NoSQL, Insertion de masse sur une architecture distribuée

Travail de Bachelor réalisé en vue de l’obtention du Bachelor HES

par :

Simon DI RELLA

Conseiller au travail de Bachelor :

Alexandros KALOUSIS, Directeur de travail

Genève, le < date de dépôt >

Haute École de Gestion de Genève (HEG-GE)

Filière Informatique de Gestion

Déclaration

Ce travail de Bachelor est réalisé dans le cadre de l’examen final de la Haute école de gestion de Genève, en vue de l’obtention du titre d’Informaticien de Gestion.

L’étudiant atteste que son travail a été vérifié par un logiciel de détection de plagiat.

L’étudiant accepte, le cas échéant, la clause de confidentialité. L'utilisation des conclusions et recommandations formulées dans le travail de Bachelor, sans préjuger de leur valeur, n'engage ni la responsabilité de l'auteur, ni celle du conseiller au travail de Bachelor, du juré et de la HEG.

« J’atteste avoir réalisé seul le présent travail, sans avoir utilisé des sources autres que celles citées dans la bibliographie. »

Fait à Genève, le 24 février 2016

Simon Di Rella

< Signez la déclaration ici >

Remerciements

Je remercie mon directeur de travail « Alexandros Kalousis » pour m’avoir suivi durant la rédaction et le développement de ce travail de Bachelor.

J’en profite également pour remercier la Haute Ecole de Gestion pour la qualité des modules proposés, ainsi que pour la facilité à l’accès à l’enseignement, tant financier qu’aux horaires proposés pour les étudiants qui sont en emploi.

Résumé

Ce travail est ciblé sur les bases de données dîtes « NoSQL », il sera présenté en deux parties.

La première partie est un travail écrit sur divers sujets des bases de données « NoSQL ». Il traite sur l’apparition de cette nouvelle technologie, ces différences avec une base de données relationnelle traditionnelle, les diverses conceptions et points techniques, son architecture et la spécification de la base de données « NoSQL » « MongoDB ».

La deuxième partie est un travail pratique dont le but et d’étudier les performances, sous certaines conditions, de la base de données de « MongoDB », à travers divers tests. Plus précisément, nous allons nous concentrer sur une des avantageuses fonctionnalités proposées par les bases de données « NoSQL », qui est la répartition (montée en charge horizontale).

Table des matières

[Déclaration i](#_Toc446270682)

[Remerciements ii](#_Toc446270683)

[Résumé iii](#_Toc446270684)

[Table des matières iv](#_Toc446270685)

[Liste des tableaux vi](#_Toc446270686)

[Liste des figures vi](#_Toc446270687)

[1. Introduction 1](#_Toc446270688)

[2. Base de données relationnelle 2](#_Toc446270689)

[2.1 Histoire 2](#_Toc446270690)

[2.2 Les principes 2](#_Toc446270691)

[3. Le Big Data et les bases « NoSQL » 3](#_Toc446270692)

[4. Relationnel vs NoSQL 4](#_Toc446270693)

[4.1 La structure 4](#_Toc446270694)

[4.2 La transaction 5](#_Toc446270695)

[4.3 Les catégories « NoSQL » 6](#_Toc446270696)

[4.3.1 Paires clé-valeur 6](#_Toc446270697)

[4.3.2 Orienté documents 7](#_Toc446270698)

[4.3.3 Orienté colonnes 9](#_Toc446270699)

[4.3.4 Orienté graphes 10](#_Toc446270700)

[4.3.5 ACID vs NoSQL 12](#_Toc446270701)

[4.4 Architecture distribuée 13](#_Toc446270702)

[4.5 Le théorème de CAP 13](#_Toc446270703)

[4.6 Synchrone ou asynchrone 14](#_Toc446270704)

[4.7 Distribution avec ou sans Maître 15](#_Toc446270705)

[4.7.1 Avec Maître 15](#_Toc446270706)

[4.7.2 Sans Maître 15](#_Toc446270707)

[5. Bases de données « NoSQL » 16](#_Toc446270708)

[5.1 MongoDB 16](#_Toc446270709)

[5.1.1 Réplication 16](#_Toc446270710)

[5.1.2 Répartition 16](#_Toc446270711)

[6. Partie pratique 17](#_Toc446270712)

[6.1 Introduction 17](#_Toc446270713)

[6.2 Infrastructure 18](#_Toc446270714)

[6.3 Outils 19](#_Toc446270715)

[7. SWITCH Engines 20](#_Toc446270716)

[7.1 Vue d’ensemble 20](#_Toc446270717)

[7.2 Menu 21](#_Toc446270718)

[7.3 Vue des instances 21](#_Toc446270719)

[7.4 Création d’instance 22](#_Toc446270720)

[7.5 Vue des volumes 23](#_Toc446270721)

[7.6 Vue des accès et de la sécurité 23](#_Toc446270722)

[8. Démarche 24](#_Toc446270723)

[8.1 Opération d’extraction et de transformation 25](#_Toc446270724)

[8.2 Opération d’insertions et recherches 28](#_Toc446270725)

[8.3 Opération de génération des rapports 29](#_Toc446270726)

[9. Résultat 30](#_Toc446270727)

[9.1 Graphiques 30](#_Toc446270728)

[9.2 Analyse et constations 34](#_Toc446270729)

[10. Conclusion 35](#_Toc446270730)

[Bibliographie 36](#_Toc446270731)

Liste des tableaux

[Tableau 1 : ACID 5](#_Toc446270732)

[Tableau 2 : Recherches 35](#_Toc446270733)

Liste des figures

[Figure 1 : Clé-valeur 6](#_Toc446270734)

[Figure 2 : Exemples clé-valeur 6](#_Toc446270735)

[Figure 3 : Orienté documents 7](#_Toc446270736)

[Figure 4 : Structure « JSON » 8](#_Toc446270737)

[Figure 5 : Structure « arbre inversé » 8](#_Toc446270738)

[Figure 6 : Orienté colonnes 9](#_Toc446270739)

[Figure 7 : Clé sur les colonnes 9](#_Toc446270740)

[Figure 8 : Orienté graphes 10](#_Toc446270741)

[Figure 9 : Nœuds, relations et propriétés 10](#_Toc446270742)

[Figure 10 : Recherche sur un modèle relationnel 11](#_Toc446270743)

[Figure 11 : Recherche sur un graphe 12](#_Toc446270744)

[Figure 12 : Théorème de CAP 13](#_Toc446270745)

[Figure 13 : MongoDB 16](#_Toc446270746)

[Figure 14 : Architecture distribuée 18](#_Toc446270747)

[Figure 15 : Vue d’ensemble 20](#_Toc446270748)

[Figure 16 : Menu 21](#_Toc446270749)

[Figure 17 : Instances 21](#_Toc446270750)

[Figure 18 : Créer une instance 22](#_Toc446270751)

[Figure 19 : Volumes 23](#_Toc446270752)

[Figure 20 : Accès et sécurité 23](#_Toc446270753)

[Figure 21 : Relation entre tables 25](#_Toc446270754)

[Figure 22 : Member « JSON » 26](#_Toc446270755)

[Figure 23 : Transfert des données 27](#_Toc446270756)

[Figure 24: Dépôt 27](#_Toc446270757)

[Figure 25 : Fichiers « JSON » 27](#_Toc446270758)

[Figure 26 : Popu-up - Nombre de shards 28](#_Toc446270759)

[Figure 27 : Insertions et recherches 29](#_Toc446270760)

[Figure 28 : Rapports 29](#_Toc446270761)

[Figure 29 : Recherches (1 machine) 30](#_Toc446270762)

[Figure 30 : Recherches (2 machines) 31](#_Toc446270763)

[Figure 31 : Recherches (3 machines) 31](#_Toc446270764)

[Figure 32 : Recherches (1'038'000 objets) 32](#_Toc446270765)

[Figure 33 : Recherches (2'076'000 objets) 33](#_Toc446270766)

[Figure 34 : Recherches (3'114'000 objets) 33](#_Toc446270767)

[Figure 35 : Insertions 34](#_Toc446270768)

# Introduction

Lorsque nous pensons « Base de donnée », nous pensons infailliblement à notre vieille base de données basée sur un modèle relationnel, incontournable depuis la fin des années 80.

Après 30 ans de domination, le « NoSQL », une nouvelle génération de bases de données, partage maintenant l’affiche avec le modèle relationnel. Suite à une convergence de divers points comme la baisse des prix, la hausse des performances des machines et de la bande passante, certaines entreprises ayant des besoins spécifiques se sont lancées dans la création de base de données répondant aux besoins auxquelles une base de données traditionnelle ne pouvait pas répondre. Souvent pour faciliter la montée en charge ou la recherche rapide d’informations.

Lors d’un événement consacré à l’architecture distribuée en 2009, le terme « NoSQL » fut inventé [1]. Il est une abréviation de « Not Only SQL ». On pourrait polémiquer sur ce terme, mais il a l’avantage de faire parler de lui. En réalité, ces nouvelles bases de données ne se séparent pas du langage SQL, puisque certaines d’entre elles utilisent une variante pour accéder aux données. On pourrait dire qu’elles sont non relationnelles, et donc, dû à l’absence de relation entre les tables, cela peut générer des données redondantes et une demande accrue d’espace de stockage.

En résumé, le « NoSQL » n’est ni une régression, ni une évolution du modèle traditionnel, mais tout simplement une nouvelle conception adaptée à l’analyse de grands volumes de données, appelé le « Big Data ». Comme l’on pourrait choisir une voiture, rapide mais inconfortable, à une spacieuse mais peu réactive, notre choix se fera pour combler au maximum les besoins prioritaires.

# Base de données relationnelle

## Histoire

Le concept de base de données relationnelle fut inventé par « Edgard Franck Codd » [2], En 1948, il commença à travailler pour IBM en tant que programmeur. Il obtint un doctorat en Computer Science qui lui permit de travailler par la suite comme chercheur aux laboratoires de recherche de San José. En 1969, il publia un article qui posa les fondements du modèle relationnel, ainsi que les bases mathématiques et algébriques de relations.

En 1974, le laboratoire de San Jose se lança dans le développement d’un prototype expérimental, appelé « System R », basé sur les concepts de « Edgard Franck Codd ». Afin de pouvoir manipuler les données, ils développèrent un langage nommé « Structured English Query Language – Sequel», qui fut changé en « SQL » dû au terme déjà déposé par une compagnie britannique pour désigner une marque.

Les fondements des bases de données relationnelles étaient posés. Suite aux bonnes performances du System R, Oracle, DB2 et les descendants d’Ingres implémentèrent ce concept et le rendirent plus populaire.

## Les principes

Un des principes est d’établir un modèle qui permet de séparer clairement la représentation logique des données, de son organisation physique. La représentation logique des données permet de modéliser un ensemble de données par l’utilisation de tables et de leurs relations, ce qui force l’utilisateur à se plier à ces concepts. Toutefois, le concept évite la redondance des données et permet l’application des opérations algébriques, telles que l’intersection, la jointure ou le produit cartésien.

# Le Big Data et les bases « NoSQL »

Au fur et à mesure de l’augmentation du volume des données dans certains secteurs, et surtout celui d’Internet, le modèle relationnel perdait en performance dans la gestion des données à traiter. Dopés par l’augmentation de la bande passante, la baisse des coûts des machines et la généralisation du réseau, de nouveaux concepts apparurent, comme la virtualisation et l’informatique distribuée.

Un nouveau défi vu le jour : comment traiter un volume de plusieurs téraoctets, tout en ayant des performances optimales ?

C’est en octobre 1997, que l’expression « Big Data » apparu à travers des articles scientifiques [3], qui bien sûr désigne le fait d’avoir des quantités tellement énormes de données qu’il devient difficile de garantir un traitement efficace et performant. Le défi était lancé. Plusieurs entreprises, dans la contrainte, commencèrent à rechercher et développer de nouvelles technologies utilisant les nouveaux concepts liés à l’évolution d’Internet, comme par exemple Google [4] ou Amazon [5].

La problématique du « Big Data » fît voir le jour aux premières bases de données dîtes « NoSQL ». Couplées aux concepts d’architecture distribuée et virtualisation, que nous aborderons plus en détail dans d’autres chapitres, elles solutionnèrent le problème.

Il va de soi que cette gestion de données, structurellement très souple, est spécifique à ce genre de problématique, n’est pas une évolution des bases de données traditionnelles. Elle amène une nouvelle dimension sur le choix de la basse de données à utiliser lors d’un nouveau projet. Elle n’intègre en aucun cas tous les avantages que peut fournir une base de données traditionnelle en plus des siens.

