

Uso de Hive Avanzado

- Partición y agrupación de datos con Hive
- Salida de datos con Hive
- Automatización de Hive
- Tipos de datos complejos en Hive
- Funciones de enmascaramiento y *hashing* de datos en Hive
- Procesamiento y análisis de textos con Hive

Apache Hive



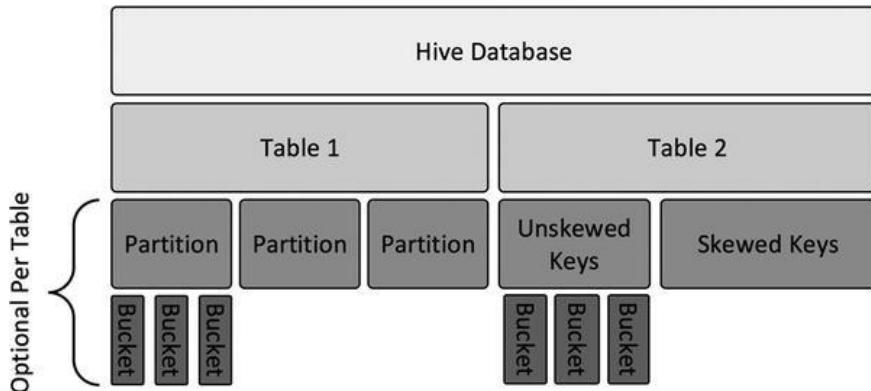
Optimización y gestión de consultas en Hive

Recordemos que, a diferencia de una plataforma de base de datos relacional tradicional, Hive no dispone de algunas de las capacidades integradas que optimizan las consultas y el acceso a los datos, como los índices. Por lo tanto, las optimizaciones de diseño que se realicen por adelantado pueden ser muy valiosas para optimizar el rendimiento. Uno de los factores en los que se puede influir es la cantidad de datos que lee una consulta concreta. Hive ofrece dos enfoques clave utilizados para limitar o restringir la cantidad de datos que una consulta necesita leer, cada uno de los cuales lo hace de una manera diferente para un propósito diferente.

Vamos a discutir ambos enfoques ahora.

Particionamiento y *bucketing*

Las tablas Hive pueden dividirse en más trozos lógicos para facilitar la gestión y mejorar el rendimiento. Existen varias formas de dividir los datos en Hive



Las tablas particionadas en Hive tienen una o más claves de partición en función de las cuales los datos se dividen en trozos lógicos y se almacenan en directorios separados. Cada clave de partición añade un nivel de estructura de directorios al almacenamiento de la tabla. Veamos un ejemplo de una tabla de transacciones de clientes con algunas claves de partición.

El *bucketing* en Hive es otra manera de dividir los datos en segmentos más pequeños. El particionamiento puede ayudar a organizar y acceder a los datos de forma eficiente. Sin embargo, un particionado eficiente requiere el uso de una clave de partición, que no dé lugar a un gran número de particiones muy pequeñas. Así que, si tiene muchos valores diferentes para

la clave de partición y no muchas filas para cada valor de clave de partición, el particionado puede no ser la mejor opción.

El *bucketing* es más adecuado para estos casos. El *bucketing* le permite definir el número máximo de *buckets* para la columna elegida para ello de la tabla. Una partición es un directorio en Hive, donde el valor de la clave de partición se almacena en el nombre real del directorio de partición y la clave de partición es una columna virtual de la tabla. Sin embargo, en el caso de *bucketing*, cada bucket es un archivo que contiene los datos reales que se agrupan sobre la base de un algoritmo *hash*. El *bucketing* no añade una columna virtual a la tabla.

Particionado en Hive

El particionado se utiliza para dividir los datos en subdirectorios basados en una o más condiciones que normalmente se utilizarían en las cláusulas WHERE de la tabla. Sin embargo, puede ejecutar consultas que abarquen varias particiones; por ejemplo, consultas que no incluyan una cláusula WHERE en la columna particionada. El listado 26 muestra una sentencia CREATE TABLE con una columna particionada.

```
hive> CREATE TABLE weblogs (
  > ip STRING,
  > time_local STRING,
  > method STRING,
  > uri STRING,
  > protocol STRING,
  > status STRING,
  > bytes_sent STRING,
  > referer STRING,
  > useragent STRING)
  > PARTITIONED BY (date STRING);
```

Listado 1 CREATE TABLE con partición

La figura 1 muestra cómo funcionaría el particionamiento con la tabla particionada de weblogs creada en el Listado 26.

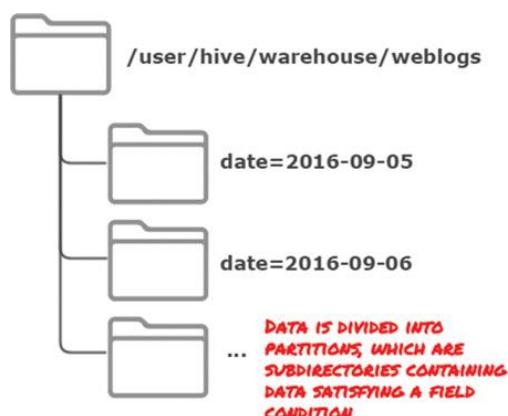


Figura 1 Partición en Hive.

Observa que el subdirectorio creado tiene el nombre y el valor de la columna incluidos.

