Simulation d'algorithmes d'équilibrage de charge dans un environnement distribué

Architecture

Kevin Barreau Guillaume Marques Corentin Salingue

31 mars 2015

Sommaire

1 Architecture de Cassandra **3**

1.1 Staged event-driven architecture (SEDA) 3

1.2 Gossip 4

1.3 Modèle de données 4

1.4 Requête de lecture 4

1.5 Statistiques 5

2 Client **6**

2.1 Architecture du pilote 6

2.1.1 Objet Cluster 6

2.1.2 Interface Session 6

2.1.3 Objet BoundStatement 7

2.2 Architecture du client 7

2.2.1 Diagramme de classe 7

2.2.2 Objet Connection 8

2.2.3 Objet KeySpace 8

2.2.4 Objet QueriesFactory 8

2.2.5 Objet ClientApp 8

3 Visualisation des métriques **9**

3.1 Présentation de Graphite 9

3.2 Enregistrement des metrics 9

1 Architecture de Cassandra

Cassandra est une base de données distribuées, écrite en langage Java. Dans sa version 2.1.2, elle est composée de 963 fichiers, répartis dans 62 dossiers, pour un total de 126 502 lignes de codes et 36 287 lignes de commentaires.

Cassandra est un projet riche et complet. Nous ne nous intéresserons qu'aux parties de son architecture sur lesquelles nous allons travailler.

Par ailleurs, nous nous intéresserons peu à la représentation sous forme de diagramme de classes de Cassandra, étant donné une forte utilisation du pattern *singleton* et des méthodes *statiques*. Cette représentation donne peu d'information sur le fonctionnement de la base de données.

1.1 Staged event-driven architecture (SEDA)

Cassandra est basée sur une architecture de type Staged Event Driven Architecture (SEDA). Cela permet de séparer des tâches dans différents emplacements, appelés *stages*, qui sont connectés par un service de messages. Chaque stage possède une file d'attente pour les messages (un message correspondant à une tâche à traiter), ainsi qu'un ensemble de threads pour traiter les tâches (voir figure 4).

Dans le cas de Cassandra, la gestion des stages se fait dans le package `org.apache.cassandra.concurrent`. Les stages sont -pour la plupart- énumérés dans `Stage`. Ils sont ensuite gérés par le `StageManager` (qui a pour particularité de ne posséder que des méthodes et des attributs statiques). Nous nous intéresserons principalement aux stages :

- **READ** : lectures locales
- **GOSSIP** : communications sur l'état des noeuds

Ces stages permettent de répondre à différents besoins.

Le stage "READ" permet de répondre aux besoins des protocoles d'affectation. L'idée est de créer un nouveau stage "READ_REMOVE" pour gérer les messages de suppression de requête d'une file d'attente. Ainsi, les tâches traitées par le stage "READ" entraînent l'envoi d'un message pour supprimer un message d'une file d'attente dans les autres noeuds ayant à traiter la même tâche. Les noeuds recevant le message le transmettent au stage "READ_REMOVE" qui s'occupe alors de supprimer le message voulu de la file d'attente s'il y est encore présent. La figure 7 montre un exemple avec une requête de lecture arrivant sur le noeud 5, et avec les données à lire sur les noeuds 1 et 2.

Le stage "GOSSIP" permet de répondre aux besoins qui concernent la communication des données locales d'un noeud. Les noeuds de la base de données s'échangent des informations sur leur état toutes les secondes. L'ajout de données locales entraîne la modification du stage "GOSSIP" pour permettre l'envoi de ces nouvelles données.

Une architecture fortement simplifiée de la gestion des stages est disponible sur la figure 5. La création et l'accès d'un stage se fait par le biais du `StageManager` et de `Stage`, avec un appel sous la forme : `StageManager.getStage(Stage.READ)`.

1.2 Gossip

Cassandra utilise le protocole Gossip pour les communications entres les noeuds. Chaque noeud envoie les informations qu'il possède -sur lui et sur les autres noeuds- à au plus 3 autres noeuds du réseau. Cela permet d'avoir pour chaque noeud une connaissance globale du réseau avec un minimum d'interaction.