En résumé, il faut garder à l’esprit que les bases de données « NoSQL » sont un autre moyen de traiter les données, et le choix de la base de données n’est que plus varié. Il faudra cependant faire une analyse des besoins afin de choisir la plus appropriée.

# Relationnel vs NoSQL

Dans ce chapitre, nous allons aborder les points techniques qui nous permettrons de connaître les différences entre les bases de données relationnelles et « NoSQL ». La compréhension de ces divers points est nécessaire lors du choix d’une base de données pour un nouveau projet. Elle vous permettra de choisir la plus adaptée.

## La structure

Relationnel

Comme nous avons pu le voir dans le « Chapitre 2 », une base de données relationnelle nécessite au préalable un modèle logique de données. C’est-à-dire qu’il faut créer une structure relationnelle à partir du modèle, où les données pourront y être stockées, représentée par des tables qui ont des colonnes et des lignes, et qui sont reliées entre elles. Pour un développeur, cela implique qu’il lui faudra acquérir également cette connaissance pour modéliser son schéma.

NoSQL [6]

La plupart des bases de données « NoSQL » utilisent l’agrégat. L’idée de ce concept et de structurer les informations sous forme de grappe, avec une entité racine, qui est le point d’accès. Par exemple, cette entité pourrait être une boîte aux lettres. A l’intérieur de celle-ci, il pourrait y avoir des enveloppes, des journaux et des magazines. Dans les enveloppes, on pourrait y trouver des lettres et des documents. Dans les lettres, des paragraphes, et ainsi de suite. De plus cette structure reste très souple, puisque le développeur n’a pas besoin de la modéliser au préalable, il n’a plus qu’à l’enrichir, et peut y ajouter de nouveaux objets. Bien sûr, pour diverses raisons, et surtout pour y rechercher des informations, il est préférable de ne pas tout mélanger. Par rapport à l’idée de la boîte aux lettres, ça serait préférable de ne pas y trouver une paire de chaussures.

## La transaction

Dans une base de données transactionnelle, lorsque celle-ci est interrogée, enrichie ou mise à jour, un mécanisme appelé « transaction » est enclenché. Ce mécanisme garantit 4 critères à respecter, représenté par l’acronyme « ACID » [7], que l’on nomme l’acidité d’une transaction.

Tableau  : ACID

|  |  |
| --- | --- |
| **A**tomicité | La transaction est validée intégralement ou annulée totalement. C’est tout ou rien. |
| **C**ohérence | La transaction doit maintenir la base de données dans un état cohérant. La base de données peut être dans un état incohérent durant l’exécution de la transaction. Une fois la transaction validée, la base de données doit être à nouveau cohérente, ou si la transaction est annulée, laisser la base de données dans l’état cohérent antérieur. |
| **I**solation | Puisque la transaction met la base de données dans un état incohérent durant son exécution, elle doit être isolée des autres transactions pour qu’elle ne puisse pas lire les données en cours de modification des autres transactions. |
| **D**isponibilité | Une fois la transaction validée, le nouvel état doit être durablement inscrit dans la base de données. |

## Les catégories « NoSQL »

Durant cette dernière décennie, plusieurs bases de données « NoSQL » ont vu le jour, avec des réponses à des besoins spécifiques. Même si elles peuvent être assez différentes d’un point de vue de leur conception, elles sont toutefois gouvernées par des principes communs. Voici les classements par schéma de données.

### Paires clé-valeur

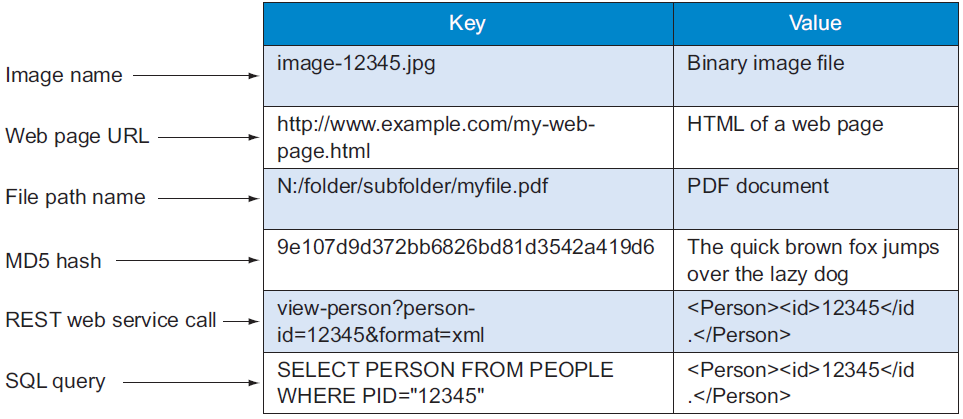
Figure  : Clé-valeur



C’est un moyen simple d’utilisation, pour chaque clé (chaîne de caractère), une valeur est stockée. Comme un dictionnaire, pour chaque mot il y a une définition. Ce point unique d’accès permet d’avoir de très bonnes performances, indépendamment du nombre de données, et permet de ne pas avoir un moteur de requêtes très complexe. Il fournit des fonctionnalités simplifiées.

Il y a d’autres avantages comme le fait de ne pas spécifier le type de données pour la valeur. Le système stockera la valeur en tant que BLOB, et permettra ainsi de stocker également des fichiers comme par exemple des « XML », images, etc...

Figure  : Exemples clé-valeur



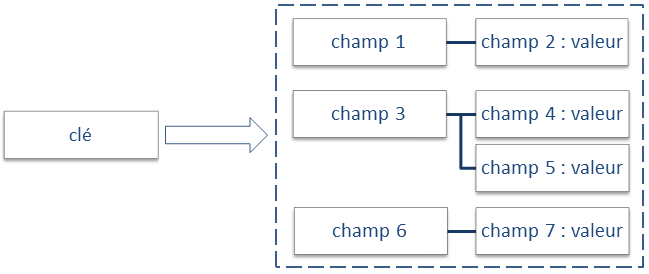
Pour manipuler les données, les trois fonctions les plus utilisées sont :

* GET($clé en tant que chaîne de caractères) pour récupérer la valeur.
* PUT($clé en tant que chaîne de caractères, $valeur) pour ajouter ou modifier la valeur.
* DELETE($clé en tant que chaîne de caractères) pour supprimer la valeur.

Exemple de bases de données clé-valeur : Dynamo, Riak, Redis et OpenLDAP.

### Orienté documents

Figure  : Orienté documents



Au contraire de la catégorie clé-valeur, dont la valeur a une structure formelle qui ne peut pas être indexée ou recherchée. La catégorie orientée documents impose une valeur structurée, appelée document, dont le contenu est automatiquement indexé dès son ajout dans le système, et permet donc de le retrouver par un de ses attributs. La clé du document devient un identifiant automatiquement généré et qui n’est pas forcément visible ou utilisé. Les documents peuvent ensuite être groupés dans une liste.

Le format privilégié est en général le « JSON » (Javascript Object Notation). Il permet d’avoir une représentation de données structurée hiérarchique de propriétés et de valeurs. Elle pourrait ressembler à un arbre inversé, avec sa racine ses branches et ses feuilles.

Figure  : Structure « JSON »

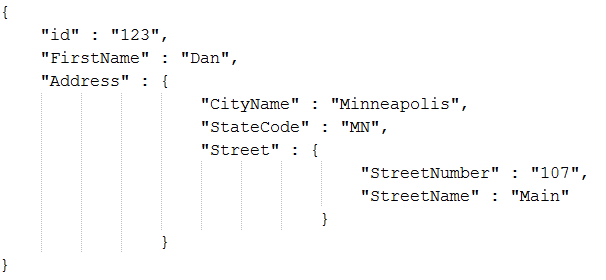
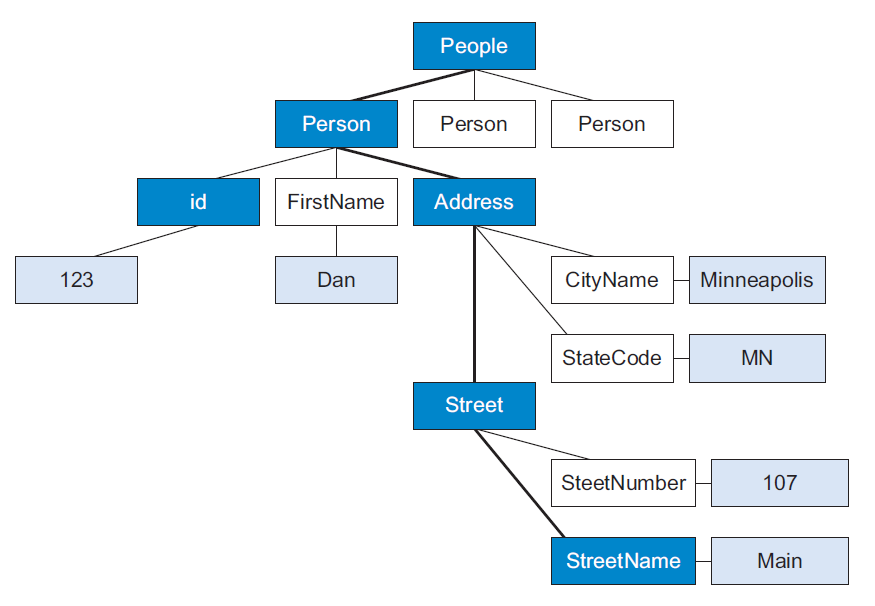


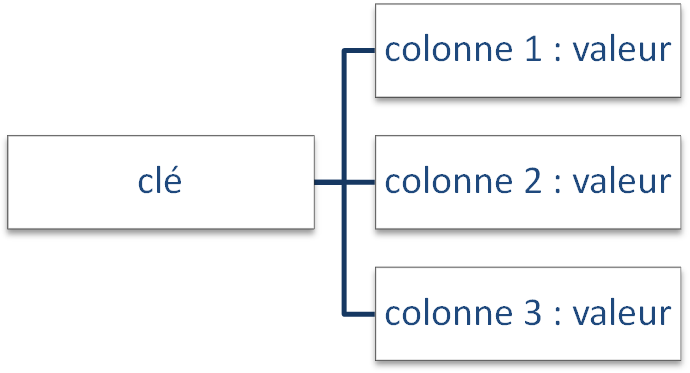
Figure  : Structure « arbre inversé »



Exemple de bases de données orientées documents : MongoDB et CouchDB.

### Orienté colonnes

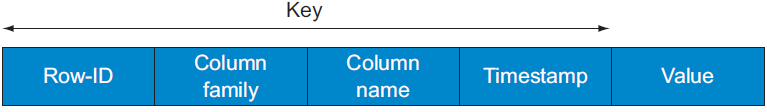
Figure  : Orienté colonnes



Elle reprend le principe des bases de données relationnelles sur le concept des tables. C’est-à-dire une structure de colonnes et lignes avec une notion de clé sur les lignes. Toutefois, la structure reste plus souple et permet de pouvoir ajouter des colonnes dynamiquement. Il ne faut pas chercher à faire du relationnel, elles ne sont pas faites pour cela.

Dans la figure ci-dessous, on pourrait très bien récupérer plusieurs lignes groupées par la colonne « Family », soit unitairement par l’identifiant de la ligne.

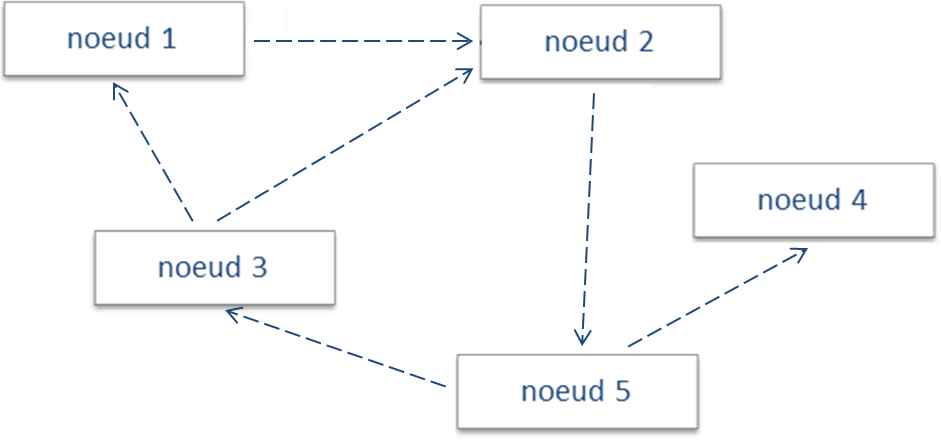
Figure  : Clé sur les colonnes



Exemple de bases de données orientées colonnes : Cassandra et HBase.

