

Introduction aux Bases de Données NoSQL

Au début des années 2000, les principaux acteurs du Web comme Yahoo, Google, Amazon ou eBay ont commencé à manipuler des volumes de données gigantesques qui atteignaient les limites des bases de données relationnelles traditionnelles. Un peu plus tard, avec le développement du Web social, des sites comme Facebook ou Twitter ont été confrontés aux mêmes problématiques. Afin de gérer leurs données, toutes ces sociétés ont dû développer en interne des solutions adaptées à leurs besoins spécifiques (GoogleFS et BigTable pour Google, Dynamo pour Amazon, Cassandra pour Facebook, ...). Ces SGBD avaient plusieurs points communs dont : ne pas être basés sur du relationnel et permettre une montée en charge horizontale (scalabilité horizontale) simplement en rajoutant des serveurs ou des datacenters.

Un certain nombre de ces SGBD sont ensuite tombés dans le domaine de l'open-source et cela a donné naissance à la mouvance NoSQL. Le NoSQL ne signifie pas 'Non SQL' mais plutôt 'Not Only SQL'. Le terme SQL n'est d'ailleurs pas très approprié (certains SGBD NoSQL comme Cassandra utilisent un dialecte SQL) car le sens du NoSQL est plutôt Not Only Relationnel. Cela signifie qu'on n'est pas obligé de stocker toutes les données dans du relationnel et que les entreprises peuvent utiliser simultanément plusieurs SGBD de types différents en fonction de leurs besoins.

Parallèlement à cela, avec le développement de l'informatique embarquée, la multiplication des capteurs et des objets connectés, le volume des données récoltées par les entreprises a commencé à croître de façon exponentielle. Certaines entreprises ont donc décidé d'utiliser les technologies développées par les géants du Web pour stocker et traiter de façon analytique les données qu'elles accumulaient ; de la business Intelligence (BI) sur des gros volumes de données non structurées. Cela a donné naissance au Big Data. Le Big Data ne se limite donc plus aux grands acteurs du Web et devrait donc concerner de plus en plus d'entreprises à l'avenir. Il s'appuie généralement sur des bases de données NoSQL.

1 Caractéristiques des bases de données NoSQL

Les bases de données NoSQL sont toutes très diverses. Mais, elles ont un certain nombre de points communs qui sont, en plus de ne pas être basées sur une technologie relationnelle, de supporter une architecture distribuée, de ne pas gérer les transactions ACID, de ne pas avoir de schéma prédéfini (schéma-less) et d'avoir des données dénormalisées.

1.1 Une architecture distribuée

Les bases de données NoSQL doivent pouvoir stocker et traiter des volumes de données très importants. Il faut donc qu'elles puissent monter en charge en ajoutant simplement des nouveaux serveurs (on parle de scalabilité horizontale). Pour cela, elles doivent proposer des mécanismes qui permettent la distribution et le traitement des données sur plusieurs nœuds.

Une architecture distribuée peut être avec ou sans maître. Avec une architecture sans maître (comme par exemple Cassandra), chaque nœud est équivalent. Lorsqu'un client émet une requête, il est mis en contact automatiquement avec le nœud qui contient les données. L'avantage d'une telle architecture est qu'il est facile de rajouter un nouveau nœud, et qu'il y a une forte résistance aux pannes (il n'y a pas de point de défaillance unique ; si un nœud tombe, le système continue à tourner). Cette architecture favorise la disponibilité des données (mais pas forcément la cohérence absolue).

Dans le cas d'une architecture avec maître (comme MongoDB), un nœud particulier joue le rôle du maître. C'est lui qui s'occupe de gérer les écritures et de les répliquer sur les nœuds esclaves (par contre les lectures peuvent être faites directement sur les nœuds esclaves).

L'architecture avec un maître favorise la cohérence (toutes les écritures passent par le nœud maître), mais il y a un point de défaillance unique (si le maître tombe) et la disponibilité des données est plus difficile à respecter. Le théorème CAP tente d'ailleurs de démontrer qu'un système distribué ne peut pas être à la fois cohérent et disponible.

En plus de partitionner les données sur les nœuds, les SGBD NoSQL dupliquent également ces données (on parle de répliques). Le but étant de garantir la disponibilité du système dans le cas où un nœud serait hors service.

