Master II Sciences de Données et Systèmes Complexes

Traitement et données réparties

Benchmark Système NoSQL

Étudiants:
Maxime MAIRE
Matheo VLAYKOV

Professeur:
Gabriel FREY

1 Introduction

Dans le cadre de notre cursus de Master 2 en Science des Données et Systèmes Complexes (SDSC), nous avons eu l'opportunité de travailler sur un projet dans l'UE **Traitement et données réparties**.

L'objectif de ce projet est de présenter et expliquer un système NoSQL choisi et de le comparer à un système SQL à l'aide d'une évaluation de performance sur les deux systèmes. L'impact sur les performances lors d'une indexation est aussi le sujet du projet. Nous avons choisi Redis comme système NoSQL et SQLite pour notre base de données relationnelle.

L'installation peut se faire en lisant le fichier README.md.

2 Présentation du système NoSQL choisi

Redis (REmote DIctionary Server) est un système de gestion de base de données NoSQL écrit en C et de type clé-valeur performant, polyvalent et open source. Il excelle dans le traitement rapide de données en mémoire.

Il prend en charge diverses structures de données telles que les chaînes, les ensembles, les listes, les hachages et les ensembles ordonnés, offrant ainsi une flexibilité considérable pour la modélisation des données. De plus, Redis propose des fonctionnalités avancées telles que la persistance des données (Snapshots) sur le disque, la réplication, les transactions (atomique avec MULTI et EXEC) et la gestion des clés expirées (SET ... EXPIRE ...).

Sa simplicité d'utilisation et sa rapidité en font un outil populaire pour les applications nécessitant une manipulation efficace des données en temps réel.

2.1 Avantages et inconvénients du système NoSQL choisi

Avantages de Redis:

- Redis permet de récupérer des données de manière très rapide, puisque celles-ci sont stockées dans la **mémoire vive** → pratique pour des services demandant une grande disponibilité. **Redis Sentinel** permet de renforcer cette disponibilité. En effet, il s'agit d'un système permettant de surveiller notre système Redis et de nous informer en cas de problème, voire même de créer des nouvelles connexions pour pouvoir tout de même accéder aux données.
- La **scalabilité**, qu'elle soit horizontale et verticale, est tout à fait faisable avec Redis et l'outil Redis Cluster.
- Afin de mieux gérer la RAM disponible sur le serveur Redis, il est possible d'utiliser un système de gestion qui transfère certaines données sur un disque dur.
- Bien qu'il s'agisse d'un système NoSQL, son architecture lui permet de faire des transactions **ACID**:
  - Atomicity et Consistency: avec Lua, qui permet d'écrire des scripts pour Redis.
  - Isolation: toujours garantie.

– Durability: lorsque la technique du AOF (Append-Only File, qui ne permet que d'ajouter des nouvelles données et pas d'en modifier) est activée.

Il se distingue des autres systèmes NoSQL par son stockage des données dans la mémoire principale du serveur Redis. Certaines tâches prenant plus de temps qu'habituellement peuvent également être mise dans des **files d'attente**. De plus, bien que Redis ne soit qu'un système clé-valeur, il permet de gérer une multitude de types de données. Avec actuellement plus d'une **centaine de clients open-source** compatibles avec divers types de langages de programmation, Redis peut facilement s'adapter aux besoins de tout un chacun. Et si cela ne suffit pas, il y a possibilité d'ajouter des intégrations dans notre client, ce qui fait de Redis un outil extrêmement polyvalent et adaptable aux besoins.

Inconvénients de Redis:

- La rapidité de Redis a un prix: son coût en performance, particulièrement en mémoire vive.
- Comme chaque donnée ne peut être récupérée que par sa clé, il est difficile d'effectuer des opérations complexes sur des données gérées avec Redis.
- Même si Redis permet de traiter des données de grandes tailles, il faut toutefois faire attention à ce que celle-ci ne soit pas "excessivement" (dépendant du système) grande, car les données sont tout de même stockées en RAM.

2.2 Use-case du système NoSQL choisi

La mémoire d'une clé peut atteindre 512MB, ce qui permet de traiter de relativement grosses données.