Les classes en rapport avec Gossip se situe dans le package `org.apache.cassandra.gms`. La classe chargée de traiter les tâches de Gossip est `Gossiper`.

`Gossiper` maintient une liste de noeuds "vivants" et "morts" (des noeuds inatteignables). Toutes les secondes, le module démarre un tour. Un tour entier de Gossip est composé de trois messages. Un noeud X envoie un message syn à un noeud Y pour initialiser Gossip. Y, à la réception de ce message syn, renvoie un message ack à X. Pour répondre à ce message ack, X envoie un message ack2 à Y pour compléter le tour (voir la figure 8).

1.3 Modèle de données

Le modèle de données de Cassandra s'appuie sur un schéma dynamique, avec un modèle de données orienté colonne (voir figure 9). On retrouve les classes gérant la modélisation de la base de données au sein du package `org.apache.cassandra.db`.

- **Keyspace** : le conteneur des données de l'application
- **Row** : une ligne dans le Keyspace, composée d'une clé et d'un ensemble de colonnes
- **DecoratedKey** : un token identifiant le positionnement d'une ligne dans la base de données
- **ColumnFamily** : un ensemble de colonnes pour une ligne donnée
- **Column** : un tuple contenant un nom, une valeur et un *timestamp* (la date de la mise à jour la plus récente de cette colonne)

Nous allons nous intéresser dans notre projet à la classe `ColumnFamily`, afin de pouvoir garder une trace de la popularité des objets. Les objets étant définis par un token, porté par la classe `DecoratedKey`, `ColumnFamily` est l'objet dont nous souhaitons connaître la popularité.

1.4 Requête de lecture

Les requêtes de lecture suivent un chemin comme décrit dans la figure 7, avec deux possibilités au niveau de la classe `AbstractReadExecutor` pour gérer le traitement de la requête. On récupère tous les noeuds qui possèdent une copie de l'objet que l'on cherche. Si le noeud actuel possède une copie, alors on ne passe par le système de messagerie et on demande directement au stage de lecture de réaliser une tâche (un Runnable qui s'appelle `LoadReadRunnable`, une classe statique interne du `StorageProxy`. Sinon, on envoie une requête de lecture aux autres noeuds responsables d'une copie de l'objet requêté.

1.5 Statistiques

Cassandra possède un système permettant d'exposer des mesures internes, grâce à la libraire **Metrics** [CH10].

Les classes s'occupant de collecter les mesures se situent dans le package `org.apache.cassandra.metrics`. Chaque classe de Metrics est chargé de collecter des statistiques sur certaines classes de Cassandra, comme par exemple sur la classe `Keyspace` vue précédemment dont les données sont collectées par la classe `KeyspaceMetrics`.

L'intérêt dans le cas de notre projet est de pouvoir exposer les mesures initiales de Cassandra, mais aussi d'ajouter des mesures sur les implémentations que nous faisons. Il nous est possible d'ajouter une nouvelle classe de Metrics pour répondre à ce besoin, ou modifier les classes existantes.

Une classe de Metrics est liée à une classe que l'on souhaite de manière bidirectionnelle. C'est à dire que la classe que l'on souhaite observée possède en attribut une instance de la classe de Metrics, à laquelle elle passe en paramètre sa propre référence. La référence n'est pas stockée par la classe de Metrics mais seulement utilisée dans des *closures*.

2 Client

Le client est une application qui permet à l'utilisateur d'intéragir avec Cassandra. Elle permet de choisir la stratégie de réplication et d'envoyer des requêtes sur une base de données distribuées Cassandra.

