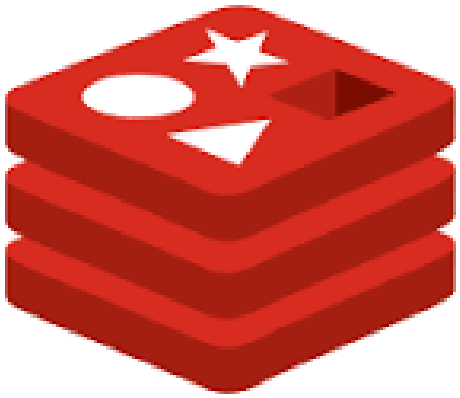
Bases de données NoSQL



redis

Nom: HAMRANI
Prénom: Amziane

I. Introduction:

La plupart d'entre nous avons déjà utilisé des bases de données relationnelles : ce sont celles où les données sont organisées en tables (lignes, colonnes), reliées entre elles par des clés primaires et étrangères, et interrogées avec le langage SQL. Ce modèle est très adapté quand les données sont bien structurées, que les règles d'intégrité sont fortes (contraintes, transactions ACID) et que l'on a besoin de requêtes complexes avec jointures.

Cependant, avec la montée en charge des applications web, des données massives et des besoins de très faible latence, ce modèle atteint ses limites en termes de performance et surtout de passage à l'échelle horizontale (ajouter facilement des machines). Pour répondre à ces contraintes, on a vu apparaître les bases de données dites *NoSQL*, qui regroupent plusieurs familles de SGBD non relationnels, pensés pour être plus flexibles sur le schéma et la cohérence afin de mieux optimiser la performance, la scalabilité et la gestion de données peu ou semi-structurées.

Les systèmes NoSQL se déclinent en plusieurs grandes familles, chacune adaptée à un type de données et de cas d'usage :

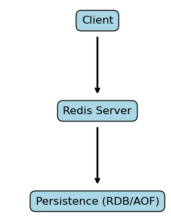
- Bases clé–valeur (comme Redis), qui s'appuie sur le stockage en mémoir, très simples et très rapides pour associer une clé à une valeur, souvent utiliser comme cache.
- Bases orientées documents (MongoDB), qui stockent des documents de type JSON.
- Bases orientées colonnes, utilisées pour de très grands volumes distribués.
- Bases orientées graphes, dédiées aux relations complexes entre entités.

Dans la suite du rapport, Redis sera présenté plus en détail comme un représentant de la famille clé–valeur.

II. Redis:

Redis fait partie de la famille des SGBD NoSQL clé–valeur et est open source. Contrairement à une base relationnelle classique qui lit/écrit principalement sur disque, Redis garde les données en RAM et n'utilise le disque que pour la persistance, ce qui le rend beaucoup plus rapide pour les lectures/écritures fréquentes.

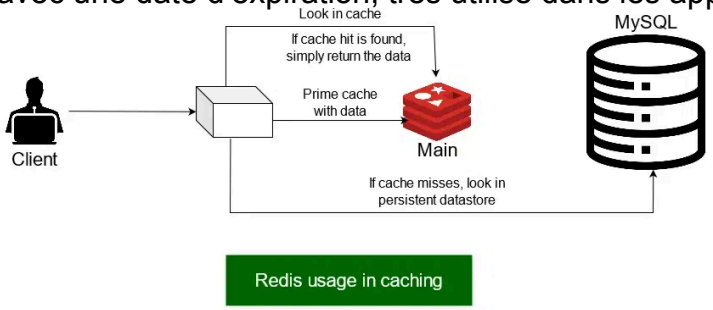
L'architecture de base est de type client–serveur : un processus serveur Redis tourne sur une machine et maintient toutes les données en mémoire, les applications clientes (en Python, Java, Node.js, etc.) se connectent au serveur via TCP et envoient des commandes (GET, SET, LPUSH



Use Cases:

Redis est surtout utilisé comme brique technique à côté d'une base principale (souvent relationnelle) pour absorber la charge et réduire la latence :

- Cache applicatif : stocker en mémoire le résultat de requêtes coûteuses (SQL, API externes) afin de répondre beaucoup plus vite lors des appels suivants.
- Gestion de sessions : enregistrer les sessions utilisateur (tokens, paniers, préférences) avec une date d'expiration, très utilisé dans les applications web.