Use-case: Twitter

Twitter reçoit plus de 300 000 requêtes de lecture par seconde. Il y a donc une nécessité de bien gérer les fils d'actualités des utilisateurs, par lequel passe une grande partie des demandes en lecture. Ces fils sont en réalité préparés à l'avance afin de pouvoir fournir immédiatement des données prêtes à être regardées.

Ainsi, lors de l'écriture (il y en a 6000 par seconde) d'une personne x, il faut pouvoir actualiser le fil d'actualité de tous les followers de cette personne x. Cela passe par la reconstruction du fil d'actualité de tous les followers. Twitter s'affranchit ainsi de traitements plus dynamiques qui seraient trop coûteux. Bien sûr, actualiser le fil d'actualité de nombreux followers nécessite tout de même des calculs coûteux, si bien que tout les fils d'actualité des utilisateurs actifs sont stockés dans une machine redis de plusieurs terabytes de RAM.

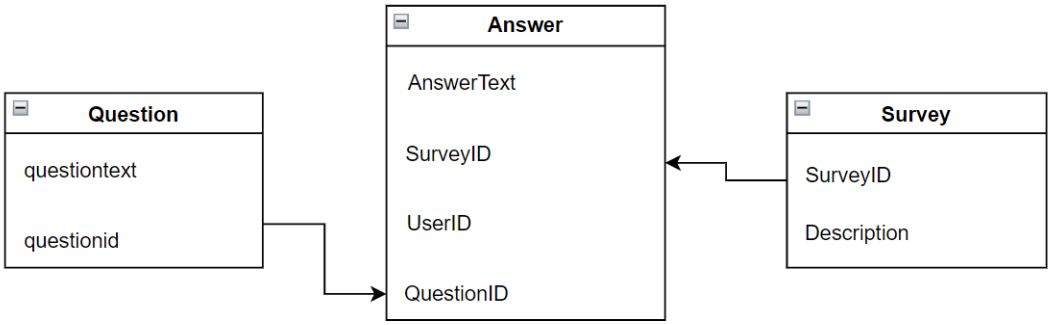
D'autres sites à fortes influences utilisent Redis, comme Github, The Guardian, Stack Overflow, Pinterest, Netflix, Medium, ...

Il est donc très utilisé pour du temps réel, mais aussi pour des applications utilisant des données de localisation, en permettant une indexation de celles-ci, dans le streaming de vidéos (à la fois pour les profils des utilisateurs et des vidéos à charger), pour le Machine Learning, les analyses en temps réel, les magasins de session, les messageries instantanées (avec **Redis Pub/Sub**), ...

2.3 Jeu de données utilisé

Le jeu de données que nous avons utilisé a été trouvé sur Kaggle. Il s'agit du jeu de données Mental Health in the Tech Industry, qui est une base de donnée SQLite composée de 3 différentes tables :

- **Answer :** Answertext (str), SurveyID (int), UserID (int), QuestionID (int)
- **Question :** questiontext (str), questionid (int)
- **Survey :** SurveyID (int), Description (str)



Schema 1: Schéma de la base de données mental_health.sqlite

La table Survey contient l'année d'un sondage ainsi qu'une description rapide de ce sondage. Elle contient 5 lignes. La table Question contient l'id des questions posées aux répondants ainsi que l'intitulé de la question. Elle contient 105 lignes. La table Answer contient l'id de la question, l'id du sondage, l'id de la réponse ainsi que la réponse en tant que telle. Elle contient 236 898 lignes.

2.4 Environnement de travail utilisé

Les programmes ont tous été lancés sur le même PC, dans les mêmes configurations (aucun programme ouvert mis à part Visual Studio Code (v. 1.85.1) et le Jupyter Notebook qui devait tourner).

