondir la compréhension des différentes commandes.

Partitionnement des collections

Nous avons maintenant un *cluster* MongoDB prêt à partitionner des collections. Ce partitionnement n'est pas obligatoire : une base est souvent constituée de petites collections pour lesquelles un partitionnement est inutile, et d'une très grosse collection laquelle cela vaut au contraire la peine.

Dans MongoDB, c'est au niveau de la *collection* que l'on choisit ou non de partitionner. Par défaut, une collection reste stockée dans un seul fragment sur un seul serveur. Pour qu'une collection puisse être partitionnée, il faut que la base de données qui la contient l'autorise (oui, c'est un peu compliqué…). Autorisons donc la base `nfe204` à contenir des collections partitionnées.

```
mongos> sh.enableSharding("nfe204")
```

Nous sommes enfin prêts à passer à l'échelle pour les collections de la base `nfe204`. Pour vérifier où vous en êtes, les commandes suivantes sont instructives (toujours avec un client connecté au routeur sur le port 27017).

```
db.adminCommand( { listShards: 1 } )
sh.status()
db.stats()
db.printShardingStatus()
```

À vous d'interpréter toutes les informations données.

Nous allons finalement partitionner la collection `movies` avec la commande `shardCollection()`. La question essentielle à se poser est celle de la clé de partitionnement. Par défaut c'est l'identifiant du document qui est choisi, ce qui garantit que le système pourra distinguer les documents individuellement et sera donc en mesure de gérer finement la distribution.

Important : Dans un partitionnement par intervalle, utiliser une clé dont la valeur croît de manière monotone présente un inconvénient fort : *toutes les insertions se font dans le dernier fragment créé*. Pour des écritures intensives, c'est un problème car cela revient à surcharger le serveur qui stocke ce fragment. Dans ce cas il vaut mieux utiliser le partitionnement par hachage.

Il faut être bien conscient que la clé de partitionnement sert *aussi* de clé de recherche. Une recherche donnant comme critère la valeur de la clé sera routée directement vers le serveur contenant le fragment stockant le document. Toute recherche portant sur d'autres critères se fera par parcours séquentiel sur l'ensemble des nœuds.

Voici la commande pour partitionner sur l'identifiant :

```
sh.shardCollection("nfe204.movies", { "_id": 1 } )
```

On peut aussi utiliser un ou plusieurs attributs des documents, par exemple le titre en priorité, et l'année pour distinguer deux films ayant le même titre.

```
sh.shardCollection("nfe204.movies", { "title": 1, "year": 1} )
```

Avec le second choix, on aura donc des recherches sur le titre ou la combinaison (titre, année) très rapides (mais pas sur l'année toute seule !). En revanche une recherche sur l'identifiant se fera par parcours séquentiel généralisé, sauf à créer des index secondaires sur les serveurs de fragment. Bref, il faut bien réfléchir aux besoins de l'application avant de choisir la clé, qui ne peut être changée à postériori. La documentation de MongoDB est assez détaillée sur ce point.

Pour constater l'effet du partitionnement, il nous faut une base d'une taille suffisante pour créer plusieurs fragments et déclencher l'équilibrage sur nos deux serveurs. Le plus simple est d'utiliser un générateur de données : ipsum est un utilitaire écrit en Python est spécifiquement conçu pour MongoDB. J'ai fait quelques adaptations pour que cela fonctionne en python3, et je vous invite donc à récupérer l'archive suivante :

— le code révisé pour python3

Note : Vous devez installer l'extension `pymongo` de Python pour vous connecter à MongoDB. En principe c'est aussi simple que `pip install pymongo`, mais si ça ne marche pas reportez-vous à http://api.mongodb.org/python.

Décompressez l'archive ZIP. Ipsum produit des documents JSON conformes à un schéma (http://json-schema.org). Pour notre base `movies`, nous utilisons le schéma JSON des documents qui est déjà placé avec les fichiers `ipsum`. La commande suivante (il faut se placer dans le répertoire `ipsum-master`) charge 100 000 pseudo-films dans la collection `movies`.

```
python ./pymonipsum.py --host <votremachine> -d nfe204 -c movies --count 1000000␣
↪movies.jsch
```

Cela peut prendre un certain temps… Pendant l'attente occupez-vous en consultant les messages apparaissant à la console pour le routeur et les deux serveurs de fragments (le serveur de configuration est moins bavard) et essayez de comprendre ce qui se passe.

Avec l'interpréteur `mongo`, on peut aussi surveiller l'évolution du nombre de documents dans la base.

```
mongos> db.movies.count()
```

Essayez d'ailleurs la même chose en vous connectant à l'un des serveurs de fragments.

```
mongo nfe204 --port 30001
mongo> db.movies.count()
```

Qu'en dites-vous ? Pendant le chargement, et à la fin de celui-ci, vous pouvez inspecter le statut du partitionnement avec les commandes données précédemment. Tout y est, et c'est relativement clair !

13.2.3 Quiz

13.2.4 Mise en pratique

Exercice MEP-S2-1 : Créer une collection partitionnée

Cet exercice consiste simplement à reproduire les commandes données ci-dessus pour partitionner une collection `movies` dans laquelle on insère quelques milliers de pseudo-documents. Si vous êtes en groupe et disposez de plusieurs serveurs, n'hésitez pas à faire du *vrai* distribué.

Exercice MEP-S2-2 : pour aller plus loin (atelier optionnel)

Cet exercice consiste simplement à reproduire les commandes données ci-dessus pour partitionner une collection `movies` dans laquelle on insère quelques milliers de pseudo-documents. Si vous êtes en groupe et disposez de plusieurs serveurs, n'hésitez pas à faire du *vrai* distribué.

Vous pouvez tenter ensuite quelques variantes et compléments.

— Définissez comme clé de partitionnement le titre, puis le genre du film, que constate-t-on ?
— Créez une collection avec quelques millions de films ; effectuez quelques requêtes, sur la clé, puis sur un autre attribut. Conclusion ? Comment faire pour obtenir de bonnes performances dans le second cas ?
— Essayez d'insérer dans une collection partitionnée en vous adressant directement à l'un des serveurs de stockage.
— Accédez à la base `config` avec `use config`; regardez les collections de cette base : ce sont les méta-données qui décrivent l'ensemble du système distribué. Vous pouvez interroger ces collections pour comprendre en quoi consistent ces méta-données.

13.3 S3 : partitionnement par hachage

Supports complémentaires :

— Diapositives: partitionnement par hachage
— Vidéo de démonstration du partitionnement par hachage

Le partitionnement par hachage en distribué repose globalement sur la même organisation que pour le partitionnement par intervalle. Un *routeur* maintient une structure qui guide l'affectation des documents aux serveurs de stockage, chaque serveur étant localement en charge de gérer le fragment qui lui est alloué. Cette structure au niveau du routage est la *table de hachage* établissant une correspondance entre les valeurs des clés et les adresses des fragments.

La difficulté du hachage est la dynamicité : ajout, suppression de serveur, évolution de la taille de la collection gérée.

13.3.1 Structure et opérations

L'idée de base est de disposer d'une table de correspondance (dite *table de hachage*) entre une suite d'entiers *[1, n]* et les adresses des *n* fragments, et de définir une fonction *h* (dite *fonction de chachage*) associant toute valeur d'identifiant à un entier compris entre 1 et *n*. La fonction de hachage est en principe extrêmement rapide ; associée à une recherche efficace dans la table de hachage, elle permet de trouver directement le fragment correspondant à une clé.

La structure de routage comprend la table de hachage et la fonction *h()*. Pour éviter d'entrer dans des détails compliqués, on va supposer pour l'instant que *h()* est le reste de la division par *n*, le nombre de fragments (fonction modulo de *n*) et que chaque identifiant est un entier. Il est assez facile en pratique de se ramener à cette situation, en prenant quelques précautions pour la fonction soit équitablement distribuée sur *[0, n-1]*.

Note : Si on prend la fonction modulo, le domaine d'arrivée est *[0, n-1]* et pas *[1, n]*, ce qui ne change rien dans le principe.

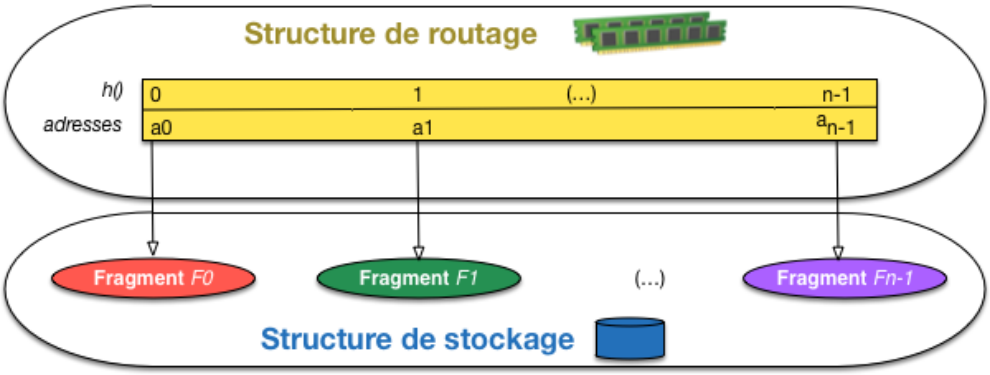


Fig. 13.10 – Partitionnement par hachage

En se basant sur l'illustration de la Fig. 13.10, on voit que tous les documents dont l'identifiant est de la forme $n \times k + r$, où k est un entier, seront stockés dans le fragment F_r. Le fragment F_1 par exemple contient les documents d'identifiant 1, *n+1*, *2n+ 1*, etc.

La table de routage contient des entrées $[i, a_i]$, où $i \in [0, n-1]$, et a_i est l'adresse du fragment F_i. En ce qui concerne sa taille, le même raisonnement s'applique que dans le cas des intervalles : elle est proportionnelle au nombre de fragments, et tient en mémoire même pour des collections extrêmement grandes.

Les opérations s'implantent de la manière suivante :

- *get(i)* : calculer $r = h(i)$, et accéder au fragment dont l'adresse est a_r, chercher le document en mémoire ;
- *put(i, d)* : calculer $r = h(i)$, insérer *d* dans le fragment dont l'adresse est a_r ;
- *delete(i)* : comme la recherche, avec effacement du document trouvé ;
- *range(i, j)* : *pas possible avec une structure par hachage*, il faut faire un parcours séquentiel complet.

Le hachage ne permet pas les recherches par intervalle, ce qui peut être contrariant. En contepartie, la distribution des documents ne dépend pas de la valeur directe de la clé, mais de la valeur de hachage, ce qui garantit une distribution uniforme sans phénomène de regroupement des documents dont les valeurs de clé sont proches. Un tel phénomène peut être intempestif ou souhaitable selon l'application.

Dynamicité

C'est ici que les choses se compliquent. Contrairement aux structures basées sur le tri qui disposent de la méthode de partitionnement pour évoluer gracieusement avec la collection indexée, le hachage (dans la version basique présentée ci-dessus) a un caractère monolithique qui le rend impropre à l'ajout ou à la suppression de fragments.

Tout repose en effet sur l'hypothèse que la fonction *h()* est *immuable*. Un simple contre-exemple suffit pour s'en convaincre. Supposons un flux continu d'insertion et de recherche de documents, parmi lesquelles l'insertion, suivi de la recherche de l'identifiant 17. Pour être totalement concret, on va prendre, initialement, un nombre de fragments *n=5*.

1. quand on effectue *put(17, d)*, la fonction de hachage affecte *d* au fragment F_2 (tout le monde suit ?) ;
2. les insertions continuent, jusqu'à la nécessité d'ajouter un sixième fragment : la fonction de hachage n'est plus $\mod 5$ mais $\mod 6$.
3. je veux effectuer une recherche *get(17)*, mais la nouvelle fonction de hachage m'envoie vers le fragment F_5 (vous voyez pourquoi ?) qui ne contient pas le document recherché.

Un peu de réflexion (en fait, beaucoup de gens très intelligents y ont longuement réfléchi) suffit pour réaliser qu'il n'existe pas de solution simple au problème de l'évolution d'une structure de hachage. Des méthodes sophistiquées ont été proposées, la plus élégante et efficace étant le *hachage linéaire* (W. Litwin) dont la présentation dépasse le cadre de ce document.

Note : Reportez-vous au cours http://sys.bdpedia.fr, au livre http://webdam.inria.fr/Jorge/ ou à toute autre source bien informée pour tout savoir sur le hachage dynamique en général, linéaire en particulier.

Voyons dans le cadre d'un système distribué comment appliquer le principe du hachage avec dynamicité.

13.3.2 Le hachage cohérent (*consistent hashing*)

Le hachage repose donc sur une fonction *h()* qui distribue les valeurs de clé vers un intervalle *[0, n-1]*, *n* correspondant au nombre de fragments. Toute modification de cette fonction rend invalide la distribution existante, et on se trouve donc à priori dans la situation douloureuse de conserver *ad vitam* le nombre de fragments initial, ou d'accepter périodiquement des réorganisation entière du partitionnement.

Le *hachage cohérent* propose une solution qui évite ces deux écueils en maintenant *toujours* la même fonction tout en adaptant la régle d'affectation d'un document à un serveur selon l'évolution (ajout / suppression) de la grappe. Cette règle d'affectation maintient la *cohérence* globale du partitionnement déjà effectué, d'où le nom de la méthode, et surtout son intérêt.

L'anneau et la règle d'affectation

Le principe du hachage cohérent est de considérer dès le départ un intervalle immuable *D = [0, n-1]* pour le domaine d'arrivée de la fonction de hachage, où *n* est choisi assez grand pour réduire le nombre de collisions (une *collision*, quand on parle de hachage, correspond à deux valeurs de clé distinctes i_1 et i_2 telles que $h(i_1) = h(i_2)$). On choisit typiquement $n = 2^{32}$ ou $n = 2^{64}$, ce qui permet de représenter la table de hachage avec un indice stocké sur 4 ou 8 octets.

On interprète ce domaine comme un anneau parcouru dans le sens des aiguilles d'une montre, de telle sorte que le successeur de $2^{64} - 1$ est 0. La fonction de hachage associe donc chaque serveur de la grappe à une position sur l'anneau ; on peut par exemple prendre l'adresse IP d'un serveur, la convertir en entier et appliquer $f(ip) = ip \mod 2^{64}$, ou tout autre transformation vers *D* suffisamment distributive.

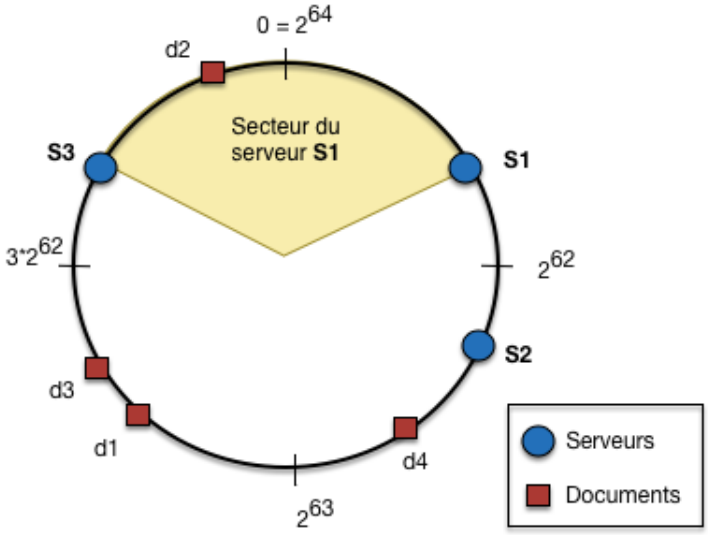


Fig. 13.11 – L'anneau du hachage cohérent et la règle d'affectation

On peut observer que le placement des serveurs sur l'anneau découpe ce dernier en arcs de cercle (Fig. 13.11). La règle d'affectation est alors définie de la manière suivante : *chaque serveur est en charge de l'arc de cercle qui le précède sur l'anneau*. Si on regarde plus précisément la Fig. 13.11 :

- le serveur S_1 est positionné par la fonction de hachage en $h(S_1) = a$, *a* étant quelque part entre 0 et 2^{62} ;
- le serveur S_2 est positionné par la fonction de hachage en $h(S_2) = b$, quelque part entre 2^{62} et 2^{63} ;
- le serveur S_3 est positionné par la fonction de hachage en $h(S_3) = c$, quelque part entre 3×2^{62} et $2^{64} - 1$.

S_1 est donc responsable de l'arc qui le précède, jusqu'à la position de S_3 (non comprise). Maintenant, les documents sont eux aussi positionnés sur cet anneau par une fonction de hachage ayant le même domaine d'arrivée que *h()*. La règle d'affectation s'ensuit : chaque serveur doit stocker le fragment de la collection correspondant aux objets positionnés sur l'arc de cercle dont il est responsable.

Note : On pourrait bien entendu également adopter la convention qu'un serveur est responsable de l'arc de cercle *suivant* sa position sur l'anneau (au lieu du précédent). Cela ne change évidemment rien au principe.

Sur la figure, S_1 stockera donc D2, S_3 stockera d1, d3, d4 et S_2 ne stockera (pour l'instant) rien du tout.

En pratique

La table de hachage est un peu particulière : elle établit une correspondance entre le découpage de l'anneau en arcs de cercle, et l'association de chaque arc à un serveur. Toujours en notant a, b et c les positions respectives de nos trois serveurs, on obtient la table suivante.

h(i)	Serveur
]c, a]	*S1*
]a, b]	*S2*
]b, c]	*S3*

Le fait de représenter des intervalles au lieu de valeurs ponctuelles est la clé pour limiter la taille de la table de hachage (qui contient virtuellement 2^{64} positions).

Un premier problème pratique apparaît immédiatement : les positions des serveurs étant déterminées par la fonction de hachage indépendamment de la distribution des données, certains serveurs se voient affecter un tout petit secteur, et d'autres un très grand. C'est flagrant sur notre Fig. 13.11 où le déséquilibre entre S_2 et S_3 est très accentué, au bénéfice (ou au détriment…) de ce dernier.

La solution est d'affecter à chaque serveur non pas en une, mais en plusieurs positions sur l'anneau, ce qui tend à multiplier les arcs de cercles et, par un effet d'uniformisation, de rendre leurs tailles comparables. L'effet est illustré avec un nombre très faible de positions (3 pour chaque serveur) sur la Fig. 13.12. L'anneau est maintenant découpé en 9 arcs de cercles et les tailles tendent à s'égaliser.

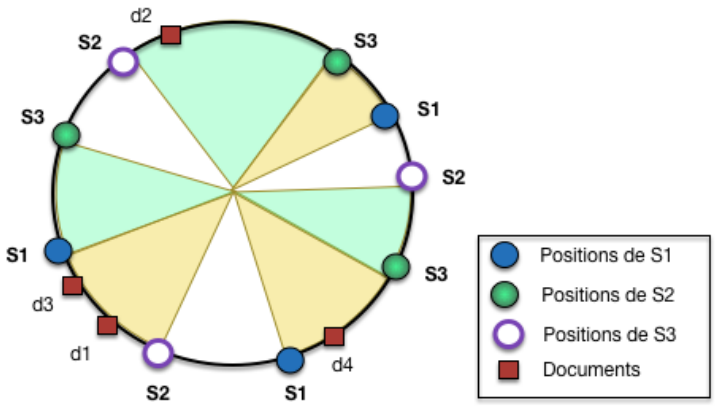


Fig. 13.12 – Positions multiples de chaque serveur sur l'anneau

En pratique, on peut distribuer un même serveur sur plusieurs dizaines de positions (128, 256, typiquement) pour garantir cet effet de lissage. Cela a également pour impact d'agrandir la taille de la table de routage. Celle donnée ci-dessous correspond à l'état de la Fig. 13.12, où a1, a2 et a3 représentent les positions de S_1, et ainsi de suite.

h(i)	Serveur
]c1, a1]	*S1*
]a1, b1]	*S2*
]b1, c2]	*S3*
]c2, a2]	*S1*
]a2, b2]	*S2*
]b2, a3]	*S1*
]a3, c3]	*S3*
]c3, b3]	*S2*
]b3, c1]	*S3*

La taille de la table de routage peut éventuellement devenir un souci, surtout en cas de modifications fréquentes (ajout ou suppression de serveur). C'est surtout valable pour des réseaux de type pair-à-pair, beaucoup moins pour des grappes de serveurs d'entreprises, beaucoup plus stables. Des solutions existent pour diminuer la taille de la table de hachage, avec un routage des requêtes un peu plus compliqué. Le plus connu est sans doute le protocole Chord ; vous pouvez aussi vous reporter à http://webdam.inria.fr/Jorge/.

Ajout/suppression de serveurs

L'ajout d'un nouveau serveur ne nécessite que des adaptations *locales* de la structure de hachage, contrairement à une approche basée sur le changement de la fonction de hachage, qui implique une recontruction complète de la structure. Quand un nouveau serveur est ajouté, ses nouvelles positions sont calculées, et chaque insertion à une position implique une division d'un arc de cercle existant. La Fig. 13.13 montre la situation avec une seule position par serveur pour plus de clarté.

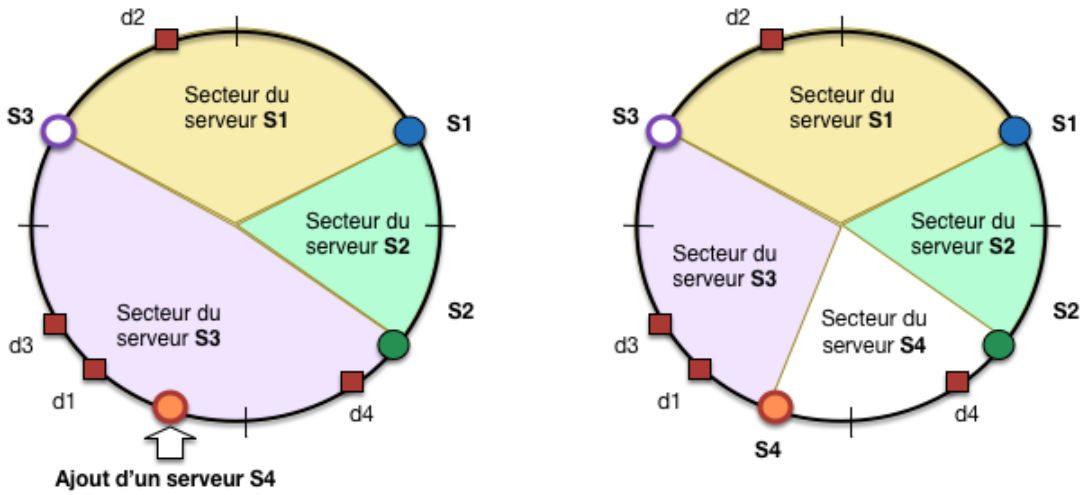


Fig. 13.13 – Ajout d'un nouveau serveur

Un serveur S_4 est ajouté (partie gauche de la figure) dans un arc de cercle existant, celui associé jusqu'à présent au serveur S_3. Une partie des documents gérés par ce dernier (ici, d4) doit donc migrer sur le nouveau serveur. C'est assez comparable avec l'éclatement d'un partitionnement par intervalle, la principale

différence avec le hachage étant que, le positionnement résultant d'un calcul, il n'y a aucune garantie que le fragment existant soit divisé équitablement. Statistiquement, la multiplication des serveurs et surtout de leurs positions doit aboutir à un partitionnement équitable.

