Rapport de stage

- Index selection pour la base de données NoSQL dans le nuage

Lei Tianqi

Juin 2014

ÉCOLE POLYTECHNIQUE
UNIVERSITÉ PARIS-SACLAY

I. Présentation du stage

1.1 Sujet de stage
Hypothèse :
Utiliser des système hétérogène de stockage et gestion de données qui peut conduire à des meilleurs performances que si on est dans un seul système.

Le but de stage :
1. Un benchmark pour vérifier cette hypothèse.
2. trouver des guidelines / règles générales pour choisir quels fragments de données stocker où dans une configuration multi-systèmes.

1.2 Équipe de travail
Inria OAK : Ioana Manolescu, Francesca Bugiotti, Tianqi LEI
OAK team web site : http ://team.inria.fr/oak
Inria Saclay web site : http://www.inria.fr/en/centre/saclay

1.3 Périodes et horaires du stage
Le stage s'effectue de 14 avril 2014 à 14 juillet (3 mois).
35 heures par semaine.

1.4 Outils utilisés :
Langage du programmation : Java
Système de base de données : PostgreSQL, HBase, Oracle Nosql
Système d'exploitation : Mac OS, ubuntu

1.5 Outil de synchronisation : SVN
1.6 Références :
Postgresql : http://docs.postgresql.fr/9.3/pg93.pdf
HBase : http://archive.cloudera.com/cdh4/cdh/4/hbase/book.html
Oracle Nosql :
http://docs.oracle.com/cd/NOSQL/html/GettingStartedGuideTables/Oracle-NoSQLDB-GSG-Tables.pdf
http://docs.oracle.com/cd/NOSQL/html/GettingStartedGuide/Oracle-NoSQLDB-GSG.pdf

II. Contexte

Pour commencer à présenter le contenu du stage, il est nécessaire d'introduire les 2 types de base de données : la base de données relationnelle et la base de données non-relationnelle et les 3 systèmes de base de données que nous avons utilisés durant mon stage : PostgreSQL, HBase, Oracle Nosql.

2.1 Introduction les 2 types de base de données

2.1.1 La base de données relationnelle

La base de données relationnelle est un stock d'informations décomposées et organisées dans des matrices relations ou tables conformément au modèle de données relationnel.

Dans une base de données relationnelle, les informations sont stockées sous forme de groupe de bailleurs : les enregistrements. Un ensemble d'enregistrements relatif à un sujet forme une relation et est stocké dans une table. La base de données comporte une ou plusieurs tables et les sujets sont connexes.

2.1.2 La base de données non-relationnelle (NoSQL)

L'expression NoSQL, la pluspart du temps interprétée comme Not Only SQL, a été utilisée pour la première fois en 1998. NoSQL(not only SQL) désigne une catégorie de systèmes de gestion de base de données (SGBD) qui n'est plus fondée sur l'architecture classique des bases relationnelles. L'unité logique n'y est plus la table, et les données ne sont en général pas manipulées avec SQL.

NoSQL renonce aux fonctionnalités classiques des SGBD relationnels au profit de la simplicité. Un modèle typique en NoSQL est le système clé-valeur. De grands acteurs d'Internet, notamment Google(BigTable), Amazon (Dynamo), LinkedIn (project Voldemort), Facebook (Cassandre Project puis HBase), SourceForge.net (MongoDB), Ubuntu One (CouchDB), etc., conçoivent et exploitent des bases de données de type NoSQL.

2.2 Introduction les 3 systèmes de base de données utilisés

i). PostgreSQL :

PostgreSQL existe depuis 1995 et il est un système de gestion de base données relationnelle et objet (SGBDRO). C'est un outil libre disponible selon les termes d'une licence de type BSD.

PostgreSQL peut stocker plus de types de données que les types traditionnels entier, caractères, date, time, boolean, …etc. L'utilisateur peut créer des types, des fonctions, utiliser l'héritage de type, etc.

PostgreSQL fonctionne sur Solaris, SunOS, Mac OS X, HP-UX, AIX, LINUX, etc. et est largement reconnu pour son comportement stable.

ii). HBase :

HBase est un système de gestion de base de données non-relationnelle distribué, écrit en Java, disposant d'un stockage structuré pour les grandes tables. HBase est inspirée par publications de Google sur BigTable. Comme BigTable, elle est une base de données orientée colonnes. HBase est un sous-projet d'Hadoop, un framework d'architecture destituée et s'installe généralement sur le système de fichiers HDFS d'Hadoop pour faciliter la distribution, même si ce n'est pas obligatoire.

iii). Oracle Nosql

Oracle Nosql est un système de gestion de base de données non-relationnelle distribué (key-value) qui est conçu pour fournir un stockage de données très fiable, scalable et données disponible dans un ensemble configurable de système qui fonctionne comme noeuds de stockage.