Les informations concernant le PC sont les suivantes:

- Modèle de l'ordinateur: Huawei MateBook D 16 AMD
- Processeur: AMD® Ryzen 5 4600h with radeon graphics × 12
- Mémoire vive: 16.0 Go
- Stockage: 512 Go
- Système d'exploitation et version: Ubuntu 22.04.3 LTS

Plus d'informations sur le PC: ici

Version de Python: Python 3.10.12

Version de Redis-server: 6.0.16

Différentes versions des composants de Jupyter:

- IPython : 8.12.0
- ipykernel : 6.22.0
- jupyter_client : 8.2.0
- jupyter_core : 5.3.0
- jupyter_server : 2.5.0
- nbclient : 0.7.3
- nbconvert : 7.3.1
- nbformat : 5.8.0
- notebook : 6.5.4
- traitlets : 5.9.0

3 Tâches effectuées

3.1 Construction de la base de données Redis

Afin de faire correspondre notre base de données Redis à notre base de données SQLite et d'en créer un équivalent, nous avons faire un join sur chacune de nos trois tables, récupérant ainsi 236 898 résultats (ce qui correspond au nombre exact de lignes dans la table Answer). Puis, nous avons transformés chacun des résultats en string. Ainsi, chaque ligne de notre base de données Redis avait un index i pour clé et une valeur correspondant à une ligne de la base de données **mental_health.sqlite** jointe.

3.2 Première expérience

La première expérience consistait à observer la **variation du temps d'exécution de différentes opérations (insertion, récupération, modification et suppression) en fonction de la taille d'origine de la base de données**. Pour cela, la taille de la base de données a été modifiée en lui donnant une taille d'origine de 2 000, 25 000, 50 000, 75 000, 100 000, 125 000, 150 000, 175 000 puis 200 000 données. Après cela, nous avons effectués 1000 fois la même opération sur Redis puis sur SQLite et nous avons calculés la moyenne du temps pour une opération. A noter que les bases ont été recrées de zéro avec pour modèle le fichier mental_health.sqlite. En effet, la table Answer contenant déjà plus de 200 000 lignes à elle seule, celle-ci aurait faussé les résultats pour des tables préremplies de 2000 données par exemple.

Les opérations sur Redis ont été effectuées de la façon suivante: une insertion, récupération, modification, suppression sur Redis se faisait à partir d'une clé et l'on effectuait notre opération directement sur tout le résultat. En revanche, les opérations sur SQLite ont été effectuées différemment. En effet, SQLite utilisant un système de table, l'équivalent d'une opération avec Redis aurait nécessité de toucher à une donnée dans deux tables différentes (la table Survey n'a pas été touchée) sur SQLite. De ce fait, nous avons calculés un temps d'opération sur la table

Answer et un temps d'opération sur la table Question. Enfin, nous avons ressorti un temps total, qui correspondait à l'addition des deux. Ce choix a été motivé par une raison: comme une modification sur Redis pouvait en réalité toucher plusieurs tables sur SQLite, il fallait rester authentique et calculer le temps nécessaire pour effectuer exactement la même opération sur SQLite que celle effectuée sur Redis. Néanmoins, le projet consiste à faire des mesures de performance, si bien que nous avons estimés que le temps nécessaire à un type d'opération sur une seule des deux tables était tout aussi valable (et nous avons donc aussi gardés les temps pris par les opérations sur chacune des deux tables).

3.3 Deuxième expérience

La même expérience que précédemment a été réalisée à la différence près que cette fois-ci, l'**indexation** a été ajoutée au système SQLite avec l'aide de PRIMARY KEY sur SurveyID dans la table Survey, sur QuestionID dans la table Question et sur SurveyID, UserID et QuestionID dans la table Answer.

3.4 Troisième expérience

Cette troisième expérience visait à voir **l'évolution du temps d'exécution de plusieurs types d'opérations (insertion, sélection, mise à jour, suppression) réalisées en une seule instruction**. L'abscisse représente le nombre d'opérations réalisées en une seule instruction (100, 500, 1 000, 2 500, 5 000, 10 000, 25 000, 50 000 , 100 000). L'ordonnée représente le temps total mis au programme pour réaliser ces opérations (et non pas la moyenne comme dans les deux premières expériences).

Les choix des temps retenus expliqué dans le deuxième paragraphe de la partie "3.2 - Première expérience" ont également été adoptés dans cette partie.

