CASSANDRA

Christophe Parageaud
Julien Dubois
Jérôme Mainaud
Raphaël Brugier

LIVRE BLANC

JUIN 2016

IPPON
Digital . Technologies . Hosting

TABLE DES MATIÈRES

1 Les auteurs 5
2 A propos d'Ippon technologies 6
2.1. Nos solutions 7
2.2. Nous contacter 8
3 Licence 9
4 Présentation du livre blanc 10
4.1. Une offre end-to-end 10
4.2. Une équipe pluridisciplinaire expérimentée sur Cassandra 10
4.3. Un investissement fort 10
5 Présentation de Cassandra 11
5.1. Introduction 11
5.2. Utilisateurs notables 12
5.3. Concepts 12
5.4. Les différentes distributions 13
5.5. Mécanisme de stockage 15
5.6. Tolérance à la panne 15
5.7. Réplication des données 15
5.8. Partitionnement des données 17
5.9. Gestion des transactions 18
5.10. Stratégies de compaction 18
5.11. Les caches dans Cassandra 20
5.12. Sécurité 20
5.13. Intégration 21
5.14. Cohérence des données (Tunable Consistency) 22
5.15. Modélisation 23
5.16. Cassandra Query Language CQL3 26
5.17. Gestion des index 28
5.18. Drivers Cassandra 28
5.19. Monitoring 28
5.20. Utilitaires 29
5.21. Tests 29
5.22. Cas d'utilisation (pattern/anti-pattern) 30
5.23. Liens 31
6 Cas pratique : Modélisation dans Cassandra 32
6.1. Factures et commandes dans Cassandra 32
6.2. Recherche multicritères 39
6.3. Modélisation d'un panier 46
6.4. Stockage de fichiers dans Cassandra 53
6.5. Conclusion 59

7 Retours d'expérience Cassandra 60
7.1. Contexte client 60
7.2. Contraintes 60
7.3. Contribution Ippon 60
7.4. Résultats obtenus 61
7.5. Contexte client 61
7.6. Contribution d'Ippon 61
7.7. Résultats obtenus 62
8 Ippon dans la communauté Cassandra 63
8.1. Lancer l'application packagée et un cluster Cassandra 64
8.2. Modifier le schéma avec JHipster et utiliser l'outil de migration 64
9 Conclusion 67

1 LES AUTEURS

Christophe **Parageaud**

Julien **Dubois**

Jérôme **Mainaud**

Raphaël **Brugier**

Découvrez tous **leurs articles** sur ***blog.ippon.fr***

L'écosystème HADOOP
Christophe Parageaud

TamTam – Java 8 functional programming : don't neglect optimisations!
Christophe Parageaud

Modélisation Cassandra : Stocker des fichiers
Jérôme Mainaud

Elasticsearch : de l'importance du mapping
Jérôme Mainaud

Simplifier l'utilisation de Cassandra pour le projet open source JHipster
Raphaël Brugier

PaaS – Développer et déployer dans le Cloud
Julien Dubois

2 A PROPOS D'IPPON TECHNOLOGIES

Ippon Technologies est une société innovante créée en 2003 par un sportif de Haut Niveau et un polytechnicien avec pour ambition de devenir le cabinet de conseil en technologies leader sur les solutions Digitales, Cloud et BigData.

Ippon accompagne les entreprises dans le développement et la transformation de leur système d'information avec des applications performantes et des solutions robustes. Ippon propose une offre de services à 360° pour répondre à l'ensemble des besoins en innovation technologique.

LES SLIDES C'EST BIEN, LE CODE
EN PROD C'EST MIEUX

En 2015, 17 **projets Cassandra livrés avec succès.**

IPPON VOTRE PARTENAIRE
END TO END

CONSEIL
FORMATIONS

UX
DESIGN

RÉALISATION
AGILITE/DEVOPS

HÉBERGEMENT

2.1. Nos solutions

Nous accompagnons nos clients sur la mise en oeuvre de projets Data ou NoSQL avec une offre structurée pour faciliter les 3 étapes d'un projet Data:

UNE IDÉE, UN POC,
UN PRODUIT...

idée

lean startup

conseil POC

product backlog

projet par itérations

équipe agile

projet run

conseil DevOps

infogérance cloud

Vos besoins, nos solutions

2.2. Nous contacter

PARIS
BORDEAUX
NANTES

RICHMOND, VA
WASHINGTON, DC
NEW-YORK

MELBOURNE
MARRAKECH

www.ippon.fr/contact
talents@ippon.fr
blog.ippon.fr

+33 1 46 12 48 48
@ippontech

3 LICENCE

Ce document vous est fourni sous licence Creative Commons Attribution Share Alike.

Vous êtes libres de reproduire, distribuer et communiquer cette création au public selon les conditions suivantes:

- paternité : vous devez citer le nom des auteurs originaux mais pas d'une manière qui suggérerait qu'ils vous soutiennent ou approuvent votre utilisation de l'œuvre,
- à chaque réutilisation ou distribution de cette création, vous devez faire apparaître clairement au public les conditions contractuelles de sa mise à disposition sous licence identique Creative Commons Share Alike,
- chacune de ces conditions peut être levée si vous obtenez l'autorisation du titulaire des droits sur cette œuvre,
- rien dans ce contrat ne diminue ou ne restreint le droit moral de l'auteur ou des auteurs.

4 PRÉSENTATION DU LIVRE BLANC

Au programme :

- rappel des grands principes de son fonctionnement
- Illustration de la subtilité et de l'importance de la modélisation par un cas pratique
- Deux retours d'expérience dans sa mise en oeuvre
- Implication concrète d'Ippon dans l'open source via une de nos principales contributions.

Vous découvrirez Cassandra comme probablement jamais avant. Peut être saurez-vous alors à l'issue de cette lecture si cette base peut aussi répondre à l'un de vos besoins.

Ippon a très vite cru en cette solution et l'a utilisé dès 2011 pour des projets internes (réseau social d'entreprise Tatami).

Depuis, Ippon a développé une offre complète autour de cette solution :

4.1. Une offre end-to-end

- conseil et expertise technique,
- réalisation et développement,
- formation, support, suivi post-projet,
- conseil et expertise Devops.

4.2. Une équipe pluridisciplinaire expérimentée sur Cassandra

Architectes et développeurs

- experts techniques Dev et Ops,
- consultants confirmés Dev et Ops.

4.3. Un investissement fort

- premier projet Open Source avec Cassandra en 2011,
- une équipe de 15 consultants formée sur Cassandra,

- signature d'un partenariat technique et commercial en 2014 avec DataStax.

5 PRÉSENTATION DE CASSANDRA

5.1. Introduction

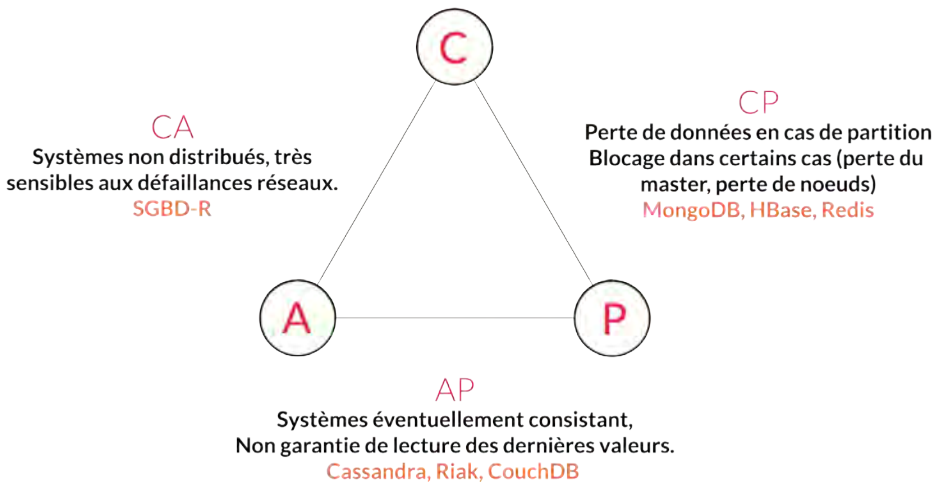
Cassandra est une base de données de la mouvance NoSQL. Initialement développée par Facebook en 2008, elle a été par la suite libérée et son développement est aujourd'hui assuré par la fondation Apache.

La société DataStax fournit du support ainsi qu'une version Entreprise avec quelques fonctionnalités supplémentaires. Cassandra est une base de données NoSQL étudiée pour des déploiements massivement distribués (éventuellement sur plusieurs datacenters).
Son architecture complètement décentralisée lui assure une résistance à la panne très importante.
Elle est particulièrement adaptée aux problématiques "time series" (suite de données chronologiques).
Un cluster est facilement extensible (ajout de nœuds) et les données sont réparties sur les nœuds grâce au hash de la clé primaire.

Cassandra est une base de données qui va favoriser la disponibilité plutôt que la cohérence

(même si on peut choisir au moment de la requête le niveau souhaité grâce au "consistency level").

POSITIONNEMENT CAP DES BASES NOSQL



5.2. Utilisateurs notables

- Apple (probablement le plus grand nombre de nœuds au monde),
- Ebay, Netflix,
- Instagram, Spotify, ...

5.3. Concepts

- RING : Cluster au sens Cassandra (ensemble des nœuds).
- KEYSPACE : c'est l'équivalent d'un schéma dans le monde des bases de données relationnelles. À noter qu'il est possible d'avoir plusieurs « Keyspaces » sur un même serveur.

5.3.1.Keyspace

C'est l'équivalent d'un schéma dans le monde des bases de données relationnelles. À noter qu'il est possible d'avoir plusieurs « Keyspaces » sur un même serveur.

Une valeur dans Cassandra est caractérisée par :

- RowKey (identifiant unique),

- nom colonne,

- valeur colonne,

- Timestamp automatiquement géré par Cassandra,

- date d'expiration de la donnée (option).

5.4. **Les différentes distributions**

5.4.1.Apache Cassandra

- licence : Apache License, Version 2.0,
- comprend la base de données et les utilitaires.

5.4.2.Datastax

- DSC : version communautaire (Apache Cassandra + exemples + Ops Center light),
- DSE : version entreprise.

Fonctionnalités	DSC (Communautaire)	DSE (Entreprise)
Sécurité basique	✓	✓
Sécurité avancée	-	✓
Hadoop	-	✓
In Memory	-	✓
DSE Search (Solr)	-	✓
Intégration Spark	-	✓
DSE Graph	-	✓
Migrations (outils)	-	✓
OpsCenter	Basic	Avancé
Support	-	✓

5.5. Mécanisme de stockage

Cycle de stockage d'une écriture :

- écriture dans le commit log,
- écriture en mémoire : Memtable,
- écriture sur disque : SSTable.

Lorsque la Memtable est pleine, Cassandra crée une nouvelle SSTable (avec un timestamp) contenant les évolutions. L'ordre d'arrivée n'est pas nécessairement cohérent mais le timestamp permet de les ordonner.
Les suppressions sont marquées (tombstone) sans que la donnée soit effacée. Un mécanisme nommé compaction est en charge de fusionner plusieurs fichiers SSTables dans un nouveau. C'est aussi lui qui va supprimer réellement les colonnes marquées au bout d'un certain temps (864000 secondes soit 10 jours par défaut). Il est possible de comprimer les données sur disque (LZ4, Snappy, Deflate).

5.6. Tolérance à la panne

Cassandra s'appuie sur deux mécanismes afin d'assurer la continuité de service ainsi que la non perte de données :

5.6.1.Sauvegarde

- il est possible de créer une image de l'état des données dans un fichier.