Beaucoup de bases de données NoSQL gèrent la distribution des traitements (et la montée en charge horizontale) grâce au pattern MapReduce. Une fonction `map` permet d'étiqueter des données, et une fonction `reduce` de réaliser un traitement en fonction de cette étiquette. Ces fonctions peuvent être exécutées en parallèle sur plusieurs nœuds en même temps.

1.2 Pas de gestion ACID des transactions

Dans un contexte centralisé, les contraintes ACID sont faciles à gérer. Mais dans les systèmes distribués comme les bases de données NoSQL, si on souhaite garder un certain niveau de performance, cela devient quasiment impossible. Certains SGBD NoSQL peuvent quand même être paramétrés pour gérer partiellement la durabilité, l'atomicité ou la cohérence mais cela peut faire décroître les performances. Quoi qu'il en soit, si la cohérence n'est pas gérée par le SGBD, ce sera au(x) client(s) de s'assurer du respect des contraintes d'intégrité.

1.3 Pas de schéma (schema-less)

La plupart des bases de données NoSQL sont sans schéma. Il n'y a pas une structure prédéfinie pour stocker les données. Ce type de paradigme convient assez bien aux processus de développement itératif où on ne connaît pas, à priori, le schéma de la base de données. Le fait de ne pas avoir de schéma permet également de ne pas devoir traiter les NULL (qui sont une source de problèmes dans les bases de données relationnelles).

1.4 Pas de normalisation

Comme les données sont partitionnées sur plusieurs nœuds, les SGBD NoSQL ne permettent pas de faire de jointures dans des bonnes conditions. C'est quelque chose qu'il faut prendre en compte lorsqu'on modélise les données d'une base de données NoSQL. Ainsi, la modélisation d'une base de données NoSQL dépendra toujours des requêtes qu'on souhaite réaliser ; afin que ces requêtes s'exécutent le plus rapidement possible. On va donc très souvent être amené à introduire de la redondance d'information. La logique est ici complètement différente des bases de données relationnelles où on modélise les données indépendamment des requêtes qui seront faites.

2 Les principales familles de bases de données NoSQL

On classe généralement les SGBD NoSQL dans quatre catégories (même si certains SGBD NoSQL plus marginaux peuvent avoir du mal à entrer dans une de ces catégories) :

1. Les bases de données clé-valeur (Redis, Riak, ...)

Ce sont les bases de données NoSQL les plus basiques. Chaque objet est identifié par une clé unique. La structure de l'objet est libre et laissée à la charge du développeur. Cela peut être une valeur scalaire ou bien un document plus complexe (JSON ou XML) ; mais le SGBD ne fournit aucun moyen d'effectuer des requêtes sur la structure de cet objet.

2. Les bases de données orientées documents (MongoDB, CouchDB, ...)

Les bases de données documentaires sont constituées de collections de documents. Il s'agit d'un raffinement du modèle clé-valeur où la valeur est obligatoirement un document (généralement JSON ou XML). Ces SGBD proposent un langage de requêtes sophistiqué pour extraire les données des documents de la collection.

3. Les bases de données orientées colonnes (Cassandra, HBase, ...)

Ces bases de données proposent des familles de colonnes. Pour une clé d'une famille de colonnes, correspond un ensemble de colonnes (chacune ayant une valeur). En théorie, chaque objet de la même famille peut avoir des colonnes différentes (schéma-less).

4. Les bases de données orientées graphes (Neo4j, GraphDB, ...)

Il s'agit des bases de données NoSQL les moins connues. Elles permettent de créer des objets qui sont reliés entre eux avec des arcs orientés disposant de propriétés.

3 Les principaux SGBD NoSQL

Le site DB-engine publie tous les mois un classement des SGBD par rapport à leur popularité (à savoir le nombre de fois où le SGBD est mentionné sur des sites Web, des forums, des moteurs de recherche, des offres d'emploi de réseaux professionnels, des réseaux sociaux, etc.)

