MongoDB : MapReduces, Optimisations et performances

MapReduces

Introduction

- Limites de aggregate
- Copie des données transformées dans une autre collection
- traitement plus rapide
- fonctions pure JS

Utilisation

```
>db.collection.mapReduce(
   function() {emit(key,value);},
   function(key,values) {return reduceFunction},
   {
      out: collection,
      query: document,
      sort: document,
      limit: number,
      finalize : function
   }
)
```

3 Paramètres :

- map: Fonction javascript qui mappe une valeur à une clé et emet la paire clé/valeur via la fonction `emit(key, value)`
- reduce: Fonction javascript qui réduit ou regroupe tout les documents ayant la même clé
- options : Objet contenant les options de fonctionnement du mapReduce

Options :

- out: Spécifie l'emplacement du résulat du mapReduce.
- query: Précise les critères de sélection
- sort: Précise les critères de tri
- limit: Indique le nombre maximal de documents à retourner
- finalize : Fonction permettant de modifier la valeur retournée par le reduce.

Exemple : Affichage du nombre de films par genre depis la collection suivante :

```
{
    nom : "nom du film",
    genre : "genre du film"
}
```

Réponse :

```
var map = function() {
    emit(this.genre, 1);
};

var reduce = function(key, values) {
    return Array.sum(values)
};

var count = db.movies.mapReduce(map, reduce, {
    out: "movies_result"
});

>db.movies_result.find();
```

Exlications :

- Pour chaque film, dans la fonction map, on prend comme clé le genre du film et comme valeur `1` pour dire qu'on a un film de ce genre.
- Dans la fonction reduce, on dit qu'on fait un `SUM` de toutes les valeurs correspondant à un genre (donc tous les `1` retournés pour ce genre).
- On dit qu'on met le résultat dans la collection `movies_result`
- On affiche le contenu de `movies_result`

Bien :

- Gain de temps : La requête du mapReduce n'est faite qu'une seule fois. pour voir le résultat plus tard, vous devrez juste regarder la collection contenant le résultat.
- Simple : Tout marche par groupe de clé/valeur

Pas bien :

- Trop simple : Comment ça se passe si on veut travailler sur plusieures clés en même temps ?
- Temps réel: Comment faire pour avoir les résultats qui se mettent automatiquement à jour dans la table résultat ?

Travailler sur plusieurs clés

- le `emit()` peut être appelé autant de fois que necessaire dans la fonction `map`.
- Vous pouvez très bien faire des groupement de `if`, `while`, `for`, ... dans vos différentes fonctions vu que c'est du JS.
- `emit(key, value)` peut prendre un objet pour le paramètre `value`.

Nous verrons comment faire exactement en TP.

Mise à jour de la table résultat

- Modification du paramètre `out` pouvant prendre l'objet suivant :

```
{
    <action>: <collectionName>,
    [db: <dbName>]
}
```

- action : Peut prendre les valeurs suivantes :
  - replace : Remplace tout le contenu de la collection du résultat par le contenu du mapReduce.
  - merge : Ajoute un résultat si la clé n'existe pas et écrase les clés existantes.
  - reduce : Fait comme le merge, sauf que si la clé existe, réapplique la fonction `reduce` sur les anciens et nouveaux résultats pour cette clé.
- collectionName : Collection contenant le résultat de la requête.
- db : Base de donnée où vous voulez créer la collection.

Optimisation et performances

- Les index
- `explain`
- profiling
- replicaSet et Sharding

Les index

Différents types d'index :

- simple field
- compound (plusieurs fields)
- multi key (plusieurs fields de sous collections)
- text
- hash
- geospatial

Création d'un index

```
db.collection.createIndex(
  <key and index type specification>,
  <options>
);
```

Exemple :

```
db.test.createIndex({ name: 1 });
db.vendeurs.createIndex({ 'objectif.ca' : -1 });
```

- Créer un index sur le champ `name` dans la collection `test` avec un tri ascendant.
- Créer un index sur le champ `ca` du field `objectif` dans la collection `vendeurs` avec un tri descendant.

Text

```
db.collection.createIndex({
  name : "text",
  city : "text"
});
db.collection.createIndex({ "$**": "text" });
```

> Le field `$**` est un mot clé permetant de sélectionner tous les champs compatibles (objet courant, sous objet et sous collection) avec l'index Text.

Attention : Une collection ne peut avoir qu'un seul index de type texte.

Text

Permet l'utilisation de l'opérateur `$text` dans une query :

```
db.collection.find({
  $text : {
    $search : "John Doe Ingénieur"
  }
});
```

> Recherche dans tous les fields de l'index si les mots `John`, `Doe` ou `Ingénieur` existent. Si oui, retourne le résultat.

Depuis la version 3.2, les recherches avec `$text` sont insensitives et les accents sont remplacés par la lettre de base. Les `É, é, E, e, ...` sont vu comme un `e`.