III. Générateur de données et Les requêtes

Mon stage peut être séparé en 2 parties (A et B) :

A. 3 installations « locales » :

Dans cette partie-là, nous ne nous intéressons que la performance d'un seul système, c'est-à-dire que nous détectons les 7 requêtes sur les 3 systèmes respectivement comme suivante :

- Nous utilisons seulement PostgreSQL pour stocker les données et lancer les 7 requêtes :
  - Données + Requêtes -> PostgreSQL
- Nous utilisons seulement HBase pour stocker les données et lancer les 7 requêtes :
  - Données + Requêtes -> HBase
- Nous utilisons seulement Oracle Nosql pour stocker les données et lancer les 7 requêtes :
  - Données + Requêtes -> Oracle Nosql

B. Installations « mixtes » :

Selon le résultat de la première partie (A), nous concevions un nouveau système hétérogène de stockage et gestion de données qui peut conduire à des meilleurs performance que si on est dans un seul système. Par example :

- Soit nous utilisons les 2 système pour stocker les données et lancer les 7 requêtes :
  - Données + Requêtes -> Postgresql + HBase
  - Données + Requêtes -> HBase + Oracle Nosql

    …
- Soit nous utilisons tous 3 systèmes pour stocker les données et lancer les 7 requêtes :
  - Données + Requêtes -> Postgreql + HBase + Oracle Nosql

Afin de rechercher les performances sur les différents systèmes de base de données, il faut d'abord avoir un « générateur de données » qui peut générer les données automatiquement et charger dans les 3 systèmes de base de données (PostgreSQL, HBase, Oracle Nosql).

Pour la première partie, l'idée est comme suivant :

- Développer un Java code qui peut générer les donnée et écrire dans un fichier de « .txt ».
- Après d'avoir générer les données, le programme commence à les charger dans les 3 système de base de données (nous devons assurer que les 3 système de base de données stockent les même informations).

3.1 Conception de base de données et les requêtes

Après avoir discuté avec mon tuteur de stage, nous avons décidé à concevoir une base de données d'un magasin qui simule Amazon. Car nous avons 3 systèmes de base de données, il faut concevoir pour chaque système un model propre.

Nous avons les entités comme suivants :

1. Users (user id, name, age)
2. Shops (shop id, shop name, url)
3. items (item id, item name, price, url, description)
4. friend (userId1, userId2)
   - Signification : userId1 and userId2 sont amis.

5. order(order id, user id, shop id, date of order, total price)
   - Signification : l'utilisateur « user id » a créé une commande avec l'id : order id à la date « date of order » et le prix total est « total price ».

6. visit (user id, item id, date of visit, time of visit, buy)
   - Signification : L'utilisateur « user id » a visité le website de item : « item id » le « date of visit » à « time of visit ».
   - « buy » est un attribut boolean qui signifie si cet utilisateur a acheté cet article ou pas.

7. addr (street, city, state, userId)
   - Signification : L'utilisateur dont id est « user id » habite à street, city, state.

8. lineitem (orderId, itemId, quantity)
   - Signification : la commande avec l'id : « order id » contient « quantity » article(s) avec « itemId ».

La relation entre les nombres tuples de chaque entité est comme suivante :

```
public static final int N = *;
public static final int NB_USR = N;
public static final int NB_SHOP = N;
public static final int NB_ITEM = N;
public static final int NB_ADDR = N;
public static final int NB_FRIENDS = 5 * N;
public static final int NB_VISIT = 10 * N;
public static final int NB_ORDER = 10 * N;
public static final int NB_LINEITEM = 10 * N;
```

Nous devons assurer que le programme est assez générique, car à la fin, nous probablement avons 3 systèmes de différents dimensions. Pour l'instant, mon directeur décide à avoir 3 dimensions comme suivante :

- Dimension « small » : N = 1000
- Dimension « medium » : N = 100,000
- Dimension « large » : N = 10,000,000

3.2 Description du générateur de données

- Les ids

  Nous utilisons la façon simple pour définir les id de chaque entité, par example :

  L'id de user : usr1, usr2, … usr100 …
  L'id de shop : shop1, shop2 … shop100 …
  …

- Url :

  Nous assurons que les urls doivent être unique, donc nous utilisons les ids de chaque tuple, par example : item34, shop95, etc. Comme cela, nous pouvons obtenir les urls uniques comme suivantes :

  - Url de « shop » :

    http://A.B.C/shop1
    http://A.B.C/shop2
    …

  - Url d'item :

    http://A.B.C/item1
    http://A.B.C/item2
    …

- Les textes :

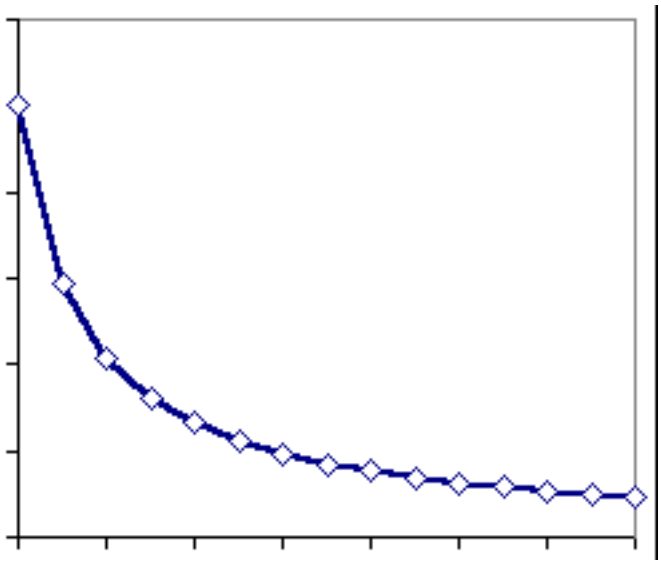
  Il y a beaucoup de texte qui seront stocké dans la base de donnée comme « nom de utilisateur », « nom de magasin », « nom d'item », « description d'item », etc.

