MapReduce Fest - Desafío 6
Patrones de Join

Lorena Etcheverry
Instituto de Computación,
Facultad de Ingeniería,
lorenae@fing.edu.uy

18 de septiembre de 2013

Índice

1. Patrones de join

De acuerdo con [1] el join es una operación compleja de implementar usado mapReduce. En esta sección se describe brevemente cada uno de los patrones presentados en [1], los cuales varían en la complejidad de su implementación, en el uso de recursos (por ejemplo: ancho de banda) y en que tipos de join soportan (e.g: inner join, outer join, antijoin y producto cartesiano).

1.1. Reduce side join

Este patrón, en comparación con los otros, es el más sencillo de implementar y soporta todos los tipos de join, pero es el más costoso en recursos (en particular en tiempo de cómputo). Esto se debe a que los datos no se reducen en volumen en la fase de *map* y deben ser transferidos en su totalidad a la fase

de *reduce*. Otra característica importante de esta implementación es que permite combinar cualquier cantidad de datasets a la vez y no tiene restricciones en cuanto al tamaño de cada dataset.

Para aplicar este patrón, durante la fase de *map*, se comienza por asignar un identificador a cada uno de los datasets a procesar. Luego, para cada registro en cada conjunto de datos, se genera una pareja clave,valor donde la clave es el valor del atributo que se va a utilizar para realiza el join y el valor es el conjunto de valores para los demás atributos del registro más el identificador del dataset del cual proviene el registro. En la fase de *reduce* cada *reducer* utiliza estructuras auxiliares (por ejemplo listas) para almacenar los registros provenientes de cada dataset y luego recorre estas estructuras para implementar el join. El algoritmo a aplicar para combinar estas estructuras depende del tipo de join a implementar. Por ejemplo, en el caso del *inner join* es necesario anidar iteraciones donde para cada elemento de un dataset se buscan todos sus correspondientes en los demás datasets. La Figura 1 presenta gráficamente este patrón [1].

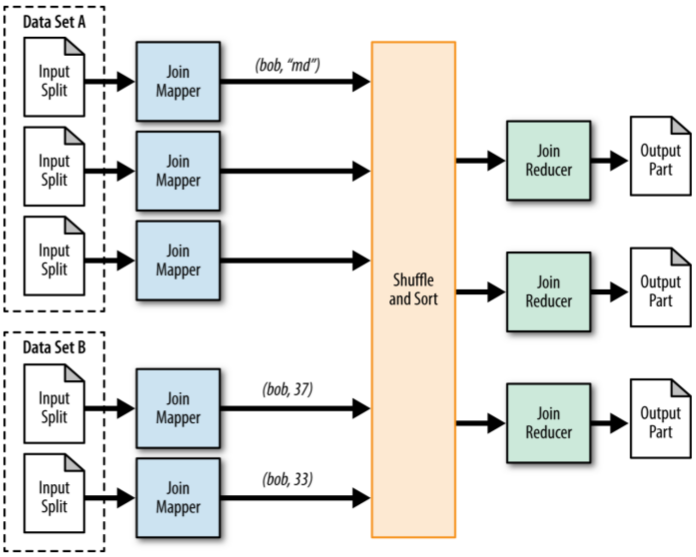

Figura 1: Patrón de diseño *reduce side join*[1]

Se podría optimizar un poco esta implementación si se ordenan los datasets por tamaño y luego se colocan los más grandes en los ciclos exteriores de la iteración. El cálculo del tamaño no implica un costo extra ya que se puede mantener en una variable durante el proceso de construcción de las estructuras auxiliares.

1.2. Reduce side join con bloom filter

Cuando la operación de join aparece ligada a una operación de selección es posible implementar ambas de una sola vez utilizando un bloom filter para filtrar registros en la fase de map y luego aplicando el patrón presentado en la Sección 1.1.

1.3. Replicated join

Este patrón sólo se puede aplicar cuando todos los *datasets* menos uno pueden ser mantenidos en memoria. Su aplicación consiste en cargar en el cache distribuído, durante la fase de *setup* de cada *mapper*, a todos los *datasets* pequeños. Luego, para cada registro del *dataset* restante (el más grande en tamaño), se realiza el join y se transmite el resultado a la fase de *reduce* en la cual no se hace nada. Dado que cada mapper no conoce por completo al dataset grande, este patrón sólo soporta *inner joins* o *left outer joins*. La Figura 2 representa gráficamente este patrón [1].

