Rapport TD1- Stockage clef-valeur, sharding, IO-bound et concurrency

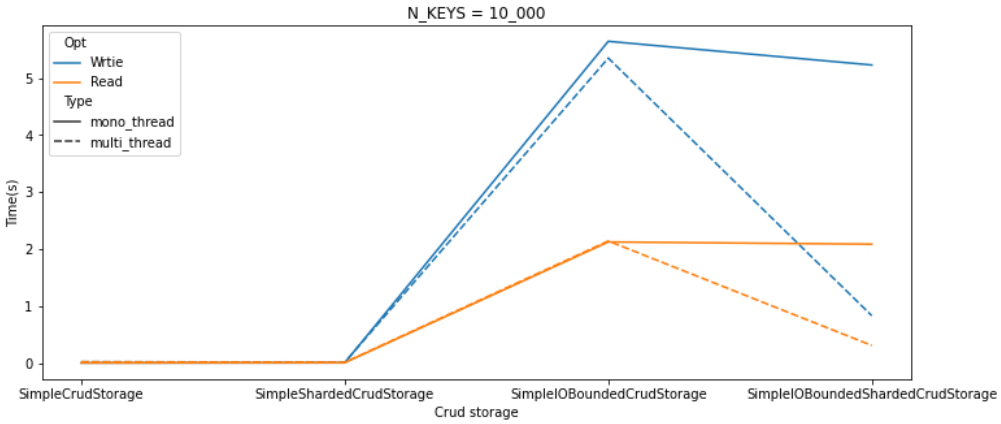

FIGURE 1 – Graphique du benchmark

En analysant dans la FIGURE 1, on peut donner des explications et des conclusions ci-dessous :

1. Tant que toutes les opérations 'Read' ou 'Write' s'interagissent directement avec la mémoire CPU (Pour le cas SimpleCrudStorage et SimpleShardedCrudStorage), quel que soit ils sont en mono_thread ou multi_thread, quel que soit qu'il y a de sharding ou pas, il n'y a peu de différence entre les deux. Car le temps de calcul et d'opération dans la mémoire CPU est si courte qu'on puisse le négliger.

2. Tant que toutes les données doivent être écrites dans le disque ou lues du Disque sans sharding (Pour le cas SimpleIOBoundedCrudStorage), le temps augmente due à nombreuses opérations I/O et l'application de Multithreading n'a aucun intérêt. Mais en appliquant sharding (Pour le cas SimpleIOBoundedShardedCrudStorage), le processus de la lecture et l'écriture sont en Concurrence avec un grand Throughput, grâce au Multithreading.

3. Dans la Base de Données Distribuée, le sharding permet de séprarer, diviser des données gigantesques en des partitions de taille réduite afin d'accélérer leur traitement ou de les gérer plus facilement. Il rend le stockage de données dans plusieurs serveurs soit possible, avec un coût d'hébergement réduit et plus sécurisé. Comme il y a souvent plusieurs copies de données réparties dans des différents noeuds du réseau, plus de sharding qu'on a, plus difficile pour nous d'assurer la tolérance de partition et le consistency.

4. GIL est un verrou mutex en Python qui empêche plusieurs threads d'exécuter simultanément, garantissant qu'un seul thread s'exécute dans un seul processeur à la fois et que plusieurs processeurs ne peuvent pas exécuter plusieurs threads. Dans des applications IO-Bound, la différence de performances entre le Multithreading et le Multiprocessing n'est pas grande. Car l'opération I/O va libérer immédiatement le GIL les uns après les autres.