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

Identifications des besoins

Kevin Barreau Guillaume Marques Corentin Salingue

10 mars 2015

Résumé

Dans une première partie, nous présenterons le projet, le contexte et les hypothèses. Ensuite, nous développerons les besoins fonctionnels et les besoins non-fonctionnels. De plus, nous dégageons une première version de la plannification du projet (GANTT). Enfin, nous présenterons les livrables.

Sommaire

Table des figures

1 Présentation du projet

Dans le présent document, nous considérons que le lecteur possède des notions en informatique et que chaque mot est défini par son sens commun. Cependant, si un terme utilisé présentant une définition différente que celle admise par tous, nous ne manquerons pas de le préciser et de le définir.

1.1 Utilisation d'une base de données par un client

Une *base de données* est un outil permettant de stocker et récupérer des *données*, : codage (une représentation sous forme binaire), propre au système de base de données, d'une information quelconque.

Dans un premier temps, le client se connecte à la base de données. Le client intéragit avec celle-ci en lui envoyant des *requêtes*, message, dont la forme dépend de la base de données et permettant de stocker, récupérer ou modifier des données.

Selon les requêtes émisent par le client, la base de données envoie des resultats.

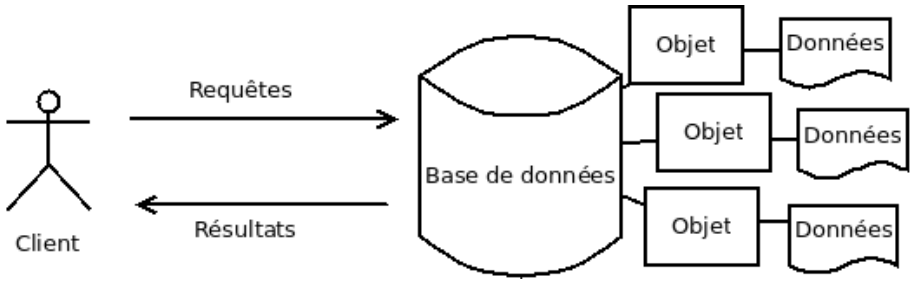


FIGURE 1 – Intéractions client/base de données

On distingue deux types de requêtes :

— Les requêtes de **lecture** : requêtes ne modifiant pas les données contenues dans la base de données. Il s'agit de récupérer des objets contenus dans la base de données.

— Les requêtes d' **écriture** : requêtes modifiant les données contenues dans la base de données.

Le client peut être une personne physique ou un logiciel. Dans notre cas, il s'agit d'un logiciel permettant l'importation de fichiers contenant des requêtes ou de génerer des requêtes pseudo-aléatoirement.

1.2 Base de données distribuée

La base de données utilisée par le client est plus précisément une base de données dites *distribuée*. Le client ne voit pas de différence, lorsqu'il l'utilise, entre une base de données classique et une base de données distribuée. On dit qu'une base de données est distribuée lorsque les données qu'elle stocke sont réparties sur plusieurs machines ou emplacements physiques, appelés *noeuds*. Les noeuds sont capables de communiquer entre eux afin de s'échanger des informations.

On peut rassembler des noeuds pour former un *data center*. Un rassemblement de data center correspond à un *cluster* (voir la figure 3).

La base de données va stocker les données sous forme d'*objets*. Un *objet* est composé d'une clé d'identification appelée *token* et d'un ensemble de *données*.

Pour savoir quel noeud doit stocker quelle donnée, on utilise une méthode de *partitionnement*. Cette méthode se base sur les *tokens*. Chaque noeud a un token qui lui est attribué. Un noeud prend en charge des objets dont le token est compris entre celui que le noeud possède et celui qui est le plus grand dans ses prédécesseurs (voir la figure 4). Ainsi dans cet exemple, le noeud 2 a le token 25 qui lui est attribué. Il s'occupe donc des objets dont le token est compris entre 25 et 0 (qui est le token le plus grand dans ses prédécesseurs). On parle alors de l'*intervalle* de tokens dont s'occupe le noeud.

En positionnant les noeuds suivant leur token, on obtient alors une forme d'anneau (ou de *ring*).

Afin de garantir une meilleure disponibilité, chaque objet possède des copies, appelées *réplicas*, disposées sur d'autres noeuds que le noeud initial (le noeud qui s'occupe du token de cet objet). La méthode pour choisir l'emplacement des copies d'un objet est variable. C'est ce que l'on appelle la *stratégie de réplication*.

1.3 Gestion des requêtes dans la base de données distribuée

1.3.1 Requêtes de lecture

Il est possible de réaliser des requêtes de lecture sur un objet, ce qui consiste à vouloir récupérer une donnée contenue dans un objet. Pour expliquer le cheminement d'une requête de lecture dans la base de données, nous allons prendre un exemple (voir la figure 6).

Un client réalise une requête de lecture R. Il envoie la requête à n'importe quel noeud du réseau. On appelle alors ce noeud le noeud *coordinateur* pour cette requête. Ce noeud ne contient pas forcément l'objet de la requête, mais il va faire la liaison entre le réseau et le client.

Le noeud coordinateur va avoir cette requête dans une file d'attente dédiée aux requêtes des clients. Il les traite les unes à la suite des autres. Lorsque le noeud commence à traiter cette requête, il va d'abord identifier les noeuds responsables de l'objet de la requête. Cela inclut le noeud possédant l'objet *original* (dont le token est géré par ce noeud) ainsi que les noeuds possédant un réplica. Cette étape exige une connaissance complète du réseau sur chaque noeud et une connaissance de la stratégie de réplication mise en place.

Dès que les noeuds sont identifiés, le noeud coordinateur leur envoie un message pour traiter la requête de lecture (les flèches rouges sur le schéma entre le noeud coordinateur et les autres noeuds). Ce message est mis dans la file d'attente des requêtes de lecture de ces noeuds.

A un moment, l'un des noeuds qui possède cette requête dans sa file d'attente va la défiler et la traiter. Ce noeud s'*affecte* la requête. Il avertit les autres noeuds possédant cette même requête dans leur file d'attente (c.à.d tous les autres noeuds possédant une copie de l'objet de la requête) qu'ils n'auront pas besoin de la traiter, et qu'ils peuvent

la supprimer de leur file d'attente (les flèches oranges sur le schéma). Le noeud qui s'est affecté la requête la traite et renvoie le résultat au noeud coordinateur, qui peut transmettre le résultat obtenu au client (les flèches vertes sur le schéma).