Note : Notez au passage que plus un arc est grand, plus il a de chance d'être divisé par l'ajout d'un nouveau serveur, ce qui soulage d'autant le serveur en charge du fragment initial. C'est la même constatation qui pousse à multiplier le nombre de positions pour un même serveur.

13.3.3 Cassandra en mode distribué

Ressources complémentaires

— Sur le *Hash Ring* de Cassandra, un document concis et assez précis, http://salsahpc.indiana.edu/b534projects/sites/default/files/public/1_Cassandra_Gala,%20Dhairya%20Mahendra.pdf

---

L'architecture distribuée de Cassandra est basée sur le *consistent hashing*, et fortement inspirée de la conception du système Dynamo.

Note : Cette partie s'appuie largement sur une contribution de Guillaume Payen, issue de son projet NFE204. Merci à lui !

Le *Hash-Ring*

Les nœuds sont donc affectés à un anneau directionnel, ou *Hash Ring* couvrant les valeurs $[-2^{63}, 2^{63}]$. Lorsque l'on ajoute un nouveau nœud dans le cluster, ce dernier vient s'ajouter à l'anneau. C'est notamment à partir de cette caractéristique qu'une phrase est souvent reprise dans la littérature lorsqu'il s'agit de faire de la réplication avec Cassandra : *Just add a node !* Rien de nouveau ici : c'est l'architecture présentée initialement par le système Dynamo (Amazon).

Chaque nœud *n* est positionné sur l'anneau à un emplacement (ou *token*) qui peut être obtenu de deux manières :

— Soit, explicitement, par l'administrateur du système. Cette méthode peut être utile quand on veut contrôler le positionnement des serveurs parce qu'ils diffèrent en capacité. On placera par exemple un serveur peu puissant de manière à ce que l'intervalle dont il est responsable soit plus petit que ceux des autres serveurs.

— Soit en laissant Cassandra appliquer la fonction de hachage (par défaut, un algorithme nommé MurMur3, plusieurs autres choix sont possibles).

Le serveur *n* obtient un token t_n. Il devient alors responsable de l'intervalle de valeurs de hachage sur l'anneau $]t_{n-1}, t_n]$. Au moment d'une insertion, la fonction de hachage est appliquée à la clé primaire de la ligne, et le résultat détermine le serveur sur lequel la ligne est insérée.

Pour chaque nœud physique, il est possible d'obtenir plusieurs positions sur l'anneau (principe des nœuds dits « virtuels »), et donc plusieurs intervalles dont le nœud (physique) est responsable. La configuration du

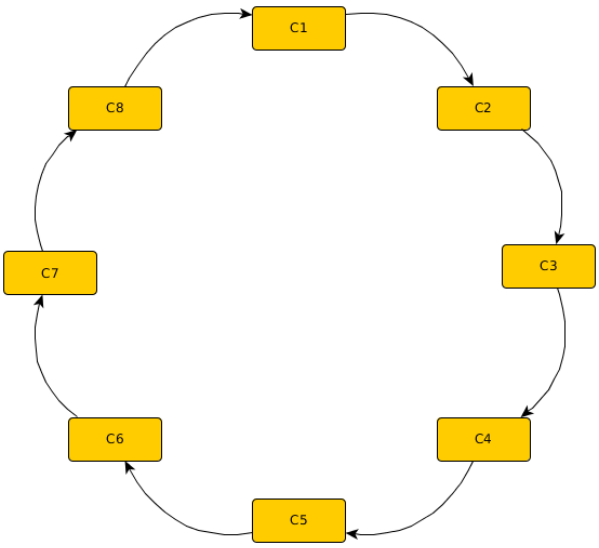


Fig. 13.14 – Représentation d'un cluster Cassandra avec le *Hash Ring*

nombre de nœuds virtuels est donnée par le paramètre `num_token` du fichier de configuration `cassandra.yaml`.

Certains nœuds jouent le rôle de points d'entrée dans l'anneau, et sont nommés *seed* (« graine », « semence ») dans Cassandra. En revanche, tous les nœuds peuvent répondre à des requêtes des applications clients. La table de routage est en effet dupliquée sur tous les nœuds, ce qui permet donc à chaque nœud de rediriger directement toute requête vers le nœud capable de répondre à cette requête. Pour cela, les nœuds d'une grappe Cassandra sont en intercommunication permanente, afin de détecter les ajouts ou départs (pannes) et les refléter dans leur version de la table de routage stockée localement.

Routage des requêtes

Un *cluster* Cassandra fonctionne en mode multi-nœuds. La notion de nœud maître et nœud esclave n'existe donc pas. Chaque nœud du cluster a le même rôle et la même importance, et jouit donc de la capacité de lecture et d'écriture dans le cluster. Un nœud ne sera donc jamais préféré à un autre pour être interrogé par le client.

Pour que ce système fonctionne, chaque nœud du *cluster* a la connaissance de la topologie de l'anneau. Chaque nœud sait donc où sont les autres nœuds, quels sont leurs identifiants, quels nœuds sont disponibles et lesquels ne le sont pas.

Un client qui interroge Cassandra contacte un nœud au hasard parmi tous les nœuds du cluster. Le partitionnement implique que tous les nœuds ne possèdent pas localement l'information recherchée. Cependant, tous les nœuds sont capables de dire quel est le nœud du cluster qui possède la ressource recherchée.

Note : Le rôle du coordinateur est donc dans ce cas légèrement différent de ce que nous avons présenté dans le chapitre précédent. Au lieu de se charger lui-même d'une écriture locale, puis de transmettre des demandes de réplication, le coordinateur envoie f demandes d'écriture en parallèle à f nœuds de l'anneau, où f est le

facteur de réplication.

Stratégies de réplication

Cassandra peut tenir compte de la topologie du cluster pour gérer les réplications. Avec la stratégie simple, tout part de l'anneau. Considérons un cluster composé de 8 nœuds, c1 à c8, et un facteur de réplication de 3. Comme expliqué précédemment, n'importe quel nœud peut recevoir la requête du client. Ce nœud, que l'on nommera *coordinateur* va prendre en compte

- la méthode de hachage,
- les *token range* (intervalles représentant les arcs de cercle affectés à chaque serveur) des nœuds du cluster
- la clé du document inséré

pour décider quel sera le nœud dans lequel ce dernier sera stocké. Le coordinateur va alors rediriger la requête pour une écriture sur le nœud choisi par la fonction de hachage. Comme le facteur de réplication est de 3, le coordinateur va aussi rediriger la requête d'écriture vers les 2 nœuds suivant le nœud choisi, dans le sens de l'anneau.

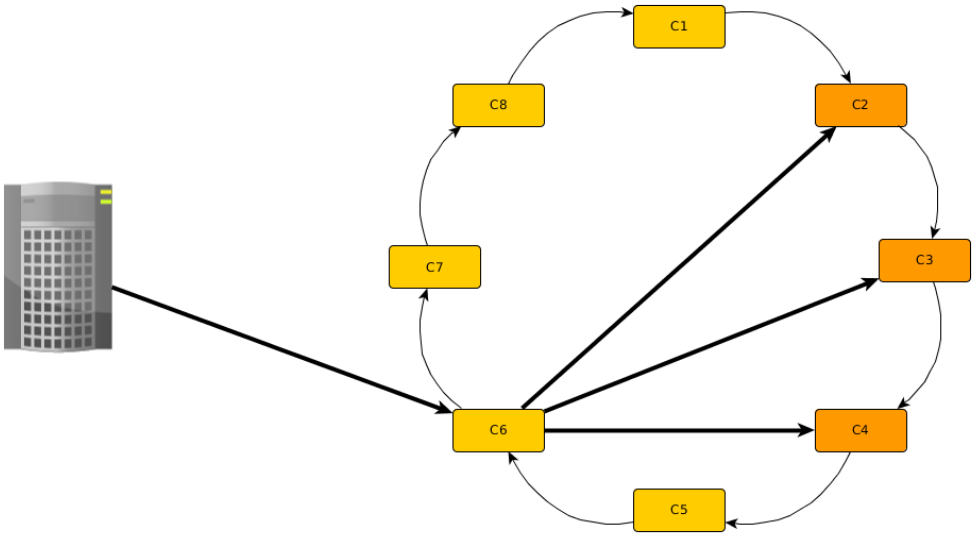


Fig. 13.15 – Stratégie de réplication simple

Comme on le voit dans la Fig. 13.15, lorsque le client effectue la requête sur le cluster, c'est le nœud c6 auquel le client s'est adressé pour traiter la demande. Ce dernier calcule que c'est le nœud c2 qui doit être sollicité pour traiter la requête. Il va donc rediriger la requête vers c2, mais également vers c3 et c4. Ce schéma vaut aussi bien pour la lecture que pour l'écriture.

La stratégie par topologie du réseau présente un intérêt lorsque l'infrastructure est répartie sur différents clusters. Ces derniers peuvent être éloignés physiquement, ou dans le même local. Avec cette stratégie, Cassandra adopte (par défaut) les principes suivants :

- les données sont répliquées dans le *même data center*, pour éviter le coût réseau des transferts d'un centre à un autre
- la réplication se fait sur des serveurs situés dans des baies distinctes, car deux serveurs d'une même baie ont plus de chance d'être indisponibles ensemble en cas de panne réseau affectant la baie.

Cette stratégie est intéressante pour des ressources localisées dans différents endroits du monde. L'architecture est toujours celle d'un anneau directionnel, chaque nœud étant lié au nœud suivant. L'écriture d'un document va se faire de la manière suivante :

— on détermine le nœud N en charge du secteur contenant la valeur hachée de la clé
— on parcourt ensuite l'anneau jusqu'à trouver situés dans le *même* centre de données que *N*, sur lequels on effectue alors la réplication.

N définit donc le centre de données dans lequel le document sera inséré.

Mise en pratique

Voici un exemple de mise en pratique pour tester le fonctionnement d'un *cluster* Cassandra et quelques options. Pour aller plus lon, vous pouvez recourir à l'un des tutoriaux de Datastax, par exemple http://docs.datastax.com/en/cql/3.3/cql/cql_using/useTracing.html pour inspecter le fonctionnement des niveaux de cohérence.

Notre *cluster*

Créons maintenant un cluster Cassandra, avec 5 nœuds. Pour cela, nous créons un premier nœud qui nous servira de point d'accès (*seed* dans la terminologie Cassandra) pour en ajouter d'autres.

```
docker run -d -e "CASSANDRA_TOKEN=1" \
     --name cass1 -p 3000:9042 spotify/cassandra:cluster
```

Notez que nous indiquons explicitement le placement du serveur sur l'anneau. En production, il est préférable de recourir aux nœuds virtuels, comme expliqué précédemment. Cela demande un peu de configuration, et nous allons nous contenter d'une exploration simple ici.

Il nous faut l'adresse IP de ce premier serveur. La commande suivant extrait l'information `NetworkSettings.IPAddress` du document JSON renvoyé par l'instruction `inspect`.

```
docker inspect -f '{{.NetworkSettings.IPAddress}}' cass1
```

Vous obtenez une adresse. Par la suite on supppose qu'elle vaut 172.17.0.2.

Créons les autres serveurs, en indiquant le premier comme serveur-*seed*.

```
docker run -d -e "CASSANDRA_TOKEN=10" -e "CASSANDRA_SEEDS=172.17.0.2" \
    --name cass2 spotify/cassandra:cluster

docker run -d -e "CASSANDRA_TOKEN=100" -e "CASSANDRA_SEEDS=172.17.0.2" \
    --name cass3 spotify/cassandra:cluster

docker run -d -e "CASSANDRA_TOKEN=1000" -e "CASSANDRA_SEEDS=172.17.0.2" \
    --name cass4 spotify/cassandra:cluster

docker run -d -e "CASSANDRA_TOKEN=10000" -e "CASSANDRA_SEEDS=172.17.0.2" \
    --name cass5 spotify/cassandra:cluster
```

Nous venons de créer un cluster de 5 nœuds Cassandra, qui tournent tous en tâche de fond grâce à Docker.

Keyspace et données

Insérons maintenant des données. Vous pouvez utiliser le client *DevCenter*. À l'usage, il est peut être plus rapide de lancer directement l'interpréteur de commandes sur l'un des nœuds avec la commande :

```
docker exec -it cass1 /bin/bash
[docker]$ cqlsh 172.17.0.X
```

Créez un *keyspace*.

```
CREATE keyspace repli
         with replication = {'class':'SimpleStrategy', 'replication_factor':3};
USE repli;
```

Insérons un document.

```
CREATE TABLE data (id int, value text, PRIMARY KEY (id));
INSERT INTO data (id, value) VALUES (10, 'Premier document');
```

Nous venons de créer un *keyspace*, qui va répliquer les données sur 3 nœuds. La table *data* va utiliser la clé primaire *id* et la fonction de hashage du *partitioner* pour stocker le document dans l'un des 5 nœuds, puis répliquer dans les 2 nœuds suivants sur l'anneau. Il est possible d'obtenir avec la fonction *token()* la valeur de hachage pour la clé des documents.

```
select token(id), id from data;
```

Vérifions avec l'utilitaire *nodetool* que le cluster est bien composé de 5 nœuds, et regardons comment chaque nœud a été réparti sur l'anneau. On s'attend à ce que les nœuds soient placés par ordre croissant de leur identifiant.

```
docker exec -it cass1 /bin/bash
[docker]$ /usr/bin/nodetool ring
```

Testons que le document inséré précedemment a bien été répliqué sur 2 nœuds.

```
docker exec -it cass1 /bin/bash
[docker]$ /usr/bin/nodetool cfstats -h 172.17.0.2 repli
```

Regardez pour chaque nœud la valeur de *Write Count*. Elle devrait être à 1 pour 3 nœuds consécutifs sur l'anneau, et 0 pour les autres. Vérifions maintenant qu'en se connectant à un nœud qui ne contient pas le document, on peut tout de même y accéder. Considérons par exemple que le nœud cass1 ne contient pas le document.

```
docker exec -it cass1 /bin/bash
[docker]$ cqlsh 172.17.0.X
```

(suite sur la page suivante)

(suite de la page précédente)

```
cqlsh > USE repli;
cqlsh:repli > SELECT * FROM data;
```

Cohérence des lectures

Pour étudier la cohérence des données en lecture, nous allons utiliser la ressource stockée, et stopper 2 nœuds Cassandra sur les 3. Pour ce faire, nous allons utiliser Docker. Considérons que la donnée est stockée sur les nœuds cass1, cass2 et cass3

```
docker pause cass2
docker pause cass3
docker exec -it cass1 /bin/bash
[docker]$ /usr/bin/nodetool ring
```

Vérifiez que les nœuds sont bien au statut *Down*.

Nous pouvons maintenant paramétrer le niveau de cohérence des données. Réalisons une requête de lecture. Le système est paramétré pour assurer la meilleure cohérence des données. On s'attend à ce que la requête plante car en mode ALL, Cassandra attend la réponse de tous les nœuds.

```
docker exec -it cass1 /bin/bash
[docker]$ cqlsh 172.17.0.X
cqlsh > use repli;
# devrait renvoyer Consistency level set to ALL.
cqlsh:repli > consistency all;
# devrait renvoyer Unable to complete request: one or more nodes were␣
↪unavailable.
cqlsh:repli > select * from data;
```

Comme attendu, la réponse renvoyée au client est une erreur. Testons maintenant le mode `ONE`, qui devrait normalement renvoyer la ressource du nœud le plus rapide. On s'attend à ce que la ressource du nœud 172.17.0.X soit renvoyée.

```
docker exec -it cass1 /bin/bash
[docker]$ cqlsh 172.17.0.X
cqlsh > use repli;
cqlsh:repli > consistency one;  # devrait renvoyer Consistency level set to ONE.
cqlsh:repli > select * from data;
```

Dans ce schéma, le système est très disponible, mais ne vérifie pas la cohérence des données. Pour preuve, il renvoie effectivement la ressource au client alors que tous les autres nœuds qui contiennent la ressource sont indisponibles (ils pourraient contenir une version pus récente). Enfin, testons la stratégie du quorum. Avec 2 nœuds sur 3 perdus, la requête devrait normalement renvoyer au client une erreur.

```
docker exec -it cass1 /bin/bash
[docker]$ cqlsh 172.17.0.X
```

(suite sur la page suivante)

(suite de la page précédente)

```
cqlsh > use repli;
# devrait renvoyer Consistency level set to QUORUM.
cqlsh:repli > consistency quorum;
# devrait renvoyer Unable to complete request: one or more nodes were␣
↪unavailable.
cqlsh:repli > select * from data;
```

Le résultat obtenu est bien celui attendu. Moins de la moitié des réplicas est disponible, la requête renvoie donc une erreur. Réactivons un nœud, et re-testons.

```
docker unpause cass2
docker exec -it cass1 /bin/bash
[docker]$ nodetool ring
[docker]$ cqlsh 172.17.0.X
cqlsh > use repli;
# devrait renvoyer Consistency level set to QUORUM.
cqlsh:repli > consistency quorum;
cqlsh:repli > select * from data;
```

Lorsque le nœud est réactivé (via Docker), il faut tout de même quelques dizaines de secondes avant qu'il soit effectivement réintégré dans le cluster. Le plus important est que la règle du quorum soit validée, avec 2 nœuds sur 3 disponibles, Cassandra accepte de retourner au client une ressource.

Cassandra & données massives

Cassandra est considéré aujourd'hui comme l'une des bases de données NoSQL les plus performantes dans un environnement Big Data. Lorsque le projet requiert de travailler sur de très gros volumes de données, le défi est de pouvoir écrire les données rapidement. Et sur ce point, Cassandra a su démontrer sa supériorité. Comme vu auparavant, le passage à l'échelle chez Cassandra est très efficace, et donc particulièrement adapté à un environnement où les données sont distribuées sur plusieurs serveurs. Grâce à l'architecture de Cassandra, la distribution implique une maintenance gérable sans être trop lourde, et assure automatiquement une gestion équilibrée des données sur l'ensemble des nœuds.

On pourrait croire que mettre un cluster Cassandra en production se fait en quelques coups de baguette magique. En réalité, l'opération est beaucoup plus délicate. En effet, Cassandra propose une modélisation des données très ouverte, ce qui donne accès à énormément de possibilités, et permet surtout de faire n'importe quoi. Contrairement aux bases de données relationnelles, avec Cassandra, on ne peut pas se contenter de juste stocker des documents. Il faut en effet avoir une connaissance fine des données qui vont être stockées, la manière dont elles seront interrogées, la logique métier qui conditionnera leur répartition sur les différents nœuds. La conception du modèle de données sur Cassandra demande donc une attention particulière, car une modélisation peu performante en production avec des pétaoctets de données donnera des résultats catastrophiques.

Cassandra permet aussi de ne pas contraindre le nombre de paires clé/valeur dans les documents. Lorsqu'un document a beaucoup de valeurs, on parle alors de *wide row*. Les *wide rows* permettent de profiter des possibilités offertes en terme de modélisation. En revanche, plus un document a de valeurs, plus il est lourd. Il faut donc estimer finement à partir de combien de valeurs le modèle va s'écrouler tellement les briques sont

lourdes… N'oublions pas que Cassandra est une base de données NoSQL, et donc le concept de jointures n'existe pas.

Les ressemblances avec le modèle relationnel et particulièrement SQL apportent une aide certaine, particulièrement à ceux qui ont une grosse expérience sur SQL. En revanche, elles peuvent amener les utilisateurs à sous-estimer cette base de données extrêmement riche. Cassandra offre des performances élevées, à condition de concevoir le modèle de données adéquat. Vous trouverez sur Internet nombre d'anecdotes de grosses structures qui se sont cassées les dents avec Cassandra, et qui ont été obligées de refaire intégralement leur modèle de données, et ce plusieurs fois avant de pouvoir enfin toucher du doigt cette performance tant convoitée.

13.3.4 Quiz

13.4 Exercices

Exercice Ex-Sharding-1 : Scalabilité ElasticSearch

Réfléchissons : la taille de notre collection augmente, et nous ajoutons de nouveaux serveurs au *cluster* ElasticSearch.

- À partir de quel nombre de serveurs peut-on soupçonner que le gain devient négligeable ou nul (et donc que la scalabilité n'est pas respectée) ?
- Est-ce la même réponse pour les écritures et les lectures ?
- Que faire alors ?

Répondez en vous basant sur le configuration par défaut, puis en général.

Pour approfondir, vous pouvez vous reporter à la documentation ElasticSearch https://www.elastic.co/guide/en/elasticsearch/guide/current/scale.html. À lire avec l'esprit critique affuté par les leçons du cours NFE204 bien sûr.

Correction

Le nombre de réplicas et de fragments primaires est défini dans la configuration. Appelons-les *R* et *F*. Il y a donc en tout et pour tout $R * F$ fragments (réplication comprise).

- les lectures peuvent accéder à n'importe quel fragment, donc la distribution des lectures s'améliore jusqu'à ce que l'on ait 1 fragment par serveur, soit $R * F$ serveurs.
- en revanche les écritures doivent se faire sur les fragments primaires, et au-delà de F serveurs on ne gagne plus rien.

Il est assez facile d'augmenter le nombre de réplicas, mais pour changer le nombre de fragments il faut reconstruire tout l'index.

Outre la mise en œuvre de Cassandra en exécutant les commandes données précédemment, voici quelques propositions.

Exercice Ex-S3-1 : ajout d'un serveur avec hachage cohérent

La figure *Ajout d'un serveur* montre l'anneau de la figure *Positions multiples de chaque serveur sur l'anneau* avec ajout d'un nouveau serveur S4 en trois positions p1, p2, et p3.

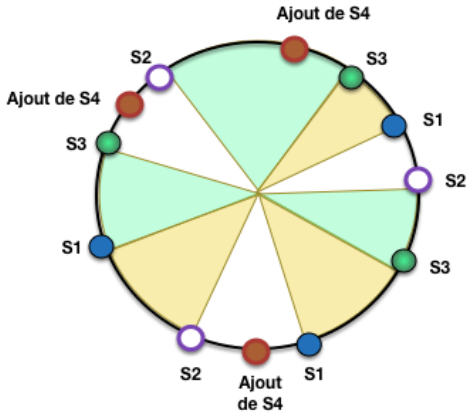


Fig. 13.16 – Ajout d'un serveur

Déterminez la nouvelle table de routage après ajout de S4.

Exercice Ex-S3-2 : les tables de hachage distribuées (DHT), atelier optionnel

Ceux qui ont de l'apétit pour les structures de données sophistiquées peuvent se pencher sur les différentes *tables de hachage distribuées* (DHT pour *distributed hash tables*). Dans cet exercice je vous propose d'explorer une des plus célèbres, *Chord*. C'est une variante du *consistent hashing* dans laquelle, contrairement à Dynamo ou Cassandra, on considère que la table de routage varie trop fréquemment pour pouvoir être synchronisée en permanence sur tous les serveurs. Pour des réseaux pair à pair, c'est une hypothèse pertinente. On va donc limiter fortement sa taille et par là le nombre de mises à jour qu'elle doit subir.