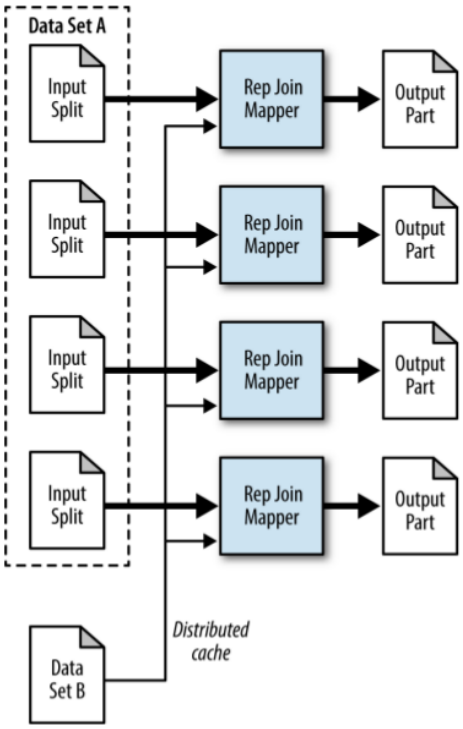

Figura 2: Patrón de diseño *replicated join*[1]

1.4. Composite join

Este patrón permite implementar inner join o full outer join levantando la restricción respecto al tamaño de los datasets a combinar y realizando todo el cómputo en la fase de *map*. Para lograr esto es necesario que los datasets cumplan con una serie de restricciones:

- todos los datasets tienen la misma cantidad de particiones
- cada partición está ordenada por el/los atributos que se van a usar para combinar y cada valor de estos atributos sólo está presente en una partición. Por ejemplo, la partición X de los datasets A y B contienen los mismos valores para estos atributos y estos valores sólo están en la partición X.

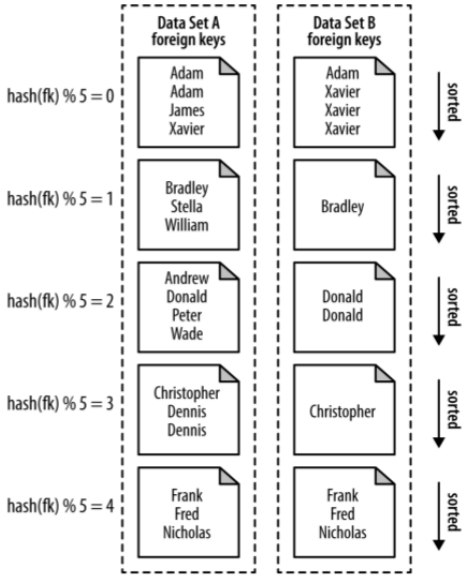

Figura 3: Patrón de diseño *composite join*[1]

- todos los datasets son lo suficientemente grandes para permitir las restricciones anteriores
- dado que los datasets deben ser preparados éstos no deben cambiar frecuentemente

Dado el costo que puede tener preparar los datos para que satisfagan estas restricciones se recomienda la aplicación de este patrón sólo si los datos cumplen las mismas naturalmente, es decir si no es necesario preparar los mismos. La Figura 3 presenta gráficamente este patrón [1].

1.5. Producto cartesiano

La paralelización del producto cartesiano no es trivial, dado que este operador combina todos los registros entre sí. La implementación de este operador usando mapReduce consiste en particionar los datasets de entrada y salida para luego generar parejas de particiones tales que se cubran todas las posibilidad de combinación. Durante la fase de *map* cada *mapper* toma parejas de registros (uno de cada una de las particiones que recibe como entrada) y así genera la salida. La Figura 4 presenta gráficamente este patrón donde el *datasets* A fue particionado en 2 (A-1 y A-2) mientras que el *dataset* B fue particionado en 3 (B-1, B-2 y B-3).

2. Implementación de Replicated Join Pattern

En [1] se implementa este patrón para agregar datos de los usuarios a los comentarios de StackOverflow. Las Figuras 5 y 6 presentan respectivamente

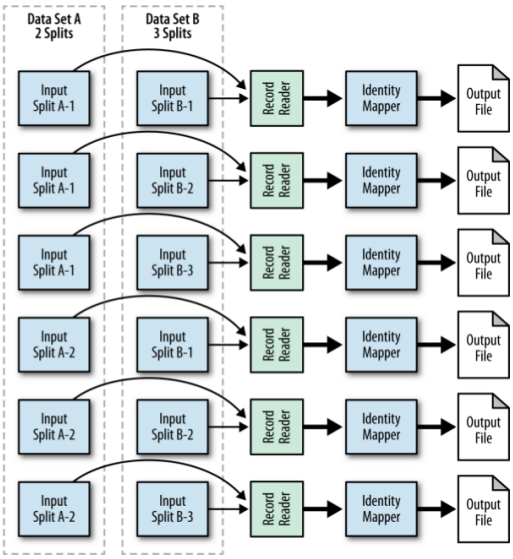

Figura 4: Patrón de diseño para el producto cartesiano[1]