1.3.2 Requêtes d'écriture

Il est possible de réaliser des requêtes d'écriture d'un objet, ce qui consiste à stocker des données dans la base de données, sous forme d'objet. Pour expliquer le cheminement d'une requête d'écriture dans la base de données, nous allons prendre un exemple (voir la figure 7). Le cheminement est plus simple que pour une requête de lecture car il n'y a pas le mécanisme d'affectation.

Un client réalise une requête d'écriture R. Il envoie la requête à n'importe quel noeud du réseau. On appelle alors ce noeud le noeud *coordinateur* pour cette requête. Ce noeud n'est pas forcément celui qui va stocker les données, mais il va faire la liaison entre le réseau et le client.

Le noeud coordinateur va avoir cette requête dans une file d'attente dédiée aux requêtes des clients. Il les traite les unes à la suite des autres. Lorsque le noeud commence à traiter cette requête, il va d'abord identifier les noeuds responsables de l'objet de la requête. Cela inclut le noeud qui se charge de l'objet *original* (dont le token est géré par ce noeud) ainsi que les noeuds devant posséder un réplica. Cette étape exige une connaissance complète du réseau sur chaque noeud et une connaissance de la stratégie de réplication mise en place.

Dès que les noeuds sont identifiés, le noeud coordinateur leur envoie un message à tous pour traiter la requête d'écriture (les flèches rouges sur le schéma entre le noeud coordinateur et les autres noeuds). Ce message est mis dans la file d'attente des requêtes d'écriture de ces noeuds.

Tous les noeuds recevant le message vont alors stocker les données envoyées par la requête. Le noeud coordinateur peut demander un certain nombre de message de retour pour s'assurer que les requêtes d'écritures se sont bien déroulées. Dans l'exemple, le noeud coordinateur demande **1** retour. L'un des messages envoyés aux noeuds demandera donc un message de retour pour confirmer que l'écriture s'est bien passée (la flèche verte entre les noeuds sur le schéma). Dès que le noeud coordinateur reçoit le message, il indique au client que sa requête s'est terminée et bien passée.

1.4 Stockage des données

Chaque base de données possède sa propre manière de stocker les données dans un espace de stockage. Pour le projet, la méthode de stockage n'est pas un problème sur lequel nous allons travailler. La seule contrainte imposée pour la base de données est qu'elle stocke les données sous la forme d'objet. C'est à dire qu'un objet est identifiable par son token, une clé d'identification générée le plus souvent par une fonction de hachage.

Le token est généré par la base de donnée à partir de la *clé primaire* d'une table. Une clé primaire est, comme le token, une donnée permettant d'identifier un objet. Sauf que que la clé primaire est une donnée choisit par le client. Elle peut être un entier, une chaîne de caractères, toutes les représentations possibles d'une donnée au sein de la base de données (voir la figure 8).

1.5 Protocoles de réaffectation des requêtes de lecture

Lorsqu'une requête de lecture est envoyée par un client, on a vu précédemment que cette requête était transmise à tous les noeuds possédant une copie de l'objet à lire. Si un nombre important de requêtes de lecture arrivent en même temps, les files d'attentes dans les noeuds pour les requêtes de lecture vont commencer à se remplir plus vite que les requêtes ne sont traitées. Les charges des files d'attentes ne seront pas forcément uniformes entre les noeuds, certains pouvant avoir plus de requêtes à traiter que d'autre.

C'est pourquoi on met en place un système de *réaffectation* des requêtes de lecture, afin de rééquilibrer la charge des noeuds. La réaffectation consiste, de manière périodique, à stopper le traitement des requêtes de lecture et enclencher un processus permettant de décider de l'affectation des requêtes suivant l'état actuel du réseau.

Les algorithmes de réaffectation à implémenter, **SLVO** et **AverageDegree**, ont un comportement similaire qui se base sur la connaissance des charges de chaque noeud du réseau. L'algorithme consiste à comparer, pour tous les noeuds, sa propre charge par rapport à une certaine valeur.

Pour SLVO, la valeur est la charge minimale sur le réseau. Pour AverageDegree, la valeur est la charge moyenne sur le réseau.

Si la valeur est inférieure ou égale (strictement égale dans le cas de SLVO), alors le noeud s'affecte toutes les requêtes de sa file d'attente et avertit tous les autres noeuds. Les noeuds possédant les requêtes qui ont été affectées les suppriment de leur file d'attente, modifiant ainsi leur charge. L'opération est renouvelée jusqu'à ce que tous les noeuds se sont attribués leurs requêtes (voir la figure 9 pour un exemple avec SLVO).

1.6 Gestion de la popularité des objets

Pour mieux équilibrer la charge du réseau, nous nous intéressons à la *popularité* des objets. En effet, plus un objet va recevoir de requêtes, plus il sera populaire et occasionnera une grande charge pour les noeuds qui s'en occupent. Afin de répartir cette charge, il faudra alors augmenter ou diminuer le nombre de réplicas. Si un objet est populaire, il suffira de créer de nouveaux réplicas, ce qui permettra d'envoyer une partie de la charge sur d'autres noeuds. A l'inverse, si un objet n'est pas populaire, diminuer le nombre de copies fera gagner de l'espace mémoire et du temps (quand on a besoin de contacter tous les noeuds qui gèrent un objet, le nombre de noeuds joue sur le temps nécessaire à réaliser l'action...).

Il y a plusieurs méthodes pour calculer la popularité des objets durant un intervalle de temps T défini par l'utilisateur :

— La première consiste à ce que chaque noeud possède un vecteur de la taille du nombre d'objets dont il a la gestion. A chaque nouvelle requête, la case de l'objet est incrémentée. Au début de chaque période T, les noeuds envoient la popularité aux autres noeuds et décident du nombre de copies à faire.

— La seconde méthode est une variante visant à réduire la taille du vecteur d'objets et est défini par le Space-Saving Algorithm [**?**].

1.7 Gestion des copies d'un objet

Une *fonction de hachage* est une fonction mathématique qui possède les propriétés suivantes :

— Ensemble d'entrée : une clé primaire ;

— Ensemble d'arrivée : Un entier ;

On lui associe souvent d'autres propriétés pour équilibrer la répartition des données (cf paragraphe suivant).

Pour fabriquer un token, qui sert à placer les données sur le réseau, une clé primaire est hachée avec une fonction de hachage. Nous obtenons une valeur (ici un entier). Chaque noeud du réseau est responsable d'un intervalle d'entiers de l'ensemble des valeurs du domaine. Avec une bonne fonction de hachage, les hash des clés primaires seront distribués uniformément dans l'ensemble des entiers.