Dans Chord, chaque nœud N_p maintient une table de routage référençant un sous-ensemble des autres nœuds du système, nommé $friends_p$. Ce sous-ensemble contient au plus 64 autres serveurs (pour un espace de hachage de taille 2^{64}). Chaque entrée $i \in [0, 63]$ référence le nœud N_i tel que

— $h(N_i) \geq h(N_p) + 2^{i-1}$
— il n'existe pas de nœud p' tel que $h(N_i) > h(N_{p'}) \geq h(N_p) + 2^{i-1}$

En clair, le nœud N_i est celui dont l'arc de cercle contient la clé $h(N_p) + 2^{i-1}$. Notez que la distance entre les clés couvertes par les « amis » croît de manière exponentielle : elle est de 2 initialement, puis de 4, puis de 8, puis de 16, jusqu'à une distance de 2^{63} correspondant à la moitié de l'anneau !

La Fig. 13.17 illustre la situation pour *m=4*, avec donc $2^4 = 16$ positions sur l'anneau. prenons un nœud S1 placé en position 1. Son premier ami est celui dont l'arc de cercle contient $2^0 = 1$. Son second ami doit contenir la position $2^2 = 2$, son troisième ami la position $2^2 = 4$ et son quatrième et dernier ami la position $2^3 = 8$. ct

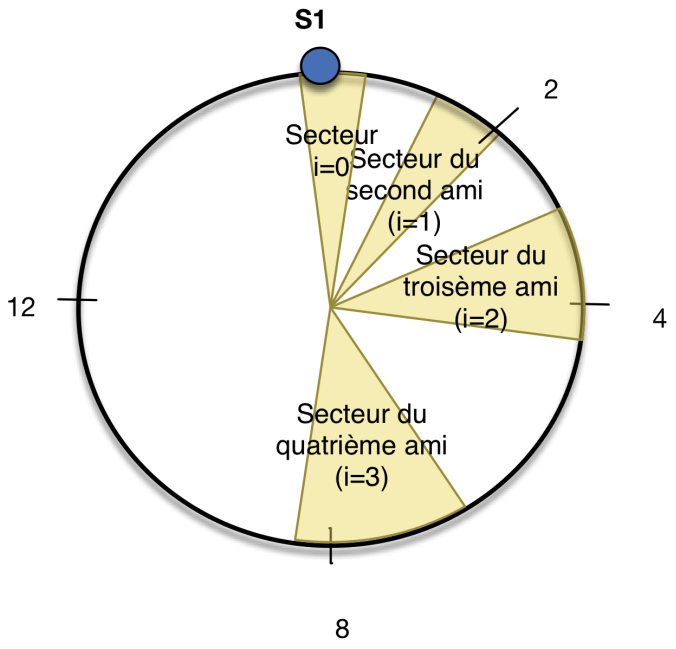


Fig. 13.17 – Illustration de la table de routage dans Chord

On remarque que de larges secteurs de l'anneau sont inconnus, et qu'ils deviennent de plus en plus larges. Après le dernier ami, c'est pratiquement la moitié de l'anneau qui est inconnue. *Contrairement à la table de routage de Cassandra, la table de routage de Chord est petite (sa taille est logarithmique dans le nombre de positions) mais ne permet pas toujours à un nœud de rediriger la requête vers le serveur contenant les données*.

En revanche, et c'est l'idée clé, le nœud a un ami qui est mieux placé. Pourquoi ? Parce que chaque nœud connaît d'autant mieux un secteur qu'il en est proche. Il suffit donc de trouver l'ami le mieux placé pour répondre et lui transmettre la requête.

À partir de là c'est à vous de jouer.

— Copiez la Fig. 13.17 et faites quatre dessins équivalents montrant les amis des amis de *S1* pour *i=2* et *i=3*.
— En supposant que chaque nœud couvre 2 clés, expliquez comment on peut trouver le document de clé *k=4* en s'adressant initialement à *S1*. Même question avec la clé *k=8*.
— Expliquez comment on peut trouver le document de clé *k=6* en s'adressant initialement à *S1*. Même question avec la clé *k=12*.
— Et pour la clé *k=14*, comment faire ? En déduire l'algorithme de recherche,
— Quel est le nombre de redirections de messages qu'il faut effectuer (c'est la *complexité en communication* de l'algorithme).

Vous avez le droit de fouiller sur le web bien sûr, mais l'important est de savoir restranscrire correctement ce que vous aurez trouvé.

Exercice Ex-S3-3 : découverte d'un système basé sur le hachage cohérent (atelier optionnel)

Vous pouvez tester votre capacité à comprendre, installer, tester par vous-même un système distribué en découvrant un des systèmes suivants qui s'appuient sur le hachage cohérent pour la distribution :

— Riak, http://basho.com/riak/
— Redis, http://redis.io/
— Voldemort, http://www.project-voldemort.com/voldemort/
— Memcached, http://memcached.org/

Et sans doute beaucoup d'autres. Objectif : installer, insérer des données, créer plusieurs nœuds, comprendre les choix (architecture maître-esclave ou multi-nœuds, gestion de la cohérence, etc.)

CHAPITRE 14

Etude de cas : Apache Spark

Avec le système Spark, nous récapitulons une bonne partie des sujets abordés dans ce cours. Spark est un environnement dédié au calcul distribué à grande échelle, proposant des fonctionnalités bien plus puissantes que le simple MapReduce des origines, toujours disponible dans l'écosystème Hadoop.

Ces fonctionnalités consistent notamment en un ensemble d'opérateurs de second ordre (voir cette notion dans le chapitre *Le cloud, une nouvelle machine de calcul*) qui étendent considérablement la simple paire constituée du Map et du Reduce. Nous avons eu un aperçu de ces opérateurs avec Pig, qui reste cependant lié à un contexte d'exécution MapReduce (un programme Pig est compilé et exécuté comme une séquence de *jobs* MapReduce).

Entre autres limitations, cela ne couvre pas une classe importante d'algorithmes : ceux qui procèdent par *itérations* sur un résultat progressivement affiné à chaque exécution. Ce type d'algorithme est très fréquent dans le domaine général de la fouille de données : PageRank, *kMeans*, calculs de composantes connexes dans les graphes, etc.

Ce chapitre propose une vision d'ensemble du système Spark, avec des aspects pratiques, et une illustration de son intégration avec un système de stockage distribué comme Cassandra.

14.1 S1 : Introduction à Spark

Supports complémentaires

— Diapositives: Introduction à Spark
— Vidéo d'introduction à Spark

MapReduce repose sur un mécanisme de progression consistant à écrire sur disque les résultats intermédiaires. En présence de chaînes de traitement complexes, incluant parfois des itérations sur une même source

de données, ce mécanisme de sérialisation/désérialisation sur disque devient extrêmement pénalisant pour les performances.

Dans Spark, la méthode est très différente. Elle consiste à placer ces jeux de données en mémoire RAM et à éviter la pénalité des écritures sur le disque. Le défi est alors bien sûr de proposer une reprise sur panne automatique efficace.

14.1.1 Architecture système

Spark est un *framework* qui coordonne l'exécution de *tâches* sur des *données* en les répartissant au sein d'un *cluster* de machines. Il est voulu comme extrêmement modulaire et flexible. Le programmeur envoie au *framework* des *Spark Applications*, pour lesquelles Spark affecte des ressources (RAM, CPU) du cluster en vue de leur exécution. Une application Spark se compose d'un processus *driver* et d'*executors*. Le *driver* est essentiel pour l'application car il exécute la fonction *main()* et est responsable de 3 choses :

— conserver les informations relatives à l'application ;
— répondre aux saisies utilisateur ou aux demandes de programmes externes ;
— analyser, distribuer et ordonnancer les tâches (cf plus loin).

Un *executor* n'est responsable que de 2 choses : exécuter le code qui lui est assigné par le *driver* et lui rapporter l'état d'avancement de la tâche.

Le *driver* est accessible programmatiquement par un point d'entrée appelé *SparkSession*, que l'on trouve derrière une variable `spark`.

La figure Fig. 14.1 illustre l'architecture système de Spark. Dans cet exemple il y a un *driver* et 4 *executors*. La notion de nœud dans le cluster est absente : les utilisateurs peuvent configurer combien d'exécutors reposent sur chaque nœud.

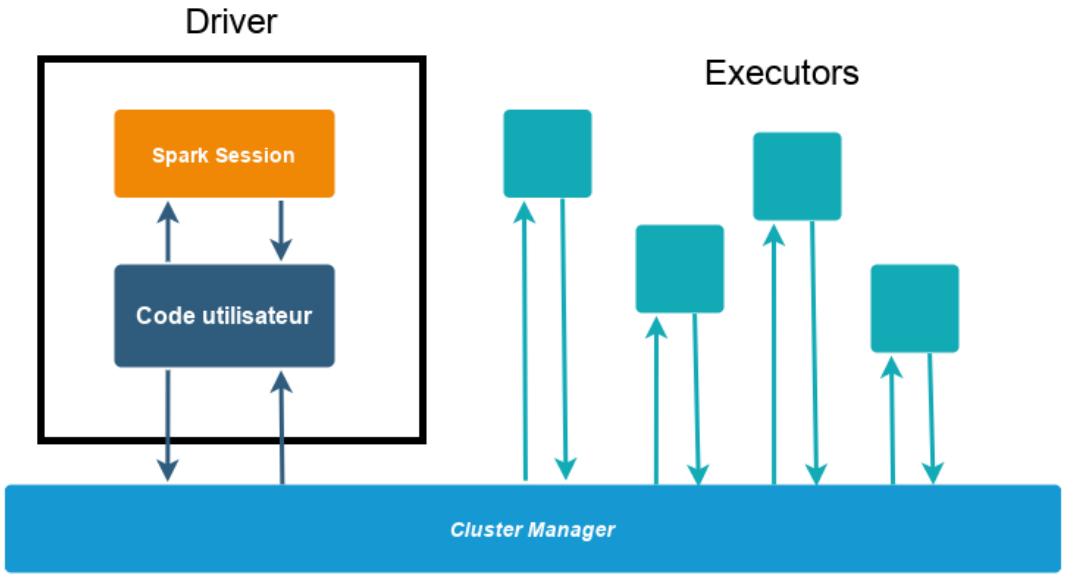


Fig. 14.1 – L'architecture système de Spark

Spark est un *framework* multilingue : les programmes Spark peuvent être écrits en Scala, Java, Python, SQL et R. Cependant, il d'abord écrit en Scala, il s'agit de son langage par défaut. L'API est complète en Scala et Java, pas nécessairement dans les autres langages.

Note : Spark peut aussi fonctionner en mode *local*, dans lequel *driver* et *executors* ne sont que des processus de la machine. La puissance de Spark est de proposer une transparence (pour les programmes) entre une exécution locale ou sur un cluster.

14.1.2 Architecture applicative

L'écosystème des API de Spark est hiérarchisé et comporte essentiellement 3 niveaux :
- les APIs bas-niveau, avec les RDDs (*Resilient Distributed Dataset*) ;
- les APIs de haut niveau, avec les *Datasets*, *DataFrames* et SQL ;
- les autres bibliothèques (*Structured Streaming*, *Advanced Analytics*, etc.).

Nous allons laisser de côté dans ce cours le dernier niveau : l'exploration des bibliothèques de *machine learning* relève du cours RCP216.

Initialement, les RDDs ont été au centre de la programmation avec Spark (ce qui a pour conséquence que de nombreuses ressources que vous trouverez sur Spark reposeront dessus). Aujourd'hui, on leur préfère des APIs de plus haut niveau, que nous allons explorer en détail, les *Datasets* et *DataFrames*. Celles-ci présentent l'avantage d'être proches de structures de données connues (avec une vision tabulaire), donc de faciliter le passage à Spark. En outre, elles sont gérées efficacement par le *framework* grâce au contrôle des types de données qu'elles lui apportent, d'où des gains de performance.

L'innovation des RDDs

La principale innovation apportée par Spark est le concept de *Resilient Distributed Dataset* (RDD). Un RDD est une collection (pour en rester à notre vocabulaire) calculée à partir d'une source de données (par exemple une base de données Cassandra, un flux de données, un autre RDD) et placée en mémoire RAM. Spark conserve l'historique des opérations qui a permis de constituer un RDD, et la reprise sur panne s'appuie essentiellement sur la préservation de cet historique afin de reconstituer le RDD en cas de panne. Pour le dire brièvement : Spark n'assure pas la préservation des données en *extension* mais en *intention*. La préservation d'un programme qui tient en quelques lignes de spécification (cf. les programmes Pig) est beaucoup plus facile et efficace que la préservation du jeu de données issu de cette chaîne. C'est l'idée principale pour la *résilience* des RDDs.

Par ailleurs, les RDDs représentent des collections partitionnées et distribuées. Chaque RDD est donc constitué de ce que nous avons appelé *fragments*. Une panne affectant un fragment individuel peut donc être réparée (par reconstitution de l'historique) indépendamment des autres fragments, évitant d'avoir à *tout* recalculer.

Les DataFrames et Datasets que nous utiliserons plus loin reposent sur les RDDs, c'est-à-dire que Spark transforme les opérations sur les DataFrames/Datasets en opérations sur les RDDs. En pratique, vous n'aurez que rarement besoin de RDDs (sauf si vous maintenez du code ancien, ou que votre expertise vous amène à aller plus loin que les *Structured APIs*).

Actions et transformations : la chaîne de traitement Spark

Un élément fondamental de la pratique de Spark réside dans **l'immutabilité** des collections (RDD ou autres). Elles ne peuvent être modifiées après leur création. C'est un peu inhabituel et cela induit des manières nouvelles de travailler.

En effet, pour passer des données d'entrée à la sortie du programme, on devra penser une chaîne de collections qui constitueront les étapes du traitement. La (ou les) première(s) collection(s) contien(nen)t les données d'entrée. Ensuite, chaque collection est le résultat de **transformations** sur les précédentes structures, l'équivalent de ce que nous avons appelé *opérateur* dans Pig. Comme dans Pig, une transformation sélectionne, enrichit, restructure une collection, ou combine deux collections. On retrouve dans Spark, à peu de choses près, les mêmes opérateurs/transformations que dans Pig, comme le montre la table ci-dessous (qui n'est bien sûr pas exhaustive : reportez-vous à la documentation pour des compléments).

Opérateur	Description
`map`	Prend un document en entrée et produit un document en sortie
`filter`	Filtre les documents de la collection
`flatMap`	Prend un document en entrée, produit un ou plusieurs document(s) en sortie
`groupByKey`	Regroupement de documents par une valeur de clé commune
`reduceByKey`	Réduction d'une paire *(k, [v])* par une agrégation du tableau *[v]*
`crossProduct`	Produit cartésien de deux collections
`join`	Jointure de deux collections
`union`	Union de deux collections
`cogroup`	Cf. la description de l'opérateur dans la section sur Pig
`sort`	Tri d'une collection

Les collections obtenues au cours des différentes étapes d'une chaîne de traitement sont stockées dans des RDDs, des DataFrames, etc., selon l'API employée. C'est exactement la notion que nous avons déjà étudiée avec Pig. La différence essentielle est que dans Spark, les RDD ou DataFrames sont, par défaut, *transients*, c'est-à-dire non matérialisés sur un support externe comme le disque. Ils peuvent cependant être marqués comme étant *persistants*, dans le cas où l'on souhaite les réutiliser à plusieurs reprises (cas d'une itération). Spark fait son possible pour conserver les structures persistantes en mémoire RAM, pour un maximum d'efficacité.

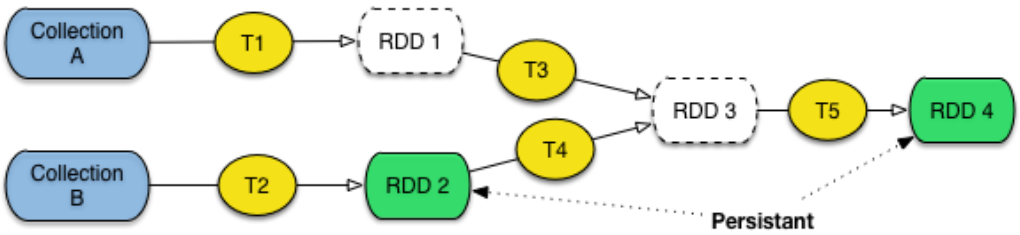


Fig. 14.2 – RDD persistants et transitoires dans Spark.

Les collections forment un graphe construit par application de transformations à partir de collections stockées (Fig. 14.2). S'il n'est pas marqué comme persistant, le RDD/DataFrame sera transitoire et ne sera pas

conservé en mémoire après calcul (c'est le cas des RDD 1 et 3 sur la figure). Sinon, il est stocké en RAM, ou mis sur disque s'il n'y a pas assez de mémoire, et disponible comme source de données pour d'autres transformations.

Par opposition aux transformations qui produisent d'autres RDD ou DataFrames, les **actions** produisent des *valeurs* (pour l'utilisateur). L'évaluation des opérations en Spark est dite « paresseuse », c'est-à-dire que Spark attend le plus possible pour exécuter le graphe des instructions de traitement. Comme dans Pig, une action déclenche donc l'exécution de l'ensemble des transformations qui la précèdent.

L'évaluation paresseuse (*lazy evaluation*) permet à Spark de compiler de simples transformations de DataFrames en un plan d'exécution physique efficacement réparti dans le cluster. Un exemple de cette efficacité est illustrée par le concept de *predicate pushdown* : si un `filter()` à la fin d'une séquence amène à ne travailler que sur une ligne des données d'entrée, les autres opérations en tiendront compte, optimisant d'autant la performance en temps et en espace.

RDDs, *Dataset* et *DataFrame*

Un RDD, venant de l'API bas-niveau, est une « boîte » destinée à contenir n'importe quel document, sans aucun préjugé sur la structure (ou l'absence de structure) de ce dernier. Cela rend le système très généraliste, mais empêche une manipulation fine des constituants des documents, comme par exemple le filtrage en fonction de la valeur d'un champ. C'est le programmeur de l'application qui doit fournir la fonction effectuant le filtre. Cela impose un « décodage » des éléments du RDD dans un format reconnu par le langage de programmation utilisé. C'es ce décodage (ou, pour le dire plus techniquement, la sérialisation/désérialisation) qui pénalise l'utilisation directe des RDD.

On l'a dit, Spark implémente une API de plus haut niveau avec des structures assimilables à des tables relationnelles : les *Dataset* et *DataFrame*. Ils comportent un *schéma*, avec les définitions des colonnes. La connaissance de ce schéma – et éventuellement de leur type dans le cas des *datasets* – permet à Spark de proposer des opérations plus fines, et des optimisations inspirées des techniques d'évaluation de requêtes dans les systèmes relationnels. En fait, on se rapproche d'une implantation distribuée du langage SQL. En interne, un avantage important de la connaissance du schéma est d'éviter de recourir à la sérialisation des objets Java (opération effectuée dans le cas des RDD pour écrire sur disque et échanger des données en réseau).

Note : Saluons au passage le mouvement progressif de ces systèmes vers une ré-assimilation des principes du relationnel (schéma, structuration des données, interrogation à la SQL, etc.), et la reconnaissance des avantages, internes et externes, d'une modélisation des données. Du *NoSQL* à *BackToSQL* !

On distingue les *Dataset*, dont le type des colonnes est connu, et les *DataFrames*. Un *DataFrame* n'est rien d'autre qu'un *Dataset* (`DataFrame = Dataset[Row]`) contenant des lignes de type *Row* dont le schéma précis n'est pas connu. Ce typage des structures de données est lié au langage de programmation : Python et R étant dynamiquement typés, ils n'accèdent qu'aux DataFrames. En Scala et Java en revanche, on utilise les Datasets, des objets JVM fortement typés.

Tout cela est un peu abstrait ? Voici un exemple simple qui permet d'illustrer les principaux avantages des *Dataset/DataFrame*. Nous voulons appliquer un opérateur qui filtre les films dont le genre est « Drame ». On va exprimer le filtre (en simplifiant un peu) comme suit :

```
films.filter(film.getGenre() == 'Drame');
```

Si `films` est un RDD, Spark n'a aucune idée sur la structure des documents qu'il contient. Spark va donc instancier un objet Java (éventuellement en dé-sérialisant une chaîne d'octets reçue par réseau ou lue sur disque) et appeler la méthode `getGenre()`. Cela peut être long, et impose surtout de créer un objet pour un simple test.

Avec un *Dataset* ou *DataFrame*, le schéma est connu et Spark utilise son propre système d'encodage/décodage à la place de la sérialisation Java. De plus, dans le cas des *Dataset*, la valeur du champ `genre` peut être testée directement sans même effectuer de décodage depuis la représentation binaire.

Il est, en résumé, tout à fait préférable d'utiliser les *Dataset* dès que l'on a affaire à des données structurées.

14.1.3 Exemple : analyse de fichiers *log*

Prenons un exemple concret : dans un serveur d'application, on constate qu'un module *M* produit des résultats incorrects de temps en temps. On veut analyser le fichier journal (*log*) de l'application qui contient les messages produits par le module suspect, et par beaucoup d'autres modules.

On construit donc un programme qui charge le *log* sous forme de collection, ne conserve que les messsages produits par le module *M* et analyse ensuite ces messages. Plusieurs analyses sont possibles en fonction des causes suspectées : la première par exemple regarde le *log* de *M* pour un produit particulier, la seconde pour un utilisateur particulier, la troisième pour une tranche horaire particulière, etc.

Avec Spark, on va créer un DataFrame `logM` persistant, contenant les messages produits par *M*. On construira ensuite, à partir de `logM` de nouveaux DataFrames dérivés pour les analyses spécifiques (Fig. 14.3).

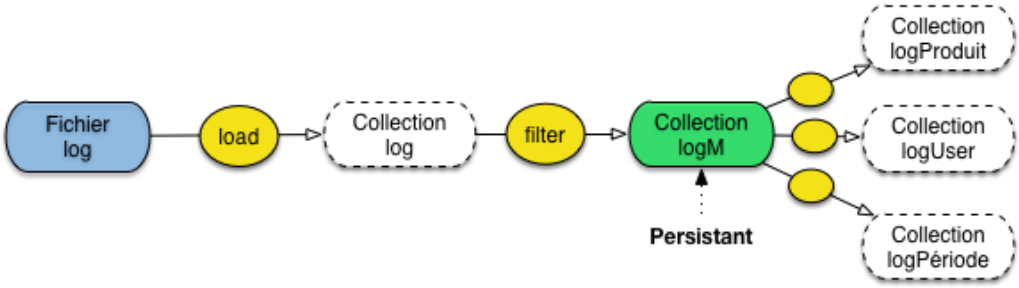


Fig. 14.3 – Scénario d'une analyse de *log* avec Spark

On combine deux transformations pour construire `logM`, comme le montre le programme suivant (qui n'est pas la syntaxe exacte de Spark, que nous présenterons plus loin).

```
// Chargement de la collection
log = load ("app.log") as (...)
// Filtrage des messages du module M
logM = filter log with log.message.contains ("M")
// On rend logM persistant !
logM.persist();
```

On peut alors construire une analyse basée sur le code produit directement à partir de `logM`.

```
// Filtrage par produit
logProduit = filter logM with log.message.contains ("product P")
// .. analyse du contenu de logProduit
```

Et utiliser également `logM` pour une autre analyse, basée sur l'utilisateur.

```
// Filtrage par utilisateur
logUtilisateur = filter logM with log.message.contains ("utilisateur U")
// .. analyse du contenu de logProduit
```

Ou encore par tranche horaire.

```
// Filtrage par utilisateur
logPeriode = filter logM with log.date.between d1 and d2
// .. analyse du contenu de logPeriode
```

`logM` est une sorte de « vue » sur la collection initiale, dont la persistance évite de refaire le calcul complet à chaque analyse.