La partición puede realizarse de una de las dos maneras siguientes:

- Particionado estático
- Particionado dinámica

Particionado estático

En el particionado estático, es necesario insertar manualmente los datos en las diferentes particiones de una tabla. Utilicemos una tabla particionada por el campo **pid** que representa el código de una región. Para cada región, es necesario insertar manualmente los datos de la fuente de datos a una partición de estado en la tabla particionada. Por lo tanto, para 29 regiones, es necesario escribir el número equivalente de consultas Hive para insertar datos en cada partición. Vamos a entender esto usando el siguiente ejemplo.

```
CREATE TABLE sales_part(id int, fname string, state string, zip string, ip
string) partitioned by (pid string) row format delimited fields terminated by
'\t';

Insert into sales_part partition (pid= 'PI_03') select id,fname,state,zip,ip
from sales where pid= 'PI_03';

Insert into sales_part partition (pid= 'PI_02') select id,fname,state,zip,ip
from sales where pid= 'PI_02';

Insert into sales_part partition (pid= 'PI_05') select id,fname,state,zip,ip
from sales where pid= 'PI_05';
```

Particionado dinámico

Veamos un escenario en el que tenemos 50 ID de producto y necesitamos particionar los datos para todos los ID de producto únicos disponibles en el conjunto de datos. Si optamos por el particionamiento estático, tendremos que ejecutar el comando INSERT INTO para los 50 ID de producto distintos. En este caso es mejor optar por el particionado dinámico. En este tipo, las particiones se crearán para todos los valores únicos del conjunto de datos para una columna de partición determinada.

Por defecto, Hive no permite el particionado dinámico. Necesitamos habilitarlo configurando las siguientes propiedades en el CLI o en **hive-site.xml**:

```
hive> set hive.exec.dynamic.partition = true;
hive> set hive.exec.dynamic.partition.mode = nonstrict;
```

Una vez habilitado el particionado dinámico, podemos crear particiones para todos los valores únicos de cualquier columna, por ejemplo, sobre el campo *state* de la tabla, de la siguiente manera:

```
hive> create table sales_part_state (id int, fname string, zip string, ip
string, pid string) partitioned by (state string) row format delimited fields
terminated by '\t';

hive> Insert into sales_part_state partition(state) select
id,fname,zip,ip,pid,state from sales;
```

Observar que el campo o campos sobre el que se partitiona la tabla quedarán siempre como los últimos de la misma.

Las particiones pueden añadirse posteriormente utilizando la sentencia ALTER TABLE ADD PARTITION que se muestra en el Listado 2.

```
hive> ALTER TABLE weblogs  
> ADD PARTITION (date='2016-09-05');
```

Listado 2 Sentencia ADD PARTITION

Por el contrario, las particiones pueden eliminarse utilizando la cláusula DROP PARTITION de una sentencia ALTER TABLE como se muestra en el Listado 3.

```
hive> ALTER TABLE weblogs  
> DROP PARTITION (date='2016-09-05');
```

Listado 3 Sentencia DROP PARTITION

Se puede utilizar el comando SHOW PARTITIONS para ver las particiones conocidas de cualquier tabla particionada en Hive como se muestra en el Listado 4.

```
hive> SHOW PARTITIONS weblogs;  
date=2016-09-05  
date=2016-09-06
```

Listado 4 Declaración SHOW PARTITIONS

Si se crean subdirectorios en HDFS sin realizar una operación ADD PARTITION, los datos de estos subdirectorios no se devolverán para ninguna sentencia SELECT en la tabla, incluyendo una sentencia SELECT con una cláusula WHERE que haga referencia a la partición creada manualmente. Una solución alternativa cuando se crean o eliminan manualmente particiones añadiendo o eliminando subdirectorios del objeto de tabla en HDFS es utilizar el comando MSCK REPAIR TABLE, que encontrará las particiones recién creadas en HDFS y actualizará el metastore en consecuencia. Un ejemplo del comando MSCK REPAIR TABLE se muestra en el Listado 5.

```
hive> MSCK REPAIR TABLE weblogs;
```

Listado 5 Comando MSCK REPAIR TABLE

Se deben considerar cuidadosamente las columnas utilizadas para el particionamiento en función de la distribución y la unicidad de los valores en la tabla. Los campos que son generalmente únicos (como las columnas de ID o las marcas de tiempo que incluyen milisegundos) no deben utilizarse como columnas de partición, ya que crearán un exceso de particiones. En su lugar, se utilizan columnas que aparezcan normalmente en una cláusula WHERE y que tengan un rango de valores conocido, como un mes o un año.

Agrupación de elementos en Hive (*bucketing*)

El *bucketing* se utiliza para agrupar elementos basados en una clave *hash* y es un método alternativo para subdividir los datos en una tabla Hive. La agrupación en lotes implica el cálculo de un código hash para los valores insertados en las columnas agrupadas. Los rangos de códigos hash se asignan entonces a uno o a un número predefinido de *buckets*. En el Listado 6 se muestra un ejemplo de sentencia CREATE TABLE con una columna agrupada.

```
hive> CREATE TABLE customers_buckets
    > (cust_id INT,
    > fname STRING,
    > lname STRING,
    > email STRING
    > )
    > CLUSTERED BY (cust_id) INTO 3 BUCKETS;
```

Listado 6 CREATE TABLE con bucketing

Como se muestra en la Figura 2, los registros con el mismo **cust_id** se almacenarán siempre en el mismo *bucket* (segmento de ficheros). Las columnas de *bucket* se definen mediante palabras clave CLUSTERED BY. Es bastante diferente de las columnas de partición, ya que las columnas de partición se refieren al directorio, mientras que las columnas de bucket tienen que ser columnas de datos reales de la tabla. Mediante el uso de *buckets*, una consulta HQL se puede realizar de forma más fácil y eficiente (muestreos, uniones y agrupaciones sobre campo del *bucket*,...).