Placer les copies sur le même noeud revient à n'avoir théoriquement l'équivalent d'aucune copie puisque la charge reste sur le même noeud. Il existe plusieurs stratégies de placement de copies des données et nous n'en développerons que deux ici. La première permet de comprendre les mécanismes de base d'une stratégie de placement. La seconde décrit celle que nous allons développer.

La plus simple stratégie consiste à prendre les noeuds qui gèrent les intervalles suivants. La base de données se base sur des intervalles d'entiers. Si notre donnée est placée sur le noeud gérant les tokens [1, 25], pour placer les 3 prochaines copies, on choisira les noeuds [26, 50], [51, 75] et [76, 0] de la figure 4.

La seconde consiste à utiliser les fonctions de hachage. En effet, dans la stratégie précédente, une seule fonction de hachage était définie pour placer la donnée et les réplicas était ensuite déterminée à partir de sa position. Ici, nous avons n fonctions de hachage, toutes numérotées de 0 à $n-1$. La fonction de hachage numéro 0 sert à placer la donnée. La fonction numéro 1 sert à placer le premier réplica et ainsi de suite...

1.8 Visualisation des statistiques de fonctionnement de la base de données distribuée

Le but est de visualiser les statistiques de fonctionnement de la base de données pour permettre une comparaison de l'efficacité des algorithmes d'équilibrage de charge.

On souhaite récuperer :

— la charge effective de chaque noeud ou taille de la file d'attente des requêtes de lecture.

— une représentation de la file d'attente des requêtes de lecture

— la popularité de chaque objet

— la requête en cours de traitement

On enregistre les statistiques de fonctionnement de la base de données distribuée dans des fichiers. Un outil de visualisation traite ces fichiers et affiche ensuite les statistiques.

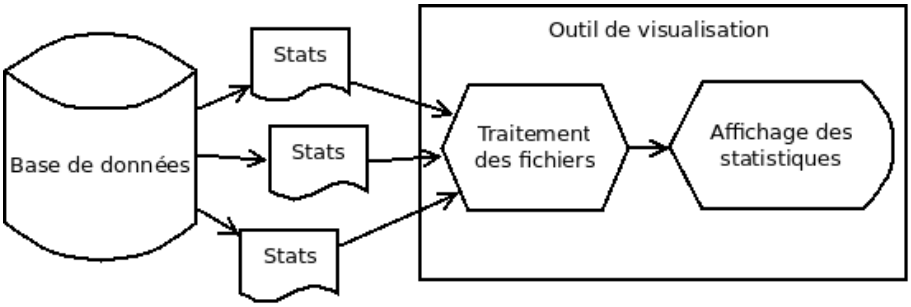


FIGURE 2 – Processus pour la visualisation des statistiques

2 Ordonnancement des besoins

Nous avons dégagé une liste de besoins fonctionnels et non-fonctionnels. Pour mieux les comparer, nous les avons ordonnés en fonction de leur priorité.

La priorité est un indicateur de l'ordre dans lequel nous devrons implémenter les fonctionnalités afin de satisfaire les besoins du client.

Valeur	Signification
1	Priorité haute
2	Priorité moyenne
3	Priorité faible

3 Besoins fonctionnels

3.1 Communication entre les noeuds

Tous les besoins concernent un seul noeud. Tous les noeuds du réseau doivent répondre à ces besoins.

Envoyer les informations du noeud à n'importe quel autre noeud (*Priorité*:1)

Recevoir les informations provenant d'un autre noeud (*Priorité*:1)

Stocker les informations de tous les noeuds du réseau (*Priorité*:1) Cela concerne tous les noeuds, y compris soi-même.

Les informations d'un noeud doivent permettre d'envoyer un message à ce dernier.

3.2 Gestion des requêtes

Tous les besoins concernent un seul noeud. Tous les noeuds du réseau doivent répondre à ces besoins.

3.2.1 Requêtes client

Créer une file d'attente des requêtes client (*Priorité*:1)

Ajouter une requête à la file d'attente des requêtes client (*Priorité*:1)

Défiler une requête de la file d'attente des requêtes client (*Priorité*:1)

Traiter une requête client (*Priorité*:1)

Identifier les noeuds responsables d'un objet (*Priorité*:1) Cela nécessite de connaitre plusieurs informations :

- la stratégie de réplication
- la token de chaque noeud
- le nombre de copie de chaque objet

Créer une requête de lecture (*Priorité*:1) Les requêtes de lecture doivent être identifiable, ceci afin de pouvoir les supprimer. Il faut donc générer un identifiant pour chaque requête de lecture lors de sa création.

Envoyer une requête de lecture (*Priorité*:1)

Créer une requête d'écriture (*Priorité*:1)

Envoyer une requête d'écriture (*Priorité*:1)

3.2.2 Requêtes de lecture

Créer une file d'attente des requêtes de lecture (*Priorité*:1)

Recevoir une requête de lecture (*Priorité*:1)

Ajouter une requête à la file d'attente des requêtes de lecture (*Priorité*:1)

Supprimer une requête de la file d'attente des requêtes de lecture (*Priorité*:1)

Défiler une requête de la file d'attente des requêtes de lecture (*Priorité*:1)

Traiter une requête de lecture (*Priorité*:1)

Créer un message de suppression de requête de lecture (*Priorité*:1)
Envoyer un message de suppression de requête de lecture (*Priorité*:1)
Recevoir un message de suppression de requête de lecture (*Priorité*:1)
Traiter un message de suppression de requête de lecture (*Priorité*:1)

Créer un message de résultat (*Priorité*:1)
Envoyer un message de résultat au noeud coordinateur (*Priorité*:1)
Recevoir un message de résultat (*Priorité*:1)
Transmettre un message de résultat au client (*Priorité*:1)

3.2.3 Requêtes d'écriture

Créer une file d'attente des requêtes d'écriture (*Priorité*:1)
Recevoir une requête d'écriture (*Priorité*:1)
Ajouter une requête à la file d'attente des requêtes d'écriture (*Priorité*:1)
Défiler une requête de la file d'attente des requêtes d'écriture (*Priorité*:1)
Traiter une requête d'écriture (*Priorité*:1)

Créer un message de résultat (*Priorité*:1)
Envoyer un message de résultat au noeud coordinateur (*Priorité*:1)
Recevoir un message de résultat (*Priorité*:1)
Transmettre un message de résultat au client (*Priorité*:1)

3.3 Réaffectation des requêtes de lecture

Les besoins sur la réaffectation des requêtes de lecture se recoupent avec ceux de gestion des requêtes de lecture en ce qui concerne les messages envoyés entre les noeuds.