Source : <https://db-engines.com/en/ranking>

416 systems in ranking, November 2023

Rank			DBMS	Database Model	Score		
Nov 2023	Oct 2023	Nov 2022			Nov 2023	Oct 2023	Nov 2022
1.	1.	1.	Oracle	Relational, Multi-model	1277.03	+15.61	+35.34
2.	2.	2.	MySQL	Relational, Multi-model	1115.24	-18.07	-90.30
3.	3.	3.	Microsoft SQL Server	Relational, Multi-model	911.42	+14.54	-1.09
4.	4.	4.	PostgreSQL	Relational, Multi-model	636.86	-1.96	+13.70
5.	5.	5.	MongoDB	Document, Multi-model	428.55	-2.87	-49.35
6.	6.	6.	Redis	Key-value, Multi-model	160.02	-2.95	-22.03
7.	7.	7.	Elasticsearch	Search engine, Multi-model	139.62	+2.48	-10.70
8.	8.	8.	IBM Db2	Relational, Multi-model	136.00	+1.13	-13.56
9.	9.	10.	SQLite	Relational	124.58	-0.56	-10.05
10.	10.	9.	Microsoft Access	Relational	124.49	+0.18	-10.53
11.	11.	12.	Snowflake	Relational	121.00	-2.24	+10.84
12.	12.	11.	Cassandra	Wide column, Multi-model	109.17	+0.34	-8.96
13.	13.	13.	MariaDB	Relational, Multi-model	102.09	+2.43	-2.82
14.	14.	14.	Splunk	Search engine	97.32	+4.95	+3.10
15.	15.	16.	Microsoft Azure SQL Database	Relational, Multi-model	83.17	+2.24	-0.49
16.	16.	15.	Amazon DynamoDB	Multi-model	82.24	+1.32	-3.16
17.	17.	19.	Databricks	Multi-model	77.22	+1.40	+16.33
18.	18.	17.	Hive	Relational	68.64	-0.54	-13.25
19.	20.	22.	Google BigQuery	Relational	59.31	+2.74	+5.18
20.	19.	18.	Teradata	Relational, Multi-model	57.33	-1.23	-7.90
21.	21.	21.	FileMaker	Relational	52.44	-0.88	-1.87
22.	23.	20.	Neo4j	Graph	49.70	+1.26	-7.60
23.	22.	23.	SAP HANA	Relational, Multi-model	49.12	-0.32	-2.33
24.	24.	24.	Solr	Search engine, Multi-model	44.62	-0.74	-6.71
25.	25.	25.	SAP Adaptive Server	Relational, Multi-model	41.49	-0.43	-2.10
26.	26.	26.	HBase	Wide column	34.16	-0.53	-6.25
27.	27.	27.	Microsoft Azure Cosmos DB	Multi-model	34.11	-0.18	-5.64
28.	28.	29.	InfluxDB	Time Series, Multi-model	29.02	-0.72	-0.94
29.	29.	28.	PostGIS	Spatial DBMS, Multi-model	26.92	-0.39	-3.86
30.	30.	34.	Microsoft Azure Synapse Analytics	Relational	26.79	-0.34	+3.76

Si les 10 premiers SGBD restent majoritairement des SGBD Relationnels, notamment les quatre premiers (Oracle, SQL Server, PostgreSQL, MySQL) qui maintiennent leur rang depuis très longtemps, certains SGBD non-relationnels commencent à acquérir une certaine popularité, notamment MongoDB, Redis et Cassandra qui sont régulièrement classés dans les dix premières places.

Nous verrons cette année trois SGBD NoSQL :

- **Cassandra** qui est le SGBD orienté colonnes le plus populaire.
- **MongoDB** qui est un SGBD orienté documents et qui est le SGBD NoSQL le plus populaire.
- **Neo4j** qui bien qu'il soit moins populaire que les SGBD précédents reste malgré tout le SGBD orienté graphes le plus connu.

Nous ne verrons pas de SGBD clé-valeur qui sont plus rudimentaires et dont le fonctionnement ressemble un peu aux HSTORE que vous avez vu sous PostgreSQL en deuxième année. Et nous ne verrons pas non plus de Bases de Données de type moteur de recherche tel que Elasticsearch ou Solr.

Dossier 3

Cassandra

Cassandra est un SGBD orienté colonne écrit en Java. A l'origine, il a été développé par Facebook mais il est dans l'espace open-source depuis 2008. Cassandra est actuellement portée par la fondation Apache et est utilisé par des entreprises qui gèrent des grands volumes de données - comme par exemple Tweeter, eBay ou Netflix. L'entreprise DataStax propose la distribution et le support d'une version entreprise de Cassandra.

