Apache Hive Warehouse and HQL

Analytics Vidhya

What Is Apache Hive?

- **Hive is data warehouse infrastructure for distributed systems like Apache Hadoop**

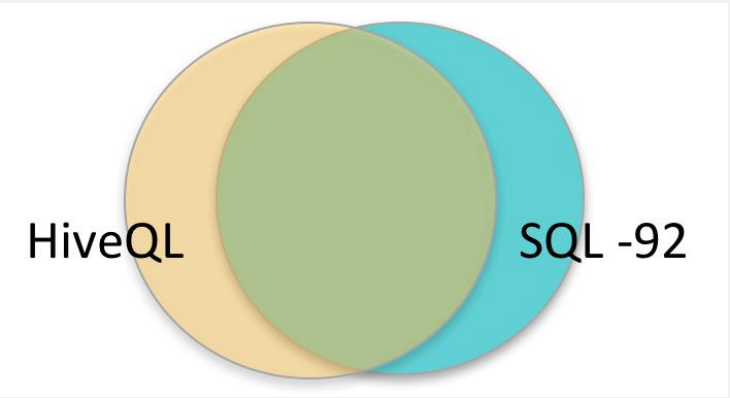
  - Alternative to writing low-level MapReduce code
  - Uses a SQL-like language called HiveQL
  - Generates jobs that run on the distributed system
  - Originally developed by Facebook
  - Now an open source Apache project

Analytics Vidhya

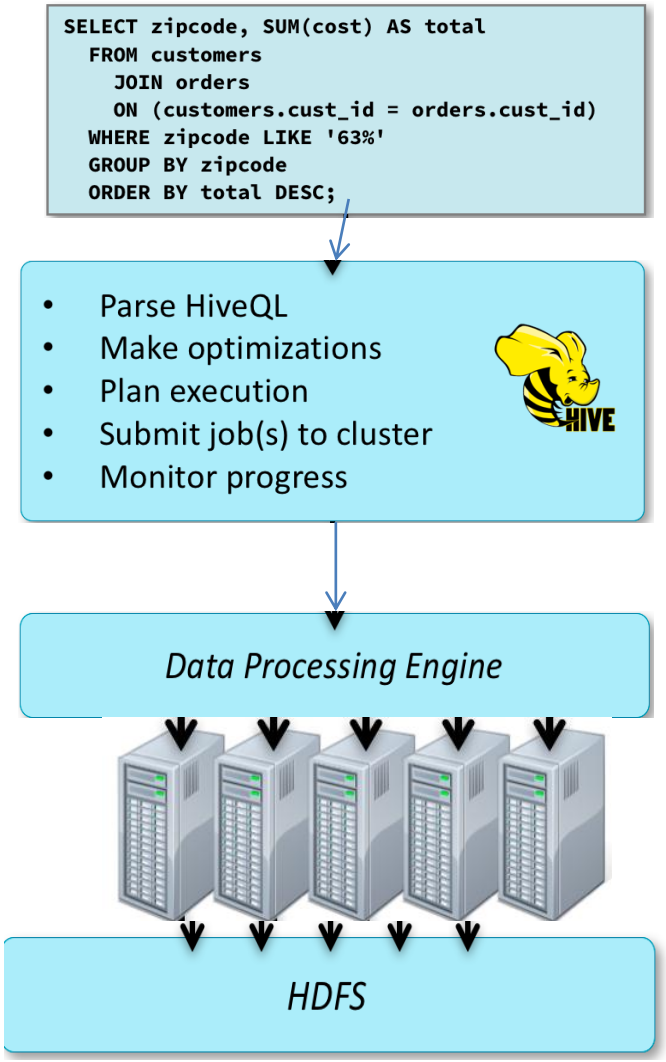
HiveQL

- **HiveQL implements a subset of SQL-92**
  - Plus a few extensions found in MySQL and Oracle SQL dialects

```
SELECT zipcode, SUM(cost) AS total
FROM customers
JOIN orders
ON (customers.cust_id = orders.cust_id)
WHERE zipcode LIKE '63%'
GROUP BY zipcode
ORDER BY total DESC;
```



Hive Overview



- **Hive turns HiveQL queries into data processing jobs**
- **Then it submits those jobs to the data processing engine (MapReduce or Spark) to execute on the cluster**

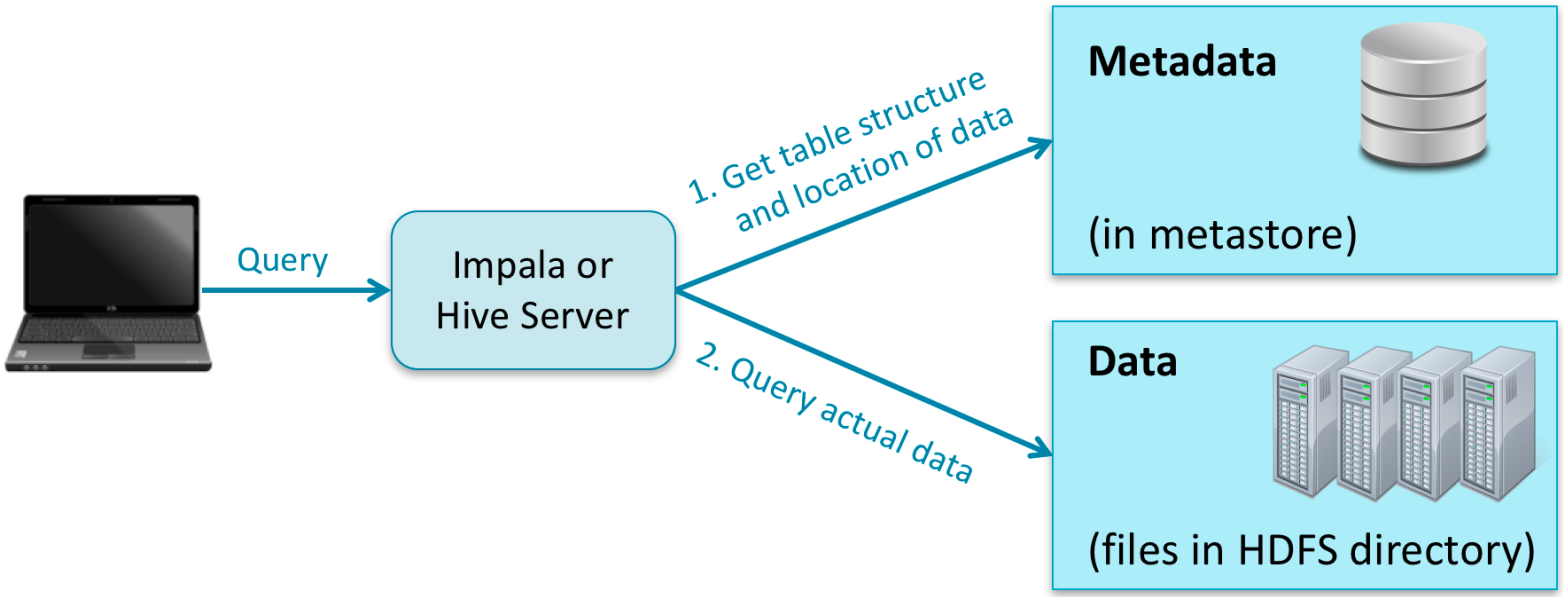
Analytics Vidhya

How Hive and Impala Query Data (1)

- **Queries operate on tables, just as queries do in an RDBMS**
- **A table has two components:**
  - Metadata
    - Specifies structure and location of the data
    - Defined when table is created
    - Stored in the *metastore*, which is contained in an RDBMS
  - Data
    - Typically in an HDFS directory with one or more files
    - Default path: /user/hive/warehouse/*tablename*
    - Can be in any of several formats for storage and retrieval

How Hive and Impala Query Data (2)

- **Hive and Impala use the metastore to determine data format and location**
- **The query itself operates on data stored in a filesystem (typically HDFS)**



Analytics Vidhya

Tables

- **Data for Hive tables is stored on the filesystem (such as HDFS)**
  - Each table maps to a single directory
- **A table's directory may contain multiple files**
  - Typically delimited text files, but many formats are supported
  - Subdirectories are not allowed
  - Exception: partitioned tables
- **The metastore gives context to this data**
  - Helps map raw data in filesystem to named columns of specific types

Databases

- **Each table belongs to a specific database**
- **Early versions of Hive supported only a single database**
  - It placed all tables in the same database (named default)
  - This is still the default behavior
- **Hive support multiple databases**
  - Helpful for organization and authorization

Exploring Databases and Tables (1)

- **Use** SHOW DATABASES **to list the databases in the metastore**

```
> SHOW DATABASES;
```

- **Use** DESCRIBE DATABASE **to display metadata about a database**

```
> DESCRIBE DATABASE ;
```

Analytics Vidhya

Exploring Databases and Tables (2)

- **Hive and Impala both connect to the** default **database by default**
- **Switch between databases with the** USE **command**

```
> SELECT * FROM customers;
> USE sales;
> SELECT * FROM customers;
```

Analytics Vidhya

Exploring Databases and Tables (3)

- **Use** SHOW TABLES **to list the tables in a database**

```
> USE accounting;
> SHOW TABLES;
+-----------+
| tab_name |
+-----------+
| invoices |
| taxes |
+-----------+
```