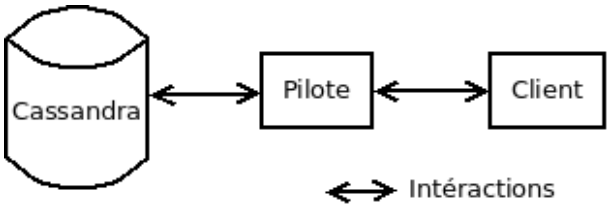

FIGURE 1 – Intéractions client/pilote/Cassandra

Nous utilisons un *pilote*, programme informatique permettant à un programme d'intéragir avec un autre programme, que nous présentons dans la partie suivante. Il permet notamment de se connecter à Cassandra, d'envoyer des requêtes et de récupérer les résultats de ces requêtes.

Le client initialise la connexion, c'est-à-dire récupère les informations nécessaires à la connexion (ip de l'host par exemple), importe les données, prépare les requêtes.

2.1 Architecture du pilote

La communication entre le client et Cassandra se faît grâce à un pilote développé par Datastax, DataStax Java Driver for Apache Cassandra, écrit en Java. Nous nous interessons uniquement aux parties du pilote permettant d'implémenter les fonctionnalités du client, c'est à dire les objets permettant de se connecter et d'envoyer des requêtes sur une base de données Cassandra.

Nous ne présentons pas les objets stockant les résultats des requêtes, comme `ResultSet` ou `Row`, qui ne sont que des structures de données.

2.1.1 Objet Cluster

Package `com.datastax.driver.core.Cluster`

Description L'objet **Cluster** permet de se connecter à une *grappe de serveurs*, ensemble de serveurs appelés *noeuds* qui sont reliés entre eux (par exemple par Internet) et qui communiquent ensemble. Plus précisément, l'objet se connecte à un noeud de la grappe de serveurs et grâce à cette connexion, il peut connaître l'ensemble des noeuds de la grappe.

2.1.2 Interface Session

Package `com.datastax.driver.core.Session`

Description Les objets implémentant l'interface **Session** créent une *session*, connexion entre un programme et Cassandra permettant au programme d'exécuter des requêtes sur Cassandra et de récuperer le résultat de ses requêtes.

Pour exécuter une requête, les objets disposent d'une méthode `execute`.

2.1.3 Objet BoundStatement

Package `com.datastax.driver.core.BoundStatement`

Avec `CQL`, le langage dans lequel sont écrites les requêtes de Cassandra, il est possible de *préparer des requêtes*. Ce sont des requêtes incomplètes dont seules les clauses (`SELECT`, `FROM`, ...) sont définies.

Exemple `SELECT` ? `FROM` ? `WHERE` ? `=` ? est une requête préparée.

Description L'objet `BoundStatement` permet de compléter des requêtes préparées afin de les rendre exécutables.