### Orienté graphes

Figure  : Orienté graphes

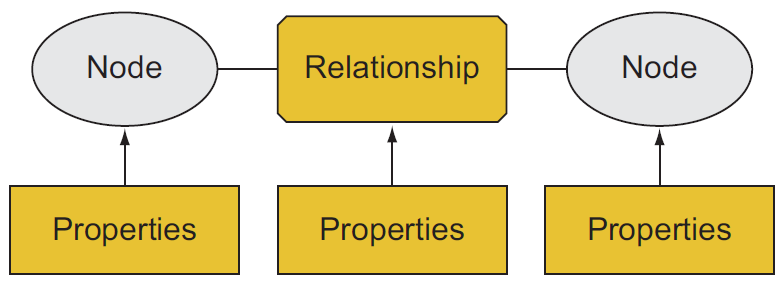


La base de donnée orientée graphe est surtout utilisé pour des cas de relations complexes entre objets, elle permet également d’atteindre les nœuds de manière « Traversal » que nous décrirons plus loin dans le chapitre.

C’est un système qui contient des séquences de nœuds et de relations, leurs combinaisons créé un graphe. Les trois types de données que l’on retrouve dans cette conception sont les nœuds, les relations et les propriétés.

Prenons un exemple de relations entre personnes. Le nœud serait donc une personne qui contient des propriétés comme le nom et le prénom. Ce nœud serait lié à un autre nœud par une relation qui pourrait être « meilleur ami », et qui pourrait également contenir des propriétés comme la date de leur rencontre. On pourrait également à tout moment ajouter un autre nœud d’un autre type, comme celui de voiture, et avoir une relation entre une personne et une voiture « possède ».

Figure  : Nœuds, relations et propriétés



#### Les avantages

Les bases de données orientée graphe suivent généralement les critères suivants, bien entendu, si le besoin est de stocker des données ayant une structure de relations complexes :

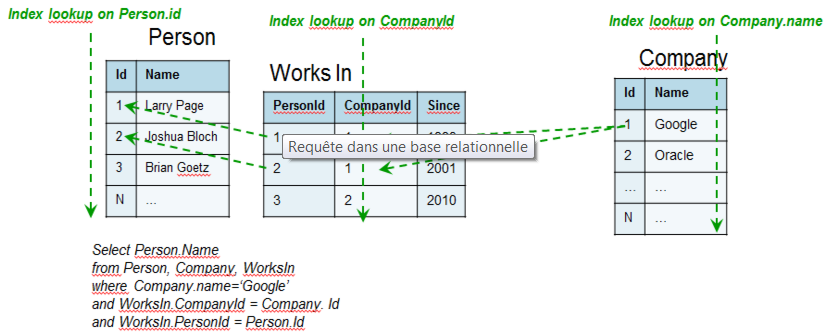
* Le stockage est optimisé pour les données.
* Le stockage est optimisé pour la lecture et le parcours de données « Traversal ».
* Le modèle de données est flexible pour certains produits, il n’y a pas besoin de créer explicitement une entité.

#### Orienté graphe et transactionnelle

Nous allons essayer de comprendre dans l’exemple qui suit pourquoi la base de données orientée graphe est plus adaptée à un besoin complexe mettant en relation des objets, qu’une base de données relationnelle. Prenons le cas de personnes qui travaillent dans une société et cela depuis combien de temps. Avec une base de données relationnelle, on modéliserait les sociétés, les personnes et le lien entre elles.

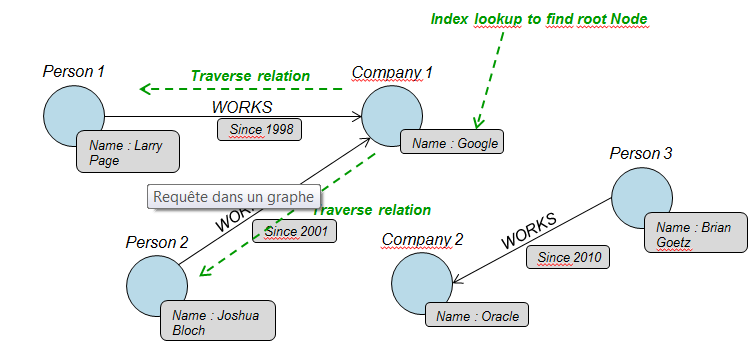
D’un point de vue technique, si par exemple, on aimerait savoir quelles sont les personnes qui travaillent dans la société « Google ». Pour une base de données relationnelle, l’exécution se ferait via trois recherches en fonction des indexes et des clés étrangères.

Figure  : Recherche sur un modèle relationnel



Pour une base de données orientée graphe, l’exécution se fera via une recherche d’indexe pour trouver le nœud racine ou de départ, puis un parcours par pointeurs physiques des relations « Traversal ».

Figure  : Recherche sur un graphe



Exemple de bases de données orientées graphes : Neo4j et HyperGraphDB.

### ACID vs NoSQL

Lors d’une transaction dans une base de données relationnelle, l’opération est sujette à un mécanisme d’isolation. Une donnée complexe à insérer, sera décomposée dans plusieurs tables et seras soumise à plusieurs opérations d’écriture.

Un moteur NoSQL, basé sur le principe de l’agrégat, ne sera pas soumis à ce genre de problématique. Par exemple avec MongoDB, un document « JSON », comportant une donnée structurée, sera inséré par une seule opération d’écriture et sera donc atomique. La transaction perdra donc sa complexité.

Lorsque les données sont distribuées, dans un système « NoSQL », elles sont dupliquées. C’est lors de la réplication, et donc sur le fait d’avoir des données redondantes que des questions se posent autour de l’acidité de la transaction et de sa cohérence.

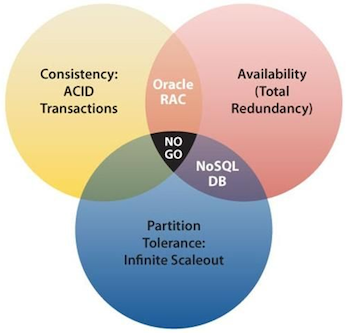
## Architecture distribuée

Comme il est cité dans les chapitres précédents, l’évolution de diverses technologies et surtout dans le domaine des communications réseaux, a fait voir le jour à de nouveaux concepts de bases de données appelées « NoSQL ». En effet, les machines sont plus puissantes et elles peuvent dialoguer entre elles avec plus de rapidité. Ce concept appelé « architecture distribuée » permet d’avoir une base de données ayant plusieurs nœuds. Un nœud est en réalité la base de données elle-même qui est installée sur une machine. Il faut imaginer qu’il y a plusieurs nœuds, donc plusieurs machines, et qu’ils doivent sans cesse communiquer pour se synchroniser. Ce procédé permet ainsi de distribuer la charge et donc de garder des temps de réponse assez rapides.

## Le théorème de CAP

D’après « Eric Brewer » [8], professeur à l’université de Berkeley en Californie, un système informatique de calcul distribué doit répondre à deux des trois contraintes que nous allons voir ci-dessous.

Figure  : Théorème de CAP



Cohérence (Consistency)

Tous les nœuds sont à jour sur les données au même moment.

Disponibilité (Availability)

La perte d’un nœud n’empêche pas le bon fonctionnement du système et de servir l’intégralité des données.

Résistance au morcellement (Partition Tolerance)

Chaque nœud doit pouvoir fonctionner de manière autonome.

Le système ne peut répondre à ces trois contraintes en même temps. A partir de ces trois contraintes, il existe trois catégories de classification qui sont simplement les intersections de ces contraintes.

## Synchrone ou asynchrone

Il y a deux manières de concevoir la distribution des données. La distribution synchrone garantie que tous les nœuds soient synchronisés pour qu’une écriture soit terminée. La distribution asynchrone ne le garantie pas, bien qu’au final, les nœuds finissent par être synchronisés. Cette manière de distribution gagne en performance par rapport à celle synchrone, mais ne garantit pas la cohérence. Les bases de données relationnelles doivent forcément avoir une architecture distribuée respectant la « cohérence », dû à l’acidité de leurs transactions.

Certaines bases de données « NoSQL » ont décidé de ne plus être contraintes par la cohérence, et ont donc adopté une architecture distribuée asynchrone, afin de gagner en performance.

## Distribution avec ou sans Maître

### Avec Maître

Une des solutions de réplication des données est de définir un des nœuds comme maître. Celui-ci aura la responsabilité d’intégrer tous changements de données, tandis que les nœuds esclaves peuvent être accessibles seulement en lecture. Le maître se chargera aussi de répliquer les données sur les esclaves. Selon le caractère synchrone ou asynchrone de la base de données, une donnée lue sur un nœud esclave pourrait être obsolète durant un cours laps de temps.

### Sans Maître

Dans une réplication de données sans maître, il existe plusieurs solutions afin de pouvoir répliquer l’information, mais qui demande de se poser plusieurs questions. Comment diriger les requêtes des clients sur les machines ou comment maintenir l’état du système, c’est-à-dire en résumé, comment la réplication doit se faire, de quelle machine à quelle machine ?

# Bases de données « NoSQL »

## MongoDB

Figure  : MongoDB



Précédemment nommée 10gen, « MongoDB » a été créé par « Kevin Ryan » et « Dwight Merriman ». L’histoire commence dans l’entreprise de ces messieurs appelée « DoubleClick », bien connue dans la publicité en ligne. Etant dans l’impossibilité d’effectuer des réglages dans la montée en charge du système. Ils décidèrent de développer leur propre plate-forme applicative distribuée.

« MongoDB » est développée en C++, langage qui offre de grandes performances. C’est une base de données orientées documents, ce qui lui permet d’avoir une représentation structurée, comme on a pu l’aborder dans un chapitre précédent.

### Réplication

Pour divers cas utiles, il est possible de répliquer les données. Un processus « MongoDB » lancé est appelé un nœud. Il est de toute évidence conseillé de créer un nœud par machine, au cas où l’une d’entre elles tomberait en panne.

La topologie de réplication est de type maître-esclave, c’est-à-dire qu’un maître est désigné pour recevoir toutes les écritures qu’il distribue aux esclaves. Dans le cas où le maître ne répondrait plus, les esclaves désignent un nouveau maître.

### Répartition

La montée en charge horizontale est réalisée par répartition, c’est-à-dire que les données peuvent être réparties sur plusieurs nœuds, au maximum jusqu’à mille. Pour cela, il faut que les données soient intégrées à une collection configurée pour la répartition. Dès lors, « MongoDB » se chargera de répartir le plus équitablement possible les données sur les divers nœuds.

# Partie pratique

## Introduction

Le but de la partie pratique est de mettre en évidence les performances d’une montée en charge horizontale d’une base de données « NoSQL ».

Notre choix s’est porté sur la base de données « MongoDB » pour les raisons suivantes :

Premièrement, c’est une base de données orientée documents « Open Source » qui a fait ses preuves dans le domaine professionnel. Elle comprend les fonctionnalités recherchées, comme la réplication et la répartition. C’est sur la répartition que nos tests de performances vont s’appuyer, même si la réplication est également configurée.

Une des autres raisons est la multitude de connecteurs développés dans divers langages de programmation qu’elle propose (Actionscript, C, C#, C++, Clojure, ColdFusion, D, Dart, Delphi, Erlang, Go, Groovy, Haskell, Java, JavaScript, Lisp, Lua, MatLab, Perl, PHP, PowerShell, Prolog, Python, R, Ruby, Scala, Smalltalk). Le choix est vaste et me permet de choisir le langage avec lequel je suis le plus à l’aise.

Les divers tests de performance vont porter sur les insertions et les recherches dont le résultat sera illustré graphiquement.

Afin de pouvoir effectuer ces tests, il a fallu mettre en place une infrastructure intégrant le concept d’architecture distribuée. C’est-à-dire avoir plusieurs machines pouvant communiquer entre elles, et configurées de telle manière à pouvoir appliquer la répartition.

Indépendamment de cette infrastructure, une machine « client » exécute un processus afin d’alimenter, de requêter et de récolter les données de performance via le point d’entrée de celle-ci.