  Pour éviter de générer des textes qui n'a aucun sens comme : « qsvtqn », « vwjiqzo », « vjqzp », mon programme lit un mot aléatoire depuis un dictionnaire que j'ai téléchargé sur Internet et j'ai stocké dans « ./tools/dictionary.txt ». Même si les mot lus ne correspondent pas du tout les nom des items, les noms des magasins, ou description d'item, mais c'est beaucoup mieux, voici un example des utilisateurs :

id_usr	name	sex	age
usr0	bunns haj	f	19
usr1	kingdoms haunter	f	10
usr2	lockup snakier	m	71
…	…	…	…

- La distribution des amis :

  Pour bien simuler la distribution des amis en réel, mon directeur m'introduit la distribution suivi la loi de Zipf. Voici une image qui décrit la distribution qui suivi la loi de Zipf, on peut supposer que l'axe de X est les ids des utilisateurs et l'axe de Y est le nombre des amis relié à cet utilisateur :



Pour bien détecter la performance, Nous devons assurer que pour chaque système, nous stockons les mêmes informations. Les entités que je viens de montrer est plutôt conçu pour la base de données relationnelle, mais pas celle de non-relationnelle (NoSQL). L'introduction de model de stockage de Nosql est dans le chapitre prochain.

3.3 Les requêtes utilisées pour détecter les performances des 3 systèmes différents

Pour détecter la performance de chaque base de données, nous supposons qu'il y a 7 requêtes différentes qui utilisent très souvent pendant l'utilisation :

Requête 1 : tout le profil de user t.q id= "usr1".
Requête 2 : l'âge de tous les utilisateurs de country : "Fr".
Requête 3 : le nombre de fois que chaque items a été acheté.
Requête 4 : les 10 items les plus vendus.
Requête 5 : les 10 items pas vendu du tout.
Requête 6 : les 100 pages les plus visitées
Requête 7 : les amis ayant achetés le même article

Nous Lançons les 7 requêtes dans les 3 bases de données pour savoir qui peut obtenir les résultats plus rapides. Après cela, nous pouvons trouver des guidelines / règles générales pour choisir quels fragments de données stocker où dans une configuration multi-systèmes (qui répond le but de mon stage).

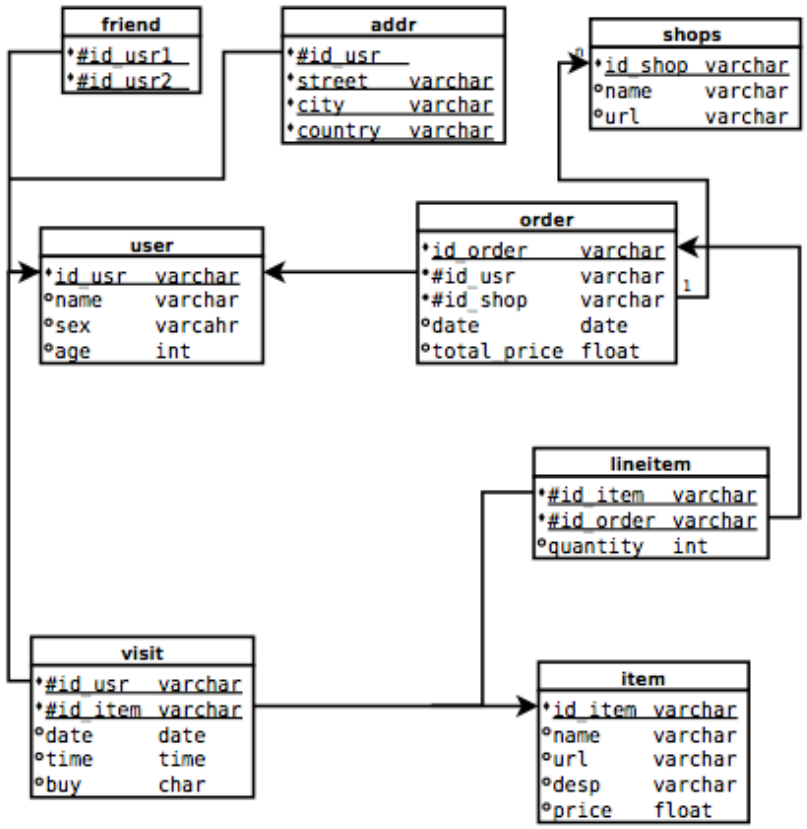
IV. Modèles de stockage des données

4.1 Le modèle de stockage dans PostreSQL

1. L'environnement

- Langage de programmation : Java
- Version de la base de données de Postgresql : v9.2
- Interface utilisateur : pgAdmin
- Pilotes : JDBC (postresql-9.2-1002)

Le premier générateur de données que j'ai fait est celui de PostgreSQL. Le schéma est comme suivant :



Creation en 1er ordre :
1. Le nom de la table : usr
Liste d'attributs (nom, type, description) :
id_usr : varchar (clé primaire)
name : varchar
sex : varchar
age : int

2. Nom de la table : shop
Liste d'attributs (nom, type, description) :
id_shop : varchar (clé primaire)
name : varchar
url : varchar

3. Nom de la table : item
Liste d'attributs (nom, type, description) :
id_item : varchar (clé primaire)
name : varchar
price : float
url : varchar
description : varchar (description d'item)

Creation en 2nd ordre

4. Nom de la table : friend
Liste d'attributs (nom, type, description) :
id_usr1 : varchar (clé étrangère)
id_usr2 : varchar (clé étrangère)
(clé primaire : id_usr1 et id_usr2)
signification : id_usr1 et id_usr2 sont amis.