2.2 Architecture du client

2.2.1 Diagramme de classe

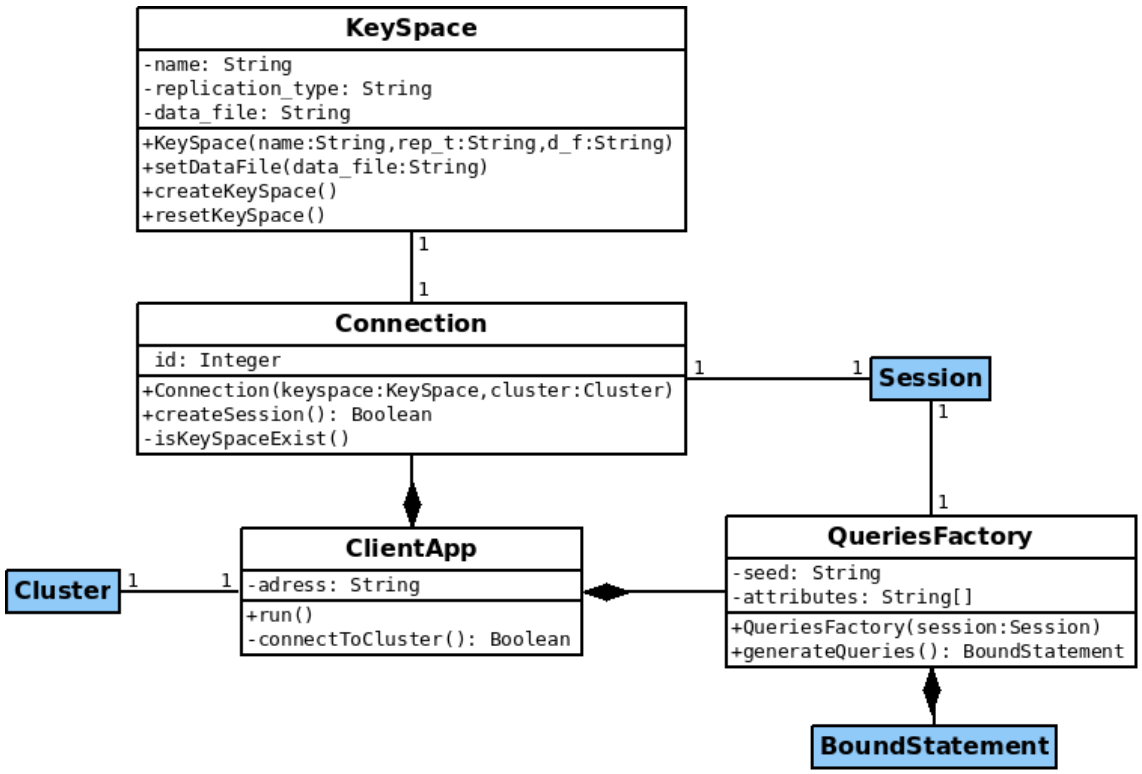

FIGURE 2 – Diagramme de classe du client. Avec un fond bleu, les classes du pilote

2.2.2 Objet Connection

Un *keyspace* est un objet contenant un ensemble de données. C'est l'équivalent d'un *schéma* dans une base de données relationnelle.

Cet objet permet de créer une session pour exécuter des requêtes dans un keyspace donné. Si le keyspace n'existe pas, l'objet `KeySpace` le crée.

2.2.3 Objet KeySpace

L'objet `KeySpace` permet de créer un keyspace et d'initialiser ses données en fonction d'un jeu de données contenu dans un fichier `data_file` écrit au format `CQL`. Si l'utilisateur souhaite importer un autre jeu de données, il suffit de réinitialiser le keyspace avec la méthode `resetKeySpace`.

Attributs

- `name` : nom du keyspace
- `replication_type` : choix de l'algorithme d'équilibrage des charges.
- `data_file` : fichier au format `CQL` contenant un jeu de données, suite de requête permettant d'importer des données dans Cassandra.

2.2.4 Objet QueriesFactory

Cet objet permet de génerer ou d'importer des requêtes `CQL` que nous exécutons sur une session donnée.

L'attribut `seed` permet la génération de requêtes de manière pseudo-aléatoire. Si on change le seed, la suite de requêtes générée sera différente.

2.2.5 Objet ClientApp

C'est le client, la méthode `run` permet de l'exécuter.

3 Visualisation des métriques

Pour visualiser les métriques, nous utilisons l'application nommée `Graphite` écrite en Python.

3.1 Présentation de Graphite

Graphite est l'union de trois logiciels :
- **carbon**, un logiciel permettant de traiter des données temporelles, comme l'évolution de la charge d'un noeud en fonction du temps.
- **whisper**, une base de données stockant toutes ces données
- **graphite webapp**, une application web permettant la visualisation de ces données.

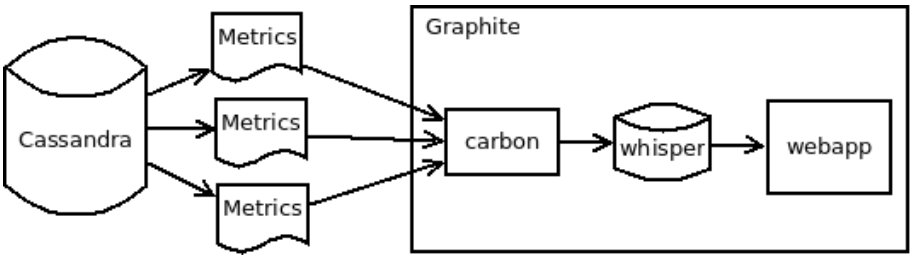

FIGURE 3 – Graphite, union de trois logiciels

Cassandra enregistre ses metrics dans des fichiers. Carbon lit ces fichiers et stocke les données dans whisper. Enfin, graphite webapp traite ces données pour les afficher dans une page web.

