

Hive



Índice

- INTRODUCCIÓN A HIVE
- TIPOS DE DATOS
- OPERADORES
- CREACIÓN DE TABLAS
- MODIFICACIÓN DE TABLAS
- CARGA Y EXPORTACIÓN DE DATOS
- CONSULTAS
 - SELECT...FROM
 - GROUP BY y HAVING
 - JOIN
 - VIEW
- TIPOS COMPLEJOS
- OPTIMIZACIÓN
- VENTANAS
- PREGUNTAS

¿Qué es Hive?

- Es una infraestructura para el **almacenaje y consulta** de datos basada en Hadoop
- Hadoop proporciona **escalabilidad** masiva y con capacidades de **tolerancia a fallos** para el procesamiento y almacenamiento de datos
- Hive está diseñado para facilitar la **consulta y el análisis** de grandes volúmenes de datos
- Proporciona un sencillo lenguaje de consulta llamado **HiveQL** (o HQL), basado en SQL-92
- Las consultas expresadas en HQL se traducen a trabajos **MapReduce** que se ejecutan en **Hadoop**



¿Qué NO es Hive?

- Hive **no es un gestor de bases de datos relacionales**
- **Es un sistema de procesamiento por lotes (*batch*)**. Por tanto, tiene una **latencia muy alta**
- Para **conjuntos de datos pequeños** su rendimiento no es comparable al de sistemas tradicionales (Oracle, MySQL, etc.)
- Hive no está diseñado para procesamiento de datos **al vuelo** ni ofrece consultas en **tiempo real**

Niveles de granularidad

- **Bases de datos:** espacio de nombres que agrupa tablas y otras unidades de datos
- **Tablas:** unidades de datos homogéneas que comparten un mismo esquema
- **Particiones:** cada tabla puede tener una o más claves de particionado que determinan cómo se almacenan los datos. Optimizan consultas
- **Buckets (o Clusters):** los datos dentro de cada partición pueden, a su vez, dividirse en *buckets* basados en el valor de una función de dispersión sobre alguna columna de la tabla. Optimizan joins.

Tipos de datos primitivos

- Numéricos

Nombre	Sufijo	Tamaño	Precisión
TINYINT	Y	1 byte	-128 a 127
SMALLINT	S	2 bytes	-32,768 a 32,767
INT		4 bytes	-2,147,483,648 a 2,147,483,647
BIGINT	L	8 bytes	-9,223,372,036,854,775,808 a 9,223,372,036,854,775,807
FLOAT		4 bytes	
DOUBLE		8 bytes	
DECIMAL		Arbitrario	Definido por el usuario

Tipos de datos primitivos

■ Fecha y hora

Nombre	Formato	
TIMESTAMP	Numérico (entero)	UNIX timestamp en segundos
	Numérico (coma flotante)	UNIX timestamp en segundos con precisión decimal
	STRING	YYYY-MM-DD HH:MM:SS.ffffffff
DATE	STRING	YYYY-MM-DD

■ Caracteres

Nombre	Longitud
STRING	Variable
VARCHAR	Variable con máximo (1 a 65535)
CHAR	Fija (máximo 255)

■ Otros

Nombre	Valores
BOOLEAN	TRUE/FALSE
BINARY	Bytes arbitrarios (datos en binario)

Tipos de datos complejos

Nombre	Ejemplo de Formato	Creación	Acceso
Arrays	ARRAY<STRING>	array('John', 'Doe')	name[0]
Maps	MAP<STRING, STRING>	map('first', 'John', 'last', 'Doe')	name['first']
Structs	STRUCT {first STRING; last STRING}	struct('John', 'Doe')	name.first
Unions	UNIONTYPE<data_type, data_type, ...>		

Operadores nativos: Relacionales

Operador	Tipos	Descripción
A = B	Primitivos	TRUE si la expresión A es equivalente a la expresión B, sino FALSE
A != B A <> B	Primitivos	TRUE si la expresión A <i>no</i> es equivalente a la expresión B, sino FALSE
A < B	Primitivos	TRUE si la expresión A es menor que la expresión B, sino FALSE
A <= B	Primitivos	TRUE si la expresión A es menor o igual que la expresión B, sino FALSE
A > B	Primitivos	TRUE si la expresión A es mayor que la expresión B, sino FALSE
A >= B	Primitivos	TRUE si la expresión A es mayor o igual que la expresión B, sino FALSE
A IS NULL	Todos	TRUE si la expresión A evalúa a NULL, sino FALSE
A IS NOT NULL	Todos	FALSE si la expresión A evalúa a NULL, sino TRUE
A LIKE B	Strings	TRUE si la cadena A coincide con la expresión regular SQL sencilla B, sino FALSE
A RLIKE B A REGEXP B	Strings	NULL si A o B son NULL, TRUE si cualquier subcadena (posiblemente vacía) de A coincide con la expresión regular de Java B (ver expresiones regulares de Java), sino FALSE

Operadores nativos: Aritméticos

Operador	Tipos	Descripción
A + B	Todos los numéricos	Suma
A - B	Todos los numéricos	Resta
A * B	Todos los numéricos	Multiplicación
A / B	Todos los numéricos	División
A % B	Todos los numéricos	Módulo: resto de la división entera
A & B	Todos los numéricos	Y lógico a nivel de bit (AND)
A B	Todos los numéricos	O lógico a nivel de bit (OR)
A ^ B	Todos los numéricos	O exclusivo lógico a nivel de bit (XOR)
~A	Todos los numéricos	Negación lógica a nivel de bit (NOT)

Operadores nativos: Otros

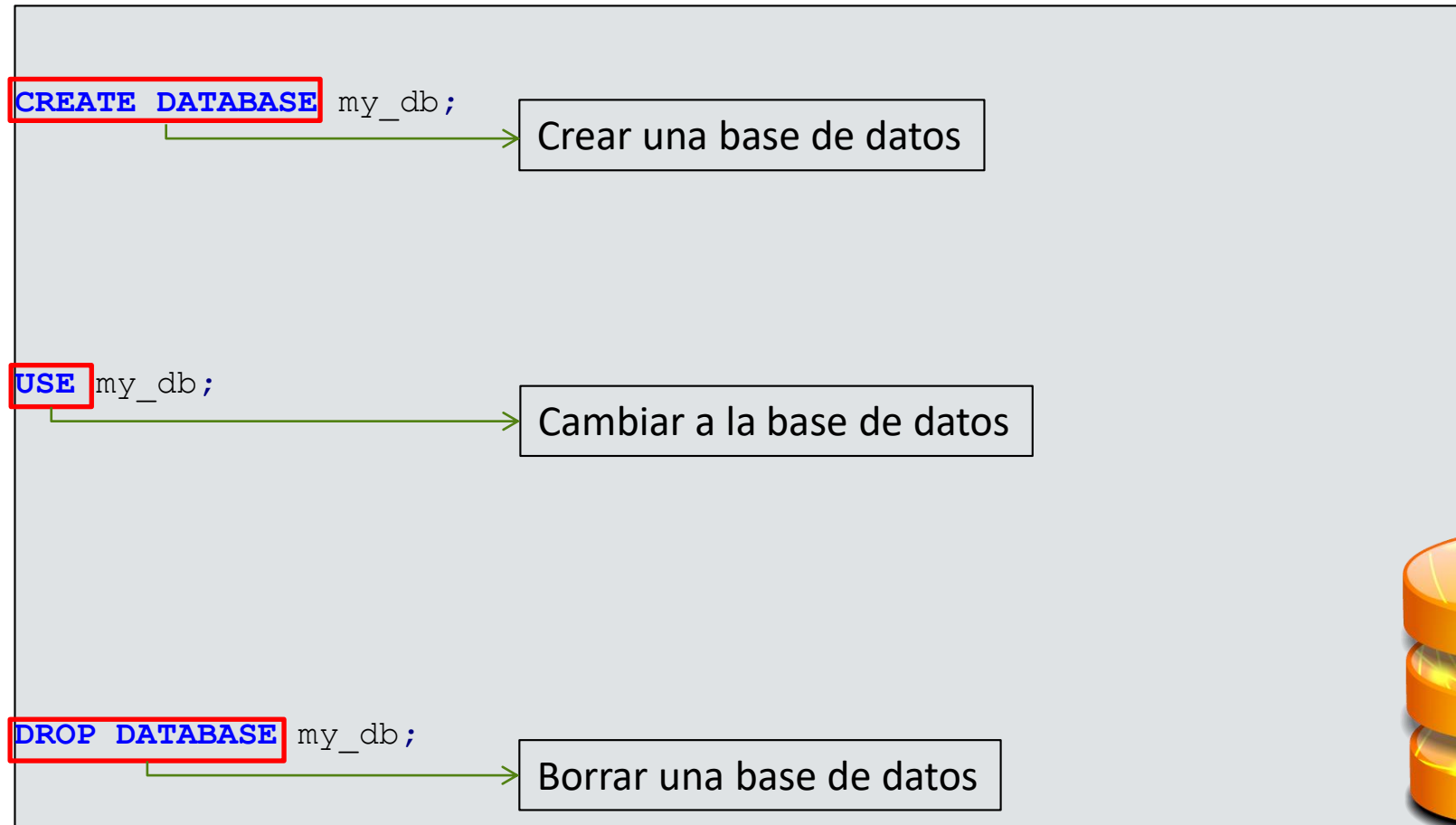
■ Lógicos

Operador	Tipos	Descripción
A AND B A && B	boolean	TRUE si ambos A y B son TRUE, sino FALSE
A OR B A B	boolean	TRUE si A o B o ambos son TRUE, sino FALSE
NOT A !A	boolean	TRUE si A es FALSE, sino FALSE

■ Sobre tipos complejos

Operador	Tipos	Descripción
A[n]	A es un Array y n es un int	Devuelve el n-ésimo elemento del array A. El primer elemento tiene índice 0
M[key]	M es un Map<K, V> y key tiene tipo K	Devuelve el valor correspondiente a la clave en el map
S.x	S es un struct	Devuelve el campo x de S