5. Nom de la table : ordered
List d'attributs (nom, type, description) :
id_order : varchar (primary key)
id_usr : varchar
id_shop : varchar
date_order : date
total_price : float
signification :
L'utilisateur : id_usr a créé une commande dont l'id est id_order, daté date, et le prix total de cette commande est total_price.

6. Nom de la table : visit
Liste d'attributs (nom, type, description) :
id_usr : varchar (clé étrangère)
id_item : varchar (clé étrangère)
date_visit : date
time_visit : time
buy : char
(primary key : id_usr, id_item, date_visit, time_visit)

signification :
L'utilisateur dont l'id est id_usr a visité le site du item : id_item daté date à time.
L'attribut « buy » signifie si cet utilisateur a acheté cet item :
« y » signifie que « il a visité cet article et il l'a acheté ».
« n » signifie que « il a visité cet article, mais il ne l'a pas acheté ».

7. Nom de la table : addr (address)
Liste d'attributs (nom, type, description) :
street : varchar
city : varchar
country : varchar
id_usr: varchar (clé étrangère)
(clé primaire : id_usr, street, city, country)
signification :
L'utilisateur dont l'id est id_usr habite à street, city, country.

Creation en 3ème ordre

8. Nom de la table : lineitem
Liste d'attributs :
id_order : varchar (clé étrangère)
id_item : varchar (clé étrangère)
quantity : int (combien de l'article acheté)
(clé primaire : id_order et id_item)
signification :
La commande dont l'id est id_order contient « quantity » article dont l'id est « id_item ».

2. Chargement les données générés dans PostgreSQL
En général, il y a 4 façon de chargement les données dans PostgreSQL en utilisant JDBC :

- L'exécution des requêtes en utilisant « insert » pour chaque tuple.
- Utilisation de « prepared statements ».
- Bulk insert.
- Insert basé sur « COPY FROM ».

Après les conseils de mon tuteur de stage, la vitesse de chargement les données plus vite est comme suivant :

- Générer les données et les écrire dans des fichiers (le fichier « txt », par example).
- Utiliser la commande « Copy » qui lit les données depuis les fichiers, et voici une introduction de la commande « Copy » :

```
COPY table_name [ ( column_name [, ...] ) ]
    FROM { 'filename' | STDIN }
    [ [ WITH ] ( option [, ...] ) ]

COPY { table_name [ ( column_name [, ...] ) ] | ( query ) }
    TO { 'filename' | STDOUT }
    [ [ WITH ] ( option [, ...] ) ]
```

Voici un example de chargement les données en Java :

```
public void copyFromFile(String fileName, String tableName)
        throws SQLException, IOException {
    CopyManager copyManager = new CopyManager((BaseConnection) conn);
    FileReader fileReader = new FileReader(fileName);

    copyManager.copyIn("COPY " + tableName
            + " FROM STDIN WITH DELIMITER ';'", fileReader);
}
```

Explication du code:

1. Créer un objet « FileReader » que le fichier contient toutes les tuples d'un entité.
2. Utiliser la commande « Copy » pour charger les tuples dans la base de données.

4.2 Le modèle de stockage dans HBase

1. L'environnement

- Langage de programmation : Java
- Version de la base de données de HBase : 0.97.17
- Interface utilisateur : non
- Pilotes : hbase-0.94.17

HBase est une base de données non-relationnelle, donc la façon de conception le modèle de données est vraiment différent que celle de base de données relationnelle. Et voici les connaissances de base :

- **Table** : dans HBase les données sont organisées dans des tables. Les noms de tables sont des chaîne de caractères.
- **Row** : dans chaque table les données sont organisées dans des ligne. Une ligne est identifiée par une clé unique(RowKey). La Rowkeys n'a pas un type, elle est traité comme un tableau d'octets.
- **Column Family :** Les données au sein d'une ligne sont regroupées par column family. Chaque ligne de la table a les mêmes column family, qui peuvent être peuplées ou pas. Les column family sont définit à la création de la table dans HBase. Les noms des column family sont des chaines de caractères.
- **Column qualifier :** L'accès aux données au sein d'une column family se fait via le column qualifier ou column. Ce dernier n'est pas spécifié à la création de la table mais plus tôt à l'insertion de la donnée. Comme les rowkeys, le column qualifier n'est pas typé, il est traité comme comme un tableau d'octets.
- **Cell :** La combinaison du RowKey, de la Column Family ainsi que la Column qualifier identifie d'une manière unique une cellule. Les données stockées dans une cellule sont appelée les valeurs de cette cellule. Les valeurs n'ont pas de type, ils sont toujours considérés comme tableau d'octets.

Il faut savoir que le schéma de la base de données non-relationnelle dépend de son utilisation, donc pour concevoir le schema de HBase, il faut bien penser les requêtes qui seront utilisées pour évaluer sa performance.