4 Résultats

4.1 Première expérience

La Figure 1 et la Figure 2 correspondent aux résultats d'exécution de **sqlite_redis_no_index.ipynb**. Les résultats sont visibles respectivement dans les fichiers **graph_redis_no_index.html** et **graph_sqlite_no_index.html**.

Pour les deux figures: sur l'axe des abscisses, on retrouve **nb_data**, qui correspond au nombre de données préalablement insérées dans la base de données. Sur l'axe des ordonnées, on retrouve le temps moyen pris par chaque type d'opération (insertion, récupération, modification et suppression).

Par exemple, 1000 modifications ont été effectuées sur la base de données préremplies de nb_data données. On récupère le temps pour chaque modification effectuée, puis on calcule la moyenne et l'écart-type de l'ensemble des modifications. L'écart-type correspond à l'intervalle de confiance du temps pris par les types d'opérations. Ici, seul la taille originelle de la base de donnée change.

Le temps d'exécution a été de 7 minutes et 37.7 secondes pour le programme principal puis de 5.1 secondes et 0.8 seconde pour créer les visualisations.

La première visualisation a mis plus de temps à être créer que la seconde. En effet, lors de la génération des visualisations, celles-ci tentent de s'afficher sur notre navigateur par défaut et vont l'ouvrir, ce qui va fortement ralentir le processus. Un test a été réalisé en lançant le programme tandis que Firefox était également ouvert et la création des deux visualisations ont pris 0.6 seconde (ceci a également été vérifié pour les expériences suivantes).

Nos graphes ci-dessous sont de la forme suivante: la moyenne est une courbe nette, tandis que l'écart-type pour une courbe correspond à la zone floutée de la même couleur.