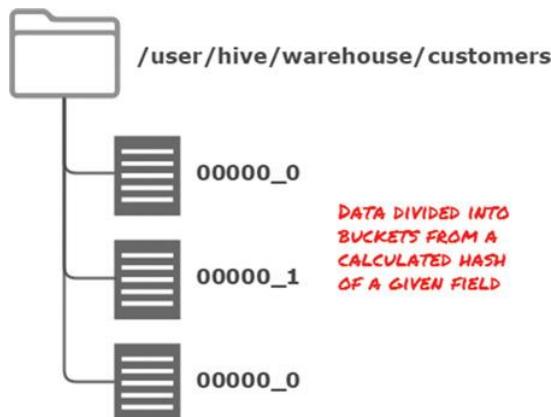


Figura 2. La creación de cubos en Hive.

La creación de *buckets* depende estrechamente del proceso de carga de datos. Para cargar correctamente los datos en una tabla con *buckets*, tenemos que establecer el número máximo de reductores en el mismo número de *buckets* especificado en la creación de la tabla (por ejemplo, 3), o activar *enforce bucketing* en HIVE (recomendado), de la siguiente manera:

```
> set map.reduce.tasks = 3;
No rows affected (0.026 seconds)

> set hive.enforce.bucketing = true; -- This is recommended
No rows affected (0.002 seconds)
```

Para cargar los datos de una tabla de *bucket*, no podemos utilizar la sentencia LOAD DATA, porque no verifica los datos contra los metadatos. En su lugar, se debe utilizar INSERT para poblar la tabla con buckets:

```
INSERT OVERWRITE TABLE customers_buckets SELECT * FROM customers
```

Ahora, cuando el usuario consulta la tabla **customers_buckets** por un ID o un rango de IDs, Hive sabe en qué *bucket* buscar un ID concreto. El motor de consulta sólo escaneará ese *bucket* y devolverá el conjunto de resultados.

Nota: Aplicación de la función de buckets durante las operaciones INSERT

La agrupación en *buckets* no se aplica por defecto cuando se insertan datos. Para habilitar la agrupación automática de datos, debe establecer la propiedad `hive.enforce.bucketing` a `true` en el archivo de configuración `hive-site.xml` o mediante el comando `SET` en una sesión de Hive. También debe considerar la posibilidad de cambiar el número de reductores utilizados en las operaciones de procesamiento contra la tabla para que coincida con el número de cubos de la tabla utilizando la propiedad `mapred.reduce.tasks`.

Salida de datos con Hive

Hasta ahora hemos visto el uso de la sentencia `SELECT` para ejecutar consultas HiveQL. La sentencia `SELECT` en Hive envía los resultados de la consulta al cliente Hive, de forma muy similar a como lo haría una herramienta de consulta SQL. A menudo, como parte de un flujo de trabajo ETL, por ejemplo, es necesario persistir la salida de una consulta en un sistema de archivos local o distribuido, concretamente en HDFS.

Hive admite varios métodos para conseguirlo, entre los que se incluyen los siguientes

- `INSERT OVERWRITE`, que enviará los resultados de la consulta a otra tabla Hive (un directorio en HDFS), sobrescribiendo el contenido existente
 - `INSERT INTO TABLE`, que enviará los resultados de la consulta a otra tabla Hive, añadiendo la salida al contenido de la tabla existente (directorio)
 - `INSERT OVERWRITE DIRECTORY`, que guardará los resultados en un directorio de HDFS que puede o no estar asignado a una tabla Hive
 - `INSERT OVERWRITE LOCAL DIRECTORY`, que guardará los resultados en un directorio local, no en HDFS
- Sintaxis del comando:

En el Listado se muestran ejemplos de uso de los distintos comandos `INSERT` para obtener resultados de la consulta.

- Insertar un único registro en la tabla.

```
hive> INSERT INTO employee VALUES  
(13,'Mari',51,'M'),  
(14,'Pat',34,'F');
```

También se puede utilizar el resultado de la consulta select en una tabla. Aquí se ha creado una nueva tabla Hive e insertado los datos del resultado de la consulta select.

```
CREATE TABLE employee_tmp LIKE employee;  
INSERT INTO employee_tmp SELECT * FROM employee;
```

- Con Tabla Particionada

Para explicar INSERT INTO con una Tabla particionada, supongamos que tenemos una tabla ZIPCODES con STATE como clave de partición.

Es necesario especificar la cláusula opcional PARTITION para insertar en una partición específica.