Après discuter avec Francesca et mon tuteur, nous arrivons à avoir un modèle de base de données comme suivant :

Creation en 1er ordre :

1. Nom de la table : usr
   Row key : id_usr
   Column Family1 : info : name | sex | age
   Column Family2 : country : c1 | c2 | … | cN
   Column Family3 : address : a1 | a2 | … | aN

Example :

Row	Column+Cell
usr1	column=info : name=‘’Tianqi’’
usr1	column=info : sex=‘m’
usr1	column=info : age=35
usr1	column=country : c1=‘’Fr’’
…	…
usr1	column=country : cN=‘’Ch’’
usr1	column=address : a1=‘’street1, Paris, Fr’’
…	…
usr1	column=address : aN=‘’streetN, Peking, Ch’’

2. Nom de la table : shop
   Row key : id_shop
   Column Family : info : name | url

Example :

Row	Column+Cell
shop0	column=info : name=‘’Fnac’’
shop0	column=info : url=‘’http://A.B.C/shop0’’

3. Nom de la table : item
   Row key : id_item
   Column Family : info : name | price | url | description

Example :

Row	Column+Cell
item2	column=info : name=‘’item1’’
item2	column=info : price=45.43 (euros)
item2	column=info : url=‘’http://A.B.C/item2’’
item2	column=info : description=‘’It is very good’’

4. Nom de la table : friendship
   Row key : id_usr1/id_usr2
   Column Family : f : friendid

Example :

Row	Column+Cell
usr1/usr2	column=f : friendid=usr2
usr1/usr3	column=f : friendid=usr3

5. Nom de la table : visit
   Row key : id_item/id_usr/date/time
   Column Family : info : buy

example

Row	Column+Cell
item99/usr76/2002-9-14/1:34:22	column=info:buy=y
item99/usr74/2000-2-9/10:17:3	column=info:buy=n
…	…

Creation en 2nd ordre :

6. Nom de la table : ordered
   Row key : id_item/id_order
   Column Family1 : info : id_usr | id_shop | date_order | total_price | quantity

example :

Row	Column+Cell
item0/order1	column=info : id_usr=‘’usr1’’
item0/order1	column=info : id_shop=‘’shop4’’
item0/order1	column=info : year=‘’2002-04-13’’
item0/order1	column=info : total_price=153.54 (euros)
item0/order1	column=info : quantity=50

La table de HBase est trié par « RowKey » en l’ordre d’alphabétique, donc il provoque un problème :
Si nous utilisons les anciennes ids, nous obtenons la table comme suivante :

Row key	**…**	**…**	**…**
usr47	…	…	…
usr48	…	…	…
usr49	…	…	…

Row key	…	…	…
usr5	…	…	…
usr50	…	…	….
usr51	…	…	…
usr52	…	…	…

Comme vous pouvez voir dans la table précédente, nous pouvons voir que les « Row key » ne sont pas bien triés (car « usr5 » est après « usr49 »). Pour résoudre ce genre de problème, j'ai fait just un petit peu de changement que j'ai rempli « 0 » avant le « numéro » des utilisateurs pour obtenir la même longueur, et voici le nouveau « ids » des utilisateurs (supposons que nous avons 100 utilisateur en total):

Row key	…	…	…
usr000	…	…	…
usr001	…	…	…
usr002	…	…	…
…	…	…	…
usr048	…	…	…
usr049	…	…	…
usr050	…	…	…
usr051	…	…	…

Comme cela, ce problème a bien résolu.

Par default, toutes les données de HBase sont stockées dans RAM qui seront disparu après de redémarrer. Après d'avoir lu le documentation de HBase, j'ai vu qu'il faut changer le fichier « hbase-site.xml » qui se trouve dans « ./conf ». Voici le nouveau fichier « .xml » pour configurer HBase (les autres configurations ne changent pas) :

```
<?xml version="1.0"?>
<?xml-stylesheet type="text/xsl" href="configuration.xsl"?>
<configuration>
    <property>
        <name>hbase.rootdir</name>
        <value>file:///Users/tianqilei/Documents/hbase</value>
    </property>
    <property>
        <name>hbase.zookeeper.property.dataDir</name>
        <value>/Users/tianqilei/Documents/zookeeper</value>
    </property>
</configuration>
```

Car pour l'instant, mon stage ne concerne pas le système de base de données distribué (stocker les données dans les machines différentes), Je n'ai pas besoin d'installer « Hadoop ».

4.3 Le modèle de données dans Oracle Nosql

1. L'environnement
   - Langage de programmation : Java
   - Version de la base de données de Oracle Nosql : kv-2.1.97
   - Interface utilisateur : non
   - Pilotes : hbase-0.94.17

Oracle Nosql, comme HBase, est un système de base de données non-relationnelle. Oracle Nosql propose 2 type de api :

- Tables API
- Key/Value API

Dan notre programme, nous utilisons Key/Value API. Oracle Nosql organise les tuples en utilisant « key » . Toutes les tuples ont une ou plusieurs « major key », et, une ou plusieurs « minor key » qui ne sont pas obligatoires. Si l'on utilise « minor key », la combination de « major key » et « minor key » identifie une unique tuple dans la base de données. Toutes les tuples qui utilisent la même « major key » seront stocker dans la même partition dans la base de données, c'est-à-dire qu'on peut récupérer les données efficacement en utilisant « major key ». Et aussi, « minor key » augmente la performance si l'on utilise correctement.

L'introduction « Combination des clés » :
« Combination des clés » est une chaîne de caractères en Java. On peut supposer que « Combination des clés » est le chemin dans système de fichiers.

Par example :
/Usr/Jack/Documents/
On peut imaginer que Usr, Jack, Documents sont les répertoires dans un ordinateur et toutes les informations de « Documents » se trouve dans le chemin : /Usr/Jack/Documents/.