Synchroniser l'horloge interne des noeuds du réseau (*Priorité*:1)
Définir un intervalle de temps périodique (*Priorité*:1)
Modifier l'intervalle de temps périodique par une configuration (*Priorité*:3) La configuration est accessible par l'utilisateur, et la valeur de l'intervalle de temps doit être la même pour tous les noeuds du réseau
Suspendre le traitement des requêtes de lecture (*Priorité*:1)
Exécuter du code à temps périodique (*Priorité*:1)
Reprendre le traitement des requêtes de lecture après une suspension (*Priorité*:1)

Connaître la charge des files d'attentes de requêtes de lecture de chaque noeud du réseau (*Priorité*:1) Cette information fait partie des informations de chaque noeud, communiqué entre eux comme vu précédemment dans la partie *Communication entre les noeuds*.
Définir un protocole de réaffectation (*Priorité*:1)
Modifier le protocole de réaffectation par une configuration (*Priorité*:3) La configuration est accessible par l'utilisateur, et le protocole de réaffectation doit être la même pour tous les noeuds du réseau
Exécuter le code d'un protocole de réaffectation défini (*Priorité*:1)

3.4 Gestion d'un réseau

3.4.1 Popularité d'un objet

Stockage de la popularité

Tous les besoins concernent un seul noeud. Tous les noeuds du réseau doivent répondre à ces besoins.

Créer un vecteur d'entiers compatabilisant le nombre de requêtes (*Priorité*:1)

Augmenter la taille du vecteur (*Priorité*:1) d ans le cas où on a l'algorithme qui calcule la popularité de tous les objets

Créer un identifiant permettant de relier la popularité à un objet (*Priorité*:1)

Calcul de la popularité

Tous les besoins concernent un seul noeud. Tous les noeuds du réseau doivent répondre à ces besoins.

Incrémenter la popularité de l'objet requêté dans le vecteur à chaque requête sur celui-ci (*Priorité*:1)

Communication de la popularité

Tous les besoins concernent un seul noeud. Tous les noeuds du réseau doivent répondre à ces besoins.

Identifier le noeud responsable d'un objet (*Priorité*:1)

Créer un message de popularité (*Priorité*:1)

Envoyer un message de popularité au noeud responsable de l'objet (*Priorité*:1)

Recevoir un message de popularité (*Priorité*:1)

Traiter un message de popularité (cf paragraphe suivant) (*Priorité*:1)

Traitement d'un message de popularité

Tous les besoins concernent un seul noeud. Tous les noeuds du réseau doivent répondre à ces besoins.

Stocker la popularité du message dans le vecteur du noeud traitant le message (*Priorité*:1)

Vérifier avoir reçu tous les messages concernant les objets dont le noeud a la gestion (*Priorité*:1)

Décider de créer ou non de nouveau réplicas (*Priorité*:1)

Réinitialiser le vecteur après la création des nouveaux objets les plus populaires (*Priorité*:1)

3.4.2 Réplication d'un objet

Tous les besoins concernent un seul noeud. Tous les noeuds du réseau doivent répondre à ces besoins.

Créer une nouvelle stratégie de réplication (*Priorité*:2)

Permettre la définition par l'utilisateur de fonctions de hachage et leur ordre d'utilisation (*Priorité*:2)

Stocker chaque fonction de hachage et son ordre (*Priorité*:2)

Définir un ordre dans les réplicas (*Priorité*:2)

Utiliser la première fonction de hachage pour placer le premier réplica (*Priorité*:2)
l a seconde pour le second, et ainsi de suite...

Retrouver les réplicas en fonction des fonctions de hachage (*Priorité*:2)

3.5 Client

3.5.1 Intéractions avec Cassandra

1 (*Priorité*:1)

1 (*Priorité*:1) A fin de pouvoir générer des requêtes.

3.5.2 Initialisation des données

1 (*Priorité*:
)

1 (*Priorité*:
)

3.5.3 Gestion de requêtes

Pour tester la validité des algorithmes, l'application devra posséder une fonction de génération de requêtes. Si l'utilisateur ne détient pas de suites de requêtes prêtes, il pourra demander à l'application d'en créer pour lui. L'application, ne connaissant pas la nature des données, ne pourra qu'effectuer un nombre restreint de requêtes différentes. Elle pourra par exemple, compter le nombre de données sauvegardées, chercher si une donnée existe réellement, mais ne pourra pas en modifier une.

2 (*Priorité*:1) I l s'agit de coder une fonction f, qui aura pour ensemble des antecedants des suites de caractères alpha-numériques (par exemple : 5832fg4gh52) et pour image un jeu de données.

1 (*Priorité*:1) L 'utilisateur peut importer son propre jeu de requêtes.

3.6 Visualisation des données

Afin de suivre l'évolution des charges de chaque noeud lors de l'exécution des algorithmes, on enregistre les données locales de chaque noeud à chaque modifications de celles-ci.

3.6.1 Enregistrement des données

Ecriture dans un fichier Lorsque les données locales d'un noeud sont modifiées, on les enregistre dans un fichier. L'écriture est de la forme `itération de l'algorithme; identifiant du noeud; charge du noeud;`

3.6.2 Affichage des données

Définition Un *graphe* est un ensemble de points appelés *sommets*, dont certaines paires sont directement reliées par un (ou plusieurs) lien(s) appelé(s) *arêtes* [**?**].

Noeuds L'application doit permettre la représentation de chaque noeud par un sommet.

Analyse syntaxique Lors de l'éxecution d'un algorithme, la charge de chaque noeud est enregistrée dans un fichier. Un analyseur syntaxique (un programme qui possède des règles et qui agit sur un fichier donné en entrée selon celles-ci) découpe chaque ligne du fichier pour récupérer le moment auquel a été enregistrée l'information (`itération de l'algorithme`), le noeud concerné (`identifiant du noeud`) et la charge de ce noeud à ce moment (`charge du noeud`).

Charge des noeuds A chaque sommet est associée une valeur correspondant à la charge de ce noeud. Ces données sont récupérées grâce à l'analyseur syntaxique.

Film de l'éxecution Cela consiste à afficher la charge des noeuds dans l'ordre chronologique, c'est à dire dans l'ordre des itérations croissant.

4 Besoins non fonctionnels

4.1 Cassandra

Cassandra est une base de données distribuée. Nous créons notre environnement distribué à partir de la dernière version stable de Cassandra.