5.6.2.Journal des modifications

- le commit log sert à stocker les modifications, avant qu'elles ne soient stockées dans les SSTables lors de la compaction. Il est ensuite purgé.

5.7. Réplication des données

Cassandra supporte la réplication via un modèle "masterless" à des fins de résistance aux pannes et de répartition de la charge. Les nœuds se partagent les données qui sont répliquées un certain nombre de fois.

Tous les nœuds ont le même rôle, on peut lire/écrire sur n'importe quel nœud, le nœud contacté sert de coordinateur et est en charge de solliciter les nœuds qui hébergent (lecture), hébergeront (écriture) la donnée. Toutefois pour des raisons évidentes de performances, le driver contacte directement le nœud concerné (grâce au token).

5.7.1.Facteur de Réplication

Nombre de nœuds sur lesquels une donnée va être répliquée.
La réplication permet à Cassandra d'assurer la haute disponibilité en lecture/écriture et de continuer à offrir le service même en cas d'indisponibilité d'un nœud.

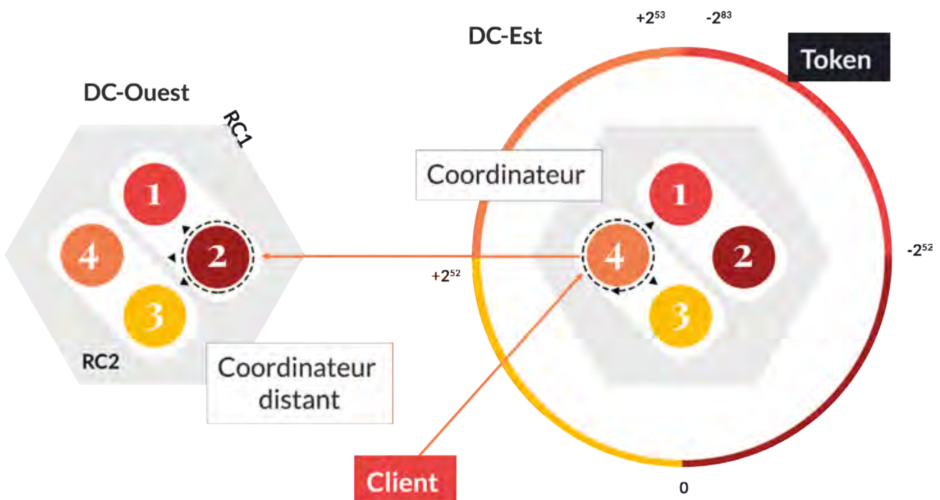
5.7.2.Réplication multi-datacenter

La gestion multi-datacenter dans Cassandra est de type actif-actif. Un cluster Cassandra peut être réparti sur plusieurs Data Center afin de permettre :

- l'isolation des traitements (pour créer un Data Center analytique par exemple),

d'implémenter un site de secours (PRA)

CASSANDRA ET LE MULTI DATA CENTER



5.8. Partitionnement des données

Les données sont réparties uniformément sur les réplicas (facteur de réplication). Chaque partition se voit affecter un lot de données à héberger en fonction du hash de la clé primaire. Pour chaque nœud il faut définir le premier token à héberger (limite basse : initial_token) ou bien utiliser les nœuds virtuels.

Partitioners disponibles :

- Murmur3 (défaut) : Distribution uniforme en fonction du hash «MurMur» de la clé primaire,
- Random : Distribution uniforme en fonction du hash «MD5» de la clé primaire,
- ByteOrdered : Classe les données en fonction de leur ordre naturel (mais au format sérialisé).

5.8.1.Nœuds virtuels

Pour éviter de se retrouver avec de trop grandes partitions la version 1.2 de Cassandra a introduit les nœuds virtuels (Virtual nodes).
Simplification des tâches administratives :

- calcul et affectation manuel des partitions non nécessaires,
- en cas d'ajout/suppression d'un nœud, la répartition des données est automatique.

Afin de mieux gérer la résilience du cluster à la panne d'un nœud il est possible de localiser une instance Cassandra : dans un datacenter comme vu plus haut mais aussi dans un rack.

5.9. Gestion des transactions

Cassandra n'offre pas de consistance au sens ACID car comme beaucoup de solutions NoSQL

il troque les transactions isolées et atomiques pour la haute disponibilité et la rapidité d'écriture.

Cassandra offre cependant l'atomicité et l'isolation au niveau d'une ligne : ajouter ou modifier des colonnes dans une ligne est considéré comme une seule opération d'écriture.
Cassandra propose tout de même du transactionnel avec les Lightweight Transactions qui permettent de conditionner une mise à jour.

En cas de conflit et de modifications simultanées, c'est le timestamp de l'update qui fait foi dans le commit log.
Un batch (caractérisé par le mot clé Batch en CQL) est un regroupement de commandes de modification uniquement : si une des instructions aboutit, toutes sont jouées jusqu'à réussite complète.

5.10. Stratégies de compaction

Plusieurs stratégies de compaction sont présentes dans Cassandra et correspondent à des cas d'usage différents.
La stratégie de compaction est définie par table.

Type de stratégie	Description	Pourquoi l'utiliser ?
SizeTiered Compaction Strategy (STCS)	C'est la stratégie par défaut, elle se déclenche lorsque plusieurs SSTables de même taille sont présentes (4 par défaut).	Cette stratégie est pertinente dans le cas d'insertion massive et dans les cas usuels. Elle est par contre inadaptée aux séries temporelles. Cette stratégie limite les passages de la compaction et n'est donc pas la plus optimale en terme d'occupation disque.
Leveled Compaction Strategy (LCS)	Avec cette stratégie les SSTables sont regroupées en fonction de leur taille dans un palier (level). Chaque palier est 10 fois plus important que le précédent. Le premier palier est fixé à 160Mo.	Cette stratégie est pertinente dans le cas de lectures et de mise à jours massives car il garantit que 90% des lectures se font sur une seule table. Cette stratégie favorise plus de déclenchement de compaction que STCS mais limite l'occupation disque.
DateTiered Compaction Strategy (DTCS)	Cette stratégie est spécialement dédiée aux séries temporelles et a été développée par Spotify. Elle permet de regrouper les SSTables par date d'écriture. Par défaut les écritures les plus récentes (une heure) sont regroupées dans une seule SSTable. Les anciennes SSTables deviennent de plus en plus importantes au fur et à mesure des compactions successives. A partir d'une certaine ancienneté (365 jours) les SSTables ne sont plus compactées.	Cette stratégie est très performante à condition de respecter certaines contraintes. - Eviter la mise à jour de données anciennes - Ne pas insérer les données sans respecter l'ordre d'écriture

5.11. Les caches dans Cassandra

Cassandra propose deux caches afin d'améliorer les performances en lecture :

- cache de ligne,
- cache de clé de partition.

Le cache de clé de partition permet de mettre en cache les clés de partitions des éléments d'une table et donc d'accélérer la recherche (mais pas la récupération) des résultats. Le cache de lignes, quant à lui, permet de mettre en cache les éléments d'une table. Ces deux caches ne sont pas activés par défaut et utilisent la mémoire off-heap. Une limite de taille permet de restreindre le nombre d'éléments à mettre en cache pour l'ensemble des tables du cluster. L'activation se fait au niveau table. Les caches sont sauvegardés régulièrement sur disque afin d'assurer la résilience en cas de redémarrage d'un nœud.

5.12. Sécurité

L'accès aux clusters peut être protégé par un mot de passe. Pour des fonctionnalités plus avancées (LDAP, chiffrement des données) il faut recourir aux versions commerciales.

5.12.1.Apache Cassandra

- authentification par utilisateur / mot de passe,
- gestion de droits classique (GRANT/REVOKE),
- chiffrement des données client à destination des nœuds Cassandra grâce à SSL.

5.12.2.DataStax Enterprise

- Kerberos authentication : permet aux nœuds de communiquer

sur un réseau non sécurisé pour prouver leur identité à l'aide de tickets,
- transparent data encryption : au flush des memtables de la mémoire vers le disque dans les SSTables, les données sont encodées pour être illisibles des utilisateurs non autorisés,
- data auditing : Possibilité de créer un journal d'audit d'activité détaillé.

5.13. **Intégration**

5.13.1.Spark

Un connecteur Spark-Cassandra permet l'intégration des deux technologies afin de former une solution complète de traitement et de stockage Big Data.

- Cassandra en entrée/sortie de traitements Spark,
- prise en compte des partitions Cassandra dans Spark (parallélisation).
- gestion du cluster unifiée avec Mesos (Version entreprise DSE),
- monitoring avec OpsCenter.

5.13.2.Solr

Dans la version entreprise de Cassandra il est possible d'intégrer Cassandra et DSE Search (Solr) de manière automatique.
L'activation de DSE Search sur une table Cassandra indexe automatiquement toutes les colonnes. Les index Solr sont en réalité des index secondaires de Cassandra et il est possible d'effectuer des recherches sur ces index avec CQL3.

5.13.3.Hadoop

Dans la version entreprise de Cassandra il est possible d'intégrer Cassandra et Hadoop afin d'exécuter des traitements MapReduce avec Cassandra en entrée/sortie.
Il est aussi possible d'utiliser Sqoop afin d'importer ou exporter des données en provenance de SGBD R dans Cassandra.

5.13.4.Connecteurs divers

Il existe de nombreux connecteurs pour Cassandra :

- Log4j,
- Talend ETL,
- Mule,
- ODBC/JDBC,
- Lucene/ElasticSearch, ...

5.14. **Cohérence des données (Tunable Consistency)**

5.14.1.Eventualy consistent : cohérence à terme

Configurable grâce au “consistency level” qui représente le nombre de replicas qui vont acquitter une lecture ou écriture au client.

Le choix du niveau impacte soit la cohérence, soit la disponibilité : No Free Lunch!

5.14.1.1.Fiabilité de la lecture

Une donnée lue sera forcément cohérente si W + R > RF

- W : nombre de nœuds écrits,
- R : nombre de nœuds lus,
- RF : replication factor (nombre total de nœuds).

5.14.2.Mécanisme de reprise sur crash

C’est la capacité de Cassandra à s’auto-réparer en cas de nœud hors ligne. C’est rendu possible grâce au timestamp des enregistrements et aux deux mécanismes suivants.

5.14.2.1.Hinted off repair

Le nœud conserve la réplication dans la table system.hints si un nœud est KO pour la rejouer quand il sera à nouveau disponible.

- temps de conservation par défaut de 3h,
- mécanisme effectué en tâche de fond sans aucune action utilisateur (client).

5.14.2.2.Read repair

Lors d'une lecture, Cassandra va interroger plus de nœuds que nécessaire. En cas d'incohérence la donnée est resynchronisée.
L'échange est peu coûteux car il utilise une fonction de type digest.
Par défaut 10% des lectures vont déclencher ce mécanisme (configurable).

> Anti entropy repair (nodetool repair)

C'est un mécanisme qui permet de "resynchroniser" un nœud avec les dernières modifications. Cette tâche doit être exécutée régulièrement et après certains événements (redémarrage d'un noeud après une panne, ajout d'un noeud, ...).

5.15. Modélisation

La modélisation est primordiale dans Cassandra du fait de ces caractéristiques :

Seuls les champs indexés sont requêtables.

Pour des raisons de performances, une seule partition doit être impliquée dans une requête de sélection.

5.15.1.Types de données supportés

5.15.1.1.Types simples

- text, varchar, ascii,
- int, bigint, smallint, tinyint, varint,
- double, float, decimal,
- boolean,
- date, time, timestamp,
- blob.

5.15.1.2.Types spécifiques

- inet,
- uuid, timeuuid,
- counter.