Installation:

Pour une installation rapide de Redis dans le cadre du TP, on peut utiliser un conteneur Docker, ce qui évite de configurer Redis "à la main" sur la machine hôte.

```
# 1. Récupérer l'image officielle de Redis
docker pull redis

# 2. Lancer un conteneur Redis en arrière-plan
docker run --name tp-redis -p 6379:6379 -d redis

# 3. Entrer dans le conteneur et lancer le client redis-cli
docker exec -it tp-redis redis-cli
```

On peut également installer Redis "classiquement", via les binaires fournis pour chaque système d'exploitation (Linux, Windows, macOS). Dans ce cas, il suffit d'aller sur le site officiel de Redis, rubrique "Download/Install", et suivre la procédure adaptée à ton OS :
https://redis.io/

Manipulation:

Dans Redis, les données ne sont pas seulement des chaînes simples : il existe plusieurs structures de données natives, chacune avec ses propres commandes.

Clé-Valeure:

La structure la plus simple de Redis est la paire clé–valeur, où la clé est une chaîne et la valeur est aussi une chaîne (qui peut contenir du texte, des nombres, du binaire, etc.). C'est l'équivalent d'un dictionnaire ou d'une map dans un langage de

programmation : on associe une clé unique à une valeur, puis on lit ou on modifie cette valeur avec des commandes très simples.

Création, lecture, suppression:

- `SET key value:`

Rôle : créer ou mettre à jour la valeur associée à une clé.
Exemple : `SET user:1 "Amziane"`.

- `GET key:`

Rôle : lire la valeur associée à une clé.
Exemple : `GET user:1 → "Amziane"` si la clé existe.

- `DEL key [key ...]:`

Rôle : supprimer une ou plusieurs clés et leurs valeurs.

- `EXISTS key [key ...]:`

Rôle : tester l'existence d'une ou plusieurs clés.

Opérations sur des entiers:

- `INCR key:`

Rôle : incrémenter de 1 la valeur entière stockée sous la clé.

- `DECR key:`

Rôle : décrémenter de 1 la valeur entière stockée sous la clé.

Gestion du temps de vie:

- `EXPIRE key seconds:`

Rôle : définir un temps de vie (TTL) en secondes sur une clé.

- `TTL key:`

Rôle : connaître le temps de vie restant (en secondes) d'une clé.
Retourne : une valeur ≥ 0 : nombre de secondes restantes; -1 si aucun TTL défini, -2 si la clé n'existe plus.

<u>Listes:</u>

Une liste Redis est une séquence ordonnée de valeurs (Strings). On peut ajouter ou retirer des éléments aux deux extrémités, ce qui permet de faire des files(queue) ou des piles (stack).

Ajouter des éléments:

- `LPUSH key value1 [value2 ...]:`

Rôle : Ajoute un ou plusieurs éléments en tête de la liste.
Exemple : `LPUSH tasks "t1" "t2"` → la liste `tasks` contient `["t2", "t1"]`.

- `RPUSH key value1 [value2 ...]:`

Rôle : Ajoute un ou plusieurs éléments en queue de la liste.
Exemple : `RPUSH tasks "t3"` → `["t2", "t1", "t3"]`.

Retirer des éléments:

- `LPOP key`

Supprime et retourne le premier élément (tête) de la liste.
Utile pour implémenter une file FIFO (on consomme par le début).

- `RPOP key`

Supprime et retourne le dernier élément (queue) de la liste.
Utile pour implémenter une pile LIFO (on consomme par la fin).

Lire et inspecter la liste:

- `LRANGE key start stop`

Retourne une plage d'éléments par index. `0 -1` signifie "toute la liste".
Exemple : `LRANGE tasks 0 -1` → affiche tous les éléments dans l'ordre.

- `LLEN key`

Donne la longueur (nombre d'éléments) de la liste.