14.1.4 Reprise sur panne

Pour comprendre la reprise sur panne, il faut se pencher sur le second aspect des RDD : la *distribution*. Un RDD est une collection *partitionnée* (cf. chapitre *Systèmes NoSQL : le partitionnement*), les DataFrames le sont aussi. La Fig. 14.4 montre le traitement précédent dans une perspective de distribution. Chaque DataFrame, persistant ou non, est composé de fragments répartis dans la grappe de serveurs.

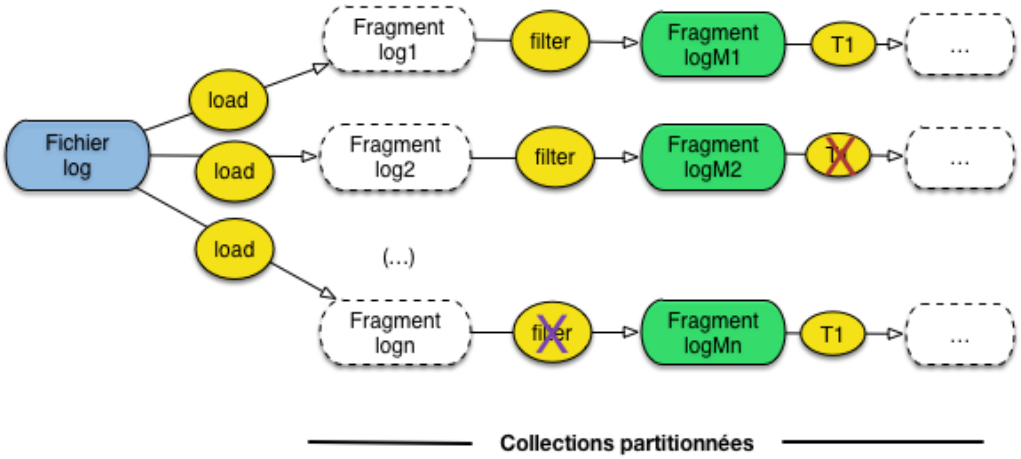


Fig. 14.4 – Partitionnement et reprise sur panne dans Spark.

Si une panne affecte un calcul s'appuyant sur un fragment F de DataFrame persistant (par exemple la transformation notée `T` et marquée par une croix rouge sur la figure), il suffit de le relancer à partir de F. Le gain en temps est considérable !

La panne la plus sévère affecte un fragment de DataFrame *non* persistant (par exemple celui marqué par une croix violette). Dans ce cas, Spark a mémorisé la chaîne de traitement ayant constitué le DataFrame, et il suffit de ré-appliquer cette chaîne en remontant jusqu'aux fragments qui précèdent dans le graphe des calculs.

Dans notre cas, il faut parcourir à nouveau le fichier `log` pour créer le fragment `logn`. Si les collections stockées à l'origine du calcul sont elles-mêmes partitionnées (ce qui n'est sans doute pas le cas pour un fichier *log*), il suffira d'accéder à la partie de la collection à l'origine des calculs menant au DataFrame défaillant.

En résumé, Spark exploite la capacité à reconstruire des fragments de RDD/DataFrame par application de la chaîne de traitement, et ce en se limitant si possible à une partie seulement des données d'origine. La reprise peut prendre du temps, mais elle évite un recalcul complet. Si tout se passe bien (pas de panne) la présence des résultats intermédiaires en mémoire RAM assure de très bonnes performances.

14.1.5 Quiz

14.2 S2 : Mise en pratique

Il est temps de passer à l'action. Nous allons commencer par montrer comment effectuer des transformations sur des données non-structurées avec des DataFrames standard. Les exemples qui suivent sont proposés en Python, mais d'autres interfaces existent, notamment en Scala et en R. Scala est un langage fonctionnel, doté d'un système d'inférence de types puissant, ce qui le rend particulièrement approprié pour exprimer des chaînes de traitements sous la forme d'une séquence d'appels de fonctions. Je fais l'hypothèse que la plupart de mes lecteurs seront plus familiers avec Python.

Le plus simple pour reproduire ces commandes est de télécharger dans un répertoire `spark` la dernière version de Spark depuis le site http://spark.apache.org. L'installation comprend un sous-répertoire `bin` dans lequel se trouvent les commandes qui nous intéressent (et notamment l'interpréteur `pyspark`). Vous pouvez placer le chemin vers `spark/bin` dans votre variable `PATH`, selon des spécificités qui dépendent de votre environnement : à ce stade du cours vous devriez être rôdés à ce type de manœuvre.

```
pyspark
```

Aux numéros de version près, vous devriez obtenir l'affichage suivant :

```
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /__ / .__/\_,_/_/ /_/\_\   version 3.4.3
      /_/

Using Python version 3.10.0 (v3.10.0:b494f5935c, Oct  4 2021 14:59:20)
Spark context Web UI available at http://163.173.78.98:4040
Spark context available as 'sc' (master = local[*], app id = local-
↪1718116101741).
SparkSession available as 'spark'.
>>>
```

C'est parti !

14.2.1 Transformations et actions

Vous pouvez récupérer le fichier http://b3d.bdpedia.fr/files/loups.txt pour faire un essai (il est temps de savoir à quoi s'en tenir à propos de ces loups et de ces moutons !), sinon n'importe quel fichier texte fait l'affaire. Copiez-collez les commandes ci-dessous.

```
loupsEtMoutons = spark.read.text("loups.txt")
```

Nous avons créé un premier DataFrame nommé `loupsEtMoutons` contenant autant de documents que de lignes dans le fichier en entrée, avec une unique colonne `value`. Spark propose des *actions* directement applicables à un DataFrame et produisant des résultats scalaires. (Un DataFrame est interfacé comme un objet auquel nous pouvons appliquer des méthodes). Voici des exemples des méthodes `count()` et `first()`.

```
loupsEtMoutons.count() # Nombre de documents dans ce RDD
  res0: Long = 4

loupsEtMoutons.first() // Premier document du RDD
  res1: String = Le loup est dans la bergerie.
```

La fonction `show()` est particulière : elle affiche le contenu du Dataframe sous forme de table.

```
loupsEtMoutons.show() // Récupération du dataframe complet

+--------------------+
|               value|
+--------------------+
|Le loup est dans ...|
|Les moutons sont ...|
|Un loup a mangé u...|
|Il y a trois mout...|
+--------------------+
```

Note : Petite astuce : en entrant le nom de l'objet (`loupsEtMoutons.`) suivi de la touche TAB, l'interpréteur vous affiche la liste des méthodes disponibles.

Passons aux *transformations*. Elles prennent un (ou deux) DataFrame en entrée, produisent un DataFrame en sortie. On peut sélectionner (filtrer) les documents (lignes) qui contiennent « bergerie ».

```
bergerie = loupsEtMoutons.filter(loupsEtMoutons.value.contains("bergerie"))
```

Notez l'accès à l'attribut `value` qui est simplement le contenu textuel de chaque document du Dataframe. La fonction `filter()` reçoit un booléen (ici, application de la fonction Python standard `contains()`) et ne conserve dans la collection résultante que les lignes pour lesquelles `True` était retourné.

Nous avons créé un second DataFrame nommé `bergerie`. Nous sommes en train de définir une chaîne de traitement qui part ici d'un fichier texte et applique des transformations successives.

À ce stade, rien n'est calculé, on s'est contenté de déclarer les étapes. Dès que l'on déclenche une *action*, comme par exemple l'affichage du contenu d'un DataFrame (avec `show()`), Spark va déclencher l'exécution.

```
    bergerie.show()

+--------------------+
|               value|
+--------------------+
|Le loup est dans ...|
|Les moutons sont ...|
|Un loup a mangé u...|
+--------------------+
```

On peut combiner une transformation et une action. En fait, comme avec pig, on peut chaîner les opérations et ainsi définir très concisément le *workflow*. Combien de documents contiennent le mot « loup » ?

```
loupsEtMoutons.filter(loupsEtMoutons.value.contains("loup")).count()
3
```

Et pour conclure cette petite session introductive, voici comment on implante en le compteur de termes dans une collection.

Compteur de termes, en DataFrames

Nous allons avoir besoin de la librairie des fonctions Spark/Python. On l'importe comme suit :

```
from pyspark.sql import functions as sf
```

On crée un premier DataFrame constitué de tous les termes obtenus en appliquant la fonction (standard) Python `split()` aux documents :

```
termes = loupsEtMoutons.select(sf.split(loupsEtMoutons.value, "\s+").name("mots
↪"))
```

La méthode `split` décompose une chaîne de caractères (ici, en prenant comme séparateur un espace) en une liste de mots. On donne un nom à la colonne avec `name()` (sinon la colonne est nommée par défaut `split(value, \s+, -1)split(value, \s+, -1)`, ce qui n'est pas très pratique). Notez que `split`, comme beaucoup d'autres fonctions, crée une colonne, et qu'il faut appeler la fonction `select` pour construire un *dataframe* à partir de cette colonne (ou de plusieurs).

```
+--------------------+
|                mots|
+--------------------+
|[Le, loup, est, d...|
|[Les, moutons, so...|
```

(suite sur la page suivante)

(suite de la page précédente)

```
|[Un, loup, a, man...|
|[Il, y, a, trois,...|
+--------------------+
```

Nous allons maintenant « aplatir » chaque tableau pour, à partir d'une ligne de la colonne `mots`, obtenir autant de lignes qu'il y a de mots. C'est l'équivalent de la fonction `flatten` dans Pig. Concrètement :

```
listeMots = termes.select(sf.explode(termes.mots).alias("mot"))
```

Ce qui donne un nouveau Dataframe `listeMots` :

```
+---------+
|      mot|
+---------+
|       Le|
|     loup|
|      est|
|     dans|
|       la|
|bergerie.|
|      ...|
```

Groupons maintenant les mots :

```
compteurTermes = listeMots.groupBy("mot")
```

On obtient une structure intermédiaire de type `GroupedData` sur laquelle on peut appliquer des opérations d'agrégation, la plus simple étant `count`.

```
compteurTermes.count().show()

+---------+-----+
|      mot|count|
+---------+-----+
|bergerie.|    3|
|       du|    1|
|     pré,|    1|
|    mangé|    1|
|       Le|    1|
|   autres|    1|
|     sont|    2|
```

Et voilà ! On a décomposé chaque étape, mais on aurait pu exprimer toute la chaîne de traitement en une seule fois.

```
compteurTermes = loupsEtMoutons.select(sf.split(loupsEtMoutons.value, "\s+"
                 ).name("mots")
```

(suite sur la page suivante)

(suite de la page précédente)

```
                    ).select(sf.explode(sf.col("mots")
                    ).name("mot")
                    ).groupBy("mot"
                    ).count(
                    ).show()
```

Note : Attention aux indentations en Python… Si vous voulez reproduire la commande, le plus simple est de tout mettre sur une seule ligne.

Le résultat pourra vous sembler un peu étrange (`pré,`) car il manque les diverses étapes de simplification du texte qui sont de mise pour un moteur de recherche (vues dans le chapitre *Recherche approchée* pour les détails). Mais l'essentiel est de comprendre l'enchaînement des opérateurs.

Finalement, si on souhaite conserver en mémoire le DataFrame final pour le soumettre à divers traitements, il suffit d'appeler la fonction `cache()` :

```
compteurTermes.cache()
```

14.2.2 Spark SQL, gestion de données structurées

Allons maintenant un peu plus loin en étudiant l'import de données structurées dans Spark et leur manipulation avec Spark SQL, une forme de SQL adaptée aux spécificités des *dataframes*. Nous allons prendre le fichier des films `films.json`, dans le format proposé sur https://deptfod.cnam.fr/bd/tp/datasets/ (il convient également pour un import dans MongoDB).

Pour la création du *dataframe* initial, on applique simplement la fonction de lecture JSON.

```
 df = spark.read.json("films.json")

# Le schéma a été inferré d'après le contenu
df.printSchema()

 # Regardons un extrait de ce contenu
df.show()
```

Le *dataframe* peut maintenant être inspecté en s'appuyant sur le schéma et des fonctions spécifiques aux types de données importées. Quelques exemples :

```
    # Affichage de quelques colonnes
df.select(df["director"], df["year"]).show()
    # Filtrage
    df.filter(df['year'] > 2000).show()
    # Regroupement et comptage
    df.groupBy("year").count().show()
```

On peut même pousser l'illusion un cran plus loin et créer une représentation relationnelle du *dataframe*.

```
df.createOrReplaceTempView("films")
```

`films` est alors une table sur laquelle on peut exprimer des requêtes SQL.

```
sqlDF = spark.sql("SELECT * FROM films where year > 2000")
```

Spark SQL connaît les types imbriqués, comme le montre l'exemple suivant :

```
sqlDF = spark.sql("SELECT director.first_name FROM films where year > 2000").
↪show()
```

Et on peut effectuer des transformations structurelles, comme « l'aplatissement » d'un tableau. Voici comment créer le *dataframe* associant à chaque acteur le titre et le metteur en scène de chacun des films dans lesquels il a joué.

```
roles = df.select(df.title,sf.explode (df.actors).alias("acteur"),df.director)
```

C'est l'équivalent d'un `Map` dans lequel on émettrait une paire clé valeur pour chaque acteur d'un film. L'équivalent du `Reduce` est obtenu par la combinaison de `groupBy` suivi d'une fonction d'agrégation. Voici le nombre de rôles joués par chaque acteur.

```
acteurs = roles.groupBy(roles.acteur).count()
```

14.2.3 Mise en pratique

Exercice MEP-SPark-1 : à vous de jouer

Vous vous doutez de ce qu'il faut faire à ce stade : reproduire les commandes qui précèdent, et explorer l'interface de Spark jusqu'à ce que tout soit clair. Vous y passerez peut-être un peu de temps mais à cette mise en pratique vous mettra très concrètement au cœur d'un système très utilisé, et qui repose sur une bonne partie des concepts vus en cours.

Exercice MEP-SPark-2 : Passons à PageRank

Note : cet exercice est donnée en Scala, la version Python viendra prochainement, mais en attendant considérez qu'il s'agit d'une proposition optionnelle.

Essayons d'implanter notre PageRank avec Spark. On va supposer que notre graphe est stocké dans un fichier texte `graphe.txt` avec une ligne par arête,

```
url1 url2
url1 url3
url2 url3
url3 url2
```

Commençons par créer la matrice (ou plus exactement les vecteurs représentant les liens sortants pour chaque URL).

```
val graphe = spark.read.textFile("graphe.txt")
val matrix = graphe.map{ s =>
                    val parts = s.split("\\s+")
                    (parts(0), parts(1))
                }.distinct().groupByKey()
```

Initialisons le vecteur initial des rangs

```
var ranks = matrix.mapValues(v => 1.0)
```

Appliquons 20 itérations.

```
for (i <- 1 to 20) {
  val contribs =
      matrix.join(ranks)
            .values
            .flatMap{ case (urls, rank) =>
                        val size = urls.size
                        urls.map(url => (url, rank / size))
                    }
      ranks = contribs.reduceByKey(_ + _)
}
```

Finalement exécutons le tout

```
ranks.show()
```

Une fois que cela fonctionne, vous pouvez effectuer quelques améliorations

1. Ajoutez des opérateurs `persist()` ou `cache()` où cela vous semble pertinent.
2. Raffinez PageRank en introduisant une probabilité (10 % par exemple) de faire un « saut » vers une page quelconque au lieu de suivre les liens sortants.

Correction

```
matrix.cache()
ranks = contribs.reduceByKey(_ + _).mapValues(0.15 + 0.85 * _)
```

14.3 S3 : Traitement de données structurées avec Cassandra et Spark

Supports complémentaires
— Vidéo Spark: de Cassandra aux Datasets

Voyons maintenant les outils de traitement proposés par Spark sur des données structurées issues, par exemple, d'une base de données, ou de collections de documents JSON. On interagit dans ce cas évidemment de façon privilégiée avec les *DataFrames* et les *Datasets*. On l'a dit, les deux structures sont semblables à des tables relationnelles, mais la seconde est, de plus, fortement typée puisqu'on connaît le type de chaque colonne. Cela simplifie considérablement les traitements, aussi bien du point de vue du concepteur des traitements que de celui du système.

— Pour le concepteur, la possibilité de référencer des champs et de leur appliquer des opérations standard en fonction de leur type évite d'avoir à écrire une fonction spécifique pour la moindre opération, rend le code beaucoup lisible et concis.
— Pour le système, la connaissance du schéma facilite les contrôles *avant* exécution (*compile-time checking*, par opposition au *run-time checking*), et permet une sérialisation très rapide, indépendante de la sérialisation Java, grâce à une couche composée d" *encoders*.

Nous allons en profiter pour instancier un début d'architecture réaliste en associant Spark à Cassandra comme source de données. Dans une telle organisation, le stockage et le partitionnement sont assurés par Cassandra, et le calcul distribué par Spark. Idéalement, chaque nœud Spark traite un ou plusieurs fragments d'une collection partitionnée Cassandra, et communique donc avec un des nœuds de la grappe Cassandra. On obtient alors un système complètement distribué et donc *scalable*.

14.3.1 Préliminaires

La base Cassandra que nous prenons comme support est celle des restaurants New-Yorkais. Reportez-vous au chapitre *Cassandra - Travaux Pratiques* pour la création de cette base. Dans ce qui suit, on suppose que le serveur Cassandra est dans un conteneur Docker qui effectue un renvoi sur le port 3000 de la machine hôte. On se connecte donc sur Cassandra avec la machine `localhost` et le port 3000. Je suppose également qu'à ce stade du cours vous êtes capables d'identifier votre propre configuration.

Pour associer Spark et Cassandra, il faut utiliser un connecteur disponible sur GitHub : https://github.com/datastax/spark-cassandra-connector. Nous allons nous appuyer sur l'interface Python, `pyspark` qui effectue automatiquement un téléchargement des librairies nécessaires quand on le lance avec l'option suivante (la version indiquée ici est la 3.5.0, prise en janvier 2025).

```
pyspark --packages com.datastax.spark:spark-cassandra-connector_2.12:3.5.0
```

Une fois `PySpark` lancé, vous devez pouvoir établir une connexion avec Cassandra comme suit (en reprenant les paramètres à adapter selon votre contexte).

```
session = SparkSession.builder \
        .appName("NFE204") \
        .config("spark.cassandra.connection.host", "localhost") \
        .config("spark.cassandra.connection.port", "3000") \
```

(suite sur la page suivante)

(suite de la page précédente)

```
        .config("spark.sql.extensions", "com.datastax.spark.connector.
↪CassandraSparkExtensions") \
        .getOrCreate()
```

L'objet `session` permet de communiquer avec Cassandra. Essayez d'afficher un extrait des restaurants :

```
restaurants_df = session.read.format("org.apache.spark.sql.cassandra") \
                .options(table="restaurant", keyspace="resto_ny") \
                .load()
restaurants_df.show()
```

Tout va bien ? Nous avons donc créé un *DataFrame* Spark nommé `restaurant_df`, dont le schéma (noms des colonnes) a été directement obtenu depuis Cassandra. En revanche, les colonnes ne sont pas typées (on pourrait espérer que le type est récupéré et transcrit depuis le schéma de Cassandra, mais ce n'est malheureusement pas le cas).

Vous remarquerez peut-être que si vous exécutez plusieurs fois la commande `show()`, seule la première prends un peu de temps. Pour les suivantes, Spark a mis en cache le contenu du dataset `restaurants_df` et l'affichage est quasiment instantané. C'est une règle générale : Spark tente de préserver les calculs intermédiaire, même s'ils ne sont pas marqués comme persistants. On peut forcer la conservation du dataframe

```
restaurants_df.cache()
```

Tant que nous y sommes, nous allons créer un *Dataframe* pour les inspections.

```
inspections_df = session.read.format("org.apache.spark.sql.cassandra") \
                .options(table="inspection", keyspace="resto_ny") \
                .load()
```

Vous pouvez le rendre persistant si vous voulez.

14.3.2 Traitements Spark/Cassandra

Voici quelques exemples de transformations Spark appliquées à des données issues de Cassandra. Commençons par les *projections* (malencontreusement référencées par le mot-clé `select` depuis les débuts de SQL) consistant à ne conserver que certaines colonnes. La commande suivante ne conserve que trois colonnes.

```
restaus_simples = restaurants_df.select("name", "phone", "cuisinetype")
restaus_simples.show()
```

Voici maintenant comment on effectue une sélection (avec le mot-clé `filter`, correspondant au `where` de SQL).

```
manhattan = restaurants_df.filter("borough =  'MANHATTAN'")
manhattan.show()
```

Par la suite, nous omettons l'appel à *show()* que vous pouvez ajouter si vous souhaitez consulter le résultat. Tout cela aurait aussi bien pu s'exprimer en CQL (voir exercices). Mais Spark va définitivement plus loin

en termes de capacité de traitements, et propose notamment la fameuse opération de jointure qui nous a tant manqué jusqu'ici.

```
restaus_inspections = restaurants_df.join(inspections_df, restaurants_df.id ==␣
↪inspections_df.idrestaurant)
```

Le calcul peut prendre un peut de temps pour la première action déclenchée sur `restaus_inspections`. Par la suite, Spark aura sans doute mis le résultat en *cache*, ce que l'on peut forcer avec :

```
restaus_inspections.cache()
```

On peut effectuer des agrégats, comme par exemple le regroupement des restaurants par arrondissement (*borough*) :

```
comptage_par_borough = restaus_inspections.groupBy("borough").count()
```

Et un exemple complet : la moyenne des notes des restaurants de tapas.

```
from pyspark.sql import functions as sf

comptage_tapas = restaurants_df.filter("cuisinetype > 'Tapas'") \
        .join(inspections_df, restaurants_df.id == inspections_df.idrestaurant) \
        .groupBy("name") \
        .agg(sf.avg("score"))
```

14.3.3 L'interface de contrôle Spark

Spark dispose d'une interface Web qui permet de consulter les entrailles du système et de mieux comprendre ce qui est fait. Elle est accessible sur le port 4040, donc à l'URL http://localhost:4040 pour une exécution du *shell*. Pour explorer les informations fournies par cette interface, nous allons exécuter notre *workflow* calculant la moyenne des scores des restaurants de tapas. Lancez *pyspark* est exécutez ce *workflow*.

Maintenant, vous devriez pouvoir accéder à l'interface et obtenir un affichage semblable à celui de la Fig. 14.5. En particulier, le *job* que vous venez d'exécuter devrait apparaître, avec sa durée d'exécution et quelques autres informations.

L'onglet *jobs*

Chaque exécution d'une action correspond à un *job*, lui-même décomposé en *stages* (étapes). Cette décomposition correspond à l'identification des étapes du *workflow* qui peuvent s'exécuter en parallèle.