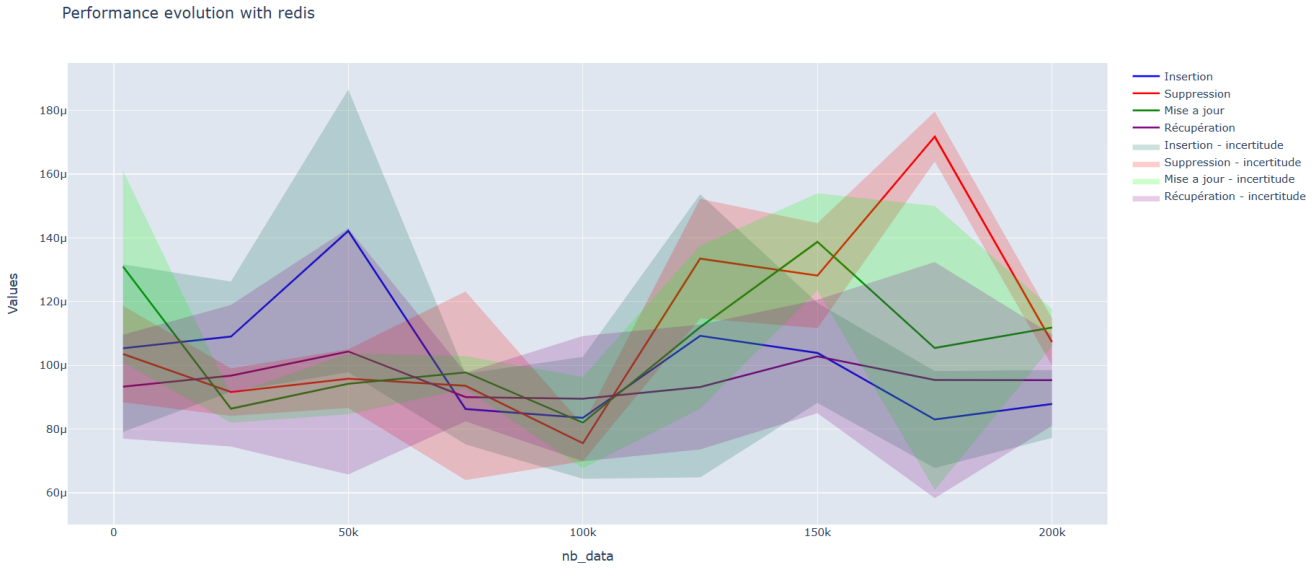


Figure 1: Graphe Redis sans indexation

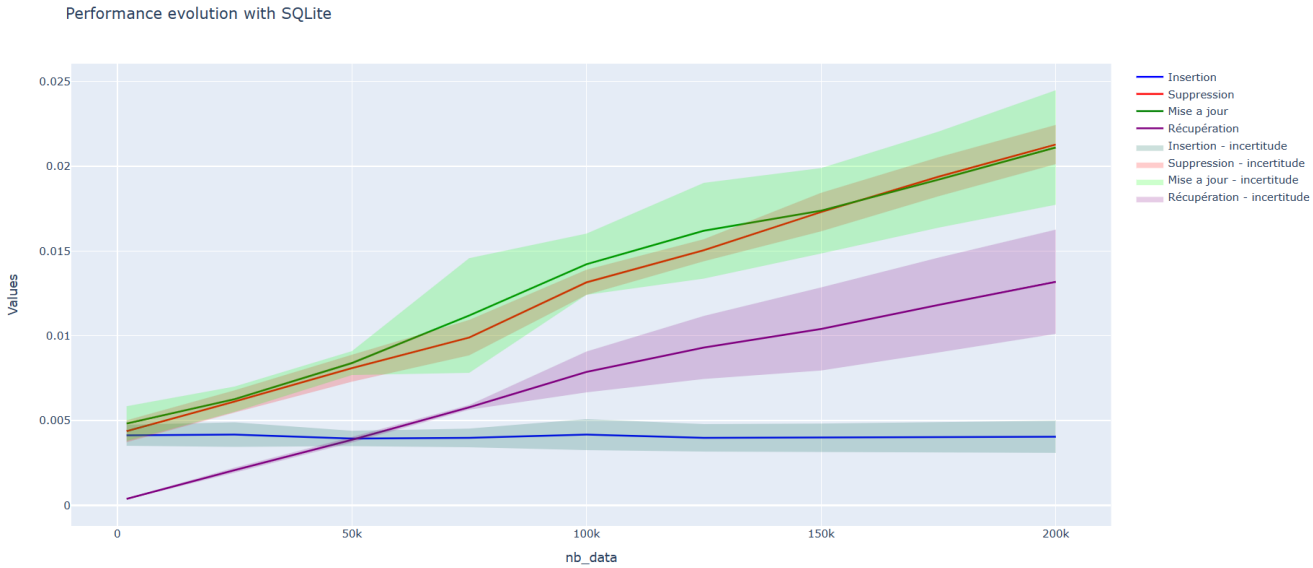


Figure 2: Graphe SQLite sans indexation

4.2 Seconde expérience - indexation

La Figure 3 et la Figure 4 correspondent aux résultats d'exécution de **sqlite_redis_index.ipynb**. Les résultats sont visibles respectivement dans les fichiers **graph_redis_index.html** et **graph_sqlite_index.html**.

L'axe des abscisses et des ordonnées représentent ici la même chose que dans l'expérience précédente.

Nos graphes ci-dessous sont de la forme suivante: la moyenne est une courbe nette, tandis que l'écart-type est la zone floutée.