Et voici un example possible de « Combination des clés » :
Supposons que l'on utilise multiple « major key » (last name et first name) pour identifier une tuple. On peut avoir les « major key » comme suivantes :
/Smith/Bob
/Smith/Patricia
/Wong/Bill

De plus, si l'on veut stocker les informations comme « age », « birthdate », « phonenumber » …etc. Pour chaque utilisateur, on peut les stocker dans la base de données et les identifier en utilisant « minor key ».
P.S. Pour bien distinguer « major key » et « minor key », entre eux, on ajout un signe : /-/
Et voici un autre example quand on utilise « minor key » pour stocker les informations de chaque utilisateur.
/Smith/Bob/-/birthdate
/Smith/Bob/-/age
/Smith/Bob/-/phonenumber
…
Smith/Patricia/-/birthdate
Smith/Patricia/-/age
Smith/Patricia/-/phonenumber
…
Wong/Bill/-/birthdate
Wong/Bill/-/age
Wong/Bill/-/phonenumber

2. Avro Schema
Il faut faire attention ici que la façon précédente ne représente pas la façon le plus efficace. Oracle Nosql propose d'utiliser « Avro schema ».

Toutes les données dans la base de données d'Oracle Nosql sont organisées comme paire de « clé/valeur ». Les « valeur » est la donnée que nous voulons stocker, gérer et récupérer, donc nous devons la créer en utilisant « Avro schema » pour décrire le schema de donnée. L'utilisation de « Avro schema » permet de sérialiser les valeurs dans le système d'Oracle Nosql efficacement.

« Avro schema » est créé en utilisant le format de JSON(JavaScript Object Notation).

Pour décrire « Avro schema », on doit créer un record de JSON qui identifie le schema et voici un example :

```
{
    "type": "record",
    "namespace": "com.example",
    "name": "FullName",
    "fields": [
      { "name": "first", "type": "string" },
      { "name": "last", "type": "string" }
    ]
}
```

- « type » : Identifier le type de JSON (il est toujours « record » si l'on utilise Oracle Nosql)
- « namespace » : Identifier le « namespace » où se trouve les objets.
- « name » : le nom de schema.
- « fields » : la définition de schema qui défini quel « fields » est contenu dans la valeur.

Voici le modèle de base de données que j'ai utilisé dans mon programme :
1. Nom de la table : usr
Usr-usrid/name
Usr-usrid/age
Usr-usrid/sex
Usr-usrid/visits
Usr-usrid/addresses
Usr-usrid/friendship

Description de la table :
visits : décrire la liste des websites que cet utilisateur a visités
addresses : décrire la liste des toutes les adresses où cet utilisateur habite.
friendship : décrire la liste des amis de cet utilisateur.

2. Nom de la table : shop
Shop-shopid/name
Shop-shopid/url

3. Nom de la table : Country
Country/country_name : user list

Description de la table :
Cette table décrit tous les utilisateur de country_name
Cette table est créé just pour faciliter la requête que nous lancerons dans le futur (Requête 2 : l'âge de tous les utilisateurs de country : « Fr »).

4. Nom de la table : Sale
Sale - itemid : users list

Description de la table :
Cette table décrit la liste des utilisateurs qui ont achetés l'article dont l'id est itemid.

5. Nom de la table : item
Item - itemid/name
Item - itemid/price
Item - itemid/url
Item - itemid/description
Item - itemid/shopid
Item - itemid/orderquantity

Description de la table :
orderquantity : orderquantity est un « Avro schema » imbriqué qui contient deux attribut (order_id, quantity) et signifie que l'article dont l'id est « itemid » est contenu dans la commande avec l'id « order_id » et la quantité de cet article est « quantity ».

6. Nom de la table : Order
Order - orderid/date

Order - orderid/totalprice
Order - orderid/usrid
Order - orderid/itemline

Description de la table :
itemline : décrire la liste des articles qui ont relié à la command avec l'id « orderid ».

Pour avoir les structures précédentes, il faut définir les schémas. Donc j'ai créé les fichier comme suivants :

- addresses.avsc, friendship.avsc, item.avsc, itemlines.avsc, itemusers.avsc, ordered.avsc, orderquantity.avsc, shop.avsc, users.avsc, usr.avsc, visits.avsc

J'ai stocké tous ces fichiers dans « ./src/conn2database ». Après d'avoir défini tous les schemas qu'on utilisera dans le programme, on doit les ajouter dans la base de données d'Oracle Nosql. Supposons que nous avons un fichier « shop.avsc », si nous voulons l'ajouter dans la base de données d'Oracle Nosql, nous utilisons la commande comme suivante :

```
kv -> ddl add-schema -file ./src/conn2database/shop.avsc
```

Après d'ajouter les schemas dans la base de données, la dernière chose est de générer automatiquement les fichier de « .java », après cela, nous pouvons utiliser tous les objets que nous définissons dans les schemas. La commande est comme suivant :

```
ant -f generate-specific.xml
```

Après de lancer la commande en haut, il y a un package nommé « avro » est créé automatiquement qui contient tous les objets de Java, comme suivants :

- « Address.java », « Usr.java », « Visit.java », « Shop.java », « Friendship.java », « Item.java », « ItemLine.java », « ItemLines.java », « ItemUsrs.java », « Ordered.java », « OrderQuantity.java », « Shop.java », « Usr.java », « UsrBuy.java », « Usrs.java », « Visit.java », « Visits.java »