Bases de datos



Creación de tablas

```
CREATE TABLE page_view  
  viewTime INT, userid BIGINT,  
  page_url STRING, referrer_url STRING,  
  friends ARRAY<BIGINT>,  
  properties MAP<STRING, STRING>,  
  ip STRING COMMENT 'IP Address of the User')  
ROW FORMAT DELIMITED  
  FIELDS TERMINATED BY '1'  
STORED AS SEQUENCEFILE;
```

Nombre de la tabla

Creación de tablas

```
CREATE TABLE page_view(  
  viewTime INT, userid BIGINT,  
  page_url STRING, referrer_url STRING,  
  friends ARRAY<BIGINT>,  
  properties MAP<STRING, STRING>,  
  ip STRING COMMENT 'IP Address of the User')  
  
ROW FORMAT DELIMITED  
  FIELDS TERMINATED BY '1'  
  
STORED AS SEQUENCEFILE;
```

Campos de la tabla

Creación de tablas

```
CREATE TABLE page_view(  
    viewTime INT, userid BIGINT,  
    page_url STRING, referrer_url STRING,  
    friends ARRAY<BIGINT>,  
    properties MAP<STRING, STRING>,  
    ip STRING COMMENT 'IP Address of the User')  
ROW FORMAT DELIMITED  
    FIELDS TERMINATED BY '1'  
STORED AS SEQUENCEFILE;
```

Nombre del campo

Creación de tablas

```
CREATE TABLE page_view(  
    viewTime INT, userid BIGINT,  
    page_url STRING, referrer_url STRING,  
    friends ARRAY<BIGINT>,  
    properties MAP<STRING, STRING>,  
    ip STRING COMMENT 'IP Address of the User')  
ROW FORMAT DELIMITED  
    FIELDS TERMINATED BY '1'  
STORED AS SEQUENCEFILE;
```

Tipo del campo

Creación de tablas

```
CREATE TABLE page_view(  
    viewTime INT, userid BIGINT,  
    page_url STRING, referrer_url STRING,  
    friends ARRAY<BIGINT>,  
    properties MAP<STRING, STRING>,  
    ip STRING COMMENT 'IP Address of the User'  
ROW FORMAT DELIMITED  
    FIELDS TERMINATED BY '1'  
STORED AS SEQUENCEFILE;
```

Descripción del campo

Creación de tablas

```
CREATE TABLE page_view(  
    viewTime INT, userid BIGINT,  
    page_url STRING, referrer_url STRING,  
    friends ARRAY<BIGINT>,  
    properties MAP<STRING, STRING>,  
    ip STRING COMMENT 'IP Address of the User')  
COMMENT 'This is the page view table'
```

Descripción de la tabla

```
ROW FORMAT DELIMITED
```

```
    FIELDS TERMINATED BY '1'
```

```
STORED AS SEQUENCEFILE;
```

Creación de tablas

```
CREATE TABLE page_view(  
    viewTime INT, userid BIGINT,  
    page_url STRING, referrer_url STRING,  
    friends ARRAY<BIGINT>,  
    properties MAP<STRING, STRING>,  
    ip STRING COMMENT 'IP Address of the User')  
COMMENT 'This is the page view table'  
PARTITIONED BY(dt STRING, country STRING)  
ROW FORMAT DELIMITED  
    FIELDS TERMINATED BY '1'  
STORED AS SEQUENCEFILE;
```

Particionada por los
campos dt y country

Creación de tablas

```
CREATE TABLE page_view(  
    viewTime INT, userid BIGINT,  
    page_url STRING, referrer_url STRING,  
    friends ARRAY<BIGINT>,  
    properties MAP<STRING, STRING>,  
    ip STRING COMMENT 'IP Address of the User')  
COMMENT 'This is the page view table'  
PARTITIONED BY(dt STRING, country STRING)  
CLUSTERED BY(userid) SORTED BY(viewTime) INTO 32 BUCKETS  
ROW FORMAT DELIMITED  
    FIELDS TERMINATED BY '1'  
STORED AS SEQUENCEFILE;
```

Los datos se agruparán en
buckets por el campo userid

Creación de tablas

```
CREATE TABLE page_view(  
    viewTime INT, userid BIGINT,  
    page_url STRING, referrer_url STRING,  
    friends ARRAY<BIGINT>,  
    properties MAP<STRING, STRING>,  
    ip STRING COMMENT 'IP Address of the User')  
COMMENT 'This is the page view table'  
PARTITIONED BY(dt STRING, country STRING)  
CLUSTERED BY(userid) SORTED BY(viewTime) INTO 32 BUCKETS  
ROW FORMAT DELIMITED  
    FIELDS TERMINATED BY '1'  
STORED AS SEQUENCEFILE;
```

Dentro de cada bucket se
ordenarán por el campo `viewTime`

Creación de tablas

```
CREATE TABLE page_view(  
    viewTime INT, userid BIGINT,  
    page_url STRING, referrer_url STRING,  
    friends ARRAY<BIGINT>,  
    properties MAP<STRING, STRING>,  
    ip STRING COMMENT 'IP Address of the User')  
COMMENT 'This is the page view table'  
PARTITIONED BY(dt STRING, country STRING)  
CLUSTERED BY(userid) SORTED BY(viewTime) INTO 32 BUCKETS  
ROW FORMAT DELIMITED  
    FIELDS TERMINATED BY '1'  
STORED AS SEQUENCEFILE;
```

Habr  32 buckets

Creación de tablas

```
CREATE TABLE page_view(  
  viewTime INT, userid BIGINT,  
  page_url STRING, referrer_url STRING,  
  friends ARRAY<BIGINT>,  
  properties MAP<STRING, STRING>,  
  ip STRING COMMENT 'IP Address of the User')  
COMMENT 'This is the page view table'  
PARTITIONED BY(dt STRING, country STRING)  
CLUSTERED BY(userid) SORTED BY(viewTime) INTO 32 BUCKETS  
ROW FORMAT DELIMITED  
  FIELDS TERMINATED BY '1'  
STORED AS SEQUENCEFILE;
```

El formato de las filas será campos
separados por un delimitador

Creación de tablas

```
CREATE TABLE page_view(  
  viewTime INT, userid BIGINT,  
  page_url STRING, referrer_url STRING,  
  friends ARRAY<BIGINT>,  
  properties MAP<STRING, STRING>,  
  ip STRING COMMENT 'IP Address of the User')  
COMMENT 'This is the page view table'  
PARTITIONED BY(dt STRING, country STRING)  
CLUSTERED BY(userid) SORTED BY(viewTime) INTO 32 BUCKETS  
ROW FORMAT DELIMITED  
  FIELDS TERMINATED BY '1'  
STORED AS SEQUENCEFILE;
```

El delimitador de campos
será el carácter '1'.

Creación de tablas

```
CREATE TABLE page_view(  
  viewTime INT, userid BIGINT,  
  page_url STRING, referrer_url STRING,  
  friends ARRAY<BIGINT>,  
  properties MAP<STRING, STRING>,  
  ip STRING COMMENT 'IP Address of the User')  
COMMENT 'This is the page view table'  
PARTITIONED BY(dt STRING, country STRING)  
CLUSTERED BY(userid) SORTED BY(viewTime) INTO 32 BUCKETS  
ROW FORMAT DELIMITED  
  FIELDS TERMINATED BY '1'  
  COLLECTION ITEMS TERMINATED BY '2'  
STORED AS SEQUENCEFILE;
```

El delimitador de colecciones
(arrays y maps) será el carácter '2'.

Creación de tablas

```
CREATE TABLE page_view(  
  viewTime INT, userid BIGINT,  
  page_url STRING, referrer_url STRING,  
  friends ARRAY<BIGINT>,  
  properties MAP<STRING, STRING>,  
  ip STRING COMMENT 'IP Address of the User')  
COMMENT 'This is the page view table'  
PARTITIONED BY(dt STRING, country STRING)  
CLUSTERED BY(userid) SORTED BY(viewTime) INTO 32 BUCKETS  
ROW FORMAT DELIMITED  
  FIELDS TERMINATED BY '1'  
  COLLECTION ITEMS TERMINATED BY '2'  
  MAP KEYS TERMINATED BY '3'  
STORED AS SEQUENCEFILE;
```

El delimitador de maps será el carácter '3'.

Creación de tablas

```
CREATE TABLE page_view(  
    viewTime INT, userid BIGINT,  
    page_url STRING, referrer_url STRING,  
    friends ARRAY<BIGINT>,  
    properties MAP<STRING, STRING>,  
    ip STRING COMMENT 'IP Address of the User')  
  
COMMENT 'This is the page view table'  
  
PARTITIONED BY(dt STRING, country STRING)  
  
CLUSTERED BY(userid) SORTED BY(viewTime) INTO 32 BUCKETS  
  
ROW FORMAT DELIMITED  
  
    FIELDS TERMINATED BY '1'  
  
    COLLECTION ITEMS TERMINATED BY '2'  
  
    MAP KEYS TERMINATED BY '3'  
  
    LINES TERMINATED BY '4'  
  
STORED AS SEQUENCEFILE;
```

El delimitador de fin de línea será el carácter '4'.
Por fedecto es '\n'

Creación de tablas

```
CREATE TABLE page_view(  
    viewTime INT, userid BIGINT,  
    page_url STRING, referrer_url STRING,  
    friends ARRAY<BIGINT>,  
    properties MAP<STRING, STRING>,  
    ip STRING COMMENT 'IP Address of the User')  
COMMENT 'This is the page view table'  
PARTITIONED BY(dt STRING, country STRING)  
CLUSTERED BY(userid) SORTED BY(viewTime) INTO 32 BUCKETS  
ROW FORMAT DELIMITED  
    FIELDS TERMINATED BY '1'  
    COLLECTION ITEMS TERMINATED BY '2'  
    MAP KEYS TERMINATED BY '3'  
    LINES TERMINATED BY '4'  
STORED AS SEQUENCEFILE;
```

