Base de données NoSQL – TD5
Clé-Valeur : `Redis`

Master 2 BI&BD - 2020–2021 - 22/11/2021

1 Introduction

1.1 Les BD NoSQL Clé-Valeur

Les bases de données clé-valeur fonctionnent comme un grand tableau associatif. Les données sont simplement représentées par un couple clé-valeur. La valeur peut être une simple chaîne de caractères ou un objet sérialisé. Cette absence de structure ou de typage a un impact important sur le requêtage : toute l'intelligence portée auparavant par les requêtes SQL devra être portée par l'applicatif qui interroge la BD.

Dans une BD clé-valeur, chaque objet est identifié par une clé unique, seule façon de le requêter. La structure de l'objet est libre, souvent laissée à la charge du développeur de l'application (XML, JSON, ...), la base ne gérant généralement que des chaînes d'octets.

Leur exploitation est basée sur 4 opérations (CRUD) :

1. **C**reate : créer un nouvel objet avec sa clé → create(key, value)
2. **R**ead : lit un objet à partir de sa clé → read(key)
3. **U**pdate : met à jour la valeur d'un objet à partir de sa clé → update(key, value)
4. **D**elete : supprime un objet à partir de sa clé → delete(key)

Les BD clé-valeur disposent généralement d'une simple **interface de requêtage HTTP REST** accessible depuis n'importe quel langage de développement. Elles ont des performances très élevées en lecture et en écriture et une scalabilité horizontale considérable.

1.2 Utilisations principales

Les bases de données de type clés-valeurs sont utilisées pour :

- les dépôts de données avec besoins de requêtage très simples
- les systèmes de stockage de cache ou d'information de sessions distribuées (quand l'intégrité relationnelle des données est non significative)
- les profils, préférences utilisateurs
- les données de panier d'achat
- les données de capteur
- …

1.3 Forces et faiblesses

- Forces :
  - modèle de données simple
  - évolutivité (scalable)
  - disponibilité
  - maintenances minimum
- Faiblesses :
  - interrogation seulement sur clé

— déporte une grande partie de la complexité vers l'application

1.4 Redis

Nous allons explorer Redis, ses principes, son API et ce qu'on peut en faire. Redis est une base de données open source de type clé-valeurs mono-threadée. C'est en gros une grosse HashMap, mais avec des données structurées que nous allons détailler : des chaînes de caractères, des listes, des hash, des set, des set triés. L'utilisation de redis est très simple et la vitesse de lecture et d'écriture est vertigineuse.

Toutes les opérations sont atomiques, vous ne risquez pas d'avoir des soucis de concurrence d'accès à vos données. Par contre, il est impossible de requêter les valeurs comme on le fait habituellement avec un WHERE en SQL, mais avec un peu d'astuce, d'habitude et de dénormalisation des données, on arrive très vite à nos fins en demandant "la bonne clé". Il faut également savoir que vous êtes limités par la taille de la RAM car Redis garde toutes ses données en mémoire (c'est aussi pour cela que c'est très rapide).

1.4.1 Persistance

Redis est par nature une base de données en mémoire. Un système de persistance sur disque y a été ajouté, lequel sauvegarde l'état des bases de données jusqu'au prochain redémarrage de Redis. Le système par défaut est nommé RDB, qui se base sur la prise régulière ou manuelle d'un snapshot, nommé dump.rdb. Ce snapshot est stocké sur le disque dans le répertoire courant de Redis (celui dans lequel il a démarré) ou dans un répertoire indiqué dans le fichier de configuration si celui-ci est spécifié au démarrage.

1.4.2 Réplication

Un serveur Redis peut être configuré pour être le secondaire d'un autre serveur, dans une architecture maître-esclave arborescente (un secondaire peut se baser sur un secondaire). Pour cela, il suffit de mentionner dans le fichier de configuration : `slaveof ip port` en indiquant l'adresse IP et le port du serveur maître. Le serveur secondaire est la copie exacte du serveur maître. L'option `slave-serve-stale-data` indique le comportement à observer si la connexion avec le maître est perdue. La valeur par défaut est `yes` : le serveur continue à répondre avec des données potentiellement désynchronisées. Si l'option vaut `no`, une requête sur le serveur secondaire retourne une erreur "SYNC with master in progress". La commande `SLAVEOF` permet de déterminer si le serveur est un secondaire. L'option `slave-read-only`, par défaut à `yes`, indique si les écritures sont interdites sur le secondaire. Écrire sur un Redis secondaire n'a d'intérêt que pour des données éphémères : il n'y a pas de synchronisation vers le serveur maître et les données pourront donc être écrasées par des modifications effectuées sur le maître.