Este ejemplo inserta los registros en la partición FL de la tabla particionada Hive.

```
INSERT INTO zipcodes PARTITION(state='FL') VALUES  
(891,'US','TAMPA',33605);
```

- INSERT OVERWRITE borra todos los datos de la tabla Hive e inserta la fila especificada con los VALORES.

```
INSERT OVERWRITE TABLE EMPLOYEE VALUES (11,'Ram',51,'M');
```

- INSERT OVERWRITE con cláusula PARTITION elimina los registros de la partición especificada e inserta los nuevos registros en la partición sin tocar otras particiones.

```
INSERT OVERWRITE TABLE zipcodes PARTITION(state='FL') VALUES  
(896,'US','TAMPA',33607);
```

- La sentencia INSERT OVERWRITE también se utiliza para exportar la tabla Hive al directorio HDFS o LOCAL, para ello es necesario utilizar la cláusula DIRECTORY.

```
INSERT OVERWRITE DIRECTORY '/user/data/output/export'  
ROW FORMAT DELIMITED FIELDS TERMINATED BY ','  
SELECT * FROM empleado;
```

También puede exportar directamente la tabla al directorio LOCAL.

```
INSERT OVERWRITE DIRECTORY LOCAL '/tmp/export'  
ROW FORMAT DELIMITED FIELDS TERMINATED BY ','  
SELECT * FROM EMPLOYEE;
```

Como se ha comentado, el formato de salida por defecto de Hive utiliza caracteres de control no imprimibles para los delimitadores de campo. Esto a menudo no es deseable para la salida que se utilizará fuera de Hive. Puede controlar este comportamiento creando una tabla Hive de salida con el delimitador que prefiera, por ejemplo, delimitado por comas o por tabulaciones, y luego utilizar INSERT OVERWRITE o INSERT INTO TABLE para almacenar la salida de su consulta en el formato deseado.

Automatización de Hive

Parametrización de consultas Hive

Hive admite la parametrización mediante la sustitución de variables dentro de los scripts HiveQL. Las variables de Hive, que se indican con llaves ({}), se sustituyen por el valor de una variable en tiempo de ejecución. En el listado 7 se muestra un ejemplo de consulta Hive que contiene una variable.

```
hive> SELECT SUM(amt) FROM sales  
> WHERE state = '{hiveconf:state}'  
> GROUP BY state;
```

Listado 7 Consulta HiveQL parametrizada

El valor de la variable puede establecerse de dos maneras:

- Puede establecerse en la consulta o en la sesión mediante el comando SET.
- Se puede pasar en la línea de comandos cuando se ejecuta el script.

En el listado 8 se muestra un ejemplo de establecimiento de una variable en una sesión Hive

```
hive> SET state=NV;
```

Listado 8 Declaración SET

En el listado 8 se muestra un ejemplo de cómo pasar el valor de una variable desde la línea de comandos.

```
$ hive -hiveconf state=NV -f salesbystate.hql > nv_total.txt
```

Listado 8 Establecimiento de una variable Hive a través de la línea de comandos

Uso de scripts personalizados en las consultas de Hive

Al igual que el API de *streaming* de MapReduce, Hive permite utilizar *scripts* externos (es decir, programas externos a Hive) para realizar transformaciones, manipulaciones o cálculos sobre los datos procesados en una consulta Hive. Para ello se utiliza el operador TRANSFORM en Hive.

El operador TRANSFORM en Hive es una abstracción de la API de Streaming de MapReduce. Los datos se pasan a esta API desde Hive y los resultados se devuelven al proceso principal de Hive para su salida o para su incorporación en un procesamiento posterior.

Los *scripts* utilizados para procesar los datos con el operador TRANSFORM pueden escribirse en cualquier lenguaje. El programa externo recibe cada registro como una línea de texto en la entrada estándar (STDIN). El programa realiza sus transformaciones o funciones y devuelve en la salida estándar (STDOUT) un registro de salida delimitado por tabulaciones y terminado en una nueva línea. A los campos de salida se les pueden asignar nombres y tipos de datos para ser utilizados en el procesamiento posterior.

En el Listado 9 se muestra un ejemplo de script de transformación personalizado escrito en Python.

```
#!/usr/bin/env python
import sys
while True:
    line = sys.stdin.readline()
    if not line:
        break
    sales = int(line)
    commission = sales * .1
    print(str(commission))
```

Listado 9 Script de transformación personalizado escrito en Python

Para utilizar tu script personalizado, primero tienes que añadir el archivo a tu programa utilizando la sentencia ADD FILE. Una vez añadido el archivo, puedes referenciar el script utilizando el operador TRANSFORM como se muestra en el Listado 10.

```
hive> ADD FILE commissions.py;
hive> SELECT salesrepid, amt, TRANSFORM(amt)
> USING 'commissions.py' AS commission
> FROM sales;
```

Listado 10 Declaraciones ADD FILE y TRANSFORM

Nota: El uso de scripts externos puede reducir el rendimiento

Al igual que con la API de Streaming en MapReduce y el operador STREAM en Pig, el operador TRANSFORM debe ser utilizado con cuidado, ya que puede degradar el rendimiento. Siempre se debe sopesar el impacto de usar este enfoque contra el esfuerzo requerido para escribir un UDF en Java, para determinar cuál sería más eficiente.

Repasso tipos de datos complejos en Hive

Hive también ofrece soporte para tipos de datos complejos o anidados, que son útiles para el procesamiento iterativo o en varias etapas. Muchos objetos de datos complejos aparecen, como resultado de una función; por ejemplo, una función de procesamiento de texto que devuelve una lista de palabras. Sin embargo, también encontramos estructuras complejas que se producen de forma natural con mayor frecuencia debido a la proliferación de servicios web, API, XML, JSON y otras fuentes de datos semiestructuradas.