El formato de almacenamiento
será SEQUENCEFILE

Creación de tablas

```
CREATE EXTERNAL TABLE page_view(  
  viewTime INT, userid BIGINT,  
  page_url STRING, referrer_url STRING,  
  friends ARRAY<BIGINT>,  
  properties MAP<STRING, STRING>,  
  ip STRING COMMENT 'IP Address of the User')  
  
COMMENT 'This is the page view table'  
  
PARTITIONED BY(dt STRING, country STRING)  
  
CLUSTERED BY(userid) SORTED BY(viewTime) INTO 32 BUCKETS  
  
ROW FORMAT DELIMITED  
  
  FIELDS TERMINATED BY '1'  
  
  COLLECTION ITEMS TERMINATED BY '2'  
  
  MAP KEYS TERMINATED BY '3'  
  
  LINES TERMINATED BY '4'  
  
STORED AS SEQUENCEFILE  
  
LOCATION '/user/data/staging/page_view';
```

La tabla será **externa**



Los datos de una tabla externa **NO** se borran en HDFS al borrar la tabla

Una tabla externa necesita especificar la ubicación de la tabla

Exploración de tablas y particiones

- Mostrar las tablas existentes en la base de datos actual

```
SHOW TABLES;
```

- Mostrar las tablas existentes que comiencen por el prefijo page

```
SHOW TABLES 'page.*';
```

- Mostrar las particiones de la tabla page_view

```
SHOW PARTITIONS page_view;
```

- Mostrar información básica (columnas y sus tipos) de la tabla page_view

```
DESCRIBE page_view;
```

- Mostrar la información básica de la tabla page_view y otras propiedades. Útil para depuración

```
DESCRIBE EXTENDED page_view;
```

Modificación de tablas

- Cambio de nombre de una tabla

```
ALTER TABLE old_table_name RENAME TO new_table_name;
```

- Modificación de nombres y/o tipos de columnas

```
ALTER TABLE old_table_name REPLACE COLUMNS (col1 TYPE, ...);
```

- Añadir columnas a una tabla

```
ALTER TABLE tab1 ADD COLUMNS (c1 INT COMMENT 'a new int column', c2 STRING DEFAULT 'def val');
```

- Borrar una tabla

```
DROP TABLE pv_users;
```

- Borrar una partición

```
ALTER TABLE pv_users DROP PARTITION (ds='2008-08-08');
```

Carga de datos

- Copiar datos de un fichero local a una tabla

```
LOAD DATA LOCAL INPATH '/tmp/pv_2008-06-08_us.txt'  
INTO TABLE page_view PARTITION(date='2008-06-08', country='US')
```

- Mover datos de un fichero en HDFS a una tabla

```
LOAD DATA INPATH '/user/data/pv_2008-06-08_us.txt'  
INTO TABLE page view PARTITION(date='2008-06-08', country='US')
```

- Sobrescribir la tabla/ficheros en destino

```
LOAD DATA INPATH '/user/data/pv_2008-06-08_us.txt'  
OVERWRITE INTO TABLE page view PARTITION(date='2008-06-08', country='US')
```


Exportación de datos

■ Copiar datos de una tabla a un fichero local

```
INSERT OVERWRITE LOCAL DIRECTORY '/tmp/ca_employees'  
SELECT name, salary, address FROM employees WHERE se.state = 'CA';
```

Mismo significado:

- LOCAL copia y asume ruta local
- Ausencia de LOCAL mueve los datos y asume ruta en HDFS

Mismo significado:

OVERWRITE sobrescribe datos en destino

Consultas sencillas

- Selección de todas las filas un campo de la tabla page_views

```
SELECT date  
FROM page_views;
```

- Selección de todas las filas varios campos de la tabla page_views

```
SELECT date, user_id  
FROM page_views;
```

- Selección de todas las filas todos los campos de la tabla page_views

```
SELECT *  
FROM page_views;
```

Consultas

- **Cláusula LIMIT:** selección de un cierto número de filas

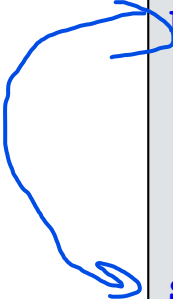
```
SELECT *  
FROM page_views  
LIMIT 10;
```

- Alias de columnas

```
SELECT salary, round(salary - taxes) AS salary_minus_taxes  
FROM employees;
```

Consultas

- Sentencias SELECT anidadas



```
FROM (  
    SELECT round(salary - taxes) AS salary_minus_taxes  
    FROM employees  
    ) e  
SELECT e.salary_minus_taxes;
```

- Sentencias CASE...WHEN...THEN

```
SELECT name, salary,  
    CASE  
        WHEN salary < 50000.0 THEN 'low'  
        WHEN salary >= 50000.0 AND salary < 70000.0 THEN 'middle'  
        WHEN salary >= 70000.0 AND salary < 100000.0 THEN 'high'  
        ELSE 'very high'  
    END AS bracket  
FROM employees;
```

Cláusulas WHERE

- Selección sólo de las filas que cumplan cierta condición

```
SELECT *  
FROM employees  
WHERE country = 'US';
```

- Unión de predicados: AND y OR

```
SELECT *  
FROM employees  
WHERE country = 'US' AND state = 'CA';
```

- Uso de cálculos en los predicados

```
SELECT e.* FROM  
  (SELECT name, salary, deductions["Federal Taxes"] AS ded,  
    salary * (1 - deductions["Federal Taxes"]) AS salary_minus_fed_taxes  
    FROM employees) e  
WHERE round(e.salary_minus_fed_taxes) > 70000;
```

LIKE y RLIKE/REGEXP

- LIKE: Expresiones regulares sencillas

_ : Cualquier carácter una vez

% : Cualquier carácter un número arbitrario de veces

```
SELECT * FROM employees WHERE address LIKE '%Ave';
```

Cadenas que terminen con 'Ave'

```
SELECT * FROM employees WHERE address LIKE 'Ave%';
```

Cadenas que comiencen por 'Ave'

```
SELECT * FROM employees WHERE address LIKE '%Ave%';
```

Cadenas que comiencen, terminen o contengan 'Ave'

- RLIKE (y REGEXP): Expresiones regulares Java

```
SELECT * FROM employees WHERE address RLIKE '.*(Chicago|Ontario).*';
```

Ver expresiones regulares de Java: <http://docs.oracle.com/javase/7/docs/api/java/util/regex/Pattern.html>

LIKE y RLIKE/REGEXP

- Algunos ejemplos

- 1: Un string Literal que concide con un valor literal.
- 2: Matches un digito, el primero que encuentra.
- 3: Matches exactamente 5 dígitos consecutivos.
- 4: Un dígito seguido de un espacio en blanco (espacio, tabulador, ...) y a continuación uno o más caracteres (cualquiera)
- 5: Palabras de entre 5 y 9 caracteres.
- 6: Cualquier caracter seguido de un punto.
- 7: Cualquier número de caracteres seguido de un punto.

Regular Expression	String (matched portion in bold)
Dualcore	I wish Dualcore had 2 stores in 90210.
\\d	I wish Dualcore had 2 stores in 90210.
\\d{5}	I wish Dualcore had 2 stores in 90210 .
\\d\\s\\w+	I wish Dualcore had 2 stores in 90210.
\\w{5,9}	I wish Dualcore had 2 stores in 90210.
.?\\.	I wish Dualcore had 2 stores in 9021 0 .
.*\\.	I wish Dualcore had 2 stores in 90210 .



Solución

REGEX SerDe

- A veces hay que analizar datos que no tienen delimitadores específicos o claros

- Por ejemplo Logs.

```
05/23/2013 19:45:19 312-555-7834 CALL_RECEIVED ""
05/23/2013 19:45:23 312-555-7834 OPTION_SELECTED "Shipping"
05/23/2013 19:46:23 312-555-7834 ON_HOLD ""
05/23/2013 19:47:51 312-555-7834 AGENT_ANSWER "Agent ID N7501"
05/23/2013 19:48:37 312-555-7834 COMPLAINT "Item not received"
05/23/2013 19:48:41 312-555-7834 CALL_END "Duration: 3:22"
```

- Para estos casos, RegexSerDe leerá registros en base a una expresión regular
- Cosa que nos permite crear tablas a partir del dato que estemos leyendo, en este caso un Log

REGEX SerDe: ejemplo

05/23/2013 19:45:19 312-555-7834 CALL_RECEIVED ""
05/23/2013 19:48:37 312-555-7834 COMPLAINT "Item not received"

Log excerpt

```
> CREATE TABLE calls (  
  event_date STRING,  
  event_time STRING,  
  phone_num STRING,  
  event_type STRING,  
  details STRING)  
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.RegexSerDe'  
WITH SERDEPROPERTIES ("input.regex" =  
  "([^ ]*) ([^ ]*) ([^ ]*) ([^ ]*) \"([^\"]*)\"");
```

RegexSerDe

event_date	event_time	phone_num	event_type	details
05/23/2013	19:45:19	312-555-7834	CALL_RECEIVED	
05/23/2013	19:45:37	312-555-7834	COMPLAINT	Item not received

Table excerpt

- Cada par de paréntesis corresponde a uno de los campos

REGEX SerDe

- En otras ocasiones, los datos tienen un formato fijo y delimitado



- Para estos casos, RegexSerDe también nos servirá.
 - Por defecto RegexSerDe lee datos de tipo string.
 - Si queremos leer numéricos hay que castear.

