NF26 — Stockage en haute volumétrie et applications

TD2 — Stockage clef-valeur, et *Consistent Hashing*

P2021 — 29 avril 2021

Julien Jerphanion, Florent Chehab

Contents

1 Dynamicité des shards: *Consistent hashing*

Nous avons vu la dernière fois deux systèmes de stockage de données clef / valeurs relativement simples pour illustrer le *sharding* avec un nombre de *shards* constant. Que se passe-t-il si le nombre de *shard*s change dynamiquement ? Comment répartir la charge dans ce cas ?

Cette session vise à introduire un nouveau système de stockage reposant sur le *consistent hashing*, abordé en cours.

1.1 Introduction

1.1.1 Sur l'organisation du TD

Une fois le code du projet associé à ce TD cloné depuis votre projet nominatif, vous pouvez utiliser les commandes:

```
make help
make init
make style
make test
```

Pour toute cette partie 1, nous vous demandons de compléter le code dans le fichier `/key_value_part2/consistent_crud_storage.py` afin d'implémenter les opérations *CRUD* de telle sorte à ce que les tests fonctionnent.

Vous pouvez lancer ces tests avec:

```
make test
```

Vous êtes libre de rajouter des tests. Soyez vigilants au fait que les tests utilisés par le système automatisé d'évaluation seront différents et plus exhaustifs. Pour connaitre les critères d'évaluation, rendez-vous en section 2.2.

8 tests existent pour l'implémentation de base. Vous pouvez les lancer avec:

```
make test-base
```

1.1.2 Sur le *Consistent Hashing*

Le *consistent hashing* est une technique de répartition de charge introduite par [1] dont nous allons implémenter une version plus simple dans ce TD [1].

On définit la *fonction de condensat*:

$$h : k \to [\![N]\!] = \{0, 1, ..., N-1\}$$

qui assigne à un élément k, nommé clef, k un entier $h(k)$ de $[\![N]\!]$ nommé condensat ou *hash*.

On dispose initialement de Q nœuds de stockage (”*shards* ”) $0 \leq q < Q$ associés à des *hashes* $\{h(j)\}_{0 \leq j < Q}$ que l'on peut représenter dans $[\![N]\!]$ sous la forme d'un tableau circulaire.

[1] Pour reprendre les termes introduit dans [1]: l'étendue (”*spread*”) des shards sera ici égale à un et donc $R = 1$.

Le stockage des données $S \triangleq \{(k_i, v_i)\}_{1 \leq i < N}$ est réalisé ainsi:

1. chaque donnée (k_i, v_i) est associé à son *hash* $h(k_i)$
2. chaque *shard* j stocke les données entre son *hash* et celle du *shard* l suivant (non incluse). C'est-à-dire, j stocke l'ensemble des données suivant:

$$V_j \triangleq \{(k_i, v_i) \mid (k_i, v_i) \in S,\ h(k_i) \in [\![(h(j), h(l))[\![_N\}$$

où, pour reprendre les notations du cours:

$$[\![n, m[\![_N \triangleq \begin{cases} \{n, n+1, n+2, \cdots, m-1\} & \text{si } n < m \\ \{n, n+1, n+2, \cdots, N-1\} \cup \{0, 1, \cdots, m-1\} & \text{sinon} \end{cases}$$

1.1.3 Présentation de la nouvelle classe `ConsistentHashingCrudStorage`

Nous reprenons l'architecture de classes du TD 1 pour les opérations *CRUD* en y ajoutant une nouvelle classe `ConsistentHashingCrudStorage`, comme illustré sur la figure 1.

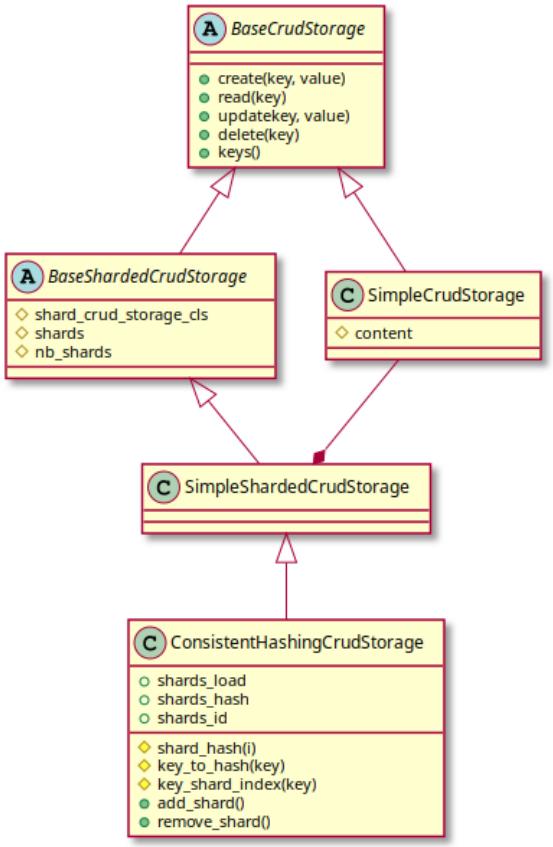

Figure 1: Hiérarchie des classes

La classe est initialisée avec le nombre de *shards* Q, et le nombre de hash N. De même, les informations des *shards* sont disponibles via les propriétés publiques `shards_load`,

`shards_hash` et `shards_id` triés selon leur *hash* vu comme leur position dans le tableau circulaire représentant $[\![N]\!]$.

