cloudera®
Ask Bigger Questions

# Relational Data Analysis With Impala and Hive

Chapter 12

# Course Chapters

| | |
|---|---|
| ▪ Introduction<br>▪ Hadoop Fundamentals | Course Introduction |
| ▪ Introduction to Pig<br>▪ Basic Data Analysis with Pig<br>▪ Processing Complex Data with Pig<br>▪ Multi-Dataset Operations with Pig<br>▪ Pig Troubleshooting and Optimization | Data ETL and Analysis With Pig |
| ▪ Introduction to Impala and Hive<br>▪ Querying With Impala and Hive<br>▪ Impala and Hive Data Management<br>▪ Data Storage and Performance | Introduction to Impala and Hive |
| ▪ **Relational Data Analysis With Impala and Hive**<br>▪ Working with Impala<br>▪ Analyzing Text and Complex Data with Hive<br>▪ Hive Optimization<br>▪ Extending Hive | **Data Analysis With Impala and Hive** |
| ▪ Choosing the Best Tool for the Job<br>▪ Conclusion | Course Conclusion |

# Relational Data Analysis With Impala and Hive

**In this chapter, you will learn**

- **How to join data from different data sets**
- **How to use many of Hive and Impala's built-in functions**
- **How to group rows to aggregate data**
- **How to use window functions to group data**

# Chapter Topics



- **Joining Datasets**
- Common Built-in Functions
- Aggregation and Windowing
- Conclusion
- Hands-On Exercise: Relational Data Analysis

# Joins

- **Joining disparate data sets is a common operation**
- **Hive and Impala support several types of joins**
  - Inner joins
  - Outer joins (left, right, and full)
  - Cross joins
  - Left semi joins
- **Only equality conditions are allowed in joins**
  - Valid: `customers.cust_id = orders.cust_id`
  - Invalid: `customers.cust_id <> orders.cust_id`
  - Outputs records where the specified key is found in each table
- **For best performance in Hive, list the largest table last in your query**

# Join Syntax

- **Use the following syntax for joins**

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

- **The above example is an inner join**
  - Can replace `JOIN` with another type (e.g. `RIGHT OUTER JOIN`)
- **Implicit joins do the same thing**
  - Supported in Impala and Hive 0.13 and later

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

# Inner Join Example

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 of query

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

# Left Outer Join Example

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 of query

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

# Right Outer Join Example

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 of query

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

# Full Outer Join Example

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 of query

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

# 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 of query

| cust_id | name | total |
|---|---|---|
| d | Dieter | NULL |
| NULL | NULL | 2137 |

# Cross Join Example

disks table

| name |
| --- |
| Internal hard disk |
| External hard disk |

sizes table

| size |
| --- |
| 1.0 terabytes |
| 2.0 terabytes |
| 3.0 terabytes |
| 4.0 terabytes |

```
SELECT * FROM disks
CROSS JOIN sizes;
```

Result of query

| name | size |
| --- | --- |
| Internal hard disk | 1.0 terabytes |
| Internal hard disk | 2.0 terabytes |
| Internal hard disk | 3.0 terabytes |
| Internal hard disk | 4.0 terabytes |
| External hard disk | 1.0 terabytes |
| External hard disk | 2.0 terabytes |
| External hard disk | 3.0 terabytes |
| External hard disk | 4.0 terabytes |

# Left Semi Joins (1)

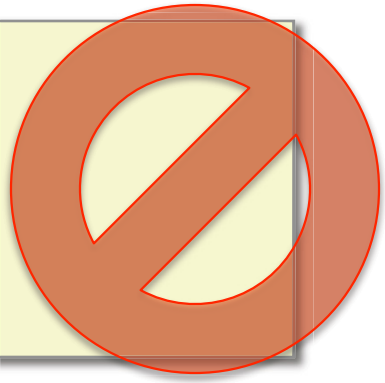
- **A less common type of join is the `LEFT SEMI JOIN`**
  - It is a special (and efficient) type of inner join
  - It behaves more like a filter than a join
- **Left semi joins include additional criteria in the `ON` clause**
  - Only unique records that match these criteria are returned
  - Fields listed in `SELECT` are limited to the left-side table

```
SELECT c.cust_id
  FROM customers c
  LEFT SEMI JOIN orders o
  ON (c.cust_id = o.cust_id
    AND YEAR(o.order_date) = 2012);
```

# Left Semi Joins (2)

- **IN/EXISTS subqueries are not supported**

```
SELECT c.cust_id FROM customers c
  WHERE o.cust_id IN
    (SELECT o.cust_id FROM orders o
     WHERE YEAR(o.order_date) = 2012);
```



- **Using a `LEFT SEMI JOIN` is a common workaround**

```
SELECT c.cust_id
  FROM customers c
  LEFT SEMI JOIN orders o
  ON (c.cust_id = o.cust_id
    AND YEAR(o.order_date) = 2012);
```

# Chapter Topics



- Joining Datasets
- **Common Built-in Functions**
- Aggregation and Windowing
- Conclusion
- Hands-On Exercise: Relational Data Analysis

# Built-in Functions (1)

- **Hive and Impala offer dozens of built-in functions**
  - Many are identical to those found in SQL
  - Others are Hive- or Impala-specific
- **Example function invocation**
  - Function names are not case-sensitive

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

# Built-in Functions (2)

- **To see a list of all functions (Hive only)**


```
SHOW FUNCTIONS;
  abs
  acos
  and
  …
```

- **To see information about a function (Hive only)**


```
DESCRIBE FUNCTION UPPER;
  UPPER(str) - Returns str with all characters
  changed to uppercase
```

# Example Built-in Functions: Numeric Functions

- **These functions operate on numeric values**

| Function Description | Example Invocation | Input | Output |
|---|---|---|---|
| Rounds to specified # of decimals | `ROUND(total_price, 2)` | `23.492` | `23.49` |
| Returns nearest integer above | `CEIL(total_price)` | `23.492` | `24` |
| Returns nearest integer below | `FLOOR(total_price)` | `23.492` | `23` |
| Return absolute value | `ABS(temperature)` | `-49` | `49` |
| Returns square root | `SQRT(area)` | `64` | `8` |
| Returns a random number | `RAND()` | | `0.584977` |

# Example Built-in Functions: Timestamp Functions

- **These functions operate on timestamp values**

| Function Description | Example Invocation | Input | Output |
|---|---|---|---|
| Convert to UNIX format | `UNIX_TIMESTAMP(order_dt)` | `2013-06-14 16:51:05` | `1371243065` |
| Convert to string format | `FROM_UNIXTIME(mod_time)` | `1371243065` | `2013-06-14 16:51:05` |
| Extract date portion | `TO_DATE(order_dt)` | `2013-06-14 16:51:05` | `2013-06-14` |
| Extract year portion | `YEAR(order_dt)` | `2013-06-14 16:51:05` | `2013` |
| Returns # of days between dates | `DATEDIFF(order_dt, ship_dt)` | `2013-06-14, 2013-06-17` | `3` |

# Example Built-in Functions: String Functions

- **These functions operate on strings**

| 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 | `RTRIM(name)` | `␣Bob␣` | `␣Bob` |
| Extract portion of string | `SUBSTRING(name, 3, 4)` | `Samuel` | `muel` |

# Example Built-in Functions: String Concatenation

- **CONCAT combines one or more strings**
  - The `CONCAT_WS` variation joins them with a separator

| Example Invocation | Output |
|---|---|
| `CONCAT('alice', '@example.com')` | `alice@example.com` |
| `CONCAT_WS(' ', 'Bob', 'Smith')` | `Bob Smith` |
| `CONCAT_WS('/', 'Amy', 'Sam', 'Ted')` | `Amy/Sam/Ted` |

*

# Example Built-in Functions: Parsing URLs

- **The `PARSE_URL` function parses Web addresses (URLs)**
- **The following examples assume the following URL as input**
  - `http://www.example.com/click.php?A=42&Z=105`

| Example Invocation | Output |
|---|---|
| `PARSE_URL(url, 'PROTOCOL')` | `http` |
| `PARSE_URL(url, 'HOST')` | `www.example.com` |
| `PARSE_URL(url, 'PATH')` | `/click.php` |
| `PARSE_URL(url, 'QUERY')` | `A=42&Z=105` |
| `PARSE_URL(url, 'QUERY', 'A')` | `42` |
| `PARSE_URL(url, 'QUERY', 'Z')` | `105` |

# Example Built-in Functions: Others

- **Here are some other interesting functions**

| Function Description | Example Invocation | Input | Output |
|---|---|---|---|
| Selectively return value | `IF(price > 1000, 'A', 'B')` | `1500` | `A` |
| Convert to another type | `CAST(weight as INT)` | `3.581` | `3` |

# Chapter Topics



- Joining Datasets
- Common Built-in Functions
- **Aggregation and Windowing**
- Conclusion
- Hands-On Exercise: Relational Data Analysis

# Aggregation and Windowing

- **The functions covered earlier work on data from a single row**
- **Some functions work by combining values from different rows**
- **Aggregation**
  - Groups rows based on a column expression
    - **`GROUP BY column`**
  - Rows in the group are combined
    - Individual row values are not available
- **Windowing**
  - Calculates values within a 'window' without combining rows

# Example: Record Grouping and Aggregate Functions

- **GROUP BY groups selected data by one or more columns**
  - **Caution**: Columns not part of aggregation must be listed in `GROUP BY`

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

products table

| 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 |

query results

| brand | num |
|---|---|
| Dualcore | 3 |
| Gigabux | 4 |

# Built-in Aggregate Functions

- **Hive and Impala offer many aggregate functions, including**

| Function Description | Example Invocation |
|---|---|
| Count all rows | `COUNT(*)` |
| Count all rows where field is not null | `COUNT(fname)` |
| Count all rows where field is unique and not null | `COUNT(DISTINCT fname)` |
| Returns the largest value | `MAX(price)` |
| Returns the smallest value | `MIN(price)` |
| Adds all supplied values and returns result | `SUM(price)` |
| Returns the average of all supplied values | `AVG(price)` |

- **You can also define custom User Defined Aggregate Functions (UDAFs)**

# Window Aggregation

- **Standard aggregation groups rows together into a single set using a function**
  - Details of the individual rows are lost
- **Windowing performs a function on a set of rows without grouping them**
  - Individuals rows are preserved
- **Supported in Hive since 0.11 (CDH 5.0)**
- **Partial support in Impala 2.0 (CDH 5.2)**

# Windows

- A *window* defines a set of rows in a table
- `OVER (window-specification)` specifies a window on which to perform an aggregation or windowing function
- Example: `OVER (PARTITION BY brand)`

| 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 |

Window

Window

# Example: Aggregation Over a Window (1)

- **Question: What is the price of the least expensive product in each brand?**

```
SELECT prod_id, brand, price,
  MIN(price) OVER(PARTITION BY brand) AS m
  FROM products;
```

products table

| 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 |

query results

| prod_id | brand | price | m |
|---|---|---|---|
| 1 | Dualcore | 18.39 | 1.99 |
| 2 | Dualcore | 11.99 | 1.99 |
| 3 | Dualcore | 1.99 | 1.99 |
| 4 | Gigabux | 40.50 | 20.00 |
| 5 | Gigabux | 50.50 | 20.00 |
| 6 | Gigabux | 20.00 | 20.00 |
| 7 | Gigabux | 20.00 | 20.00 |

# Example: Aggregation Over a Window (2)

- **Question: For each product, how does the price compare to the minimum price for that brand?**

```
SELECT prod_id, brand, price,
  price - MIN(price) OVER(PARTITION BY brand) AS d
  FROM products;
```

products table

| 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 |

query results

| prod_id | brand | price | d |
|---|---|---|---|
| 1 | Dualcore | 18.39 | 16.40 |
| 2 | Dualcore | 11.99 | 10.00 |
| 3 | Dualcore | 1.99 | 0.00 |
| 4 | Gigabux | 40.50 | 20.50 |
| 5 | Gigabux | 50.50 | 30.50 |
| 6 | Gigabux | 20.00 | 0.00 |
| 7 | Gigabux | 20.00 | 0.00 |