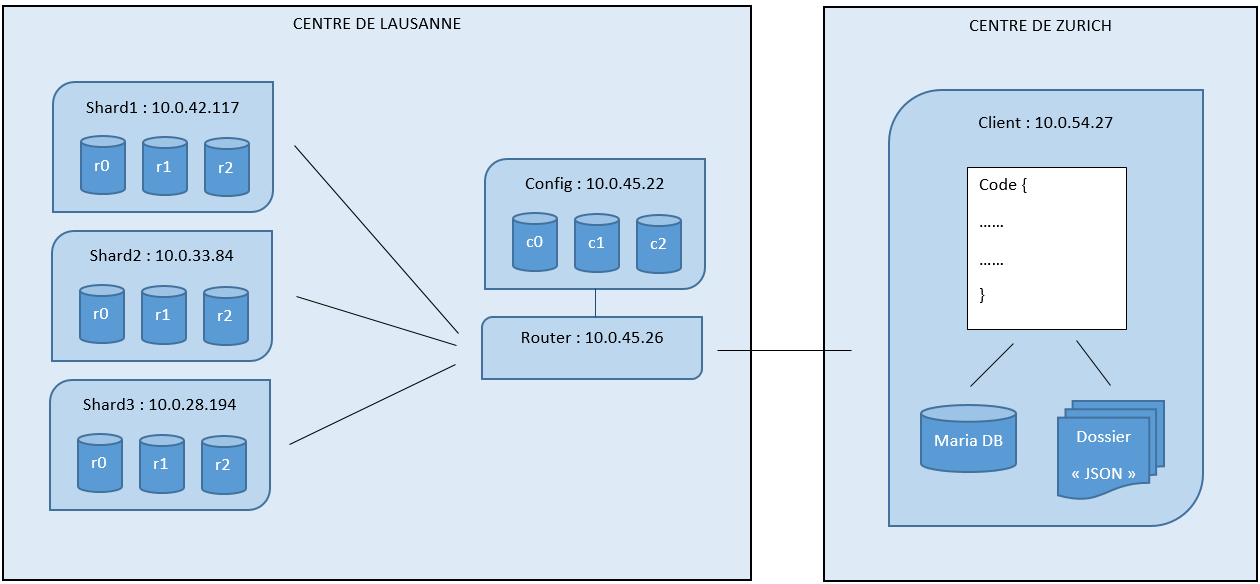
Les données « source » qui y seront insérées, proviennent d’une base de données « MariaDB ».

## Infrastructure

Afin de faciliter la création de machines pouvant intégrer les nœuds « MongoDB », nous avons utilisé un service permettant de gérer la gestion de machines virtuelles. Ce service nous a été fourni par SWITCHengines dont la configuration se fait à travers une application-web. Le chapitre suivant sera dédié à sa configuration.

Pour interagir avec cette infrastructure, il faut définir un point d’entrée. La machine ayant ce rôle est la machine « router », elle sert donc d’interface, et absorbera ainsi les requêtes de la machine « client ». Grâce à la répartition, les données seront intégrées dans une collection partagée, et distribuées équitablement sur les trois autres machines nommées « shard1 », « shard2 » et « shard3 ». La configuration de communication des shards est stockée sur la machine de configuration « config ».

Figure  : Architecture distribuée



## Outils

Diverses technologies ont été utilisées pour le développement du programme installé sur la machine « client », que nous décrirons ci-dessous. Toutes les machines utilisent le système d’exploitation « Windows 2012 Server EVAL ».

Le programme, comme mentionné précédemment, a été développé avec le langage de programmation « Java », et principalement à l’aide d’un framework de développement nommé « Spring ». Les données proviennent d'’une base de données créée sur le moteur « MariaDB ». Elles sont transformées dans le format « JSON » et sont stockées dans un répertoire afin de pouvoir les réutiliser sans les charger à nouveau. Cette opération de transformation qui comporte donc pour résumer, le requêtage, la création de l’objet « JSON » et l’écriture du fichier, peut durer plusieurs jours. Tout dépend de la quantité et de la puissance de la machine. Pour avoir une idée, il a fallu un peu moins de trois jours pour un gigaoctet de données.

Une fois les données « JSON » chargées, le programme les insère dans une collection partagée sur « MongoDB ». Une collection est, dans l’univers de « MongoDB », tout simplement une liste d’objets « JSON ».

Les rapports sont générés dans le format « pdf », contenant chacun un graphique créé par la librairie « jfreechart ».

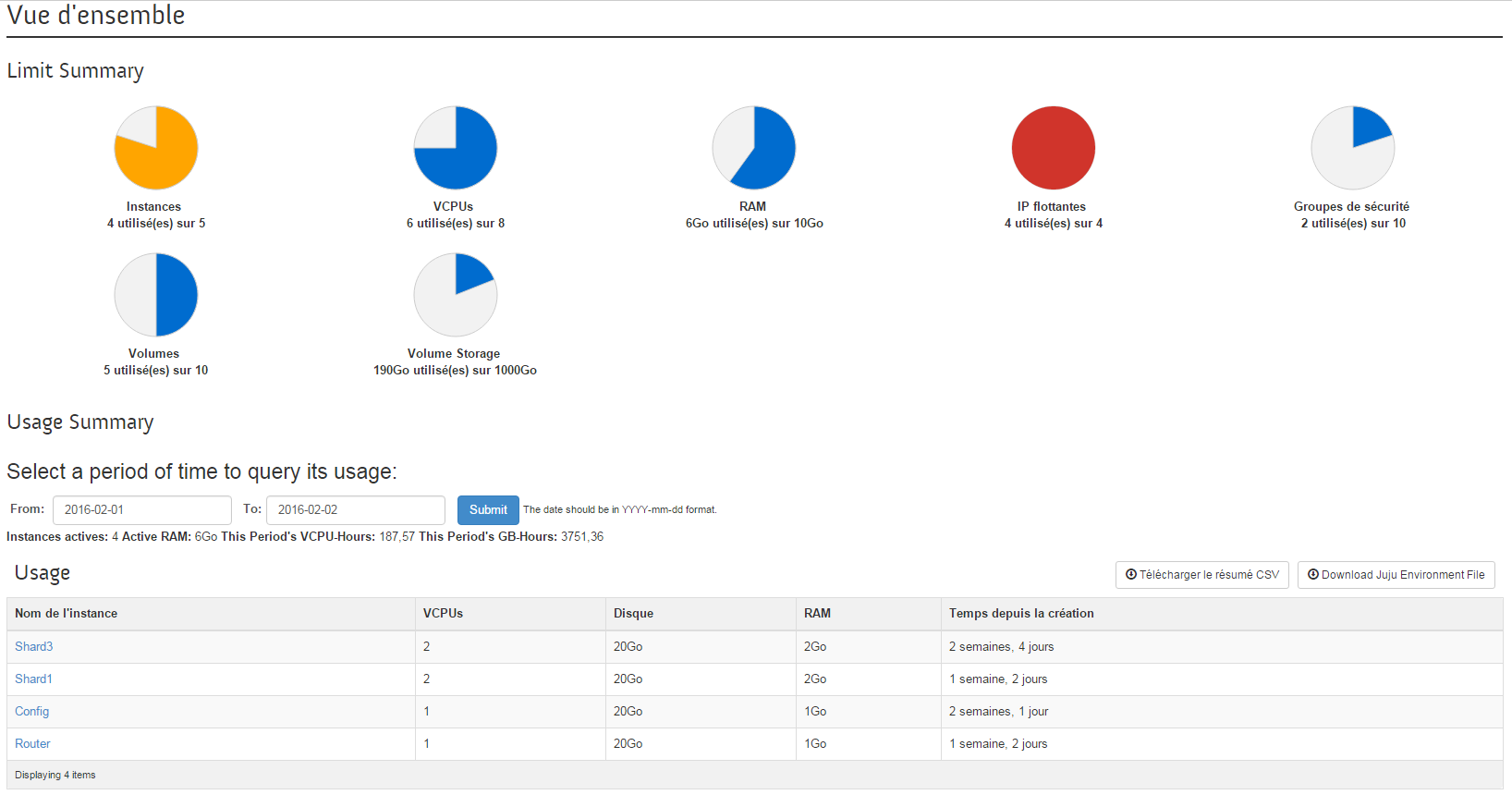
# SWITCH Engines

Comme mentionné précédemment, SWITCHengines permet de gérer des machines virtuelles à travers une application-web. La prise en main de l’application se fait assez facilement, et la navigation à travers les diverses vue de configuration est assez intuitive et rapide.

## Vue d’ensemble

Après s’être authentifié, nous accédons directement sur la vue d’ensemble des ressources disponibles et utilisées pour la création de nos instances (machines virtuelles), sous forme de camembert.

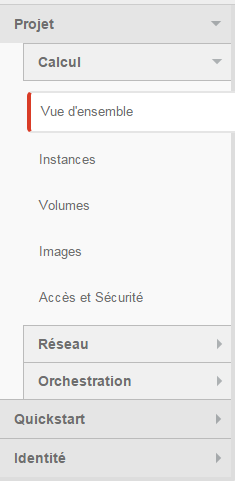
Figure  : Vue d’ensemble



## Menu

Par le biais du menu gauche, nous pouvons accéder sur d’autres pages. La plupart des configurations se font sur les vues « Instances », « Volumes », « Images » et « Accès et Sécurité ».

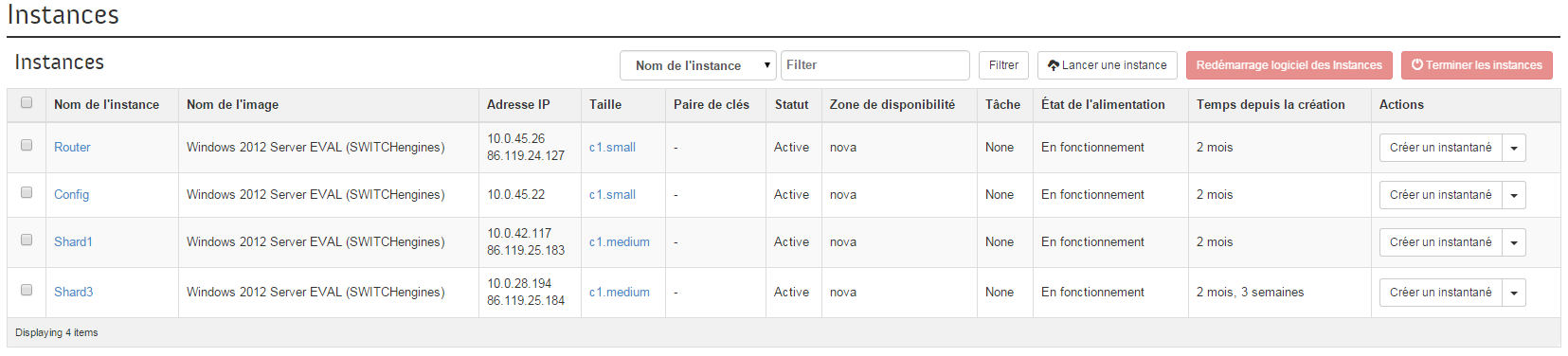
Figure  : Menu



## Vue des instances

A partir de cette page, il est possible de lancer diverses actions sur les instances, comme les éteindre, redémarrer, reconstruire, supprimer et associer une adresse « IP » externe.

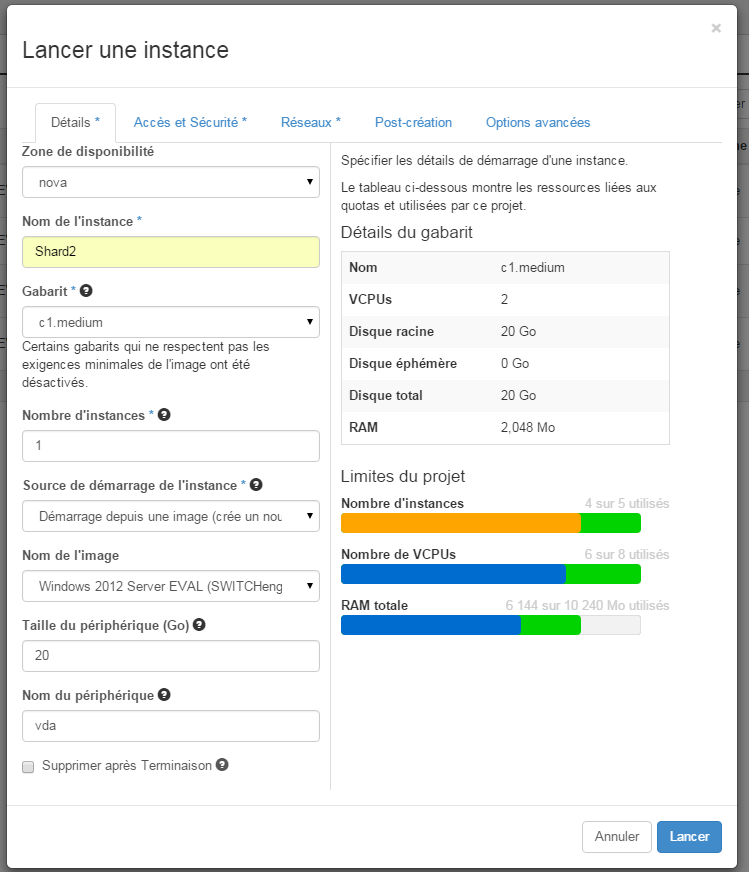
Figure  : Instances



## Création d’instance

La création d’une instance se fait assez facilement. Pour cela il faut la nommer, choisir son gabarit (les gabarits sont préconfigurés et permettent de choisir les ressources de l’instance), le nombre d’instance et l’image du système d’exploitation. Nous avons essayé de répartir les ressources disponibles de telle manière que les machines les plus sollicitées soit les plus performantes.