Dans l'implémentation, plusieurs structures de données sont utilisées:

1. la liste `shards` stocke dans l'ordre de leur identifiants, les *shards* instances de la classe `ConsistentHashingCrudStorage.shard_crud_storage_cls`
2. les tableaux numpy `sorted_shards_ids` et `sorted_shards_hashs` stockent respectivement les identifiants et les *hashes* des *shards* selon leur position dans le tableau circulaire représentant $[\![N]\!]$
3. le tableau numpy `hash_to_shard_id` stocke les attributions de l'identifiant du *shard* pour chaque *hash*

1.2 Propriétés et expérimentations

1. **[4pts]** Donner (ou traduire) la définition de la *charge* ("*load*") à l'aide de [1]. Étudier l'évolution la distribution empirique de la charge pour N fixé à 10000 et Q croissant.
2. **[4pts]** Donner (ou traduire) la définition de la *régularité* ("*smoothness*") à l'aide de [1]. Dans le cas considéré pour ce TD, comment pouvez vous la calculer à partir des valeurs des charges? Étudier l'évolution de la régularité pour N fixé à 10000 et Q croissant.

Dans les deux cas commenter brièvement les résultats.[2]

Pour cette partie, un script minimal très simple est donné `key_value_part2/distribution.py`.

1.3 Ajout d'un nouveau *shard*

La méthode `ConsistentHashingCrudStorage.add_shard` est dédiée à l'ajout d'un nouveau *shard* dans le système de stockage. Cette partie vise à implémenter progressivement cette méthode.

1.3.1 Formalisation de l'ajout, et algorithme

L'ajout d'un nouveau *shard* p se réalise ainsi:

1. p est associé à son *hash* $h(p)$, encadré par deux autres $h(m)$, $h(l)$ respectivement du shard m précédent et du shard l suivant
2. V_p est construit comme sous ensemble de V_m, le stockage du *shard* m précédent :

$$V_p \leftarrow \{\{k_i, v_i\} \in V_m \mid h(k_i) \in [\![h(p), h(l)[\![_N\}$$

[2] Qu'observez-vous? Quelles explications de vos observations pouvez-vous donner? Répondre avec des phrases simples et courtes, idéalement une phrase par observation ou explication et maximum 3 phrases: pensez à vos correcteurs.

3. V_m se voit retrancher V_p

$$V_m \leftarrow V_m \backslash V_p$$

Avec ces structures de données, on se donne l'algorithme suivant pour l'ajout d'un *shard* p.

Algorithme pour l'ajout de *shard*

1. Sachant que les identifiants sont attribués de manière contigüe à partir de 0, trouver p, l'identifiant du dernier *shard* .
2. Instancier le nouveau *shard* et l'insérer à la fin de la liste privée `shards`. On utilisera l'attribut de classe `ConsistentShardingCrudStorage.shard_crud_storage_cls`
3. Chercher la position du *shard* dans $[\![N]\!]$ grâce à son *hash* .
4. Mettre à jour les deux tableaux `sorted_shard_ids` et `sorted_shard_hashs` en préservant l'ordre sur les *hashes* .
5. Récupérer les *shards* le précédent et le suivant en récupérant d'abord leurs identifiants.
6. Trouver la plage de *hashes* à attribuer à ce nouveau *shard* .
7. Mettre à jour le tableau `hash_to_shard_id` en conséquence.
8. Effectuer la migration de éléments du précédent *shard* contenus dans cette plage vers le nouveau *shard* . On pourra utiliser la méthode `keys` des shards.

1.3.2 Implémentation

[6pts] Implémenter la méthode `ConsistentShardingCrudStorage.add_shard`, en suivant les étapes de l'algorithme.

Nous recommandons l'usage des fonctions numpy suivantes:

1. `searchsorted`
2. `insert`
3. `concatenate`
4. `arange`

Si votre implémentation est correcte, 18 tests associés doivent passer. Vous pouvez les lancer avec.

```
make test-add
```

1.3.3 Analyse de la complexité

1. **[4pts]** En supposant une charge uniforme sur les *shards* , donner dans un tableau et **sans justifier** la complexité temporelle en temps des étapes 1 à 8 données précédemment en fonction de N, Q, et de n le nombre de données de S stockées dans le système.