# Example: RANK and ROW_NUMBER functions

- **Question: Rank the products by price within each brand**

```
SELECT prod_id, brand, price,
  RANK() OVER(PARTITION BY brand ORDER BY price) as rank,
  ROW_NUMBER() OVER(PARTITION BY brand ORDER BY price) as n
  FROM products;
```

products table

| 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 |

query results

| prod_id | brand | price | rank | n |
|---|---|---|---|---|
| 3 | Dualcore | 1.99 | 1 | 1 |
| 2 | Dualcore | 11.99 | 2 | 2 |
| 1 | Dualcore | 18.39 | 3 | 3 |
| 6 | Gigabux | 20.00 | 1 | 1 |
| 7 | Gigabux | 20.00 | 1 | 2 |
| 4 | Gigabux | 40.50 | 3 | 3 |
| 5 | Gigabux | 50.50 | 4 | 4 |

# Example: Using Windowing for Subqueries

- **Question: What is the least expensive from each brand?**

```
SELECT prod_id, brand, price FROM(
  SELECT prod_id, brand, price,
     RANK() OVER(PARTITION BY brand ORDER BY price)
     AS rank FROM products) p
 WHERE rank=1;
```

subquery results

| prod_id | brand | price | rank |
|---|---|---|---|
| 3 | Dualcore | 1.99 | 1 |
| 2 | Dualcore | 11.99 | 2 |
| 1 | Dualcore | 18.39 | 3 |
| 6 | Gigabux | 20.00 | 1 |
| 7 | Gigabux | 20.00 | 1 |
| 4 | Gigabux | 40.50 | 3 |
| 5 | Gigabux | 50.50 | 4 |

query results

| prod_id | brand | price |
|---|---|---|
| 3 | Dualcore | 1.99 |
| 6 | Gigabux | 20.00 |
| 7 | Gigabux | 20.00 |

# Other Windowing Functions

| Function | Description |
|---|---|
| `DENSE_RANK` | Rank of current value within the window (with consecutive rankings) |
| `NTILE(n)` | Which N-tile (of n) is the current value in within the window |
| `PERCENT_RANK` | Ratio of current value to maximum value within the window |
| `CUME_DIST` | Cumulative distribution of current value within the window |
| `LEAD(column,n,default)` * | The value in the specified column in the *n*th following row |
| `LAG(column,n,default)` * | The value in the specified column in the *n*th preceding row |

* Hive only

# Example: RANK and DENSE_RANK

- **Question: Rank the products by price within each brand**

```
SELECT prod_id, brand, price,
  RANK() OVER(PARTITION BY brand ORDER BY price)
     AS rank,
  DENSE_RANK() OVER(PARTITION BY brand ORDER BY price)
     AS d_rank
  FROM products;
```

| prod_id | brand | price | rank | d_rank |
|---|---|---|---|---|
| 6 | Gigabux | 20.00 | 1 | 1 |
| 7 | Gigabux | 20.00 | 1 | 1 |
| 4 | Gigabux | 40.50 | 3 | 2 |
| 5 | Gigabux | 50.50 | 4 | 3 |

# Time-based Windows

- **Analyzing data over time is a common use of windowing**
- **Example: Analyzing ad display data**

ads table

| campaign_id | display_date | display_site | cost | … |
|---|---|---|---|---|
| A1 | 05/01/2013 | audiophile | 76 | … |
| A1 | 05/01/2013 | photosite | 64 | … |
| A3 | 05/01/2013 | photosite | 68 | … |
| A2 | 05/02/2013 | audiophile | 82 | … |
| A3 | 05/02/2013 | dvdreview | 66 | … |
| A3 | 05/02/2013 | audiophile | 82 | … |
| A1 | 05/02/2013 | audiophile | 70 | … |
| A3 | 05/03/2013 | audiophile | 66 | … |
| … | … | … | … | … |

# Example: Time-based Windows

- **Question: How many ads were displayed per site per day?**
- **A non-windowed aggregation:**

```
SELECT display_date, display_site, COUNT(display_time) AS n
    FROM ads GROUP BY display_date, display_site;
```

query results

| display_date | display_site | n |
|---|---|---|
| 05/01/2013 | audiophile | 1 |
| 05/01/2013 | photosite | 2 |
| 05/02/2013 | audiophile | 3 |
| 05/02/2013 | dvdreview | 1 |
| 05/03/2013 | audiophile | 1 |
| … | … | … |

# Example: Rank Over Time-based Windows

- **Question: How did each site rank per day in number of displays?**

```
SELECT display_date, display_site, n,
    RANK() OVER (PARTITION BY display_date ORDER BY n) AS dayrank
  FROM (
    SELECT display_date, display_site, count(display_time) AS n
    FROM ads GROUP BY display_date,display_site) ads
  ORDER BY display_date;
```

| display_date | display_site | n | dayrank |
|---|---|---|---|
| 05/01/2013 | audiophile | 1 | 2 |
| 05/01/2013 | photosite | 2 | 1 |
| 05/02/2013 | audiophile | 3 | 1 |
| 05/02/2013 | dvdreview | 1 | 2 |
| 05/03/2013 | audiophile | 1 | 1 |
| … | … | … | |

# Example: Using LAG with Time-based Windows


- **Question: How did each site's daily count compare to the previous day?**

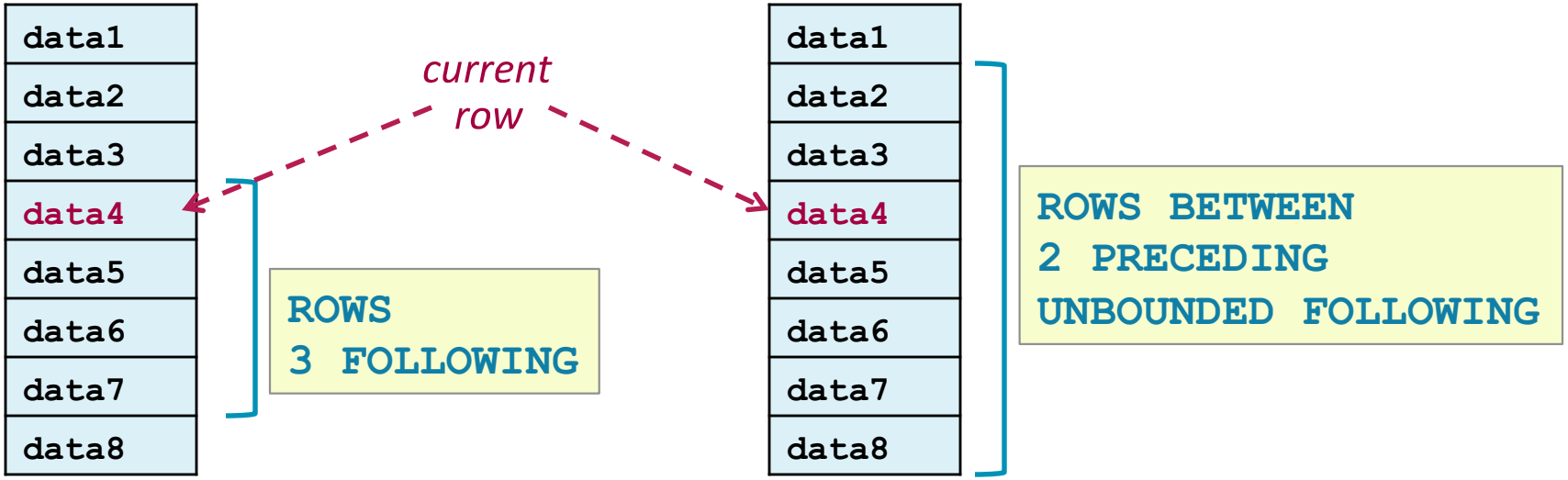
```
SELECT display_date, display_site, n,
    LAG(n) OVER
      (PARTITION BY display_site ORDER BY display_date) AS nprev,
  FROM (
    SELECT display_date, display_site, count(display_time) AS n
    FROM ads GROUP BY display_date,display_site) ads
  ORDER BY display_date;
```

| display_date | display_site | n | nprev |
|---|---|---|---|
| 05/01/2013 | audiophile | 1 | NULL |
| 05/01/2013 | photosite | 2 | NULL |
| 05/02/2013 | audiophile | 3 | 1 |
| 05/02/2013 | dvdreview | 1 | NULL |
| 05/03/2013 | audiophile | 1 | 3 |
| … | … | … | … |

* Hive only

# Sliding Windows

- **There are three parts to a window specification**
  - Partition – `PARTITION BY` (optional in Hive; required in Impala)
  - Ordering – `ORDER BY` (`ASC` or `DESC`) (optional)
  - Frame boundaries (size of the window) – `ROWS` or `RANGE` (Hive only)
    - Sliding window relative to current row



# Example: Time-based Sliding Window


- **Question: What is the average count for each site for the week ending on the current date?**

```
SELECT display_date, display_site, n,
    AVG(n) OVER
      (PARTITION BY display_site ORDER BY display_date
       ROWS BETWEEN 7 PRECEDING AND CURRENT ROW) AS wavg,
  FROM ( …
```

| display_date | display_site | n | wavg |
|---|---|---|---|
| 05/01/2013 | audiophile | 1 | 1 |
| 05/01/2013 | photosite | 2 | 2 |
| 05/02/2013 | audiophile | 3 | 2 |
| 05/02/2013 | dvdreview | 1 | 1 |
| 05/03/2013 | audiophile | 1 | 1.66 |
| … | … | … | … |

* Hive only

# Chapter Topics



- Joining Datasets
- Common Built-in Functions
- Aggregation and Windowing
- **Conclusion**
- Hands-On Exercise: Relational Data Analysis

# Essential Points

- **Hive and Impala support joining data across disparate datasets in several ways**
- **Many standard SQL functions are available**
  - e.g. operations on strings, dates, URLs, numeric values
- **Aggregation functions group multiple rows into single values**
- **Windowing functions calculate values based on a set of rows without combining the rows**

# Bibliography

**The following offer more information on topics discussed in this chapter**

- **Hive Built-In Functions**
  - `http://tiny.cloudera.com/hivefunctions`
- **Impala Built-In Functions**
  - `http://tiny.cloudera.com/impalafunctions`
- **Hive Windowing and Analytics Functions**
  - `http://tiny.cloudera.com/hivewindow`

# Chapter Topics



- Joining Datasets
- Common Built-in Functions
- Aggregation and Windowing
- Conclusion
- **Hands-On Exercise: Relational Data Analysis**

# Hands-On Exercise: Relational Data Analysis

- **In this Hands-On Exercise, you will analyze sales and product data**
- **Please refer to the Hands-On Exercise Manual for instructions**

cloudera®
Ask Bigger Questions

# Working With Impala

## Chapter 13

# Course Chapters

| | |
|---|---|
| ▪ Introduction<br>▪ Hadoop Fundamentals | Course Introduction |
| ▪ Introduction to Pig<br>▪ Basic Data Analysis with Pig<br>▪ Processing Complex Data with Pig<br>▪ Multi-Dataset Operations with Pig<br>▪ Pig Troubleshooting and Optimization | Data ETL and Analysis With Pig |
| ▪ Introduction to Impala and Hive<br>▪ Querying With Impala and Hive<br>▪ Impala and Hive Data Management<br>▪ Data Storage and Performance | Introduction to Impala and Hive |
| ▪ Relational Data Analysis With Impala and Hive<br>▪ **Working with Impala**<br>▪ Analyzing Text and Complex Data with Hive<br>▪ Hive Optimization<br>▪ Extending Hive | **Data Analysis With Impala and Hive** |
| ▪ Choosing the Best Tool for the Job<br>▪ Conclusion | Course Conclusion |

# Working With Impala

**In this chapter, you will learn**

- **How Impala executes queries in a cluster**
- **How to improve Impala performance**
- **How to extend Impala with User Defined Functions**