1.4.3 Redis-sentinel et Redis Cluster

Redis-sentinel (http ://redis.io/topics/sentinel) est disponible sous la dernière version stable 6.2.6. Il utilisé très souvent en production, étant donné qu'il est parfaitement stable. Il s'agit d'un outil de monitoring et de gestion de haute disponibilité. Il permet, conjointement à l'option de configuration `slave_priority`, de promouvoir automatiquement un secondaire en maître si ce dernier est indisponible.

Redis Cluster (http ://redis.io/topics/cluster-tutorial) est un système décentralisé où les noeuds communiquent à l'aide un canal TCP supplémentaire. Les données sont *shardées* automatiquement par Redis en utilisant la technique des *vnodes*, nommées ici des `hashslots`. Le nombre de slots est fixé à 16 384, et le placement de la clé est simplement effectué en calculant le modulo 16 384 du hash de la clé. La réplication est assurée selon un modèle maître-secondaire. Si vous voulez créer un cluster de

trois shards, vous ajoutez trois machines pour accueillir les répliquas. Vous devez avoir plus de deux maîtres, car le basculement automatique se fait par quorum : il faut donc être au moins deux pour prendre la décision.

1.5 Installation

Avec Docker, il vous sera possible d'utiliser Redis dans un environnement virtuel.

```
$ docker search redis
$ docker pull redis
$ docker run --name nom-serveur-redis -p 6379:6379 -d redis
```

Vérifiez que votre conteneur est bien lancé. Si oui, vous êtes prêts à vous connecter au serveur Redis et à interagir avec la base de données.

```
$ docker ps
```

1.6 Le client

Des clients graphiques existent. Mais pour nos besoins nous allons utiliser le client `redis-cli` disponible dans votre conteneur docker.

```
docker exec -it nom-serveur-redis bash
```

Un fois identifié sur votre conteneur lancer la commande `redis-cli`

```
# redis-cli
```

`127.0.0.1:6379>` apparaître sur votre écran quand vous êtes connecté à votre serveur redis.
La commande suivante vous listera les informations concernant votre serveur Redis.

```
127.0.0.1:6379> INFO
```

Par défaut et au lancement de Redis, vous avez 16 bases de données. Vous êtres automatiquement sur la base de données numéro 0. Vous pouvez basculer sur une autre base de données (par exemple la numéro 1) avec la commande :

```
127.0.0.1:6379> SELECT 1
```

2 Prise en main de Redis

2.1 Les commandes de base

Comme vu dans la section précédente, Redis est une base de données clé-valeur. L'essence même d'une BD clé-valeur est la capacité de stocker des données appelé "valeur", à l'intérieur d'une clé. Cette donnée ne peut-être récupérée de la base de données que si on connait la clé dans laquelle elle a été stockée. La commande `SET` est utilisée pour stocker une valeur "Business Intelligence et BigData" dans la clé "dis :bibd" :

Essayez dans votre client Redis la commande suivante :

```
127.0.0.1:6379> SET dis:bibd "Business Intelligence et Big Data"
```

Redis vous renvoie OK pour vous notifier qu'il a stocké la valeur "Business Intelligence et Big Data". Vous pouvez récupérer cette valeur en utilisant la commande `GET dis:bibd`

Il existe d'autres commandes de base fournit par Redis :

- `DEL` supprime une clé et la valeur qui y est associée
- `SETNX` ajoute une clé contenant une valeur si la clé n'existe pas
- `GETSET` récupère une valeur et en crée une en même temps
- `INC` incrémente automatiquement une clé dont la valeur est de type numérique
- `DECR` décrémente automatiquement une clé dont la valeur est de type numérique

```
127.0.0.1:6379> SET compteur 10
OK
127.0.0.1:6379> INCR compteur
(integer) 11
127.0.0.1:6379> INCR compteur
(integer) 12
127.0.0.1:6379> GET compteur
"12"
127.0.0.1:6379> DEL compteur
(integer) 1
127.0.0.1:6379> get compteur
(nil)
```

Redis propose aussi des commandes pour manipuler des clés et des valeurs multiples en même temps. Ces commandes sont préfixées par `M` (pour Multiples), sauf pour la commande `DEL`.