Cassandra permet de partitionner les données sur différents nœuds avec une architecture décentralisée. Chaque nœud est indépendant et il n'y a pas besoin de serveur maître. Le client peut interroger n'importe quel nœud. Il sera dirigé vers un des nœuds qui contient les données qu'il souhaite obtenir. Afin d'assurer la disponibilité des données (même si un des nœuds est indisponible), Cassandra duplique les données. La cohérence des écritures et des lectures est alors paramétrable pour chaque instruction. Par exemple, avec le niveau par défaut `ONE`, les écritures ou lectures seront réalisées sur un nœud (il n'y a pas de cohérence absolue). Avec l'option `QUORUM`, elles seront réalisées sur la moitié des nœuds plus un. Et avec l'option `ALL`, elles devront être réalisées sur tous les nœuds. Depuis sa version 2, Cassandra dissimule le stockage interne des données (historiquement orienté colonnes) et propose un langage d'interrogation, le CQL (Cassandra Query Language), qui est très proche dans sa syntaxe du SQL ; si bien que le développeur peut avoir l'impression de manipuler une base de données relationnelle. Mais comme nous le verrons par la suite, cela n'est qu'une illusion.

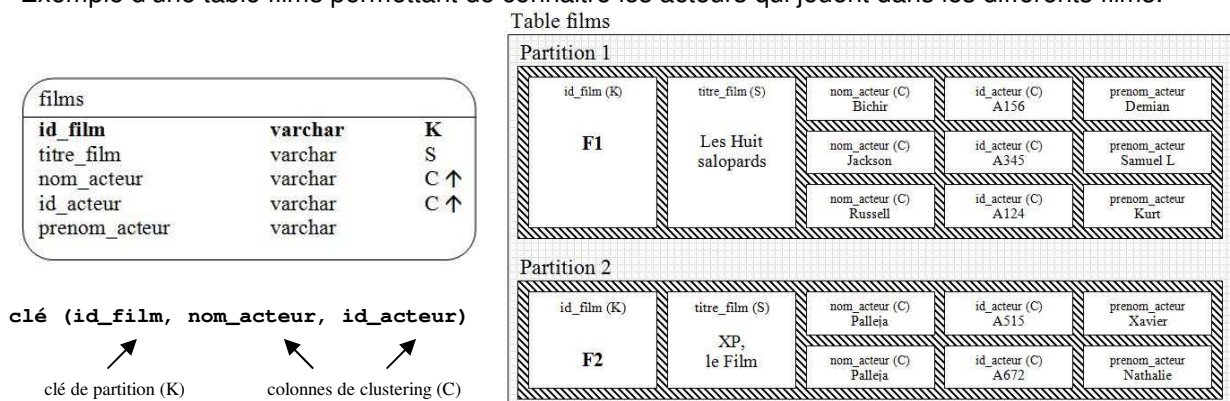
1 Concepts de base

1.1 Stockage des données

Cassandra était à l'origine un SGBD orienté colonnes. Mais depuis sa version 2, il n'est plus possible de stocker des colonnes différentes pour chaque objet d'une `column family`. A présent, chaque ligne de la `column family` doit avoir une même structure, définie à l'avance. Cassandra n'est donc plus un SGBD schéma-less. D'ailleurs, dans les dernières versions, les `column family` peuvent être appelées des `table`. Et avec l'ajout du langage CQL, cela peut donner l'illusion que l'on travaille sur une base de données relationnelle. Mais, en réalité, il n'en est rien car les données d'une table sont partitionnées sur différents nœuds.

Les tables (ou `column family` dans les précédentes versions de Cassandra), possèdent une clé composée par une **clé de partition** (qui assure que toutes les données qui ont la même clé de partition sont stockées ensemble sur le même nœud – partition ou fragmentation horizontale) et éventuellement d'une ou plusieurs **colonnes de clustering** ordonnées qui permettent de classer les données d'une partition dans un ordre précis (par défaut, les données sont classées dans l'ordre ascendant de la colonne de clustering). Les colonnes d'une table peuvent être déclarées `static` et toutes les lignes de la même partition partagent la même valeur. Cela permet de limiter la redondance, mais il faut savoir que sous Cassandra 3, les colonnes statiques ne peuvent pas être utilisées pour faire des vues matérialisées.

Exemple d'une table films permettant de connaître les acteurs qui jouent dans les différents films.