REGEX SerDe

1030929610759620120829012215Oakland

CA94618

Input data

```
CREATE TABLE fixed (  
  cust_id STRING,  
  order_id STRING,  
  order_dt STRING,  
  order_tm STRING,  
  city STRING,  
  state STRING,  
  zip STRING)  
ROW FORMAT SERDE  
  'org.apache.hadoop.hive.serde2.RegexSerDe'  
WITH SERDEPROPERTIES ("input.regex" =  
  "(\\d{7})(\\d{7})(\\d{8})(\\d{6})(.{20})(\\w{2})(\\d{5})");
```

RegexSerDe

cust_id	order_id	order_dt	order_tm	city	state	zipcode
1030929	6107596	20120829	012215	Oakland	CA	94618

GROUP BY y HAVING

- Cláusula GROUP BY, habitualmente usada junto con una función de agregación

```
SELECT year(ymd), avg(price_close) FROM stocks  
WHERE exchange = 'NASDAQ' AND symbol = 'AAPL'  
GROUP BY year(ymd);
```

Se agrupan los resultados por una o más columnas

Se realiza una operación de agregación sobre cada grupo

La(s) columna(s) usadas para la agrupación deben ser seleccionadas en la consulta

- Cláusula HAVING: restringe los grupos producidos por GROUP BY

```
SELECT year(ymd), avg(price_close) FROM stocks  
WHERE exchange = 'NASDAQ' AND symbol = 'AAPL'  
GROUP BY year(ymd)  
HAVING avg(price_close) > 50.0;
```

JOIN

- Une filas de dos o más tablas que tengan el mismo valor en una o más columnas

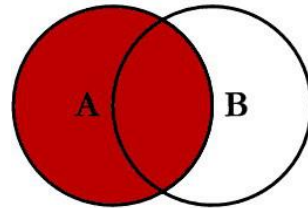
```
SELECT a.ymd, a.price_close, b.price_close  
FROM stocks a JOIN stocks b  
ON a.ymd = b.ymd  
WHERE a.symbol = 'AAPL' AND b.symbol = 'IBM';
```

Une filas de las tablas a y b

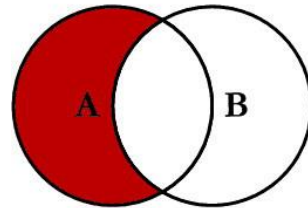
Cuyos respectivos campos
ymd sean iguales

Tipos de JOIN

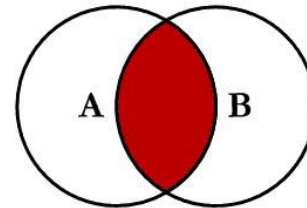
SQL JOINS



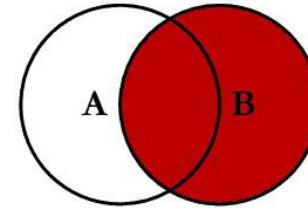
```
SELECT <select_list>  
FROM TableA A  
LEFT JOIN TableB B  
ON A.Key = B.Key
```



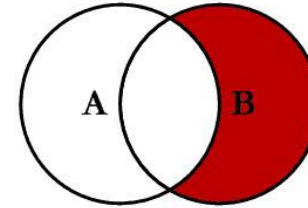
```
SELECT <select_list>  
FROM TableA A  
LEFT JOIN TableB B  
ON A.Key = B.Key  
WHERE B.Key IS NULL
```



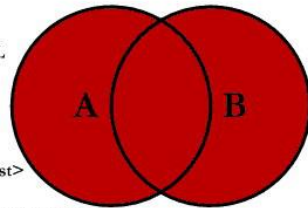
```
SELECT <select_list>  
FROM TableA A  
INNER JOIN TableB B  
ON A.Key = B.Key
```



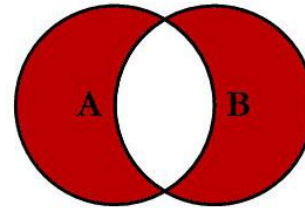
```
SELECT <select_list>  
FROM TableA A  
RIGHT JOIN TableB B  
ON A.Key = B.Key
```



```
SELECT <select_list>  
FROM TableA A  
RIGHT JOIN TableB B  
ON A.Key = B.Key  
WHERE A.Key IS NULL
```



```
SELECT <select_list>  
FROM TableA A  
FULL OUTER JOIN TableB B  
ON A.Key = B.Key
```



```
SELECT <select_list>  
FROM TableA A  
FULL OUTER JOIN TableB B  
ON A.Key = B.Key  
WHERE A.Key IS NULL  
OR B.Key IS NULL
```

© C.L. Moffatt, 2008

ORDER BY y SORT BY

- ORDER BY: orden total, se envían todos los resultados a un reducer

```
SELECT s.ymd, s.symbol, s.price_close  
FROM stocks s  
ORDER BY s.ymd ASC, s.symbol DESC;
```

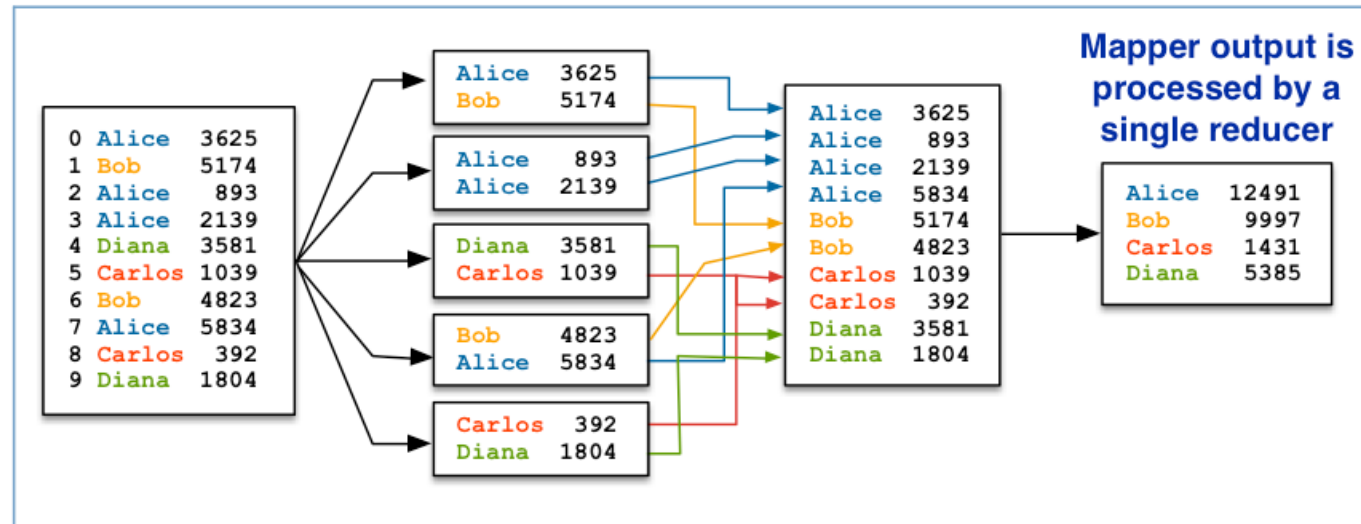
- SORT BY: orden local, cada reducer ordena sus resultados

```
SELECT s.ymd, s.symbol, s.price_close  
FROM stocks s  
SORT BY s.ymd ASC, s.symbol DESC;
```

Hive:Optimización: Sort By

- Sabemos que Order By ordena las filas de una tabla por los valores de una columna.
- Ejemplo

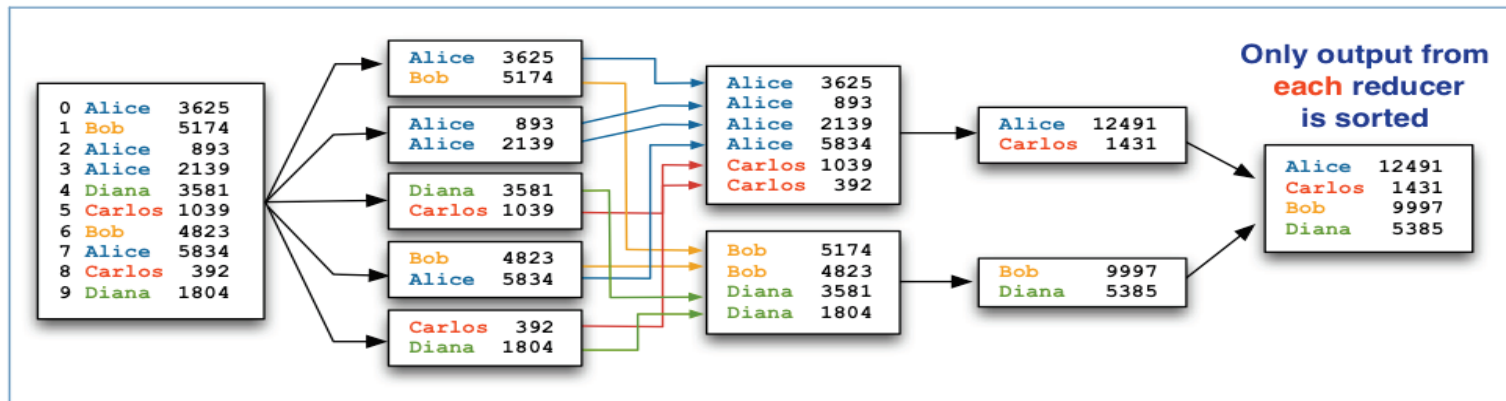
```
SELECT name, SUM(total)
FROM order_info GROUP BY name
ORDER BY name;
```



Hive:Optimización: Sort By

- Esto obliga a tener que ordenar toda la tabla.
- Hay veces en las que no es necesario tener un orden GLOBAL de la tabla para el propósito de nuestra query, por lo que podemos utilizar la función SORT BY, que solo ordena localmente las filas, por lo que es mucho más rápido de ejecutar.

```
SELECT name, SUM(total)
FROM order_info GROUP BY name
SORT BY name;
```



DISTRIBUTE BY con SORT BY

- **DISTRIBUTE BY**
 - **Controla cómo la salida del map se divide entre los reducers**
 - Hive usa esta característica internamente cuando convierte las consultas a trabajos MapReduce
 - Habitualmente no es necesario preocuparse por esta funcionalidad, a menos que se use Hive Streaming o algunas UDAFs (User Define Aggregation Funtion)
 - Por defecto, Hadoop intenta distribuir las filas uniformemente entre los reducers
- **DISTRIBUTE BY...SORT BY: asegura que las filas que tengan el mismo valor en cierto campo vayan al mismo reducer**
- **CLUSTER BY campo = DISTRIBUTE BY campo SORT BY campo ASC**