Analytics Vidhya

Exploring Databases and Tables (4)

- **The** DESCRIBE **command displays basic structure for a table**
- **Use the** DESCRIBE FORMATTED **command for detailed information**
  - Input and output formats, file locations, and other information

```
> DESCRIBE orders;
+------------+-----------+---------+
| name | type | comment |
+------------+-----------+---------+
| order_id | int                  | |
| cust_id | int                   | |
| order_date | timestamp     | |
+------------+-----------+---------+
```

Analytics Vidhya

An Introduction to HiveQL

- **HiveQL is Hive's query language**
  - Based on a subset of SQL-92, plus Hive-specific extensions
- **Some differences compared to standard SQL**
  - Some features are not supported
  - Others are only partially implemented

Syntax Basics

- ***Keywords* are words that have a particular meaning**
  - Examples: **SELECT, FROM, WHERE, AS**
  - Not case-sensitive, but often capitalized by convention
  - Are *reserved* for their specific use
- **Statements are terminated by a semicolon**
  - May span multiple lines
- **Comments begin with --*(double hyphen)***

```
SELECT cust_id, fname, lname
FROM customers
WHERE zipcode='60601'; -- downtown Chicago
```

Analytics Vidhya

Identifiers

- ***Identifiers* are words used to identify a column, table, or database**
- ***Quoted identifiers* are identifiers enclosed in backquotes (such as** `column`**)**
  - Allows reserved words such as keywords to be used as identifiers
  - Sometimes considered best practice for saved queries (such as in scripts)
  - Protects query from future changes in the list of reserved words
  - Not used in this course for convenience and readability
  - Example:

```
SELECT `select` FROM `from`;
```

Selecting Data from Tables

- **The** SELECT **statement retrieves data from tables**
  - Can specify an ordered list of individual columns

```
SELECT cust_id, fname, lname FROM customers;
```

  - An asterisk matches all columns in the table

```
SELECT * FROM customers;
```

- **Use** DISTINCT **to remove duplicates**

```
SELECT DISTINCT zipcode FROM customers;
```

Analytics Vidhya

Sorting Query Results

- **The** ORDER BY **clause sorts the result set**
  - Default order is ascending (same as using ASC keyword)
  - Specify DESC keyword to sort in descending order
  - Hive requires the field(s) you ORDER BY to be included in the SELECT

SELECT brand, name, price FROM products ORDER BY price DESC;

To sort by an expression, put it in the SELECT list and use an alias

SELECT brand, name, price - cost AS profit FROM products ORDER BY profit;

Analytics Vidhya

Limiting Query Results

- **The** LIMIT **clause sets the maximum number of rows returned**

```
SELECT brand, name, price FROM products LIMIT 10;
```

- **Caution: no guarantee regarding *which* 10 results are returned**
  - Use ORDER BY with LIMIT for *top-N* queries

```
SELECT brand, name, price FROM products ORDER BY price DESC
LIMIT 10;
```

Using a WHERE Clause to Filter Results

- WHERE **clauses restrict rows to those matching specified criteria**
  – String comparisons are case-sensitive

```
SELECT * FROM orders WHERE order_id=1287;
SELECT * FROM customers WHERE state
IN ('CA', 'OR', 'WA', 'NV', 'AZ');
```

- **You can combine expressions using** AND **or** OR

```
SELECT * FROM customers
WHERE fname LIKE 'Ann%'
AND (city='Seattle' OR city='Portland');
```

Table Aliases

- **Table aliases can help simplify complex queries**
  - Using AS to specify table aliases is also supported

```
SELECT o.order_date, c.fname, c.lname
FROM customers c JOIN orders AS o
ON (c.cust_id = o.cust_id)
WHERE c.zipcode='94306';
```

Subqueries in the FROM Clause

- **Hive support subqueries in the** FROM **clause**
  - The subquery must be named (high_profits in this example)

```
SELECT prod_id, brand, name
FROM (SELECT * FROM products WHERE (price - cost) / price > 0.65
ORDER BY price DESC LIMIT 10) high_profits
WHERE price > 1000 ORDER BY brand, name;
```

Analytics Vidhya

Data Types

- **Each column has an associated data type**
- **Hive support more than a dozen types**
  - Most are similar to ones found in relational databases
  - Hive and Impala also support certain *complex types*
- **Use the** DESCRIBE **command to see data types for each column in a table**

Integer Types

- **Integer types are appropriate for whole numbers**
- — Both positive and negative values are allowed

TINYINT **Range: -128 to 127** e.g 17

SMALLINT **Range: -32,768 to 32,767** e.g 5842

INT Range: **-2,147,483,648 to 2,147,483,647 84** e.g127213

BIGINT Range: **≈ -9.2 quintillion to ≈ 9.2 quintillion** e.g 632197432180964

Analytics Vidhya

Decimal Types

- **Decimal types are appropriate for numbers that can include fractional parts**
  - Both positive and negative values allowed
  - Use DECIMAL when exact values are required!
  - Blue digits in the examples below are not accurate
- FLOAT Decimals 3.141592**7410125732**
- DOUBLE More precise decimals 3.14159265358979**31**
- DECIMAL(p,s)* Exact precision 3.141592653589793**2** using DECIMAL(17,16)

Analytics Vidhya

Character Types

- **Character types are used to represent alphanumeric text values**
- STRING Character sequence

  e.g Impala rules!
- CHAR(*n*) Fixed-length character sequence

  e.g Impala rules!␣␣␣ using CHAR(16)
- VARCHAR(*n*) Variable length character sequence (maximum length *n*)

  e.g Impala rule using VARCHAR(10)

Other Simple Types

- **There are a few other data types**

- BOOLEAN True or false **e.g true**
- TIMESTAMP Instant in time **e.g 2016-06-14 16:51:05**
- BINARY (Hive-only) Raw bytes

Data Type Conversion

- **Hive auto-converts a** STRING **column used in numeric context**

> SELECT zipcode FROM customers LIMIT 1;

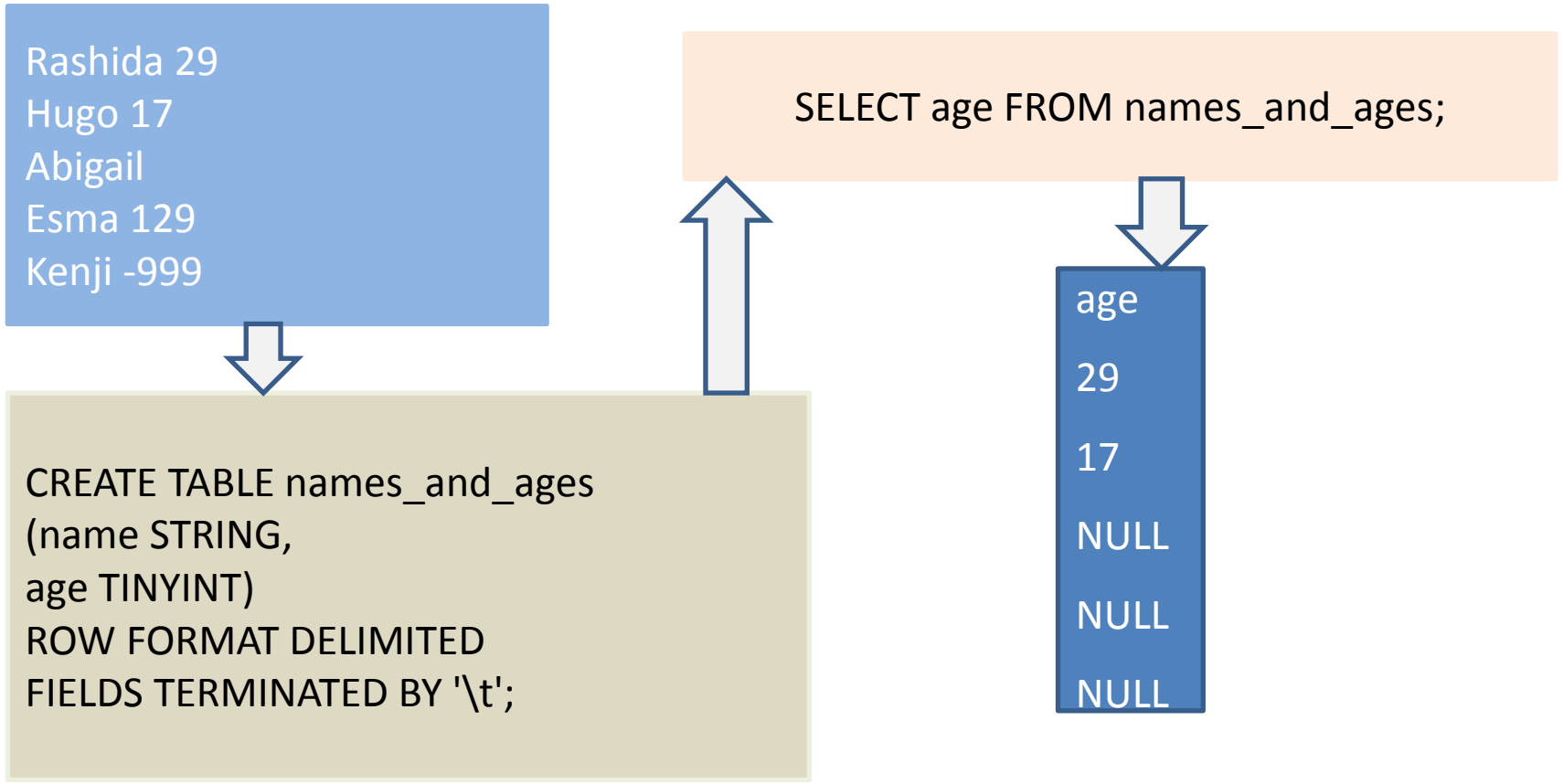
60601

> SELECT zipcode + 1.5 FROM customers LIMIT 1;