<u>Les ensembles (Sets):</u>

Un Set Redis est une collection non ordonnée de valeurs uniques. Pas de doublons, et l'ordre n'est pas garanti.

Commandes de base des Sets:

- `SADD key member [member ...]`

Ajoute un ou plusieurs éléments dans l'ensemble. Crée le set si la clé n'existe pas.

- `SMEMBERS key`

Retourne tous les éléments du set.

- `SREM key member [member ...]`

Supprime un ou plusieurs éléments du set.

- `SISMEMBER key member`

Teste si un élément appartient au set (renvoie 1 ou 0).

- `SCARD key`

Donne la taille (nombre d'éléments) du set.

Les ensembles triés (Sorted Sets):

Un Sorted Set (type ZSET) ressemble à un Set, mais chaque élément a un score numérique, ce qui permet un ordre total.

Commandes de base des Sorted Sets:

- `ZADD key score member [score member ...]`

Ajoute des éléments avec leurs scores (et met à jour le score si l'élément existe).

- `ZRANGE key start stop [WITHSCORES]`

Retourne les éléments triés par score croissant, entre deux indices. Avec `WITHSCORES`, affiche aussi les scores.

- `ZREVRANGE key start stop [WITHSCORES]`

Comme `ZRANGE` mais par score décroissant.

- `ZREM key member [member ...]`

Supprime un ou plusieurs éléments.

- `ZCARD key`

Nombre d'éléments dans le sorted set.

- `ZSCORE key member`

Donne le score d'un élément.

Les hachages (Hashes) avec HSET:

Un Hash Redis est une table de champs/valeurs sous une même clé, proche d'un objet JSON simple ou d'un dictionnaire. Les hashes sont parfaits pour représenter des objets (utilisateur, produit, etc.) sans passer par un schéma relationnel.

Commandes de base des Hashes:

- `HSET key field value [field value ...]`

Ajoute ou met à jour un ou plusieurs champs dans le hash.
Exemple : `HSET user:1 name "Amziane" age "25"`.

- `HGET key field`

Lit la valeur d'un champ.

- `HGETALL key`

Retourne tous les champs et valeurs du hash.

- `HDEL key field [field ...]`

Supprime un ou plusieurs champs.

- `HLEN key`

Donne le nombre de champs dans le hash.

- `HEXISTS key field`

Teste si un champ existe.

Pub/Sub dans Redis:

Redis propose un mécanisme de messagerie appelé Publish/Subscribe (Pub/Sub). L'idée est simple : des producteurs de messages (éditeurs) envoient des messages sur des canaux, des consommateurs (abonnés) se subscribent à ces canaux, chaque message publié sur un canal est immédiatement poussé à tous les abonnés de ce canal. Il n'y a pas de persistance des messages : si un client n'est pas abonné au moment de la publication, il ne recevra pas le message.

S'abonner à un canal:

- `SUBSCRIBE channel1 [channel2 ...]`

Le client entre en mode "abonné" et recevra tous les messages publiés sur ces canaux.
Exemple (dans un premier terminal) : `SUBSCRIBE news`
Le client affiche alors chaque message reçu sur `news`.

Publier un message:

- `PUBLISH channel message`

Envoie un message sur un canal.
Retourne le nombre d'abonnés qui ont reçu le message.
Exemple (dans un deuxième terminal) :
`PUBLISH news "Hello world"`
Le client abonné à `news` affiche le message.

Se désabonner:

- `UNSUBSCRIBE [channel1 channel2 ...]`

Se désabonne d'un ou plusieurs canaux. Sans argument, se désabonne de tous les canaux.Quand le nombre de canaux abonnés tombe à 0, le client sort du mode "abonné" et peut exécuter des commandes normales.

Abonnement avec motif:

- `PSUBSCRIBE pattern [pattern ...]`

S'abonne à tous les canaux dont le nom matche un motif (wildcards).
Exemple : `PSUBSCRIBE news:*` recevra les messages sur `news:sport`, `news:tech`, etc.

- `PUNSUBSCRIBE [pattern ...]`

Se désabonne des motifs.