5.15.1.3.Types composites

- list, map, set
- tuple

5.15.1.4.Types utilisateur (UDT)

- Type composite défini par l'utilisateur qui peut être utilisé comme type d'une colonne ou d'une valeur dans un type ensembliste.

5.15.1.5.TTL (Time To Live)

Définit la péremption d'une valeur (colonne) lors de l'écriture.
Cela permet de supprimer des données automatiquement sans traitement de purge.
Une fois la durée de vie passée, la suppression de la donnée est effectuée en tâche de fond.

5.15.1.6.Compteurs (counter)

Les compteurs sont des types spéciaux qui permettent de gérer des nombres que l'on peut incrémenter ou décrémenter.
Les compteurs sont distribués dans la base et sont stockés dans une table spéciale qui ne contient que des compteurs.

Opérations supportées :

- increment,
- decrement,
- read,
- delete.

Performance : L'utilisation des compteurs est plus lente qu'une opération normale.

5.15.1.7.Champs statiques (static)

Les colonnes marquées comme statiques sont partagées par l'ensemble des lignes d'une table. L'avantage est de ne stocker qu'une seule fois des champs qui sont partagés.

5.15.2.Gestion des relations

Pas de notion de relations entre les données dans Cassandra : les notions de jointures ou de clés étrangères lui sont inconnues

On va donc porter toute l'attention sur la phase de modélisation (dénormalisation) afin de contourner cette problématique.
Dans certains cas (rares) on va regrouper dans une seule table l'ensemble de données qui se trouvent dans différentes tables dans le modèle normalisé.
Mais plus fréquemment on va dupliquer les informations communes et regrouper les données en fonction des besoins de requête (une table -> une requête).

5.16. Cassandra Query Language CQL3

CQL3 est sortie avec Cassandra 1.2 (2014), en remplacement de Thrift qui était le moyen recommandé d'accéder à un cluster Cassandra.
CQL3 est un langage de requête similaire à SQL.

Il existe des instructions pour :

- modifier des données,
- chercher des données (clé, intervalle de clés, index, ...),
- sauver des données,
- modifier la façon dont les données sont stockées.

Avec CQL3 on perd la souplesse d'une base schemaless :

- mais possibilité de créer un méta-modèle sans surcoût,
- CQL3 permet de masquer la complexité de la modélisation par colonne (wide rows, composite key).

5.16.1. Identifiant unique : clé primaire

Comme pour les SGBD la clé primaire est obligatoire.
Elle sert à Cassandra à déterminer le nœud de stockage : partition-key

Elle est de la forme suivante : PRIMARY KEY ((partition-key1, partition-key2, ...), key3, key4, ...)

- Partition-key :

détermine où sera stockée la donnée,

- Key3, Key4 :

«cluster key», sert à ordonner les données au sein de la partition.

5.16.2.Agrégation

Les possibilités d'agrégation avec CQL sont très récentes (2.2) et assez limitées :

- min, max,
- average,
- count.

Comment faire des opérations complexes sur les données ?

- soit en java/scala/... coté serveur après retour de la requête Cassandra,
- soit avec des User Defined Function (mix CQL, Javascript),
- soit pour des fonctions plus évoluées, intégration avec un moteur de recherche (export d'un index) comme ElasticSearch ou Solr.

5.16.3.Divers

Comme avec les SGBD, il est possible d'utiliser les « prepared statement » avec Cassandra.
Le but est le même, mettre en cache le plan d'exécution des requêtes fréquentes (et le mapping des données avec les tables).

On peut aussi écrire des trigger en Java (depuis la version 2.0) qui se présentent sous la forme d'un jar déployé dans un répertoire Cassandra. Cela permet de déclencher un traitement métier sur arrivée d'une donnée.

Avec la version 3.0 de Cassandra les vues matérialisées sont apparues. Elles permettent, tout comme leur équivalent dans le monde relationnel de proposer une vue unifiée provenant de tables multiples.

Elles supportent les requêtes :

- de sélection,
- de suppression (pas recommandé à cause des performances).

5.17. Gestion des index

Une donnée non indexée ne peut être requêtée dans Cassandra, il existe deux types d'index dans Cassandra.

Index primaire : créé automatiquement à partir de la clé primaire lors de la création de la table.

Index secondaire : création spécifique par l'utilisateur.

Quel index dans quelle situation ?

Pour une interrogation efficace il faut privilégier la clé primaire.

Pour les autres cas, il faut agir en fonction de la répartition des données :

- peu de résultats et temps de réponse trop long : ajouter des clés de type "cluster key",
- beaucoup de résultats qui matchent le critère d'interrogation : utiliser un index secondaire.

5.18. Drivers Cassandra

Le nombre de driver est important, certains sont supportés officiellement, d'autres par une communauté.
Langages supportés officiellement :

- Java, Python, Ruby, C# / .NET, Node.js, PHP, C++.

Langages supportés par la communauté :

- Scala, Clojure, Go, Erlang, Haskell, Rust, Perl.

5.19. Monitoring

Comme beaucoup de produits écrits en Java, Cassandra expose ses métriques en JMX.

OpsCenter est un outil Web de supervision fourni par la société

DataStax.

OpsCenter fournit des métriques et des graphiques sur le bon fonctionnement du cluster mais permet aussi de l'administrer (ajout/suppression d'un nœud, opérations de maintenance).

OpsCenter est disponible en deux versions :

- version open source allégée,

- version complète incluse dans

DataStax Enterprise.

5.20. Utilitaires

5.20.1.Manipulation et administration

Cassandra fournit une interface de ligne de commande, qui permet le requêtage et la manipulation des schémas, commande cqlsh.

Nodetool :

- réparation du cluster, compaction, ...
- statut du cluster,
- statistiques : histograms.

5.21. Tests

5.21.1.Tests d'intégration

il existe un certain nombre de services qui permettent de créer un cluster Cassandra dans la JVM au moment de l'exécution des tests (Embedded Cassandra) :

- Cassandra Unit : https://github.com/jsevellec/cassandra-unit,
- stubbed Cassandra : http://www.scassandra.org/

5.21.2. Tests de performances

- Stress tool fourni avec la distribution Cassandra,
- il existe un plugin JMeter.

5.22. Cas d’utilisation (pattern/anti-pattern)

Le cas idéal d’utilisation de Cassandra est un mélange entre insertions et lectures :

- les données ne sont pas modifiées ce qui évite les problèmes de cohérence,

- Cassandra est très rapide pour les écritures (Les écritures peuvent être beaucoup plus rapides que les lectures si les lectures sont liées au disque).

Cassandra est moins pertinent pour les modifications et les suppressions (à cause du mécanisme de tombstones évoqué plus haut)

Pour les mêmes raisons il n’est pas recommandé d’utiliser Cassandra comme source d’événements (plutôt que des solutions comme Kafka, Redis). A moins de vouloir conserver l’historique des événements traités. Cassandra comme toutes les solutions NoSQL n’est pas indiqué lorsque le besoin transactionnel est important.
Lire plusieurs partitions dans une même requête est un anti-pattern de modélisation. En terme d’architecture, il est déconseillé d’utiliser du stockage réseau à la place des disques locaux.

5.23. Liens

DataStax : ***http://www.datastax.com/***
Apache Cassandra : ***http://cassandra.apache.org/***
Blog Planet Cassandra : ***http://planetcassandra.org***
Cours gratuits online Cassandra : ***https://academy.datastax.com/courses***
The Kashlev Data Modeler : ***Automated modeling tool for Apache Cassandra : http://kdm.dataview.org/***

6 CAS PRATIQUE : MODÉLISATION DANS CASSANDRA

Le principal impact pour le développeur de l'utilisation des bases de données NoSQL est le changement profond du modèle de données. Ce n'est pas pour rien qu'elles sont qualifiées de « NoSQL », c'est-à-dire « dont le modèle de données n'est pas relationnel ». Ce qui nous intéresse ici, c'est que nous devons revoir la façon de modéliser les données lors de la conception d'une nouvelle application.

6.1. Factures et commandes dans Cassandra

Voyons le cas concret de la modélisation des factures ou des commandes dans Cassandra. Ce premier exemple n'est pas choisi au hasard : c'est un cas concret très fréquent dans les applications de gestion. Il correspond à un cas d'école pour expliquer les jointures du SQL, mais aussi les documents de MongoDB. Nous verrons que contrairement à ce qu'on pourrait penser a priori, il se modélise très bien dans Cassandra à condition de bien comprendre le modèle CQL3.

6.1.1.Modèle conceptuel

Avant de partir dans la modélisation spécifique à Cassandra, prenons un instant pour construire un modèle conceptuel et dessinons-le sous la forme d'un diagramme UML.
Nous allons prendre le cas d'une facture, la commande étant identique à ceci près qu'il y a écrit « commande » au lieu de « facture » sur l'entête.

Une facture est un document composé principalement de deux parties :

6.1.1.1.Une entête (invoice) qui contient toutes les informations générales

- un identifiant de la facture,
- la date de la facture,
- les informations sur le client,
- nom,
- adresse,
- la date de paiement,

- le montant total de la facture, …

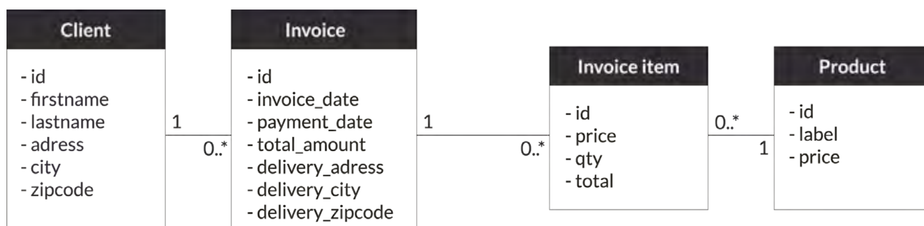
6.1.1.2.Une liste de lignes de facturation

… (invoice item) qui détaille l'ensemble des objets et des prestations facturées. Chaque ligne contient des informations comme :

- un identifiant de la ligne,
- la description de l'objet ou la prestation facturée,
- la quantité facturée,
- le prix unitaire,
- éventuellement le total de la ligne, …

Une ligne de facturation n'existe que dans le cadre d'une facture donnée. Il s'agit dont d'une relation de composition.
La facture est généralement liée à l'entité du client (client) facturé. Les lignes sont, elles, liées au produit vendu. Cependant, une facture n'étant pas modifiable, toutes les données modifiables de ces entités sont copiées pour garantir la validité des données.
Nous obtenons le diagramme suivant.

MODÉLISATION UML DES FACTURES



6.1.2.Le modèle de données de Cassandra

Avant de nous lancer dans la modélisation de notre facture dans Cassandra, il est important de bien comprendre le modèle de la base. On décrit souvent Cassandra comme une base de données de type famille de colonnes.
Pourtant, avec l'arrivée de CQL3, le modèle logique de la base a

complètement changé. Et l'ancien modèle de Map<SortedMap> inspiré de Google Big Table est sur le point de disparaître.
Actuellement, Cassandra est une base de données que je qualifierais de tabulaire partitionnée. Les données sont organisées en tables, dont les colonnes et leurs types sont définis par un schéma. Chaque ligne est identifiée par une clé primaire. Les lignes peuvent être regroupées dans une partition. Une partition est identifiée par une clé de partition qui est un préfixe de la clé primaire de la table. Cassandra garantit que toutes les lignes d'une partition sont stockées ensemble. De plus, elles sont classées par l'ordre lexicographique des colonnes de la clé primaire qui ne font pas partie de la clé de partition. Ces colonnes importantes sont appelées « clustering columns ». Grâce à cela, il est possible de demander toutes les lignes d'une partition à la fois ou de demander une tranche de lignes en ne précisant les valeurs que d'un préfixe de la clé primaire ou une inégalité sur la dernière clustering column.