60602.5

Handling of Out-of-Range Values

- Hive returns NULL



Analytics Vidhya

Interacting with Hive

- **Hive and Impala offer many interfaces for running queries**
  - Hue web UI
  - Hive Query Editor
  - Metastore Manager
  - Command-line shell
  - Hive: Beeline
  - Ambari web UI
  - ODBC / JDBC

Starting Beeline (Hive's Shell)

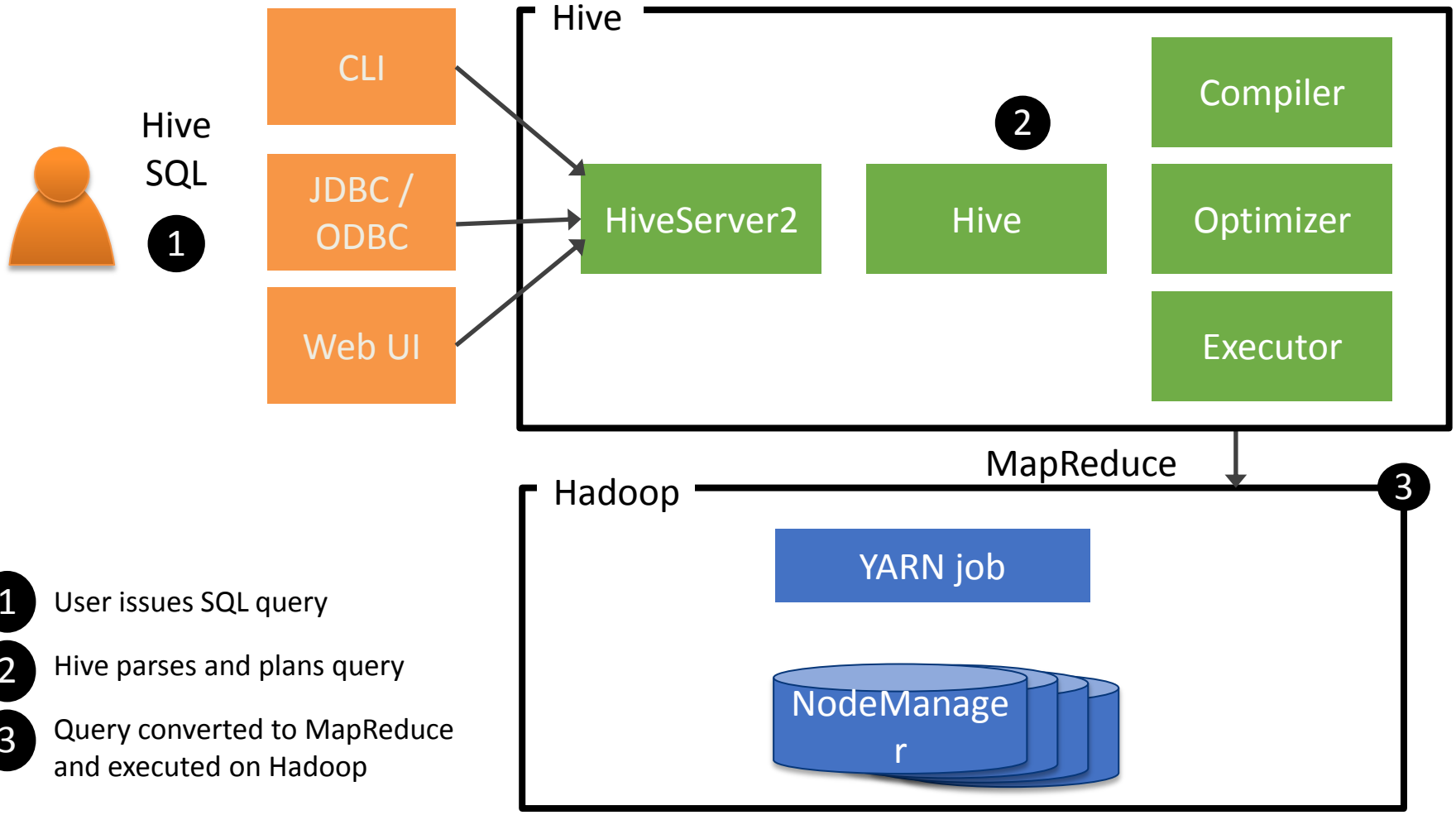
- **You can execute HiveQL statements in Beeline**
  - Interactive shell based on the SQLLine utility
  - Similar to the shell in MySQL
- **Start Beeline by specifying a URL for a Hive server**
  - Plus username and password, if required

```
$ beeline -u jdbc:hive2://host:10000 \
-n username -p password
Connecting to jdbc:hive2://host:10000
Connected to: Apache Hive (version 2.1.1-cdh6.0.0)
Beeline version 2.1.1-cdh6.0.0 by Apache Hive
0: jdbc:hive2://host:10000>
```

Starting Hive's Shell

- Hive CLI
  - Traditional Hive client that connects to a HiveServer instance
  - **$ hive -h hostname**
    hive>

Hive Architecture



1. User issues SQL query
2. Hive parses and plans query
3. Query converted to MapReduce and executed on Hadoop

Analytics Vidhya

Creating a Table

- **Use** LIKE **to create a new table using the definition of an existing table**

  CREATE TABLE jobs_archived LIKE jobs;

- **Column definitions are derived from the existing table's definition**
  - New table will contain no data

Creating Tables Based on Existing Schema

- **Basic syntax for creating a table:**

  CREATE TABLE *dbname.tablename* (*colname DATATYPE, …*)

  ROW FORMAT DELIMITED

  FIELDS TERMINATED BY *char*

  STORED AS {TEXTFILE|SEQUENCEFILE|…};

- **Creates an empty subdirectory in the database's warehouse directory in HDFS**

  – Default database:

  /user/hive/warehouse/*tablename*

  – Named database:

  /user/hive/warehouse/*dbname*.db/*tablename*

- **Creates the metadata for the table in the metastore**

Controlling Table Data Location

- **By default, table data is stored in the warehouse directory (within HDFS)**
- **This is not always ideal**
  - Data might be part of a bigger workflow
- **Use** LOCATION **to specify the directory where table data resides**

```
CREATE TABLE jobs ( id INT,
    title STRING,
    salary INT,
    posted TIMESTAMP)
    ROW FORMAT DELIMITED
    FIELDS TERMINATED BY ','
    LOCATION '/analyst/dualcore/jobs';
```

- **For locations outside HDFS, a fully qualified path will be needed**
  - **LOCATION 's3a://path.to.bucket/jobs'**
  - **LOCATION 'adl://path.to.bucket/jobs'**

Analytics Vidhya

Defining a Hive-Managed Table

- **Dropping a table removes its data in HDFS**

```
CREATE TABLE customer (
                customerID INT,
                firstName STRING,
                lastName STRING,
                birthday TIMESTAMP,
  ) ROW FORMAT DELIMITED
    FIELDS TERMINATED BY ',';
```

Analytics Vidhya

Externally Managed Tables

- **Using** EXTERNAL **when creating the table avoids this behavior**
  - Dropping an *external* (*unmanaged*) table removes only its *metadata*

```
CREATE EXTERNAL TABLE customer (
                customerID INT,
                firstName STRING,
                lastName STRING,
                birthday TIMESTAMP,
 ) ROW FORMAT DELIMITED
   FIELDS TERMINATED BY ',';
```

Data Validation

- **Hive are generally *schema-on-read***
  - Most common method to load data is simply to move files into place
  - Loading data into tables is therefore very fast
  - Errors in file format will be discovered when queries are performed
  - Exception: You can use an INSERT … VALUES statement which validates on execution
  - Not suitable for loading large amounts of data
- **Missing or invalid data typically will be represented as** NULL

Loading Data in HDFS

- **To load data, add files to the table's directory in HDFS**
  - Can be done directly using the hdfs dfs commands
  - This example loads data from the local drive to the sales table

```
$ hdfs dfs -put sales.txt /user/hive/warehouse/sales/
```

- – This example loads data from the user's directory in HDFS to sales

```
$ hdfs dfs -mv sales.txt /user/hive/warehouse/sales/
```

Loading Data in HDFS

- **Alternatively, use the** LOAD DATA INPATH **command**
  - Done from within Hive or Impala
  - This changes the file location within HDFS, just like the second command above
  - Source can be either a file or directory

```
LOAD DATA INPATH '/incoming/etl/sales.txt' INTO TABLE sales;
```

Overwriting Data from Files

- **Add the** OVERWRITE **keyword to delete all records before import**

  – Removes all files within the table's directory
  – Then moves the new files into that directory

  LOAD DATA INPATH '/incoming/etl/sales.txt' OVERWRITE INTO TABLE sales;

Removing a Database

- **Removing a database has similar syntax to creating it**

```
DROP DATABASE dualcore;
DROP DATABASE IF EXISTS dualcore;
```

- **These commands will fail if the database contains tables**
  - Add the CASCADE keyword to force removal
  - Supported in all production versions of Hive

```
DROP DATABASE dualcore CASCADE;
```

[CAUTION: This command might remove data in HDFS!]