Tipo de datos ARRAY

Un tipo de datos ARRAY se utiliza para almacenar una lista ordenada de elementos, todos del mismo tipo de datos; por ejemplo, una lista ordenada de cadenas o una lista ordenada de enteros. Las matrices son una construcción de programación común, que a menudo se denomina lista o secuencia en diferentes lenguajes de programación. Son especialmente útiles para capturar secuencias de eventos o para descomponer texto, como haremos en la siguiente sección.

Las columnas ARRAY se definen en una sentencia Hive CREATE TABLE utilizando corchetes (<>) para definir el tipo de datos de los elementos de la lista. La creación de una tabla con una columna ARRAY se muestra en el Listado 11.

```
hive> CREATE TABLE customers (
>   id INT,
>   fname STRING,
>   lname STRING,
>   email STRING,
>   orderids ARRAY<INT>)
> ROW FORMAT DELIMITED
> FIELDS TERMINATED BY '|'
> COLLECTION ITEMS TERMINATED BY ','
> ;
```

Listado 11 Tipo de datos ARRAY

La cláusula COLLECTION ITEMS TERMINATED BY determina el delimitador utilizado para leer o derivar elementos de la colección. El delimitador por defecto para los elementos de la colección es Ctrl-B, un carácter de control no imprimible. Si se especifica, el delimitador debe ser diferente al utilizado para los campos de la tabla.

Existen varias funciones que pueden utilizarse para crear objetos ARRAY en Hive. Una de ellas es la función array(), que convierte los argumentos de entrada en elementos ARRAY. La función array() se muestra en el listado 12.

```
hive> SELECT array(1,2,3,4,5);
[1,2,3,4,5]
```

Listado 12 Función array()

Dada una columna ARRAY en una tabla Hive, la columna puede ser devuelta como una lista completa, o elementos individuales pueden ser devueltos haciendo referencia a la posición del elemento (como un índice basado en cero). En el listado 13 se muestran algunos ejemplos de cómo acceder a los datos de la columna ARRAY.

```
hive> SELECT orderids[0] FROM customers;
1
```

Listado 13 Selección de elementos ARRAY

Tipo de datos STRUCT

El tipo de datos STRUCT en Hive consiste en campos con nombre que pueden ser del mismo o diferente tipo de datos. Cada atributo STRUCT se define con su tipo de datos asociado, indicado

por paréntesis angulares, de forma similar al tipo de datos ARRAY. En el Listado 14 se muestra un ejemplo de sentencia CREATE TABLE, que incluye una columna con el tipo de datos STRUCT.

```
hive> CREATE TABLE customers (
    > id INT,
    > fname STRING,
    > lname STRING,
    > email STRING,
    > orderids ARRAY<INT>,
    > email_preferences STRUCT<opcomms:boolean, promos:boolean>
    > ROW FORMAT DELIMITED
    > FIELDS TERMINATED BY '|'
    > COLLECTION ITEMS TERMINATED BY ','
    > ;
```

Listado 14 Tipo de dato STRUCT

Al igual que con el tipo ARRAY, la cláusula COLLECTION ITEMS TERMINATED BY se utiliza para separar diferentes elementos en el STRUCT. Al igual que con el ARRAY, este terminador no debe aparecer de forma natural en los propios datos, y debe ser diferente del terminador utilizado para los campos de la tabla.

Para acceder a los elementos del STRUCT en una sentencia SELECT de Hive, es necesario proporcionar el nombre de la columna y el nombre del campo separados por un punto. En el listado 15 se muestra un ejemplo.

```
hive> SELECT email_preferences.opcomms FROM customers;
TRUE
```

Listado 15 Selección de elementos del STRUCT

Tipo de datos MAP

El tipo de datos MAP se utiliza para representar pares clave-valor o pares nombre-valor con tipos de datos para la clave y el valor. Los mapas suelen aparecer en fuentes de datos semiestructuradas como JSON y tienen equivalentes programáticos en la mayoría de los lenguajes; por ejemplo, los diccionarios en Python o los hashes en Ruby.

El listado 16 muestra cómo crear una tabla con una columna del tipo de datos MAP.

```
hive> CREATE TABLE customers (
    > id INT,
    > fname STRING,
    > lname STRING,
    > email STRING,
    > orderids ARRAY<INT>,
    > email_preferences STRUCT<opcomms:boolean, promos:boolean>,
    > address_map MAP<STRING, STRING>
    > ROW FORMAT DELIMITED
    > FIELDS TERMINATED BY '|'
```

- COLLECTION ITEMS TERMINATED BY ','
- MAP KEYS TERMINATED BY ':';

Listado 16 Tipo de datos MAP

Tener en cuenta que, además de la cláusula COLLECTION ITEMS TERMINATED BY antes mencionada, los campos de MAP también incluyen la cláusula MAP KEYS TERMINATED BY, utilizada para discernir entre la clave y el valor dentro de cada elemento del objeto MAP. El carácter por defecto utilizado si no se suministra es el carácter de control no imprimible Ctrl-C.