Voici un example d'écrire un objet « usr » dans la base de données d'Oracle Nosql :

```
public void writeUsr(String key, avro.Usr usr, avro.Visits visits,
        avro.Friendship friends, avro.Addresses addresses)
        throws IOException {
    List<String> majorPath = Arrays.asList("Usr");
    List<String> minorPath = Arrays.asList(key, "name");
    Value value = Value.createValue(usr.getName().getBytes());
    store.put(Key.createKey(majorPath, minorPath), value);

    minorPath = Arrays.asList(key, "age");
    value = Value.createValue(Bytes.toBytes(usr.getAge()));
    store.put(Key.createKey(majorPath, minorPath), value);

    minorPath = Arrays.asList(key, "sex");
    value = Value.createValue(usr.getSex().getBytes());
    store.put(Key.createKey(majorPath, minorPath), value);

    minorPath = Arrays.asList(key, "visits");
    store.put(Key.createKey(majorPath, minorPath), binding.toValue(visits));

    minorPath = Arrays.asList(key, "addresses");
    store.put(Key.createKey(majorPath, minorPath),
            binding.toValue(addresses));

    minorPath = Arrays.asList(key, "friendship");
    store.put(Key.createKey(majorPath, minorPath), binding.toValue(friends));

}
```

Explication du code :

1. Définir la « major key » et « minor key » en utilisant Arrays.asList(…).
2. Créer une valeur qui associe avec cette « major key » et « minor key » en utilisant Value.createValue(…).
3. Stocker la paire « majorKey-minorKey / Value » dans la base de données en utilisant store.put(…).

V. Performance de l'évaluation des requêtes

Pour évaluer la performance des requêtes dans les 3 systèmes de base de données, nous allons mesurer les temps utilisés pour les 7 requêtes que nous avons décidé.

Remarque : Il y a 7 requêtes à évaluer dans les 3 systèmes de base de données :

- Requête 1 : tout le profil de user t.q id= "usr1".
- Requête 2 : l'âge de tous les utilisateurs de country : "Fr".

- Requête 3 : le nombre de fois que chaque items a été acheté.
- Requête 4 : les 10 items les plus vendus.
- Requête 5 : les 10 items pas vendu du tout.
- Requête 6 : les 100 pages les plus visitées
- Requête 7 : les amis ayant achetés le même article.

5.1 L'évaluation des requêtes dans PostgreSQL (SGBD relationnelle)

La façon de lancer des requêtes dans PostgreSQL que j'ai utilisée est d'utiliser la méthode « execute(String sql) » proposé par JDBC.

Et voici les sql que j'ai utilisé pour lancer les 7 requêtes :

- QUERY1 : SELECT usr.id_usr, name, sex, age, street, city, country FROM addr, usr where usr.id_usr=addr.id_usr AND usr.id_usr='usr50';

- QUERY2 : SELECT age FROM addr, usr WHERE usr.id_usr=addr.id_usr AND country='reducer';

- QUERY3 : SELECT id_item, sum(quantity) AS nb FROM lineitem GROUP BY id_item;

- QUERY4 : SELECT id_item, count(*) AS nb FROM visit WHERE buy='y' GROUP BY id_item ORDER BY nb DESC LIMIT 10;

- QUERY5 : SELECT id_item, count(*) AS nb FROM visit WHERE buy='n' GROUP BY id_item ORDER BY nb DESC LIMIT 10;

- QUERY6 : SELECT url, count(*) AS nb FROM visit, item WHERE visit.id_item=item.id_item GROUP BY url ORDER BY nb DESC LIMIT 100;

- QUERY7 : SELECT id_usr1, id_usr2, v1.id_item FROM friend, usr u1, usr u2, visit v1, visit v2 WHERE id_usr1=u1.id_usr and id_usr2=u2.id_usr and u1.id_usr=v1.id_usr and u2.id_usr=v2.id_usr and v1.id_item=v2.id_item and v1.buy='y' and v2.buy='y';

Et j'ai stocké tous les sql dans un fichier « ./notes/query/query_postgresql.txt ».

5.2 L'évaluation des requêtes dans HBase et Oracle Nosql (SDBD non-relationnelle)

Pour toutes les requêtes qui concernent une seule table, dans mon programme, il y a rien difficile, c'est juste récupérer les données en utilisant la clé.
Ce qui est dur et plus important est de récupérer les données depuis plusieurs tables. Pour simplifier mon stage, cette fois-ci, nous n'intéressons que « HashJoin » car « HashJoin » est efficace dans plusieurs cas. Mais en général, il y a Nested Loops Join, Merge sort Join, Hash Join …etc, le système choisi la jointure plus efficace.

Chez inria, il y a un projet nommé « vip2p » qui a déjà fait tous les opérateurs. il me suffit de comprendre ce projet. Après les explications Ioana et Francesca, j'ai compris la partie d' « Opérateur Physique ».

Après de comprendre cela, j'ai fait quelques changements pour importer les codes de Java à mon propre programme. Tous les fichiers importés se trouve dans le package « ./src/vip2p_copy ».

Voici les explication sur « Opérateur Physique »

- NRSMD.java : cette code de Java représente « Nested Result Set MetaData » (Meta Data de données) qui décrit, pour ce tuple, il y a combien de colonnes, les noms des colonnes, le nombre de colonnes qui sont de type d'entier, le nombre de colonnes qui sont de type de « string », et, éventuellement, les NRSMDs des enfant imbriqués.

Voici un example :

Usr_id	name	age	friendship
usr1	John	19	usr2,usr5,usr56,usr99

Dans ce cas-là, il y a 4 colonnes, leurs noms sont : usr_id, name, age, friendship et il y a 2 colonnes qui sont de type de « string » et il y a une colonne qui est de

type d'entier. La colonne de « friendship » est un enfant imbriqué qui a son propre NRSMD.