Removing a Table

- **Table removal syntax is similar to database removal**

  DROP TABLE IF EXISTS customers;

  DROP TABLE customers;

- **Managed (internal) tables**
  - **Metadata is removed**
  - **Data in HDFS is removed**
  - ***Caution: No rollback or undo feature!***

- **Unmanaged (external) tables**
  - Metadata is removed
  - Data in HDFS is *not* removed

Renaming and Moving Tables

- **Use** ALTER TABLE **to modify a table or its columns**

- **Rename an existing table**
  - Changes metadata
  - If table is managed, renames directory in HDFS

```
ALTER TABLE customers RENAME TO clients;
```

- **Move an existing table to a different database**
  - Changes metadata
  - If table is managed, moves directory in HDFS

```
ALTER TABLE default.clients
RENAME TO dualcore.clients;
```

Renaming and Modifying Columns

- **Rename a column by specifying its old and new names**
  - Type must be specified even if it is unchanged

  ALTER TABLE clients CHANGE fname first_name STRING;

  Old Name New Name Type
- **You can also modify a column's type**
  - The old and new column names will be the same
  - Does not change the data in HDFS
  - You must ensure the data in HDFS conforms to the new type

  ALTER TABLE clients CHANGE salary salary BIGINT;

  Old Name New Name Type
- **You can also change name and type using a single command**

  ALTER TABLE suppliers CHANGE supp_id supplier_id BIGINT;

Reordering Columns in Hive

- **Use** AFTER **or** FIRST **to reorder columns in Hive**
  - Does not change the data in HDFS
  - You must ensure the data in HDFS matches the new order

```
ALTER TABLE jobs CHANGE salary salary INT AFTER id;
ALTER TABLE jobs CHANGE salary salary INT FIRST;
```

Analytics Vidhya

Adding Columns

- **Add new columns to a table**
  - Appended after any existing columns
  - Does not change the data in HDFS
  - If column does not exist in data in HDFS, then values will be NULL

```
ALTER TABLE jobs ADD COLUMNS (city STRING, bonus INT);
```

The Warehouse Directory

- **By default, Hive store data in the HDFS directory**

  /user/hive/warehouse

  – Other storage options include HBase, Apache Kudu, and S3
- **In HDFS, each table's data is stored in a subdirectory named after the table.**

Using Operators in HiveQL

- **As with other SQL dialects, use operators as part of an expression**

  For example:

  – In SELECT list

  SELECT name, price, cost, price – cost FROM products;

  – In WHERE clause

  SELECT name, price, cost FROM products WHERE cost >= price;

Arithmetic Operators in HiveQL

- **Basic four:** +, -, *, /
  - 84 / 4 = 21
  - 84 - 4 = 80
- **Modulo:** %
  - This is the remainder from integer division
  - Example: 25 % 7 = 4
- **Unary negative:** -
  - Unary because it takes only one operand
  - -(3 - 5) = -(-2) = 2
- **No operator for exponentiation**
  - Use pow() function instead
  - pow(5,3) = 5 * 5 * 5 = 125

Comparison Operators in HiveQL

- **Equality and Inequality**
  - =
  - != or <>
  - <, >, <=, >=
- IN
  - 5 IN (2, 3, 5, 7): true
  - 6 IN (2, 3, 5, 7): false
- BETWEEN
  - 5 BETWEEN 2 AND 7: true
  - 9 BETWEEN 2 AND 7: false
  - 7 BETWEEN 2 and 7: true
- LIKE
  - 'name@example.com' LIKE '%.com': true
  - 'Mickey' LIKE 'M*ey': false

Comparisons and NULL Values

- **Typical operators return** NULL **when at least one operand is** NULL
  - 5 = NULL → NULL
  - 5 != NULL → NULL
  - 5 < NULL → NULL
  - NULL = NULL → NULL
  - NULL != NULL → NULL
- **Alternative operators are** IS (NOT) DISTINCT FROM **and** <=>
  - IS DISTINCT FROM
  - Same as != for non-NULL values
  - 5 IS DISTINCT FROM NULL → true
  - NULL IS DISTINCT FROM NULL → false
  - IS NOT DISTINCT FROM or <=> (NULL-safe equality)
  - Same as = for non-NULL values
  - 5 <=> NULL → false
  - NULL <=> NULL → true

Logical and Null Operators in HiveQL

- **Logical operators**
  - Binary: AND, OR
  - Unary: NOT
- **Null operators**
  - IS NULL
  - IS NOT NULL

Column Aliases

- **Column aliases can replace expressions in result headers**
  - Otherwise Hive or Impala will generate column names
  - The AS keyword is optional but recommended

```
SELECT name, price, cost, price - cost AS profit FROM products;
```

Built-In Functions

- **Hive offer dozens of built-in functions**
  - Many are identical to those found in other SQL dialects
  - Others are Hive- specific

- **Example function invocation**
  - Function names are not case-sensitive

```
SELECT concat(fname, ' ', lname) AS fullname FROM customers;
```

Analytics Vidhya

Getting Information about Functions

- **To see information about a function**

> DESCRIBE FUNCTION upper;

upper(str) - Returns str with all characters changed to uppercase

Analytics Vidhya

Trying Built-In Functions

- **To test a built-in function, use a** SELECT **statement with no** FROM **clause**

```
> SELECT abs(-459.67);
459.67

> SELECT upper('Fahrenheit');
FAHRENHEIT
```

Scalar Functions

- ***Scalar functions*** **operate independently on the values in each row**
- **The number of arguments depends on the function**
  - rand()
  - abs(-283)
  - pow(2, 5)
- **Arguments can be column references, literal values, or expressions**
  - Column references: year(order_dt)
  - Literal values: lower('Function')
  - Expressions:
  - year(order_dt + 5)
  - round(price * tax, 2)
- **By convention, scalar functions are usually written in lowercase**

Built-In Mathematical Functions

Function Description	Example Invocation	Input	Output
Round to specified # of decimals	round(total_price, 2)	23.492	23.49
Return nearest integer above	ceil(total_price)	23.492	24
Return nearest integer below	floor(total_price)	23.492	23
Return absolute value	abs(temperature)	-67	67
Return square root	sqrt(area)	64	8
Return a random number	rand()		0.584977

Built-In Date and Time Functions

Function Description	Example Invocation	Input	Output
Return current date and time	current_timestamp()		2020-06-14 16:51:05.0
Convert to UNIX format	unix_timestamp(order_dt)	2016-06-14 16:51:05	1465923065
Convert to string format	from_unixtime(mod_time)	1465923065	2016-06-14 16:51:05
Extract date portion	to_date(order_dt)	2016-06-14 16:51:05	2016-06-14
Extract date portion	year(order_dt)	2020-06-14 16:51:05.0	2020
Return # of days between dates	datediff(ship_dt, order_dt)	2020-06-17, 2020-06-14	3

Built-In String Functions

Function Description	Example Invocation	Input	Output
Convert to uppercase	upper(name)	Bob	BOB
Convert to lowercase	lower(name)	Bob	bob
Remove whitespace at start/end	trim(name)	␣Bob␣	Bob
Remove only whitespace at start	ltrim(name)	␣Bob␣	Bob␣
Remove only whitespace at end	ltrim(name)	␣Bob␣	␣Bob
Extract portion of string	substring(name, 2, 4)	Samuel	Amue
Replace characters in string	translate(name, 'uel', 'my')	Samuel	Sammy

Analytics Vidhya

String Concatenation

Example Invocation	Output
concat('alice', '@example.com')	alice@example.com
concat_ws(' ', 'Bob', 'Smith')	Bob Smith
concat_ws('/', 'Amy', 'Sam', 'Ted')	Amy/Sam/Ted

Other Built-In Functions

Function Description	Example Invocation	Input	Output
Convert to another type	cast(weight AS INT)	3.581	3
Selectively return value	if(price > 1000, 'A', 'B')	1500	A
Selectively return value (multiple cases)	case when price > 1000 then 'A' when price < 100 then 'C' else 'B' end	5	C
parses web addresses (URLs)	parse_url(; http://www.example.com/click.php?A=42&Z=105#r1', 'PROTOCOL')		http

Analytics Vidhya

Aggregate Functions

- ***Aggregate functions*** **combine values from multiple rows**
- **They can work over all rows in a table, returning only one row**

  SELECT AVG(price) FROM products;
- **Or they can work over groups of rows**
  - Group rows based on a column expression
  - GROUP BY *column*

  SELECT brand, AVG(price) FROM products GROUP BY brand;
  - Combine rows in the group
  - Results consist of one row for each group
  - Individual row values are not available
- **By convention, aggregate functions are typically uppercase**