3.2 Enregistrement des metrics

Les métriques sont sauvegardées dans des fichiers traités par `Cyanite`. Puis, `Graphite` affiche ces informations sous forme de graphes (courbes, histogrammes...) dans un navigateur web.

Références

[CH10] Yammer.com Coda Hale. Metrics. `https://github.com/dropwizard/metrics`, 2010. [Accessed 15 February 2015].

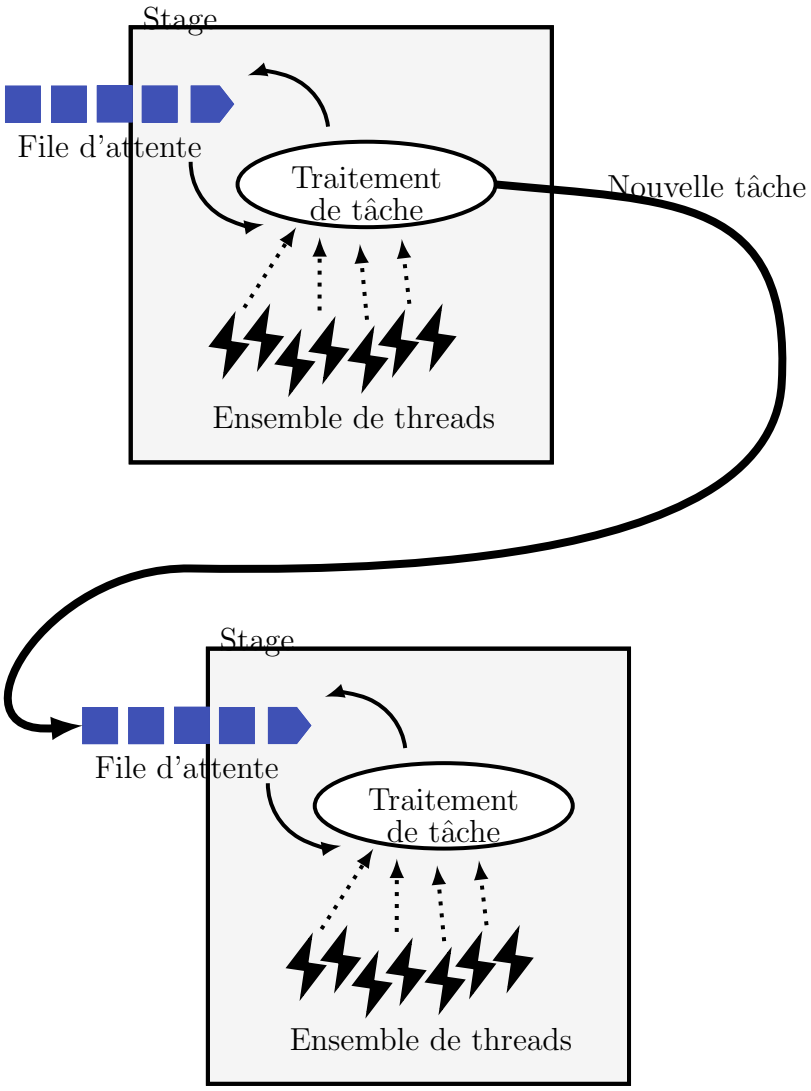

FIGURE 4 – Schéma de deux stages dans une architecture SEDA

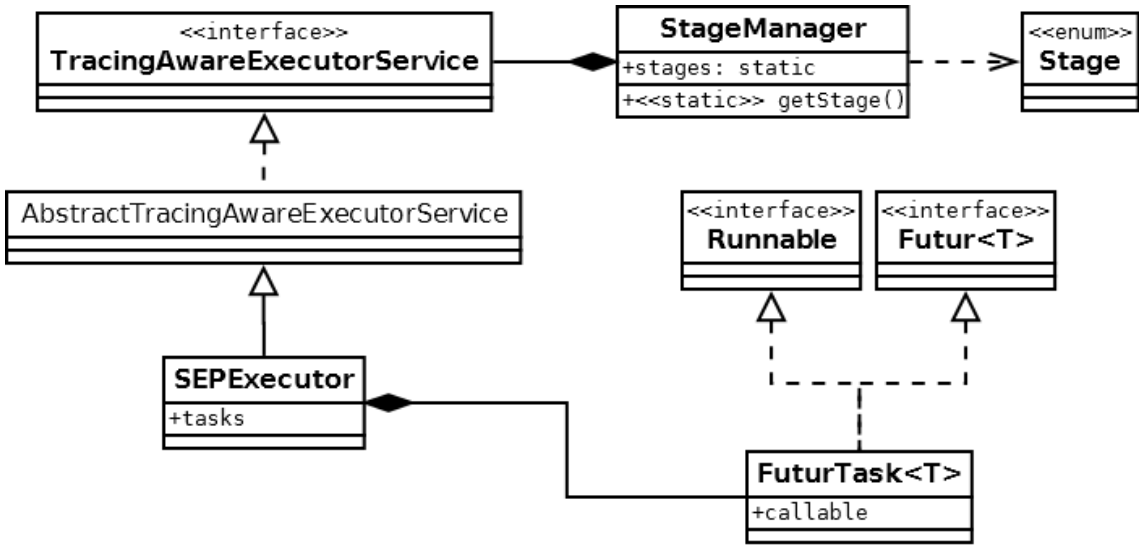

FIGURE 5 – Diagramme de classe simplifié de l'architecture SEDA dans Cassandra

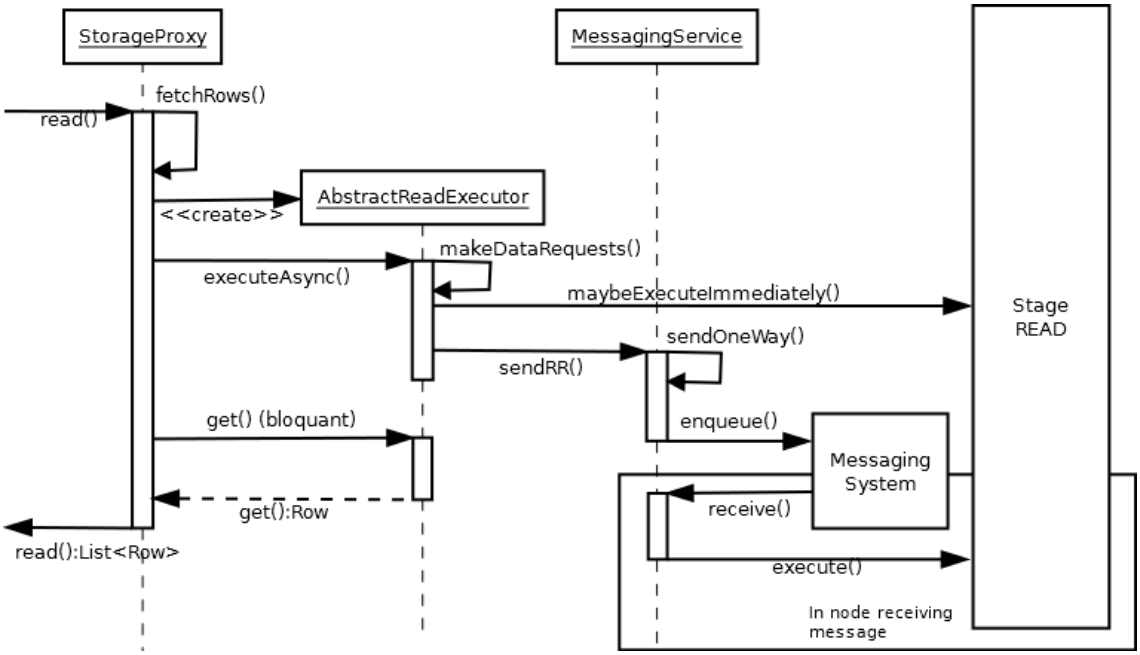

FIGURE 6 – Diagramme de séquence simplifié d'une requête de lecture dans Cassandra