los métodos *setup* y *map* de la implementación de este patrón. Tal como se presentó en la Sección 1.3 para implementar este patrón es necesario que todos los conjuntos de datos menos uno sea compartidos entre los mappers via el caché distribuido. En este caso la información de los usuarios será compartida entre todos los *mappers*, mientras que el conjunto de comentarios se particionará y distribuirá a los diferentes *mappers*.

Los datos de los usuarios se almacenan en una estructura auxiliar (HashMap declarado en la línea 2 de la Figura 5) en la cual se indexa la información de los usuarios a partir del identificador de usuario (*UserId*). Esta estrategia puede repercutir en temas de performance. En las pruebas realizadas fue necesario incrementar la cantidad de memoria de la VM de Java para que pudiese alojar los datos del usuario.

Este patrón realiza todo el procesamiento en la fase de map. Se puede apreciar que la combinación entre los datos de los comentarios, los cuales son leídos y procesados a partir de la línea 4 de la Figura 6), y los datos de los usuarios se realiza en esta fase.

3. Join usando Hive

En esta sección se presenta brevemente Apache Hive y se describen los pasos seguidos para implementar el join entre usuarios y comentarios usando este enfoque.

3.1. Apache Hive

En los últimos años han surgido varios proyectos dentro del ecosistema de Hadoop con el objetivo de hacer más amigable la utilización de Hadoop. En particular Apache Hive[3] provee un modelo de datos y un dialecto SQL, lla-

```
public static class ReplicatedJoinMapper extends Mapper<Object, Text, Text, Text> {
  private HashMap<String, String> userIdToInfo = new HashMap<String, String>();
  private Text outvalue = new Text();
  private String joinType = null;

@Override
public void setup(Context context) throws IOException, InterruptedException {
  try {
      Path[] files = DistributedCache.getLocalCacheFiles(context.getConfiguration());
          if (files == null || files.length == 0) {
            throw new RuntimeException("User information is not set in DistributedCache");
          }
          // Read all files in the DistributedCache
         for (Path p : files) {
           BufferedReader rdr = new BufferedReader(new InputStreamReader(
              new GZIPInputStream(new FileInputStream(new File(p.toString())))));
           String line;
           // For each record in the user file
           while ((line = rdr.readLine()) != null) {
             // Get the user ID for this record
                Map<String, String> parsed = MRDPUtils.transformXmlToMap(line);
                String userId = parsed.get("Id");
                if (userId != null) {
                  // Map the user ID to the record
                  userIdToInfo.put(userId, line);
                 }
               }
               rdr.close();
        }
  } catch (IOException e) {
    throw new RuntimeException(e);
  }
  // Get the join type
  joinType = context.getConfiguration().get("join.type");
}
```

Figura 5: Método setup

```
@Override
public void map(Object key, Text value, Context context) throws IOException,
     InterruptedException {
// Parse the input string into a nice map
  Map<String, String> parsed = MRDPUtils.transformXmlToMap(value.toString());
  String userId = parsed.get("UserId");
   if (userId == null) {
     return;
   }
  String userInformation = userIdToInfo.get(userId);
  // If the user information is not null, then output
  if (userInformation != null) {
    outvalue.set(userInformation);
        context.write(value, outvalue);
  } else if (joinType.equalsIgnoreCase("leftouter")) {
  // If we are doing a left outer join, output the record with an empty value
  context.write(value, new Text(""));
  }
}
```

Figura 6: Método map

mado Hive Query Language (HiveQL), los cuales permiten realizar consultas sobre datos gestionados por Hadoop, brindando de esta forma una capa de abstracción. Las consultas que se realizan en HiveQL se traducen automáticamente y en forma transparente para el usuario, en *jobs* Hadoop. La Figura 7 presenta los principales módulos de Hive y Hadoop [8] y su interacción con herramientas como Karmasphere[6], Hue[4] and Qubole[9] que implementan interfaces de usuario sobre Hive para facilitar aún más la tarea.