Analytics Vidhya

Example: Record Grouping and Aggregate Functions

- GROUP BY **groups selected data by one or more columns**
  - Columns in SELECT list must be in GROUP BY clause or aggregated

- **Question: How many products of each brand are in the** products **table?**

```
SELECT brand, COUNT(prod_id) AS num FROM products
GROUP BY brand;
```

Analytics Vidhya

Example: Record Grouping and Aggregate Functions

Prod_ID	brand	name	price
1	Dualcore	USB Card Reader	18.39
2	Dualcore	HDMI Cable	11.99
3	Dualcore	VGA Cable	1.99
4	Gigabux	6-cell Battery	40.50
5	Gigabux	8-cell Battery	50.50
6	Gigabux	Wall Charger	20.00
7	Gigabux	Auto Charger	20.00

brand	**Num**
Dualcore	3
Gigabux	4

Analytics Vidhya

Built-In Aggregate Functions

Function Description	Example Invocation
Count all rows	Count(*)
Count all rows where field is not NULL	Count(name)
Count all rows where field is unique and not NULL	COUNT(DISTINCT fname)
Return the largest value	MAX(price)
Return the smallest value	MIN(price)
Add all supplied values and return result	SUM(price)
Return the average of all supplied values	AVG(price)

The HAVING Clause

- **You cannot filter on aggregate functions using** WHERE
- **Use** HAVING **instead**
  - Put the HAVING clause after the GROUP BY clause
  - The aggregate function does *not* have to be in the SELECT list or the GROUP BY clause

- **Question: What is the average profit of products in the** products **table, by brand, for brands with at least 50 products?**

```
SELECT brand, AVG(price-cost) AS avg_profit
FROM products
GROUP BY brand
HAVING COUNT(prod_id) >= 50;
```

Analytics Vidhya

Example: GROUP BY **with** WHERE **and** HAVING

- **Question: How many employees do we have in each state with a salary of less than $20,000?**

```
SELECT state, COUNT(*) AS num FROM employees
WHERE salary < 20000
GROUP BY state;
```

- **Question: Which states have more than 400 employees whose salary is less than $20,000?**

```
SELECT state, COUNT(*) AS num FROM employees
WHERE salary < 20000
GROUP BY state
HAVING COUNT(*) > 400;
```

Table Partitioning

- **By default, all data files for a table are stored in a single directory**
  - *All* files in the directory are read during a query
- **Partitioning subdivides the data**
  - Data is physically divided during loading, based on values from one or more columns
- **Speeds up queries that filter on partition columns**
  - Only the files containing the selected data need to be read
- **Does not prevent you from running queries that span multiple partitions**

Table Partitioning

- Use the **partitioned by** clause to define a partition when creating a table:

```
create table employees (id int, name string, salary double)
partitioned by (dept string);
```

- Subfolders are created based on the partition values:

  /apps/hive/warehouse/employees

  /dept=hr/

  /dept=support/

  /dept=engineering/

  /dept=training/

Loading Data into a Partitioned Table

- **Static partitioning**
  - You manually create new partitions using ADD PARTITION
  - When loading data, you specify which partition to store it in
- **Dynamic partitioning**
  - Hive/Impala automatically creates partitions
  - Inserted data is stored in the correct partitions based on column values

Static Partitioning

- **With static partitioning, you create each partition manually**

```
ALTER TABLE customers_by_state
ADD PARTITION (state='NY');
```

- **Then add data one partition at a time**

```
INSERT OVERWRITE TABLE customers_by_state
PARTITION(state='NY')
SELECT cust_id, fname, lname, address,
city, zipcode FROM customers WHERE state='NY';
```

Dynamic Partitioning

- **When loading data with** INSERT**, use the** PARTITION **clause**
  - The partition column(s) must be included in the PARTITION clause
  - The partition column(s) must be specified *last* in the SELECT list
- **Hive or Impala creates partitions and inserts data based on values of the partition column**
  - The values of the partition column(s) are not included in the files

```
INSERT OVERWRITE TABLE customers_by_state
PARTITION(state)
SELECT cust_id, fname, lname, address,
city, zipcode, state FROM customers;
```

Dynamic Partitioning

- **Hive's default disallows using dynamic partitioning**
  - ─ Intended to prevent users from accidentally creating a huge number of partitions
  - ─ Remove this limitation by changing a configuration property

  SET hive.exec.dynamic.partition.mode=nonstrict;

- **Note: Hive properties set in Beeline are for the current session only.**
  - ─ Your system administrator can configure properties permanently

Dynamic Partitioning

- **Caution: If the partition column has many unique values, many partitions will be created**

  Three Hive configuration properties exist to limit this

  – **hive.exec.max.dynamic.partitions.pernode**

  – Maximum number of dynamic partitions that can be created by any given node involved in a query

  – Default 100

  – **hive.exec.max.dynamic.partitions**

  – Total number of dynamic partitions that can be created by one HiveQL statement

  – Default 1000

  – **hive.exec.max.created.files**

  – Maximum total files (on all nodes) created by a query

  – Default 100000

Viewing, Adding, and Removing Partitions

- **To view the current partitions in a table**

  ```
  SHOW PARTITIONS call_logs;
  ```

- **Use** ALTER TABLE **to add or drop partitions**

  ```
  ALTER TABLE call_logs  ADD PARTITION (call_date='2016-06-05');
  ALTER TABLE call_logs DROP PARTITION (call_date='2016-06-05');
  ```

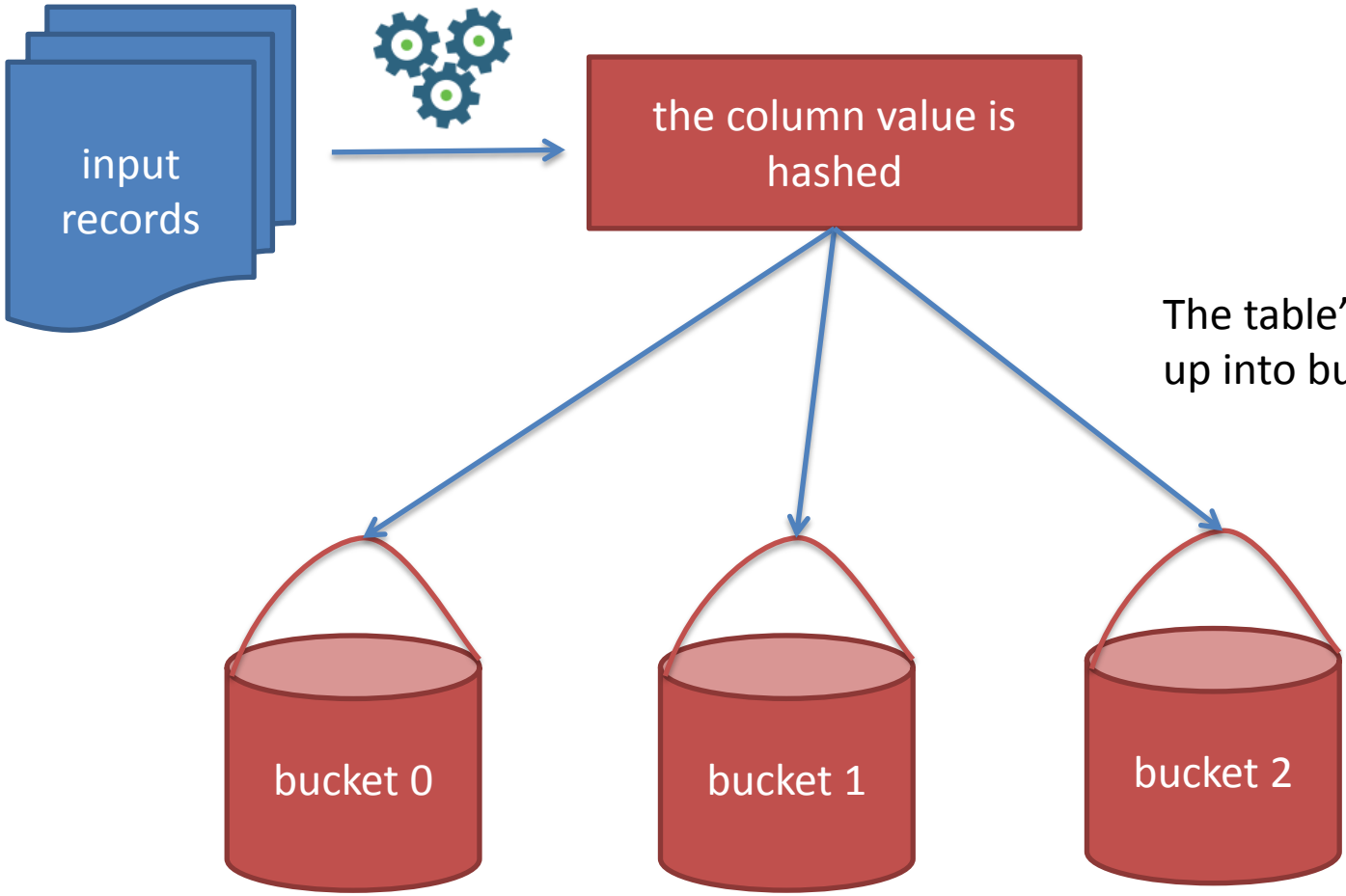
When to Use Partitioning

- **Use partitioning for tables when**
  - Reading the entire dataset takes too long
  - Queries almost always filter on the partition columns
  - There are a reasonable number of different values for partition columns
  - Data generation or ETL process splits data by file or directory names
  - Partition column values are not in the data itself