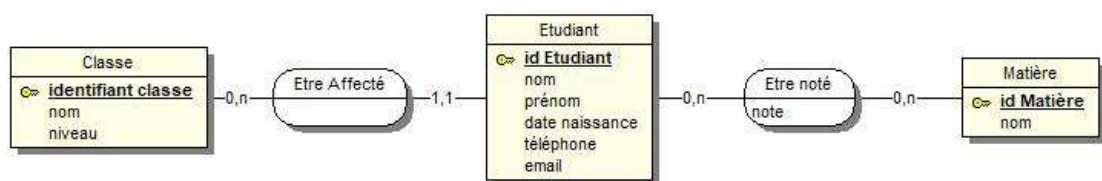


1.2 Modélisation des données

Comme nous le verrons plus tard, le CQL ne permet pas de faire des jointures ni de sous-requête. Cela ralentirait trop les requêtes, notamment parce qu'avec les partitions, les données d'une table peuvent être distribuées sur plusieurs nœuds. Le CQL ne permet pas non plus de faire des sélections sur des colonnes qui ne se trouvent pas dans la clé (à moins d'utiliser des index secondaires qui ne sont pas très efficaces) car dans les tables, les données ne sont pas triées selon cette colonne. Dans ces conditions, il est important de structurer les données en fonction des requêtes qu'on souhaite réaliser ; afin que ces requêtes puissent s'exécuter le plus rapidement possible. Même si pour cela on est amené à introduire de la redondance.

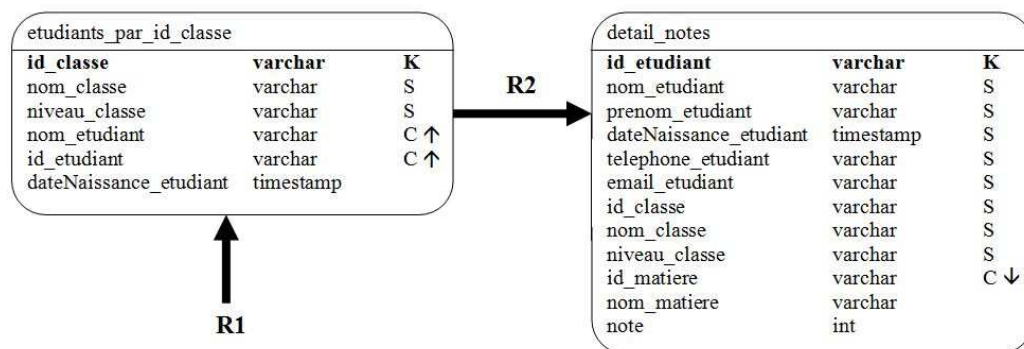
Il s'agit d'une logique de modélisation des données complètement différente de celle vue dans les bases de données relationnelles, où on structure les données indépendamment des requêtes, en ayant pour seul souci d'obtenir un schéma normalisé. Ici, on va plutôt concevoir le schéma en fonction des requêtes que l'on va devoir réaliser.

Exemple : on souhaite modéliser dans une base de données Cassandra les informations du diagramme de classes conceptuel suivant.



Les utilisateurs du système souhaitent pouvoir faire deux requêtes :

- R1 : retrouver les étudiants d'une classe à partir de l'identifiant d'une classe. Les étudiants de la classe doivent être classés par ordre alphabétique de leur nom (ordre ascendant). Si deux étudiants ont le même nom, on veut les classer par leur identifiant. On souhaite également avoir la date de naissance des étudiants ainsi que le nom et le niveau de la classe.
- R2 : à partir des identifiants d'étudiants obtenus avec la R1, retrouver toutes les informations sur les étudiants en question. Avec leur classe, ainsi que les notes qu'ils ont obtenues. Les notes doivent être classées par rapport à leur numéro de matière, dans l'ordre descendant.



```

CREATE TABLE etudiants_par_id_classe (
  id_classe varchar,
  nom_classe varchar STATIC,
  niveau_classe varchar STATIC,
  id_etudiant varchar,
  nom_etudiant varchar,
  dateNaissance_etudiant timestamp,
  PRIMARY KEY (id_classe, nom_etudiant, id_etudiant)
);

```

Modèle Logique
des Données

```

CREATE TABLE detail_notes (
  id_etudiant varchar,
  nom_etudiant varchar STATIC,
  prenom_etudiant varchar STATIC,
  dateNaissance_etudiant timestamp STATIC,
  telephone_etudiant varchar STATIC,
  email_etudiant varchar STATIC,
  id_classe varchar STATIC,
  nom_classe varchar STATIC,
  niveau_classe varchar STATIC,
  id_matiere varchar,
  nom_matiere varchar,
  note int,
  PRIMARY KEY (id_etudiant, id_matiere)
)
WITH CLUSTERING ORDER BY (id_matiere DESC);

```

1.3 Réaliser des requêtes en CQL

Pour manipuler les données, il est possible d'utiliser le langage CQL (Cassandra Query Language). Syntaxiquement, ce langage ressemble beaucoup au SQL. Il permet de créer, de modifier ou de supprimer des tables. Il permet également d'insérer, de modifier ou de supprimer des lignes dans les tables. Enfin il permet aussi d'extraire des données (requêtes SELECT) mais pour ce genre de requêtes, il est beaucoup plus limité que le SQL.