Casting

- La función cast() permite convertir explícitamente un valor de un tipo a otro tipo diferente
 - Por ejemplo, si el campo salary es de tipo STRING, lo podemos convertir a un número en coma flotante de la siguiente forma:

```
SELECT name, salary
FROM employees
WHERE cast(salary AS FLOAT) < 100000.0;
```

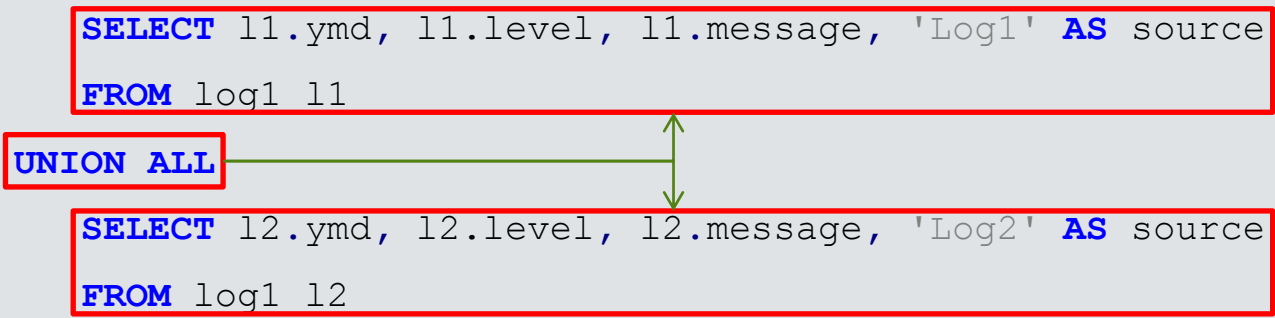
- Para hacer casting de tipos BINARY hay que convertirlos primero a STRING

```
SELECT (2.0 * cast(cast(b AS STRING) AS DOUBLE)) from src;
```

UNION ALL

- Une las filas resultantes de dos o más subconsultas
 - Para ello las subconsultas deben devolver el mismo número de columnas
 - Y cada columna debe ser del mismo tipo en cada subconsulta

```
SELECT log.ymd, log.level, log.message
FROM (
    SELECT l1.ymd, l1.level, l1.message, 'Log1' AS source
    FROM log1 l1
    UNION ALL
    SELECT l2.ymd, l2.level, l2.message, 'Log2' AS source
    FROM log1 l2
) log
SORT BY log.ymd ASC;
```



VIEW

- Cuando una consulta se vuelve demasiado complicada, se usan **vistas** para reducir la complejidad dividiendo la consulta en partes más pequeñas y manejables.
- Similares a las funciones en lenguajes de programación.

```
FROM (  
    SELECT * FROM people JOIN cart  
    ON (cart.people_id=people.id) WHERE firstname='john'  
) a SELECT a.lastname WHERE a.id=3;
```

```
CREATE VIEW shorter_join AS  
SELECT * FROM people JOIN cart  
ON (cart.people_id=people.id) WHERE firstname='john';
```

```
SELECT lastname FROM shorter_join WHERE id=3;
```

Tipos complejos en Hive

- Hive soporta varios tipos complejos de columnas
 - **Array**: listas ordenadas de valores, todos del mismo tipo
 - **Map**: pares K-V, todos del mismo tipo
 - **Struct**: estructuras con tipos de datos distintos entre sí
- Estos tipos pueden ser más eficientes
 - Una tabla en lugar de un JOIN

Tipos complejos en Hive: Ejemplos

- **Ejemplo 1:** Cómo mostrar los números de teléfono de los clientes
- El modo tradicional, usar dos tablas unidas por un JOIN

customers table

cust_id	name
a	Alice
b	Bob
c	Carlos

```
SELECT c.cust_id, c.name, p.phone
FROM customers c
JOIN phone p
ON (c.cust_id = p.cust_id)
```

phones table

cust_id	phone
a	555-1111
a	555-2222
a	555-3333
b	555-4444
c	555-5555
c	555-6666

query results

cust_id	name	phone
a	Alice	555-1111
a	Alice	555-2222
a	Alice	555-3333
b	Bob	555-4444
c	Carlos	555-5555
c	Carlos	555-6666

Tipos complejos en Hive: Ejemplos

- **Ejemplo 1:** Cómo mostrar los números de teléfono de los clientes
- Una forma más eficiente de hacerlo sería usando ARRAYS

customers_phones table

cust_id	name	phones
a	Alice	[555-1111, 555-2222, 555-3333]
b	Bob	[555-4444]
c	Carlos	[555-5555, 555-6666]

```
SELECT name,  
       phones[0],  
       phones[1]  
FROM customers_phones;
```



query results

name	phones[0]	phones[1]
Alice	555-1111	555-2222
Bob	555-4444	NULL
Carlos	555-5555	555-6666

Tipos complejos en Hive: Ejemplos

- **Ejemplo 1:** cómo almacenar números de teléfono de clientes
- La forma de definir la estructura de una tabla así es esta

```
CREATE TABLE customers_phones  
  (cust_id STRING,  
   name STRING,  
   phones ARRAY<STRING>)  
ROW FORMAT DELIMITED  
FIELDS TERMINATED BY ','  
COLLECTION ITEMS TERMINATED BY '|';
```

Data File

```
a,Alice,555-1111|555-2222|555-3333  
b,Bob,555-4444  
c,Carlos,555-5555|555-6666
```

Tipos complejos en Hive: Ejemplos

- **Ejemplo 1:** cómo almacenar números de teléfono de clientes
- Otra forma de resolverlo es con la estructura MAP: K-V
- Si un usuario tiene varios teléfonos bajo la misma clave, solo devolverá el primero

customers_phones table

cust_id	name	phones
a	Alice	{home: 555-1111, work: 555-2222, mobile: 555-3333}
b	Bob	{mobile: 555-4444}
c	Carlos	{work: 555-5555, home: 555-6666}

```
SELECT name,  
       phones['home'] as home  
FROM customers_phones;
```

query results

name	home
Alice	555-1111
Bob	NULL
Carlos	555-6666

Tipos complejos en Hive: Ejemplos

- **Ejemplo 1:** cómo almacenar números de teléfono de clientes
- La forma de definir la estructura de una tabla así es esta

```
CREATE TABLE customers_phones
(cust_id STRING,
 name STRING,
 phones MAP<STRING,STRING>)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
COLLECTION ITEMS TERMINATED BY '|'
MAP KEYS TERMINATED BY ':';
```

Data File

```
a,Alice,home:555-1111|work:555-2222|mobile:555-3333
b,Bob,mobile:555-4444
c,Carlos,work:555-5555|home:555-6666
```

Tipos complejos en Hive: Ejemplos

- **Ejemplo 1:** cómo almacenar números de teléfono de clientes
- Esta vez usando una estructura de tipo STRUCT

customers_addr table

cust_id	name	address
a	Alice	{street:742 Evergreen Terrace, city:Springfield state:OR zipcode:97477}
b	Bob	{street:1600 Pennsylvania Ave NW city:Washington state:DC zipcode:20500}
c	Carlos	{street:342 Gravelpit Terrace city:Bedrock}

```
SELECT name,  
       address.state,  
       address.zipcode  
FROM customers_addr;
```



query results

name	state	zipcode
Alice	OR	97477
Bob	DC	20500
Carlos	NULL	NULL

Tipos complejos en Hive: Ejemplos

- **Ejemplo 1:** cómo almacenar números de teléfono de clientes
- Esta vez usando una estructura de tipo STRUCT
- Sobre esta estructura se pueden seleccionar fácilmente valores de la estructura completa

```
SELECT name,  
       address  
FROM customers_addr;
```



query results

name	state
Alice	742 Evergreen Terrace Springfield OR 97477
Bob	1600 Pennsylvania Ave NW Washington DC 20500
Carlos	342 Gravelpit Terrace Bedrock

Tipos complejos en Hive: Ejemplos

- **Ejemplo 1:** cómo almacenar números de teléfono de clientes
- La forma de definir la estructura de una tabla así, es la siguiente:

```
CREATE TABLE customers_addr
(cust_id STRING,
 name STRING,
 address STRUCT<street:STRING,
                city:STRING,
                state:STRING,
                zipcode:STRING>)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
COLLECTION ITEMS TERMINATED BY '|';
```

Data File

```
a,Alice,742 Evergreen Terrace|Springfield|OR|97477
b,Bob,1600 Pennsylvania Ave NW|Washington|DC|20500
c,Carlos,342 Gravelpit Terrace|Bedrock
```

Tipos complejos en Hive: Ejemplos

- Al trabajar con estructuras complejas, especialmente ARRAYS o MAPS, es importante poder conocer el número de elementos que contienen
- Con MAPS, se devolvería el número de pares K-V

cust_id	name	phones
a	Alice	[555-1111, 555-2222, 555-3333]
b	Bob	[555-4444]
c	Carlos	[555-5555, 555-6666]

```
SELECT name, SIZE(phones)
FROM customers_phones;
```



query results

name	_c1
Alice	3
Bob	1
Carlos	2

Tipos complejos en Hive: Ejemplos

La forma de trabajar será:

- Originalmente tenemos una o varias tablas en formato tradicional

```
CREATE EXTERNAL TABLE IF NOT EXISTS a (  
  date string,  
  acct string,  
  media string,  
  id1 string,  
  val INT )  
...
```

- Se crea una nueva tabla con la estructura compleja deseada y se hace el insert de los datos originales en la nueva tabla.

```
CREATE EXTERNAL TABLE IF NOT EXISTS b (  
  date string,  
  acct string,  
  media string,  
  st1 STRUCT<id1:STRING, val:BIGINT> )  
...
```

```
FROM a  
INSERT OVERWRITE TABLE b PARTITION (day='{DATE}')  
SELECT date,  
  acct,  
  media,  
  named_struct('id1',id1,'val',sum(val))  
WHERE ...
```