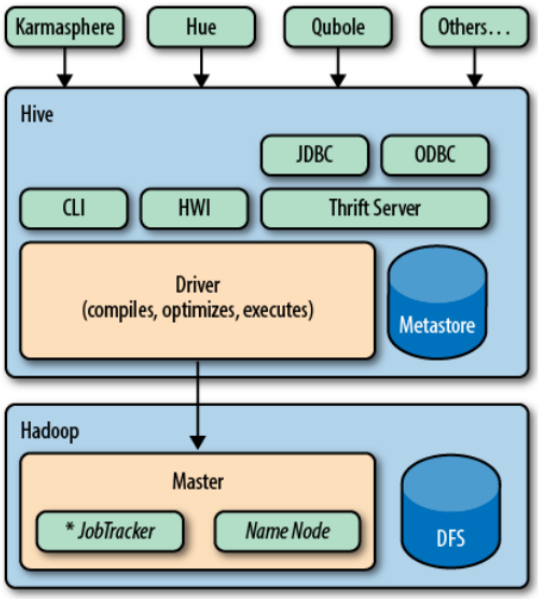

Figura 7: Arquitectura de Hive[8]

HiveQL, además de ser un lenguaje de consulta, permite definir estructuras y manipular datos. La parte de definición del lenguaje permite crear, modificar y borrar bases de datos, tablas, vistas, funciones e índices, mientras que la parte de manipulación permite realizar operaciones de inserción, borrado y modificación a nivel de tabla. Estas operaciones no pueden realizarse a nivel de tupla como es habitual en los RDBMS.

Hive provee *tipos primitivos de datos* (por ejemplo `BOOLEAN`, `INT`, etc.) y *colecciones* como `structs`, `maps`, and `arrays`. Las colecciones permiten representar relaciones N:N en la tabla donde se describe el concepto. Esta funcionalidad, raramente implementada en las bases de datos relacionales, permite desnormalizar los datos y evitar las relaciones de clave foránea entre tablas. Por otro lado, promueve la duplicación de datos y no permite representar en forma estructural restricciones de integridad referencial.

A diferencia de los RDBMS Hive no controla cómo se almacenan los datos y soporta varios formatos de archivos (cómo se codifican los registros en el archivo) y varios formatos de registro (cómo se codifican los bytes de un registro). El esquema de cada tabla se aplica a los datos a medida que estos son recuperados del almacenamiento, implementando lo que se conoce como *schema on read*.

3.2. Aplicación de Hive al caso de estudio

Para aplicar Hive al caso de estudio se comenzó por definir una tabla para almacenar los comentarios y otra para los usuarios. Dado que en este caso no es necesario que Hive almacene datos en XML no es necesario implementar un serializador-deserializador (o SerDe) y basta con especificar, usando expresiones XPath, cómo se debe poblar cada uno de los atributos de cada tabla a partir del XML. La Figura 8a presenta a modo de ejemplo una entrada del archivo XML de comentarios, mientras que en la Figura 8b se presenta: (i) la definición de la tabla *comm*, la cual contiene todo el XML de comentarios, (ii) la definición de la vista *commentsview*, la cual contiene una fila por cada elemento del XML y se indica mediante expresiones XPath como se debe poblar cada uno de los atributos de cada fila, y (iii) la tabla *commentstable* que materializa la vista anteriormente creada. Este último paso se realiza para precomputar los registros y no realizar el parsing cada vez. Se realizó el mismo procedimiento con el archivo de usuarios y se generaron las estructuras que se presentan en la Figura 8c. Una vez definidas las tablas se realizaron las operaciones que se presentan en la Figura 9.

4. Pruebas realizadas

Para realizar pruebas se utilizaron los archivos *users.xml* (98.8 MB) y *comments.xml* (959.6 MB) correspondientes al dump de Noviembre de 2010 de StackOverflow, el cual se obtuvo desde el sitio ClearBits[1].

El tiempo de ejecución para la tarea Hadoop presentada en 2 y para los archivos indicados estuvo en el entorno de los 7 minutos, mientras que el tiempo de ejecución de las operaciones de join sobre Hive estuvo en el entorno de los 50 minutos. Se ensayaron las dos operaciones de join para explorar si esto cambiaba el tiempo de respuesta, pero la diferencia en este caso es despreciable. Además es necesario tener en cuenta el tiempo que insume la creación de las tablas en Hive. La Tabla 1 presenta los valores medidos ejecutando Hive dentro de la máquina virtual provista por Hortonworks [2], donde se puede apreciar que la carga de la tabla de comentarios insumió más de 3 horas. Cabe señalar que la solución implementada sobre Hadoop fue ejecutada en un entorno con 4GB de memoria RAM mientras que las pruebas de Hive se realizaron en la VM de Hortonworks configurada con 2 GB de RAM.