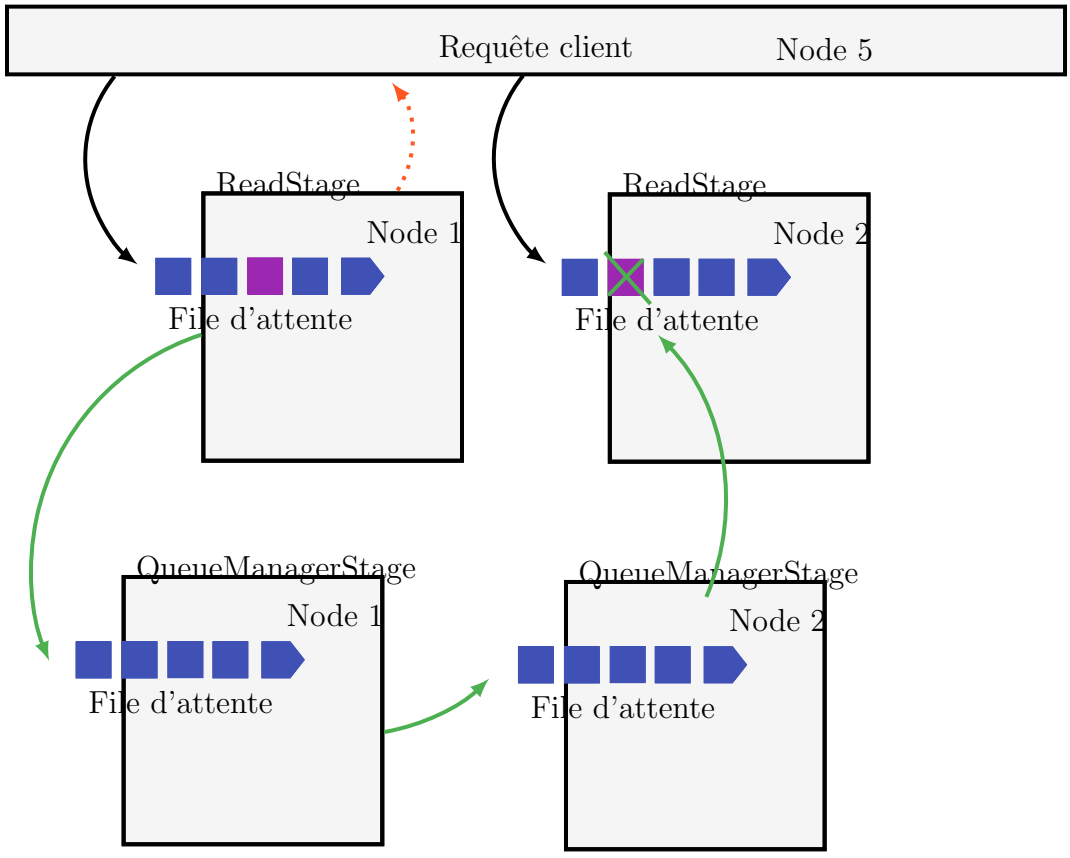

FIGURE 7 – Exemple simplifié d'une requête de lecture au niveau des stages, avec l'ajout d'un stage "QUEUE_MANAGER"