1.3.1 Créer et se connecter à un keyspace

Dans une base de données Cassandra les tables et tous les autres objets (comme par exemple les types, les fonctions ou les vues matérialisées) sont stockées dans un keyspace. Un keyspace est l'équivalent d'un schéma d'une base de données relationnelle.

Pour utiliser un keyspace et pouvoir ainsi manipuler directement tous les objets qui se trouvent dans le keyspace, il faut exécuter l'instruction `USE`.

Par exemple : `USE ks_etudiants;` permettra par la suite d'écrire directement `SELECT * FROM etudiants_par_id_classe` au lieu de `SELECT * FROM ks_etudiants.etudiants_par_id_classe`

Si on possède les privilèges système suffisants, il est possible de créer un nouveau keyspace avec la commande suivante.

```
CREATE KEYSPACE ks_etudiants
```

```
WITH REPLICATION = {'class' : 'SimpleStrategy', 'replication_factor' : 1};
```

`class` : permet d'indiquer que la stratégie de placement est simple, gérée automatiquement par Cassandra (l'autre stratégie est `NetworkTopologyStrategy`)

`replication_factor` : indique le nombre de réplicas à maintenir.

1.3.2 Créer modifier ou supprimer des tables

Comme en SQL, il est possible de créer une table avec l'instruction `CREATE TABLE` (voir exemple en bas de la page précédente). Il est également possible de modifier une table avec l'instruction `ALTER TABLE` ou de supprimer une table avec l'instruction `DROP TABLE`.

1.3.3 Les requêtes de mise à jour

Il est possible de mettre à jour les données avec les instructions `INSERT`, `UPDATE` et `DELETE`. La clause `TTL` (Time To Live) permet de faire des modifications temporaires, limitées en secondes. Cassandra vérifie quelques contraintes (clé primaire, types), mais la plupart des contrôles de la cohérence devront être réalisés par l'application cliente (ou par chacune des applications clientes) qui se connecte à la base de données.

Lors d'une insertion, la clé de partition doit obligatoirement être renseignée mais pas forcément les colonnes de clustering qui peuvent éventuellement être toutes nulles (mais si une des colonnes de clustering ou non statique est renseignée, alors toutes les colonnes de clustering doivent l'être aussi). Si on insère une ligne dont la clé (clé de partition + colonne de clustering) est déjà présente dans la table, il n'y aura pas d'erreur mais la ligne de la table sera écrasée par l'insertion (cela fonctionnera comme un `UPDATE`)

etudiants_par_id_classe		
id_classe	varchar	K
nom_classe	varchar	S
niveau_classe	varchar	S
nom_etudiant	varchar	C ↑
id_etudiant	varchar	C ↑
dateNaissance_etudiant	timestamp	

- a ➤ `INSERT INTO etudiants_par_id_classe (id_classe, nom_classe, niveau_classe, id_etudiant, dateNaissance_etudiant, nom_etudiant) VALUES ('C1','LP APIDAE', 'Licence Pro', 'E1', '1999-12-16', 'Zetofrais');`
- b ➤ `INSERT INTO etudiants_par_id_classe (id_classe, nom_classe, niveau_classe, id_etudiant, nom_etudiant) VALUES ('C2','LP ACPI', 'Licence Pro', 'E2', 'Zeblouse');`
- c ➤ `INSERT INTO etudiants_par_id_classe (id_classe, nom_classe, niveau_classe) VALUES ('C3','LP PGI', 'Licence Pro');`
- d ➤ `UPDATE etudiants_par_id_classe SET dateNaissance_etudiant = '1999-02-21' WHERE id_classe = 'C2' AND nom_etudiant = 'Zeblouse' AND id_etudiant = 'E2';`
- e ➤ `UPDATE etudiants_par_id_classe USING TTL 60 SET nom_classe = 'LP PSGI' WHERE id_classe='C3';`
- f ➤ `DELETE FROM etudiants_par_id_classe WHERE id_classe = 'C3';`

etudiants_par_id_classe		
id_classe	varchar	K
nom_classe	varchar	S
niveau_classe	varchar	S
nom_etudiant	varchar	C ↑
id_etudiant	varchar	C ↑
dateNaissance_etudiant	timestamp	