When *Not* to Use Partitioning

- **Avoid partitioning data into numerous small data files**
  - Partitioning on columns with too many unique values
- **Caution: This can happen easily when using dynamic partitioning!**
  - For example, partitioning customers by first name could produce thousands of partitions

Hive Buckets



The table's data is divided up into buckets

Analytics Vidhya

Storing Results to a File

```
INSERT OVERWRITE DIRECTORY
        '/user/train/ca_or_sd/'
        from names
                select name, state
                where state = 'CA'
                or state = 'SD';
```

```
INSERT OVERWRITE LOCAL DIRECTORY
        '/tmp/myresults/'
        SELECT * FROM bucketnames
        ORDER BY age;
```

Specifying MapReduce Properties

```
SET mapreduce.job.reduces = 12
```

```
hive -f myscript.hive
 -hiveconf mapreduce.job.reduces=12
```

```
SELECT * FROM names
      WHERE age = ${age}
hive -f myscript.hive -hivevar age=33
```

Analytics Vidhya

Combining Query Results with a Union

- UNION ALL **unifies output from multiple** SELECT**s into a single result set**
  - The order and types of columns in each query must match
  - In Hive, the names of columns also must match

```
SELECT cast(cust_id AS string) AS id, fname, lname
FROM customers
WHERE state = 'NY'
UNION ALL
SELECT emp_id AS id, fname, lname
FROM employees
WHERE state = 'NY';
```

- **Without the** ALL **keyword,** UNION **also removes duplicate values**

Analytics Vidhya

Joins

- **Joining disparate datasets is a common operation**
  - Uses a shared column between the datasets
  - Combines rows by matching values in the shared columns
- **Hive and Impala support several types of joins**
  - Inner joins
  - Outer joins (left, right, and full)
  - Cross joins
  - Left semi-joins
- **Join conditions must use equality comparisons when using Hive**
  - Valid: customers.cust_id = orders.cust_id
  - Invalid: customers.cust_id <> orders.cust_id
- **For best performance in Hive, list the largest table last in your query**

Inner Join

- ***Inner joins* exclude records with non-matching key values**

  For example, imagine an inner join of these tables on cust_id

  — What do you think the results would be?

customers table

cust_id	**name**	country
a	Alice	us
b	Bob	ca
c	Carlos	mx
d	Dieter	de

orders table

order_id	cust_id	**total**
1	a	1539
2	c	1871
3	a	6352
4	b	1456
5	z	2137

Inner Join

customers table

cust_id	**name**	country
a	Alice	us
b	Bob	ca
c	Carlos	mx
d	Dieter	de

orders table

order_id	cust_id	**total**
1	a	1539
2	c	1871
3	a	6352
4	b	1456
5	z	2137

```
SELECT c.cust_id, name, total
FROM customers c
JOIN orders o
ON (c.cust_id = o.cust_id);
```

Result

cust_id	**name**	**total**
a	Alice	1539
c	Carlos	1871
a	Alice	6352
b	Bob	1456

Analytics Vidhya

Outer Joins

- ***Outer joins* include records with non-matching key values**
- ***Left* outer joins**
  - Contain all records from the left (first) table
  - Contain records from the right only if they match
- ***Right* outer joins**
  - Contain all records from the right (second) table
  - Contain records from the left only if they match
- ***Full* outer joins include all records from both**

Left Outer Join

customers table

cust_id	name	country
a	Alice	us
b	Bob	ca
c	Carlos	mx
d	Dieter	de

orders table

order_id	cust_id	total
1	a	1539
2	c	1871
3	a	6352
4	b	1456
5	z	2137

```
SELECT c.cust_id, name, total
FROM customers c
LEFT OUTER JOIN orders o
ON (c.cust_id = o.cust_id);
```

Result

cust_id	name	total
a	Alice	1539
c	Carlos	1871
a	Alice	6352
b	Bob	1456
d	Dieter	NULL

Analytics Vidhya

Right Outer Join

customers table

cust_id	name	country
a	Alice	us
b	Bob	ca
c	Carlos	mx
d	Dieter	de

orders table

order_id	cust_id	total
1	a	1539
2	c	1871
3	a	6352
4	b	1456
5	z	2137

```
SELECT c.cust_id, name, total
FROM customers c
RIGHT OUTER JOIN orders o
ON (c.cust_id = o.cust_id);
```

Result

cust_id	name	total
a	Alice	1539
c	Carlos	1871
a	Alice	6352
b	Bob	1456
NULL	NULL	2137

Analytics Vidhya

Full Outer Join

customers table

cust_id	name	country
a	Alice	us
b	Bob	ca
c	Carlos	mx
d	Dieter	de

orders table

order_id	cust_id	total
1	a	1539
2	c	1871
3	a	6352
4	b	1456
5	z	2137

```
SELECT c.cust_id, name, total
FROM customers c
FULL OUTER JOIN orders o
ON (c.cust_id = o.cust_id);
```

Result

cust_id	name	total
a	Alice	1539
c	Carlos	1871
a	Alice	6352
b	Bob	1456
NULL	NULL	2137
d	Dieter	NULL

Using an Outer Join to Find Unmatched Entries

customers table

cust_id	name	country
a	Alice	us
b	Bob	ca
c	Carlos	mx
d	Dieter	de

orders table

order_id	cust_id	total
1	a	1539
2	c	1871
3	a	6352
4	b	1456
5	z	2137

```
SELECT c.cust_id, name, total
FROM customers c
FULL OUTER JOIN orders o
ON (c.cust_id = o.cust_id)
WHERE c.cust_id IS NULL
OR o.total IS NULL;
```

Result

cust_id	name	total
NULL	NULL	2137
d	Dieter	NULL

Analytics Vidhya

Left Semi-Joins

- **A less common type of join is the** LEFT SEMI JOIN
  - It is a special (and efficient) type of inner join
  - It behaves more like a filter than a join
- **Left semi-joins only return records from the table on the left**
  - There must be a match in the table on the right
  - Join conditions and other criteria are specified in the ON clause

```
SELECT c.cust_id
FROM customers c
LEFT SEMI JOIN orders o
ON (c.cust_id = o.cust_id AND o.total > 1500);
```

Hive Join Strategies

Type	Approach	Pros	Cons
Shuffle Join	Join keys are shuffled using MapReduce and joins are performed on the reduce side.	Works regardless of data size or layout.	Most resource-intensive and slowest join type.
Map (Broadcast) Join	Small tables are loaded into memory in all nodes, mapper scans through the large table and joins.	Very fast, single scan through largest table.	All but one table must be small enough to fit in RAM.
Sort-Merge-Bucket Join	Mappers take advantage of co-location of keys to do efficient joins.	Very fast for tables of any size.	Data must be sorted and bucketed ahead of time.

Analytics Vidhya

Shuffle Joins

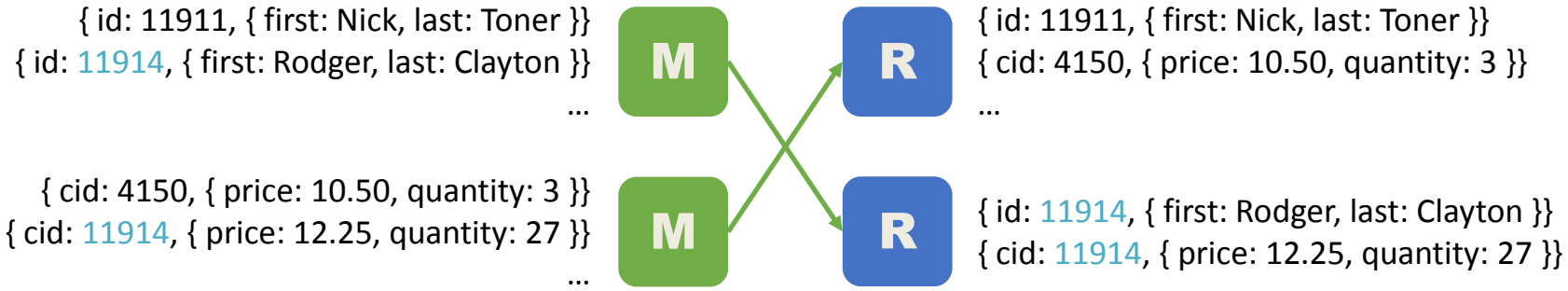
customer

first	last	id
Nick	Toner	11911
Jessie	Simonds	11912
Kasi	Lamers	11913
Rodger	Clayton	11914
Verona	Hollen	11915

order

cid	price	quantity
4150	10.50	3
11914	12.25	27
3491	5.99	5
2934	39.99	22
11914	40.50	10

```
SELECT * FROM customer JOIN order ON customer.id = order.cid;
```



Analytics Vidhya

Map (Broadcast) Joins

customer

first	last	id
Nick	Toner	11911
Jessie	Simonds	11912
Kasi	Lamers	11913
Rodger	Clayton	11914
Verona	Hollen	11915

order

cid	price	quantity
4150	10.50	3
11914	12.25	27
3491	5.99	5
2934	39.99	22
11914	40.50	10

```
SELECT * FROM customer JOIN order ON customer.id = order.cid;
```

{ id: 11914, { first: Rodger, last: Clayton }}
{ cid: 11914, { price: 12.25, quantity: 27 },
cid: 11914, { price: 12.25, quantity: 27 }}

Records are joined during the Map phase.