Tipos complejos en Hive: EXPLODE

- Una de las funciones usadas en Hive avanzado es **EXPLODE**
 - Esta función crea un registro por cada elemento en un array (es muy usado con vistas laterales)
 - Con MAPS, cada K-V se convierte en un registro separado, y cada K y V se convierten en campos separados dentro de cada registro

cust_id	name	phones
a	Alice	[555-1111, 555-2222, 555-3333]
b	Bob	[555-4444]
c	Carlos	[555-5555, 555-6666]

```
SELECT EXPLODE(phones) AS phone  
FROM customers_phones;
```

query results

phone
555-1111
555-2222
555-3333
555-4444
555-5555
555-6666



Solo se puede indicar una columna en el SELECT

Hive: Sentiment Análisis

- El análisis del sentimiento no es más que la aplicación de Text Analytics
 - Clasificación y medida de opiniones
 - Frecuentemente usado en análisis de RRSS
- El contexto es una parte esencial de la comunicación entre humanos
 - ¿Qué combinación de palabras aparecen juntas?
 - ¿Cómo frecuente es esta combinación?
- Hive dispone de funciones para ayudarnos en este tipo de análisis

Hive: Sentiment Análisis

- Como hemos visto antes, Hive posee funciones muy interesantes para trabajar con texto: ***SPLIT, EXPLODE***

```
SELECT people FROM example;  
Amy, Sam, Ted
```

```
SELECT SPLIT(people, ',') FROM example;  
["Amy", "Sam", "Ted"]
```

```
SELECT EXPLODE(SPLIT(people, ',')) AS x FROM example;  
Amy  
Sam  
Ted
```

Hive: Sentiment Análisis

- La función **SENTENCES** divide un texto en palabras
- En este caso, la entrada es un string que contiene una o más frases
- La salida es un array bidimensional de strings
 - En el caso de debajo tenemos un array que contiene dos arrays, uno por cada frase de entrada

```
SELECT txt FROM phrases WHERE id=12345;  
I bought this computer and really love it! It's very fast and  
does not crash.
```

```
SELECT SENTENCES(txt) FROM phrases WHERE id=12345;  
[["I","bought","this","computer","and","really","love","it"],  
 ["It's","very","fast","and","does","not","crash"]]
```

Hive: Sentiment Análisis: n-grams

- Un **n-grama** es una combinación de “n” palabras
- Se suele utilizar para:
 - **Sugerir** corrección de palabras cuando hacemos búsquedas
 - **Encontrar** las palabras más importantes en los textos
 - **Identificar** trending topics

Hive: Sentiment Análisis: n-grams

- Hive ofrece la función **NGRAMS** para calcular n-gramas
- Esta función requiere tres parámetros
 - Array de strings (frases), donde cada una contiene otro array de strings (palabras)
 - Número de palabras en el n-grama
 - Numero de resultados deseados (top-N, basado en frecuencia)
- La salida es un array de Structs con dos atributos
 - *Ngram*: el propio n-gram (un array de palabras)
 - *Estfrecuency*: la frecuencia estimada a la que este n-gram aparece

Hive: Sentiment Análisis: n-grams

- Los n-gramas se suelen utilizar con la función **SENTENCES** (divide una frase en palabras)
 - También se utiliza mucho la función **LOWER** para normalizar a minúsculas
 - Y la función **EXPLODE**, para convertir el array resultado a una serie de filas
- Ejemplo:

```
SELECT txt FROM phrases WHERE id=56789;  
This tablet is great. The size is great. The screen is  
great. The audio is great. I love this tablet! I love  
everything about this tablet!!!
```

```
SELECT EXPLODE(NGRAMS(SENTENCES(LOWER(txt)), 2, 5))  
AS bigrams FROM phrases WHERE id=56789;  
{"ngram":["is","great"],"estfrequency":4.0}  
{"ngram":["great","the"],"estfrequency":3.0}  
{"ngram":["this","tablet"],"estfrequency":3.0}  
{"ngram":["i","love"],"estfrequency":2.0}  
{"ngram":["tablet","i"],"estfrequency":1.0}
```

Ngram(nº
palabras, nº
filas)

Hive: Sentiment Análisis: n-grams

- Es posible especificar el contenido o parte de él que han de tener los n-gramas que buscamos. Para ello usamos la función CONTEXT_NGRAMS.
 - A la función se le pasa el array de palabras que ha de contener el n-grama
 - Los valores NULL representan contenedores

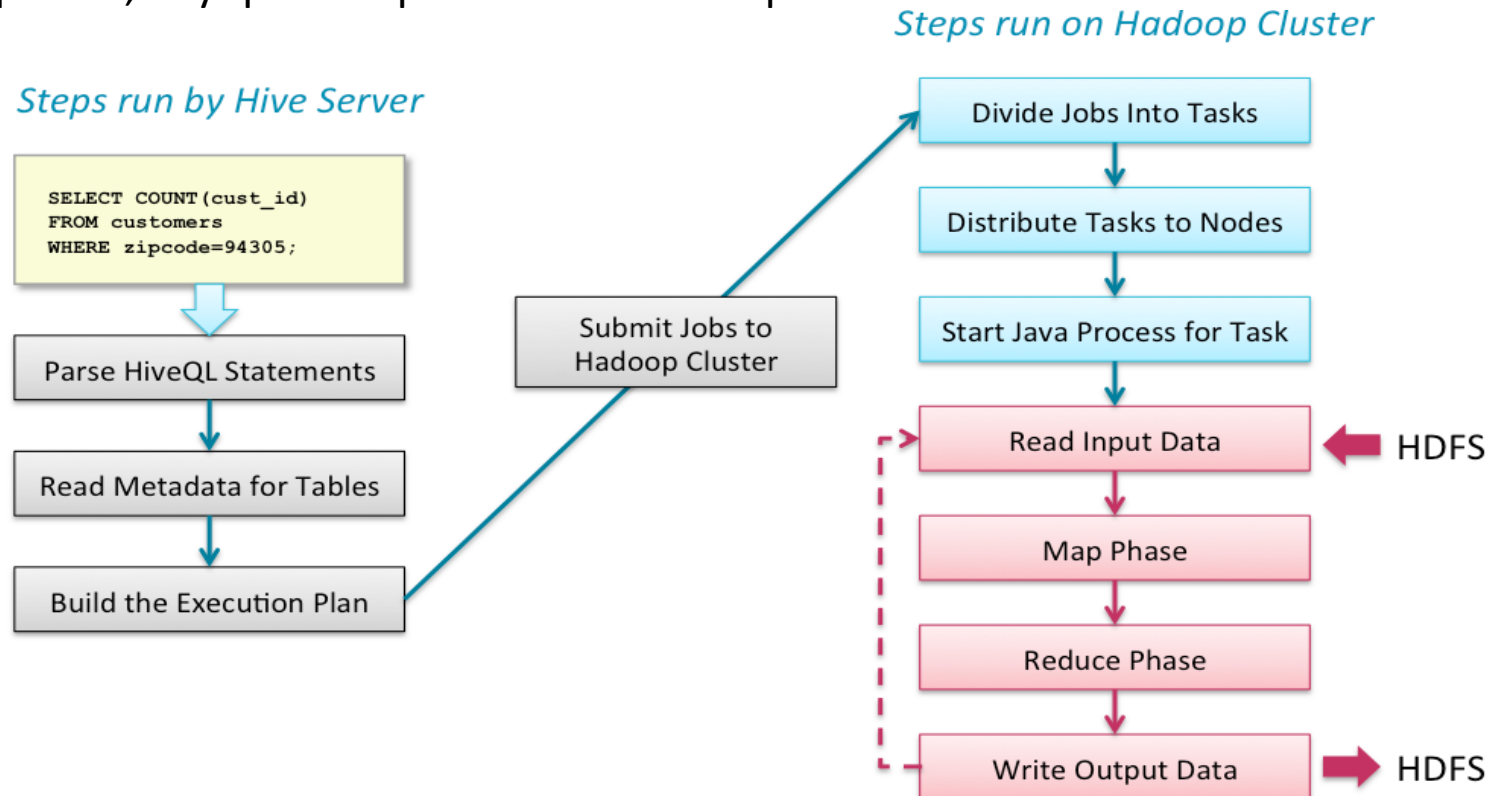
- Ejemplo:

```
hive> SELECT txt FROM phrases
        WHERE txt LIKE '%new computer%';
My new computer is fast! I wish I'd upgraded sooner.
This new computer is expensive, but I need it now.
I can't believe her new computer failed already.

hive>SELECT EXPLODE(CONTEXT_NGRAMS(SENTENCES(LOWER(phrase)),
        ARRAY("new", "computer", NULL, NULL), 4, 3)) AS ngrams
FROM phrases;
{"ngram":["is","expensive"],"estfrequency":1.0}
{"ngram":["failed","already"],"estfrequency":1.0}
{"ngram":["is","fast"],"estfrequency":1.0}
```

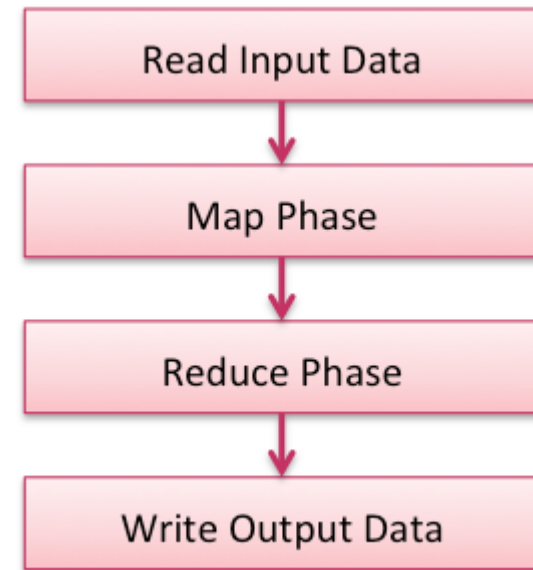

Hive:Optimización

- Como hemos dicho durante el curso, Hive ejecuta internamente Map Reduce.
- Para optimizar las queries, hay que comprender cómo son procesadas