Figure 3: Graphe Redis sans indexation

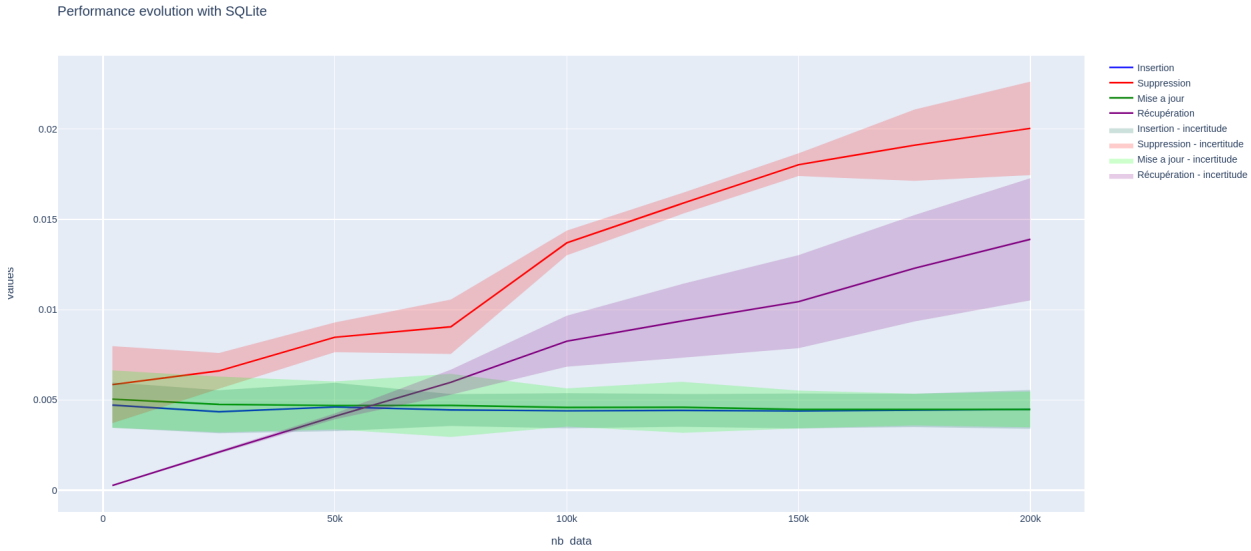


Figure 4: Graphe SQLite avec indexation

Le temps d'exécution a été de 6 minutes et 18.3 secondes pour le programme principal puis de 5.2 secondes et 0.7 seconde pour créer les visualisations.

4.3 Troisième expérience

Les résultats d'exécution de **sqlite_redis_v2.ipynb** (le nom du fichier peut ne pas sembler logique, en réalité, il souligne le fait qu'il est très différent des deux premières expériences) sont représentés dans la Figure 5 et la Figure 6. Étant donné la durée d'exécution relativement longue de ce programme (presque quatre heures), vous pouvez simplement examiner les résultats des cellules du programme. Les graphiques correspondants sont générés à la fin du programme et sont nommés respectivement **graph_redis_v2.html** et **graph_sqlite_v2.html**.

Nos graphes ci-dessous sont de la forme suivante: le temps total est représenté par une courbe nette.