Le choix de cette solution nous a été fortement recommandé par le client. En effet, celui-ci dispose de connaissances sur cette application et pourra donc plus facilement intervenir s'il souhaite faire évoluer le projet en implémentant par exemple de nouveaux algorithmes.

4.2 Maintenabilité du projet

L'envergure du projet fait qu'il est possible que d'autres personnes travaillent sur la finalité de ce projet, peu importe son état d'avancement. Afin de faciliter la compréhension, nous avons défini quelques normes pour que le projet puisse être repris :

— documentation dans le code source suivant la norme du langage utilisé ;

— document externe spécifiant les fichiers modifiés par rapport au code source original ;

— guide d'installation pour utiliser le projet et pour modifier le projet.

4.3 Protocole de test

La conformité des algorithmes implémentés est assurée par un protocole de test suivant la démarche :

— Définir un réseau R, un ensemble d'objets O et un ensemble de requêtes Q

— Faire tourner l'algorithme à la main avec R, O et Q

— Stocker l'état final du réseau

— Faire valider ce processus par le client

— Exécuter l'algorithme sur ordinateur avec R, O et Q

— Vérifier les résultats constatés avec les résultats attendus

S'il y a une différence entre les deux résultats, une vérification par le client peut être envisagée dans le cas de résultats *presque* similaires. La notion de similitude est laissée à l'appréciation de l'équipe en charge du projet, lors de la vérification.

4.4 Visualisation des données

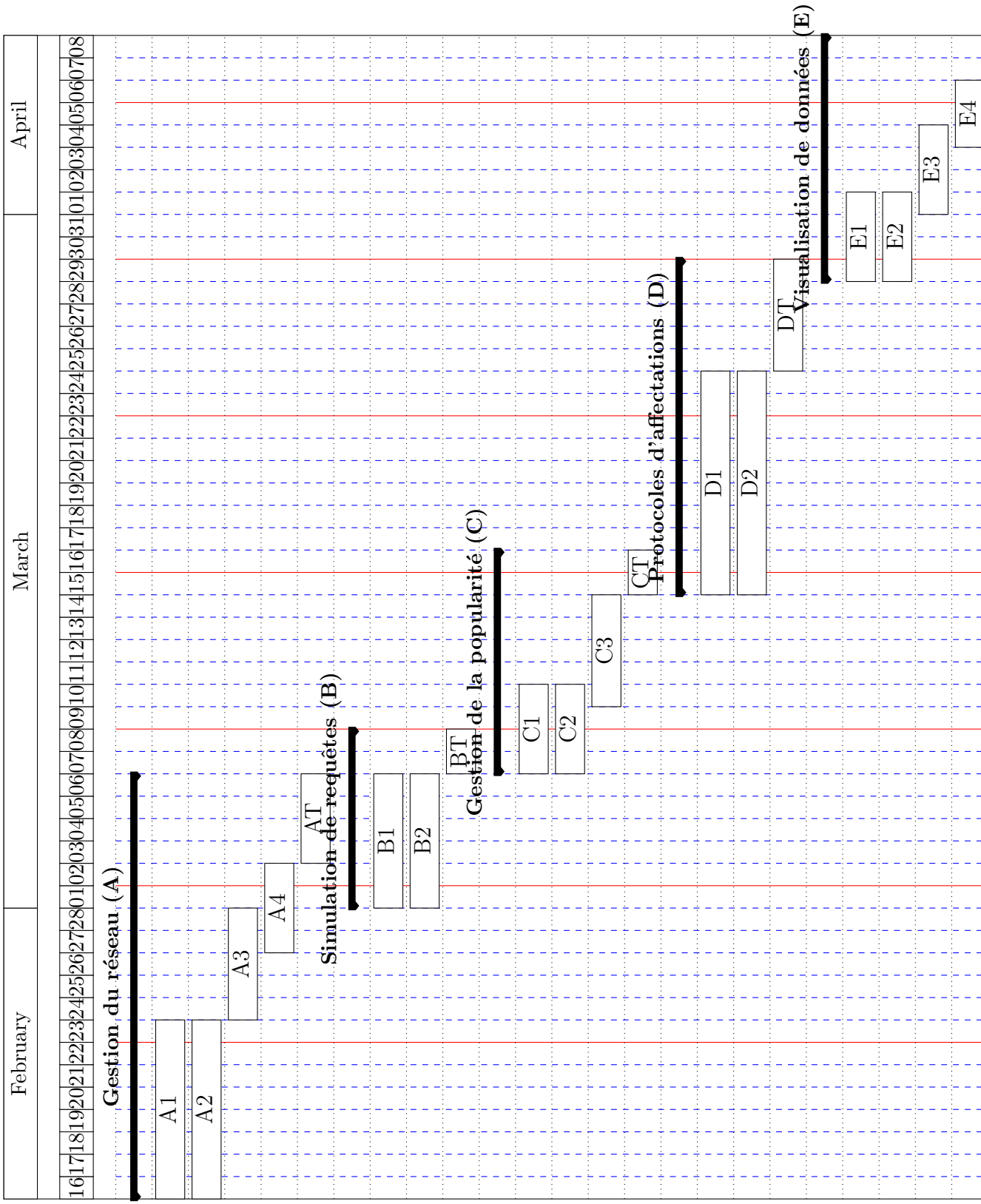
4.4.1 Actualisation de la vue

L'état du réseau doit être visible en temps réel.

La vue peut donc être actualisée toutes les 0.5 secondes. Un délai plus faible risquerait de ralentir le système, étant donné que l'obtention des données nécessaires à la visualisation se fait sur la même base de données que celle qui est testée.

5 Répartitions des tâches

5.1 Diagramme de Gantt



5.2 Affectation des tâches

Fct	Description	Développeur(s)	Commentaire
A1	Création des noeuds		
A2	Données locales des noeuds		Initialisation et implémentation
A3	Communication des données locales entre noeuds		
A4	Gestion des replicas		
AT	Tests groupe A		Vérification, tests, mémoire
B1	Générateur de requêtes		A détailler
B2	Importateur de jeu de requêtes		A détailler
BT	Tests groupe B		Vérification, tests, mémoire
C1	Popularité objet sur noeud		
C2	Space-Saving Algorithm		
C3	Popularité d'un objet dans le réseau		
CT	Tests groupe C		Vérification, tests, mémoire
D1	Implémentation SLVO		
D2	Implémentation AverageDegree		
DT	Tests groupe D		Avec client
E1	Prise en main Tulip		
E2	Ecriture des données dans un fichier		(+Analyseur syntaxique)
E2	Représentation réseau		
E3	Représentation données		
T	Tests finaux		Vérification, tests, mémoire

Remarque Il s'agit d'une première version de notre GANTT. Nous n'avons pas encore défini l'affectation des tâches aux développeurs.