2. **[3pts]** À l'étape 4, deux listes sont utilisées pour maintenir un tri sur les identifiants et les *hashes* des *shards* . Il y aurait-il des structures de données plus adaptées pour cette complexité? Si oui, quel serait alors la complexité temporelle de l'étape 4 ?

3. **Question bonus [8pts bonus]**: changer l'implémentation (à votre convenance) pour utiliser des structures de données plus adaptées. L'attribution des points pour cette question se fera en fonction de la clarté et de l'efficacité de l'implémentation.

1.4 Suppression du dernier *shard*

1.4.1 Algorithme

[4pts] Donner l'algorithme de suppression du dernier *shard* en plusieurs étapes, similairement à celui donné dans la section précédente pour l'ajout de *shard* .

1.4.2 Implémentation

Question bonus [8pts bonus] En se basant sur le code développé à la section précédente, implémenter la méthode `ConsistentHashingCrudStorage.remove_shard` dédiée à supprimer le dernier *shard* du système de stockage.

Si votre implémentation est correcte, 8 tests associés doivent passer. Vous pouvez les lancer avec.

```
make test-remove
```

1.5 Ouverture

1. **[1pts]** Le *consistent hashing* est utilisé dans certains technologies de bases de données relationnelles, comme PostGreSQL, pour répartir les données sur plusieurs machines. Donner la clause SQL du dialecte PostGreSQL pour définir un partitionnement des données reposant sur le *consistent hashing*. Fournir une référence d'autorité documentant cette clause.

2. **[1pts]** Donner le contexte historique du Web au moment de la création du *consistent hashing* et une ressource d'autorité en la matière. En particulier, on peut citer un évènement économique ayant commencé dans les années 1990 et s'étant arrêté au début des années 2000.

2 Rendus et critères d'évaluation

L'ensemble des rendus devront être fait via votre repo Gitlab nominatif. Sont attendus sur la branche `main` (seule cette branche sera utilisée pour l'évaluation):

- Votre version du dossier `/key_value_part2/`,
- Votre rapport en lieu est place du fichier `/report.pdf`.

NB: seuls ces fichiers seront copiés vers l'environnement d'évaluation: **ne modifiez pas d'autres fichiers, vous pourriez vous tirez une balle dans le pied**.

2.1 Date de rendu

La date du rendu est fixée au **11/05/2021 à 14h00** (heure de Compiègne).

- Si le rendu est en avance de t (en secondes), un bonus de $20\% \times (1 - \exp(-t/T_0))$ sera appliqué, avec $T_0 = 84600s$. Exemples:

  5 jours d'avance: 19.9% de bonus.

  2 jours d'avance: 17.3% de bonus.

  1 jour d'avance: 12.6% de bonus.

  6 heures d'avance: 4.4% de bonus.

- Si le rendu est en retard de t (en seconde), un malus de $20\% \times (1 - \exp(t/T_0))$ sera appliqué. Avec $T_0 = 84600s$. Si le malus est supérieur à 100%, la note sera mise à 0. Exemples:

  6 heures de retard: 5.7% de malus.

  12 heures de retard: 13.0% de malus.

  24 heures de retard: 34.4% de malus.

  36 heures de retard: 69.6% de malus.

  supérieur à 43 heures de retard: note mise à 0.

La date du dernier merge sur la branche `main` (la date est celle du gitlab) fera foi.

2.2 Détails de notations

Critères d'évaluation La note est donnée sur 30 et vous disposez de 15 points potentiels de bonus.

Les points sur les questions d'implémentations seront attribués ainsi:

- (70%) Les tests passent. Au prorata des tests qui passent.
- (30%) Respect du style et de la qualité du code.

Les points sur les autres questions seront attribués ainsi:

- (70%) Véridicité des réponses.
- (20%) Clarté et concision de la formulation des réponses; mise en page[3].
- (10%) Ressources mobilisées de qualité et/ou d'autorité.

Critère éliminatoire Impliqueront directement un 0/20:

- Rapport de plus d'**une page** dans cette configuration (notez bien la taille des marges et de la police):

```
\documentclass[12pt]{article}
\usepackage[a4paper,margin=3cm]{geometry}
```

- Plagiat de code[4].
- L'absence d'un des fichiers pour la correction,
- **Un fichier qui n'est pas anonyme** (mention de votre nom, login CAS, etc).

Également, merci d'avance de ne pas essayer de contourner ou de nuire au système d'évaluation...

References

[1] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web. https://www.cs.princeton.edu/courses/archive/fall07/cos518/papers/chash.pdf, 1997.

[3] Moins, c'est souvent mieux que beaucoup.

[4] Vous pouvez travailler ensemble et vous entre-aider, mais soyez honnête avec nous et avec vous-même lors de votre soumission.