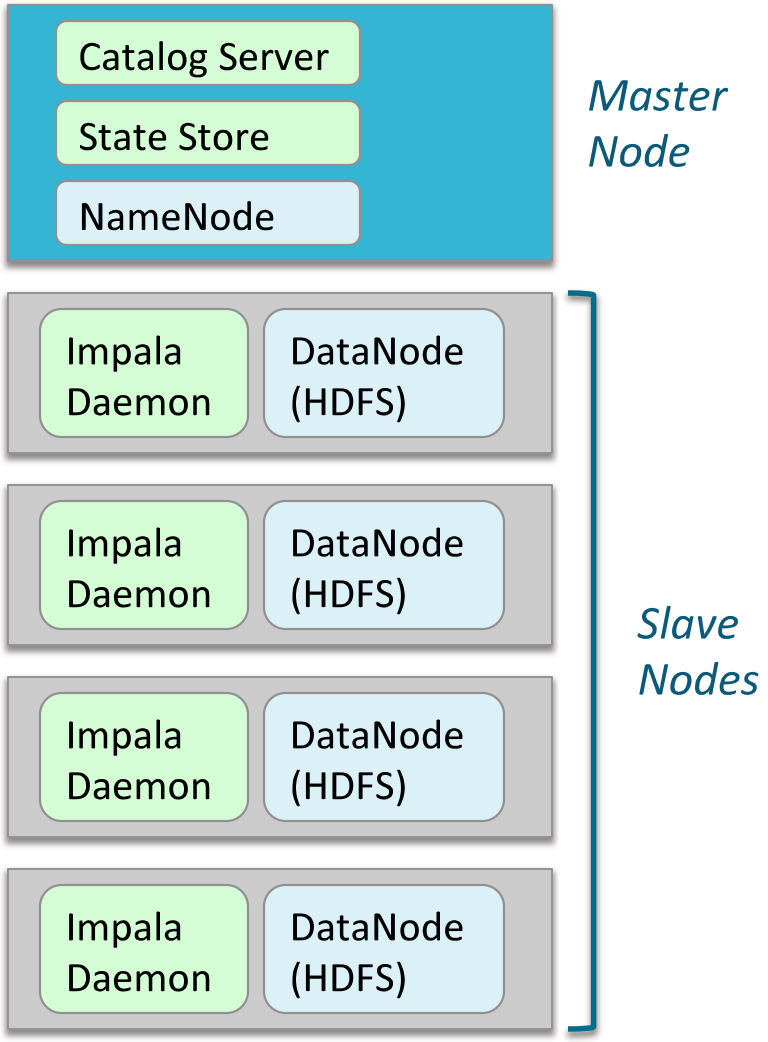
# Chapter Topics

**Working With Impala** | **Data Analysis With Impala and Hive**

- **How Impala Executes Queries**
- Improving Impala Performance
- Extending Impala with User Defined Functions
- Conclusion
- Hands-On Exercise: Working With Impala

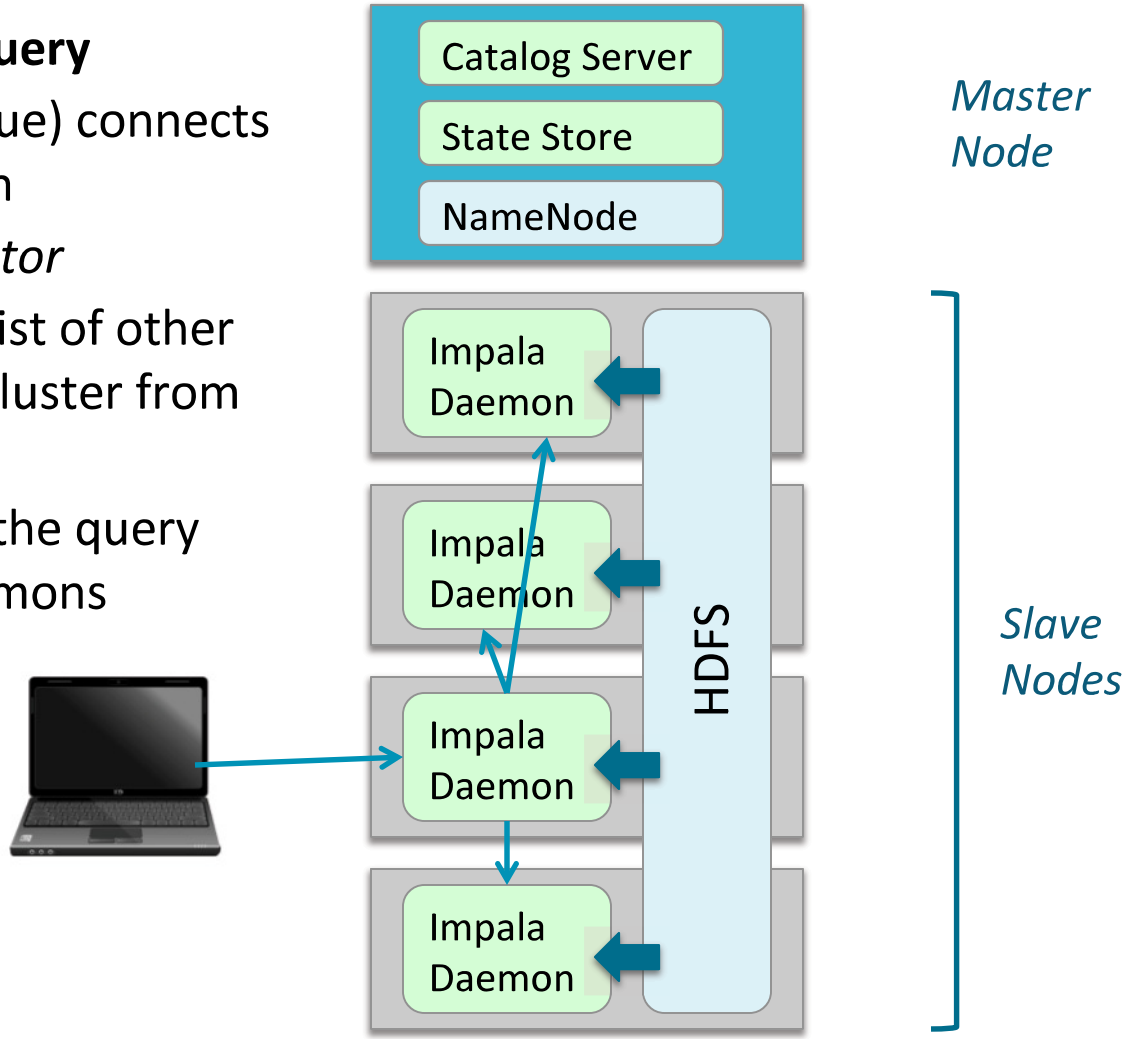
# Impala in the Cluster

- **Each slave node in the cluster runs an Impala daemon**
  - Co-located with the HDFS slave daemon (DataNode)
- **Two other daemons running on master nodes support query execution**
  - The **State Store** daemon
    - Provides lookup service for Impala daemons
    - Periodically checks status of Impala daemons
  - The **Catalog** daemon
    - Relays metadata changes to all the Impala daemons in a cluster



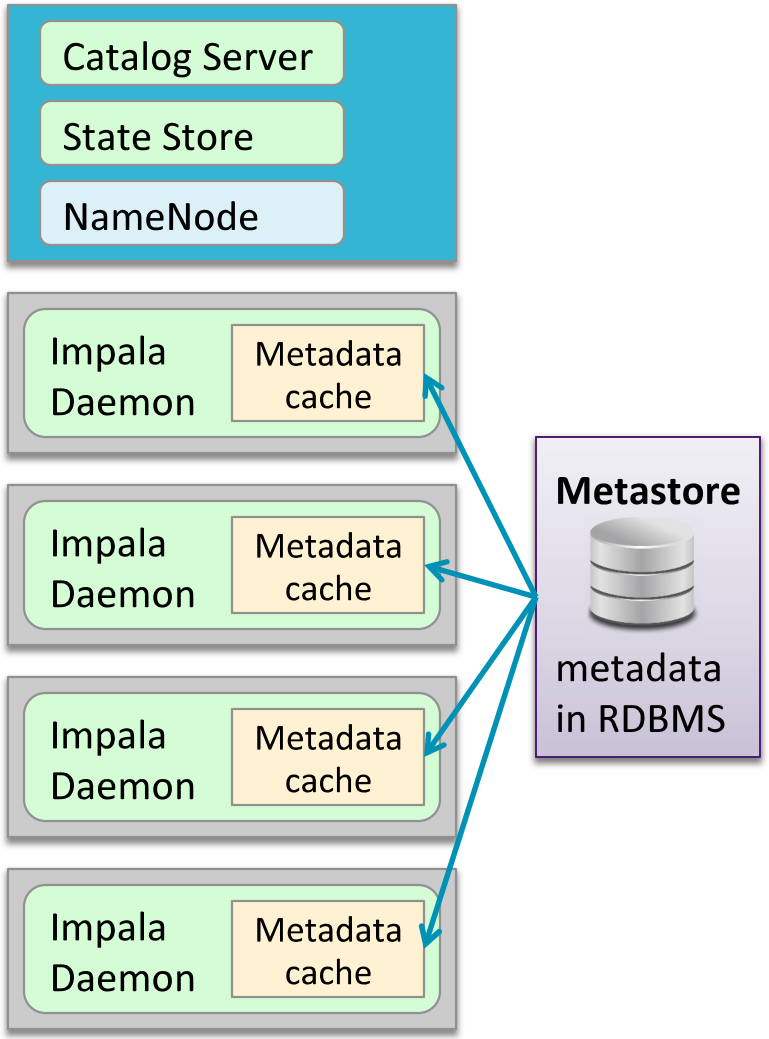
# How Impala Executes a Query

- **Impala daemon plans the query**
  - Client (impala-shell or Hue) connects to a local impala daemon
    - This is the *coordinator*
  - Coordinator requests a list of other Impala daemons in the cluster from the State Store
  - Coordinator distributes the query across other Impala daemons
  - Streams results to client



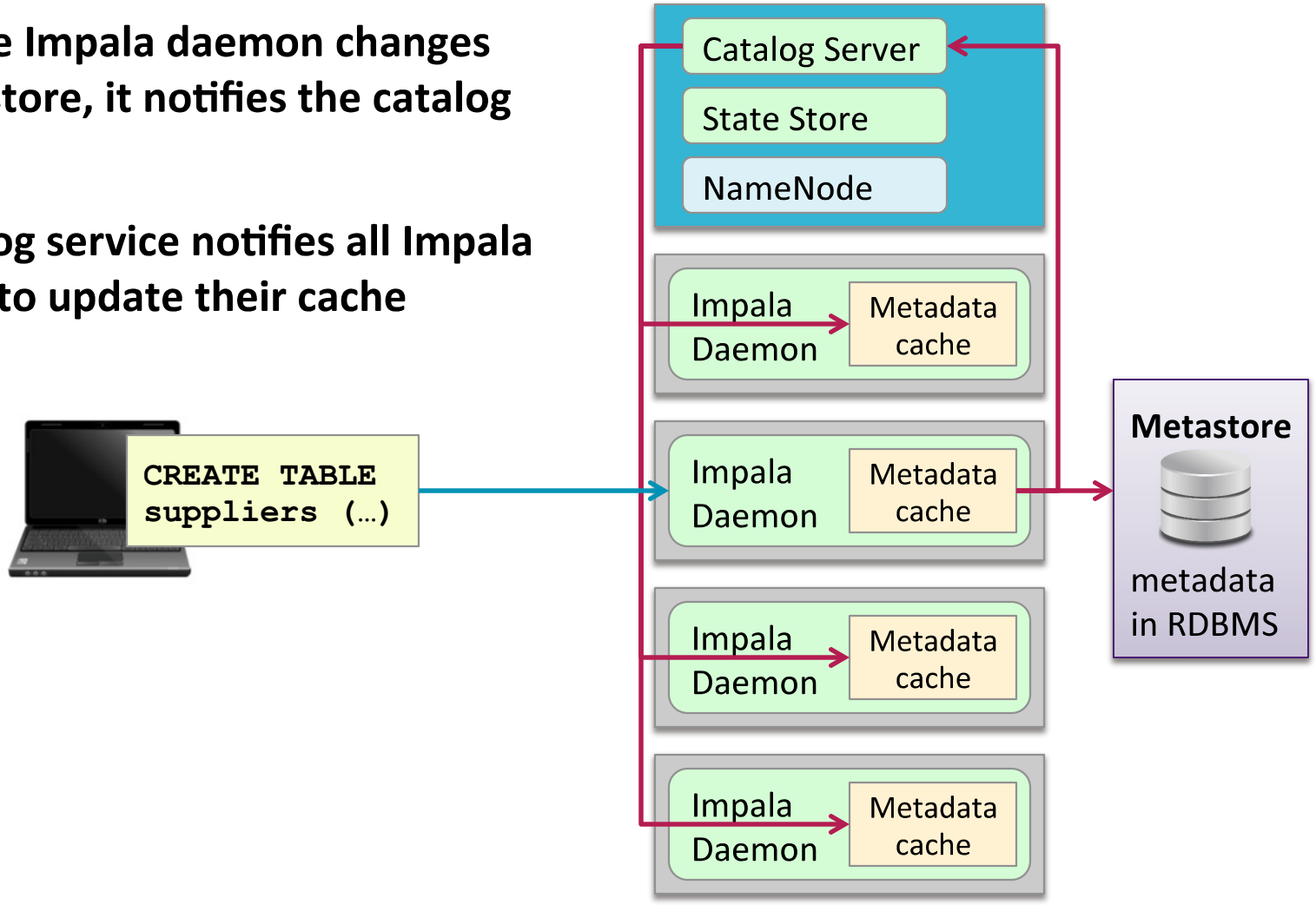
# Metadata Caching (1)