CASSANDRA : PARTITIONNEMENT DES DONNÉES

Une partition peut posséder des colonnes qui lui sont propres. Qualifiées de statiques, elles ne sont stockées qu'une fois par partition et possèdent la même valeur pour toutes les lignes de cette dernière. Le mot-clé static utilisé dans la définition a été pris du mot-clé en Java qui permet de partager une valeur entre toutes les instances d'une même classe.

6.1.3.Modèle logique

Après ce petit détour théorique, voyons comment modéliser notre facture dans Cassandra.

En général, lorsqu'on modélise une base Cassandra, on recense toutes les requêtes en lecture qu'on veut pouvoir exprimer et on construit les tables qui répondent à ce besoin.
Les requêtes auxquelles nous voudrons répondre ici sont :

- lister les factures d'un client du plus récent au plus ancien. Seul un résumé de la facture devra être affiché,
- charger le détail d'une facture à partir de l'identifiant de facture trouvé grâce à la première requête.

À partir de ces requêtes, nous allons construire un diagramme de Chebotko. Ce dernier permet de réfléchir à la modélisation des tables Cassandra. Il doit son nom au premier auteur de la formation à la modélisation de Cassandra.

6.1.3.1.Lister les factures du client

Pour répondre à la première requête, nous allons lister un résumé de la facture de chaque client. Notre clé de recherche est l'identifiant du client, elle prendra naturellement la place de clé de partition. Les informations spécifiques au client, mais indépendantes de la facture comme son nom et son prénom, seront copiées dans des colonnes statiques pour éviter une jointure trop coûteuse lorsqu'on travaille avec Cassandra. Chaque résumé de facture sera enregistré dans une ligne.
Les dernières factures étant les plus intéressantes, nous classerons les factures dans l'ordre descendant.

A cette étape, nous obtenons le schéma suivant :

DIAGRAMME DE CHEBOTKO : FACTURES DES CLIENTS

Q1

Invoice by client		
client_id	timeuuid	K
firstname	text	S
lastname	text	S
invoice_id	timeuuid	C ↓
invoice_date		
total_amount		
delivery_city		
delivery_zipcode		

La liste des factures du client obtenue, elle sera présentée à l'utilisateur d'une manière ou d'une autre. Celui-ci sera à même de choisir un élément dans la liste et d'en demander le détail.

6.1.3.2.Charger le détail d'une facture

Pour stocker le détail d'une facture, nous allons nous appuyer sur la relation de composition entre les entités Invoice et InvoiceItem. Celle-ci se traduit naturellement dans le modèle Cassandra par l'imbrication des lignes dans une partition. La partition et les colonnes statiques représentent l'entité contenante, Invoice dans notre cas, et les lignes l'entité contenue, ici InvoiceItem.

Les données utiles décrivant le produit et le client sont copiées dans les lignes et partitions.

Lors de cette modélisation, nous mettons en forme deux mécanismes importants : la duplication et l'imbrication.

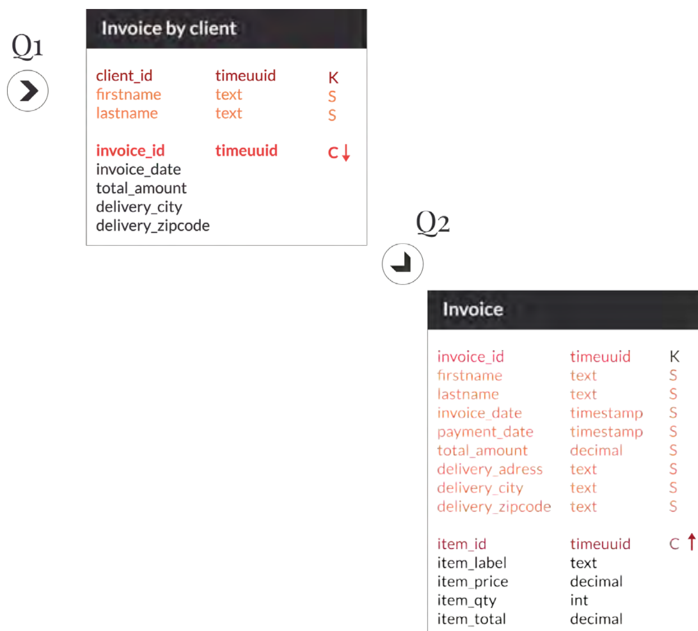
- la duplication est mise en œuvre quand les données du client ou

du produit sont copiées dans la table invoice,
- l'imbrication est mise en œuvre lorsque les lignes de factures sont incluses dans la partition qui représente la facture.

Il est important de distinguer la duplication technique du nom du client qui est une forme de dénormalisation motivée par les performances applicatives et la copie fonctionnelle de l'adresse qui fait partie du modèle conceptuel et répond à un besoin métier.
La table invoice ainsi produite ressemble fortement à la table invoice_by_client. Elles diffèrent cependant dans l'interprétation d'un même élément. La table invoice représente deux entités imbriquées. Elle est la source de vérité des données qui y sont conservées à l'exception des champs copiés depuis Client et Produit.
La table invoice_by_client représente une relation 1-* : seul les identifiants y sont significatifs, les autres données ne sont que des copies de dénormalisation.

À la fin, on obtient le diagramme suivant :

DIAGRAMME DE CHEBOTKO : FACTURES DES CLIENTS



6.1.4.Modèle physique

Le modèle logique obtenu, il est d’usage de le transformer en modèle physique.

Cela consiste en général à dégrader le modèle logique pour qu’il puisse fonctionner avec les vrais contraintes opérationnelles. Dans le cadre de Cassandra, il convient de vérifier que la taille d’une partition ne devienne jamais trop grosse. Il est souhaitable de limiter une partition à 100 000 valeurs et 100 Mo pour éviter qu’une partition trop lourde ne plombe les performances.

Ici, le nombre de partitions de la facture ne risque pas de déborder. Il nous faut vérifier qu’il n’existe pas de super client qui dispose d’un nombre de factures gigantesque. Nous supposerons ici que ce n’est pas le cas. Nous verrons par la suite comment adapter le modèle pour éviter les partitions de très grande taille.

Il ne nous reste plus qu’à produire nos scripts CQL de création de tables.

```
create keyspace invoice WITH replication = {‘class’: ‘SimpleStrategy’,
‘replication_factor’: 1 };

use invoice;

create table invoice (
 invoice_id timeuuid,
 firstname text static,
 lastname text static,
 invoice_date timestamp static,
 payment_date timestamp static,
 total_amount decimal static,
 delivery_address text static,
 delivery_city text static,
 delivery_zipcode text static,
 item_id timeuuid,
 item_label text,
 item_price decimal,
 item_qty int,
 item_total decimal,
 primary key (invoice_id, item_id)
);

create table invoice_by_client (
```

```
 client_id timeuuid,
 firstname text static,
 lastname text static,
 invoice_id timeuuid,
 invoice_date timestamp,
 total_amount decimal,
 delivery_city text,
 delivery_zipcode text,
 primary key (client_id, invoice_id)
)
with clustering order by (invoice_id desc);
```

6.2. Recherche multicritères

Nous avons vu comment modéliser une facture. Dans ce contexte, nous pouvons rechercher les factures associées à un client. Il arrive cependant que l'on souhaite rechercher les données via d'autres critères, voire selon une combinaison de critères possibles. C'est la fameuse recherche multicritères classique dans les applications traditionnelles. Si elle a tendance à disparaître pour des recherches simplifiées, elle possède ses avantages et est parfois demandée avec insistance par le client.

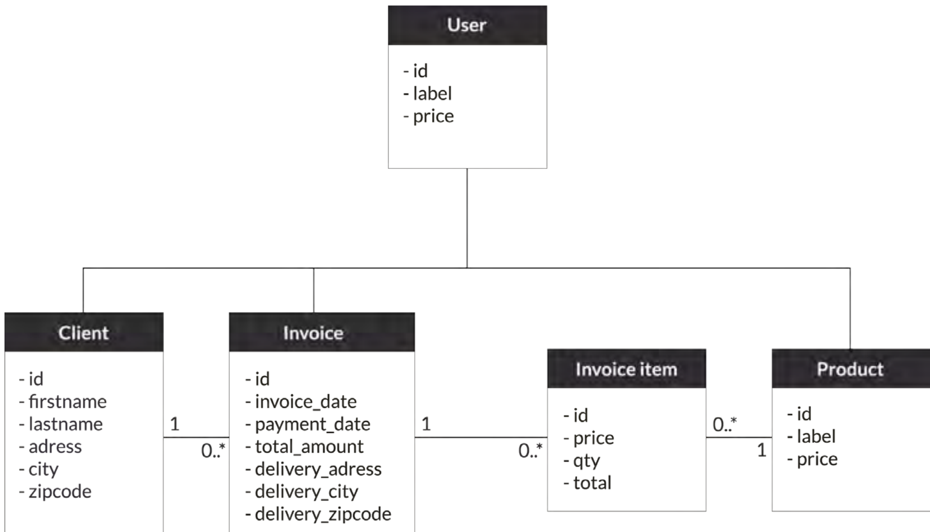
Nous allons voir maintenant comment résoudre le problème.

6.2.1.Modèle conceptuel

Pour cette partie, nous reprendrons le modèle de la gestion de facture que nous modifierons un peu. Maintenant, notre application de facturation est fournie en mode SaaS hébergée par nos soins via la meilleure solution cloud du marché qui nous permet d'avoir une immense base de données qui contient toutes les données de tous nos utilisateurs.
Nous introduisons donc la notion d'utilisateur de l'application. Toutes les données sont maintenant cloisonnées par utilisateur.

Notre modèle conceptuel est donc le suivant :

MODÈLE UML



L'ajout de l'entité utilisateur parait anecdotique, mais c'est un prérequis indispensable au bon fonctionnement de la recherche multicritères.

6.2.2.Modèle logique

DIAGRAMME DE CHEBOTKO

Invoice		
invoice_id	timeuuid	K
firstname	text	S
lastname	text	S
invoice_date	timestamp	S
payment_date	timestamp	S
total_amount	decimal	S
delivery_adress	text	S
delivery_city	text	S
delivery_zipcode	text	S
item_id	timeuuid	C ↑
item_label	text	
item_price	decimal	
item_qty	int	
item_total	decimal	

Comme nous l'avons fait lors de la modélisation des factures, nous créons une seule table qui imbrique l'entité InvoiceItem dans une partition représentant une entité Invoice. Conformément à la modélisation classique d'une relation de composition 1-n.

6.2.3.Recherche multicritères

Nous voulons pouvoir effectuer une recherche multicritères avec les caractéristiques suivantes.

Deux critères sont obligatoires:
l'utilisateur, est un critère implicite qui n'a pas à être saisi, puisque chacun ne peut chercher que dans ses données ;
la date de facturation, sous la forme d'une plage de dates, début et fin.

S'ajoutent ensuite des critères facultatifs :

- le nom du client,
- le prénom du client,
- la ville de livraison.
- le code postal de livraison.

La recherche est paginée. Les factures sont présentées dans l'ordre antéchronologique.

Pour un tel cas d'usage, la solution la plus évidente est de coupler un moteur de recherche avec notre base. Nous pourrions ainsi déployer un cluster Elasticsearch ou Solr et y indexer nos factures à la création. Cette solution est la meilleure en terme de fonctionnalités, mais reste complexe. Nous devons créer et maintenir un deuxième cluster et nous assurer que les données qu'il contient sont cohérentes avec celles de Cassandra. Et ce n'est pas aussi simple qu'on peut le croire, surtout avec un base gigantesque telle que le produit notre application SaaS.