- `MSET clé1 valeur1 [clé2 valeur2 ...]`
- `MGET clé1 [clé2 ...]`
- `MSETNX clé1 valeur1 [clé2 valeur2 ...]`
- `DEL clé1 [clé2 ...]`

2.2 Expiration

Le concept de delai et l'un des avantages clé de Redis. Etant donnée qu'il est utilisé en mémoire, pouvoir limiter dans le temps l'occupation de mémoire est très important pour ne pas la saturer et impacter les performances globales de votre ordinateur ou du cluster de calcul.

Les commandes utilisées sont :

- `EXPIRE clé secondes` permet de supprimer une clé après un nombre de secondes
- `EXPIREAR clé timestamp` permet de supprimer un clé à une *timestamp* donnée
- `SETEX clé secondes valeur` permet de définir une durée de validité de la clé et de sa valeur dès sa création
- `TTL clé` renvoie le compte à rebours (en seconde) du temps de validité restant d'une clé donnée
- `PERSIST clé` annule le compte à rebours de validité d'une clé-valeur pour rendre cette dernière persistante

```
127.0.0.1:6379> EXPIRE dis:bibd 20
(integer) 1
127.0.0.1:6379> TTL dis:bibd
(integer) 9
127.0.0.1:6379> PERSIST dis:bibd
(integer) 1
127.0.0.1:6379> TTL dis:bibd
(integer) -1
```

La commande `PERSIST` renvoie :
- `1` si l'exécution de la commande est réussie
- `0` si la clé n'existe pas ou qu'aucun compte à rebours n'a été associé à la clé

2.3 Les listes sous Redis

Les listes sous Redis sont des listes liées (linked list). Cela veut dire que même si vous avez des millions d'enregistrement dans une liste, cela sera toujours aussi rapide d'insérer un élément en tête ou en queue de liste. La contrepartie à cette vitesse est l'accès à un élément dans la liste par son index. Les utilisateurs de ArrayList ou LinkedList en Java connaissent bien cette différence. Les principales commandes commencent en général par `L` comme List et sont `RPUSH` et `LPUSH` (et leurs amies xPOP) qui permettent respectivement d'ajouter un élément en fin ou en début de liste, `LRANGE` pour obtenir une partie des éléments de la liste, `LINDEX` pour obtenir un seul élément de la liste, `LLEN` pour obtenir la taille de la liste.

```
127.0.0.1:6379> RPUSH messages "Bienvenue dans Redis"
(integer) 1
127.0.0.1:6379> RPUSH messages "Ce cours est une petite introduction à Redis"
(integer) 2
127.0.0.1:6379> RPUSH messages "Non seulement c'est pratique, mais c'est rapide"
(integer) 3
127.0.0.1:6379> LRANGE messages 0 3
1) "Bienvenue dans Redis"
2) "Ce cours est une petite introduction \xc3\xa0 Redis"
3) "Non seulement c'est pratique, mais c'est rapide"
```

2.4 Hash sous Redis

Une hash permet de stocker dans un même enregistrement plusieurs couples de clé/valeurs. Les commande de hash commencent... par un `H` comme Hash. On y trouve `HSET, HGET, HLEN`, mais aussi `HGETALL` pour obtenir tous les couples clés-valeurs, `HINCRBY` pour incrémenter un compteur dans la hash, `HKEYS` et `HVALS` pour obtenir toutes les clés ou valeurs et `HDEL` pour faire le ménage.

```
127.0.0.1:6379> HSET nosql redis "clé/valeur"
(integer) 1
127.0.0.1:6379> HSET nosql mongodb "document"
(integer) 1
127.0.0.1:6379> HSET nosql riak "fault tolerant"
(integer) 1
127.0.0.1:6379> HGET nosql redis
"clé/valeur"
127.0.0.1:6379> HGETALL nosql
1) "redis"
2) "clé/valeur"
3) "mongodb"
4) "document"
5) "riak"
6) "fault tolerant"
127.0.0.1:6379> HKEYS nosql
1) "redis"
2) "mongodb"
3) "riak"
127.0.0.1:6379> HVALS nosql
1) "clé/valeur"
2) "document"
3) "fault tolerant"
127.0.0.1:6379> HLEN nosql
(integer) 3
```