- **Impala daemons cache metadata**
  - The tables' schema definitions
  - The locations of tables' HDFS blocks
- **Metadata is cached from the Metastore at startup**



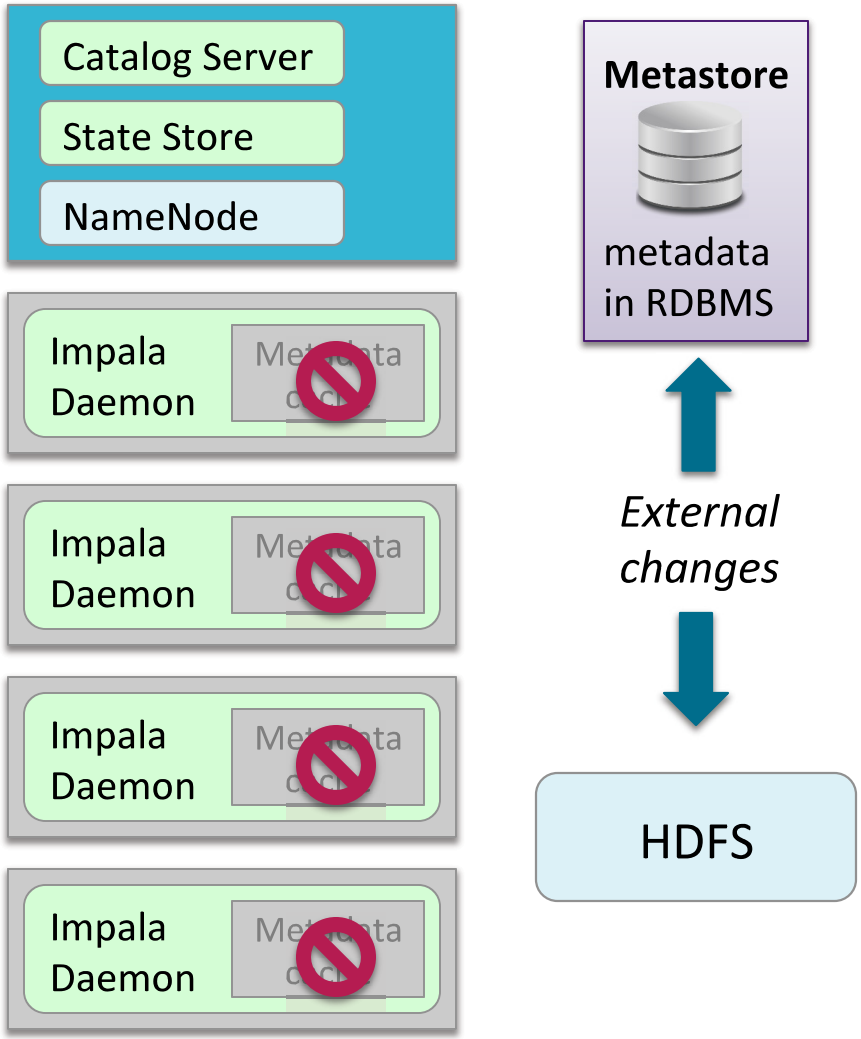
# Metadata Caching (2)

- **When one Impala daemon changes the metastore, it notifies the catalog service**
- **The catalog service notifies all Impala daemons to update their cache**



# External Changes and Metadata Caching

- **Metadata updates made *from outside of Impala* are not known to Impala, e.g.**
  - Changes via Hive
  - Changes via HCatalog
  - Hue Metadata Manager
  - Data added directly to directory in HDFS
- **Therefore the Impala metadata caches will be invalid**
- **You must manually refresh or invalidate Impala's metadata cache**



# Updating the Impala Metadata Cache

| External Metadata Change | Required Action | Effect on Local Caches |
|---|---|---|
| New table added | **`INVALIDATE METADATA`** (with no table name) | Marks the entire cache as stale; metadata cache is reloaded as needed |
| Table schema modified<br>*or*<br>New data added to a table | **`REFRESH <table>`** | Reloads the metadata for one table immediately. Reloads HDFS block locations for new data files only |
| Data in a table extensively altered, such as by HDFS balancing | **`INVALIDATE METADATA <table>`** | Marks the metadata for a single table as stale. When the metadata is needed, all HDFS block locations are retrieved |

# Query Fault Tolerance in Impala

- **Queries in both Hive and Impala are distributed across nodes**
- **Hive answers queries by running MapReduce jobs**
  - Takes advantage of Hadoop's fault tolerance
  - If a node fails during query, MapReduce runs the task elsewhere
- **Impala has its own execution engine**
  - Currently lacks fault tolerance
  - If a node fails during a query, the query will fail
  - Workaround: re-run the query

# Chapter Topics

**Working With Impala** | **Data Analysis With Impala and Hive**

- How Impala Executes Queries
- **Improving Impala Performance**
- Extending Impala with User Defined Functions
- Conclusion
- Hands-On Exercise: Working With Impala

# Impala Performance Overview

- **Query performance is affected by three broad categories**
  - Computing statistics on tables before running joins
  - The format and type of data being queried
  - The hardware and configuration of your cluster

# Query Performance Optimization (1)

- **Impala uses statistics about tables to optimize joins and similar functions**
- **You should compute statistics for tables with `COMPUTE STATS`**
  - After you load a table initially
  - When the amount of data in a table changes substantially

```
COMPUTE STATS orders;
COMPUTE STATS order_details;
SELECT COUNT(o.order_id)
  FROM orders o
  JOIN order_details d
  ON (o.order_id = d.order_id)
  WHERE YEAR(o.order_date) = 2008;
```

# Query Performance Optimization (2)

- **View using SHOW TABLE STATS and SHOW COLUMN STATS**

```
SHOW TABLE STATS orders;
```

| #Rows | #Files | Size | Bytes cached | Format |
|---|---|---|---|---|
| 1662951 | 4 | 60.26MB | NOT CACHED | TEXT |

```
SHOW COLUMN STATS orders;
```

| Column | Type | #Distinct Values | #Nulls | Max Size | Avg Size |
|---|---|---|---|---|---|
| order_id | INT | 2224714 | -1 | NULL | 4 |
| cust_id | INT | 257818 | -1 | NULL | 4 |
| order_date | INT | 1570457 | -1 | NULL | 16 |

# Impala Query Size

- **Query size is based on a query's working set size**
  - The working set of a query contains all records after
    - Filtering rows
    - Pruning unused columns
    - Performing aggregation, if applicable
- **For aggregations, the query size is the working set size for all the tables in the query**
- **For joins, the query size is the working set size for all the tables in the join excluding the largest table**
- **For older versions of Impala, queries must fit into the cluster's aggregate memory**

# Viewing the Query Execution Plan

- **To view Impala's execution plan, prefix your query with `EXPLAIN` or use the Explain button in Hue**

```
EXPLAIN SELECT *
   FROM customers
   WHERE state='NY';

+---------------------------------+
| Explain String                  |
+---------------------------------+
| Estimated Per-Host Requirements:
Memory=48.00MB VCores=1
…
```

# Example: Execution Plan (1)

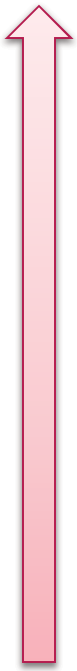
```
SELECT COUNT(o.order_id)
  FROM orders o
  JOIN order_details d
  ON (o.order_id = d.order_id)
  WHERE YEAR(o.order_date) = 2008;
```


```
Estimated Per-Host Requirements: Memory=85.49MB
VCores=1

06:AGGREGATE [FINALIZE]
|  output: count:merge(o.order_id)
|
05:EXCHANGE [UNPARTITIONED]
|
03:AGGREGATE
|  output: count(o.order_id)
|
02:HASH JOIN [INNER JOIN, BROADCAST]
|  hash predicates: d.order_id = o.order_id
|
|--04:EXCHANGE [BROADCAST]
|  |
|  00:SCAN HDFS [default.orders o]
|     partitions=1/1 size=60.26MB
|     predicates: year(o.order_date) = 2008
|
01:SCAN HDFS [default.order_details d]
   partitions=1/1 size=50.86MB
```

# Example: Execution Plan (2)

Requirements for the whole query

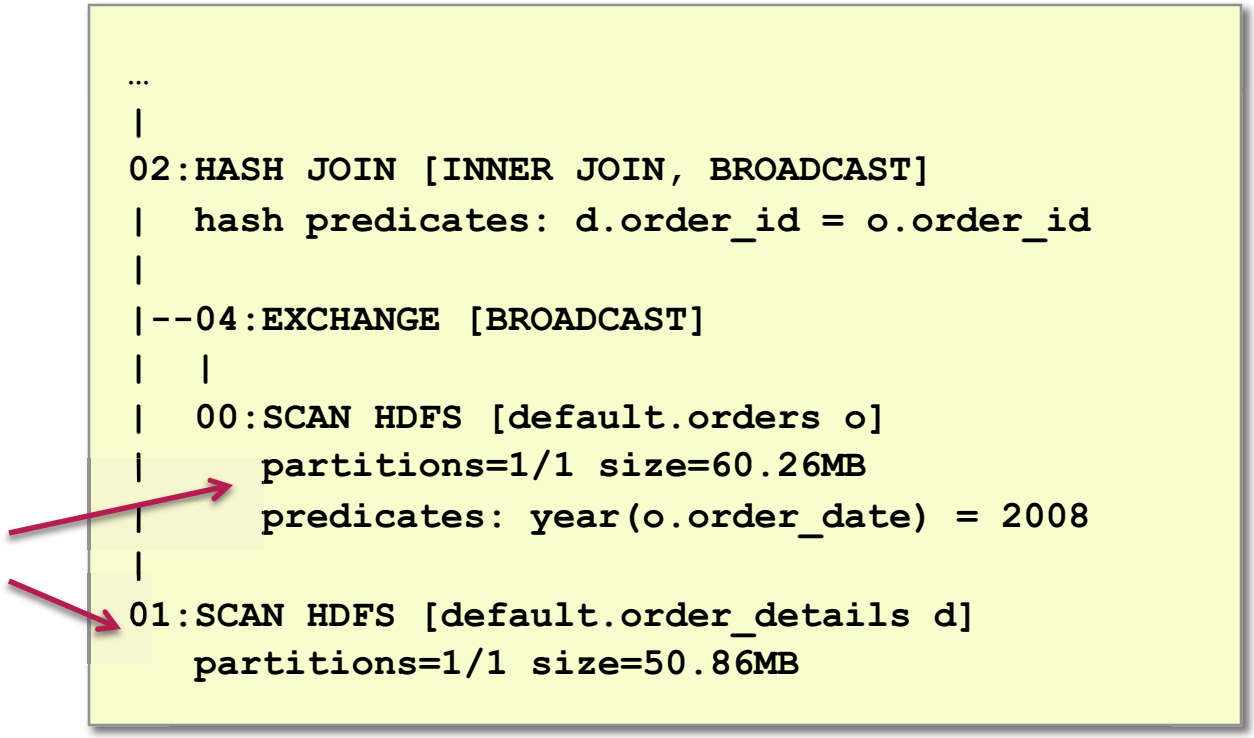
Query stages – Read from the bottom up



```
Estimated Per-Host Requirements: Memory=85.49MB
VCores=1

06:AGGREGATE [FINALIZE]
|  output: count:merge(o.order_id)
|
05:EXCHANGE [UNPARTITIONED]
|
03:AGGREGATE
|  output: count(o.order_id)
|
02:HASH JOIN [INNER JOIN, BROADCAST]
|  hash predicates: d.order_id = o.order_id
|
|--04:EXCHANGE [BROADCAST]
|  |
|  00:SCAN HDFS [default.orders o]
|     partitions=1/1 size=60.26MB
|     predicates: year(o.order_date) = 2008
|
01:SCAN HDFS [default.order_details d]
   partitions=1/1 size=50.86MB
```

