TLC projet

Antoine LEVAL, Damien Vansteen

February 10, 2019

Abstract

Le projet de TLC était à réaliser avec Google Cloud Platform. Le but était de fournir un service REST permettant d'ajouter et d'accéder à des données de suivis de course à pieds de plusieurs utilisateurs. Nous avons choisis d'utiliser Java pour ce projet, nous nous sommes servis du squelette Java 8 fournis par le gitlab du projet. Le code source du projet est téléchargeable à l'adresse suivante : https://github.com/Viinyard/TLC_Project.

1 Le service

Le service REST doit pouvoir stocker et accéder à des données NoSQL grâce à des filtres simple. (NoSQL ne permet pas de requêtes complexe sur les données).

- `public void bulkAdd(RecordList toAdd)`

```
for (Record r : toAdd) {
   Key recKey = datastore.allocateId(recordsKey.newKey());
   Entity record = Entity.newBuilder(recKey)
      .set("id", r.id)
      .set("lat", r.lat)
      .set("lon", r.lon)
      .set("user", r.user)
      .set("timestamp", r.timestamp)
      .build();
   datastore.put(record);
}
```

On laisse le datastore s'occuper de générer la clé. Gérer la clé nous même nous aurait apporté quelques avantages, notamment une recherche plus rapide et plus direct directement sur les clés, plûtot que sur des recherches un peu plus lente sur les données. En l'absence d'application côté client il nous aurait été difficile de stocker les clés en mémoire. Le serveur ne vas pas garder toute les clés, chaque client aurait dû gardé en mémoire toute les clés en rapport avec sa course pour un accès plus rapide par la suite.

- `public void bulkDelete()`

```
String[] run_ids = getRequest().getAttributes().get("list").toString().split(",");

List<Key> listKey = new ArrayList<>();
for (String r : run_ids) {
    Query q = Query.newEntityQueryBuilder().setKind("record")
    .setFilter(StructuredQuery.PropertyFilter.eq("id", Integer.valueOf(r))).build();
   for (QueryResults<Entity> it = datastore.run(q); it.hasNext(); ) {
      Entity e = it.next();
      listKey.add(e.getKey());
   }
}

if(!listKey.isEmpty()) {
   datastore.delete(batch(Key.class, listKey));
}
```

Pour le bulk delete on fait une query sur chaque run id, chaque query n'a qu'un seul filtre, on ajoute toutes les clé trouvé par la recherche à une liste de clé, on peut ensuite passer l'ensemble de clé dans une unique requête delete.

- `public RecordList search()`

```
List<StructuredQuery.PropertyFilter> filters = new ArrayList<>();

// pour chaque parametre
if(parameter.getValue().contains(",")) {
   String[] values = parameter.getValue().split(",");
   filters.add(StructuredQuery.PropertyFilter.ge(parameter.getName(),
       Long.parseLong(values[0])));
   filters.add(StructuredQuery.PropertyFilter.le(parameter.getName(),
       Long.parseLong(values[1])));
} else {
   filters.add(StructuredQuery.PropertyFilter.eq(parameter.getName(),
       Long.parseLong(parameter.getValue())));
}

// une fois la liste des filtres complete
EntityQuery.Builder eqb = Query.newEntityQueryBuilder().setKind("record");
if(!filters.isEmpty()) {
    if(filters.size() == 1) {
      eqb.setFilter(filters.get(0));
   } else {
      eqb.setFilter(StructuredQuery.CompositeFilter.and(filters.get(0),
          batch(StructuredQuery.PropertyFilter.class, filters.subList(1,
          filters.size() -1)))).build();
   }
}

Query q1 = eqb.build();
// on construit la reponse rest
RecordList res = new RecordList();
for (QueryResults<Entity> it = datastore.run(q1); it.hasNext(); ) {
   Entity r = it.next();
   res.add(new Record(
       (int) r.getLong("id"),
      r.getDouble("lat"),
       r.getDouble("lon"),
      r.getString("user"),
      r.getLong("timestamp")));
}
```

Pour la méthode de recherche on s'occupe de savoir si c'est une recherche d'égalité ou d'interval, si une seule valeur c'est une égalité sinon c'est une recherche sur un interval. Ici nous avons dû faire en sorte de bien typer les paramètres dans les méthodes PropertyFilter car en laissant le type String pour un timestamp par exemple la recherche ne fonctionnait pas.

Pour un interval nous avons pris la première valeur avec une recherche `PropertyFilter.ge(..)` (Greater Than or Equal) et une recherche `PropertyFilter.le(..)` sur la deuxième valeur (Less Than or Equal).

Pour une recherche d'égalité nous avons utilisé `PropertyFilter.eq(..)`.

Enfin nous avons lié tous les filtres via un composite filter **and**.

2 Test de montée en charge

Pour les tests de montée de charge, et le benchmarking de notre service google cloud nous avons choisis d'utiliser le célèbre outil **Apache JMeter**.

Grâce à l'outil graphique, le fait de pouvoir ajouter des scripts en Java, des assertions, etc, il est très simple d'écrire les tests de montée en charge.

Par exemple pour générer le payload d'une recherche bulk add nous avons écris un simple script générant aléatoirements jusqu'à 10 entrées par requêtes :

```
StringBuilder result = new StringBuilder();
String newline = System.getProperty("line.separator");

Random random = new Random();

String[] users = new String[]{"leo", "lea", "anais", "adrien", "erwan", "antoine", "max",
    "foo", "theo", "julie", "zoe"};
int max = random.nextInt(10);
result.append("[");
for(int i = 0; i <= max; i++) {
   result.append("{");
   result.append("\"id\":"+random.nextInt(100)+",\n");
   result.append("\"lat\":"+ ((random.nextDouble() * 10) + 40) + ",\n");
   result.append("\"lon\":"+((random.nextDouble() * 10) + 40) + ",\n");
   result.append("\"user\":" + "\"" + users[random.nextInt(users.length)] + "\"" + ",\n");
   result.append("\"timestamp\":" + System.currentTimeMillis()+"\n");
   result.append("}");
   if(i < max) {
      result.append(",\n");
   }
}
result.append("]");

vars.put("json", result.toString());
```

Nous avons crée plusieurs benchmark, une une l'insertion, une sur la suppression, et une sur chaque paramètre de recherche.

Chaque requête a une assertion sur le code réponse de la requête, afin de nous assurer pendant les tests que chaque requête passe correctement, et être notifié en cas d'erreur sur une des requêtes.

Ainsi nous avons lancé les tests sur chaque requête avec 100 utilisateurs faisant chaque 1000 requêtes.

Les premières centaines de requêtes se passaient bien jusqu'à atteindre la limite des ressources gratuites de google cloud platform, il nous aurait fallu activer la facturation pour pouvoir continuer le benchmarking. Comme on peut le voir sur la figure suivante, une fois les ressources gratuites épuisée le serveur renvois une erreur 500.

Le fichier `TLC-benchmark.jmx` est trouvable à la racine du projet sur le dépôt github.

Figure 1: Caption