creación tabla comments	2.4 segundos
creación vista commentsview	2.9 segundos
creación tabla commentstable	3.5 horas
creación tabla users	1.5 segundos
creación vista usersview	1.8 segundos
creación tabla userstable	41 minutos

Tabla 1: Tiempos de ejecución de la creación de estrucuturas en Hive

Pese a que los entornos de ejecución son diferentes, las diferencias obte-

[1] `http://www.clearbits.net/creators/146-stack-exchange-data-dump/contents`
[2] `http://hortonworks.com/products/hortonworks-sandbox/`

```
<row Id="1" PostId="35314" Score="8"
Text="not sure why this is getting downvoted -- it is correct!
Double check it in your compiler if you don't believe him!"
CreationDate="2008-09-06T08:07:10.730" UserId="1" />
```

(a) Ejemplo de registro del archivo de comentarios.

```
CREATE EXTERNAL TABLE comm (commentsxml string)
STORED AS TEXTFILE
LOCATION '/user/hive/source/comment';

CREATE VIEW commentsview(rowid, postid, score, text, creationdate, userid)
AS SELECT
xpath_int(commentsxml, 'row/@Id'),
xpath_int(commentsxml, 'row/@PostId'),
xpath_int(commentsxml, 'row/@Score'),
xpath_string(commentsxml, 'row/@Text'),
xpath_string(commentsxml, 'row/@CreationDate'),
xpath_int(commentsxml, 'row/@UserId')
FROM comm;

CREATE TABLE commentstable AS SELECT * FROM commentsview;
```

(b) Definición de tablas y vista para comentarios.

```
CREATE EXTERNAL TABLE users (usersxml string)
STORED AS TEXTFILE
LOCATION '/user/hive/source/users';

CREATE VIEW usersview(userid, reputation, creationdate, displayname,
emailhash, lastaccessdate,website,location, age, about,views, upvotes,downvotes)
AS SELECT
xpath_int(usersxml, 'row/@Id'),
xpath_int(usersxml, 'row/@Reputation'),
xpath_string(usersxml, 'row/@CreationDate'),
xpath_string(usersxml, 'row/@DisplayName'),
xpath_string(usersxml, 'row/@EmailHash'),
xpath_string(usersxml, 'row/@LastAccessDate'),
xpath_string(usersxml, 'row/@WebsiteUrl'),
xpath_string(usersxml, 'row/@Location'),
xpath_int(usersxml, 'row/@Age'),
xpath_string(usersxml, 'row/@AboutMe'),
xpath_int(usersxml, 'row/@Views'),
xpath_int(usersxml, 'row/@UpVotes'),
xpath_int(usersxml, 'row/@DownVotes')
FROM users;

CREATE TABLE userstable AS SELECT * FROM usersview;
```

(c) Definición de tablas y vista para usuarios.

Figura 8: Representación de los comentarios y usuarios en Hive

```
SELECT *
FROM commentstable c JOIN userstable u
ON (c.userid=u.userid)
```

(a) Join versión 1.

```
SELECT *
FROM userstable u JOIN commentstable c
ON (u.userid=c.userid)
```

(b) Join versión 2.

Figura 9: Operaciones de join

nidas entre el tiempo de ejecución de la solución sobre Hadoop y la solución sobre Hive son llamativas. En principio, parecería que Hive hace más fácil el procesamiento de datos a costo de tiempos de ejecución mayores en varios órdenes de magnitud. Es necesario realizar más y mejores pruebas para poder ser concluyente al respecto.

Referencias

[1] Donald Miner and Adam Shook. *MapReduce Design Patterns Building Effective Algorithms and Analytics for Hadoop and Other Systems*. O'Reilly Media, Inc., 2012.

[2] Apache Hadoop Project. `http://hadoop.apache.org/`

[3] Apache Hive Project. `http://hive.apache.org/`

[4] Cloudera Hue. `http://cloudera.github.io/hue/`

[5] Dean, Jeffrey and Ghemawat, Sanjay. *MapReduce: Simplified data processing on large clusters*. In Proceedings of the 6th Symposium on Operating System Design and Implementation (OSDI 2004), pages 137–150, San Francisco, California, 2004.

[6] Karmasphere. `http://www.karmasphere.com/`

[7] Lin, Jimmy and Chris Dyer. *Data-intensive text processing with MapReduce.* Synthesis Lectures on Human Language Technologies 3.1: 1-177, 2010

[8] Programming Hive. Capriolo, E., Wampler, D., and Rutherglen, J. 1st edition, O'Reilly Media, Inc, 2012.

[9] Qubole.`http://www.qubole.com`