Ici, nos besoins étant limités à des recherches simples et exactes, nous ne souhaitons pas ajouter cette complexité.

Les experts qui me liront diront que, comme nous avons une base installée importante, nous avons pris du support auprès de DataStax et que dans la distribution DataStax Entreprise (DSE), il y a une version

de Solr prête à l'emploi et totalement intégrée. Nous pouvons ainsi bénéficier des avantages d'un moteur de recherche documentaire en ne gérant qu'un cluster, la cohérence entre les tables et l'index étant, de plus, garantie par DSE.

Tout ceci est parfaitement vrai, mais ne pourrait nous suffire. En effet, si vous avez bien remarqué les critères de recherche, nous ne cherchons que sur des colonnes statiques or l'intégration de Solr ne les gère pas à ce jour. De plus, nous cherchons des partitions (des factures) alors que ladite recherche Solr nous retourne des lignes. L'intégration Solr ne nous est donc d'aucune aide pour l'instant.
On pourrait penser aux index secondaires. Mais d'une part, on ne peut en utiliser qu'un à la fois et, d'autre part les index secondaires de Cassandra sont tellement particuliers qu'il vaut mieux les fuir.

6.2.4.Table d'index

Le mieux est de se rapprocher du fonctionnement des index inversés. Pour cela nous allons créer une table d'index pour chaque critère secondaire. Une table d'index ne contient pas d'autres données que sa clé primaire. La clé de partition contient l'identifiant de l'utilisateur, la date de facturation (sans les heures), le critère secondaire. La clé primaire se termine par l'identifiant de la facture comme unique clustering column classée dans l'ordre décroissant pour que les factures les plus récentes soient présentées en premier.

Notez que jusqu'à la version 2.1 de Cassandra, il faut gérer soit même le contenu de la table, écrivant dans l'index en même temps. Cette gestion est facilitée parce qu'une facture n'est en principe jamais modifiée, tout changement donnant lieu à l'édition d'une nouvelle facture qui annule la précédente.
À partir de la version 3.0, nous bénéficions des vues matérialisées. Il suffit de la créer avec la commande suivante pour que Cassandra prenne en charge la dénormalisation des données.

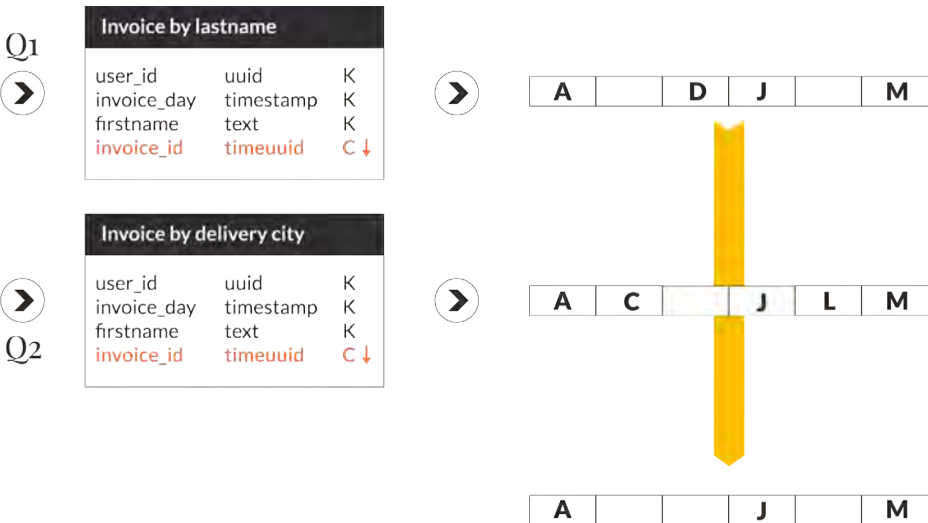
```
-- A partir de Cassandra 3.0
CREATE MATERIALIZED VIEW invoice_by_firstname
                      AS
                  SELECT invoice_id
                    FROM invoice
```

```
                  WHERE firstname IS NOT NULL
            PRIMARY KEY ((user_id, invoice_day, firstname), invoice_id)
WITH CLUSTERING ORDER BY (invoice_id DESC)
```

6.2.5.Chercher dans les index

La recherche proprement dite s'effectuera simplement :
Pour une journée donnée et pour chaque critère de recherche renseigné, on effectue une recherche sur l'index correspondant au critère. Toutes les recherches sont envoyées en parallèle grâce à l'API asynchrone du driver Cassandra. Comme les résultats sont classés dans l'ordre décroissant, l'intersection des résultats est effectuée simplement en parcourant les resultSets en parallèle en ne conservant que les valeurs présentes dans tous.
On s'arrête dès qu'on a assez de résultats pour remplir la page ou qu'un des resultSets est vide.

RECHERCHE EN PARALLÈLE SUR DEUX INDEX ET INTERSECTION DES RÉSULTATS EN MÉMOIRE



Si la première journée ne suffit pas à remplir une page de résultat, on recommence la journée précédente jusqu'à atteindre le début de la plage demandé.

La combinaison de la pagination des résultats, de la lecture paresseuse des resultSets et du classement cohérent des identifiants, permet à la fois d'économiser la mémoire de travail et de ne transférer vers le client que les données nécessaires. La mémoire nécessaire pour traiter une requête est bornée par la mémoire nécessaire pour stocker une page de résultat et les tampons des resultSets.

Une fois qu'on a obtenu une page d'identifiants, il suffit de charger toutes les factures à partir de leur identifiant. Dans ce cas, il est préférable de lancer toutes les requêtes - une par identifiant - en parallèle et de regrouper les résultats. Comme cela, on répartit les requêtes sur tous les nœuds du cluster. L'erreur serait de lancer une seule requête avec une clause where ... in (..., ...) qui charge un seul nœud coordinateur.

CHARGEMENT EN PARALLÈLE DES FACTURES DE LA PAGE DE RÉSULTAT

8f5b69ee-0ad00-11e5-a6c0-169f925ec7b7

8f5b69ee-0ad00-11e5-a6c0-169f925ec7b7

8f5b69ee-0ad00-11e5-a6c0-169f925ec7b7

8f5b69ee-0ad00-11e5-a6c0-169f925ec7b7

8f5b69ee-0ad00-11e5-a6c0-169f925ec7b7

8f5b69ee-0ad00-11e5-a6c0-169f925ec7b7

8f5b69ee-0ad00-11e5-a6c0-169f925ec7b7

8f5b69ee-0ad00-11e5-a6c0-169f925ec7b7

8f5b69ee-0ad00-11e5-a6c0-169f925ec7b7

8f5b69ee-0ad00-11e5-a6c0-169f925ec7b7

Invoice		
invoice_id	timeuuid	K
firstname	text	S
lastname	text	S
invoice_date	timestamp	S
payment_date	timestamp	S
total_amount	decimal	S
delivery_adress	text	S
delivery_city	text	S
delivery_zipcode	text	S
item_id	timeuuid	C ↑
item_label	text	
item_price	decimal	
item_qty	int	
item_total	decimal	

En Java 8, le code de réconciliation s'écrit simplement comme ceci :

```
private int loadInvoices(List<Invoice> resultList, Iterator<UUID> uuidIt,
int limit) {
    List<CompletableFuture<Invoice>> futureList = new ArrayList<>(limit);
    for (int i = 0; i < limit && uuidIt.hasNext(); ++i) {
        futureList.add(invoiceRepository.findOne(uuidIt.next()));
    }
    futureList.stream()
        .map(CompletableFuture::join)
        .forEach(resultList::add);

    return futureList.size();
}
```

Cette méthode charge les factures d'une journée dans la liste de résultats passée en argument dans la limite de limit documents.
Le deuxième argument, uuidIt, est l'itérateur qui correspond à l'intersection des lectures d'index qui est ici consommé de façon paresseuse.

6.2.6.Complexité

Tout comme nous avons limité l'empreinte mémoire utilisée, il est important de vérifier que le nombre de requêtes est maîtrisé. Un nombre de requêtes trop important risquerait de surcharger le cluster et ralentir les performances de façon générale.
Ici, nous effectuons au maximum une requête d'index pour chaque journée dans la plage de dates et pour chaque critère secondaire renseigné. Chacune de ces requêtes est de niveau de complexité partition by query qui correspond à la complexité minimale pour une requête Cassandra. À ceci s'ajoute une requête par élément trouvé dans la limite du nombre d'éléments dans la page.
Ainsi, si nous avons une requête avec 3 critères, une plage de 7 jours et une page de 100, le nombre de requêtes est inférieur ou égal à 121 (3 × 7 + 100).
Il suffit de borner la largeur possible de l'intervalle des dates et la taille des pages pour borner le nombre de requêtes et assurer que la complexité de la recherche est de type partitions by query.
C'est moins rapide qu'une requête unitaire, mais ça reste scalable. Sur le terrain, on obtient de très bonnes performances avec 5000 utilisateurs simultanés et 99% des temps de réponse sous la centaine

de millisecondes avec un seul serveur frontal et un cluster de 5 nœuds Cassandra.
Autrement dit, si Cassandra n'est pas faite pour la recherche multicritères, il est possible, dans certains cas, de lever cette limitation et de la faire fonctionner correctement.

6.3. Modélisation d'un panier

Avant de pouvoir passer une commande et plus encore de produire une facture, votre client va devoir lister les articles qu'il souhaite acheter. Pour cela, l'application propose généralement d'utiliser un panier, liste remplie au fur et à mesure par l'utilisateur. C'est de cet objet particulier et important que nous allons parler aujourd'hui.

6.3.1.Panier

Les paniers ne se limitent pas aux sites marchands. On les retrouve dans la plupart des applications où les utilisateurs sont conduits à faire une sélection de plusieurs éléments, sur lesquels ils effectuent une opération en masse par la suite. Ce peut être une liste d'articles qui seront achetés en fin de navigation. Dans un logiciel de facturation, ce pourrait être une sélection de factures sur lesquelles on veut envoyer une relance. On peut aussi penser aux listes de préférences ou aux listes de souhaits. Dans sa forme classique, un panier est donc une liste qui est remplie progressivement par ajout et retrait d'éléments. Souvent, le contenu sera exploité par une action qui provoquera son vidage.
Les éléments d'un panier ne sont pas classés. Il est fréquent que l'affichage classe les éléments, éventuellement triés par catégorie. Le classement et le tri sont des règles d'affichage qui sont réalisées par le tiers de présentation.

Par ailleurs, un panier doit pouvoir être manipulé de façon concurrente. On imagine souvent une personne seule devant son ordinateur en train de sélectionner deux livres, l'un après l'autre, puis de les commander. Mais ce n'est pas le seul cas d'usage. Vous rencontrerez aussi le cas d'un couple qui effectue ses courses en ligne chacun avec son ordinateur connecté sur le même compte. Chacun ajoutant ou supprimant des articles du panier.
À partir du moment où l'on travaille sur le web, il faut partir du principe

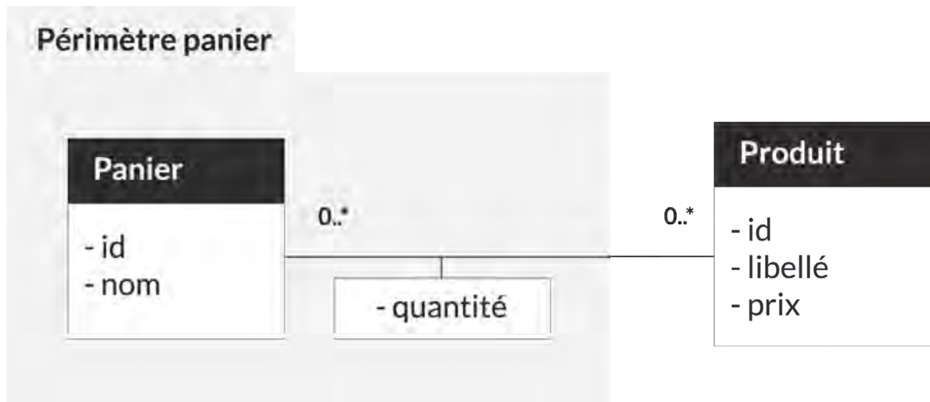
que les requêtes vont s'exécuter en parallèle.