Analytics Vidhya

Sort-Merge-Bucket Joins

customer

first	last	id
Nick	Toner	11911
Jessie	Simonds	11912
Kasi	Lamers	11913
Rodger	Clayton	11914
Verona	Hollen	11915

order

cid	price	quantity
4150	10.50	3
11914	12.25	27
11914	40.50	10
12337	39.99	22
15912	40.50	10

```
SELECT * FROM customer join order ON customer.id = order.cid;
```

Distribute and sort by the most common join key.

```
CREATE TABLE order (cid int, price float, quantity int)
CLUSTERED BY(cid) INTO 32 BUCKETS;

CREATE TABLE customer (id int, first string, last string)
CLUSTERED BY(id) INTO 32 BUCKETS;
```

Analytics Vidhya

Hive File Formats

- Text file
- SequenceFile
- RCFile
- ORC File

```
CREATE TABLE names
  (fname string, lname string)
STORED AS RCFile;
```

Analytics Vidhya

Hive SerDes

- SerDe = serializer/deserializer
- Determines how records are read from a table and written to HDFS.

- **Previously, we specified the row format using** ROW FORMAT DELIMITED **and** FIELDS TERMINATED BY
  - LazySimpleSerDe is specified implicitly

```
CREATE TABLE people(fname STRING, lname STRING)
ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t';
```

- **You can also specify the SerDe explicitly**
  - Using ROW FORMAT SERDE

```
CREATE TABLE people(fname STRING, lname STRING)
ROW FORMAT SERDE
'org.apache.hadoop.hive.serde2.lazy.LazySimpleSerDe'
WITH SERDEPROPERTIES ('field.delim'='\t');
```

Creating a Table with Regex SerDe

```
05/23/2016 19:45:19 312-555-7834 CALL_RECEIVED ""
05/23/2016 19:48:37 312-555-7834 COMPLAINT "Item damaged"
```

```
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" = "([^ ]*) ([^ ]*) ([^ ]*) ([^ ]*)
\"([^\"]*)\"");
```

event_date	event_time	phone_num	event_type	details
05/23/2016	19:45:19	312-555-7834	CALL_RECEIVED	
05/23/2016	19:48:37	312-555-7834	COMPLAINT	Item damaged

Creating a Table with Regex SerDe


```
CREATE TABLE fixed (
cust_id INT, order_id INT, 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})");
```

cust_id	order_id	order_dt	order_tm	city	state	**Zip**
1030929	6107596	20160829	012215	Oakland	CA	94618

Analytics Vidhya

Hive ORC Files

- The ***Optimized Row Columnar*** (ORC) file format provides a highly efficient way to store Hive data.

```
CREATE TABLE tablename (
...
) STORED AS ORC;
```

```
ALTER TABLE tablename SET FILEFORMAT ORC;
```

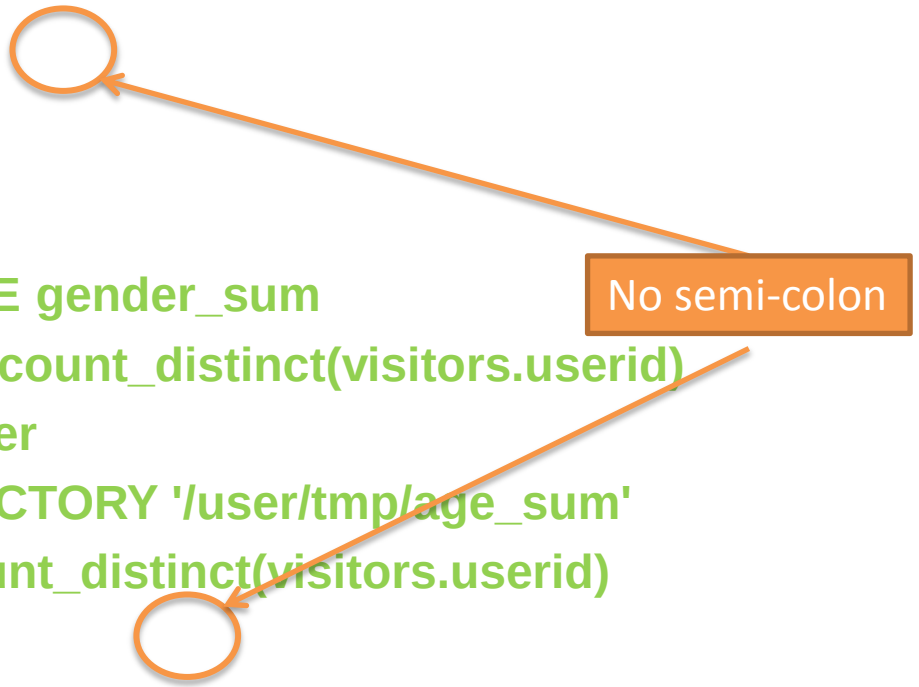
```
SET hive.default.fileformat=Orc
```

Analytics Vidhya

Performing a Multi Table/File Insert

```
insert overwrite directory '2014_visitors' select * from wh_visits where
visit_year='2014'
insert overwrite directory 'ca_congress' select * from congress where
state='CA' ;


from visitors
INSERT OVERWRITE TABLE gender_sum
    SELECT visitors.gender, count_distinct(visitors.userid)
    GROUP BY visitors.gender
 INSERT OVERWRITE DIRECTORY '/user/tmp/age_sum'
    SELECT visitors.age, count_distinct(visitors.userid)
    GROUP BY visitors.age;
```



Analytics Vidhya

Creating Views

- **A *view* is a saved query on a table or set of tables**
  - ─ You can query a view as if it were a table

```
CREATE VIEW order_info AS
SELECT o.order_id, order_date, p.prod_id, brand, name
FROM orders o
JOIN order_details d ON (o.order_id = d.order_id)
JOIN products p ON (d.prod_id = p.prod_id);
```

- **The query is now greatly simplified**

```
SELECT * FROM order_info WHERE order_id=6584288;
```

- **Views are also helpful to provide limited access to table data**
  - ─ Prohibit access to sensitive information in some columns or rows
- **You cannot directly add data to a view**

Exploring Views

- SHOW TABLES **lists the tables *and views* in a database**
  - There is no separate command to list only views

```
SHOW TABLES;
```

- **Use** DESCRIBE FORMATTED **to see a view's underlying query**

```
DESCRIBE FORMATTED order_info;
```

- **Use** SHOW CREATE TABLE **to display a statement to create the view**

```
SHOW CREATE TABLE order_info;
```

Modifying and Removing Views

- **Use** ALTER VIEW **to change the underlying query**

```
ALTER VIEW order_info AS
SELECT order_id, order_date FROM orders;
```

- **Or to rename a view**

```
ALTER VIEW order_info
RENAME TO order_information;
```

- **Use** DROP VIEW **to remove a view**

```
DROP VIEW order_info;
```

Defining Indexes

```
CREATE INDEX zipcode_index ON TABLE Customers (zipcode) AS
'COMPACT' WITH DEFERRED REBUILD;

ALTER INDEX zipcode_index ON Customers REBUILD;

SHOW INDEX ON Customers;

DROP INDEX zipcode_index ON Customers;
```

Analytics Vidhya

Complex Data Types

- **Hive has support for complex data types**
  - Represent multiple values within a single row/column position

Data Type	Description
Array	Ordered list of values, all of the same type
Map	Key-value pairs, each of the same type
Struct	Named fields, of possibly mixed types

Why Use Complex Values?

- **Can be more efficient**
  - Related data is stored together
  - Avoids computationally expensive join queries
- **Can be more flexible**
  - Store an arbitrary amount of data in a single row
- **Sometimes the underlying data is already structured this way**
  - Other tools and languages represent data in nested structures
  - Avoids the need to transform data to flatten nested structures

Using ARRAY Columns with Hive

- **All elements in an** ARRAY **column have the same data type**
- **You can specify a delimiter (default is** control+B**)**

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

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

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

Using MAP Columns with Hive

- Key-value pairs

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

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

Name	Phone
Alice	555-1111
Bob	Null
Carlos	555-6666

Analytics Vidhya

Using MAP Columns with Hive

- MAP keys must all be one data type, and values must all be one data type
  - MAP<*KEY-TYPE*,*VALUE-TYPE*>
- You can specify the key-value separator (default is control+C)

```
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 ':';
```

cust_id	Name	phone
a	Alice	[home:555-1111, Work: 555-2222, Mobile:555-3333]
b	Bob	[mobile555-4444]
c	Carlos	[work:555-5555, Home:555-6666]

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

Analytics Vidhya

Using STRUCT Columns with Hive

- STRUCT **items have names and types**

```
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 '|';
```

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, state:NULL, zipcode:NULL}

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
```

Analytics Vidhya

The OVER Clause

orders

cid	price	quantity
4150	10.50	3
11914	12.25	27
4150	5.99	5
2934	39.99	22
11914	40.50	10

result set

cid	max(price)
2934	39.99
4150	10.50
11914	40.50

```
SELECT cid, max(price) FROM orders GROUP BY cid;
```

Analytics Vidhya

The OVER Clause

orders

cid	price	quantity
4150	10.50	3
11914	12.25	27
4150	5.99	5
2934	39.99	22
11914	40.50	10

result set

cid	max(price)
2934	39.99
4150	10.50
4150	10.50
11914	40.50
11914	40.50

```
SELECT cid, max(price) OVER (PARTITION BY cid) FROM orders;
```

Analytics Vidhya

Using Windows

orders

cid	price	quantity
4150	10.50	3
11914	12.25	27
4150	5.99	5
4150	39.99	22
11914	40.50	10
4150	20.00	2

result set

cid	sum(price)
4150	5.99
4150	16.49
4150	36.49
4150	70.49
11914	12.25
11914	52.75

```
SELECT cid, sum(price) OVER (PARTITION BY cid ORDER BY price ROWS
BETWEEN 2 PRECEDING AND CURRENT ROW) FROM orders;
```

Analytics Vidhya

Using Windows (cont.)