À quoi correspondent ces *étapes* ? En fait, si vous avez bien suivi ce qui précède dans le cours, vous avez les éléments pour répondre : une *étape* dans Spark regroupe un ensemble d'opérations qu'il est possible d'exécuter *localement*, sur une seule machine, sans avoir à effectuer des échanges réseau. C'est une généralisation de la phase de *Map* dans un environnement MapReduce. Les étapes sont logiquement séparées par des phases de *shuffle* qui consistent à redistribuer les données afin de les regrouper selon certains critères. Relisez le chapitre *Le cloud, une nouvelle machine de calcul* pour revoir vos bases du calcul distribué si ce n'est pas clair.

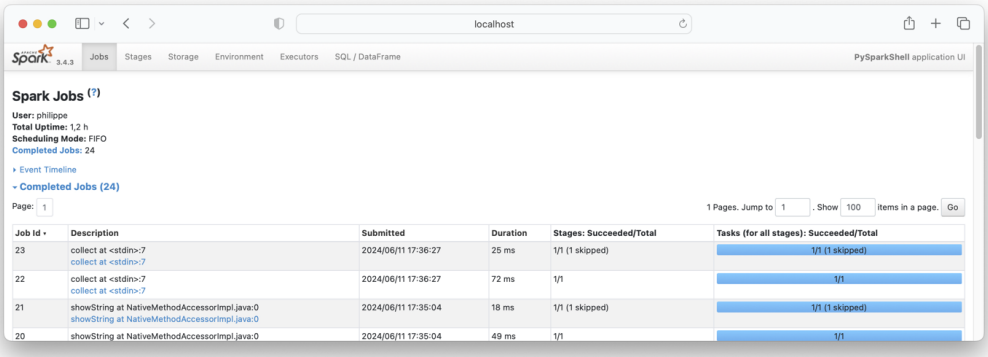


Fig. 14.5 – L'interface Web de Spark

Quand le traitement s'effectue sur des données partitionnées, une *étape* est effectuée en parallèle sur les fragments, et Spark appelle *tâche* l'exécution de l'étape sur un fragment particulier, pour une machine particulière. Résumons :

— Un *job* est l'exécution d'une chaîne de traitements (*workflow*) dans un environnement distribué.
— Un *job* est découpé en *étapes*, chaque étape étant un segment du *workflow* qui peutêtre parallélisé.
— L'exécution d'une étape se fait par un ensemble de tâches, une par machine hébergeant un fragment du RDD servant de point d'entrée à l'étape.

Entre deux *stages*, il y a donc nécessairement une étape de distribution des données (*shuffle*) qui permet d'initialiser l'état de départ du *stage* qui suit. On retrouve une fonctionnement de base illustré déjà par MapReduce : les deux phases, Map et Reduce, sont parallélisables, mais le passage de l'une à l'autre correspond à une forme de synchronisation des données pour les rendre adapté à l'entrée de l'étape qui suit.

Cliquez sur le nom du *job* pour obtenir des détails sur les étapes du calcul (Fig. 14.6). Spark nous dit que l'exécution s'est faite en trois étapes. Ce n'est pas forcément très clair, mais la première comprend les transformations textuelle, et la seconde les opérations d'agrégation. Les deux étapes sont séparées par une phase de *shuffle*.

Vous pouvez noter que certaines étapes sont grisées et marquées comme *Skipped*. Elles correspondent à celles pour lesquelles le résultat est placé en *cache*. Elles font donc partie du flot logique d'exécution, mais ne sont pas ré-exécutées.

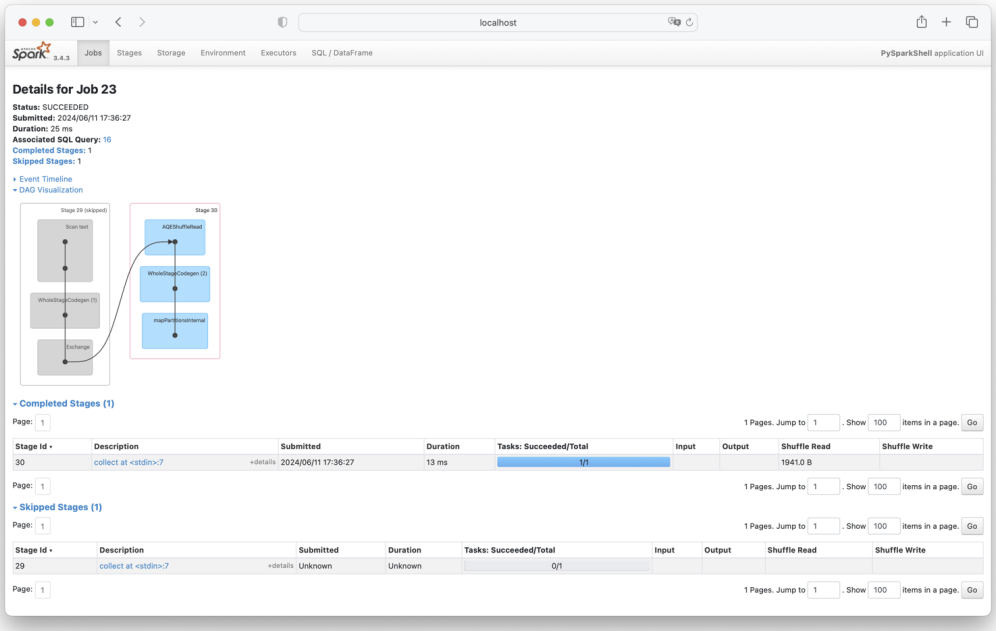


Fig. 14.6 – Plan d'exécution d'un *job* Spark : les étapes.

L'onglet *Stages*

Vous pouvez obtenir des informations complémentaires sur chaque étape avec l'onglet *Stages* (qui veut dire *étapes*, en anglais). En particulier, l'interface montre de nombreuses statistiques sur le temps d'exécution, le volume des données échangées, etc. Tout cela est très précieux quand on veut vérifier que tout va bien pour des traitements qui durent des heures ou des jours.

L'onglet *Storage*

Maintenant, consultez l'onglet *Storage*. Il montre les *datasets* et *RDD* persistants, placés en cache. Introduisez par exemple l'opération de persistance `cache()` dans le *workflow* :

```
comptage_tapas = restaurants_df.filter("cuisinetype > 'Tapas'") \
.join(inspections_df, restaurants_df.id == inspections_df.idrestaurant) \
.cache()
```

Et exécutez à nouveau l'action `comptage_tapas.show()`. Le RDD correspondant devrait apparaître dans l'onglet *Storage*.

Exécutez une nouvelle fois l'action `show()` et consultez les statistiques des temps d'exécution. La dernière exécution devrait être significativement plus rapide que les précédentes. Comprenez-vous pourquoi ? Regardez les étapes, et clarifiez tout cela dans votre esprit.

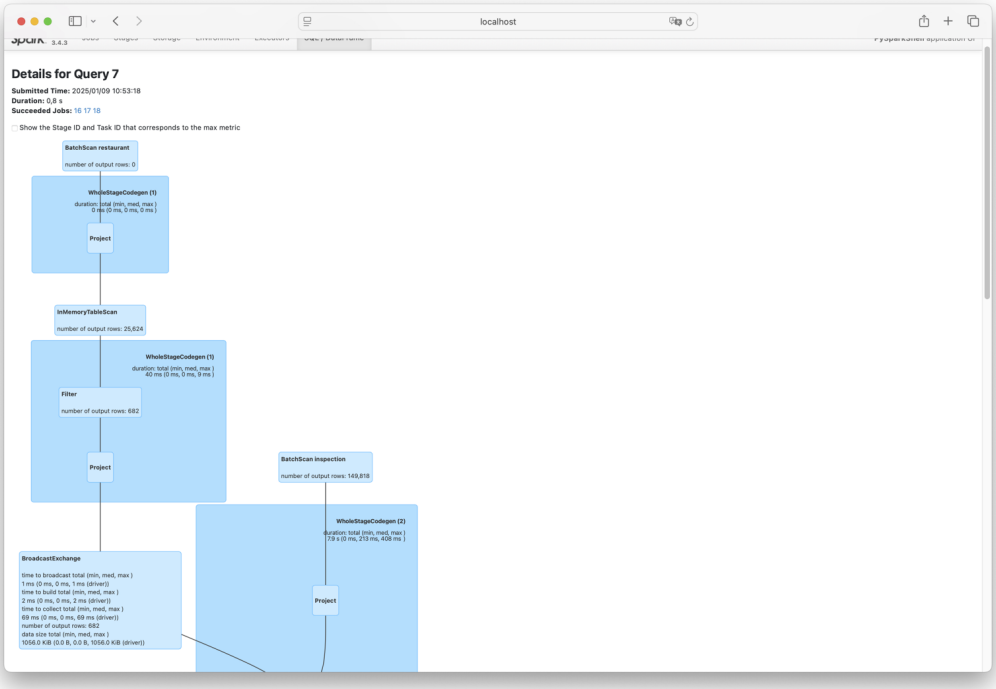


Fig. 14.7 – L'onglet SQL/Dataframe et le plan d'exécution détaillé.

L'onglet *SQL/Dataframe*

Cet onglet montre de manière assez complète le plan d'exécution Spark pour la jointure et l'agrégation. C'est le plus proche de ce que l'on obtient avec la commande *explain* dans les systèmes relationnels : un plan d'exécution qui décrit les phases de traitement des données. Ici, ce plan correspond à une jointure par tri-fusion en environnement distribué (Fig. 14.6). Si vous avez déjà exploré des plans d'exécution relationnels vous devriez y retrouver les éléments esentiels.

14.3.4 Mise en pratique

Exercice MEP-SPark-3 : à vous de jouer

La mise en pratique de cette session est plus complexe. Si vous choisissez de vous y lancer, vous aurez un système quasi complet (à toute petite échelle) de stockage et de calcul distribué.

14.4 Exercices

Exercice Ex-Spark-1 : Réfléchissons aux traitements itératifs

Le but de cet exercice est de modéliser le calcul d'un algorithme itératif avec Spark. Nous allons prendre comme exemple celui que nous connaissons déjà : PageRank. On prend comme point de départ un ensemble de pages Web contenant des liens, stockés dans un système comme, par exemple, Elastic Search.

Pour l'instant il ne vous est pas demandé de produire du code, mais de réfléchir et d'exposer les principes, et notamment la gestion des RDD.

— Partant d'un stockage distribué de pages Web, quelle chaîne de traitement permet de produire la représentation matricielle du graphe de PageRank ? Quelles opérations sont nécessaires et où stocker le résultat ?
— Quelle chaîne de traitement permet de calculer, à partir du graphe, le vecteur des PageRank ? Vous pouvez fixer un nombre d'itérations (100, 200) ou déterminer une condition d'arrêt (beaucoup plus difficile). Indiquez les RDD le long de la chaîne complète.
— Indiquez finalement quels RDD devraient être marqués persistants. Vous devez prendre en considération deux critères : amélioration des performances et diminution du temps de reprise sur panne.

Correction

— Pour chaque document (une page web), il faut extraire la liste des liens *sortants* (donc, dans le cas du HTML, toutes les balises `<a href='lien' .../>`. Il s'agit typiquement d'une opération de *Map*, sans *Reduce* ! La clé d'émission est l'URL de la page analysée, la valeur est la liste des URL sortantes.
Possibilité plus paresseure : on émet chaque lien au fur et à mesure de leur rencontre, et on ajoute après le *Map* une opération de regroupement par clé. Cela semble cependant un peu idiot de disperser les liens pour les regrouper ensuite. Dans tous les cas on obtient un RDD `Matrix` avec tous les vecteurs de la matrice PageRank. Chaque unité d'information (chaque « document ») dans ce RDD est donc de la forme

$$(u, V_u[u_i, u_j, \cdots])$$

où chaque u^x est un lien sortant de :math :u`. NB : on ne représente pas une matrice avec des 0 et des 1, à l'échelle du Web ce serait
— Deuxième étape : on dispose de la matrice. Il faut évaluer la probabilité d'aboutir à une page u'. Initialement, on suppose que l'on part de n'importe que page u et on simule un processus aléatoire de déplacement qui nous donne la probablité d'arriver à u'. À chaque étape i, ces probabilités sont représentés par un RDD dit « des rangs », $Rank_i$, dont chaque unité d'information (chaque « document ») est de la forme

$$(u, p_u)$$

Où p_u est la probablité d'être arrivé en u. Initialement cette probabilité dans $Rank_0$ est égale à 1 : on suppose que l'on part de u.
Effectuons une **jointure** (sur l'URL) entre `Matrix` et `Rank`. On obtient, pour chaque URL `u`, des unités d'information de la forme

$$(u, V_u[u_i, u_j, \cdots, u_k, \ldots], p_u)$$

On effectue alors un calcul simulant le choix aléatoire et équiprobable de suivre un des liens sortants de `u`. La probabilité de suivre chacun des liens sortants est $p_u/|V_u|$. On doit donc produire :

$$(u_i, p_u/|V_u|)$$
$$(u_j, p_u/|V_u|)$$
$$\dots$$
$$(u_k, p_u/|V_u|)$$

Chacune de ces paires $(u', p_u^{u'})$ est la probabilité $p_u^{u'}$ d'arriver sur l'URL *en provenance de u*. Il reste à cumuler toutes ces probabilités pour toutes les provenances possibles. **Ces paires sont produites par une opération de Map**.

Et finalement il faut obtenir la probabilité de se retrouver sur un lien `u'` en cumulant les probabilités d'arriver sur `u'` en provenance de toutes les URLs `u` dont `u'` est un lien sortant. **C'est une opération de Reduce** (ou de regroupement par clé, ce qui revient au même).

Il faut itérer un certain nombre de fois (10 ou 20 fois). À chaque itération on obtient les rangs dans nouveau RDD qui sert d'entrée à la prochaine itération.

En résumé, à chaque étape *i* :

— On effectue la jointure entre `Matrix` et $Rank_i$
— On applique un *Map* qui émet des paires $(u', p_u^{u'})$
— On applique un *Reduce* qui regroupe sur la clé u' et cumule les probabilités $p_u^{u'}$ pour toutes les provenances u

— Quels RDD marquer comme persistants ? Il faut évidemment ne pas recalculer `Matrix` à chaque itération, et le conserver en RAM pour ne pas dégrader les calculs.
Si on fait beaucoup d'itérations, il faut envisager la situation en cas de panne : sans persistance intermédiaire il faudra recommencer les calculs à zéro. On peut donc rendre pesistants les RDD stockant les calculs intermédiaires, au moins quelques-uns (1 sur 5 ?).
https://github.com/abbas-taher/pagerank-example-spark2.0-deep-dive

---

Exercice : Ex-Spark-2 qu'est-il arrivé à CQL ?

Vous avez sans doute noté que Spark surpasse CQL. On peut donc envisager de se passer de ce dernier, ce qui soulève quand même un inconvénient majeur (lequel ?). Le connecteur Spark/Cassandra permet de déléguer les transformations Spark compatibles avec CQL grâce à un paramètre *pushdown* qui est activé par défaut.

— Enoncez clairement l'inconvénient d'utiliser Spark en remplacement de CQL.
— Etudiez le rôle et fonctionnement de l'option *pushdown* dans la documentation du connecteur.
— Quelles sont les requêtes parmi celles vues ci-dessus qui peuvent être transmises à CQL ?

Correction

L'inconvénient majeur est que l'on va effectuer un transfert entre Cassandra et Spark de données qui vont ensuite être immédiatement filtrées. Il serait bien préférable d'effectuer ce filtre dès l'origine avec une requête CQL.

La fonction « pushdown » a justement pour but de transférer les clauses qui peuvent l'être de Spark vers CQL.

14.4.1 Et pour aller plus loin

Exercice Ex-Spark-3 : plans d'exécution

Avec l'interface de Spark vous pouvez consulter le graphe d'exécution de chaque traitement. Comme nous sommes passés avec l'API des *DataFrame* à un niveau beaucoup plus *déclaratif*, cela vaut la peine de regarder, pour chaque traitement effectué (et notamment la jointure) comment Spark évalue le résultat avec des opérateurs distribués.

Exercice Ex-Spark-4 : exploration de l'interface *Dataset*

L'API des *Datasets* est présentée ici :

https://spark.apache.org/docs/latest/api/scala/index.html#org.apache.spark.sql.Dataset

Etudiez et expérimentez les transformations et actions décrites.

Exercice Ex-Spark-5 : Cassandra et Spark, système distribué complet

En associant Cassandra et Spark, on obtient un environnement distribué complet, Cassandra pour le stockage, Spark pour le calcul. La question à étudier (qui peut faire l'objet d'un projet), c'est la bonne intégration de ces deux systèmes, et notamment la correspondance entre le partitionnement du stockage Cassandra et le partitionnement des calculs Spark. Idéalement, chaque fragment d'une collection Cassandra devrait devenir un fragment RDD dans Spark, et l'ensemble des fragments traités en parallèle. À approfondir !

CHAPITRE 15

Annales des examens

Les examens de NFE204 durent 3 heures, les documents et autres soutiens ne sont pas autorisés, à l'exception d'une calculatrice.

Le but de l'examen est de vérifier la bonne compréhension des concepts et techniques vus en cours. Dans les rares cas où un langage informatique est impliqué, nous n'évaluons pas les réponses par la syntaxe mais par la clarté, la concision et la précision.

15.1 Examen du 3 février 2015

15.1.1 Première partie : recherche d'information (8 pts)

Voici quelques extraits d'un discours politique célèbre (un peu modifié pour les besoins de la cause). Chaque extrait correspond à un document, numéroté a_i.

1. (a1) Moi, président de la République, je ne serai pas le chef de la majorité, je ne recevrai pas les parlementaires de la majorité à l'Elysée.
2. (a2) Moi, président de la République, je ne traiterai pas mon Premier ministre de collaborateur.
3. (a3) Moi, président de la République, les ministres de la majorité ne pourraient pas cumuler leurs fonctions avec un mandat parlementaire ou local.
4. (a4) Moi, président de la République, il y aura un code de déontologie pour les ministres et parlementaires qui ne pourraient pas rentrer dans un conflit d'intérêt.

Questions.

— Rappeler la notion de *stop word* (ou « mot vide ») et donner la liste de ceux que vous choisiriez dans les textes ci-dessus.

— Outre ces mots vides, pouvez-vous identifier certains mots dont l'idf tend vers 0 (en appliquant le logarithme) ? Lesquels ?

— Présentez la matrice d'incidence pour le vocabulaire suivant : *majorité*, *ministre*, *déontologie*, *parlementaire*. Vous indiquerez l'idf pour chaque terme (sans logarithme), et le tf pour chaque paire (terme, document). Bien entendu, on suppose que les termes ont été fait l'objet d'une normalisation syntaxique au préalable.
— Donner les résultats classés par similarité cosinus basée sur les tf (on ignore l'idf) pour les requêtes suivantes.
  — `majorité` ; expliquez le classement ;
  — `ministre` ; expliquez le classement du premier document ;
  — `déontologie et ministre` ; qu'est-ce qui changerait si on prenait en compte l'idf ?
  — `majorité et ministre` ; qu'obtiendrait-t-on avec une requête Booléenne ? Commentaire ?
— Calculez la similarité cosinus entre a3 et a4 ; puis entre a3 et a1. Qui est le plus proche de a3 ?

15.1.2 Seconde partie : Pig et MapReduce (6 pts)

Un système d'observation spatiale capte des signaux en provenance de planètes situées dans de lointaines galaxies. Ces signaux sont stockés dans une collection *Signaux* de la forme *Signaux (idPlanète, date, contenu)*.

Le but est de déterminer si ces signaux peuvent être émis par une intelligence extra-terrestre. Pour cela les scientifiques ont mis au point les fonctions suivantes :

1. *Fonction de structure* : $f_S(c)$: $Bool$, prend un contenu en entrée, et renvoie `true` si le contenu présente une certaine structure, `false` sinon.
2. *Fonction de détecteur d'Aliens* : $f_D(< c >)$: $real$, prend une liste de contenus *structurés* en entrée, et renvoie un indicateur entre 0 et 1 indiquant la probabilité que ces contenus soient écrits en langage extra-terrrestre, et donc la *présence d'Aliens* !

Bien entendu, il y a beaucoup de signaux : c'est du Big Data.

Questions.

1. Ecrire un programme Pig latin qui produit, pour chaque planète, l'indicateur de présence d'Aliens par analyse des contenus provenant de la planète.
2. Donnez un programme MapReduce qui permettrait d'exécuter ce programme Pig en distribué (indiquez la fonction de Map, la fonction de Reduce, dans le langage ou pseudo-code qui vous convient).
3. Ecrire un programme Pig latin qui produit, pour chaque planète et pour chaque jour, le rapport entre contenus structurés et non structurés reçus de cette planète.

15.1.3 Troisième partie : questions de cours (6 pts)

Concision et précision s'il vous plaît.
— En recherche d'information, qu'est-ce que le rappel ? qu'est-ce que la précision ?
— Deux techniques fondamentales vues en cours sont la réplication et le partitionnement. Rappelez brièvement leur définition, et indiquez leurs rôles respectifs. Sont-elles complémentaires ? Redondantes ?
— Qu'est-ce qu'une architecture multi-nœuds, quels sont ses avantages et inconvénients ?
— Vous avez 500 TOs de données, et vous pouvez acheter des serveurs pour votre *cloud* avec chacun 32 GO de mémoire et 10 TOs de disque. Le coût unitaire d'un serveur eest de 500 Euros.
Quelle est la configuration de votre grappe de serveurs la moins coûteuse (financièrement) et combien de temps prend au minimum la lecture complète de la collection avec cette solution ?

— Même situation : quelle est la configuration qui assure le maximum d'efficacité, et quel est son coût financier ?
— Rappelez le principe de l'éclatement d'un fragment dans le partitionnement par intervalle.

15.2 Examen du 14 avril 2015

15.2.1 Première partie : recherche d'information (8 pts)

Voici notre base documentaire :
— *d1* : Le loup et les trois petits cochons.
— *d2* : Spider Cochon, Spider Cochon, il peut marcher au plafond. Est-ce qu'il peut faire une toile ? Bien sûr que non, c'est un cochon.
— *d3* : Un loup a mangé un mouton, les autres moutons sont restés dans la bergerie.
— *d4* : Il y a trois moutons dans le pré, et un autre dans la gueule du loup.
— *d5* : L'histoire extraordinaire des trois petits loups et du grand méchant cochon.

On va se limiter au vocabulaire `loup`, `mouton`, `cochon`.
— Donnez la matrice d'incidence *booléenne* (seulement 0 ou 1) avec les termes en ligne (NB : on suppose une phase préalable de normalisation qui élimine les pluriels, majuscules, etc.)
— Expliquez par quelle technique, avec cette matrice d'incidence, on peut répondre à la requête booléenne « loup et cochon mais pas mouton ». Quel est le résultat ?
— Maintenant donnez une matrice d'incidence contenant les tf, et un tableau donnant les idf (sans le *log*), pour les trois termes précédents.
— Donner les résultats classés par similarité cosinus basée sur les tf (on ignore l'idf) pour les requêtes suivantes.
  — cochon ;
  — loup et mouton ;
  — loup et cochon.
— On ajoute le document *d6* : « Shaun le mouton : une nuit de cochon ».
  Quel est son score pour la requête « loup et mouton », quel autre document a le même score et qu'est-ce qui change si on prend en compte l'idf ?
— Pour la requête « loup et cochon » et les documents d1 et d5, qu'est-ce qui change si on ne met pas de restriction sur le vocabulaire (tous les mots sont indexés) ?