6.3.2.Modèle fonctionnel

Le modèle fonctionnel du panier est donc simple : c'est un sac, c'est-à-dire un ensemble qui peut contenir plusieurs fois le même élément. Le nombre de répétitions de l'élément étant conservé. Pour traiter le cas général, nos paniers seront nommés pour être distingués entre eux.

Le modèle conceptuel est donc le suivant :

MODÉLISATION DU PANIER

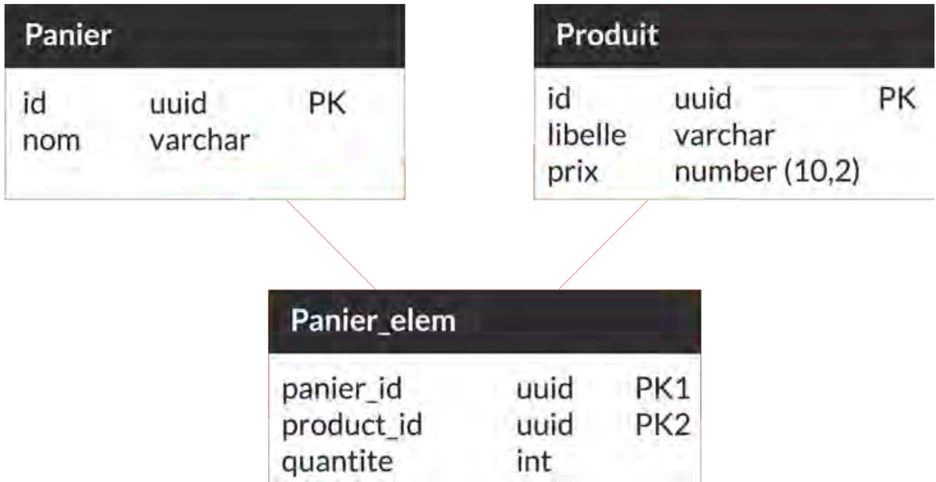


6.3.3.Modèle logique

Avec une base relationnelle, le modèle de base se décline très simplement par l'utilisation de deux tables : l'une représente l'entité panier et l'autre la jointure N-N entre les entités Panier et Produit.
La modélisation de l'entité Produit ne sera pas traitée ici. On simplifiera le propos en considérant qu'il est représenté par une table unique.

On obtient le schéma suivant :

MODÈLE MERISE



6.3.3.1.Modèle naïf

Prenons le cas de Dave. Il connaît bien les bases de données relationnelles avec lesquelles il travaille régulièrement. Il utilise Cassandra depuis peu de temps, mais pense en avoir compris le fonctionnement. D'ailleurs, le modèle n'est-il pas proche ? Cassandra stocke les données dans des tables et CQL ressemble à s'y méprendre à SQL. Bien sûr, il faut s'adapter aux spécificités de la base et à l'absence de jointure. C'est pourquoi Dave sait qu'il doit stocker toutes les informations concernant le panier et sa jointure avec les produits dans une seule table. Heureusement, il a déjà lu la partie "Modélisation facture et commande". Ici, la relation n'est pas une relation de composition, les lignes ne représenteront pas l'entité Produit, mais le lien vers l'entité.

En suivant ce principe, Dave modélise sa table ainsi :

- la clé de partition est l'identifiant du panier,

- le nom du panier est contenu dans une colonne statique (attribut de la partition),
- l'identifiant du produit est utilisé comme l'unique colonne de clustering,
- la quantité est une colonne normale,
- le libellé du produit et son prix peuvent être dénormalisés sous la forme de colonnes normales.

MODÉLISATION NAÏVE

Panier		
id	uuid	K
product_id	uuid	C ↑
nom	text	S
quantite	in	
libelle	text	
prix	decimal	

Ainsi, nous garantissons l'unicité de la présence d'un produit, la gestion du nom du panier, du nombre d'éléments et les dénormalisations utiles. Pour lire le contenu du panier, une requête suffit :

```
select *
  from panier
 where id = ?
```

Et notre développeur est content. Sauf que ...

6.3.3.2.Gestion de la concurrence

Sauf que maintenant, il faut pouvoir gérer les modifications concurrentes. Et nous tombons dans le cas classique d'une modification par lecture puis écriture de la valeur modifiée.
Lors de l'utilisation d'une base de données relationnelle, la solution consiste à poser un verrou à la lecture puis à faire la modification. Le cas

particulier de la première insertion devant être prise en compte.
Or Cassandra ne permet pas de poser de verrou. «Mais, se dit Dave, il y a les transactions légères qui fonctionnent comme un compare-n-swap (CAS) au niveau de la ligne». Il suffit donc pour lui de suivre l'algorithme suivant :

```
select quantite
  from panier
 where id = :panier_id
   and product_id = :product_id
```

Si on a un résultat alors on définit :
`nv_quantite ← anc_quantite + quantite_ajoute`
et on exécute la commande suivante :

```
update panier
   set quantite = :nv_quantite
 where id = :panier_id
   and product_id = :product_id
    if quantite = :anc_quantite
```

Sinon on exécute :

```
insert (id, product_id, quantite, libelle, prix)
values (:id, :product_id, :quantite_ajoute, :libelle, :prix)
if not exists
```

Dans les deux cas, la deuxième commande peut échouer en cas de modification parallèle. Dans ce cas, on recommence à la première étape. Avec ce fonctionnement, Dave Lopernahif obtient rapidement une application qui fonctionne.

6.3.3.3.Une modélisation efficace

Cependant, la modélisation de notre ami a deux problèmes importants :

- l'utilisation des transactions légères est lente. Elle est même très lente, car elle requiert quatre échanges entre le nœud coordinateur de la requête et les répliques,
- de plus, le niveau de cohérence est de type SERIAL, qui est une cohérence immédiate forte et ne permet pas de bénéficier de la cohérence à terme offerte par Cassandra. Il en résulte une réduction de la haute disponibilité.

Cette modélisation n'est donc pas satisfaisante et il nous faut revoir complètement la modélisation.

On pourrait vouloir utiliser les compteurs. Seulement, une table contenant des colonnes de type compteur ne peut pas contenir de colonne d'un autre type, ce qui complique le modèle. Mais aussi, les requêtes d'augmentation et de réduction du compteur ne sont pas idempotentes, ce qui les rend peu fiables et limite leur utilisation à des comptages statistiques où un certain taux d'erreur est autorisé. Pour obtenir une représentation efficace, il faut se souvenir que Cassandra est très fort pour ajouter des données et moins pour les modifier. Ainsi, plutôt que de stocker le contenu de la liste, il est préférable de stocker les événements qui la modifient. Chaque événement est identifié par un identifiant unique UUID de type 1, c'est-à-dire horodaté.

Notre table sera donc structurée ainsi :

- la clé de partition est l'identifiant du panier,
- le nom du panier est contenu dans une colonne statique (attribut de la partition),
- l'identifiant de l'évènement constitue l'unique colonne de clustering classé par ordre croissant,
- les données de l'évènement (product_id, quantité et valeurs dénormalisées) sont stockées comme des colonnes ordinaires.

MODÈLE EFFICACE

Panier		
id	uuid	K
product_id	uuid	C ↑
nom	text	S
product_id	uuid	
quantite	in	
libelle	text	
prix	decimal	

Lors de la lecture, les événements sont lus dans l'ordre chronologique et le panier est reconstitué par application successive des effets. Il n'y a

ici qu'un type d'événement qui correspond à une variation de la quantité qui peut être positive (ajout dans le panier) ou négative (retrait). Il faudra juste penser à retirer du panier final les articles dont la quantité est inférieure ou égale à zéro.

Cette modélisation a l'avantage de ne nécessiter que des ajouts de données, de gérer naturellement la concurrence et de ne nécessiter aucune lecture avant l'écriture. On est dans le cas optimal qui nous permettra de tirer le maximum de performances de Cassandra.

6.3.4.Gestion de la cohérence

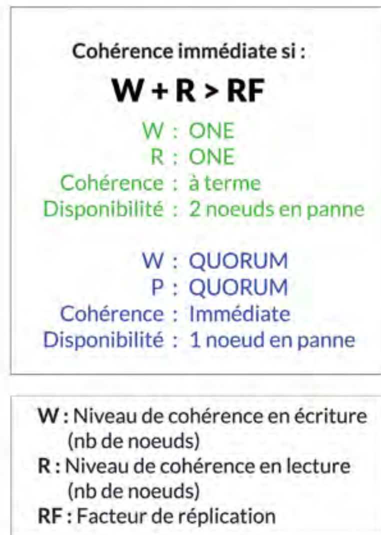
De plus, cette représentation du panier nous permet de relâcher la cohérence.
En effet, Cassandra, sur le modèle de Dynamo créée par Amazon pour gérer les paniers du site marchand, est conçue initialement comme une base cohérente à terme. L'idée est de privilégier la disponibilité et la résistance aux partitions réseau plutôt que la cohérence des données sur tous les serveurs. Amazon préfère ajouter un élément au panier, même s'il n'arrive pas à mettre à jour toutes ses répliques plutôt que de lever une erreur et voir son client potentiel partir finir son achat ailleurs. Les écarts entre les données étant réconciliées par le système par la suite.

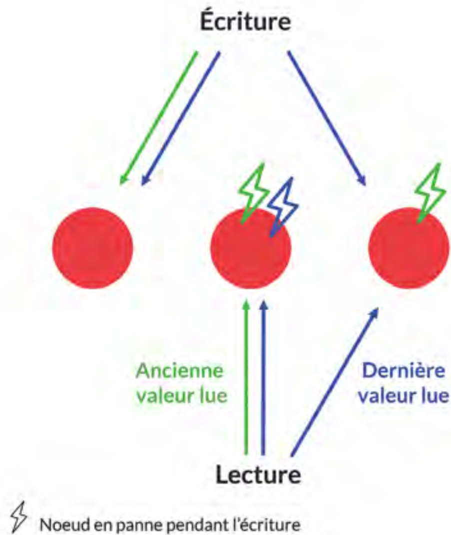
En fait, Cassandra donne la liberté au développeur de choisir s'il veut plus de cohérence ou une plus grande disponibilité. Selon votre cas métier, vous devrez choisir la très haute disponibilité (et la latence d'écriture la plus faible) ou la cohérence immédiate.

Le réglage s'effectue en choisissant le niveau de cohérence de chaque requête. Au moment de l'écriture, le niveau de cohérence est le nombre de nœuds qui ont acquitté l'écriture. Au moment de la lecture, c'est le nombre de nœuds qui ont répondu. Lorsque deux nœuds fournissent une valeur différente, la plus récente est conservée et un mécanisme de réparation se met en place.
Les principaux niveaux de cohérence offerts par Cassandra sont ONE, QUORUM et ALL. Souvent utilisés en ONE-ONE pour une cohérence à terme ou QUORUM-QUORUM pour une cohérence immédiate. D'autres combinaisons sont possibles, mais elles sont rares et ne doivent

être utilisées que si l'on en maîtrise les conséquences.