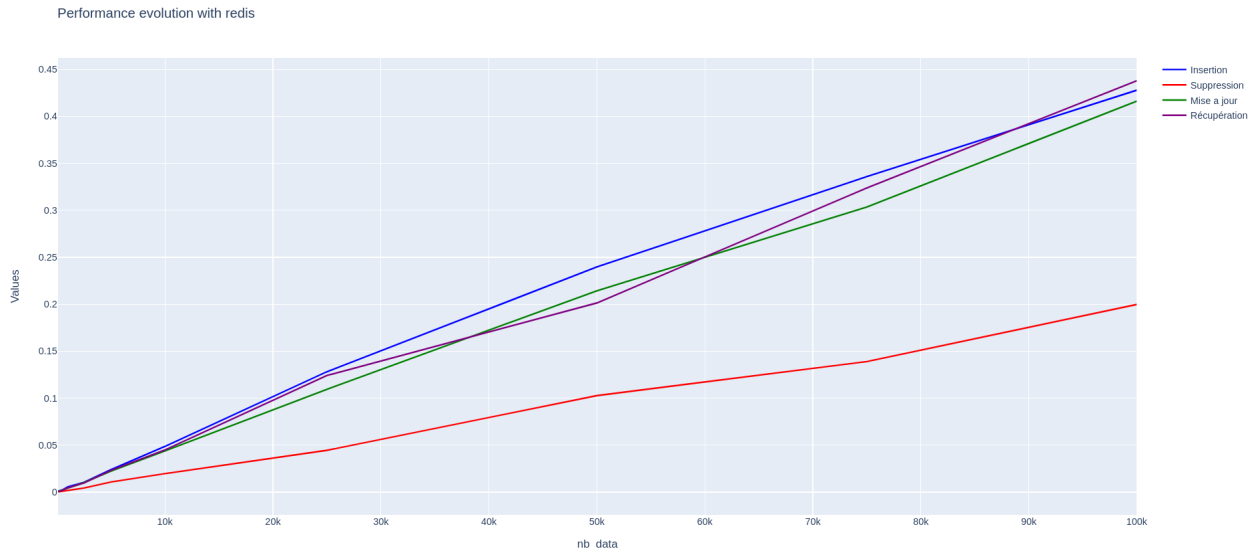


Figure 5: Graphe Redis sans indexation

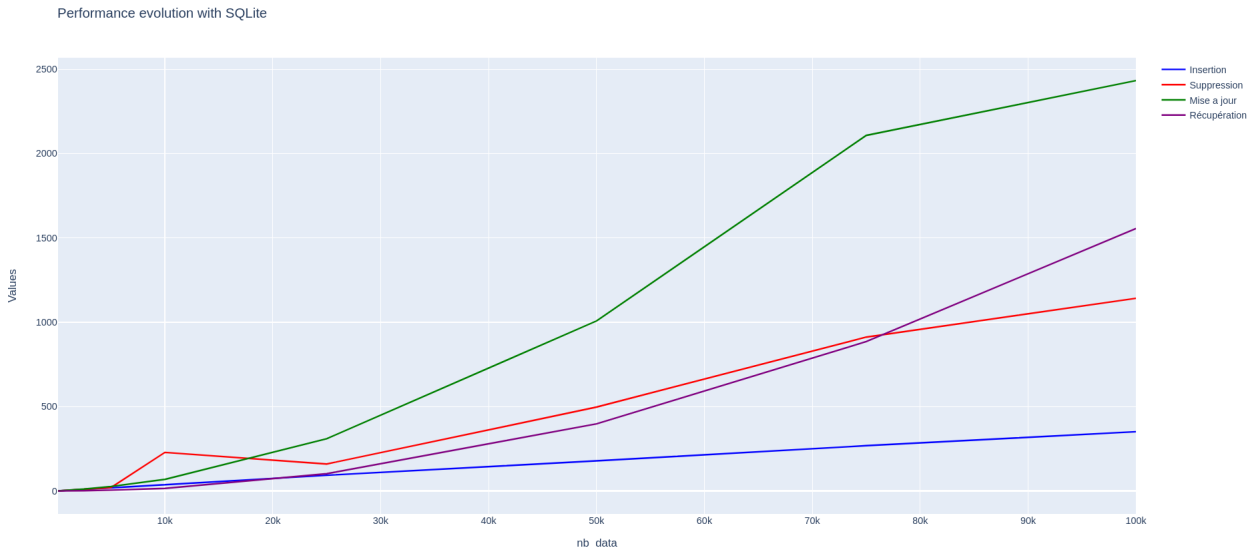


Figure 6: Graphe SQLite sans indexation

Le temps d'exécution a été de 214 minutes et 48.1 secondes pour le programme principal puis de 4.9 secondes et 0.3 seconde pour créer les visualisations.

4.4 Visualisation des résultats

Les temps pour chaque type d'opération et pour chaque expérience sont visualisables dans nos fichiers Jupyter. Il y a également des temps de connexion à chacune des bases de données (Redis et SQLite) qui sont affichés. On remarque que le temps de connexion à une base de données SQLite est bien plus rapide que le temps de connexion à une base de données Redis. Néanmoins, Redis est plus rapide dans ces opérations (voir section "5. Interprétation").

5 Interprétation

5.1 Première expérience

Sans indexation, on remarque que l'insertion se fait en un temps constant mais que la récupération, la modification et la suppression de données se font en un temps croissant de manière linéaire. Plus la base de données est grande, plus la récupération, la modification et la suppression de données nécessitent un plus grand temps afin d'être effectué puisqu'il y a plus de données à traiter avant d'arriver à trouver celle(s) que l'on souhaite(nt). En revanche, l'insertion ne nécessite pas de parcourir toutes les données et peut se faire sans délai d'attente supplémentaire en rapport avec la taille de la base de données.