15.2.2 Seconde partie : Pig et MapReduce (6 pts)

Une organisation terroriste, le Spectre, envisage de commettre un attentat dans une station de métro. Heureusement, le MI5 dispose d'une base de données d'échanges téléphoniques et ses experts ont mis au point un décryptage qui identifie la probabilité qu'un message provienne du Spectre d'une part, et fasse référence à une station de métro d'autre part.

Après décryptage, les messages obtenus ont la forme suivante :

```
{
 "id": "x1970897",
 "émetteur": "Joe Shark",
```

(suite sur la page suivante)

(suite de la page précédente)

```
"contenu": "Hmm hmm hmmm",
"probaSpectre": 0.6,
"station": "Covent Garden"
```

Il y en a des milliards : vous avez quelques heures pour trouver la solution.

1. Spécifier les deux fonctions du programme MapReduce qui va identifier la station-cible la plus probable. Ce programme doit sélectionner les messages émis par le Spectre avec une probabilité de plus de 70%, et produire, pour chaque station le nombre de tels messages où elle est mentionnée.
2. Quel est le programme Pig qui exprime ce traitement MapReduce ?
3. On veut connaître les 5 mots les plus couramment employés par chaque émetteur dans le contenu de ses messages. Expliquez, avec la formalisation de votre choix, comment obtenir cette information (vous avez le droit d'utiliser un opérateur de tri).

15.2.3 Troisième partie : questions de cours (6 pts)

— En recherche d'information, que signifient les termes « faux positifs » et « faux négatifs » ?
— Je soumets une requête $t_1, t_2, \cdots, t_n$. Quel est le poids de chaque terme dans le vecteur représentant cette requête ? La normalisation de ce vecteur est elle importante pour le classement (justifier) ?
— Donnez trois bonnes raisons de choisir un système relationnel plutôt qu'un système NoSQL pour gérer vos données.
— Donnez trois bonnes raisons de choisir un système NoSQL plutôt qu'un système relationnel pour gérer vos données.
— Rappeler la règle du quorum (majorité des votants) en cas de partitionnement de réseau, et justifiez-la.
— Dans un traitement MapReduce, peut-on toujours se contenter d'un seul *reducer* ? Avantages ? Inconvénients ?

15.3 Examen du 15 juin 2015

15.3.1 Première partie : documents structurés (6 pts)

Une application s'abonne à un flux de nouvelles dont voici un échantillon.

```
<myrss version="2.0">
  <channel>
    <item>
        <title>Angela et nous</title>
        <description>Un nouveau sommet entre la France et l'Allemagne
            est prévu en Allemagne la semaine prochaine pour relancer l'UE.
        </description>
        <links>
            <link>lemonde.fr</link>
            <link>lesechos.fr</link>
        </links>
```

(suite sur la page suivante)

(suite de la page précédente)

```
    </item>
    <item>
        <title>L'Union en crise</title>
        <description> L'Allemagne, la France, l'Italie doivent à nouveau se
            réunir pour étudier les demandes de la Grèce.
        </description>
        <links>
            <link>lefigaro.fr</link>
        </links>
    </item>
    <item>
        <title>Alexis à l'action</title>
        <description>Le nouveau premier ministre de la Grèce
            a prononcé son discours d'investiture.
        </description>
        <links>
            <link>libe.fr</link>
        </links>
    </item>
    <item>
        <title>Nos cousins transalpins</title>
        <description>La France et l'Italie partagent plus qu'une frontière
            commune: le point de vue de l'Italie.
        </description>
        <links>
            <link>courrier.org</link>
        </links>
    </item>
  </channel>
</myrss>
```

Chaque nouvelle (`item`) résume donc un sujet et propose des liens vers des médias où le sujet est développé. Les quatre éléments `item` seront désignés respectivement par d1, d2, d3 et d4.

1. Donnez la forme arborescente de ce document (ne recopiez pas tous les textes : la structure du document suffit).
2. Proposez un format JSON pour représenter le même contenu (idem : la structure suffit).
3. Dans une base relationnelle, comment modéliser l'information contenue dans ce document ?
4. Quelle est l'expression XPath pour obtenir tous les éléments `link` ?
5. Quelle est l'expression XPath pour obtenir les titres des items dont l'un des `link` est `lemonde.fr` ?

15.3.2 Deuxième partie : recherche d'information (6 pts)

On veut indexer les nouvelles reçues de manière à pouvoir les rechercher en fonction d'un pays. Le vocabulaire auquel on se restreint est donc celui des noms de pays (Allemagne, France, Grèce, Italie).

1. Donnez une matrice d'incidence contenant les tf pour les quatre pays, et un tableau donnant les idf (sans appliquer le *log*). Mettez les noms de pays en ligne, et les documents en colonne.
2. Donner les résultats classés par similarité cosinus basée sur les tf (on ignore l'idf) pour les requêtes suivantes. Expliquez brièvement le classement.
   — Italie ;
   — Allemagne et France ;
   — France et Grèce.
3. Reprenons la dernière requête, « France et Grèce » et les documents d2 et d3. Qu'est-ce que la prise en compte de l'idf changerait au classement ?

15.3.3 Troisième partie : MapReduce (4 pts)

Voici quelques analyses à exprimer avec MapReduce et Pig.

1. On veut compter le nombre de nouvelles consacrées à la Grèce publiées par chaque média. Décrivez le programme MapReduce (fonctions de Map et de Reduce) qui produit le résultat souhaité. Donnez le pseudo-code de chaque fonction, ou indiquez par un texte clair son déroulement.
   Vous disposez d'une fonction *contains(:math : `texte, :math : `mot)* qui renvoie vrai si `:math:`texte` contient `:math:`mot`.
2. Quel est le programme Pig qui exprime ce traitement MapReduce ?

15.3.4 Quatrième partie : questions de cours (6 pts)

Questions reprises des Quiz.

15.4 Examen du 1er juillet 2016 (FOD)

15.4.1 Exercice corrigé : documents structurés et MapReduce (8 pts)

Note : Ce premier exercice (légèrement modifié et étendu) est corrigé entièrement. Les autres exercices consistaient en un énoncé classique de recherche d'information (8 points) et 4 brèves questions de cours (4 points).

Le service informatique du Cnam a décidé de représenter ses données sous forme de documents structurés pour faciliter les processus analytiques. Voici un exemple de documents centrés sur les étudiant.e.s et incluant les Unités d'Enseignement (UE) suivies par chacun.e.

```
[{
   "_id": 978,
   "nom": "Jean Dujardin",
   "annee": "2016",
     "UE": [{"ue":11, "note": 12,
              {"ue":27, "note": 17,
               {"ue":37,  "note": 14
              ]
  ,
 {
 "_id": 476,
 "nom": "Vanessa Paradis",
 "annee": "2016",
 "UE": [{"ue": 13, "note": 17,
         {"ue":27, "note": 10,
         {"ue":76,  "note": 11
       ]

]
```

15.4.2 Question 1 : documents et base relationnelle

Question

Sachant que ces documents sont produits à partir d'une base relationnelle, reconstituez le schéma de cette base et indiquez le contenu des tables correspondant aux documents ci-dessus.

Il y a clairement une table `Inscription (id, nom année)` et une table `Note (idInscription, ue, note)`. Il y a probablement aussi une table `UE` mais elle n'est pas strictement nécessaire pour produire le document ci-dessus.

15.4.3 Question 2 : restructuration de documents

Question

Proposez une autre représentation des mêmes données, centrée cette fois, non plus sur les étudiants, mais sur les UEs.

Voilà, à compléter avec les UEs 13 et 37.

```
[
 {
   "_id": 387,
```

(suite sur la page suivante)

(suite de la page précédente)

```
    "UE": 11,
    "inscrits": [
       {"nom": "Jean Dujardin", "annee": "2016", "note": 12
     ]
    ,
    {
    "_id": 3809,
    "UE": 27,
    "inscrits": [
       {"nom": "Jean Dujardin", "annee": "2016", "note": 17,
       {"nom": "Vanessa Paradis", "annee": "2016", "note": 10,
     ]
    ,
    {
    "_id": 987,
    "UE": 76,
    "inscrits": [
       {"nom": "Vanessa Paradis", "annee": "2016", "note": 11
     ]

]
```

15.4.4 Question 3 : MapReduce et la notion de document « autonome »

Question

On veut implanter, par un processus MapReduce, le calcul de la moyenne des notes d'un étudiant. Quelle est la représentation la plus appropriée parmi les trois précédentes (une en relationnel, deux en documents structurés), et pourquoi ?

La première représentation est très bien adaptée à MapReduce, puisque chaque document contient l'intégralité des informations nécessaires au calcul. Si on prend le document pour Jean Dujardin par exemple, il suffit de prendre le tableau des UE et de calculer la moyenne. Donc, pas besoin de jointure, pas besoin de regroupement. Le calcul peut se faire intégralement dans la fonction de Map, et la fonction de Reduce n'a rien à faire.

C'est l'iiustration de la notion de *document autonome* : pas besoin d'utiliser des références à d'autres documents (ce qui mène à des jointures en relationnel) ou de distribuer l'information nécsesaire dans plusieurs documents (ce qui mène à des regroupements en MapReduce).

Si on a choisit de construire les documents structurés en les centrant sur le UEs, il y a beaucoup plus de travail, comme le montre la question suivante.

15.4.5 Question 4 : MapReduce, outil de restructuration/regroupement

Question

Spécifiez le calcul du nombre d'étudiants par UE, en MapReduce, en prenant en entrée des documents centrés sur les étudiants (exemple donné ci-dessus).

Cette fois, il va falloir utiliser toutes les capacités du modèle MapReduce pour obtenir le résultat voulu. Comme suggéré par la question précédente, la représentation centrée sur les UE serait beaucoup plus appropriée pour disposer d'un document *autonome* contenant toutes les informations nécessaires. C'est exactement ce que l'on va faire avec MapReduce : transformer la représentation centrée sur les étudiants en représentation centrée sur les UEs, le reste est un jeu d'enfant.

La fonction de Map

Une fonction de Map produit des paires (clé, valeur). La première question à se poser c'est : quelle est la clé que je choisis de produire ? Rappelons que la clé est une sorte d'étiquette que l'on pose sur chaque valeur et qui va permettre de les regrouper.

Ici, on veut regrouper par UE pour pouvoir compter tous les étudiants inscrits. On va donc émettre une paire intermédiaire pour chaque UE mentionnée dans un document en entrée. Voici le pseudo-code.

```
function fonctionMap (:math:`doc) # doc est un document centré étudiant
{
  # On parcourt les UEs du tableau UEs
  for    [:math:`ue in :math:`doc.UEs] do
     emit (:math:`ue.id, :math:`doc.nom)
  done
```

Quand on traite le premier document de notre exemple, on obtient donc trois paires intermédiaires :

```
{"ue:11", "Jean Dujardin"
{"ue:27", "Jean Dujardin"
{"ue:37", "Jean Dujardin"
```

Et quand on traite le second document, on obtient :

```
{"ue:13", "Vanessa Paradis"
{"ue:27", "Vanessa Paradis"
{"ue:76", "Vanessa Paradis"
```

Toutes ces paires sont alors transmises à « l'atelier d'assemblage » qui les regroupe sur la clé. Voici la liste des groupes (un par UE).

```
{"ue:11", ["Jean Dujardin"]
{"ue:13", "Vanessa Paradis"
{"ue:27", ["Jean Dujardin", "Vanessa Paradis"]
```

(suite sur la page suivante)

(suite de la page précédente)

```
{"ue:37", "Jean Dujardin"
{"ue:76", "Vanessa Paradis"
```

Il reste à appliquer la fonction de Reduce à chaque groupe.

```
function fonctionReduce (:math:`clé, :math:`tableau)
{
  return (:math:`clé, count(:math:`tableau)
```

Et voilà.

15.4.6 Question 5 : MapReduce = group-by SQL

Question

Quelle serait la requête SQL correspondant à ce dernier calcul sur la base relationnelle ?

Avec SQL, c'est direct, à condition d'accepter de faire des jointures.

```
select u.id, u.titre, count(*)
from Etudiant as e, Inscription as i, UE as u
where e.id = i.idEtudiant
and   u.id = i.idUE
group by u.id, u.titre
```

15.5 Examen du 1er février 2017 (Présentiel)

15.5.1 Première partie : documents structurés (5 pts)

Un institut chargé d'analyser l'opinion publique collecte des articles parus dans la presse en ligne, ainsi que les commentaires déposés par les internautes sur ces articles. Ces informations sont stockées dans une base relationnelle, puis mises à disposition des analystes sous la forme de documents JSON dont voici deux exemples.

```
{
 "_id": 978,
"_source": "lemonde.fr",
"date": "07/02/2017",
"titre": "Le président Trump décide d'interdire l'entrée de tout étranger aux USA
↪",
"contenu": "Suite à un décret paru ....",
"commentaires": [
     {"internaute": "chocho@monsite.com",
```

(suite sur la page suivante)

(suite de la page précédente)

```
    "note": 5,
    "commentaire": "Les décisions du nouveau président sont inquiétantes ..."
    ,
    {"internaute": "alain@dugenou.com",
    "note": 2,
    "commentaire": "Arrêtons de critiquer ce grand homme..."

  ]

{
"_id": 54,
"source": "nimportequoi.fr",
"date": "07/02/2017",
"titre": "La consommation de pétrole aide à lutter contre le réchauffement",
"contenu": "Contrairement à ce qu'affirment les media officiels ....",
"commentaires": [
    {"internaute": "alain@dugenou.com",
     "note": 5,
    "commentaire": "Enfin un site qui n'a pas peur de dire la vérité ..."

 ]
```

Les notes vont de 1 à 5, 1 exprimant un fort désacord avec le contenu de l'article, et 5 un accord complet.

A) À votre avis, quel est le schéma de la base relationnelle d'où proviennent ces documents ? Montrez comment les informations des documents ci-dessus peuvent être représentés avec ce schéma (ne recopiez pas tous les textes, donnez les tables avec quelques lignes montrant la répartition des données).

B) On collecte des informations sur les internautes (année de naissance, adresse). Où les placer dans la base relationnelle ? Dans le document JSON ?

C) On veut maintenant obtenir une représentation JSON centrée sur les internautes et pas sur les sources d'information. Décrivez le processus Map/Reduce qui transforme une collection de documents formatés comme les exemples ci-dessus, en une collection de documents dont chacun donne les commentaires déposés par un internaute particulier.

15.5.2 Deuxième partie : recherche d'information (4 pts)

On veut indexer les articles pour pouvoir les analyser en fonction des candidats qu'ils mentionnent. On s'intéresse en particulier à 4 candidats : Clinton, Trump, Sanders et Bush. Voici 4 extraits d'articles (ce sont nos documents d_1, d_2, d_3, d_4).

— L'affrontement entre Trump et Clinton bat son plein. Clinton a-t-elle encore une chance ?
— Tous ces candidats, Clinton, Trump, Sanders et Bush, semblent encore en mesure de l'emporter.
— La surprise, c'est Sanders, personne ne l'attendait à ce niveau.
— Ce que Bush pense de Trump ? A peu de choses près ce que ce dernier pense de Bush.

Questions :

A) Donnez une matrice d'incidence contenant les tf pour les quatre candidats, et un tableau donnant les idf (sans appliquer le $\log$). Mettez les noms de candidats en ligne, et les documents

en colonne.

Réponse :

	d1	d2	d3	d4
Clinton (2)	2	1	0	0
Trump (4/3)	1	1	0	1
Sanders (2)	0	1	1	0
Bush (2)	0	1	0	2

B) Donner les résultats classés par similarité cosinus basée sur les tf (on ignore l'idf) pour les requêtes suivantes. Expliquez brièvement le classement.
— Bush
— Trump et Clinton
— Trump et Sanders

Réponse :

Les normes
— $||d_1|| = \sqrt{4+1} = \sqrt{5}$
— $||d_2|| = \sqrt{1+1+1+1} = 2$
— $||d_3|| = \sqrt{1}$
— $||d_4|| = \sqrt{1+4} = \sqrt{5}$

Les cosinus (requête non normalisée).

— Bush : d2 : $\frac{1}{2} = 0,5$; d4 : $\frac{2}{\sqrt{5}} \simeq 0,89$; d4 est premier car il mentionne deux fois Bush.
— Trump et Clinton : d1 : $\frac{2+1}{\sqrt{5}}$; d2 : $\frac{1+1}{2}$; d4 : $\frac{1}{\sqrt{5}}$;
d1 est premier comme on pouvait s'y attendre : il parle exclusivement de Trump et Clinton. Viennent ensuite d2 puis d4.
— Trump et Sanders
  — d1 : $\frac{1}{\sqrt{5}}$
  — d2 : $\frac{2}{2}$
  — d3 : $\frac{1}{1}$
  — d4 : $\frac{1}{\sqrt{5}}$

  d2 et d3 arrivent à égalité. Intuitivement, il parlent tous les deux « à moitié » de Trump et Sanders. d1 et d4 parlent de Trump *ou* de Sanders et aussi des autres candidats.

C) Reprenons la requête, « Trump et Clinton » et le premier document du classement. Quelle légère modification de ce document lui donnerait une mesure cosinus encore plus élevée ? Expliquez pourquoi.

Réponse : Il suffit d'ajouter une fois la mention de Trump.

1) Je fais une recherche sur « Sanders ». Pouvez-vous indiquer le classement sans faire aucun calcul ? Expliquez.

Réponse : Sanders apparaît dans les documents d2 et d3, et d3 ne parle que de lui alors que d@ parle de tous les candidats. D'où le classement.

15.5.3 Troisième partie : systèmes distribués (3 pts)

On considère un système Cassandra avec un facteur de réplication *F= 3* (donc, 3 copies d'un même document). Appelons *W* le nombre d'acquittements reçus pour une écriture, *R* le nombre d'acquittements reçus pour une lecture.

A) Décrivez brièvement les caractéristiques des configurations suivantes :
   — R=1, W=3
   — R=1,W=1
   Réponse : La première configuration assure des écritures synchrones, et se contente d'une lecture qui va prendre indifféremment l'une des trois versions. La lecture est cohérente.
   La seconde privilégie l'efficacité. On aquitte après une seule écriture, on lit une seule copie (peut-être pas la plus récente).

B) Quelle est la formule sur *R, W* et *F* qui assure la cohérence immédiate (par opposition à la cohérence à terme) du système ? Expliquez brièvement.

   Réponse : $W+R = F+1$. Voir le cours.

15.5.4 Quatrième partie : MapReduce (4 pts)

Toujours sur nos documents donnés en début d'énoncé (première partie) : on veut analyser, pour la source « lemonde.fr », le nombre de commentaires ayant obtenus respectivement 1, 2, 3, 4 ou 5.

— Décrivez la modélisation MapReduce de ce calcul. Donnez le pseudo-code de chaque fonction, ou indiquez par un texte clair son déroulement.
— Donnez la forme de la chaîne de traitement (workflow), avec des opérateurs Pig ou Spark, qui implante ce calcul.

15.5.5 Cinquième partie : questions de cours (4 pts)

— Qu'entend-on par « bag of words » pour le modèle des documents textuels en recherche d'information ?
— Expliquez la notion de noeud virtuel dans la distribution par *consistent hashing*.
— Que signifie, pour une structure de partitionnement, être *dynamique*. Avons-nous étudié un système où le partitionnement n'était pas dynamique ?
— Donnez une définition de la scalabilité.

15.6 Examen du 6 février 2018 (Présentiel)

15.6.1 Première partie : modélisation NoSQL (7 pts)

Pour cette partie, nous allons nous pencher sur le petit tutoriel proposé par la documentation en ligne de Cassandra, et consacré à la modélisation des données dans un contexte BigData. L'application (très simplifiée) est un service de musique en ligne, avec le modèle de données de la figure Fig. 15.1.

On a donc des chansons, chacune écrite par un artiste, et des *playlists*, qui consistent en une liste ordonnée de chansons.

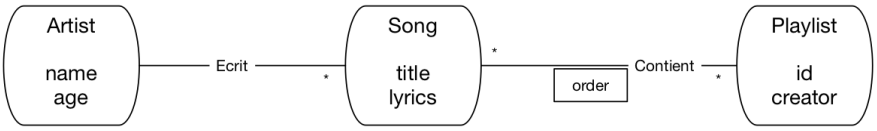


Fig. 15.1 – Le cas d'école Cassandra

— Commencer par proposer le schéma relationnel correspondant à ce modèle. Il est sans doute nécessaire d'ajouter des identifiants. Donnez les commandes SQL de création des tables. (1 pt).

Réponse :

```
create table Artiste (id int not null, name varchar(50), age int,␣
↪primary key(id))
create table Song (id int not null, title varchar(50), lyrics text,␣
↪primary key(id))
create table Playlist (id int not null, creator varchar(50), primary␣
↪key(id))
create table SongInPlaylist (id_playlist int not null,
   id_song int not null, position int, primary key(id_playlist, id_
↪song))
```

— Le tutoriel Cassandra nous explique qu'il faut concevoir le schéma Cassandra en fonction des *access patterns*, autrement dit des requêtes que l'on s'attend à devoir effectuer. Voici les deux *access patterns* envisagés :
  — *Find all song titles*
  — *Find all songs titles of a particular playlist*

Lesquels de ces *access patterns* posent potentiellement problème avec un système relationnel dans un contexte *BigData* et pourquoi ? Vous pouvez donner les requêtes SQL correspondantes pour clarifier votre réponse (1 pt).

Réponse : le premier implique un parcours séquentiel de la table `Song` : à priori un système relationnel peut faire ça très bien. La seconde implique une jointure : les systèmes relationnels font ça très bien aussi mais ça ne passe pas forcément à très grande échelle. C'est en tout cas l'argument des systèmes NoSQL.

— Le tutoriel Cassandra nous propose alors de créer une unique table

```
CREATE TABLE playlists (
  id_playlist int,
creator text,
song_order int,
song_id int,
title text,
lyrics text,
name text,
age int,
PRIMARY KEY  (id_playlist, song_order ) );
```

Discutez des avantages et inconvénients en répondant aux questions suivantes : combien faut-il d'insertions (au pire) pour ajouter une chanson à une *playlist*, en relationnel et dans

Cassandra ? Que peut-on dire des requêtes qui affichent une *playlist*, respectivement en relationnel et dans Cassandra (donnez la requête si nécessaire) ? Combien de lignes dois-je mettre à jour quand l'âge d'un artiste change, en relationnel et en Cassandra ? Conclusion ? (2 pts)

Réponse : Il faut (au pire, c'est à dire si la chanson, l'artiste, la playlist n'existent pas au préalable) 4 insertions en relationnel, une seule avec Cassandra. Pour la recherche, jointures indispensables en relationnel. Pour les mises à jour en revanche, en relationnel, il suffit juste de mettre à jour la ligne de l'artiste dans la table Artist. En Cassandra il faudra mettre à jour toutes les lignes contenant l'artiste dans la table `Playlists`.