# Example: Execution Plan (3)

Joins require scans of both tables



```
...
|
02:HASH JOIN [INNER JOIN, BROADCAST]
|  hash predicates: d.order_id = o.order_id
|
|--04:EXCHANGE [BROADCAST]
|  |
|  00:SCAN HDFS [default.orders o]
|     partitions=1/1 size=60.26MB
|     predicates: year(o.order_date) = 2008
|
01:SCAN HDFS [default.order_details d]
   partitions=1/1 size=50.86MB
```

# Setting Explain Level

- **Set the EXPLAIN_LEVEL property to see more or less detail about a query plan**
  - 0 (MINIMAL) – Useful for checking the join order in very long queries
  - 1 (STANDARD) – Shows the logical way that work is split up
  - 2 (EXTENDED) – Detail about how the query planner uses statistics
  - 3 (VERBOSE) – For Impala developers

```
> SET EXPLAIN_LEVEL=0;
> EXPLAIN SELECT COUNT(o.order_id) FROM orders o
  JOIN order_details d ON (o.order_id = d.order_id)
  WHERE YEAR(o.order_date) = 2008;
Estimated Per-Host Requirements: Memory=85.49MB VCores=1

06:AGGREGATE [FINALIZE]
05:EXCHANGE [UNPARTITIONED]
03:AGGREGATE
02:HASH JOIN [INNER JOIN, BROADCAST]
…
```

# Query Details After Execution

- **The Impala Shell provides commands to show details after running a query (not available in Hue)**
  - **SUMMARY** – overview of timings for query phases
  - **PROFILE** – detailed report of query execution

```
SELECT COUNT(o.order_id) FROM orders o
  JOIN order_details d ON (o.order_id = d.order_id)
  WHERE YEAR(o.order_date) = 2008;
…
SUMMARY;

Operator      #Hosts   Avg Time   Max Time  #Rows  Est. #Rows  Peak Mem  Est. Peak Mem  Detail
------------------------------------------------------------------------------------------------
06:AGGREGATE    1  142.597ms  142.597ms    1     1    16.00 KB -1.00 B  FINALIZE
05:EXCHANGE     1  116.213us  116.213us    1     1    0        -1.00 B  UNPARTITIONED
03:AGGREGATE    1  148.320ms  148.320ms    1     1    10.71 MB 10.00 MB
02:HASH JOIN    1   68.250ms   68.250ms 52.14K 3.33M 6.52 MB  3.49 MB  INNER JOIN, BROADCAST
…
```

# Chapter Topics

**Working With Impala** | **Data Analysis With Impala and Hive**

- How Impala Executes Queries
- Improving Impala Performance
- **Extending Impala with User Defined Functions**
- Conclusion
- Hands-On Exercise: Working With Impala

# Overview of Impala User-Defined Functions (UDFs)

- **Impala supports User-Defined Functions (UDFs)**
  - Version 1.2 and later
- **There are two types of UDFs in Impala**
  - Standard UDFs
  - User-Defined Aggregate Functions (UDAFs)
- **Impala UDFs can be written in Java or C++**
  - C++ UDFs are implemented as shared objects
- **Impala C++ UDFs cannot be used in Hive**
- **Hive UDFs can be used in Impala with no changes**
  - With a few exceptions

# Using a Java UDF in Impala (1)

- **Register the function with Impala**
  - Specify data types that correspond to the method signature of the UDF class' `evaluate` method after the function name
  - Specify data types that correspond to the return type of the UDF class' `evaluate` method in the `RETURNS` clause
  - Identify the jar file containing the UDF class in the `LOCATION` clause
  - Specify the UDF class name in the `SYMBOL` clause
  - You do not need to run a separate `ADD JAR` step

```
CREATE FUNCTION STRIP(STRING) RETURNS STRING
  LOCATION '/user/hive/udfs/MyUDFs.jar'
  SYMBOL='com.example.hive.ql.udf.UDFStrip';
```

# Using a Java UDF in Impala (2)

- **You may then use the function in Impala queries**

```
SELECT STRIP(email_address) FROM employees;
```

# Using a C++ UDF in Impala

- **Register the function with Impala**

```
CREATE FUNCTION COUNT_VOWELS(STRING)
  RETURNS INT
  LOCATION '/user/hive/udfs/sampleudfs.so'
  SYMBOL='CountVowels';
```

- **You may then use the function in your query**

```
SELECT COUNT_VOWELS(email_address) FROM employees;
```

# Chapter Topics

**Working With Impala** | **Data Analysis With Impala and Hive**

- How Impala Executes Queries
- Improving Impala Performance
- Extending Impala with User Defined Functions
- **Conclusion**
- Hands-On Exercise: Working With Impala

# Essential Points

- **Impala has a custom query execution engine to distribute queries in a cluster**
- **Impala caches metadata from the metastore it shares with Hive**
  - Use `INVALIDATE METADATA` or `REFRESH` to update the cache following external changes
- **Impala's query planner uses table and column statistics to optimize join operations**
  - Use `COMPUTE STATS` to calculate stats before querying
- **Use `EXPLAIN` to understand a query plan**
  - Use `SUMMARY` or `PROFILE` to see how the query executed afterwards
- **You can extend Impala with custom User Defined Functions in Java or C++**
  - Java UDFs work with Hive, too

# Bibliography

**The following offer more information on topics discussed in this chapter**

- **Cloudera blog detailing Impala Features and Performance**
  - `http://tiny.cloudera.com/dac15c`
- **Other Cloudera blog posts about Impala**
  - `http://tiny.cloudera.com/impalablog`
- **Impala Language Reference**
  - `http://tiny.cloudera.com/dac16b`

# Chapter Topics

**Working With Impala** | **Data Analysis With Impala and Hive**

- How Impala Executes Queries
- Improving Impala Performance
- Extending Impala with User Defined Functions
- Conclusion
- **Hands-On Exercise: Working With Impala**

# Hands-On Exercise: Working With Impala

- **In this Hands-On Exercise, you will explore query execution plans for various types of queries**
- **Please refer to the Hands-On Exercise Manual for instructions**

cloudera®
Ask Bigger Questions

# Analyzing Text and Complex Data with Hive

## Chapter 14

# Course Chapters

| | |
|---|---|
| ▪ Introduction<br>▪ Hadoop Fundamentals | Course Introduction |
| ▪ Introduction to Pig<br>▪ Basic Data Analysis with Pig<br>▪ Processing Complex Data with Pig<br>▪ Multi-Dataset Operations with Pig<br>▪ Pig Troubleshooting and Optimization | Data ETL and Analysis With Pig |
| ▪ Introduction to Impala and Hive<br>▪ Querying With Impala and Hive<br>▪ Impala and Hive Data Management<br>▪ Data Storage and Performance | Introduction to Impala and Hive |
| ▪ Relational Data Analysis With Impala and Hive<br>▪ Working with Impala<br>▪ **Analyzing Text and Complex Data with Hive**<br>▪ Hive Optimization<br>▪ Extending Hive | **Data Analysis With Impala and Hive** |
| ▪ Choosing the Best Tool for the Job<br>▪ Conclusion | Course Conclusion |

# Analyzing Text and Complex Data with Hive

**In this chapter, you will learn**

- **How Hive stores and queries complex values**
- **How to use regular expressions**
- **What n-grams are and why they are useful (Hive only)**
- **How to estimate how often words or phrases occur in text (Hive only)**

# Chapter Topics

**Analyzing Text and Complex Data with Hive** | **Data Analysis With Impala and Hive**

- **Complex Values in Hive**
- Using Regular Expressions in Hive
- Sentiment Analysis and n-grams
- Conclusion
- Hands-On Exercise: Analyzing Text and Complex Data With Hive

# Complex Column Types

- **Hive has support for complex data types**
  - Not yet supported in Impala

| Column 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 – one table instead of a join**
- **Sometimes the underlying data is already structured this way**

# Example: Customer Phone Numbers

- **How would you store multiple phone numbers for customers?**
- **The traditional way uses two tables joined on a key**

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 |

# Example: Array (1)

- **Hive allows us to store the data in one table**
  - This example uses an Array

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 |

# Example: Array (2)

- **All elements in an array column have the same type**
- **You can specify a delimiter (default is ^B)**

```
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
```

# Example: Maps (1)

- **Another complex data type is a Map**
  - Key-value pairs

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 |

# Example: Maps (2)

- **Map columns have the same type for each key, and for each value**
  - **MAP<*KEY-TYPE*,*VALUE-TYPE*>**
- **You can specify a map key delimiter (default is ^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 ':';
```

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
```

# Example: Structs (1)

- **Structs store structured values as property values**
  - Each item can have a different type

customers_addr table

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

# Example: Structs (2)

- **Queries can select single values or the whole struct**

```
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 |

# Example: Structs (3)

- **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 '|';
```

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
```

# Returning the Number of Items in a Collection

- **The `SIZE` function returns the number of items in an Array or Map**

| 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 |

# Converting Array to Records with EXPLODE

- **The EXPLODE function creates a record for each element in an array**
  - An example of a *table generating function*
  - The alias is required when invoking table generating functions

| 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 |

# Chapter Topics

**Analyzing Text and Complex Data with Hive**

**Data Analysis With Impala and Hive**

- Complex Values in Hive
- **Using Regular Expressions in Hive**
- Sentiment Analysis and n-grams
- Conclusion
- Hands-On Exercise: Analyzing Text and Complex Data With Hive

# Text Processing Overview

- **Traditional data processing relies on highly-structured data**
  - Carefully curated information in rows and columns
- **What types of data are we producing today?**
  - Free-form notes in electronic medical records
  - Application and server log files
  - Social network connections
  - Electronic messages
  - Product ratings
  - …
- **These types of data also contain great value**
  - But extracting it requires a different approach

# Regular Expressions

- **A regular expression (*regex*) matches a pattern in text**
  - Useful when exact matching is not practical

| 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.** |

# Regular Expression Functions

- **Hive and Impala have two important functions that use regular expressions**
  - `REGEXP_EXTRACT` returns the matched text
  - `REGEXP_REPLACE` substitutes another value for the matched text
- **These examples assume that `txt` has the following value**
  - `It's on Oak St. or Maple St in 90210`