- Ntuple.java : cette code de Java représente « Nested tuple » et défini chaque tuple qu'on va utiliser pour faire « Opérateur Physique » (par example : projection, selection, hashjoin) et leurs NRSMDs.

- NIterator.java : c'est une classe abstraite que chaque opérateur physique doit hériter. Principalement, elle contient 4 méthodes :
    - open() : initialiser iterator.
    - hasNext() : vérifier s'il y a un tuple dans iterator.
    - next() : récupérer un tuple.
    - close() : fermer iterator.

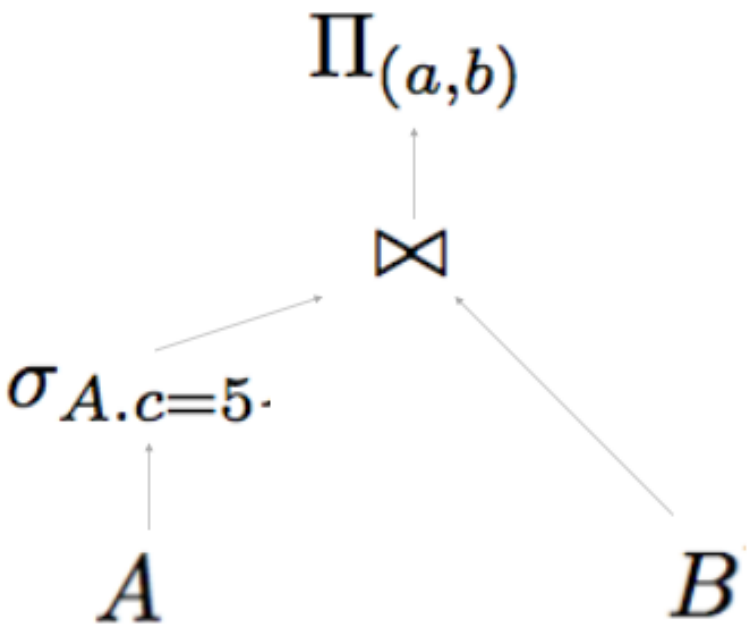
Dans mon programme, ArrayIterator.java, MemoryHashJoin.java, SimpleProjection.java, SimpleSelection.java héritent NIterator.java. Nous pouvons supposer que pour lancer une requête, nous avons un arbre dont les noeuds sont NIterator(soit ArrayIterator, soit MemoryHashJoin, soit SimpleProjection, soit SimpleSelection, …etc) qui exécute récursivement.

- ArrayIterator.java : qui contient une liste de NTuple et son NRSMD (des feuille dans un arbre)
- MemoryHashJoin.java : qui prend 2 NIterators et un Predicate comme arguments et renvoie un NIterator après « HashJoin » comme le résultat (Ici, le Predicate ne peut être que égalité). MemoryHashJoin.java peut être un noeud interne ou la racine d'arbre.
- SimpleProjection.java : qui prend un NIterator et une liste de numéro de colonne qu'on veut garder comme arguments, et qui renvoie un NIterator comme le résultat. SimpleProjection.java peut être un noeud interne ou la racine d'arbre.
- SimpleSelection.java : qui prend un NIterator et un Predicate comme arguments et qui renvoie un Niterator comme le résultat. SimpleSelection.java peut être un noeud interne ou la racine d'arbre.

For example, si l'on veut lancer une requête comme suivant :

$$\Pi_{(a,b)}(\sigma_{A.c=5}A \bowtie B)$$

Supposons que l'on a deux relation A(a,c,d) et B(a,b). On peut construire un arbre comme suivant :



Ce qui concerne le code en Java :

1. Construire 2 ArrayIterator objets qui contiennent tous les tuples de A et B, respectivement (les feuilles de l'arbre).
2. Construire un SimpleSelection objet avec 2 arguments : ArrayIterator de tous les tuples de A et un Predicate : A.c=5. Supposons que le résultat est noté comme C.
3. Choisir la relation plus petit entre C et B comme « build input » si l'on peut distinguer quelle relation est plus petit (mais dans certains cas, ce n'est pas possible), l'autre comme « probe input », et puis construire un object « MemoryHashjoin » (prédicat est toujours « égalité »)
4. À la fin, construire un objet « SimpleProjection » dont les argument sont le résultat de l'ancienne étape et les numéros de colonnes qu'on veut garder.

VI. Résultat obtenu

Je suis sur le point de calculer le temps utilisé pour les 7 requêtes dans les 3 systèmes, mais je n'ai pas encore un résultat car il y a toujours un problème sur mon code et je suis en train d'essayer de résoudre.

VII. Conclusions

Ce stage est très util et très pratique pour moi. il est ouvert une porte d'apprendre plusieurs types de la base de données et commet concevoir les modèles de base de données non-relationnelles. Et aussi, j'ai amélioré mon technique de programmer et le technique de travailler avec mes collègues.

De plus, il m'a donné une idée de concevoir un système avec plusieurs types de base de données comme une seule qui est très intéressante. La base de données NoSQL était très dur à comprendre pour moi au début parce que je concevais la base de données non-relationnelle avec la moyen que j'avais utilisé pour concevoir la base de données relationnelle, mais je trouvais que le modèle était très dur à utiliser. Mais après d'avoir lu beaucoup de documents concernant NoSQL, je commence à comprendre comment le concevoir correctement et raisonnablement.

Et pendant mon stage, j'ai aussi appris les performances des 3 systèmes différents et, cela, je pense qu'il est très util dans mon futur projet.

Merci Mme. Ioana et Mme Francesca, c'est un stage super!