— Vous remarquez que l'identifiant de la table est composite (*compound* en anglais) : `(id_playlist, song_order )`. Voici ce que nous dit le tutoriel :
« *A compound primary key consists of the partition key and the clustering key. The partition key determines which node stores the data. Rows for a partition key are stored in order based on the clustering key.* »
Sur la base de cette explication, quelles affirmations sont vraies parmi les suivantes :

- — Une chanson est stockée sur un seul serveur (vrai/faux) ?
- — Les chansons d'une même *playlist* sont toutes sur un seul serveur (vrai/faux) ?
- — Les chansons stockées sur un serveur sont triées sur leur identifiant (vrai/faux) ?
- — Les chansons d'une même *playlist* sont stockées les unes après les autres (vrai/faux) ?

Réponse : réponses 2 et 4. L'identifiant de la *playlist* définit le serveur de stockage. De plus, les chansons d'une même *playlist* sont stockées dans l'ordre et contigument. Cf. Fig. 15.2.

Faites un petit dessin illustrant les caractéristiques du stockage des *playlists* dans un système Cassandra distribué (2 pts).

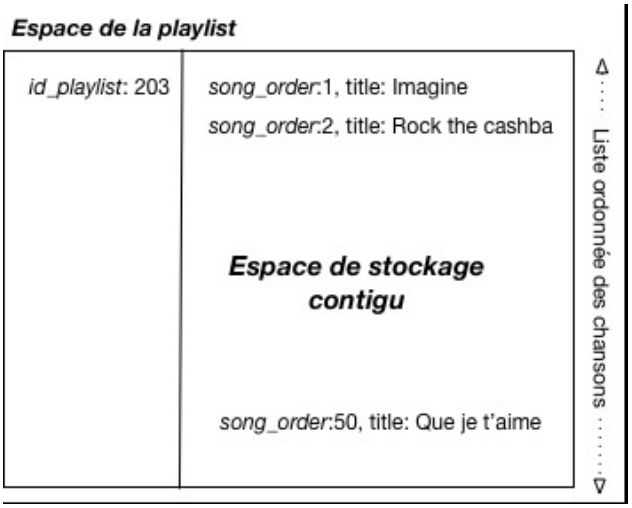


Fig. 15.2 – Le stockage optimal d'une *playlist* dans Cassandra

— Pour finir, reprenez les *access patterns* donnés initialement. Lesquels vont pouvoir être évalués très efficacement avec cette organisation des données, lesquels posent des problèmes potentiels de cohérence (1 pt) ?

Réponse : réponses 2 et 4. L'*access patterns* qui peut être évalué facilement est « *find all songs titles of a particular playlist* » car il suffit d'avoir `id_playlist` pour afficher les chansons. L'évaluation de l'autre pattern est plus difficile et surtout plus longue car il faut parcourir tous les enregistrements et ensuite retirer les doublons pour pouvoir afficher toutes les chansons de la base

15.6.2 Deuxième partie : recherche d'information (5 pts)

On veut maintenant équiper notre système d'une fonction de recherche plein texte.

— Un premier essai avec le langage CQL de Cassandra est évalué à partir d'un jeu de tests. On obtient les indicateurs du tableau suivant :

	Pertinent	Non pertinent
Positif	200	50
Négatif	100	1800

Sur 250 chansons ramenées dans le résultat, 200 sont pertinentes, 50 ne le sont pas, et il en manque en revanche 100 qui seraient pertinentes.
Quel est le rappel de votre système ? Quelle est sa précision ? (1 pt)

Correction

Précision : proportion de vrais positifs. Rappel : proportion de documents pertinents dans le résultat

— Précision = 200 / (200 + 50)
— Rappel = 200 / (200 + 100)

— Essayons de faire mieux en associant Cassandra à ElasticSearch pour pouvoir faire de la recherche avec classement. Voici quelques extraits de grandes chansons françaises.
  — *Y a vraiment qu'l'amour qui vaille la peine*
  — *Que je t'aime, que je t'aime, que je t'aime*
  — *Dix ans de chaînes sans voir le jour, c'était ma peine forçat de l'amour*
  — *C'est (c'est) pas la peine c'est (c'est) (c'est) pas la peine*

On va considérer que la phase d'indexation unifie les termes « amour » et « aime ». Sans faire de calcul, expliquez le résultat attendu (classement compris) pour les recherches suivantes (2 pts)

— amour
— peine
— peine et aime

Correction

Avec l'hypothèse que la fréquence des termes (Term frequency) possède un poids pour la similarité de la requête. Les phrases dont aucun mot ne correspond à la requête ne sont pas concernés par le classement.

— Requête « Amour » : Classement proposé - Phrase 2 car « Amour » est présent 3 fois, suivi du même score pour Phrase 1 et Phrase 3

— Requête « Peine » : Classement proposé - Phrase 4 car « Peine » est présent 2 fois, suivi de Phrase 1 et Phrase 3. La phrase 3 est plus longue donc moins spécifique au mot « peine » : elle sera certainement classée en dernier.
— Requête « Amour » et « Peine » : Classement proposé - Phrase 1 et phrase 3 car possèdent les 2 mots de la requête ; puis Phrase 2 car possède 2 fois le terme « Amour », et enfin Phrase 4 car possède 2 fois « Peine ».

— Un de vos collègues n'a pas suivi le cours NFE204 et vous affirme que la similarité cosinus entre un vecteur-document v et un vecteur-requête q est définie par la formule suivante :

$$cos\theta = \sum_{i=1}^{n} v[i] \times q[i]$$

Où $x[i]$ désigne le nombre d'occurrences du *i*ème terme dans un vecteur *x*. Expliquez pourquoi votre collègue a tort, et prouvez en prenant un des exemples de recherche ci-dessus que le résultat avec cette formule serait différent de celui attendu (2 pts).

Correction

Le calcul de la similarité cosinus correspond à la valeur de la projection des vecteurs « normalisé ». C'est donc la proximité des vecteurs qui est recherché avec le calcul de la similarité Cosinus. Plus les vecteurs sont proches, plus la similarité calculé sera importante.

Le nombre d'occurences de chaque terme (fréquence dans un document et dans l'ensemble des documents) permet de construire les vecteurs descripteurs.

Dans le cas de la présence multiple d'un terme, celui-ci aurait un poids plus important. Avec l'exemple de la requête « Amour » et « Peine », la phrase 2 obtiendrait le plus le poids dans le classement avec la formule, car « Amour » est présent 3 fois.

15.6.3 Troisième partie : Comprendre MapReduce tu devras (5 pts)

À une époque très lointaine, en pleine guerre intergalactique, deux Jedis isolés ne peuvent communiquer que par des messages cryptés. Le protocole de cryptage est le suivant : le **vrai** message est mélangé à *N* **faux** messages, *N* étant très très grand pour déjouer des services de décryptage de l'Empire. Les messages sont tous découpés en mots, et l'ensemble est transmis en vrac sous la forme suivante :

```
{"idMessage": "Xh9788&&", "mot": "force"
```

Les Jedi disposent d'une fonction secrète $f()$ qui prend l'identifiant d'un message et renvoie **vrai** ou **faux**.

— Vous devez fournir à Maître Y. le programme MapReduce qui reconstituera le contenu des **vrais** messages envoyés par Obiwan K. à partir d'un flux massifs de documents ayant la forme précédente. On accepte pour l'instant que les mots d'un message ne soient pas dans l'ordre. Donnez ce programme sous la forme que vous voulez, pourvu que ce soit clair (1 pt).

Correction

Le programme « fonction secrète », va donc prendre tous les couples (clé, mots) `{"idMessage": "Xh9788&&", "mot": "force"` en entrée, afin de concaténer les paires(k,[v]) par une fonction reducer. Le reducer va concatener les mots sans ordre particulier.
La fonction Map réalise le filtrage :

```
Fmap(d)

   si ffiltre(d.idMessage) alors emit('motsDuMessage',d.mot)
```

La fonction Reduce réassemble le message :

```
Fred(clé,<mots>)

    message = freorder(<d>)
    return message
```

— Quel est à votre avis (expliquez) le degré maximal de parallélisme que l'on peut obtenir pour ce traitement (2 pts) ?

Correction

Le degré maximal de parallélisme est limité par le nombre de groupes transmis à la phase de Reduce. Ici il n'y a qu'un seul groupe donc aucun parallélisme possible.

— Notez que, même après reconstitution, les mots d'un message sont dans n'importe quel ordre. Maître Y. a la faculté unique de lire et d'écrire des messages dans n'importe quel ordre, mais comment remettre dans l'ordre les mots des messages reçus par Obiwan K. ? Proposez une extension de la forme des messages et du programme MapReduce pour que chaque message soit reconstitué dans l'ordre (2 pts).

Correction

Les paires ainsi créés seront de type :

```
[{"idMessage": "Xh9788&&", "pos": {"indice": "1", "mot": "la",
 {"idMessage": "Xh9788&&", "pos": {"indice": "3", "mot": "force",
 {"idMessage": "Xh9788&&", "pos": {"indice": "9", "mot": "avec",
 {"idMessage": "Xh9788&&", "pos": {"indice": "14", "mot": "toi"
]
```

Puis le reducer va recevoir les paires : { « idMessage » : « Xh9788&& », { « place » : « 1 », « mot » : « la », puis va les ordonner en fonction de leur « place », pour enfin reconstrituer la phrase. Le reducer devra également combiner plusieurs fonctions. Ordonner les valeurs, puis concaténer les mots.

Quatrième partie : questions de cours (3 pts)

— Dans quelle architecture distribuée peut-on aboutir à des écritures conflictuelles ? Donnez un exemple.

— Que signifie, pour une structure de partitionnement, être *dynamique*. Avons-nous étudié un système où le partitionnement n'était pas dynamique ?
— Quel est le principe de la reprise sur panne dans Spark ?

15.7 Examen du 30 juin 2020

En raison du COVID19, cet examen s'est tenu à distance. Les documents étaient donc implicitement autorisés.

15.7.1 Première partie : modélisation NoSQL (8 pts)

En période d'épidémie, nous voulons construire un système de prévention. Ce système doit être informé rapidement des nouvelles infections détectées, retrouver et informer rapidement les personnes ayant rencontré récemment une personne infectée, et détecter enfin les foyers infectieux (*clusters*). On s'appuie sur une application installée sur les téléphones mobiles. Elle fonctionne par *bluetooth* : quand deux personnes équipées de l'application sont proches l'une de l'autre, l'une d'entre elles (suivant un protocole d'accord) envoie au serveur un *message de contact* dont voici trois exemples :

```
{"_id": "7ytGy", "pseudo1": "xuyh57", "pseudo2": "jojoXYZ", "date": "30/06/2020"
{"_id": "ui9xiuu", "pseudo1": "jojoXYZ", "pseudo2": "tat37HG", "date": "30/06/
↪2020"
{"_id": "Iuuu76", "pseudo1": "jojoXYZ", "pseudo2": "xuyh57", "date": "30/06/2020"
```

Ce message contient donc un identifiant unique, les pseudonymes des deux personnes et la date de la rencontre. Un pseudonyme est une clé chiffrée identifiant une unique personne. Pour des raisons de sécurité, un nouveau pseudo est généré régulièrement et chaque application conserve *localement* la liste des pseudos engendrés. Les messages de contact (mais pas les listes de pseudos) sont stockés sur un serveur central dans une base DB_1.

Questions

— DB_1 est asymétrique (un pseudo est stocké parfois dans le champ `pseudo1`, parfois dans le champ `pseudo2`) et redondante puisque chaque contact apparaît plusieurs fois si deux personnes se sont rencontrées souvent dans la journée (cf. les exemples ci-dessus). Proposez un traitement de type MapReduce qui produit, à partir de la base DB_1, une base DB_2 des *rencontres quotidiennes* contenant un document par pseudo et par jour, avec la liste des contacts effectués ce jour-là. Pour les exemples précédents, on devrait obtenir en ce qui concerne `jojoXYZ` :

```
{ "pseudo": "jojoXYZ",
  "date": "30/06/2020",
  "contacts": [{"pseudo": "tat37HG", "count": 1,
                {"pseudo": "xuyh57", "count": 2
               ]
```

Vous disposez d'une fonction *groupby()* qui prend un ensemble de valeurs et produit une liste contenant chaque valeur et son nombre d'occurrences. Par exemple $groupby(x, y, x, z, y, x) = [(x, 3), (y, 2), (z, 1)]$. Attention à l'asymétrie de représentation des pseudos dans les messages.
Voici quatre documents. les deux premiers sont stockés sur le serveur S_1, le troisième sur le serveur S_2, et le dernier sur le serveur S_3.

— d_1 : {"_id", "pseudo1": "X1", "pseudo2": "X2", "date": "30/06/2020"}
— d_2 : : {"_id", "pseudo1": "X3", "pseudo2": "X2", "date": "30/06/2020"}
— d_3 : : {"_id", "pseudo1": "X2", "pseudo2": "X4", "date": "30/06/2020"}
— d_4 : : {"_id", "pseudo1": "X1", "pseudo2": "X4", "date": "30/06/2020"}
Inspirez-vous de la figure Fig. 11.13. pour montrer le déroulement du traitement MapReduce avec deux *reducers*.
Initialement la base DB_2 est relationnelle : quel est son schéma pour pouvoir représenter correctement le contenu des documents des rencontres quotidiennes ?
Vérifiez que votre schéma est correct en donnant le contenu des tables pour la rencontre du pseudo « jojoXYZ » le 30 juin (document ci-dessus).
Maintenant DB_2 est une base NoSQL documentaire et les rencontres sont stockées dans une collection `Rencontres`. Quand une personne est infectée, elle transmet au serveur la liste de ses pseudos. Voici par exemple la liste des pseudos conservés sur mon application mobile :

```
{"mesPseudos": ["jojoXYZ", "johj0N", "fjhukij87", "kodhvy"]
```

Tous les jours on stocke les listes dans une collection `Infections`. Donnez la chaîne de traitement qui construit la collection de tous les pseudos qui ont été en contact avec une personne infectée.
Pour spécifier les chaînes de traitement, donner les fonctions de Map et de Reduce, en javascript, en Pig, en pseudo-code, au pire en langage naturel en étant le plus précis possible.

Correction

— Appelons ce traitement `Rencontres`. Voici la fonction de Map. La clé est constituée de deux champs, et il faut bien penser à emettre deux messages pour corriger l'asymétrie des messages de rencontre.

```
function mapRencontre (doc)
   {
       emit ([doc.pseudo1, doc.date], "pseudo": doc.pseudo2)
       emit ([doc.pseudo2, doc.date], "pseudo": doc.pseudo1)
```

La fonction de Reduce se contente de renvoyer la liste obtenue après élimination des doublons.

```
function reduceRencontre ([pseudo, date], listeContacts)
{
    return ([pseudo, date], groupby (listeContacts))
```

Pour la base relationnelle, il faut une table des rencontres quotidiennes :

```
create table Rencontre (
    id: int not null,
    pseudo: int no null,
    dateRencontre: date not null,
    primary key (id)
    unique (pseudo, dateRencontre)
 )
```

NB : on peut aussi prendre comme clé primaire la paire `(pseudo, dateRencontre)` Et une table représentant les contacts

```
create table Contacts (
```

(suite sur la page suivante)

(suite de la page précédente)

```
    idRencontre: int not null,
    pseudo: int no null,
    count: varchar,
    primary key (idRencontre, pseudo),
    foreign key (idRencontre) references Rencontre
    )
```

Prenons Pig. Il faut commencer par un `flatten` sur les listes de pseudos pour obtenir une collection donnant tous les pseudos des personnes infectées.

```
pseudosInfectés = foreach Infection generate "pseudoInfecté" ,␣
↪flatten(mesPseudos);
```

En effectuant la jointure entre `pseudosInfectés` et `Rencontre` on obtient les contacts à prévenir.

```
contactsAPrevenir = join Rencontre by "pseudo" , pseudosInfectés by
↪"pseudoInfecté";
```

15.7.2 Deuxième partie : recherche d'information (5 pts)

La base des rencontres peut être représentée comme une matrice dans laquelle chaque vecteur horizontal représente les rencontres effectuées par une personne dans le passé avec ses contacts. Nous considérons la matrice suivante. Notez qu'elle est symétrique et que nous plaçons sur la diagonale le nombre total de contacts effectués par une personne.

	tat37HG	xuyh57	jojoXYZ	Ubbdyu
tat37HG	5	1	1	3
xuyh57	1	3	2	0
jojoXYZ	1	2	3	0
Ubbdyu	3	0	0	3

On veut étudier les foyers infectieux en appliquant des méthodes vues en cours NFE204, et une ébauche de classification *kMeans*.

— Donnez les normes des vecteurs.

— On soupçonne que `jojoXYZ` et `Ubbdyu` sont à l'origine de deux foyers C_1 et C_2. Donnez une mesure de distance basée sur la similarité cosinus entre ces deux pseudos, entre `jojoXYZ` et `xuyh57` et finalement entre `Ubbdyu` et `tat37HG`. En déduire la composition des deux foyers.

— Outre la fonction cosinus, on dispose d'une fonction *centroid(G)* qui calcule le centroïde d'un ensemble de vecteurs *G*.
Quelle chaîne de traitement scalable permet de produire la classification de tous les pseudos dans l'un des deux foyers et d'obtenir les nouveaux centroïdes ?

— La chaîne de traitement précédente doit être répétée jusqu'à convergence pour obtenir un *kMeans* complet. Si on travaille avec Spark, quelles informations devraient être conservées dans un RDD persistant selon vous ?

Correction

— Normes : $\sqrt{36} = 6$; $\sqrt{14} = 3,74$; $\sqrt{14} = 3,74$; $\sqrt{18} = 4,25$.

— Calculons la similarité cosinus

— $cos(\text{tat37HG}, \text{xuyh57}) = \frac{5+3+2}{6} \times 3,74 \approx 0,44$

— $cos(\text{tat37HG}, \text{jojoXYZ}) = \frac{5+2+3}{6} \times 3,74 \approx 0,44$

— $cos(\text{tat37HG}, \text{Ubbdyu}) = \frac{15+9}{6} \times 4,25 \approx 0,95$

— $cos(\text{xuyh57}, \text{jojoXYZ}) = \frac{1+6+6}{14} \approx 0,92$

— $cos(\text{xuyh57}, \text{Ubbdyu}) = \frac{3}{3,74} \times 4,25 \approx 0,18$

— $cos(\text{jojoXYZ}, \text{Ubbdyu}) = \frac{3}{3,74} \times 4,25 \approx 0,18$

On distingue donc bien deux groupes : `jojoXYZ` est très proche de `xuyh57`, et `tat37HG` est très proche de `Ubbdyu`.

— La fonction de map calcule *cosinus()* pour les deux centroïdes $C1.centre$ et $C2.centre$ et affecte le vecteur au foyer le plus proche.

```
function mapKmeans (vecteur)
{
  if ( cosinus (C1.centre, vecteur) > cosinus (C2.centre, vecteur)) then
         emit ("C1", vecteur)
  else
        emit ("C2", vecteur)
  end;
```

La fonction de reduce calcule le centroide.

```
function reduceKmeans (key, L: liste(vecteurs))
{
  return (key, centroid(L));
```

— La matrice doit être dans un RDD persistant. À chaque étape il faut également conserver les centroïdes en mémoire RAM.

15.7.3 Troisième partie : analyse à grande échelle (5 pts)

On veut maintenant contrôler la propagation du virus grâce aux informations recueillies. On prend la matrice des rencontres suivantes (la même que précédemment mais cette fois la diagonale est à 0 puisqu'on s'intéresse à la transmission d'une personne à l'autre).

	tat37HG	xuyh57	jojoXYZ	Ubbdyu
tat37HG	0	1	1	3
xuyh57	1	0	2	0
jojoXYZ	1	2	0	0
Ubbdyu	3	0	0	0

Faisons quelques hypothèses simplificatrices :

— le nombre de rencontres de chaque pseudo avec un contact représente la probabilité de rencontrer chacun des contacts. Par exemple, le premier document ci-dessus nous dit que `jojoXYZ` a deux fois

plus de chances de rencontrer `xuyh57` que de rencontrer `tat37HG`.
- La liste des contacts est fixe.
- Chaque personne fait exactement une rencontre par jour.

En résumé, chaque personne rencontre exactement un de ses 3 contacts, chaque jour, avec une probabilité proportionnelle à la fréquence antérieure des rencontres avec ces mêmes contacts.

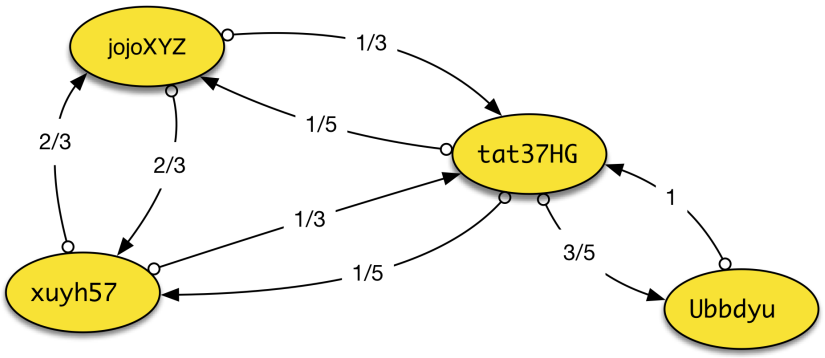
- Donnez la matrice des probabilités de contact obtenue à partir de la matrice des rencontres
- Dessinez le graphe en étiquetant les arêtes par leur probabilité.
- On reçoit l'information que `jojoXYZ` est infecté : quelle est la probabilité que `Ubbdyu` le soit également deux jours plus tard ?
- Si l'une des personnes de notre groupe est infectée, expliquez comment on calcule les probabilités d'infection de chaque personne *n* jours plus tard. Donnez l'exemple de la première étape.

Correction

- Voici la matrice de transition. La somme des termes sur chaque ligne doit être égale à 1.

	tat37HG	xuyh57	jojoXYZ	Ubbdyu
tat37HG	0	1/5	1/5	3/5
xuyh57	1/3	0	2/3	0
jojoXYZ	1/3	2/3	0	0
Ubbdyu	1	0	0	0

- Voir figure `reseausocial`



- On part avec le vecteur $(0, 0, 1, 0)$ représentant le fait que `jojoXYZ` est infecté. On multiplie par chaque vecteur de transition et on obtient la probabilité $\frac{1}{3}$ d'infecter `tat37HG`, $\frac{2}{3}$ d'infecter `xuyh57` Seconde étape : le vecteur est $(1/3, 2/3, 0, 0)$. Si on part de `tat37HG`, ça nous laisse une probabilité $\frac{3}{15}$ d'infecter `Ubbdyu`.
- C'est exactement PageRank : on multiplie le vecteur et la matrice jusqu'à convergence (garantie).

15.8 Examen du 5 septembre 2020

En raison du COVID19, cet examen s'est tenu à distance. Les documents étaient donc implicitement autorisés.