Hive:Optimización: Plan Ejecución

- Como ya sabemos, un **job** consiste en dos fases, **Map** y **Reduce**.
 - La salida del map es la entrada del reduce
- El **Map** siempre se ejecuta primero
 - Se utiliza para filtrar, transformar o parsear el dato
 - Las filas se procesan de una en una
- El **reduce** es opcional
 - Se utiliza para agrupar los datos de un map
 - Agrega múltiples filas
- Recordad el diagrama del primer día



Hive:Optimización : Plan Ejecución

- El query plan de Hive tiene tres secciones principales

```
EXPLAIN CREATE TABLE cust_by_zip AS
  SELECT zipcode, COUNT(cust_id) AS num
  FROM customers GROUP BY zipcode;

ABSTRACT SYNTAX TREE:
  (TOK_CREATETABLE (TOK_TABNAME cust_by_zip) ...

STAGE DEPENDENCIES:
  ... (excerpt shown on next slide)

STAGE PLANS:
  ... (excerpt shown on upcoming slide)
```

contendría algo así:

Poco útil para nosotros.

```
(TOK_CREATETABLE (TOK_TABNAME cust_by_zip) TOK_LIKETABLE
(TOK_QUERY (TOK_FROM (TOK_TABREF (TOK_TABNAME customers)))
(TOK_INSERT (TOK_DESTINATION (TOK_DIR TOK_TMP_FILE))
(TOK_SELECT (TOK_SELEXPR (TOK_TABLE_OR_COL zipcode))
(TOK_SELEXPR (TOK_FUNCTION COUNT (TOK_TABLE_OR_COL
cust_id)))) (TOK_GROUPBY (TOK_TABLE_OR_COL zipcode))))))!
```

Hive:Optimización : Plan Ejecución

- La segunda sección, la de las dependencias, tiene cuatro estados (no 4 jobs)
 - Stage 1. Primero.
 - Stage 0
 - Stage 3
 - Stage 2. Último.
- Solo se muestran las partes más relevantes

ABSTRACT SYNTAX TREE:

... (shown on previous slide)

STAGE DEPENDENCIES:

Stage-1 is a root stage

Stage-0 depends on stages: Stage-1

Stage-3 depends on stages: Stage-0

Stage-2 depends on stages: Stage-3

STAGE PLANS:

... (shown on next slide)

Hive:Optimización : Plan Ejecución

- El stage 1 es un job Map Reduce

- Fase Map
 - Lee la tabla customers
 - Selecciona zipcode y cust_id
- Fase reduce
 - Group by zipcode
 - Count cust_id

STAGE PLANS:

Stage: Stage-1
Map Reduce

Alias -> Map Operator Tree:

TableScan

alias: customers

Select Operator

zipcode, cust_id

Reduce Operator Tree:

Group By Operator

aggregations:

expr: count(cust_id)

keys:

expr: zipcode

Hive:Optimización : Plan Ejecución

- El **stage 0** es una acción HDFS
 - Mueve la salida del stage anterior al directorio warehouse de Hive

STAGE PLANS:

Stage: Stage-1 (covered earlier)...

Stage: Stage-0

Move Operator

files:

hdfs directory: true

destination: (HDFS path...)

Hive:Optimización : Plan Ejecución

- El **stage 3** es una acción en el Metastore
 - Crea una tabla con dos columnas
- El **stage 2** es una colección de estadísticas
 - Número de filas y columnas en la tabla
 - Número de valores únicos en cada columna
 - Etc.

STAGE PLANS:

Stage: Stage-1 (covered earlier) ...

Stage: Stage-0 (covered earlier) ...

Stage: **Stage-3**

Create Table Operator:

Create Table

columns: **zipcode** string,
num bigint

name: **cust_by_zip**

Stage: **Stage-2**

Stats-Aggr Operator

Hive:Optimización: Bucketing

- El **partitioning** es una técnica de optimización basada en dividir los datos de una tabla en función del valor de sus columnas, en función del uso que le vayamos a dar.
- El **bucketing** es otra forma de subdividir datos para ganar eficiencia
 - Calcula un código Hash para los valores insertados en las columna “bucket”
 - Divide los datos en base a esa columna
 - El código Hash es usado para asignar nuevos valores al bucket correspondiente
 - $\text{Hash_fn}(\text{bucketing_column}) \bmod \text{num_buckets}$
- En **bucketing**, el número de particiones es fija. En partitioning no.
- En **partitioning**, todas las filas de la partición tienen la misma key.
- El objetivo del **bucket** es el de distribuir filas a lo largo de un número predefinido de buckets
 - Útil para Jobs donde se necesitan samples aleatorios de datos
 - El join de dos tablas que han sido bucketizadas sobre la misma columna del join, puede ser implementado más eficientemente como un Map Join. TDG3e p.433.
- Es una opción muy válida para columnas donde el partitioning no es válido por el problema de “*muchos valores con pocos registros*”

Hive:Optimización: Bucketing

- Para crear una tabla que soporte 20 buckets (5% del total de los datos sobre cada uno) sobre el la columna order_id se usa el comando **CLUSTERED BY**
- Es importante recordar que la columna sobre la que se hace el bucket tenga datos bien distribuidos.

```
CREATE TABLE orders_bucketed  
  (order_id INT,  
   cust_id INT,  
   order_date TIMESTAMP)  
  CLUSTERED BY (order_id) INTO 20 BUCKETS;
```

Hive:Optimización: Bucketing

- Los datos no se insertan automáticamente en una tabla preparada para tal efecto.
- Físicamente, en el directorio HDFS donde se almacenan los datos, no se crean subdirectorios como en partitioning, sino ficheros en el mismo directorio.
 - `/user/hive/warehouse/orders_bucketed/000000_0`
 - `/user/hive/warehouse/orders_bucketed/000001_0`
 - ... (files 000002_0 through 000017_0 omitted for brevity)
 - `/user/hive/warehouse/orders_bucketed/000018_0`
 - `/user/hive/warehouse/orders_bucketed/000019_0`
- Hay que habilitar la propiedad: `set.hive.enforce.bucketing` a true
 - Alinea el número de reducers al número de buckets en la definición de la tabla

```
SET hive.enforce.bucketing=true;
INSERT OVERWRITE TABLE orders_bucketed
SELECT * FROM orders;
```

Hive:Optimización: Bucketing

- Un **ejemplo** del uso de buckets es el siguiente:
- Seleccionar uno de cada 10 registros de nuestra tabla (10%) (es decir 2 de cada 20)

```
SELECT * FROM orders_bucketed  
TABLESAMPLE (BUCKET 1 OUT OF 10 ON order_id);
```

- Si por ejemplo ejecutamos “*BUCKET 1,3 OUT OF 10 ON order_id*”, recuperaremos el **20% de los datos**, y los buckets de devolverá la query serán:
 - 1,3, 11, 13
- Si obviamos la clausula TABLESAMPLE, haremos un escaneo de toda la tabla.

Hive:Optimización: Indexado

- **Hive** también **soporta el indexado**
- Es parecido al de las RDBMS pero mucho más limitado
- Mejora el performance de algunas **queries** a costa de consumir más espacio en disco y CPU
- Su sintaxis es la siguiente:

```
CREATE INDEX idx_orders_cust_id  
ON TABLE orders(cust_id)  
AS 'handler_class'  
WITH DEFERRED REBUILD;
```

Se crea el índice vacío, aunque la tabla tenga datos

- El handler class es una clase java de este tipo:

```
org.apache.hadoop.hive.ql.index.compact.CompactIndexHandler
```

Hive:Optimización: Indexado

- Otras operaciones

Create/build, show, y drop index:

```
CREATE INDEX table01_index ON TABLE table01 (column2) AS 'COMPACT';  
SHOW INDEX ON table01;  
DROP INDEX table01_index ON table01;
```

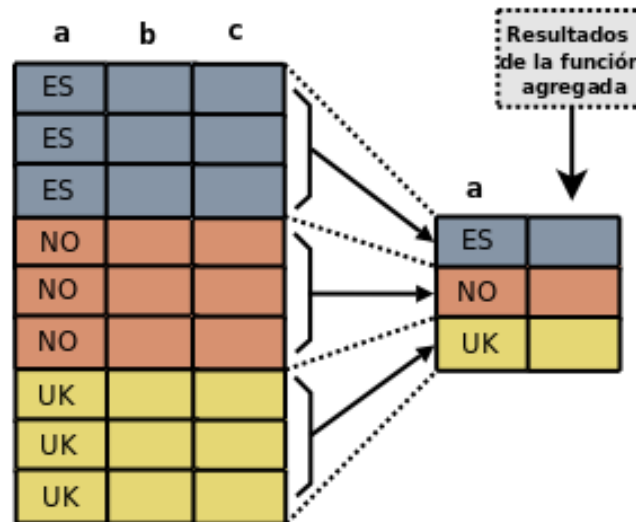
Create, después, build, show formatted (con column names), y drop index:

```
CREATE INDEX table02_index ON TABLE table02 (column3) AS 'COMPACT' WITH DEFERRED REBUILD;  
ALTER INDEX table02_index ON table02 REBUILD; (en caso de modificación. Proceso largo)  
SHOW FORMATTED INDEX ON table02;  
DROP INDEX table02_index ON table02;
```