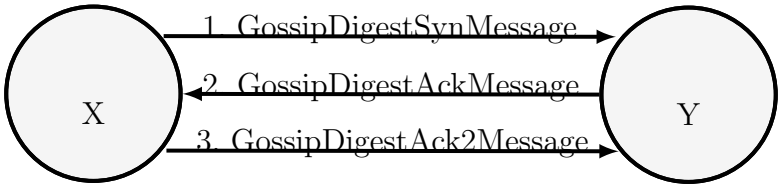

FIGURE 8 – Tour de messages Gossip entre les noeuds X et Y

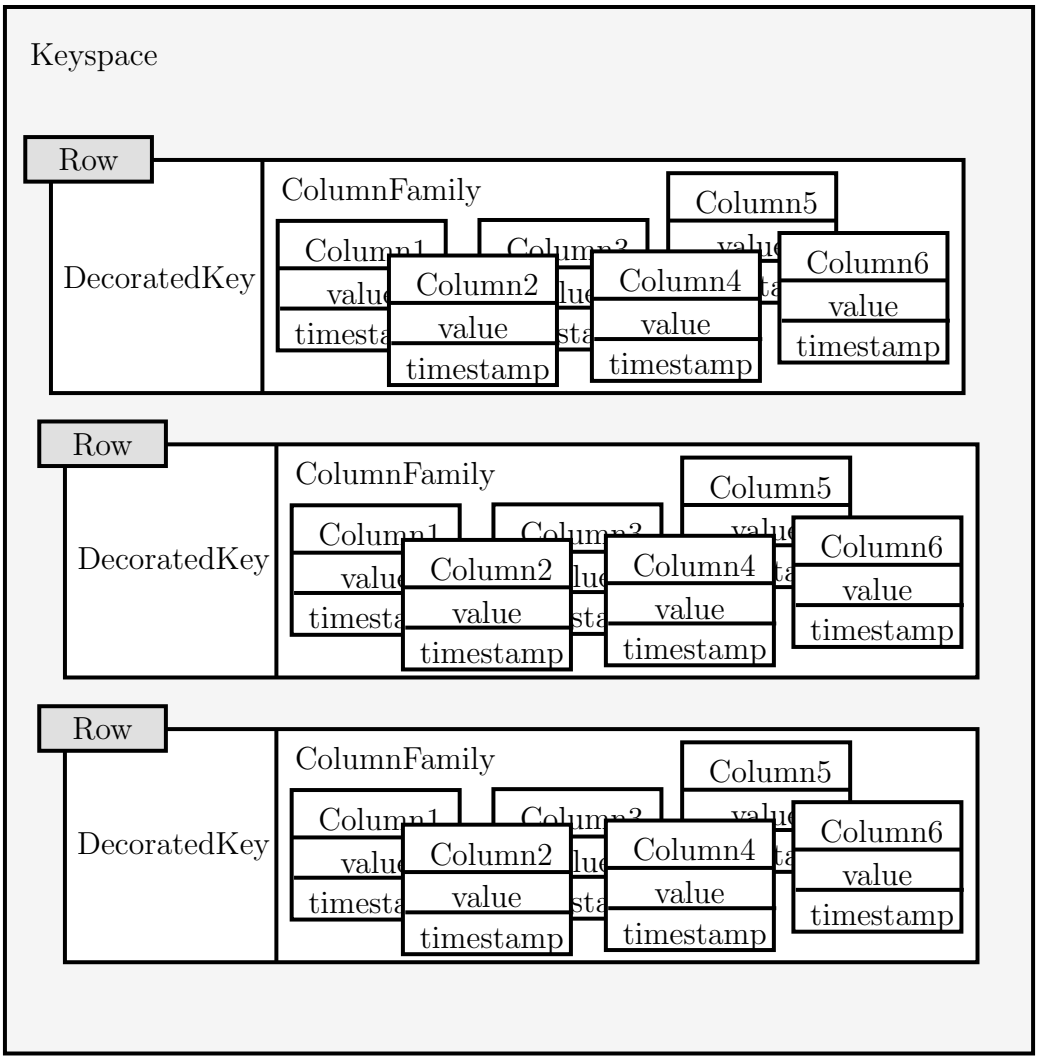

FIGURE 9 – Modèle de données de Cassandra