COHÉRENCE ET PANNE D'UN NŒUD





6.4. Stockage de fichiers dans Cassandra

Nous allons voir comment stocker des fichiers dans Cassandra. C'est une opération assez facile qui permet de bénéficier de tous les avantages d'un cluster hautement disponible dont les données sont réparties et répliquées dans plusieurs centres de calcul.

Enregistrer des fichiers est un besoin courant. Les réseaux sociaux enregistrent les photos de leurs utilisateurs. Les catalogues de produits stockent leurs images. Les applications de messagerie ou de gestion de dossiers permettent d'attacher des pièces jointes. Sans parler des GED ou des systèmes de gestion d'archives qui dépassent le cadre de cet article. À chaque fois, vous aurez le choix entre conserver les fichiers dans la base de données applicatives ou dans un système spécialisé à part. Il s'agit d'un choix d'architecture qui dépend des besoins et des volumes de chaque cas.

Stocker dans la base de données permet de simplifier l'architecture du système en évitant d'ajouter un composant technique supplémentaire. Cela facilite aussi la gestion des sauvegardes puisque toutes les données sont présentes au même endroit. Avec un seul système à sauvegarder et à restaurer, la cohérence des données après restauration est garantie. Ce choix est adapté aux applications qui conservent peu de petits fichiers.
Disposer d'un système spécifique, c'est avoir la garantie de conserver les fichiers dans un système adapté. On évite ainsi de perturber les performances de la base de données avec les données binaires. Et l'on peut utiliser des mécanismes de stockage moins chers. Ce choix sera généralement mis en œuvre quand le nombre ou la taille des fichiers devient important.
Notez que nous ne choisissons pas une solution particulière. Le stockage dans Cassandra peut résulter d'un stockage dans la base applicative (qui sera ici C*) ou d'un stockage dans un système dédié.

6.4.1.Première modélisation

Nous allons commencer par le cas le plus simple à modéliser. Il s'agit tout simplement de créer une table pour conserver l'intégralité des données. Cette table unique est identifiée par l'identifiant du fichier. Elle est interrogée à partir de celui-ci (Requête Q1).
Elle contient en plus des métadonnées, ou données descriptives, qui dépendent de votre application.

Dans l'exemple, les métadonnées sont :

- la taille du fichier,
- le propriétaire,
- la date de création,
- un ensemble de permissions.

Le contenu y est enregistré à côté des métadonnées descriptives.

Q1

Fichier		
file_id	timeuuid	K
owner	text	
taille	int	
creation_date	timestamp	
permissions	set<text>	
content	blob	

Q1 : lire un fichier à parir de son identifiant

Cette solution est actuellement mise en œuvre chez un de nos clients.

Elle présente des avantages majeurs :

- le modèle est simple et facile à comprendre.
- une seule requête suffit pour récupérer le fichier et ses métadonnées.

Elle présente aussi un inconvénient vital :

- ce modèle n'est viable que si la taille des fichiers est petite et bornée. En effet, Cassandra est sensible aux partitions de trop grandes tailles qui saturent sa mémoire et sollicitent le garbage collector.

Ainsi, vous pouvez envisager cette solution lorsque vos fichiers ont une taille moyenne de 50 Ko et maximale de 5 à 10 Mo.

6.4.2.Découper les fichiers

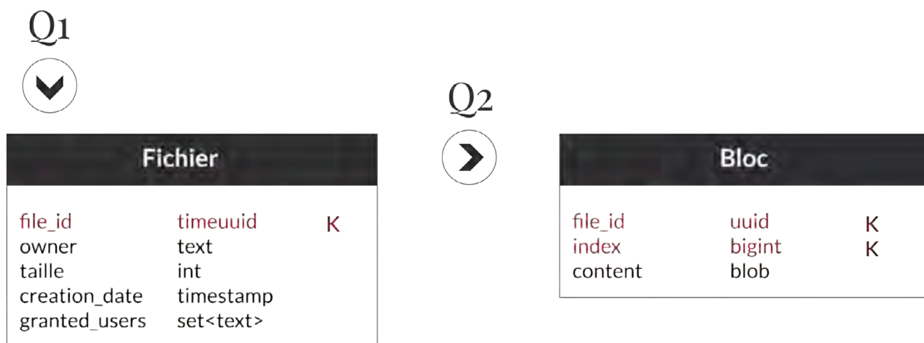
Dans le cas général, il sera préférable de découper les fichiers en blocs qui seront stockés dans des partitions différentes.
Pour cela, deux tables seront utilisées :

- la première contient les métadonnées du document,
- et l'autre le contenu.

Le choix de la taille d'un bloc est l'objet d'un compromis entre le nombre de blocs, et donc le nombre de requêtes nécessaires pour la lecture et l'écriture du contenu, et la fluidité de leur gestion. Une taille de 64 Kio semble raisonnable de nos jours. C'est d'ailleurs, la taille des blocs de MongoDB GridFS.

On obtient alors le modèle suivant :

DIAGRAMME DE CHEBOTKO



Q1 : lire un fichier à partir de son identifiant- Q2 : lire le contenu d'un fichier bloc par bloc

Cette fois-ci, on a :

- une table contenant les métadonnées,
- une table contenant les blocs de données.

Indiquer la taille du fichier est devenue nécessaire, car elle permet au lecteur d'en déduire le nombre de blocs à lire. À la place, on aurait pu ajouter le nombre de blocs, mais la taille exacte est plus intéressante.

Chaque bloc est identifié par l'ID du fichier et un index représentant sa position dans le fichier. La clé primaire est identique à la clé de partition pour s'assurer de la maîtrise de la taille de cette dernière.

Le modèle s'organise autour de deux requêtes:

- Q1 vérifie l’existence du fichier et lit les métadonnées, dont la taille,
- Q2 lit un bloc de données.

En général, on boucle sur l’index pour lire toutes les valeurs.

6.4.3.Content Addressable Storage

La modélisation actuelle fonctionne correctement. Cependant, il est possible d’optimiser l’espace consommé. En effet, jusqu’à présent, un fichier déposé par des chemins différents sera enregistré plusieurs fois. Ce sera le cas si plusieurs personnes téléversent le même document. Imaginez le nombre de copies d’un courriel ou de ses pièces jointes dans un système d’archivage de la messagerie d’une grande entreprise !
Pour éviter le gaspillage, nous allons tenter de dédoublonner les fichiers et ne conserver qu’une copie du contenu.

Une solution naïve consisterait à chercher les fichiers de même contenus. Évidemment, on ne cherche pas le contenu lui-même, mais un condensat, ou une empreinte, qui permet de trouver le doublon. Si cette solution fonctionne en principe, elle n’est pas adaptée à Cassandra puisqu’elle impose une lecture avant l’écriture (read-before-write). En plus, il faudra prévoir une vue matérialisée pour permettre cette lecture.
On peut faire mieux en utilisant directement l’empreinte comme identifiant. Au lieu d’être arbitraire, l’identifiant devient intrinsèque au contenu. Sa valeur est déterministe et indépendante du contexte.

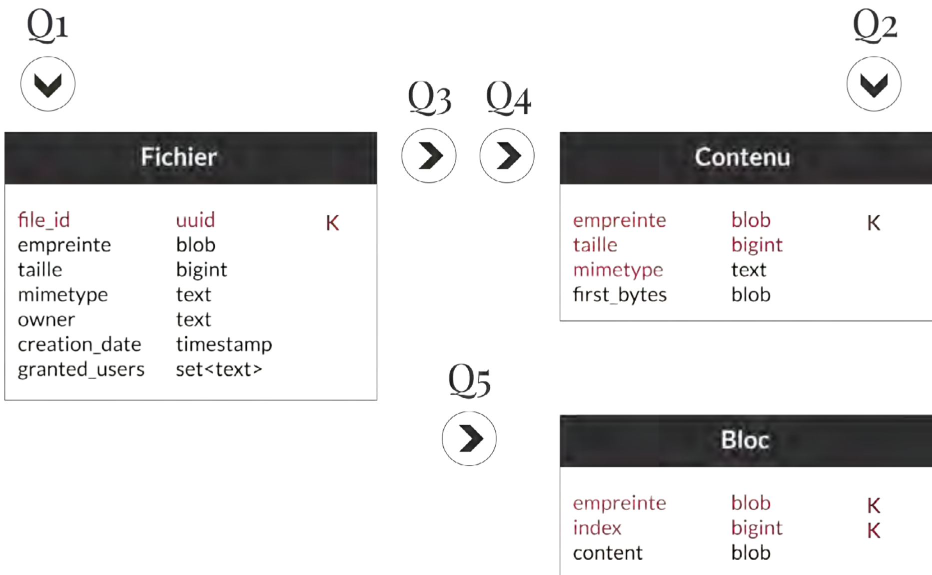
Cette pratique est suffisamment courante pour avoir un nom : Content Addressable Storage, ou stockage indexé par le contenu. Et vous l’utilisez tous les jours, puisque Git qui identifie ses objets par leur SHA-1 repose sur un CAS. Cependant, le CAS présente des contraintes. Par exemple, le contenu d’un fichier ne peut pas changer puisqu’un changement de contenu change l’identifiant. De plus, dans le cadre d’une gestion de fichiers, deux fichiers de même contenu sont identiques.

Dans le cas pratique, on utilisera le CAS pour la conservation du contenu et de sa description intrinsèque. On utilisera un stockage traditionnel pour la description contextuelle du fichier.

Ainsi, on a :

- une table fichier identifiée par un uuid contenant :
- les métadonnées contextuelles (propriétaire, date de création, permissions),
- le lien vers le contenu et des champs dénormalisés ;
- une table contenu identifiée par l'empreinte et contenant les informations descriptives du contenu,
- une table bloc avec le contenu découpé en bloc.

DIAGRAMME DE CHEBOTKO



Q1 : lire un fichier à partir de son identifiant - Q2 : trouver un contenu à partir d'une empreinte Q3 : lire le descriptif d'un contenu à partir de son empreinte - Q4 : lire les premiers octets d'un fichier - Q5 : lire le contenu d'un fichier bloc par bloc

Une optimisation utile permet de stocker les premiers octets, souvent le 1er Kio, dans les métadonnées (champ first_bytes). Ce peut être utile dans le cas où il est fréquent de ne lire que les entêtes (pour déterminer le type du fichier par exemple) ou lorsque l'application gère de nombreux petits fichiers.

Ce modèle fonctionne bien pour l'écriture sans modification des contenus. Dans le cas où un fichier peut être supprimé ou que son contenu peut être modifié, il faudra ajouter un mécanisme de collection des contenus inutilisés. Il s'appuiera sur un champ permettant de savoir si un contenu est utilisé.

6.5. Conclusion

Comme nous l'avons vu, le stockage efficace de fichiers dans Cassandra est relativement facile. La complexité dépendra du cas fonctionnel et des optimisations souhaitées. La seule difficulté se situe dans la maîtrise de la taille des partitions qui nous oblige à découper le fichier dans le cas général.

7 — RETOURS D'EXPÉRIENCE CASSANDRA

Cas client : refonte d'une plateforme d'acquisition/ valorisation de la donnée

7.1. Contexte client

Le positionnement métier concerne la délivrance d'informations financières et économiques stratégiques concernant des sociétés auprès de clients professionnels. Cette information est issue du traitement de sources de données multiples intégrées au fil de l'eau et dont l'historique est conservé pour les besoins statistiques et la production de rapports détaillés.
Pour répondre à des ambitions d'extension de l'offre, accompagnée d'une amélioration de l'expérience client, une refonte de la plateforme a été envisagée. Les enjeux sont de pouvoir intégrer de nouveaux flux de données, d'améliorer sensiblement la performance de la chaîne de traitement et de permettre de valoriser la donnée par l'utilisation de nouveaux algorithmes (Machine Learning).