Hive:Ventanas

Funciones de Ventana

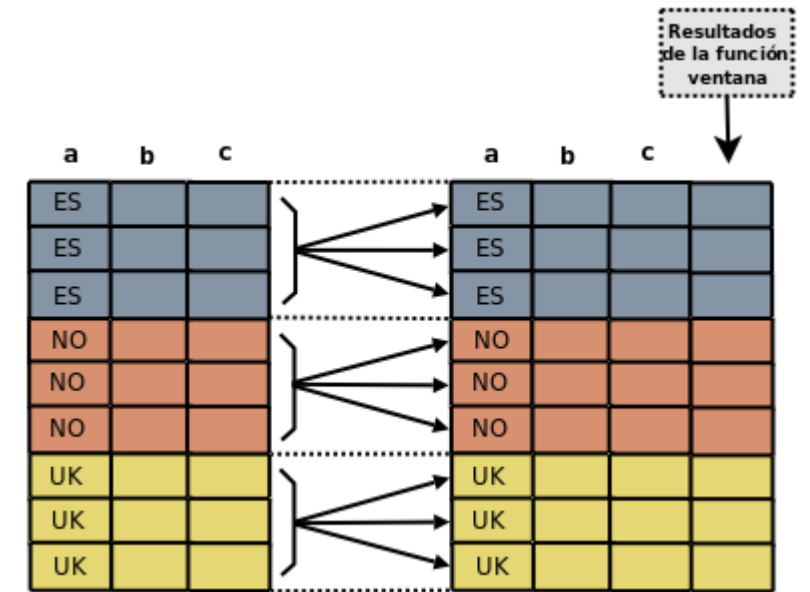
- Se utilizan para realizar funciones de agregación a una partición o subconjunto de filas. Ofrecen muchas más posibilidades que las funciones de agregación típicas
- En funciones agregadas normales utilizamos típicamente el **GROUP BY**



Hive:Ventanas

Funciones de Ventana

- En las funciones de ventana, tenemos más posibilidades
 - Se define usando la cláusula **OVER** después de la función
 - Esta cláusula define la partición o subconjunto que forma la ventana
 - Devuelve un valor por cada fila
 - Está formada por una partición y un marco
 - Una partición se define con la cláusula **PARTITION BY**
 - Si no la usamos, todas las filas se consideran dentro de la misma ventana
 - Un marco se define con la cláusula **ORDER BY**
 - Este marco permite definir el orden de las filas de la ventana



Hive: Ventanas

Funciones de Ventana principales

- **LEAD**
 - Se usa para acceder a filas posteriores a la actual
- **LAG**
 - Se usa para acceder a filas anteriores a la actual
- Ambas necesitan las cláusulas **ORDER BY** y **PARTITION BY**
- Sintaxis

```
LEAD(Column_Name, Offset, Default_Value) OVER (ORDER BY Col1, Col2, ...)  
LAG (Column_Name, Offset, Default_Value) OVER (ORDER BY Col1, Col2, ...)
```

 - **Column_Name:** es la columna sobre la que queremos hacer el LEAD o LAG
 - **Offset:** número de filas sobre las que hacer LEAD o LAG. Si no lo especificamos, por defecto cogerá 1
 - **Default_Value:** valor que tenemos que devolver cuando el número de filas sobre las que hacer LEAD o LAG sobrepasan la primer o la última en la tabla o partición. Si no se especifica valor, se devuelve NULL

Hive: Ventanas

Funciones de Ventana principales

- **Ejemplos** (Offset =1, Default_Value=NULL, por defecto)

```
SELECT Name, Gender, Salary,  
LEAD(Salary) OVER (ORDER BY Salary) AS Lead  
FROM Employees
```

	Name	Gender	Salary	Lead
1	Mark	Male	1000	2000
2	John	Male	2000	3000
3	Pam	Female	3000	4000
4	Sara	Female	4000	5000
5	Todd	Male	5000	6000
6	Mary	Female	6000	7000
7	Ben	Male	7000	8000
8	Jodi	Female	8000	9000
9	Tom	Male	9000	9500
10	Ron	Male	9500	NULL

Hive: Ventanas

Funciones de Ventana principales

- Ejemplos:

```
SELECT Name, Gender, Salary,  
LEAD(Salary, 2, -1) OVER (ORDER BY Salary) AS Lead  
FROM Employees
```

	Name	Gender	Salary	Lead
1	Mark	Male	1000	3000
2	John	Male	2000	4000
3	Pam	Female	3000	5000
4	Sara	Female	4000	6000
5	Todd	Male	5000	7000
6	Mary	Female	6000	8000
7	Ben	Male	7000	9000
8	Jodi	Female	8000	9500
9	Tom	Male	9000	-1
10	Ron	Male	9500	-1

```
SELECT Name, Gender, Salary,  
LEAD(Salary, 2, -1) OVER (ORDER BY Salary) AS Lead,  
LAG(Salary, 1, -1) OVER (ORDER BY Salary) AS Lag  
FROM Employees
```

	Name	Gender	Salary	Lead	Lag
1	Mark	Male	1000	3000	-1
2	John	Male	2000	4000	1000
3	Pam	Female	3000	5000	2000
4	Sara	Female	4000	6000	3000
5	Todd	Male	5000	7000	4000
6	Mary	Female	6000	8000	5000
7	Ben	Male	7000	9000	6000
8	Jodi	Female	8000	9500	7000
9	Tom	Male	9000	-1	8000
10	Ron	Male	9500	-1	9000

Hive:Ventanas

Funciones de Ventana principales

- Ejemplos:

```
SELECT Name, Gender, Salary,  
LEAD(Salary, 2, -1) OVER (PARTITION BY Gender ORDER BY Salary) AS Lead,  
LAG(Salary, 1, -1) OVER (PARTITION BY Gender ORDER BY Salary) AS Lag  
FROM Employees
```

	Name	Gender	Salary	Lead	Lag
1	Pam	Female	3000	6000	-1
2	Sara	Female	4000	8000	3000
3	Mary	Female	6000	-1	4000
4	Jodi	Female	8000	-1	6000
5	Mark	Male	1000	5000	-1
6	John	Male	2000	7000	1000
7	Todd	Male	5000	9000	2000
8	Ben	Male	7000	9500	5000
9	Tom	Male	9000	-1	7000
10	Ron	Male	9500	-1	9000

Hive:Ventanas

Funciones de Ventana principales

- Cláusula OVER
 - Permite ejecutar funciones agregadas sobre una partición o subconjunto de filas
 - Las principales funciones de agregación son: COUNT, SUM, MAX, MIN, AVG
 - También soporta la cláusula PARTITION BY
 - Permite especificar el rango de filas de la ventana sobre los que ejecutar la función de agregación. Las especificaciones por defecto son RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW

- **Ejemplos**

```
SELECT a, COUNT(b) OVER (PARTITION BY c)
FROM T;
```

```
SELECT a, COUNT(b) OVER (PARTITION BY c, d)
FROM T;
```

```
SELECT a, SUM(b) OVER (PARTITION BY c ORDER BY d)
FROM T;
```

```
SELECT a, AVG(b) OVER (PARTITION BY c ORDER BY d ROWS BETWEEN 3 PRECEDING AND CURRENT ROW)
FROM T;
```

Hive:Preguntas



1. Cita algunas razones por las que no reemplazarías una RDBM por Hive

Porque Hive no es un gestor de bases de datos relacionales, esto se debe a que se encuentra distribuidos por lo que hay tarda mucho más que un gestor de bases de datos tradicional. Hive se utiliza cuando el volumen de datos es demasiado elevado además si trabajamos con pocos datos perdemos cualquier ventaja que pueda aportar la programación dirigida

2. Cuáles son los beneficios de Hive y Hadoop sobre DWH tradicionales

Escalabilidad, tolerancia a fallos y portabilidad

3. Qué datos almacena el metastore de hive

El metastore es el repositorio central de Apache Hive metadata. Almacena los metadatos de las tablas de Hive como su esquema o su localización y las particiones de las bases de datos relacionales.

4. Cuando hacemos una consulta en hive sobre una tabla, dónde reside físicamente esa tabla

Una tabla puede tener una o varias particiones por lo que puede estar almacenada en distintos nodos. Esta información se encuentra en el metastore de hive

5. Qué comando se usa para cambiar el foco a otra tabla en hive

Load data inpath '/data/empnew.csv' into table emp

6. Cuál es el comando usado para combiar el resultado de varias queries en un solo resultado

UNION ALL

Hive:Preguntas

7.Cuál es el directorio por defecto del warehouse de hive

/user/warehouse/

8. Dónde se almacenan las tablas particionadas en Hive

En los distintos nodos del cluster

9. Cual es la diferencia entre el tipo de datos SequenceFile y Parquet

Debido a la complejidad de la lectura de archivos secuenciales, normalmente solo se utilizan para los datos en vuelo como los datos intermedios que se generan por MapReduce mientras que son otro tipo de datos columnares que ha creado hadoop.

10.Cuál es la diferencia entre Arrays y Maps

Los arrays son colecciones ordenadas de elementos mientras que los maps son desordenados y se basan en parejas clave-valor.

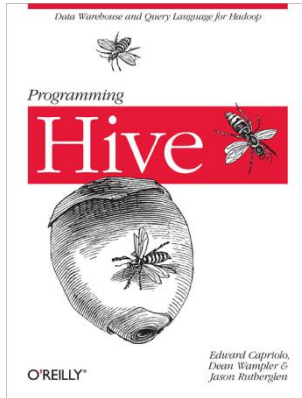
11.Cuál es la query más rápida en Hive

Las que se ejecutan localmente sobre los nodos y no necesitan de un shuffle

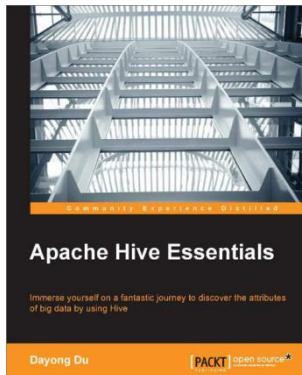
Hive: Ejercicios



Bibliografía



Edward Capriolo, Dean Wampler, and Jason Rutherglen.
Programming Hive. O'Reilly Media, Inc., 2012



Dayong Du. *Apache Hive Essentials*.
Packt Publishing Ltd, 2015