1.3.4 Les requêtes de base d'extraction de données

Le CQL ne permet pas de faire de jointure. Sans index secondaire, on ne peut faire des sélections que sur des colonnes définies dans la clé primaire ; et il faut que la condition de sélection de la requête respecte l'ordre dans lesquelles les colonnes ont été définies dans la clé primaire. Il n'est donc pas possible de faire une sélection (dans le WHERE) sur une colonne de clustering si la clé de partition n'est pas dans le WHERE. De même, il n'est pas possible de mettre dans le WHERE la seconde colonne de clustering si la première n'y est pas.

Les DISTINCT sont soumis aux mêmes contraintes. Les tris (avec ORDER BY) ne peuvent porter que sur des colonnes de clustering, et il faut qu'il y ait au moins la clé de partition dans le WHERE (un ORDER BY ne peut pas porter sur une clé de partition).

Les requêtes suivantes **retourneront un résultat** :

- g ➤ `SELECT nom_etudiant FROM etudiants_par_id_classe
WHERE id_classe = 'C1' LIMIT 10;`
- h ➤ `SELECT COUNT(id_etudiant) FROM etudiants_par_id_classe
WHERE id_classe = 'C2' AND nom_etudiant = 'Terrieur';`
- i ➤ `SELECT COUNT(id_etudiant) FROM etudiants_par_id_classe
WHERE id_classe = 'C2'
AND nom_etudiant = 'Terrieur' AND id_etudiant = 'E5';`
- j ➤ `SELECT * FROM etudiants_par_id_classe
WHERE id_classe IN ('C1', 'C2');`
- k ➤ `SELECT DISTINCT id_classe FROM etudiants_par_id_classe;`
- l ➤ `SELECT * FROM etudiants_par_id_classe
WHERE id_classe = 'C1' ORDER BY nom_etudiant;`

Par contre, les requêtes suivantes **ne marcheront pas** (ou déclencheront des warnings). En fonction de la version de Cassandra, il sera tout de même possible d'exécuter certaines requêtes en ajoutant la clause ALLOW FILTERING mais les performances seront dégradées.

- m ❖ `SELECT * FROM etudiants_par_id_classe
WHERE id_etudiant = 'E1';`
-- il n'y a pas la clé de partition dans le WHERE
- n ❖ `SELECT * FROM etudiants_par_id_classe
WHERE nom_etudiant = 'Terrieur';`
-- Même chose mais comme nom_etudiant est la première colonne de clustering, on peut tout
-- de même forcer l'exécution de cette requête avec la clause ALLOW FILTERING
- o ❖ `SELECT * FROM etudiants_par_id_classe
WHERE id_classe = 'C1'
AND id_etudiant = 'E1';`
-- sélection sur la seconde colonne de clustering alors qu'il n'y a pas la première dans le WHERE
- p ❖ `SELECT * FROM etudiants_par_id_classe
ORDER BY nom_etudiant;`
-- tri sur une colonne de clustering alors qu'il n'y a pas la clé de partition dans le WHERE
- q ❖ `SELECT * FROM etudiants_par_id_classe
WHERE id_classe = 'C1'
ORDER BY id_etudiant;`
-- tri sur la seconde colonne de clustering alors qu'il n'y a pas la première dans le ORDER BY
- r ❖ `SELECT DISTINCT id_etudiant FROM etudiants_par_id_classe;`
-- DISTINCT n'est possible que sur la colonne de clustering ou colonnes statiques
- s ❖ `SELECT COUNT(id_etudiant) FROM etudiants_par_id_classe;`
-- COUNT sur une colonne de clustering, va fonctionner mais peut lever des warning sous
-- certaines versions de Cassandra (mais pas sur la dernière version que nous avons à l'IUT)

1.3.5 Solutions pour extraire des données selon des colonnes non référencées par la clé primaire

Lorsqu'on souhaite réaliser une requête sur une table mais que le CQL ne le permet pas car les critères de sélection de la requête ne correspondent pas aux clés primaires définies sur la table, ou alors lorsque on veut faire une requête qui nécessiterait de faire une jointure (ce qui est impossible en CQL), on a quatre solutions à notre disposition pour arriver à nos fins : utiliser un index secondaire, créer une nouvelle table de données, créer une table d'index ou encore utiliser une vue matérialisée.