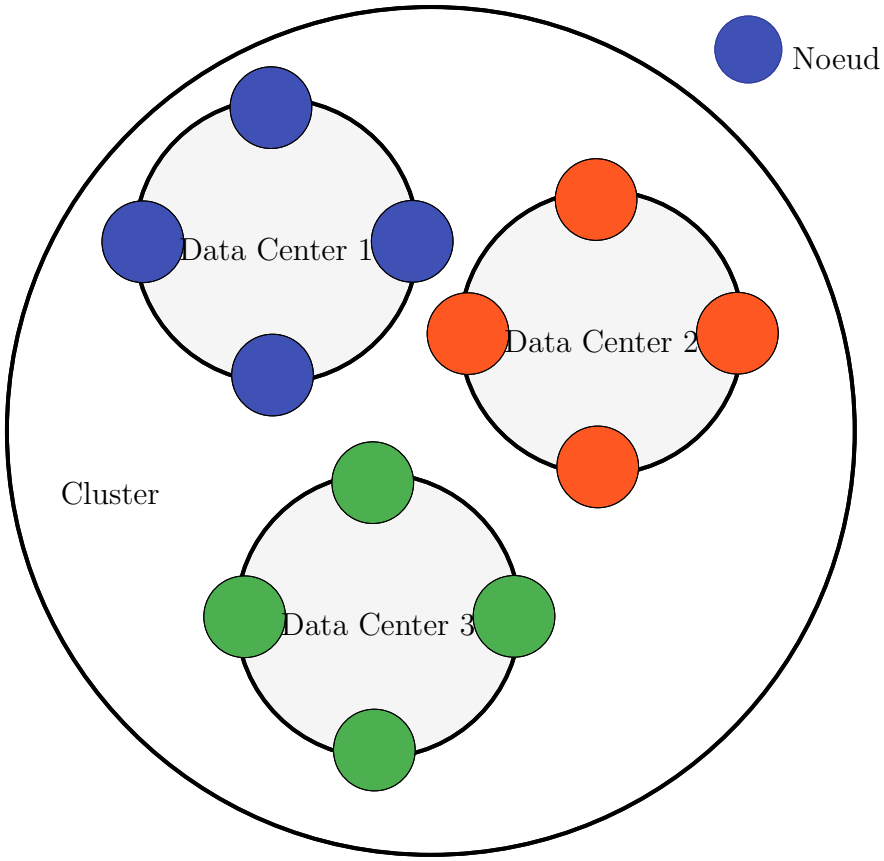


FIGURE 3 – Visualisation d'une base de données distribuée sous forme de cluster possédant trois data center

6 Livrables

6.1 Livrables intermédiaires

Un livrable intermédiaire est une ébauche de l'application. C'est à dire que seulement quelques fonctionnalités sont implémentées.

Il n'a pas encore été décidé de remettre un ou plusieurs livrables intermédiaires au client.

6.2 Livrable final

Il devra être remis le 8 Avril 2015. Il comportera les besoins de priorité 1 et 2.

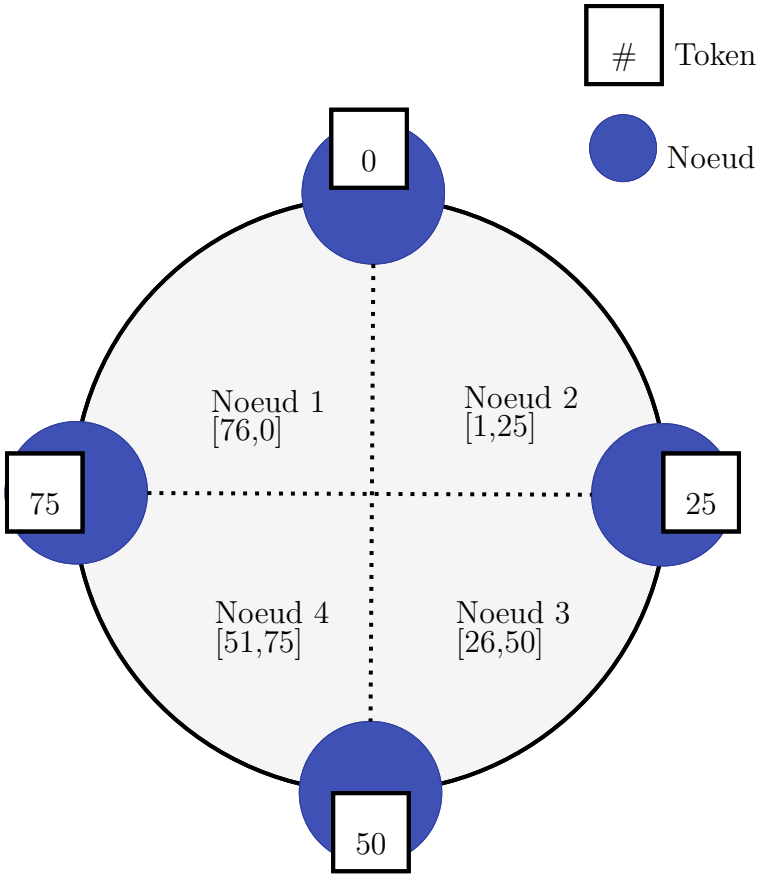


FIGURE 4 – Exemple de partitionnement des données dans une base de données distribuée