```
SELECT REGEXP_EXTRACT(txt, '(\\d{5})', 1)
  FROM message;
90210
```

```
SELECT REGEXP_REPLACE(txt, 'St.?\\s+', 'Street ')
  FROM message;
It's on Oak Street or Maple Street in 90210
```

# Regex SerDe

- **We sometimes need to analyze data that lacks consistent delimiters**
  - Log files are a common example of this

```
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"
```

- **`RegexSerDe` will read records based on supplied regular expression**
  - Allows us to create a table from this log file
  - Only available in Hive

# Creating a Table with Regex SerDe (1)

Log excerpt

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

RegexSerDe

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

- **Each pair of parentheses denotes a field**
  - Field value is text matched by pattern within parentheses

# Creating a Table with Regex SerDe (2)

Log excerpt

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

RegexSerDe

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

Table excerpt

| 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 |

# Regex SerDe in Older Versions of Hive

- **The Regex SerDe wasn't formally part of Hive prior to 0.10.0**
  - It shipped with Hive, but was part of the "hive-contrib" library
- **To use Regex SerDe in 0.9.x and earlier versions of Hive**
  - Add this JAR file to Hive
  - Change the SerDe's package name, as shown below

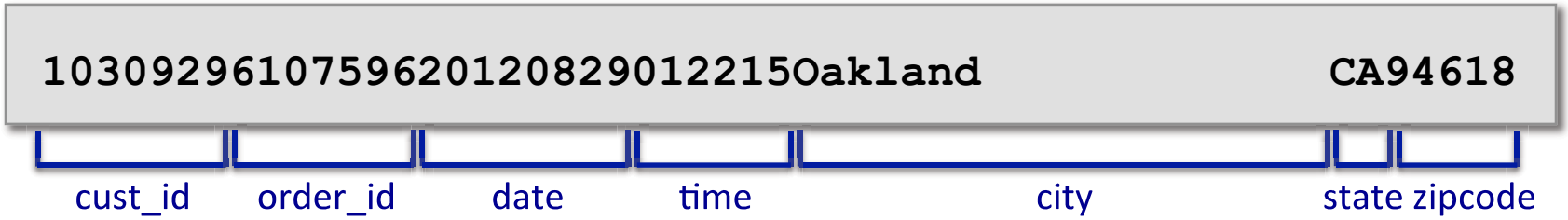
```
CREATE TABLE calls (
  details STRING)
ROW FORMAT SERDE 'org.apache.hadoop.hive.serde2.RegexSerDe'
WITH SERDEPROPERTIES ("input.regex" =
      "([^ ]*) ([^ ]*) ([^ ]*) ([^ ]*) \"([^\"]*)\"");
```

The package name used in older versions is slightly different:

**`org.apache.hadoop.hive.contrib.serde2.RegexSerDe`**

# Fixed-Width Formats

- **Many older applications produce data in fixed-width formats**




- **Unfortunately, Hive doesn't directly support these**
  - But you can overcome this limitation by using `RegexSerDe`
- **Caveat: all fields in `RegexSerDe` are of type `STRING`**
  - May need to cast numeric values in your queries

# Fixed-Width Format Example

Input data

```
1030929610759620120829012215Oakland              CA94618
```

RegexSerDe

```
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})");
```

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

# Chapter Topics

**Analyzing Text and Complex Data with Hive** | **Data Analysis With Impala and Hive**

- Complex Values in Hive
- Using Regular Expressions in Hive
- **Sentiment Analysis and n-grams**
- Conclusion
- Hands-On Exercise: Analyzing Text and Complex Data With Hive

# Sentiment Analysis

- **Sentiment analysis is an application of text analytics**
  - Classification and measurement of opinions
  - Frequently used for social media analysis
- **Context is essential for human languages**
  - Which word combinations appear together?
  - How frequently do these combinations appear?
- **Hive offers functions that help answer these questions**

# Splitting a String into Records

- Hive provides useful functions to work with text
- `SPLIT` and `EXPLODE` work together to generate individual rows for each word in a string

```
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
```

# Parsing Sentences into Words

- **Hive's `SENTENCES` function parses supplied text into words**
- **Input is a string containing one or more sentences**
- **Output is a two-dimensional array of strings**
  - Outer array contains one element per sentence
  - Inner array contains one element per word in that sentence

```
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"]]
```

# n-grams

- **An n-gram is a word combination (n=number of words)**
  - Bigram is a sequence of two words (n=2)
- **n-gram frequency analysis is an important step in many applications**
  - Suggesting spelling corrections in search results
  - Finding the most important topics in a body of text
  - Identifying trending topics in social media messages

# Calculating n-grams in Hive (1)

- **Hive offers the `NGRAMS` function for calculating n-grams**
- **The function requires three input parameters**
  - Array of strings (sentences), each containing an array (words)
  - Number of words in each n-gram
  - Desired number of results (top-N, based on frequency)
- **Output is an array of `STRUCT` with two attributes**
  - `ngram`: the n-gram itself (an array of words)
  - `estfrequency`: estimated frequency at which this n-gram appears

# Calculating n-grams in Hive (2)

- **The `NGRAMS` function is often used with the `SENTENCES` function**
  - We also used `LOWER` to normalize case
  - And `EXPLODE` to convert the resulting array to a series of rows

```
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}
```

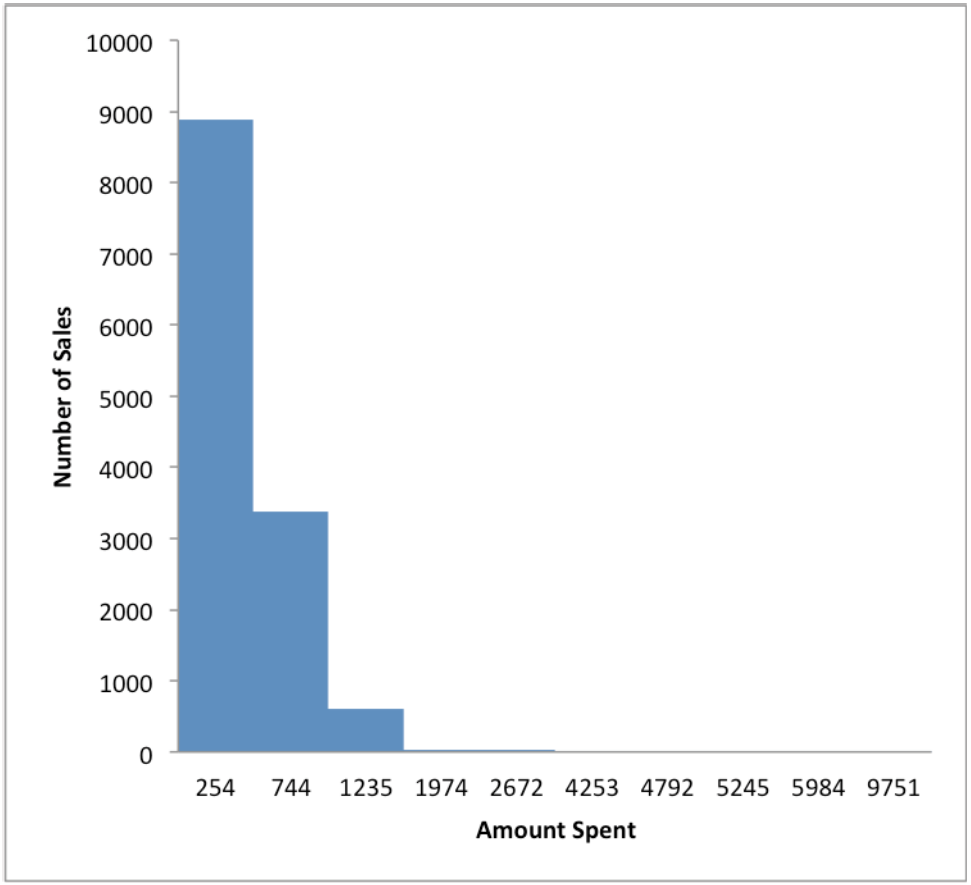
# Finding Specific n-grams in Text

- **CONTEXT_NGRAMS is similar, but considers only specific combinations**
  - Additional input parameter: array of words used for filtering
  - Any `NULL` values in the array are treated as placeholders

```
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.

SELECT EXPLODE(CONTEXT_NGRAMS(SENTENCES(LOWER(txt)),
  ARRAY("new", "computer", NULL, NULL), 2, 3)) AS ngrams
  FROM phrases;
{"ngram":["is","expensive"],"estfrequency":1.0}
{"ngram":["failed","already"],"estfrequency":1.0}
```

# Histograms

- **Histograms illustrate how values in the data are distributed**
  - This helps us estimate the overall shape of the data distribution



# Calculating Data for Histograms

- **HISTOGRAM_NUMERIC creates data needed for histograms**
  - Input: column name and number of "bins" in the histogram
  - Output: coordinates representing bin centers and heights