2.5 Sets sous Redis

Les Sets sont des collections d'objets non ordonnées. Les commandes commencent toutes avec un `S` comme Set, parmi celles-ci on trouve `SADD` pour ajouter une valeur à un set, `SCARD` pour obtenir la taille (cardinalité) d'un set, et surtout les commandes `SINTER`, `SUNION`, `SDIFF` qui permettent respectivement d'obtenir l'intersection, l'union et la différences entre 2 sets.

```
127.0.0.1:6379> SADD amis:bob "sam"
(integer) 1
127.0.0.1:6379> SADD amis:bob "steve"
(integer) 1
127.0.0.1:6379> SADD amis:bob "john"
(integer) 1
127.0.0.1:6379> SADD amis:steve "sam"
(integer) 1
127.0.0.1:6379> SADD amis:steve "steve"
(integer) 1
127.0.0.1:6379> SADD amis:steve "alice"
(integer) 1
127.0.0.1:6379> SCARD amis:steve
(integer) 3
127.0.0.1:6379> SINTER amis:steve amis:bob
1) "sam"
2) "steve"
127.0.0.1:6379> SUNION amis:bob amis:steve
1) "steve"
2) "john"
3) "sam"
4) "alice"
redis 127.0.0.1:6379> SDIFF amis:bob amis:steve
1) "john"
127.0.0.1:6379>  SDIFF amis:steve amis:bob
1) "alice"
```

NB :

— `SDIFF` ne donne pas le même résultat, l'ordre des clés est importante pour obtenir le résultat attendu.

— `SDIFFSTORE clé-destination clé1 clé2` permet de stocker dans une clé la ou les différences entre deux `Sets`.

— `SMEMBERS clé` permet de récupérer la valeur contenue dans la clé-destination

2.6 Sets triés sous Redis

Les sets triés (Sorted Set) sont similaires à des Sets mais ajoutent une notion de score aux valeurs ajoutées aux Set, ce qui permet de faire des tris. Les commandes commencent toutes par `Z` comme **Z**orted Set. On trouve donc les **Z**équivalents des commandes précédentes, à savoir `ZADD, ZCARD, ZINTER, ZUNION, ZDIFF` mais aussi `ZRANGE`, `ZRANGEBYSCORE` et `ZRANK` qui tirent parti des scores des données stockées.

```
 127.0.0.1:6379> ZADD savants 1564 "Galilee"
(integer) 1
127.0.0.1:6379> ZADD savants 1643 "Isaac Newton"
(integer) 1
127.0.0.1:6379> ZADD savants 1571 "Johannes Kepler"
(integer) 1
 127.0.0.1:6379> ZADD savants 1879 "Albert Einstein"
(integer) 1
 127.0.0.1:6379> ZADD savants 1858 "Max Planck"
(integer) 1
127.0.0.1:6379> ZADD savants 1887 "Erwin Schrodinger"
(integer) 1
127.0.0.1:6379> ZRANGE savants 0 -1
1) "Galilee"
2) "Johannes Kepler"
3) "Isaac Newton"
4) "Max Planck"
5) "Albert Einstein"
6) "Erwin Schrodinger"
 127.0.0.1:6379> ZREVRANGE savants 0 -1
1) "Erwin Schrodinger"
2) "Albert Einstein"
3) "Max Planck"
4) "Isaac Newton"
5) "Johannes Kepler"
6) "Galilee"
127.0.0.1:6379> ZREVRANGE savants 0 -1 WITHSCORES
 1) "Erwin Schrodinger"
 2) "1887"
 3) "Albert Einstein"
 4) "1879"
 5) "Max Planck"
 6) "1858"
 7) "Isaac Newton"
 8) "1643"
 9) "Johannes Kepler"
10) "1571"
11) "Galilee"
12) "1564"
```