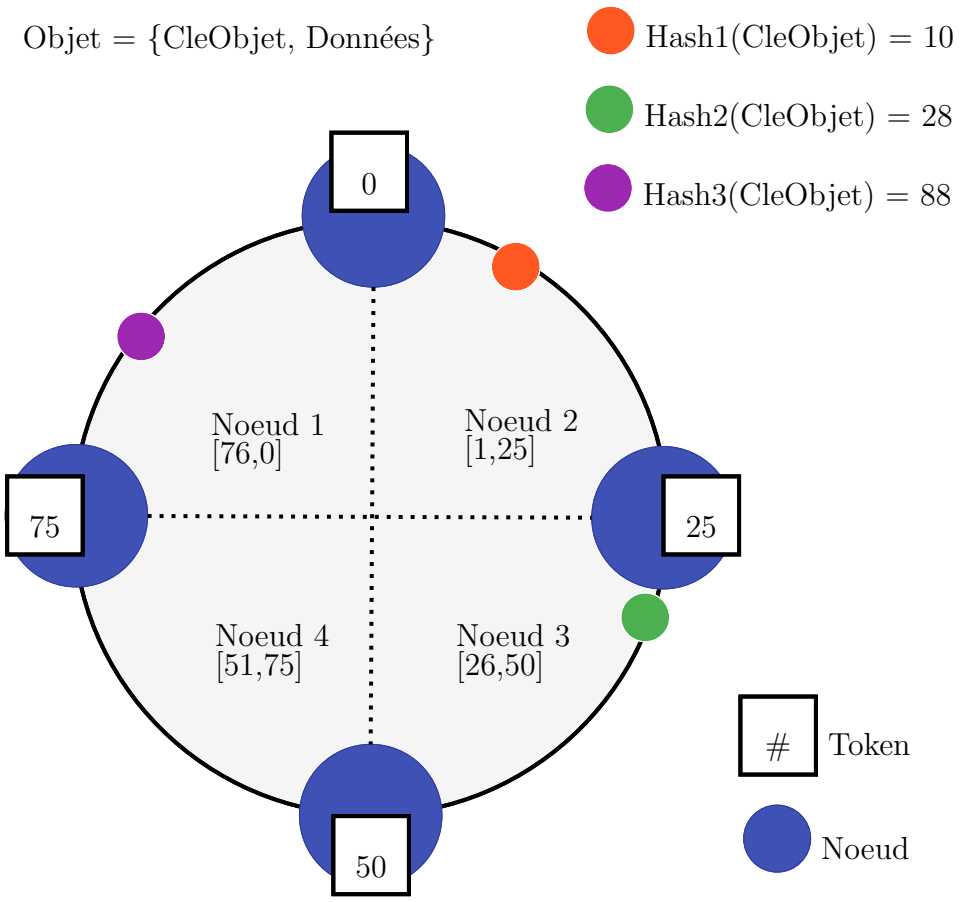


FIGURE 5 – Partitionnement des réplicas d'un objet avec une fonction de hachage pour chaque réplica

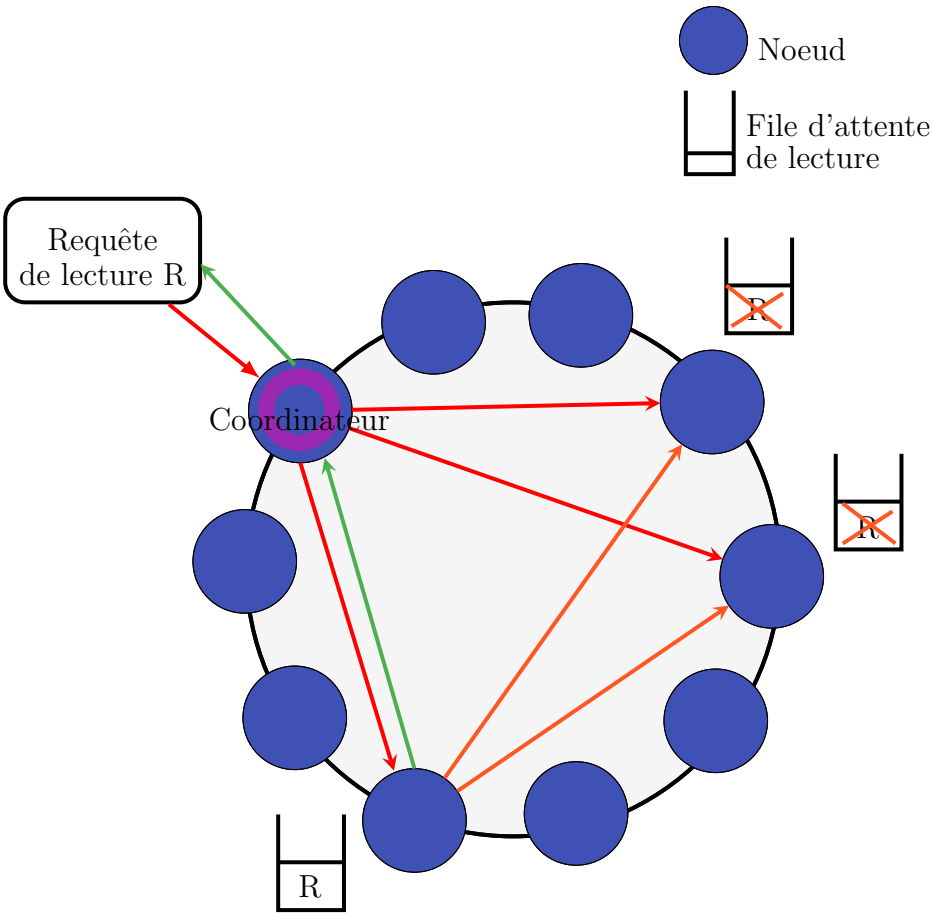


FIGURE 6 – Cheminement d'une requête de lecture dans une base de données distribuée avec la prise en charge de l'affectation (un seul noeud traite la requête)