```
SELECT EXPLODE(HISTOGRAM_NUMERIC(
       total_price, 10)) AS dist FROM cart_orders;
{"x":25417.336745023003,"y":8891.0}
{"x":74401.5041469194,"y":3376.0}
{"x":123550.04418985262,"y":611.0}
{"x":197421.12500000006,"y":24.0}
{"x":267267.53846153844,"y":26.0}
{"x":425324.0,"y":4.0}
{"x":479226.38461538474,"y":13.0}
{"x":524548.0,"y":6.0}
{"x":598463.5,"y":2.0}
{"x":975149.0,"y":2.0}
```

Import this data into charting software to produce a histogram

# Chapter Topics

**Analyzing Text and Complex Data with Hive**

**Data Analysis With Impala and Hive**

- Complex Values in Hive
- Using Regular Expressions in Hive
- Sentiment Analysis and n-grams
- **Conclusion**
- Hands-On Exercise: Analyzing Text and Complex Data With Hive

# Essential Points

- **Hive provides capabilities for analyzing both complex structured data and loosely structured data**
- **Hive support for complex values allows efficient storage and querying of complex structures**
  - Arrays, Maps, Structs
- **Text processing capabilities help analyzing unstructured or partially structured data**
- **Hive has extensive support for regular expressions**
  - You can extract or substitute values based on patterns
  - You can even create a table based on regular expressions
- **An n-gram is a sequence of words**
  - Use `NGRAMS` and `CONTEXT_NGRAMS` to find their frequency

# Chapter Topics

**Analyzing Text and Complex Data with Hive** | **Data Analysis With Impala and Hive**

- Complex Values in Hive
- Using Regular Expressions in Hive
- Sentiment Analysis and n-grams
- Conclusion
- **Hands-On Exercise: Analyzing Text and Complex Data With Hive**

# Hands-On Exercise: Analyzing Text and Complex Data With Hive

- **In this Hands-On Exercise, you will**
  - Use Hive's ability to store complex data to work with data from a customer loyalty program
  - Use a Regex SerDe to load weblog data into Hive
  - Analyze comments in product rating data with Hive
- **Please refer to the Hands-On Exercise Manual for instructions**

cloudera®
Ask Bigger Questions

# Hive Optimization

## Chapter 15

# Course Chapters

| | |
|---|---|
| ▪ Introduction<br>▪ Hadoop Fundamentals | Course Introduction |
| ▪ Introduction to Pig<br>▪ Basic Data Analysis with Pig<br>▪ Processing Complex Data with Pig<br>▪ Multi-Dataset Operations with Pig<br>▪ Pig Troubleshooting and Optimization | Data ETL and Analysis With Pig |
| ▪ Introduction to Impala and Hive<br>▪ Querying With Impala and Hive<br>▪ Impala and Hive Data Management<br>▪ Data Storage and Performance | Introduction to Impala and Hive |
| ▪ Relational Data Analysis With Impala and Hive<br>▪ Working with Impala<br>▪ Analyzing Text and Complex Data with Hive<br>▪ **Hive Optimization**<br>▪ Extending Hive | **Data Analysis With Impala and Hive** |
| ▪ Choosing the Best Tool for the Job<br>▪ Conclusion | Course Conclusion |

# Hive Optimization

**In this chapter, you will learn**

- **Which factors help determine the performance of Hive queries**
- **What command displays Hive's execution plan for a query**
- **How to enable several useful Hive performance features**
- **How to use table bucketing to sample data**
- **How to create and rebuild indexes in Hive**

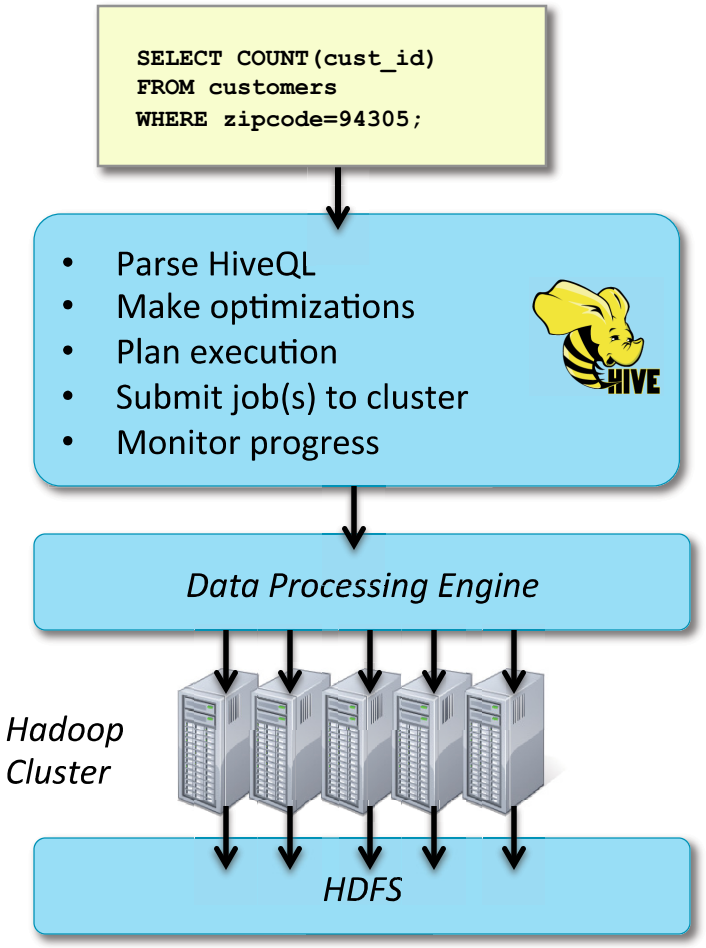
# Chapter Topics

## Hive Optimization

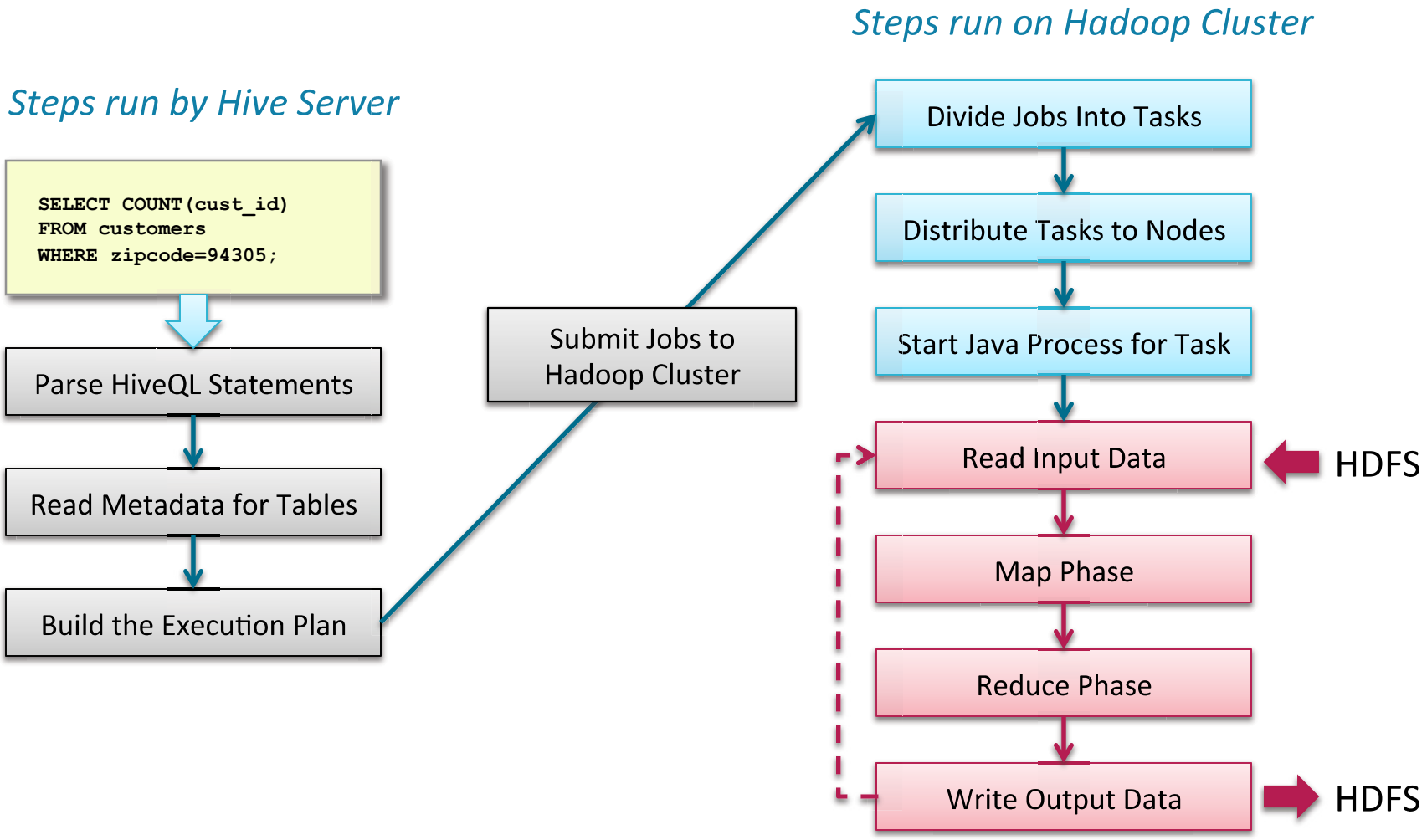
- **Understanding Query Performance**
- Controlling Job Execution
- Bucketing
- Indexing Data
- Conclusion

# Hive Query Processing

- **Recall that Hive generates jobs that are then executed by the underlying data processing engine**
  - e.g., Hadoop MapReduce
- **To optimize Hive queries, you need to understand how queries are processed**

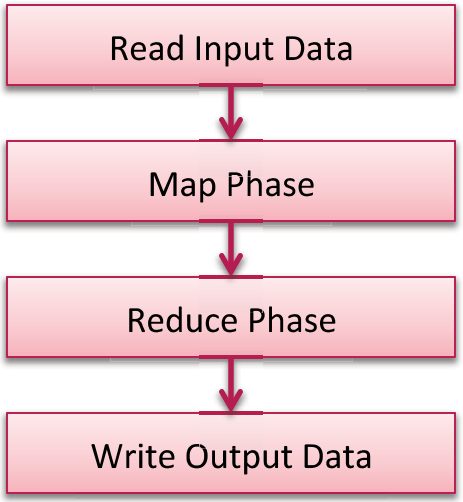


# How Hive Processes Data



# Understanding Map and Reduce

- **A MapReduce job consists of two phases: map and reduce**
  - The output from map becomes the input to reduce
- **The map function always runs first**
  - Used to filter, transform, or parse data
  - Each row processed one at a time
- **The reduce function is optional**
  - Used to summarize data from the map function
  - Aggregates multiple rows
  - Not always needed – some jobs are "map-only"



# MapReduce Example

- **The following slides will show how a query executes as a MapReduce job**
  - Example query to sum order totals by sales representative:

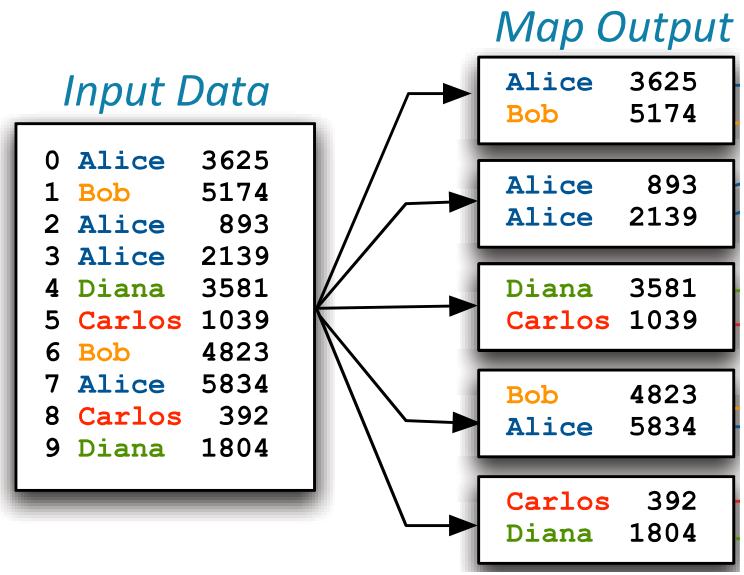
```
SELECT sales_rep, SUM(total) FROM orders
  GROUP BY sales_rep;
```

*Input Data*