7.2. Contraintes

Le système initial a atteint certaines limites et ne permet pas d'envisager les évolutions souhaitées par la Direction.
La volumétrie des données pénalise les performances sur certains traitements et met en évidence l'impossibilité d'évoluer par l'intégration de nouveaux flux, aux volumes potentiellement conséquents.
La chaîne d'acquisition manque de réactivité et rapidité pour intégrer la donnée et la mettre à disposition des clients dans les délais souhaités.

7.3. Contribution Ippon

Réalisation d'une étude d'analyse de l'existant et proposition de 3 scénarios d'architectures adaptés au contexte du client.

Réalisation en co-développement d'un POC pour valider les hypothèses du scénario retenu :

- mise en place d'un cluster Cassandra pour le stockage massif des données épaulé par le moteur de recherche Apache Solr pour répondre aux besoins de recherche multicritères,
- implémentation d'un module de calcul d'indicateurs reposant sur une architecture événementielle utilisant Kafka pour publier/consommer des événements à l'origine des traitements de la plateforme,
- mise en place du cluster sur l'infrastructure Ippon Hosting pour répondre aux contraintes de TTM du PoC.

7.4. Résultats obtenus

L'architecture permet le traitement des données de bout en bout dans le temps souhaité. Le PoC a validé les hypothèses techniques et a convaincu la Direction de déployer à l'échelle du SI la nouvelle plateforme en suivant un scénario de "progressive rollup".
Le projet intègre une dimension d'accompagnement pour l'industrialisation de la plateforme et l'intégration dans le paysage actuel du SI.

Cas client : définition et mise en oeuvre d'une architecture lambda pour un acteur du web

7.5. Contexte client

Acteur majeur du secteur des comparateurs d'assurance, notre client a souhaité faire évoluer sa plateforme historique afin de répondre aux attentes stratégiques autour de la gestion de la donnée portées par la Direction du groupe. De plus, la croissance rapide du trafic du site impose aux équipes d'adopter une architecture de gestion des données capable de gérer des volumes conséquents avec des coûts linéaires.

7.6. Contribution d'Ippon

Ippon a apporté son expertise sur Cassandra et Sprak avec un format du type : « MasterClass» pour lancer le projet et ensuite avec de l'accompagnement technique des équipes IT. Les MasterClass Ippon permettent de réunir les équipes technique d'un client et trois experts Ippon autour d'un sujet technique en particulier. L'objectif est d'apporter, par une réflexion de groupe efficace et dans un délai court (une journée), une architecture validée par notre pôle conseil.
Ippon a accompagné le client dans l'évolution du site web par la mise en oeuvre d'une architecture lambda utilisant Cassandra pour le Runtime (stockage des données live issues du site) et Cassandra/Spark pour la BI. Cette plateforme a été mise en oeuvre sans jamais arrêter le site, avec un fonctionnement en Dual Run sur 6 mois.

7.7. Résultats obtenus

La nouvelle plate-forme offre des performances élevées et un fonctionnement ininterrompu aptes à répondre aux objectifs de croissance du client. Elle a permis la construction d'une vue unique client à 360° ainsi que la mise en oeuvre de Real time analytics et de Machine learning pour l'automation.

8 — IPPON DANS LA COMMUNAUTÉ CASSANDRA

Simplifier l'utilisation de Cassandra pour le projet open source JHipster

Le projet open source JHipster est un générateur d'applications bien connu de la communauté Java.
Avec une configuration minimale, il permet de commencer rapidement le développement d'applications intégrant frontend, backend, sécurité et base de données.
Cassandra est évidemment une des base de données proposée et JHipster génère une configuration de connexion entièrement fonctionnelle.

Mais, il est souvent difficile de configurer et maintenir une base Cassandra locale pour les développeurs.
De plus, il n'existe pas d'outil standard pour gérer le suivi des changements de schéma, tel que Liquibase ou Flyway pour les bases SQL, ce qui rend difficile la synchronisation entre les environnements et les configurations locales.

La vision de JHipster étant de fournir un environnement par défaut le plus simple et productif possible pour les développeurs, Ippon a contribué le développement d'un tel outil.

Une configuration Docker - la solution de facto de containerisation - est générée par défaut pour démarrer localement un cluster en une seule commande.
Un ensemble de scripts enregistre automatiquement les migrations de schémas exécutées dans une table dédiée, ce qui permet la migration automatique de la base à chaque déploiement d'une nouvelle version de l'application.

L'outil est utilisé à la fois par les développeurs pour tester les scripts et synchroniser leur cluster local et en production.
La standardisation des scripts de migration est également utilisée par les

tests d'intégrations pour démarrer un cluster Cassandra en mémoire.

Quelques exemples d'utilisation de docke-compose et de l'outil de migration...

8.1. Lancer l'application packagée et un cluster Cassandra

Produire une image docker de l'application :

```
./mvnw package -Pprod docker:build
```

Puis lancer la configuration docker-compose :

```
docker-compose -f src/main/docker/app.yml up
```

compose va démarrer 4 containers Docker :

- l'application packagée,
- un premier noeud Cassandra qui va servir de point de contact,
- un second nœud Cassandra pour rejoindre le cluster,
- le service qui va exécuter les migrations de schémas.

Grâce à docker-compose, il est très facile de rajouter un nœud au cluster Cassandra :

```
docker-compose -f src/main/docker/app.yml scale yourapp-cassandra-node=2
```

Le nouveau nœud va automatiquement rejoindre le cluster Cassandra en utilisant le premier nœud comme point de contact.

Le container responsable de la migration va utiliser un dossier spécifique - par convention config/cql/changelogs - pour exécuter les scripts CQL qui s'y trouvent.
Tout comme **liquibase**, le service de migration va utiliser une table dédiée - nommée schema-version - pour garder une trace des scripts déjà exécutés.

8.2. Modifier le schéma avec JHipster et utiliser l'outil de migration

Avec JHipster il est possible de créer une nouvelle entité en une commande et quelques questions :

```
$ yo jhipster:entity book

Generating field #1
? Do you want to add a field to your entity? Yes
? What is the name of your field? title
? What is the type of your field? String
? Do you want to add validation rules to your field? No

Generating field #2
? Do you want to add a field to your entity? Yes
? What is the name of your field? author
? What is the type of your field? String
? Do you want to add validation rules to your field? No

Generating field #3
? Do you want to add a field to your entity? Yes
? What is the name of your field? releaseDate
? What is the type of your field? LocalDate (Warning: only compatible with cassandra v3)
? Do you want to add validation rules to your field? No

? Do you want to use a Data Transfer Object (DTO)? No, use the entity directly
? Do you want to use separate service class for your business logic? No, the REST controller should use the repository directly
```

JHipster va générer le code AngularJS pour le frontend, et le code Java pour réaliser les opérations CRUD de base. Il va également générer le script CQL pour créer la table Cassandra de cette nouvelle entité:

```
CREATE TABLE IF NOT EXISTS book (
    id uuid,
    title text,
    author text,
    releaseDate date,
    PRIMARY KEY(id)
);
```

Sans avoir stoppé le cluster, il est possible de relancer l'outil de migration avec docker-compose :

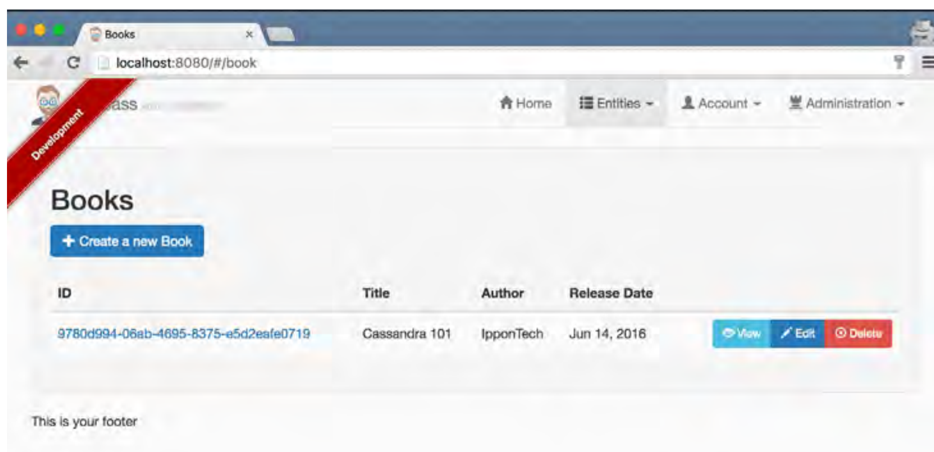
```
docker-compose -f src/main/docker/app.yml up yourapp-cassandra-migration
```

L'outil parcourt de nouveau les scripts CQL du dossier config/cql/changelogs mais n'exécute que ceux qui n'ont pas encore encore été exécutés en se basant sur les informations de la table “schema-version”.

Ensuite, une nouvelle instance de l'application peut-être démarrée :

```
./mvn spring-boot:run
```

JHipster a créé les écrans, le code java et l'outil a exécuté les scripts pour créer la nouvelle table Cassandra :



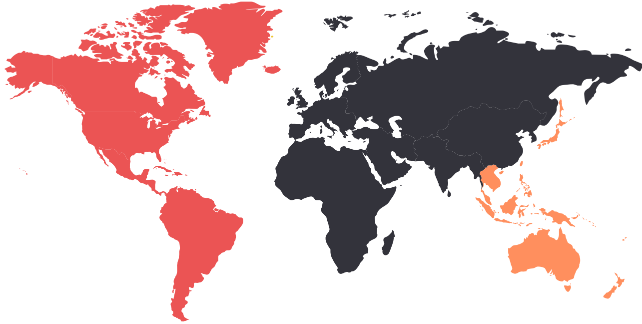
En générant un environnement par défaut fonctionnel à la fois pour le développement, les tests et le déploiement, JHipster est désormais un condensé des bonnes pratiques pour débuter le développement d'applications reposant sur Cassandra.
Le site du projet JHipster : **http://jhipster.github.io**

9 — CONCLUSION

Cassandra offre des possibilités en termes de performance et de capacité de stockage qui permettent de répondre à de nombreux nouveaux cas d'utilisation autour de la donnée. Le changement de paradigme du NoSQL, associé aux aspects techniques pointus en font aussi une solution qui demande un investissement pour en tirer le maximum de son potentiel. Il est en effet primordial d'intégrer, autant pour les "devs" que les "ops", son fonctionnement et cela passe par de l'accompagnement, de la formation et de la pratique.
Nous espérons que le potentiel de cette solution vous apparaît plus clairement après la lecture de ce livre blanc et que vous identifiez d'ores et déjà de possibles cas d'usage pour lesquels la solution offrirait un réel atout.

IPPON

Digital . Technologies . Hosting



PARIS
BORDEAUX
NANTES

RICHMOND, VA
WASHINGTON, DC
NEW-YORK

MELBOURNE
MARRAKECH

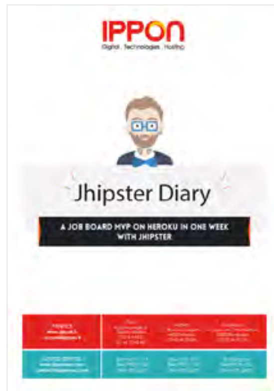
www.ippon.fr/contact
talents@ippon.fr
blog.ippon.fr

+33 1 46 12 48 48
@ippontech

Découvrez aussi nos

AUTRES RESSOURCES

sur ***http://www.ippon.fr/ressources/***



IPPON

Digital . Technologies . Hosting

www.ippon.fr
blog.ippon.fr

Paris
Nantes
Bordeaux
Washington, DC
Richmond, VA
New York, NY
Melbourne
Marrakech