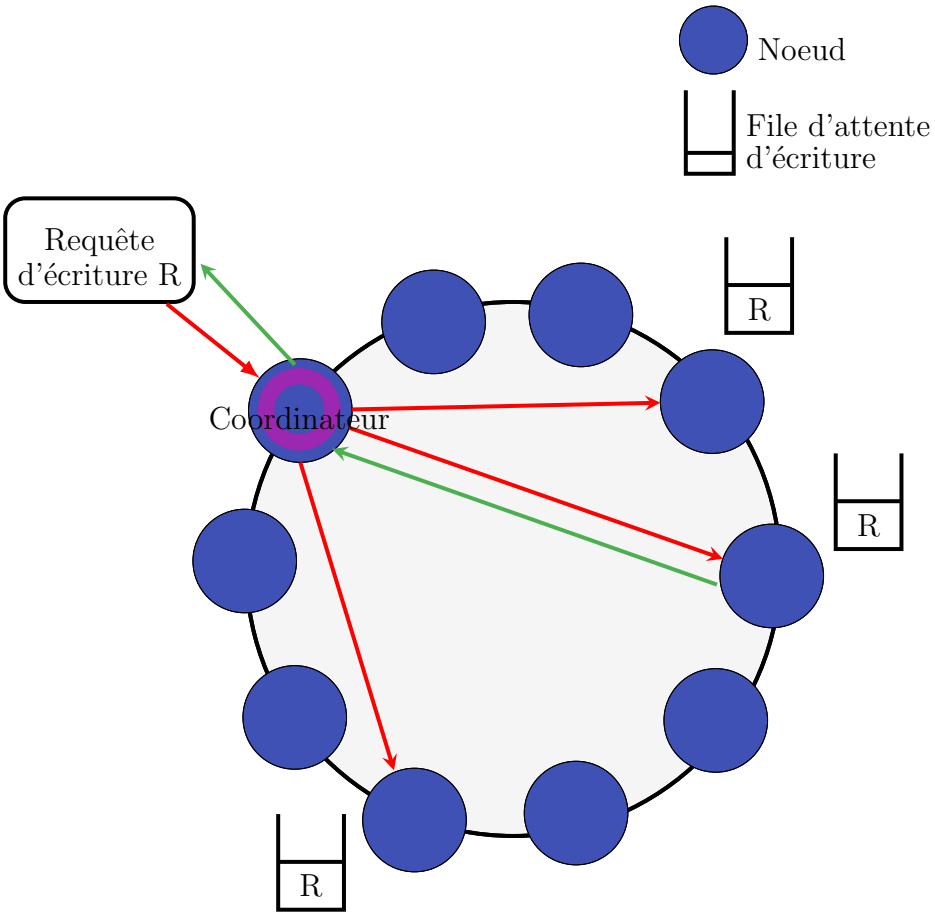


FIGURE 7 – Cheminement d'une requête d'écriture dans une base de données distribuée

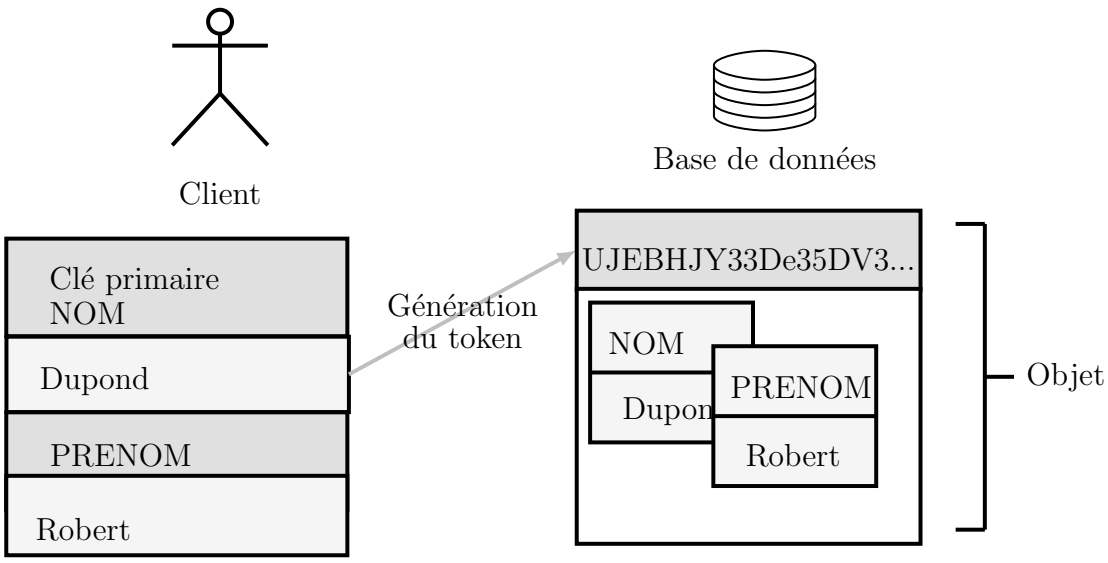


FIGURE 8 – Passage d'une répresentation des données pour le client à une représentation pour la base de données

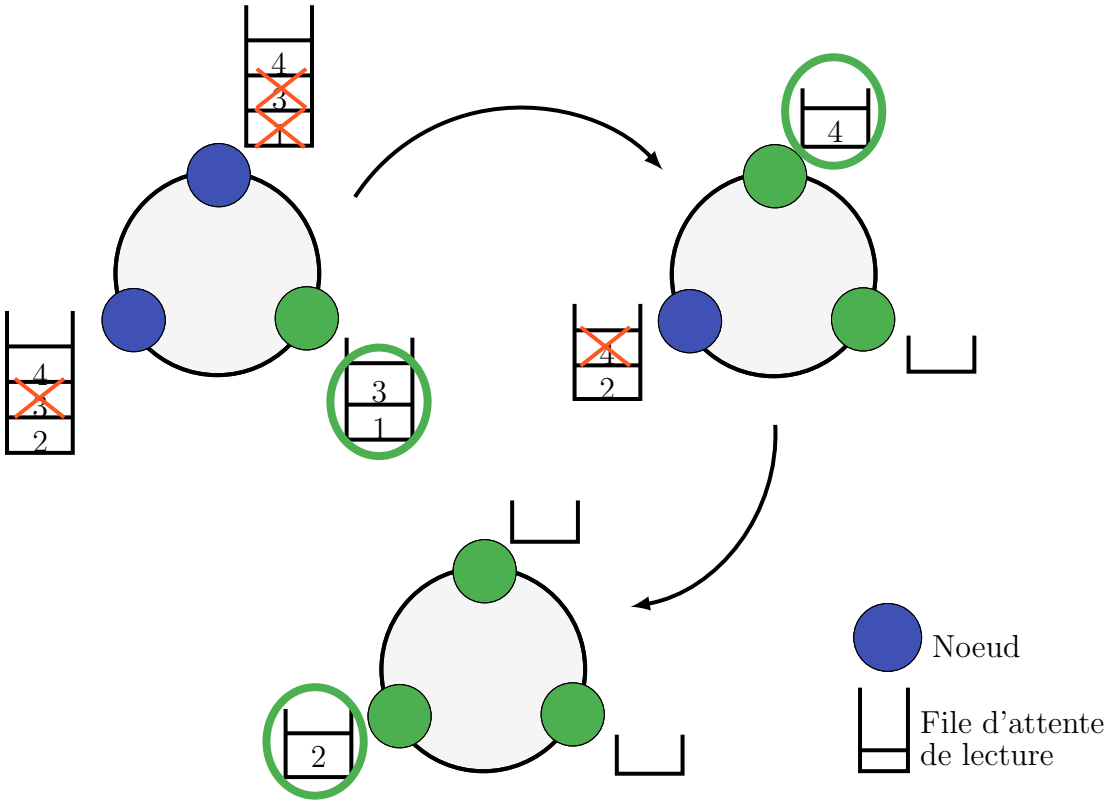


FIGURE 9 – Fonctionnement de l'algorithme de réaffectation des requêtes de lecture SLVO