Se accede a los elementos del MAP por su clave utilizando la sintaxis que se muestra en el Listado 17.

```
hive> SELECT address_map['city'] FROM customers;
Hayward
```

Listado 17 Selección de valores del MAP

Funciones de enmascaramiento y *hashing* de datos en Hive

Los datos sensibles incluyen información personal identificable (PII), como los nombres, o datos del estándar de seguridad de datos de la industria de las tarjetas de pago (PCI DSS), como los números de las tarjetas de crédito. Trabajar con estos datos suele requerir el uso de métodos de ofuscación durante el procesamiento. Hive ofrece varios enfoques para desensibilizar o anonimizar los datos, entre los que se incluyen los siguientes

- enmascaramiento
- *hashing*
- encriptación

Por supuesto, también se pueden eliminar los campos mediante una proyección de columna.

Función MASK

El enmascaramiento está disponible desde la versión 2.1.0 de Hive. Las capacidades de enmascaramiento en Hive se implementan mediante la función MASK. La función MASK simplemente sustituye las letras mayúsculas por una X, las minúsculas por una x y los números por una n. También se pueden suministrar caracteres alternativos para su uso. El listado 18 demuestra el uso de la función MASK.

```
hive> SELECT MASK('abcd-EFGH-8765-4321');
xxxx-XXXX-nnnn-nnnn
```

Listado 18 Función MASK

Funciones de *hashing*

Hive proporciona varios métodos para calcular los *hashes* o resúmenes de una cadena o campo binario determinado. Los *hashes* son valores únicos y deterministas para una entrada determinada y no son fácilmente reversibles. Las funciones de *hash* en Hive incluyen las funciones MD5, SHA, SHA1, SHA2 y CRC32. El listado 19 muestra el hash de una columna en Hive utilizando el método MD5.

```
hive> SELECT MD5('ABC');
902fbdd2b1df0c4f70b4a5d23525e932
```

Listado 19 Función MD5

Cifrado de columnas con Hive

A diferencia del *hashing*, el cifrado proporciona un valor reversible para una columna cifrada que puede recuperarse utilizando la clave secreta utilizada para cifrar el valor. Hive admite los métodos integrados AES_ENCRYPT y AES_DECRYPT para cifrar y descifrar un campo de entrada utilizando el estándar criptográfico AES (Advanced Encryption Standard).

En el Listado 20 se muestra un ejemplo de encriptación y desencriptación de un campo en Hive utilizando una clave secreta.

```
hive> SELECT BASE64(AES_ENCRYPT('ABC', '1234567890123456'));
y6Ss+zCYObpCbgfwfyNWTw==
hive> SELECT AES_DECRYPT(UNBASE64('y6Ss+zCYObpCbgfwfyNWTw=='))
     > , '1234567890123456');
ABC
```

Listado 20 Cifrado y descifrado en Hive

Análisis de direcciones URL en Hive

Las URL (o URI) son un componente común de los registros web y a menudo es necesario descomponerlas en sus elementos discretos. Hive ofrece funciones integradas para analizar las URL en los registros web mediante las funciones PARSE_URL y PARSE_URL_TUPLE.

Los argumentos de entrada de la función PARSE_URL son la columna de la URL y el elemento específico requerido. La función PARSE_URL analiza la URL de entrada y devuelve el elemento solicitado.

El listado 21 muestra la función PARSE_URL en Hive.

```
hive> SELECT PARSE_URL('http://facebook.com/path1/p.php', 'HOST');
facebook.com
```

Listado 21 Función PARSE_URL

La función PARSE_URL_TUPLE es similar a la función PARSE_URL excepto que puede extraer múltiples elementos de una URL dada en una operación, devolviendo los resultados como una tupla. El listado 22 muestra la función PARSE_URL_TUPLE.

```
hive> SELECT PARSE_URL_TUPLE('http://facebook.com/p.php?k1=v1&k2=v2'
     > , 'QUERY:k1', 'QUERY:k2');
v1    v2
```

Listado 22 Función PARSE_URL_TUPLE

Comprensión de la ejecución de consultas en Hive

Las consultas de Hive son planificadas por el optimizador de consultas de Hive. Para entender el plan de ejecución de una consulta determinada, se puede utilizar el comando EXPLAIN como se muestra en el Listado 23.

```
hive> EXPLAIN SELECT method, COUNT(*) FROM weblogs GROUP BY method;
STAGE DEPENDENCIES:
Stage-1 is a root stage
Stage-0 depends on stages: Stage-1
STAGE PLANS:
Stage: Stage-1
Map Reduce
Map Operator Tree:
TableScan
alias: weblogs
...
```

Listado 23 Declaración EXPLAIN

Ejecución paralela en Hive

La ejecución de consultas, especialmente para las operaciones más complejas, se divide generalmente en etapas. Las etapas pueden tener o no dependencias. Si las etapas no tienen dependencias, pueden ejecutarse en paralelo. Sin embargo, esta característica no está habilitada por defecto. Para habilitar la ejecución en paralelo, establezca la propiedad `hive.exec.parallel` a `true` en su archivo `hive-site.xml`.