La première solution (index secondaire) est la plus facile à mettre en œuvre mais elle dégrade les performances. La dernière (la vue matérialisée) est la solution généralement préconisée.

1. Utiliser des index secondaires. Cette solution est très simple à mettre en œuvre, mais elle est généralement déconseillée car comme l'utilisation `ALLOW FILTERING` les index secondaires dégradent les performances. En effet ces index sont beaucoup moins rapides que les index définis sur les clés primaires ; notamment lorsque la requête va porter sur plusieurs partitions qui peuvent être sur des nœuds différents. Il est à noter que la colonne sur laquelle on place l'index secondaire ne doit pas être statique.
2. Modifier la structure de la table existante (si cela ne nuit pas à d'autres requêtes) ou plutôt rajouter une nouvelle table de données. Cette solution, a le désavantage de générer de la redondance. Elle n'est intéressante que si les données sur lesquelles on travaille sont stables (car les données de la nouvelle table ne seront pas mises à jour quand on modifiera les données de la première table).
3. Rajouter une table d'index ; c'est-à-dire une table composée uniquement des attributs de sa clé primaire, qui permettra grâce à une première requête d'accéder ensuite à la table dans laquelle se trouvent les données cherchées (on fera donc deux requêtes). Par rapport à la solution précédente, cette solution permet de limiter la redondance ; même s'il y en a tout de même. Par contre il faudra faire deux requêtes. Cette solution est intéressante lorsqu'on souhaite faire des recherches multicritères (on fait alors une table d'index pour chaque critère de recherche).
4. Enfin, la solution conseillée, créer une vue matérialisée construite avec les données de la table sur laquelle on souhaite faire la recherche. Cette solution n'est disponible qu'à partir de Cassandra 3. Les vues matérialisées permettent de ne pas avoir à gérer les anomalies dues à la redondance ; la vue matérialisée est mise à jour automatiquement lorsqu'il y a des modifications sur la table qu'elle référence. Il est à noter que les vues matérialisées ne peuvent pas être implémentées sur des colonnes qui ont été déclarées `static` dans la table référencée par la vue. La vue matérialisée doit également comporter toutes les colonnes qui composent la clé de la table qu'elle référence.

Exemple : on souhaite pouvoir retrouver l'identifiant de tous les étudiants qui sont à un niveau de classe donné (par exemple tous les étudiants qui sont en Licence Pro).

etudiants_par_id_classe		
id_classe	varchar	K
nom_classe	varchar	
niveau_classe	varchar	
nom_etudiant	varchar	C ↑
id_etudiant	varchar	C ↑
dateNaissance_etudiant	timestamp	

Pour réaliser cette requête, il n'est pas possible d'utiliser la table `etudiants_par_id_classe` car `niveau_classe` n'est pas la clé de partition (il n'est d'ailleurs même pas une colonne de clustering). La requête suivante ne marchera donc pas.

```
SELECT id_etudiant FROM etudiants_par_id_classe WHERE niveau_classe = 'Licence Pro';
```

Pour résoudre ce problème, nous pouvons donc :

① Utilisation d'un index secondaire :

```
CREATE INDEX idx_etudiants_par_id_classe_niveau_classe
ON etudiants_par_id_classe(niveau_classe);
```

② Ajout d'une nouvelle table de données :

```
CREATE TABLE etudiant_par_niveau_classe (
  id_classe varchar,
  nom_classe varchar,
  niveau_classe varchar,
  id_etudiant varchar,
  dateNaissance_etudiant timestamp,
  nom_etudiant varchar,
  PRIMARY KEY (niveau_classe, nom_etudiant, id_etudiant)
);
```

③ Ajout d'une table d'index

```
CREATE TABLE etudiants_par_niveau_classe (
  niveau_classe varchar,
  id_classe varchar,
  PRIMARY KEY (niveau_classe, id_classe)
);
```

④ Ajout d'une vue matérialisée :

```
CREATE MATERIALIZED VIEW etudiants_par_niveau_classe AS
SELECT niveau_classe, id_classe, nom_etudiant, id_etudiant
FROM etudiants_par_id_classe
WHERE niveau_classe IS NOT NULL
AND id_classe IS NOT NULL
AND nom_etudiant IS NOT NULL
AND id_etudiant IS NOT NULL
PRIMARY KEY (niveau_classe, id_classe, nom_etudiant, id_etudiant);

SELECT * FROM etudiants_par_niveau_classe WHERE niveau_classe = 'Licence Pro';
```