Les options des index

Les options sur les index sont les suivantes :

- unique
- partial (arrive avec la 3.2)
- collation (arrive avec la 3.4)
- TTL

Unique

```
db.collection.createIndex({ key : 1 }, { unique : true });
db.vendeur.createIndex({
  numSecu   : 1,
  firstname : 1,
  lastname  : 1
}, {
  unique : true
});
```

> Pour le second exemple, on ne peut pas avoir 2x le même vendeur ayant à la fois le même numéro de sécu, nom et prénom.

Partial

Permet de donner un index uniquement aux documents répondant à la requête de l'index.

```
db.restaurants.createIndex(
   { cuisine: 1, name: 1 },
   { partialFilterExpression: { rating: { $gt: 5 } } }
)
```

> Index uniquement sur les champs `cuisine` et `name` les restaurants ayant une note supérieure à 5.

Le `explain()`

Permet de savoir comment une requête fonctionne selon 3 types :

- queryPlanner
- executionStats
- serverInfo

```
db.collection.find().explain('queryPlanner');
db.collection.find().explain('executionStats');
db.collection.find().explain('serverInfo');
```

queryPlanner

```
{
   "queryPlanner" : {
      "plannerVersion" : <int>,
      "namespace" : <string>,
      "indexFilterSet" : <boolean>,
      "parsedQuery" : { ... },
      "winningPlan" : {
         "stage" : <STAGE1>,
         ...
         "inputStage" : {
            "stage" : <STAGE2>,
            ...
            "inputStage" : { ... }
         }
      },
      "rejectedPlans" : [
         <candidate plan 1>,
         ...
      ]
   }
}
```

queryPlanner

Quelques fields importants :

- `explain.queryPlanner.indexFilterSet` : Un Booléen qui dit si oui ou non un index a été utilisé.
- `explain.queryPlanner.winningPlan.stage` : Nom dustage. Si jamais il a la valeur `IXSCAN`, vous retrouverez ensuite quels sont les indexs utilisés. dans `inputStage` ou `inputStages`

executionStats

```
"executionStats" : {
   "executionSuccess" : <boolean>,
   "nReturned" : <int>,
   "executionTimeMillis" : <int>,
   "totalKeysExamined" : <int>,
   "totalDocsExamined" : <int>,
   "executionStages" : { ... },
   "allPlansExecution" : [
      { <partial executionStats1> },
      { <partial executionStats2> },
      ...
   ]
}
```

executionStats

Quelques fields importants :

- `explain.executionStats.nReturned` : Nombre de documents retournés par la requête.
- `explain.executionStats.executionTimeMillis` : Temps de la requête
- `explain.executionStats.totalKeysExamined` : Nombre d'entrées d'index scannés
- `explain.executionStats.totalDocsExamined` : Nombre de documents scannés

Profiling

3 niveaux disponibles :

- `0` : aucun profiling au sein de la base de donnée
- `1` : collecte uniquement les requêtes lentes. Une requête est dite lente si elle dépasse les 100 ms d'exécution
- `2` : collecte tout de A à Z mais peut ralentir fortement la base de donnée selon le volume d'opérations.

```
db.setProfilingLevel(1);
```

Profiling

> Niveau intéressant : `1` couplé ensuite avec le `explain()` sur les requêtes lentes.

Toutes les requêtes sont stoquées dans la collection `system.profile`.

Profiling

```
{
   "op" : "query",
   "ns" : "test.c",
   "query" : {
      "find" : "c",
      "filter" : {
         "a" : 1
      }
   },
   "keysExamined" : 2,
   "docsExamined" : 2,
   "cursorExhausted" : true,
   "keyUpdates" : 0,
   "writeConflicts" : 0,
   "numYield" : 0,
   "nreturned" : 2,
   "responseLength" : 108,
   "millis" : 0,
   "execStats" : { ... }
}
```

Profiling

Quelques champs utiles :

- `op` : Type d'opération (`query` , `insert` , ...)
- `ns` namespace de la forme `<base de donnée>.<collection>` .
- `query` : La requête exécutée.
- `keysExamined` : Combien d'index ont été scannés.
- `docsExamined` : Combien de documents ont été scannés.
- `millis` : Temps d'exécution
- `execStats` : Statistiques d'exécution. Ressemble beaucoup à ce que l'on a vu dans `explain()` .

Profiling

Retourner les 10 dernières requêtes lentes :

```
db.system.profile.find().limit(10).sort({ ts : -1 }).pretty()
```

Retourne les requêtes lentes d'une collection donnée :

```
db.system.profile.find({ ns : 'mydb.test' }).pretty()
```

Retourne les 5 événements les plus récents :

```
show profile
```

Autres optimisations possibles

- Mongotop : Permet de savoir quelle est la vitesse de lecture/ecriture du serveur mongo.
- Mongostat : Fonctionne comme vmstat mais uniquement pour le seveur mongo (load average, nombre de processus en queue, ...).
- ReplicaSet : Reéplique les données sur x serveurs permettant d'éviter une coupure si un serveur tombe en panne
- Sharding : Obligatoirement couplé avec du ReplicaSet. Permet de scinder sur plusieurs pools de sharding une collection pour un accès plus rapide.