Gestión y finalización del control de las consultas de Hive (Yarn)

Una vez que envíe una consulta en Hive, la consulta se ejecutará de forma asíncrona en el clúster. No se puede matar la consulta simplemente pulsando `Ctrl-C`, por ejemplo, como se puede hacer en muchas plataformas de bases de datos convencionales. Las consultas de Hive pueden matarse utilizando el comando `yarn`. El listado 24 demuestra el uso del comando `yarn` para listar aplicaciones y luego matar una consulta Hive específica usando su `application_id`.

```
$ yarn application --list
Application-Id Application-Name ...
application_1474361632170_0005 SELECT word, COUNT(*) ...
$ yarn application --kill application_1474361632170_0005
```

Listado 24 Matar una consulta Hive

Ejecución de Hive en modo local

Hive puede ejecutarse en modo local, de forma similar al modo de ejecución local en Pig o al modo de ejecución LocalJobRunner en MapReduce. La ejecución local puede ser particularmente útil para trabajos pequeños, donde la sobrecarga de la ejecución en clúster puede ser excesiva. La ejecución local puede ser forzada usando el comando `SET` como se muestra en el Listado 25.

```
hive> SET mapreduce.framework.name=local;
...
Job running in-process (Local Hadoop)
```

Listado 25 Ejecución local en Hive

Hive puede seleccionar el modo de ejecución local automáticamente utilizando sus propios criterios. Esto se activa estableciendo la propiedad `hive.exec.mode.local.auto` a `true`, ya sea utilizando un comando `SET` en una sesión de Hive o en su archivo `hive-site.xml`.

Procesamiento de texto con Hive

Una de las características menos conocidas de Hive son sus potentes funciones incorporadas de minería de textos, procesamiento de textos y análisis de textos. Veamos ahora algunas de ellas.

Expresiones regulares en Hive

Las expresiones regulares (o Regex) son un enfoque de programación común utilizado para extraer patrones en el texto. El propósito de esta sección no es cubrir las expresiones regulares en sí mismas, ya que hay muchos otros libros, trucos y otros recursos dedicados a este tema. En su lugar, nos centraremos en algunas de las funciones específicas que soportan las expresiones regulares en Hive.

Función REGEXP_EXTRACT

La función `REGEXP_EXTRACT` devuelve el texto de una cadena que coincide con el patrón de expresión regular proporcionado. Esta función es útil para encontrar y devolver una cadena definida dentro de un cuerpo de texto mayor. Un ejemplo de la función `REGEXP_EXTRACT` se muestra en el Listado 26.

Si tengo la tabla

```
hive> select * from emp_info;
OK
emp_id  name    email_id
123      Martin  martin12@gmail.com
445      James   james44@gmail.com
652      Richard richard65@hotmail.com
Time taken: 0.058 seconds, Fetched: 3 row(s)
```

```
regex_extract(email_id, '@(.*)',1)
```

devolverá

```
hive> select name,regexp_extract(email_id,'@(.*)',1) as email_id from emp_info;
OK
name    email_id
Martin  gmail.com
James   gmail.com
Richard hotmail.com
Time taken: 0.07 seconds, Fetched: 3 row(s)
```

Listado 26 Función REGEXP_EXTRACT

Función REGEXP_REPLACE

La función REGEXP_REPLACE se utiliza para sustituir un patrón coincidente en un cuerpo de texto por otra cadena. La función REGEXP_REPLACE es especialmente útil para la manipulación de texto. En el listado 27 se muestra un ejemplo de esta función.

Si tengo la tabla:

id	name	course	institute
121	Robert	Physics	KCT institution
128	Alex	Mathematics	PSA college
178	Justin	Chemistry	KCT college

```
hive> select regexp_replace(institute,"college","University") as institute_name from student;
```

Listado 27 Función REGEXP_REPLACE

Devolverá

```
hive> select regexp_replace(institute,"college","University") as institute_name from student;
OK
institute_name
KCT institution
PSA University
KCT University
Time taken: 0.078 seconds, Fetched: 3 row(s)
```

Funciones de procesamiento de texto

Muchas de las funciones de procesamiento de texto incorporadas en Hive se consideran funciones de tabla, ya que a menudo devuelven uno de los tipos de objetos anidados o complejos tratados en las secciones anteriores. Vamos a analizar ahora algunas de las funciones de texto clave.

Función SPLIT

La función SPLIT toma una cadena y la divide por un delimitador o expresión. La salida resultante es un array de elementos de cadena. Un ejemplo de la función SPLIT se muestra en el Listado 28.

```
hive> SELECT SPLIT('1,2,3,4,5','','');
```

```
[ "1", "2", "3", "4", "5" ]
```

Listado 28 Función SPLIT

Función EXPLODE

La función EXPLODE toma un array de entrada y crea un registro para cada elemento. Consideremos el array creado a partir de la operación SPLIT en el Listado 15. El listado 29 muestra un ejemplo que utiliza la función EXPLODE para aplanar este array en registros individuales.

```
hive> SELECT EXPLODE(SPLIT('1,2,3,4,5','',''));
```

```
1
2
```

Listado 29 Función EXPLODE

Análisis de sentimientos con Hive

Gran parte de la capacidad de análisis de texto incorporada a Hive proviene de su uso en Facebook para el análisis de sentimientos, es decir, la medición de las emociones y opiniones plasmadas en el texto o la conversación. Esto suele ser una característica de los análisis de las redes sociales. Hay varias funciones creadas específicamente para medir el sentimiento en el texto. Entre ellas se encuentran las funciones SENTENCES, NGRAMS y CONTEXT_NGRAMS, que veremos a continuación.