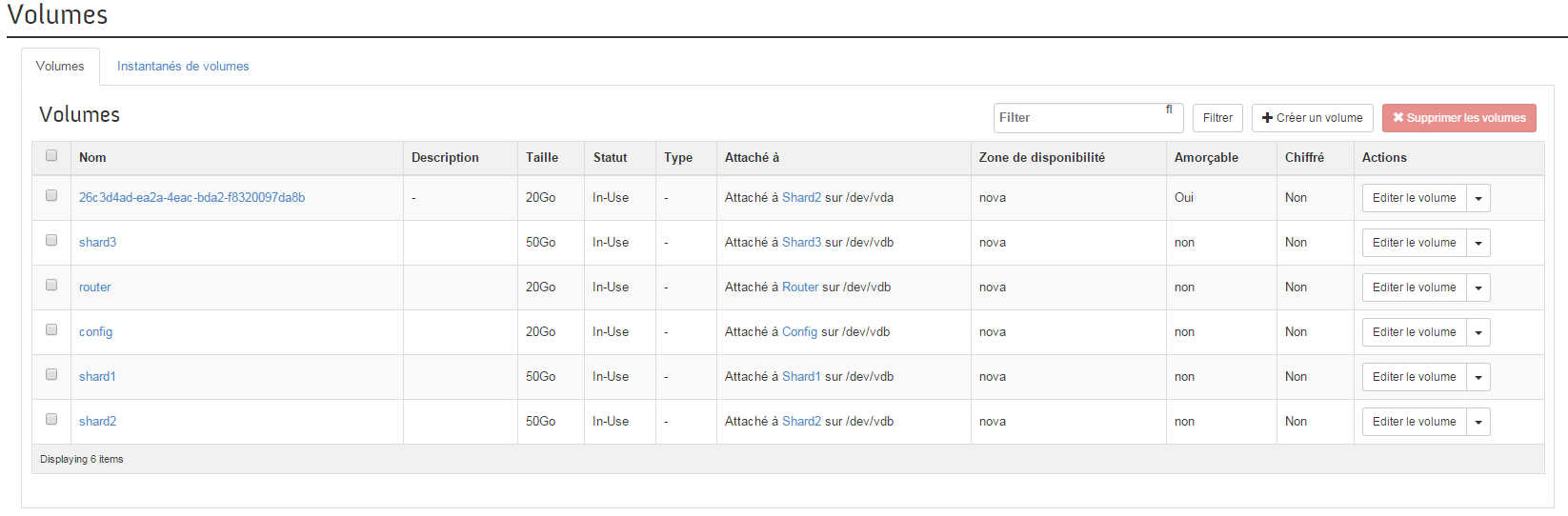
Figure  : Créer une instance



## Vue des volumes

La page des volumes permet de créer des volumes de données, et de les associer à des instances. Pour faciliter la reconfiguration des instances dans le cas où celles-ci pourraient défaillir, et ainsi ne pas perdre les données à chaque fois, j’ai préféré créer un volume pour chacune d’entre elles afin de contenir le nœud « MongoDB » et les divers scripts de lancement.

Figure  : Volumes

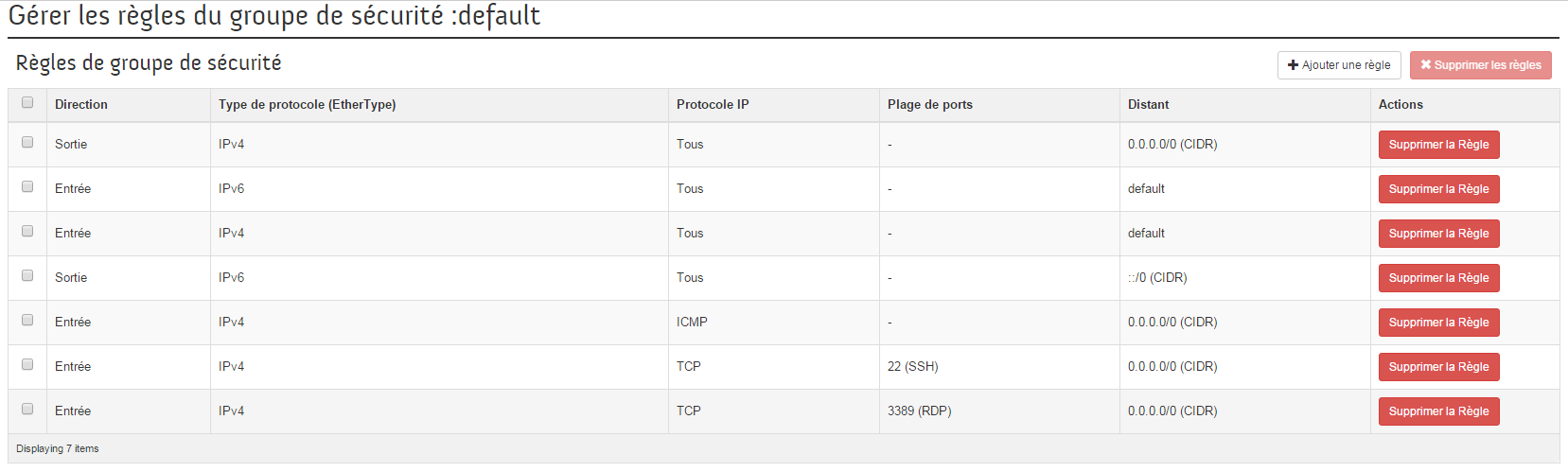


## 

## Vue des accès et de la sécurité

La sécurité et l’accès aux instances se configure via cette vue. Il est possible d’associer des adresses IP flottante ou de gérer les groupes de sécurité qui contiennent la configuration d’ouverture des ports. Cette configuration est importante pour pouvoir accéder en « Remote Access » aux instances.

Figure  : Accès et sécurité



# Démarche

Comme nous avons pu le mentionner précédemment dans l’introduction, le but de ce travail pratique est de mettre en évidence les performances d’insertions et de recherches de la base de données « MongoDB », dans un contexte d’architecture distribuée (au maximum trois machines).

Plus précisément, nous allons récupérer les temps accumulés des insertions pour pouvoir les comparer. Il y en a aura au total trois, un pour chaque machine ajoutée.

Sur le même principe, les temps accumulés seront également récupérés pour les recherches, à la différence près qu’il y aura un paramètre de plus à considérer : « le nombre de données déjà insérées ». Les sous-chapitres décriront en détail l’entièreté des processus.

Le processus est composé par les trois opérations suivantes :

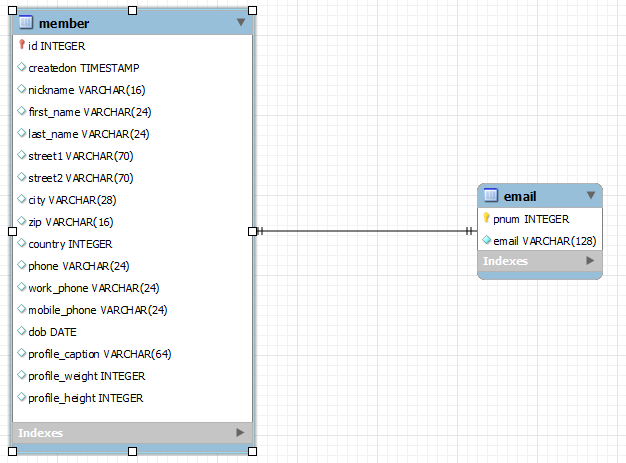
* Premièrement, l’extraction des données, la transformation en objet « JSON » et l’écriture des objets sur le système de fichiers.
* Deuxièmement, la récolte des valeurs de performances des insertions des objets et des recherches.
* Troisièmement, la génération des rapports.

## Opération d’extraction et de transformation

Cette opération est composée de deux phases, l’extraction et la transformation. Ces phases se font séquentiellement et en boucle jusqu’à atteindre la limite de taille totale configurée.

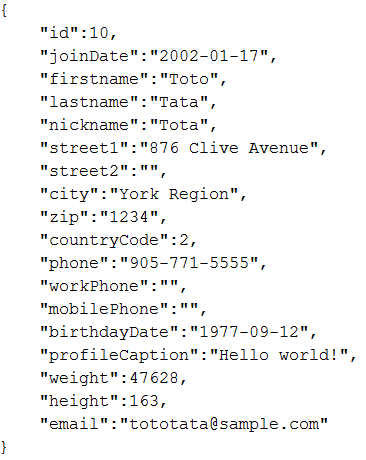
Afin que le résultat soit le plus conforme possible, il nous a fallu une grande quantité de données. Les données nous proviennent d’un dump s’exécutant sur la base de données « MariaDB », installée sur la machine « client ». Elle contient plusieurs tables, qui elles-mêmes contiennent plusieurs millions de données. Nous nous sommes limités à extraire les données d’une jointure entre deux tables.

Figure  : Relation entre tables



Une fois l’extraction terminée, pour chaque donnée, elle est transformée en objet « JSON », appelé « Member », contenant au total une vingtaine d’attributs. Au final l’objet sera écrit dans un fichier.

Figure  : Member « JSON »



Les fichiers sont regroupés par paquet de deux cents dans les divers dossiers. Le nom du fichier est l’identifiant de l’objet en base.

Cette opération peut être désactivée si les données ont déjà été extraites et transformées.

Figure  : Transfert des données

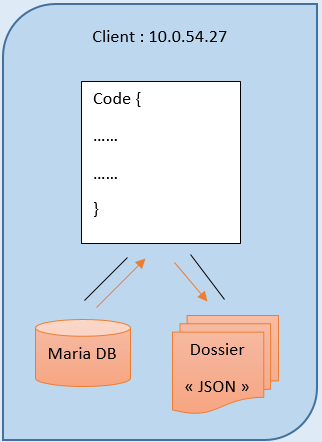


Figure : Dépôt

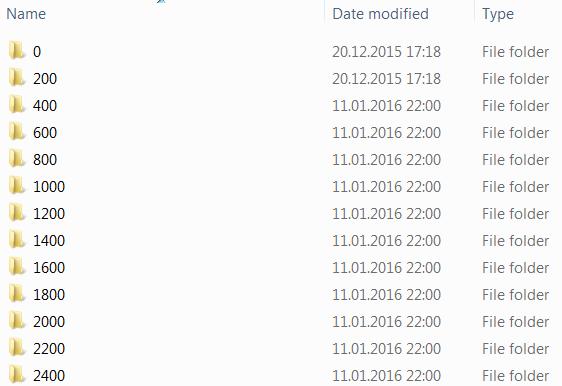
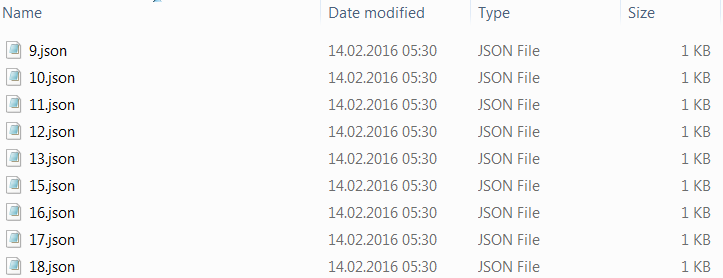


Figure  : Fichiers « JSON »

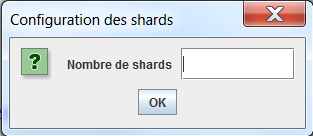


## Opération d’insertions et recherches

Une fois les données transformées, la phase d’insertions peut commencer. Le client va lire séquentiellement les fichiers « JSON » pour les insérer dans une collection partagée sur autant de machines qui font partie de l’infrastructure. Au passage, les temps effectués et le nombre de données insérées seront stockés en mémoire.

Avant le commencement de cette opération, un pop-up de dialogue s’affichera afin d’y insérer le nombre de machines ou shards.