15.8.1 Première partie : modélisation NoSQL (8 pts)

Nous créons une base généalogique dont voici l'unique table relationnelle (avec l'essentiel).

```
create table Personne
       (id integer not null,
        nom varchar not null,
        idPère integer,
        idMère integer,
        primary key (id),
        foreign key (idPère) references Personne,
        foreign key (idMère) references Personne
    )
```

Voici également un tout petit échantillon de la base.

id	nom	idPère	idMère
102	Charles IX	87	41
65	Laurent de Médicis		
87	Henri II	34	
34	François Ier		
97	Marguerite de Valois	87	41
41	Catherine de Médicis		65
43	François II	87	41

— Proposez une représentation sous forme de document structuré (JSON ou XML) de l'entité Charles IX et de ses *ascendants*.
— Que proposeriez-vous pour ajouter dans cette représentation la fratrie de Charles IX ? Discutez brièvement des avantages et inconvénients de votre solution.
— Proposez une représentation sous forme de document structuré (JSON ou XML) de l'entité François 1er et de ses *descendants*.
— Conclusion : Vers quel type de base NoSQL vous tourneriez-vous et pour quelle raison ?

15.8.2 Seconde partie : MapReduce (10 pts)

Nous disposons d'une matrice *M* de dimension $N \times N$ représentant les liens entres les N pages du Web, chaque lien étant qualifié par un facteur d'importance (ou « poids »). La matrice est représentée par une collection math :*C* dans laquelle chaque document est de la forme {« id » : &23, « lig » : *i*, « col » : *j*, « poids » : m_{ij}, et représente un lien entre la page P_i et la page P_j de poids m_{ij}

Exemple : voici une matrice M avec $N = 4$. La première cellule de la seconde ligne est donc représentée

par un document {« id » : &t5x, « lig » : 2, « col » : 1, « poids » : 7

$$M = \begin{bmatrix} 1 & 2 & 3 & 4 \\ 7 & 6 & 5 & 4 \\ 6 & 7 & 8 & 9 \\ 3 & 3 & 3 & 3 \end{bmatrix}$$

Questions

— On estime qu'il y a environ $N = 10^{10}$ pages sur le Web, avec 15 liens par page en moyenne. Quelle est la taille de la collection C, en TO, en supposant que chaque document a une taille de 16 octets ?

— Nos serveurs ont 2 disques de 1 TO chacun et chaque document est répliqué 2 fois (donc trois versions en tout). Combien affectez-vous de serveurs au système de stokage ?

— Chaque ligne L_i de M peut être vue comme un vecteur décrivant la page P_i. Spécifiez le traitement MapReduce qui calcule la norme de ces vecteurs à partir des documents de la collection C.

— Nous voulons calculer le produit de la matrice avec un vecteur $V(v_1, v_2, \cdots v_N)$ de dimension N. Le résultat est un autre vecteur W tel que :

$$w_i = \Sigma_{j=1}^{N} m_{ij} \times v_j$$

On suppose pour le moment que V tient en mémoire RAM et est accessible comme variable statique par toutes les fonctions de Map ou de Reduce. Spécifiez le traitement MapReduce qui implante ce calcul.

— Maintenant, on suppose que V ne tient plus en mémoire RAM. Proposez une méthode de partitionnement de la collection C et de V qui permette d'effectuer le calcul distribué de $M \times V$ avec MapReduce sans jamais avoir à lire le vecteur sur le disque.
Donnez le critère de partitionnement et la technique (par intervalle ou par hachage).

— Supposons qu'on puisse stocker *au plus* deux (2) coordonnées d'un vecteur dans la mémoire d'un serveur. Inspirez-vous de la figure http://b3d.bdpedia.fr/calculdistr.html#mr-execution-ex pour montrer le déroulement du traitement distribué précédent en choisissant le nombre minimal de serveurs permettant de conserver le vecteur en mémoire RAM.
Pour illustrer le calcul, prenez la matrice 4×4 donnée en exemple, et le vecteur $V = [4, 3, 2, 1]$.

— Expliquez pour finir comment calculer la similarité cosinus entre V et les L_i.

15.9 Examen du 28 juin 2023

15.9.1 Première partie : modélisation NoSQL (7 pts)

Les articles de recherche universitaires sont maintenant gérés dans des dépôts d'archive, comme HAL en France. Posons-nous quelques questions sur le fonctionnement d'une telle archive. Un modèle très sommaire du dépôt est donné dans la figure ci-dessous. Les chercheurs publient des articles (qui peuvent avoir plusieurs co-auteurs) et sont emph{affiliés à des organismes de recherche. Ces organismes sont hiérarchiquement structurés : un chercheur peut être attaché à une équipe (organisme de base), qui fait partie d'un laboratoire (organisme parent) qui lui-même fait partie d'un établissement (par exemple le Cnam), etc.

Voici également un tout petit échantillon de la base relationnelle.

ref	titre	année
hal-875	Music modeling	2018
hal-293	Recommendations on Twitter	2019

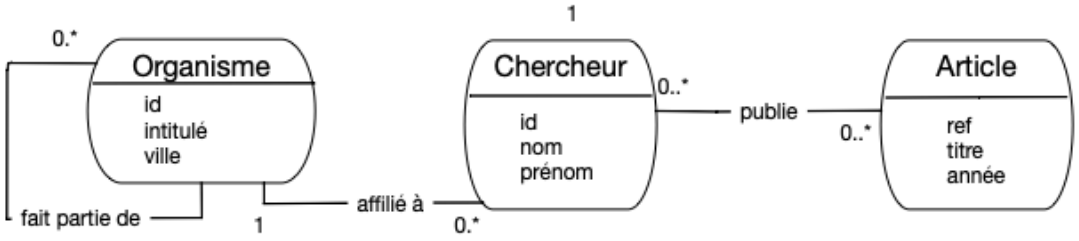


Fig. 15.3 – Modèle (sommaire) du dépôt

La table `Article`

id	nom	affiliation
nt	Travers	DVRC
pr	Rigaux	Vertigo
rfs	Fournier-S'niehotta	Vertigo
cdm	du Mouza	ISID

La table `Chercheur`

refArticle	idAuteur
hal-875	nt
hal-293	nt
hal-875	pr
hal-875	rfs
hal-293	cdm

La table `Auteur`

— Proposez un format de document JSON représentant toutes les informations relatives à l'article hal-875 présentes dans les trois tables ci-dessus (représentation A).

Correction

On part de la ligne relationnelle `hal-875` puis on « navigue » dans la base pour trouver les informations liées à l'article, soit les auteurs.

```
{
  "ref": 'hal-875',
  "titre": "Music modeling",
  "année": 2018,
  "auteurs": [
    {"id": "nt",
      "nom": "Travers",
      "affiliation": "DVRC"
    ,
```

(suite sur la page suivante)

(suite de la page précédente)

```
    {"id": "pr",
      "nom": "Rigaux",
      "affiliation": "Vertigo"
    ,
    {"id": "rfs",
      "nom": " Fournier-S'niehotta",
      "affiliation": "Vertigo"
    ,
  ]
```

— Proposez un document JSON représentant toutes les informations relatives à l'organisme Vertigo présentes dans 3 tables ci-dessus (représentation B).

Correction

On sait peu de choses sur Vertigo, à part son intitulé. On développe donc la liste des chercheurs, puis, pour chaque chercheur, de ses articles, ce qui introduit une redondance importante.

```
{
  "organisme": 'Vertigo',
  "chercheurs": [
    {"id": "pr",
      "nom": "Rigaux",
      "articles": [
                {
                  "ref": 'hal-875',
                  "titre": "Music modeling",
                  "année": 2018,
                ]
    ,
    {"id": "rfs",
      "nom": " Fournier-S'niehotta",
      "articles": [
                {
                  "ref": 'hal-875',
                  "titre": "Music modeling",
                  "année": 2018,
                ]

  ]
```

— Expliquez le calcul MapReduce permettant de produire les documents B à partir des documents A. **Important** : pour spécifier les calculs MapReduce, donner les fonctions de Map et de Reduce, en javascript, en pseudo-code, au pire en langage naturel en étant le plus précis possible.

Correction

Dans un document de type A, reçu par la fonction de Map, on doit aller chercher dans les auteurs de l'article l'affiliation, et émettre une paire dont cette affiliation est la clé.

```
function fonctionMap ($doc) # doc est un document de type Article
{
        # On parcourt les auteurs
        for    [$c in $doc.auteurs] do
                emit ($c.affiliation, $doc)
        done
```

Remarques : on va faire autant de `emit` qu'il y a d'auteurs dans l'article. Et pour chaque `emit` on va envoyer comme valeur de la paire intermédiaire l'ensemble de l'article, à charge pour la fonction de reduce d'en extraire les informations nécessaires aux documents de type B. C'est un choix paresseux : ces articles vont transiter sur le réseau pendant le *shuffle*, avec un coût important.
La fonction de Reduce reçoit donc un nom d'organisme, et la liste des articles dont l'un des auteurs est affilié à l'organisme. Il reste à reconstituer le document B. Le schéma est donné ci-dessous (on n'en demande pas plus)

```
function fonctionReduce ($organisme, $articles)
{
        # On parcourt les articles et leurs auteurs
        # pour ne garder que ceux affiliés
        document = {"organisme": $organisme
        for    [$a in $articles] do
          for [$c in $a.auteurs] do
            if $c.affiliation == $organisme then
                 # On a trouvé un chercheur de l'organisme
                 document.ajout ($c)
          done
        done
        return $document
```

— Maintenant on prend en compte la table décrivant la hiérarchie des organismes, dont voici un échantillon.

id	intitulé	ville	idParent
Vertigo	Données et apprentissage	Paris	cedric
DVRC	Systèmes intelligents	Nanterre	De Vinci
ISID	Systèmes d'information	Paris	cedric
cedric	Centre informatique	Paris	Cnam

Dessinez cette hiérarchie, et proposez une modélisation JSON pour ajouter l'information sur les organismes dans la représentation B. Illustrez cette modélisation sur le document « Vertigo » de la seconde question.

Correction
La modélisation JSON se prête bien aux structures hiérarchiques. Ici, on obtiendrait des documents de la forme suivante :

```
{
  "organisme": 'Cédric',
  "composantes": [
        {
                "organisme": 'Vertigo',
                "composantes": []
        ,
        {
                "organisme": 'ISID',
                "composantes": []

  ]
```

— Les références bibliographiques sont gérées depuis des dizaines d'années dans un format texte dit Bibtex. Voici par exemple un document Bibtex représentant un article.

```
@article{frt-02435620,
  TITLE = {{Modeling Music as Synchronized Time Series,
        AUTHOR = {Fournier-S'niehotta, Raphael and Rigaux, Philippe and␣
↪Travers, Nicolas,
        URL = {https://hal-cnam.archives-ouvertes.fr/hal-02435620,
        JOURNAL = {{Information Systems,
        PUBLISHER = {{Elsevier,
        VOLUME = {73,
        PAGES = {35-49,
        YEAR = {2018,
```

Comment qualifieriez-vous ce format par rapport à JSON ? Voyez-vous des inconvénients, des avantages, à cette représentation ?

Correction

On voit que Bibtex est un version un peu primitive de JSON ou XML, avec notamment une gestion peu robuste des structures imbriquées. Les auteurs par exemple seraient modélisés comme un tableau dans JSON, et sont ici dans une chaîne de caractères, les éléments étant conventionnellement séparés par des and. Mais on retrouve quelques idées importantes comme la flexibilité du schéma et la sérialisation textuelle qui facilite les échanges.

15.9.2 Deuxième partie : recherche d'information (7 pts)

Les articles ont des résumés, et on va construire un index sur ces résumés à des fins de classification et d'analyse. Voici un extrait des résumés de 5 articles $A_1, ..., A_5$.

— A_1 : Cet article compare les stratégies d'optimisation des échanges réseaux entre processeurs
— A_2 : Un orchestre moderne doit savoir maîtriser une harmonie atonale
— A_3 : Comparaison entre les propriétés des réseaux filaires et des réseaux optiques
— A_4 : Pourquoi un processeur quantique ne peut surmonter un processeur standard qu'en présence d'un réseau performant
— A_5 Mozart est-il mieux servi par un orchestre de chambre ou un orchestre philarmonique ?

On va se limiter au vocabulaire (`harmonie`, `processeur`, `réseau`, `orchestre`). (NB : on suppose une phase préalable de normalisation qui élimine les pluriels, majuscules, etc.)

— Donnez une matrice d'incidence contenant les tf, et un tableau donnant les idf (sans le $\log$), pour les termes précédents. Placez les termes en ligne, les résumés en colonne.

Correction

	A1	A2	A3	A4	A5
harmonie 5/1	0	1	0	0	0
processeur 5/2	1	0	0	2	0
réseau 5/3	1	0	2	1	0
orchestre 5/2	0	1	0	0	2

Donnez les normes des vecteurs représentant ces résumés.

Correction
— $||A_1|| = \sqrt{2}$;
— $||A_2|| = \sqrt{2}$;
— $||A_3|| = \sqrt{2}$;
— $||A_4|| = \sqrt{5}$
— $||A_5|| = \sqrt{4}$

— Dans un espace vectoriel à deux dimensions avec « processeur » en abcisse et « réseau » en ordonnée, placez les vecteurs représentant nos documents.
— Donner les résultats classés par similarité cosinus basée sur les tf (on ignore l'idf) pour les requêtes suivantes. Detaillez les calculs.
  — processeur et réseau
  — harmonie et orchestre

Correction
Calculs cosinus standards.

— Sans calcul, indiquez quel document est classé en tête pour la requête avec le seul mot « réseau », et expliquez pourquoi.

Correction
C'est bien sûr le A3. Comme la requête, il parle de réseau ET uniquement de réseaux. Les deux

vecteurs sont donc co linéaires.

15.9.3 Troisième partie : systèmes distribués (5 pts)

Voici quelques aspects d'Elastic Search qui devraient vous être familiers, et d'autres qui n'ont pas été abordés en cours

La documentation nous dit : *When you index a document, it is stored on a single primary shard determined by a simple formula* : $shard = hash(id) \% \text{number\_of\_primary\_shards}$

Voici la configuration initiale de votre *cluster* ElasticSearch pour les questions qui suivent (rappel : réplica = n signifie $n + 1$ copies d'un document).

— Nombre de fragments (*{shards*) : 3
— Nombre de réplicas : 2

Et je crée un index. Chacune des questions ci-dessous vaut 1 point. Une brève explication sera appréciée.

— Est-ce que mon *cluster* peut être constitué d'un seul nœud (un seul serveur) ?

Correction
Oui, mais l'index sera en situation périlleuse puisqu'il manquera deux serveurs pour héberger les réplicas. Autre réponse : non car il faut stocker les réplicas sur des serveurs distincts.

— Est-ce que je peux augmenter le nombre de *shards* sans récréer l'index ?

Correction
Non, c'est du hachage statique, toute affectation à un fragment est définitive, sauf à reconstituer l'index.

— Est-ce que je peux augmenter le nombre de réplicas ?

Correction
Oui, aucun problème, à condition d'avoir assez de serveurs sinon ça n'a pas vraiment de sens.

— Est-ce que je peux ajouter des nœuds à mon *cluster* ?

Correction
Oui, ça peut améliorer la distribution des fragments.

— Si oui, à partir de quand est-ce que ça ne sert plus à rien d'ajouter des nœuds, en prenant en compte la configuration initiale ?

Correction
Quand on a un fragment par serveur, ça ne sert à rien d'en ajouter. Donc $3 * (2 + 1) = 9$ serveurs.

Un dernier point à gagner. Un de vos collègues vous affirme qu'il vaut mieux créer un index avec beaucoup de *shards*, pour anticiper la croissance future. Voici un argument contre ce choix, issu de la documentation. *Term statistics, used to calculate relevance, are per shard. Having a small amount of data in many shards leads to poor relevance.*

Que répondez-vous à votre collègue ?

15.9.4 Quatrième partie : questions de cours (2 pts)

Nous avons vu que dans un système comme cassandra, on pouvait régler des paramètres W (acquittements en lecture).

Ce réglage correspond à la recherche d'un compromis, mais entre quelles propriétés ? Reprenez le théorème CAP et donnez des réponses argumentées aux questions suivantes :

— Si W, quelle(s) propriété(s) est/sont sacrifiée(s) avec une faible valeur de R ? Et avec la valeur de R maximale ?

Correction
Si R est faible, on sacrifie la cohérence car on augmente la probabilité de lire une valeur obsolète. En augmentant R on améliore la cohérence, mais on diminue la disponibilité et la tolérance.

— Si W est fixé à la valeur maximale, le choix de R influe-t-il sur les propriétés CAP ?

Correction
Non, si on a réussi à faire l'écriture aevc tous les acquittements, toute lecture ramène une valeur cohérente, la disponibilité et la tolérance sont aussi maximales.

15.10 Examen du 22 juin 2024

15.10.1 Première partie : documents structurés (6 pts)

Nous approchons des jeux olympiques de Paris 2024, événement majeur, mais source probable de désordres importants. Les organisateurs décident de mettre en place un système de co-voiturage à destination des sites, afin d'éviter les encombrements.

On pense au départ qu'une base relationnelle sera suffisante, et on adopte le modèle suivant (très simplifié bien sûr). Les clés primaires sont en **gras**, les clés étrangères en *italiques*.

— Personne (**id**, nom, domicile)
— Site (**id**, nom)
— Trajet (**id**, *idConducteur, *idSite*, date)
— Passager (**idTrajet, idPassager**)

Voici également un tout petit échantillon de la base.

Tableau 15.1 – La table Personne

id	nom	domicile
1	Albert	Melun
2	Aline	Versailles
3	Ktie	Créteil

Tableau 15.2 – La table des sites

id	nom
100	Stade de France
101	Parc de Versailles
102	Grand Palais

Tableau 15.3 – La table des trajets

id	idConducteur	idSite	date
A	1	100	30/07
B	3	100	4/08
C	1	101	31/07

Tableau 15.4 – La table des passagers

id	idPassager
A	2
A	3
B	1
C	2

On s'aperçoit rapidement que cette représentation impose beaucoup de jointures et ne passe pas à l'échelle. On décide de remplacer la base relationnelle par une base Cassandra.

— Voici une première table Cassandra pour représenter les trajets.

```
create table Trajet1 (idTrajet text,
    date date,
    site frozen<Site>,
    conducteur frozen<Personne>,
    passagers set< frozen<Personne>>,
       primary key idTrajet );
```

Donnez l'exemple d'un document JSON correspondant à la structure, de la table `Trajet1`, avec les informations du trajet A de la base relationnelle ci-dessus.

— Voici la proposition d'une autre modélisation, avec une table `Trajet2` et une table `Passager` dont la clé est *composite*

```
create table Trajet2 (idTrajet text,
    date date,
    site frozen<Site>,
    conducteur frozen<Personne>
 primary key (idTrajet);

create table Passager (idTrajet text,
     idPassager text,
    passager frozen<Personne>,
 primary key (idTrajet, idPassager );
```

Rappelons ce qu'est une clé composite en Cassandra, d'après la documentation : *A compound primary key consists of the partition key and the clustering key. The partition key determines which node stores the data. Rows for a partition key are stored in order based on the clustering key*. Quelles affirmations sont vraies ?
  - Les lignes de `Trajet2` et de `Passager` sont sur le même serveur
  - Les passagers d'un même trajet sont tous sur un seul serveur
  - Les passagers stockés sur un serveur sont triés sur leur identifiant
  - Les passagers d'un même trajet sont stockés contiguement
- Est-il possible de convertir la table `Trajet1` vers la table `Passager` avec un traitement Map Reduce ? Expliquez comment, en indiquant ce que font les fonctions de Map et de Reduce.

15.10.2 Deuxième partie : recherche d'information (6 pts)

L'application enregistre des commentaires déposés par les clients sur les conducteurs, et réciproquement. Voici un échantillon.

- c_1 : Conduite dangereuse, et conducteur bavard. À éviter.
- c_2 : Pour la conduite ça ne va pas : le conducteur (sympa par ailleurs) réussit à être à la fois lent et dangereux
- c_3 : Trop dangereux ! Je veux bien payer pour un co-voiturage, mais pas avec un conducteur dangereux.
- c_4 : Sympa, mais le conducteur est très bavard et lent.
- c_5 : Il est vraiment bavard de chez bavard. Heureusement, sympa et rien de dangereux dans sa conduite.

On va se limiter au vocabulaire (`dangereux`, `bavard`, `lent`, `sympa`). (NB : on suppose une phase préalable de normalisation qui élimine les pluriels, majuscules, trouve que « sympa » et « sympathique », « dangereux » et « dangereuse », sont les mêmes mots, etc.)

- Donnez une matrice d'incidence contenant les tf, et un tableau donnant les idf (sans le *log*), pour les termes précédents. Placez les termes en ligne, les commentaires en colonne.

- Donnez les normes des vecteurs représentant ces commentaires.

- Donner les résultats classés par similarité cosinus basée sur les tf (on ignore l'idf) pour les requêtes suivantes. Expliquez brièvement la raison du classement pour le premier document.
  - bavard ;
  - lent et sympa ;
  - dangereux et bavard.

- Pour la dernière requête, `dangereux` et `bavard`, on constate que deux documents sont à égalité. Est-ce que cela change si on prend en compte l'idf, comment et pourquoi ?

15.10.3 Troisième partie : MapReduce (3 pts)

Nous recevons un flux de commentaires dans un fichier `trajets.csv`. Il contient l'identifiant du conducteur, du client, et le commentaire. Voici le début.

```
1, 101, "Voiture bruyante"
1, 103, "Une vraie épave"
2, 101, "La voiture est propre, plus que le conducteur..."
..
```

Utilisant Pig latin, on charge cette collection avec la commande suivante :

```
trajets = LOAD 'trajets.csv' as (idConducteur, idClient, commentaire);
```

Et on exécute le script Pig suivant :

```
coll1 = filter trajets by contains(commentaire, "voiture")
coll2 = group coll1 by idConducteur;
coll3  = foreach coll2 generate group, COUNT(coll1.commentaire);
```

Répondez aux questions suivantes

- Expliquez le résultat produit par ce script.
- Comment ce script peut-il être traduit en MapReduce ? Donnez la fonction de Map et la fonction de Reduce, sous la forme de votre choix.
- Quelle serait la requête SQL correspondant à ce script Pig ?

15.10.4 Quatrième partie : systèmes distribués (3 pts)

- Sachant que Cassandra organise la distribution des données selon la technique du hachage cohérent, expliquez l'algorithme qui place un document de la table `Passager` sur un serveur.
- Parmi les requêtes ci-dessous, lesquelles peuvent être routées vers un seul serveur ?
  - Les trajets vers Versailles
  - Les passagers du trajet `t12` (c'est son identifiant)
  - Les sites visités par le passager `p988` (c'est son identifiant)
- À l'instant t, les valeurs de hachage de deux trajets T_1 et T_2 sont identiques. Peuvent-ils être placés sur deux serveurs différents à un instant $t + \Delta$, sachant qu'entretemps plusieurs nœuds ont été ajoutés au système Cassandra ?

15.10.5 Cinquième partie : question de cours (2 pts)

- Comment expliquer l'expression « index inversé » dans la terminologie des moteurs de recherche ?
- Rappeler le principe de *data locality*

CHAPITRE 16

Indices and tables

— genindex
— modindex
— search