Función SENTENCES

La función SENTENCIAS toma una cadena de entrada que contiene una o más frases separadas por puntos o signos de puntuación. La cadena es analizada en una salida de tipo ARRAY anidado, donde el array exterior contiene un array interior por cada frase, y el array interior contiene un elemento por cada palabra de la frase. La función SENTENCES es a menudo un precursor del procesamiento posterior para el análisis de sentimientos. El listado 30 muestra el uso de la función SENTENCIAS.

```
hive> SELECT SENTENCES('Hello there! How are you?');  
[[ "Hello", "there" ], [ "How", "are", "you" ]]
```

Listado 30 Función SENTENCIAS

Función NGRAMS

Un n-grama es una combinación de palabras o letras en la que n es el número de palabras de la secuencia. Los n-gramas pueden utilizarse para identificar combinaciones de palabras o letras que aparecen juntas con frecuencia en un cuerpo de texto. Los n-gramas son un enfoque de procesamiento de texto muy popular para encontrar palabras, conceptos, tendencias o sentimientos relevantes en el texto, así como para evaluar la similitud de los elementos, incluyendo el documento o el texto.

La función NGRAMS de Hive calcula los n-gramas a partir de una matriz de entrada de frases (normalmente producida por la función SENTENCES) junto con el valor de n. La salida de la función NGRAMS es un tipo STRUCT que consiste en cada n-grama, junto con la frecuencia estimada con la que el n-grama aparece en el texto. El listado 31 demuestra el uso de la función NGRAMS.

```
-- review:  
-- 'This Hadoop book is really helpful!  
-- This Hadoop book helped me greatly!'
```

```
hive> SELECT EXPLODE(NGRAMS(SENTENCES(review), 2, 5));
{"ngram": ["hadoop", "book"], "estfrequency": 2.0}
{"ngram": ["this", "hadoop"], "estfrequency": 2.0}
{"ngram": ["book", "helped"], "estfrequency": 1.0}
{"ngram": ["helped", "me"], "estfrequency": 1.0}
{"ngram": ["really", "helpful"], "estfrequency": 1.0}
```

Listado 31 Función NGRAMS

Consejo: normalizar las mayúsculas y minúsculas al analizar el texto

A menudo se recomienda normalizar las mayúsculas y minúsculas en el texto antes de realizar cualquier procesamiento o análisis del mismo; por ejemplo, convirtiendo las cadenas a minúsculas mediante la función LOWER. Esto evitará que la misma palabra o token se analice de forma diferente; por ejemplo, que la palabra "Hadoop" se trate de forma diferente a la palabra "hadoop".

Función CONTEXT_NGRAMS

Situar el texto en el contexto en el que se utiliza es necesario en el procesamiento del lenguaje natural y el análisis de textos. No tener en cuenta el contexto puede llevar a afirmaciones o conclusiones erróneas.

La función CONTEXT_NGRAMS es similar a la función NGRAMS, pero filtra combinaciones específicas de palabras. Se utiliza para considerar los n-gramas en el contexto en el que se utilizan y, comúnmente, para el análisis del sentimiento. Consiste en un parámetro de entrada adicional que suministra una matriz de palabras a encontrar en el texto, con valores NULL utilizados como marcadores de posición para otras palabras (por ejemplo, palabras de parada).

En el listado 32 se ofrece un ejemplo de la función CONTEXT_NGRAMS.

```
hive> SELECT EXPLODE(CONTEXT_NGRAMS(SENTENCES(review),
  > ARRAY("hadoop", "book", NULL, NULL, NULL), 4, 3));
{"ngram": ["is", "really", "helpful"], "estfrequency": 1.0}
 {"ngram": ["helped", "me", "greatly"], "estfrequency": 1.0}
```

Listado 32 Función CONTEXT_NGRAMS

Función LEVENSHTEIN

La distancia Levenshtein debe su nombre a Vladimir Levenshtein, un científico ruso especializado en la teoría de la información. Es la distancia entre dos cadenas en términos del número de ediciones de caracteres individuales necesarias para convertir una palabra en otra. Cuanto menor sea la distancia Levenshtein, más parecidas son dos palabras o cadenas entre sí. Esta técnica se utiliza habitualmente para encontrar coincidencias cercanas o identificar errores ortográficos en el texto.

Hive incluye la función integrada LEVENSHTEIN, que devuelve la distancia Levenshtein entre dos cadenas. Un ejemplo de la función LEVENSHTEIN se muestra en el listado 33.

```
hive> SELECT LEVENSHTEIN('Jeffery', 'Jeffrey');  
2
```

Listado 33 Función LEVENSHTEIN

Función SOUNDEX

Soundex es un algoritmo para clasificar las palabras por su pronunciación en el idioma inglés. Es especialmente útil para la comparación de nombres, ya que algunos nombres tienen diferentes permutaciones ortográficas, pero fonéticamente suenan igual (un homófono). El algoritmo Soundex está incluido desde hace años en muchas plataformas de gestión de bases de datos relacionales, como SQL Server y Oracle. Devuelve el código Soundex de 4 caracteres que representa cómo suena una palabra según su algoritmo.

En el listado 34 se muestra un ejemplo de la función SOUNDEX en Hive.

```
hive> SELECT SOUNDEX('Jeffrey');  
J160  
hive> SELECT SOUNDEX('Jeffery');  
J160
```

Listado 34 Función SOUNDEX