Figure  : Popu-up - Nombre de shards



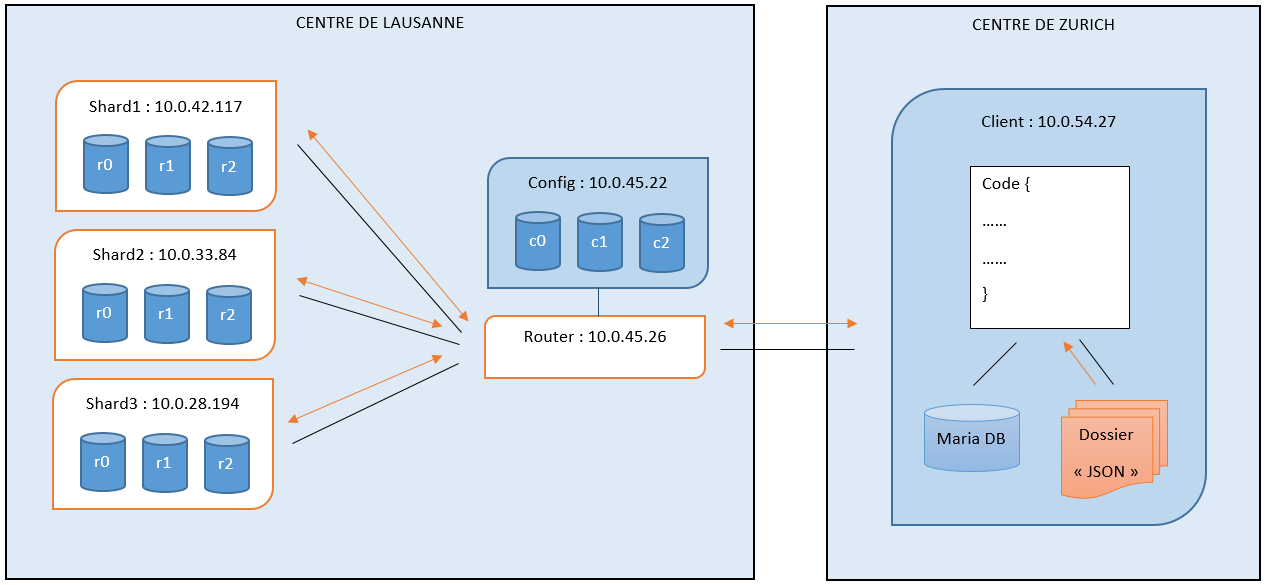
Une fois le nombre de machine inséré, l’opération peut commencer.

Après un tiers des données insérées, la phase d’insertion se mettra en attente, le temps d’exécuter la phase de recherche par l’attribut « email ». Pour cette phase, les temps effectués et le nombre de requêtes seront également stockés en mémoire. Une fois la phase de recherche terminée, la phase d’insertion continuera son exécution jusqu’au deux tiers, pour laisser à nouveau place à la phase de recherche. Et ceci se répétera une dernière fois, après la fin des insertions (trois tiers).

L’opération terminée, le pop-up s’affichera à nouveau, mettant en attente le processus. Ceci permet d’avoir le temps de reconfigurer et d’ajouter un nouveau nœud à l’architecture. Cette opération sera joué trois fois, afin de tester une architecture contenant une machine, ensuite deux machines, et pour finir trois machines.

Les données de performance sont ensuite stockées dans des fichiers « csv ».

Figure  : Insertions et recherches

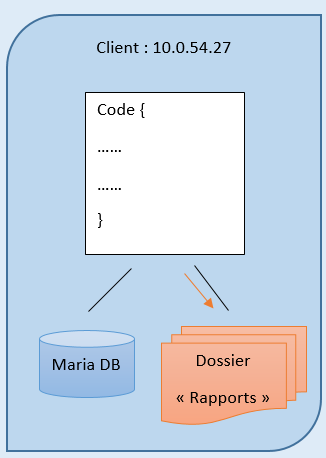


## Opération de génération des rapports

Les graphiques sont générés à partir des données de performance qui sont stockées dans les fichiers « csv ». Cela permet à tout moment de les générer indépendamment des autres opérations.

A partir de ces fichiers, sept graphiques en ligne sont générés. Le premier concerne les insertions et les six autres les recherches. Dans le chapitre « Résultats », nous décrirons chaque graphique généré.

Figure  : Rapports



# Résultat

## Graphiques

Suite à la génération des graphiques, nous pouvons maintenant comparer le temps de réponse accumulé des requêtes.

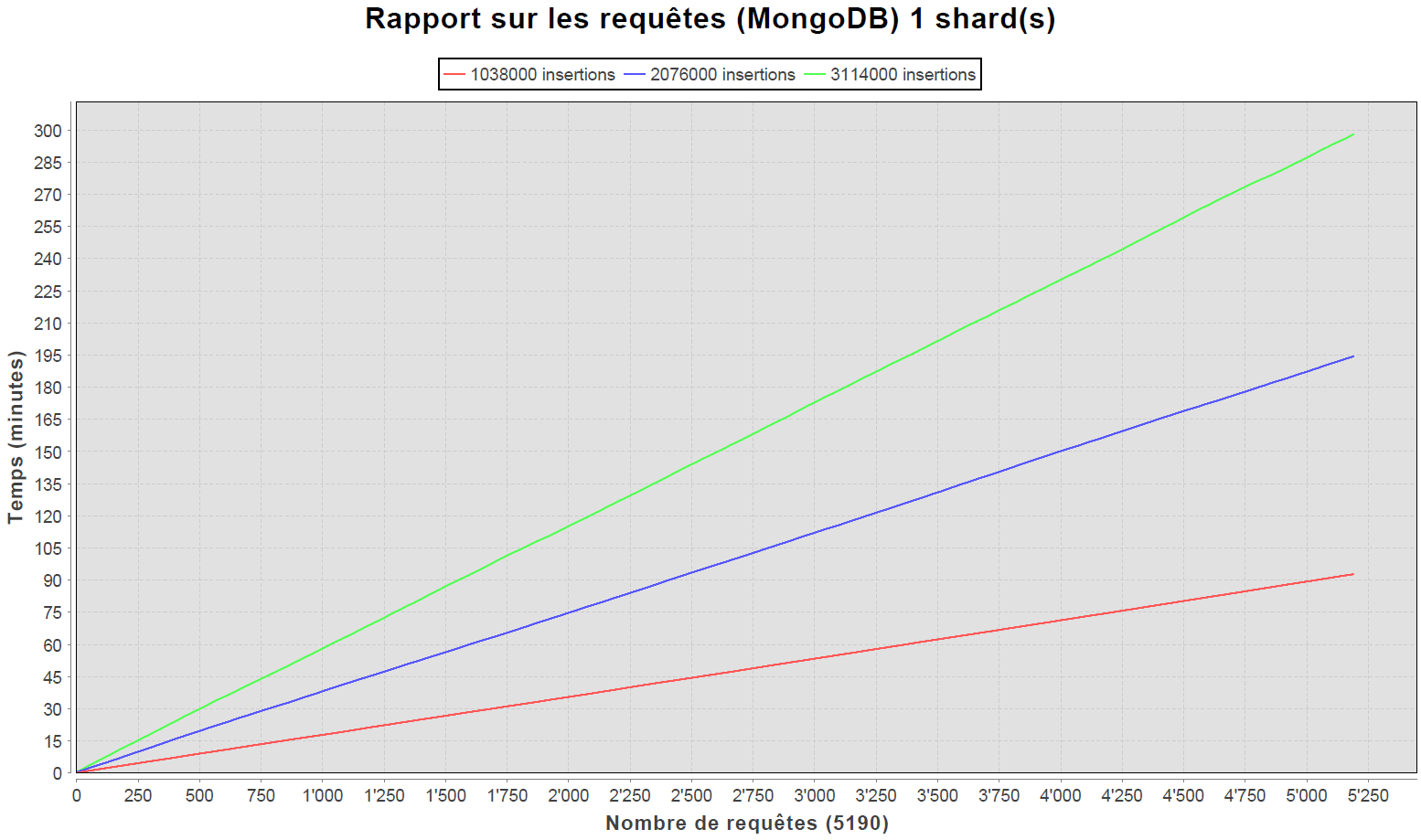
Pour tous les graphiques, l’axe « X » représente le nombre de requêtes, et l’axe « Y » le temps en minutes.

Les graphiques de recherches sont composés de deux jeux de trois graphiques.

Commençons par analyser le premier jeu. Les graphiques sont générés en fonction du nombre de machines dans l’infrastructure. Il en y a donc un pour chaque machine ajoutée.

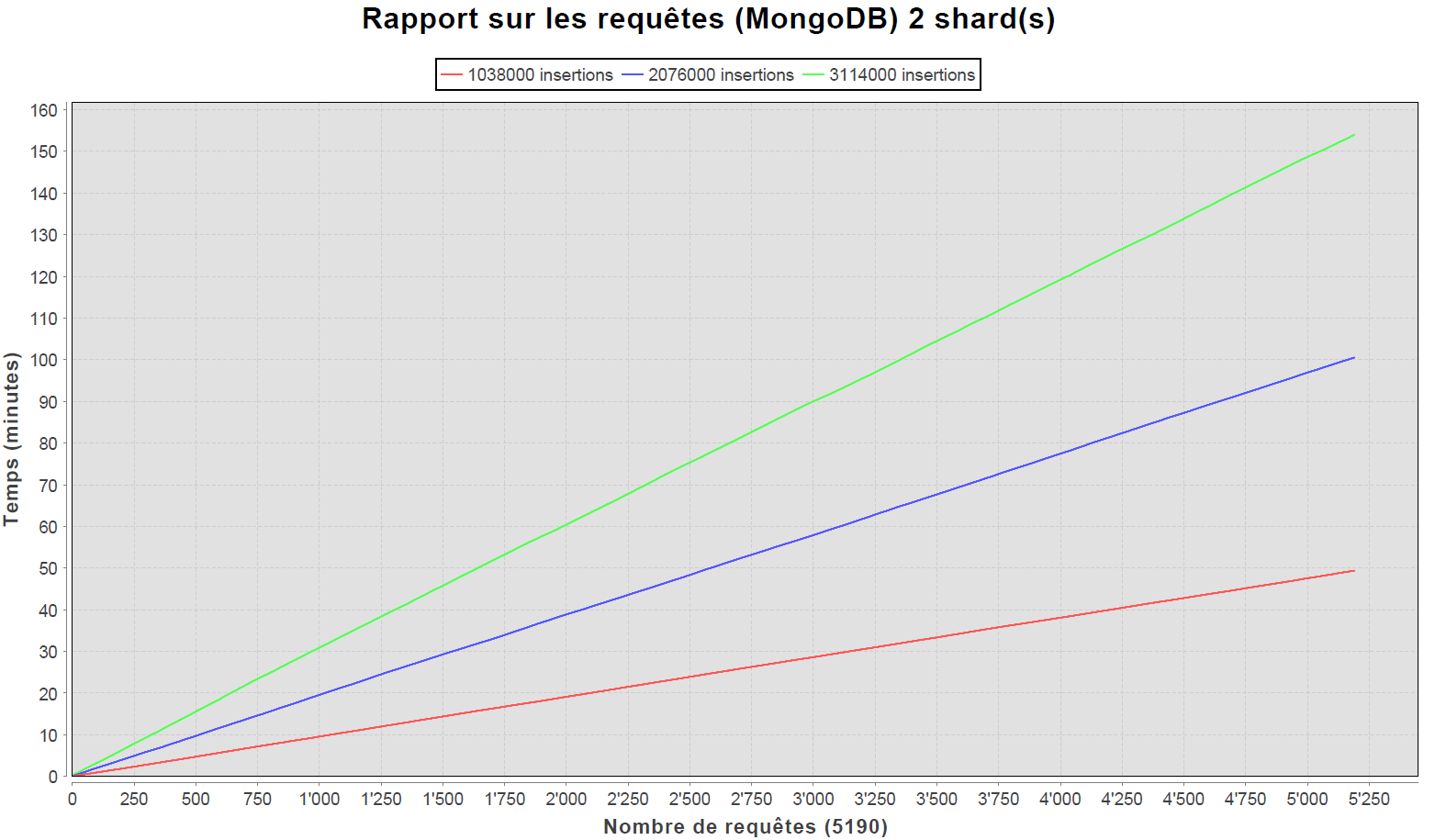
Le premier graphique met en évidence la différence de temps cumulé en fonction du nombre d’objets déjà existants dans la collection pour une machine (1 shard).