Pour Redis, quelque soit l'opération effectuée, les temps sont chaotiques et de ce fait ne semblent pas dépendre du nombre de données préalablement inscrites dans la base de données. On peut observer sur la Figure 1 que l'écart-type est énorme par rapport à la moyenne, ce qui s'interprête par de grandes variations de temps entre deux opérations similaires (deux opérations d'insertion sur une base de données préremplies de nb_data peut prendre des temps très différents). Redis étant extrêmement rapide de base (notamment dû au fait que tout se passe en mémoire), ces variations sont de l'ordre de la microseconde. On peut donc considérer que le temps pris pour chaque opération sur Redis est constant, quelque soit la taille de la base de données.

Les temps sont néanmoins très légèrement sous-évalués, car le programme a tourné sur Python dans un environnement Jupyter, ce qui s'éloigne un peu des conditions réelles rencontrées lors de la réalisation des opérations directement sur Redis / SQLite.

5.2 Seconde expérience

La base de données Redis n'a pas été indexée. De ce fait, les résultats sont exactement les mêmes que ceux de la première expérience, ce qui confirme nos résultats précédents.

Avec l'indexation pour la base de données SQLite, nous avons constatés que la mise à jour est devenu constante comme l'insertion. En ce qui concerne les autres types d'opérations, elles restent dans le même ordre de grandeur. L'indexation permet de réduire considérablement le coût des mises à jour dans une base de données contenant un nombre important de données.

5.3 Troisième expérience

On remarque que le temps, à la fois pour Redis et pour SQLite, semble linéaire croissant. Redis semble être plus rapide dans les tâches de suppression que dans les autres tâches. En revanche, SQLite présente des différences de performances significatives selon le type d'opérations (de l'ordre du millier de seconde).

Il est à noter que pour la première expérience, le temps d'insertion (par exemple) d'une donnée sur Redis prend en moyenne haute 120 microseconde. Par contre, elle est entre 5 et 10 microseconde dans notre troisième expérience. Cela est dû au fait que nous avons exécutés plusieurs instructions simultanément (avec mset ; insertion multiple par exemple) dans cette expérience. Le temps d'allocation de l'espace nécessaire pour stocker les données et le temps de commit à la base de données Redis est donc forcément allégé ce qui explique cette différence. On remarque d'ailleurs que plus le nombre de données insérées simultanément est grand, plus

la moyenne d'insertion sur une donnée est petite, tout en se stabilisant vers 1000 insertions simultanées (chacune valent environs 5 microsecondes).

6 Conclusion

Pour conclure, nous pouvons en déduire que Redis est un système bien plus rapide que SQLite, malgré le temps de connexion plus important pour une base de données Redis que SQLite. Cela s'explique par le fait que toutes les opérations sont faites et stockées en mémoire vive (proche du processeur), ce qui n'est pas le cas pour SQLite.

De plus, la taille de la base de données n'a aucun impact sur les performances des opérations de Redis alors que c'est le cas pour une base de données SQLite. Néanmoins, les coûts en performance matérielle (et non pas temporelle) pour Redis sont bien plus importants que ceux de SQLite.

Effectuer plusieurs opérations en une seule fois permet une augmentation drastique des performances de Redis.

L'indexation permet d'augmenter les performances d'une base de données SQLite sur les opérations de mises à jour uniquement, les autres restants aussi performantes qu'avant.

7 Bibliographie

[1] Positive Thinking Company *Pourquoi utiliser Redis pour vos stockages de données* https://blog.positivethinking.tech/redis-ae0cac58ff25

[2] WPRBS *Les avantages et les inconvénients de MySQL NoSQL et Redis* https://www.wprbs.com/fr/the-pros-and-cons-of-mysql-nosql-and-redis.html

[3] IONOS *Redis, les bases de données vues autrement* https://www.ionos.fr/digitalguide/hebergement/aspects-techniques/quest-ce-que-redis/

[4] IBM *Qu'est ce que Redis expliqué ?* https://www.ibm.com/fr-fr/topics/redis

[5] Redis *Documentation Redis* https://redis.io/docs/

[6] SQLite *SQLite Documentation* https://www.sqlite.org/docs.html