```
SELECT cid, sum(price) OVER (PARTITION BY cid ORDER BY price
ROWS BETWEEN 2 PRECEDING AND 3 FOLLOWING) FROM orders;
```

```
SELECT cid, sum(price) OVER (PARTITION BY cid ORDER BY price
ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW)
FROM orders;
```

Analytics Vidhya

Hive Analytics Function

orders

cid	price	quantity
4150	10.50	3
11914	12.25	27
4150	5.99	5
4150	39.99	22
11914	40.50	10
4150	20.00	2

result set

cid	quantity	rank
4150	2	1
4150	3	2
4150	5	3
4150	22	4
11914	10	1
11914	27	2

```
SELECT cid, quantity, rank() OVER (PARTITION BY cid
ORDER BY quantity) FROM orders;
```

Analytics Vidhya

Computing ngrams in Hive

```
select ngrams(sentences(val),2,100) from mytable;
```

```
select context_ngrams(sentences(val),
      array("error","code",null),
      100)
 from mytable;
```

Analytics Vidhya

HCatalog in the Ecosystem

Java MapReduce

HCatalog

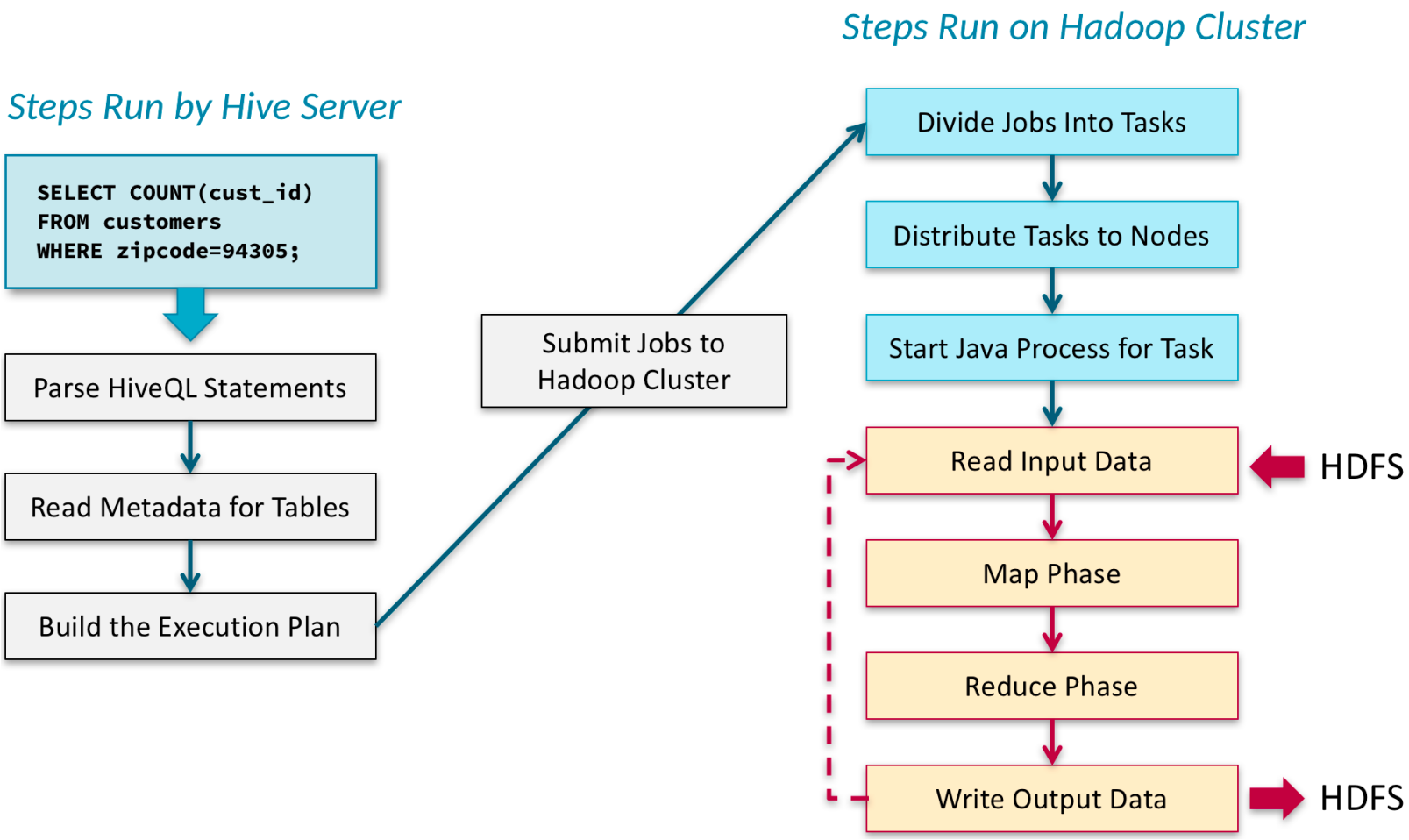
HDFS

HBase

???

Analytics Vidhya

How Hive on MapReduce Processes Data



Analytics Vidhya

Hive Query Performance Patterns

- **The fastest type of query involves only metadata**

  DESCRIBE customers;

- **The next fastest executes as a fetch task**

  SELECT * FROM customers LIMIT 10;

- **Then the type of query that requires only a map phase**

  INSERT INTO TABLE ny_customers
  SELECT * FROM customers
  WHERE state = 'NY';

Hive Query Performance Patterns

- **The next slowest type of query requires both map and reduce phases**

```
SELECT COUNT(cust_id)
FROM customers
WHERE zipcode=94305;
```

- **The slowest type of query requires multiple map and reduce phases**

```
SELECT zipcode, COUNT(cust_id) AS num
FROM customers
GROUP BY zipcode
ORDER BY num DESC
LIMIT 10;
```

Analytics Vidhya

Viewing the Execution Plan

- **How can you tell how Hive will execute a query?**
  - Does it read only metadata?
  - Can it execute the query as a fetch task?
  - Will it require a reduce phase or multiple MapReduce jobs?

```
EXPLAIN SELECT *  FROM customers;
```

- **The output of** EXPLAIN **can be long and complex**
  - Fully understanding it requires in-depth knowledge of MapReduce

Viewing the Execution Plan

- **The query plan contains two main parts**
  - Dependencies between stages
  - Description of the stages

```
EXPLAIN CREATE TABLE cust_by_zip AS
SELECT zipcode, COUNT(cust_id) AS num
FROM customers GROUP BY zipcode;
STAGE DEPENDENCIES:
... (excerpt shown on next slide)
STAGE PLANS:
... (excerpt shown on upcoming slide)
```

Viewing a Query Plan with EXPLAIN

- **Stage-1: MapReduce job**

  Map phase

  – Read customers table

  – Select zipcode and cust_id columns

  Reduce phase

  – Group by zipcode

  – Count cust_id

```
STAGE PLANS:
Stage: Stage-1
Map Reduce
Map Operator Tree:
TableScan
alias: customers
Select Operator
zipcode, cust_id
Reduce Operator Tree:
Group By Operator
aggregations:
count(cust_id)
keys:
zipcode
```

Viewing a Query Plan with EXPLAIN

- **Stage-0: HDFS action**
  — Move previous stage's output to Hive's warehouse directory

```
STAGE PLANS:
Stage: Stage-1 (covered earlier)…
Stage: Stage-0
Move Operator
files:
hdfs directory: true
destination: (HDFS path…)
```

Viewing a Query Plan with EXPLAIN

- **Stage-3: Metastore action**
  - Create new table
  - Has two columns
- **Stage-2: Collect statistics**

```
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: default.cust_by_zip
Stage: Stage-2
Stats-Aggr Operator
```

Hive Optimization Tips

- Divide data amongst different files which can be pruned out by using partitions, buckets and skews
- Sort data ahead of time to simplify joins
- Use the ORC file format
- Sort and Bucket on common join keys
- Use map (broadcast) joins whenever possible
- Take advantage of Tez

Bibliography

- https://hive.apache.org/
- https://www.cloudera.com/products/open-source/apache-hadoop/apache-hive.html
- The Hadoop Definitive Guide by Tom White

Analytics Vidhya