Figure  : Recherches (1 machine)



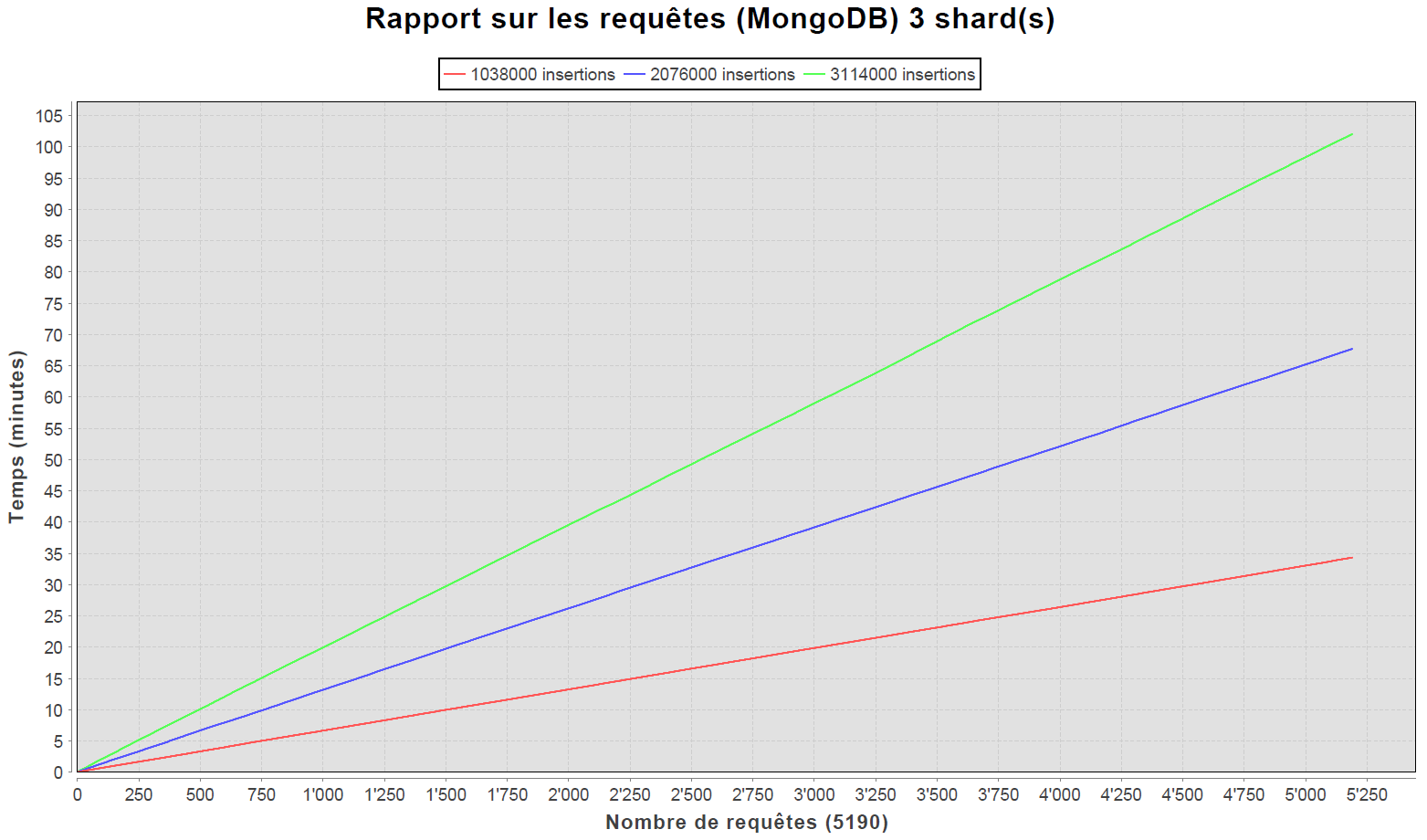
Le deuxième pour deux machines (2 shards).

Figure  : Recherches (2 machines)



Et le dernier sur trois machines (3 shards).

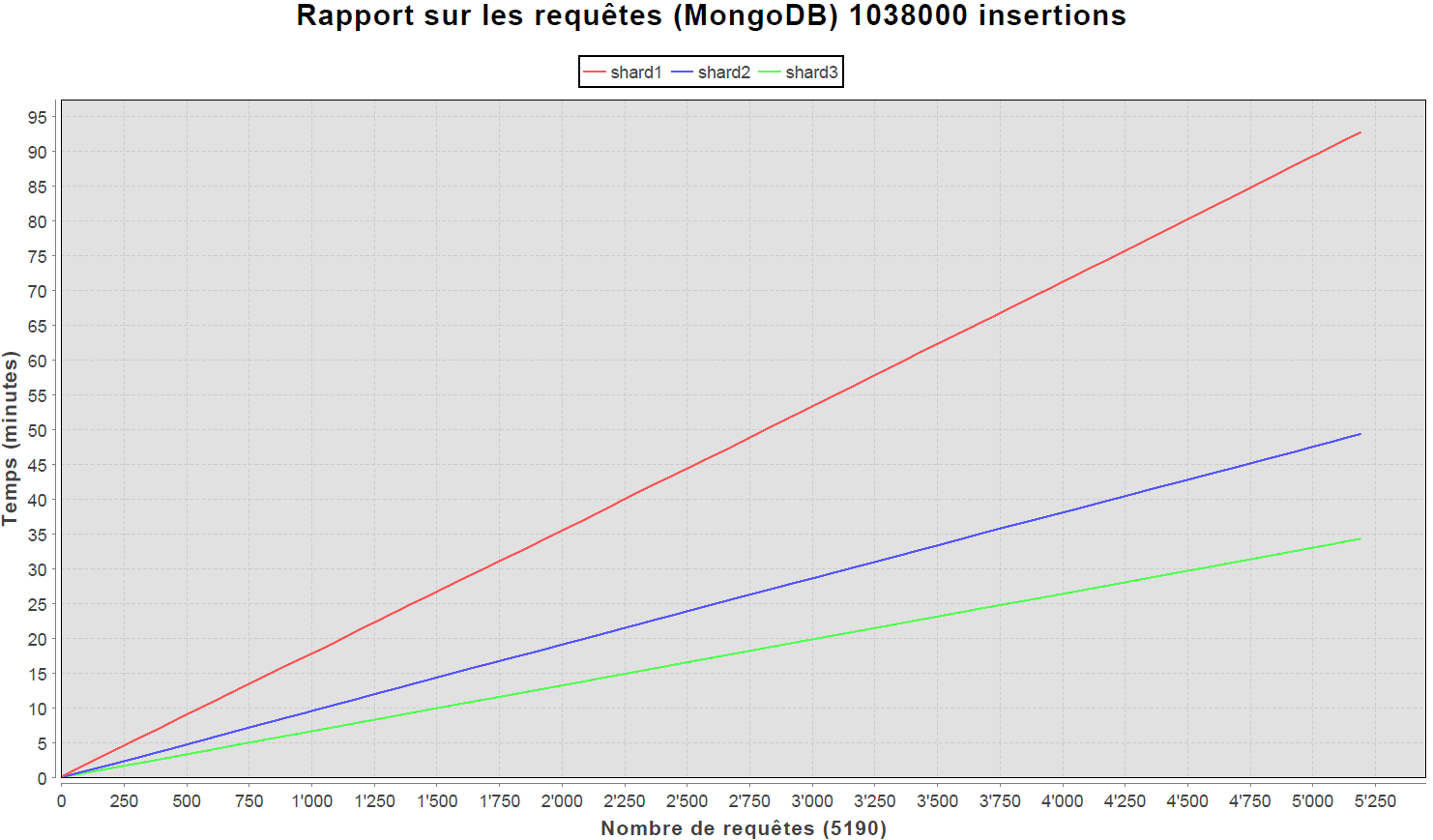
Figure  : Recherches (3 machines)



Pour le deuxième jeu de graphique, Les graphiques sont générés en fonction du nombre d’objets déjà existants dans la collection. Il en y a donc un pour chaque tiers d’objets insérés.

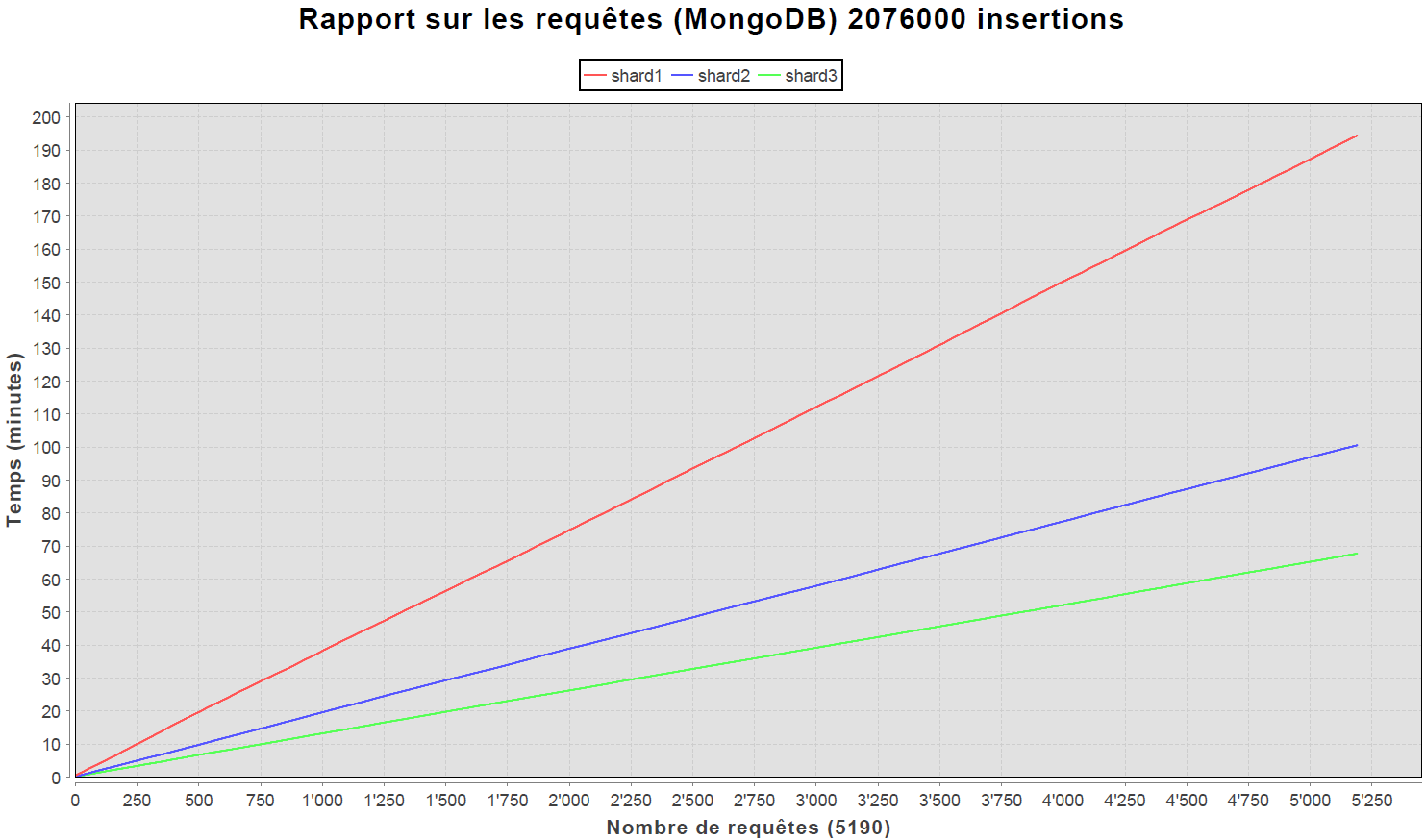
Le premier graphique met en évidence la différence de temps cumulé en fonction du nombre de machines, pour 1'038'000 objets déjà existants dans la collection.