Ce qui est intéressant, c'est de pouvoir faire des requêtes sur les scores du set :

```
127.0.0.1:6379> ZRANGEBYSCORE savants -inf 1800
1) "Galilee"
2) "Johannes Kepler"
3) "Isaac Newton"
127.0.0.1:6379> ZRANGEBYSCORE savants 1800 2010
1) "Max Planck"
2) "Albert Einstein"
3) "Erwin Schrodinger"
```

3 Exercices

Redis ne dispose pas de structure comme les `Tables` dans les SGBDR, ni de `collections` comme en mongoDB et encore moins des relations. On ne peut stocker qu'une valeur pour une clé. Cependant les relations peuvent être stockées de manière différente avec la dénormalisation, et on peut créer des ensembles hierarchisés de groupes de clés. On peut ainsi stocker dans un même sous-ensemble un groupe de clés-valeurs.
On peut donc faire quelque chose de ce genre :

```
pilote:1 => {
  NomPilote: "Blériot",
  PrenomPilote: "Louis",
  Ville : "Paris",
  Salaire : 13000
}
pilote:2 => {
  NomPilote: "Garos",
  PrenomPilote: "Roland",
  Ville : "Toulouse",
  Salaire : 13000
}
```

Pour stocker les informations concernant des pilotes.

3.1 Exercice 1

1. avec l'aide tu tableau suivant créez un groupe de clé valeur qui permettra d'enregistrer les informations sur des pilotes.

NumeroPilote	NomPilote	PrenomPilote	Ville	Salaire
1	Blériot	Louis	Paris	13 000
2	Garros	Roland	Toulouse	17 000
3	Mermoz	Jean	Toulouse	11 000
4	Saint-Exupéry(de)	Antoine	Lyon	15 000
5	Chavez	Géo	Nice	10 000

Données 1 – Pilotes

2. affichez le nom du pilote #3

3. affichez toutes les infomations concernant Antoine de Saint-Exupéry
4. modifiez le salaire du pilote #2, nouveau salaire : 16 000

3.2 Exercice 2 : les sets et hsets

Sets

1. créez des sets sur nos pilotes
2. affichez la ville des pilotes #2 et #3
3. stockez le ville des pilotes #2 et #3 dans une nouvelle clé nommée : "memeville" et affichez la valeur de "memeville"

Hsetes

1. créez des sets triés pour nos pilotes (Salaire - Nom)
2. affichez les pilotes dont le salaire est inférieur à 12 000
3. affichez les pilotes dont le salaire est compris entre 14 000 et 20 000

3.3 Exercice 3 : Structure relationelle vs Clé-Valeur

Les structures relationnelles sont destinées à assurer l'indépendance des données et à offrir. Elles permettent de manipuler les données comme des ensembles en effectuant des opérations de la théorie des ensembles.

Le tableau de vol suivant décrit la liste des vols effectués par nos pilotes (Données 1).

NumeroVol	NumeroPilote	Avion	VilleDepart	VilleArrivee	Date	HeureDepart	HeureArrivee
1	1	A300	Paris	Lyon	01-03-2015	13h	15h
2	3	B707	Toulouse	Paris	01-03-2015	12h	15h
3	2	A300	Toulouse	Lyon	01-03-2015	16h	17h
4	5	B747	Nice	Paris	01-03-2015	13h	16h
5	4	A310	Lyon	Paris	01-03-2015	17h	19h
6	1	A300	Lyon	Toulouse	02-03-2015	10h	11h
7	3	B707	Paris	Lyon	02-03-2015	13h	15h
8	2	A300	Lyon	Toulouse	02-03-2015	14h	15h
9	5	B747	Paris	Nice	02-03-2015	14h	17h
10	4	Mercure	Paris	Bordeaux	02-03-2015	16h	19h

Données 2 – Vols

1. proposez un modèle en clé-valeur capable de stocker les données des deux tables
2. proposez un modèle en clé-valeur susceptible de contenir les relations entre vols et pilotes
3. créez les couples clés-valeurs qui enregistreront sous Redis les données de vols
4. re-créez les couples clés-valeurs des données concernant les pilotes
5. créez les couples clés-valeurs qui décriveront les relations entres pilotes et vols
6. affichez la ville de départ du vol #9
7. définissez un compte à rebours d'une heure pour le vol #2
8. affichez l'heure de départ du vol #1
9. affichez le nom du pilote du vol #7
10. combien de temps reste t-il au compte à rebours du vol #2