Figure  : Recherches (1'038'000 objets)



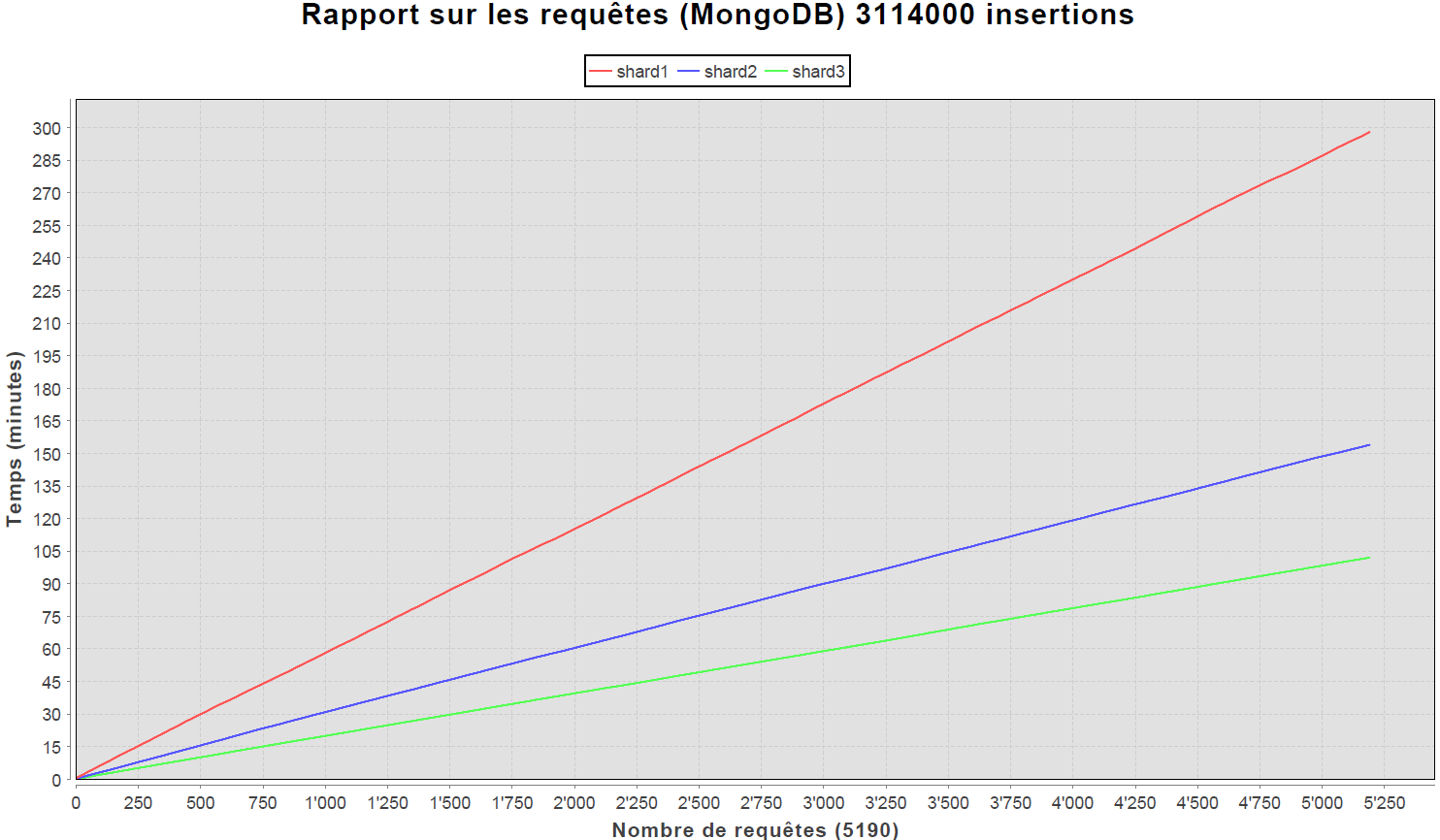
Le deuxième pour 2'076’000 objets déjà existants dans la collection.

Figure  : Recherches (2'076'000 objets)



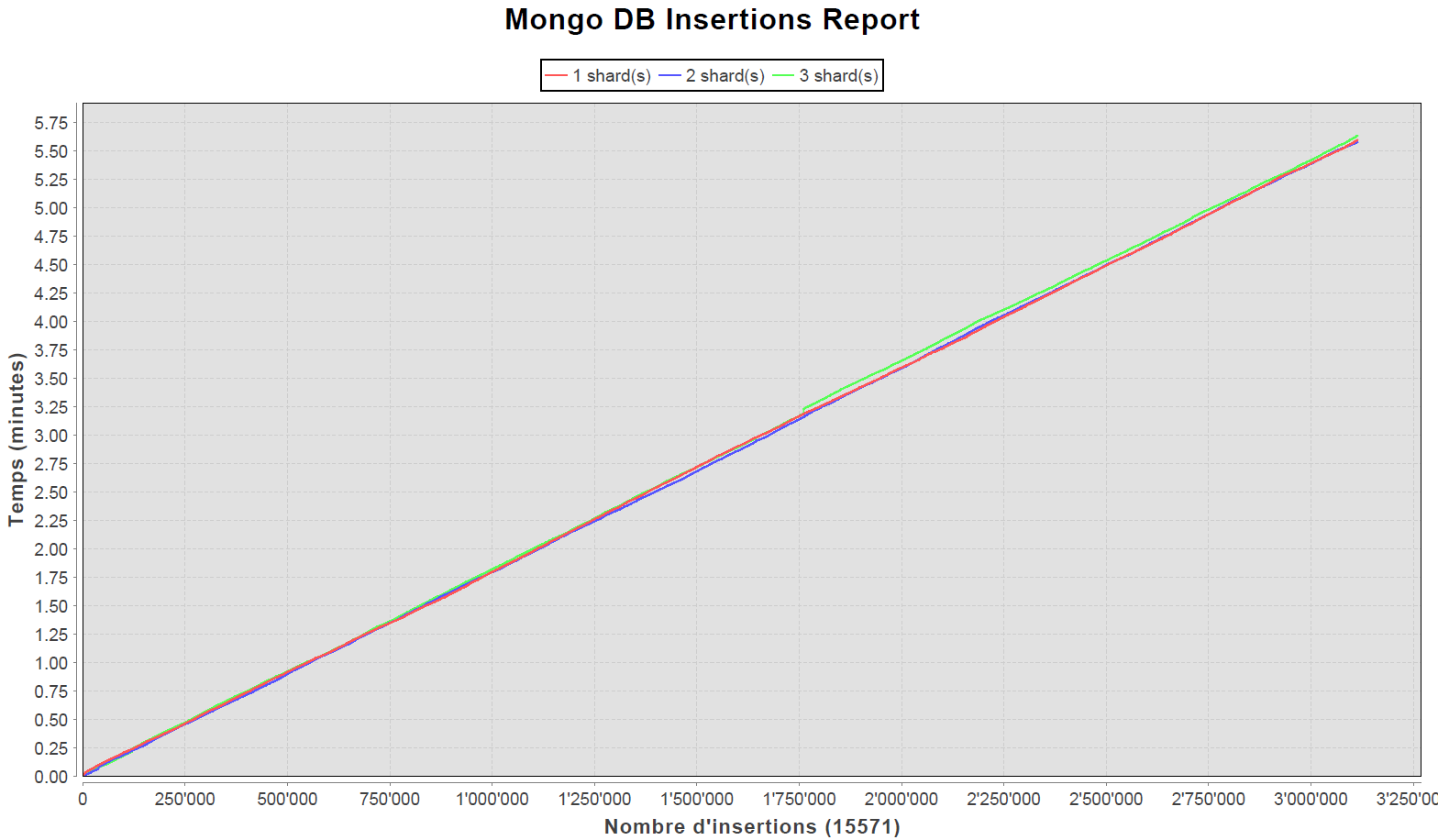
Le deuxième pour 3'114’000 objets déjà existants dans la collection.

Figure  : Recherches (3'114'000 objets)



Pour les insertions, il n’y a qu’un seul graphique qui représente le temps accumulé en fonction du nombre de machines.

Figure  : Insertions



## Analyse et constations

La première chose qui nous saute aux yeux lorsqu’on regarde ces sept graphiques, est la constance des lignes. Elles sont bien droites. On peut en conclure que les fichiers « JSON » sont tous pratiquement de la même taille.

Deuxièmement, les lignes sont bien séparées pour les recherches tandis qu’elles sont pratiquement identiques pour les insertions.

Voici ci-dessous un tableau mettant en valeur le temps cumulé total des recherches. On remarque de suite que le temps accumulé diminue en fonction du nombre de machines et du nombre d’objets déjà contenus dans la collection.

Tableau  : Recherches

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| nombre de machines | Nombre d’objets déjà existants | | | |
|  | **1038000** | **2076000** | **3114000** |
| **1 machine** | 91 min. | 195 min. | 300 min. |
| **2 machines** | 50 min. | 100 min. | 152 min. |
| **3 machines** | 35 min. | 58 min. | 102 min. |

On peut affirmer approximativement :

* Le temps total double si le nombre de données existantes double.
* Le temps total pour une machine est partagé par le nombre de machines ajoutées.

# Conclusion

Tout d’abord, ma conclusion s’appuie sur les données de performance relevées, et sur l’infrastructure mise en place. Il est tout à fait possible que les données de performance soient différentes en fonction d’autres paramètres, comme la puissance des machines, la vitesse de la bande passante, le fait d’avoir plusieurs clients et pas qu’un seul, ou la taille des objets dans une collection.

Par rapport à mes données, il est clair que les performances d’insertions ne varient pas en fonction des machines. On voit clairement que les lignes sont pratiquement identiques.

Par contre, on remarque nettement que pour les requêtes de recherche, les performances varient en fonction de deux paramètres, le nombre de machines et le nombre d’objets déjà existants dans une collection. Vu la différence de temps cumulé total, il est clairement conseillé d’avoir une architecture distribuée pour gagner en performance.

Bibliographie

BRUCHEZ, Rudi, 2015. Les bases de données NoSQL et le Big Data (2ème édition), Paris : Eyrolles. ISBN 978-2-212-14155-9.

McCREARY, Dan et KELLY, Ann, 2014. Making Sense of NoSQL, Shelter Island : Manning. ISBN 9781617291074.

[1]. BRUCHEZ, Rudi, 2015. Les bases de données NoSQL et le Big Data (2ème édition), Partie I, Qu’est-ce qu’une base NoSQL ?, Paris : Eyrolles. ISBN 978-2-212-14155-9.

[2]. BRUCHEZ, Rudi, 2015. Les bases de données NoSQL et le Big Data (2ème édition), Chapitre I, Edgard Frank Codd, Paris : Eyrolles. ISBN 978-2-212-14155-9.

[3]. FORBES, 2013, A Very Short History Of Big Data. [Consulté le 20 juillet 2015] Disponible à l’adresse : <http://www.forbes.com/sites/gilpress/2013/05/09/a-very-short-history-of-big-data/#210b5b9655da>

[3]. WIKIPEDIA, 2015, Big Data. [Consultée le 20 juillet 2015] Disponible à l’adresse : <https://fr.wikipedia.org/wiki/Big_data>

[4]. BRUCHEZ, Rudi, 2015. Les bases de données NoSQL et le Big Data (2ème édition), Chapitre I, BigTable, encore Google !, Paris : Eyrolles. ISBN 978-2-212-14155-9.

[5]. BRUCHEZ, Rudi, 2015. Les bases de données NoSQL et le Big Data (2ème édition), Chapitre I, L’influence d’Amazon, Paris : Eyrolles. ISBN 978-2-212-14155-9.

[6]. BRUCHEZ, Rudi, 2015. Les bases de données NoSQL et le Big Data (2ème édition), Chapitre II, Les structures de données, Paris : Eyrolles. ISBN 978-2-212-14155-9.

[7]. BRUCHEZ, Rudi, 2015. Les bases de données NoSQL et le Big Data (2ème édition), Chapitre II, Les principes du relationnel en regard du NoSQL, Paris : Eyrolles. ISBN 978-2-212-14155-9.

[7]. McCREARY, Dan et KELLY, Ann, 2014. Making Sense of NoSQL, Part 1, 2.5 Comparing ACID and BASE – two methods of reliable database transactions, Shelter Island : Manning. ISBN 9781617291074.

[8]. BRUCHEZ, Rudi, 2015. Les bases de données NoSQL et le Big Data (2ème édition), Chapitre II, Le théorème CAP, Paris : Eyrolles. ISBN 978-2-212-14155-9.

[8]. McCREARY, Dan et KELLY, Ann, 2014. Making Sense of NoSQL, Part 1, 2.7 Understanding trade-offs with Brewer’s CAP theorem, Shelter Island : Manning. ISBN 9781617291074.

DOMENJOUD, Michel, 2012. Base de données graphes : un tour d’horizon. [Consultée le 15 mars 2016] Disponible à l’adresse : <http://blog.octo.com/bases-de-donnees-graphes-un-tour-dhorizon/>

Loïc Guillois, 2011, Mettre en place le sharding et le failover avec MongoDB. [Consultée le 15 septembre 2015] Disponible à l’adresse : [http://www.loicguillois.fr/mettre-en-place-le-sharding-et-le-failover-avec-mongodb](https://fr.wikipedia.org/wiki/Th%C3%A9or%C3%A8me_CAP)

E. Renaux, 2012, NoSQL. [Consultée le 2 mars 2016] Disponible à l’adresse : [https://manurenaux.wp.mines-telecom.fr/2012/04/18/nosql/](https://fr.wikipedia.org/wiki/Th%C3%A9or%C3%A8me_CAP)