```
0 Alice   3625
1 Bob     5174
2 Alice    893
3 Alice   2139
4 Diana   3581
5 Carlos  1039
6 Bob     4823
7 Alice   5834
8 Carlos   392
9 Diana   1804
```

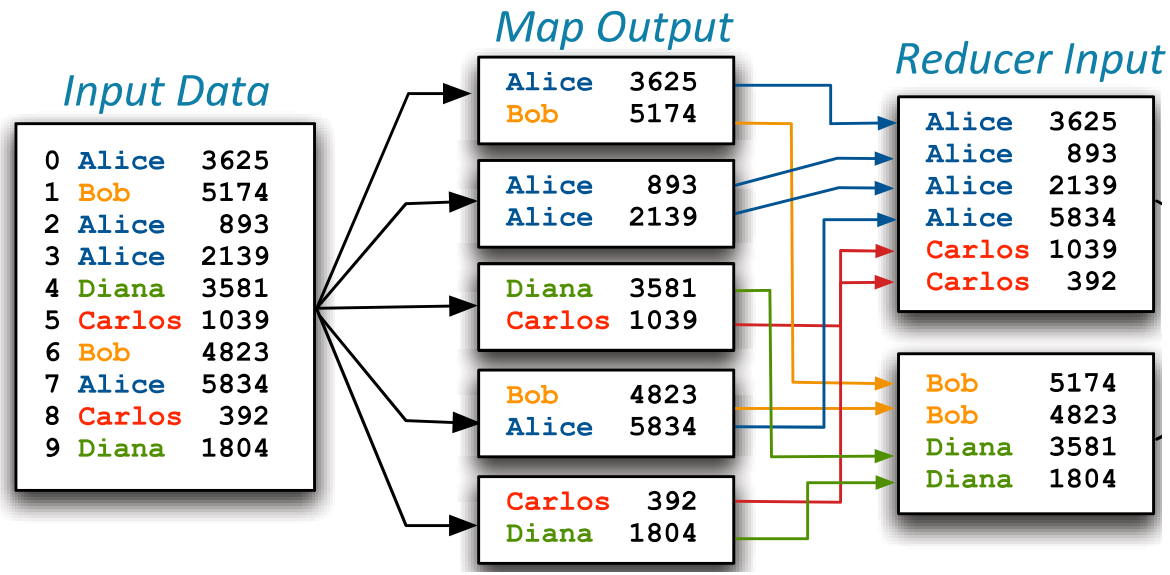
# Explanation of the Map Function

- **Hadoop splits the job into many individual map tasks**
  - Number of map tasks is determined by the amount of input data
  - Each map task receives a portion of the overall job input to process
- **In this example, the map task simply reads the input record**
  - And then emits the name and price fields for each as output



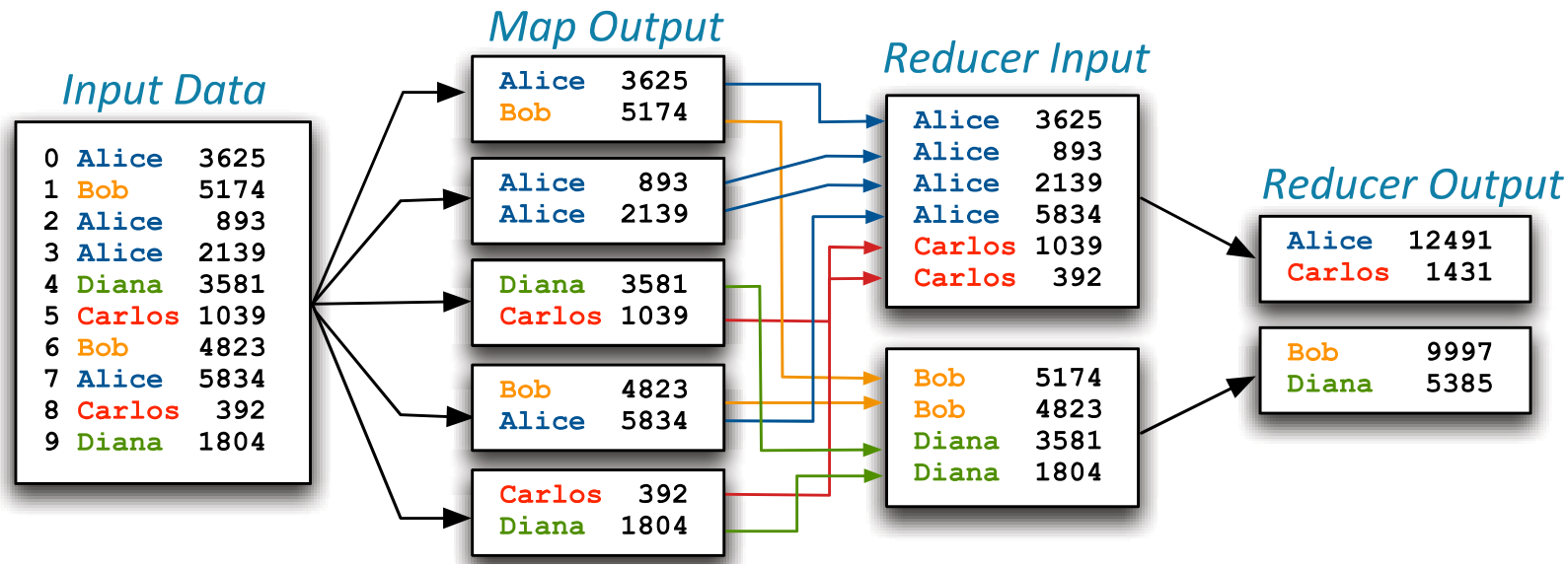
# Shuffle and Sort

- **Hadoop automatically sorts and merges output from all map tasks**
  - This intermediate process is known as the "shuffle and sort"
  - The result is supplied to reduce tasks
  - In this example, the data is sorted by sales rep name, because that is how our query aggregates the data



# Explanation of Reduce Function

- **Reducer input comes from the shuffle and sort process**
  - Reducers process multiple records
  - In this example, reducers processes all the records with the same sales rep name
  - The reducer aggregates by computing the sum of all the order totals for the grouped rows



# Hive Query Performance Patterns (1)

- **The fastest queries involve only metadata**

```
DESCRIBE customers;
```

- **The next fastest simply read from HDFS**

```
SELECT * FROM customers;
```

- **Then a query that requires a map-only job**

```
SELECT * FROM customers WHERE zipcode = 94305;
```

# Hive Query Performance Patterns (2)

- **The next slowest type of query requires both Map and Reduce phases**

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

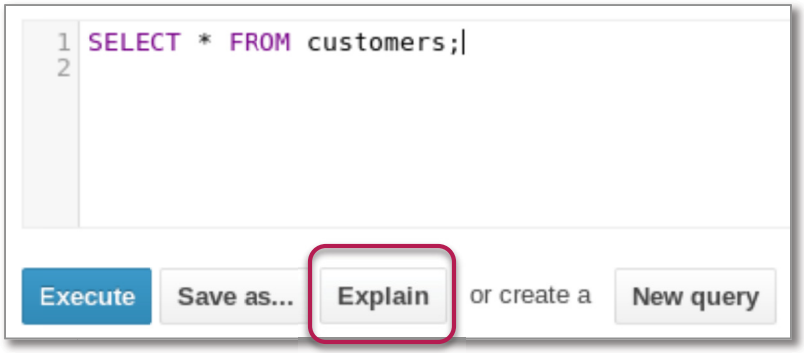
- **The slowest queries require multiple MapReduce jobs**

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

# Viewing the Execution Plan

- **How can you tell how Hive will execute a query?**
  - Does it read only metadata?
  - Can it return data directly from HDFS?
  - Will it require a reduce phase or multiple MapReduce jobs?
- **To view Hive's execution plan, prefix your query with `EXPLAIN` or use the Explain button in Hue**

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




- **The output of `EXPLAIN` can be long and complex**
  - Fully understanding it requires in-depth knowledge of MapReduce
  - We will cover the basics here…

# Viewing a Query Plan with EXPLAIN (1)

- **The query plan contains three main sections**
  - Abstract syntax tree details how Hive parsed query (excerpt below)
  - The stage dependencies and plans are more useful to most users

```
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)
```

# Viewing a Query Plan with EXPLAIN (2)

- **Our query has four stages**
- **Dependencies define order**
  1. Stage-1 (first)
  2. Stage-0
  3. Stage-3
  4. Stage-2 (last)

```
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)
```

# Viewing a Query Plan with EXPLAIN (3)

- **Stage-1: MapReduce job**
- **Map phase**
  - Read customers table
  - Selects `zipcode` and `cust_id` columns
- **Reduce phase**
  - 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
```

# Viewing a Query Plan with EXPLAIN (4)

- **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 (5)

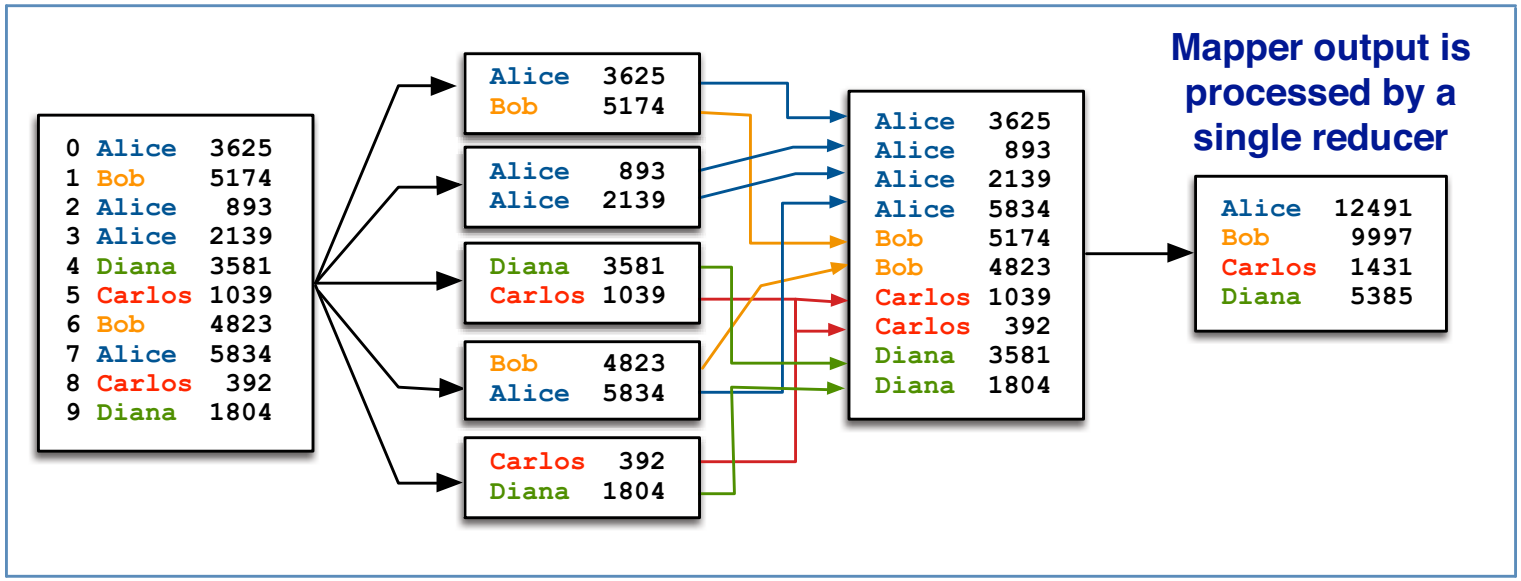
- **Stage-3: Metastore action**
  - Create new table
  - Has two columns
- **Stage-2: Collect statistics**

```
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: cust_by_zip

  Stage: Stage-2
   Stats-Aggr Operator
```

# Sorting Results

- **As in SQL, `ORDER BY` sorts specified fields in HiveQL**
  - Consider the result from the following query

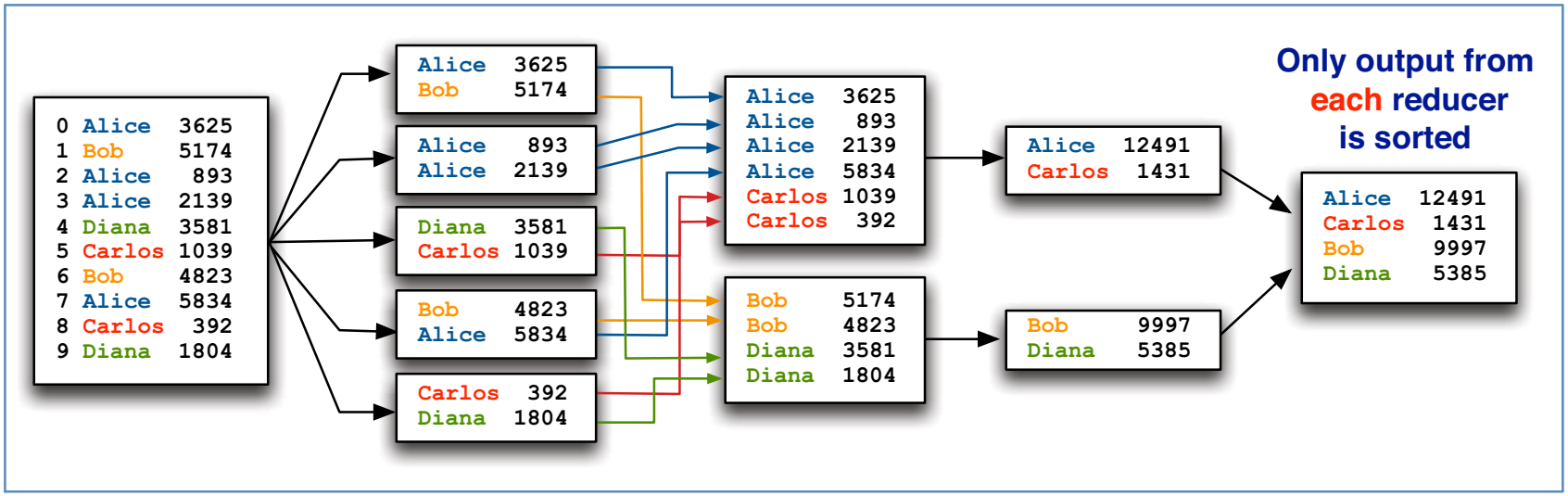
```
SELECT name, SUM(total)
  FROM order_info GROUP BY name
  ORDER BY name;
```



# Using SORT BY for Partial Ordering

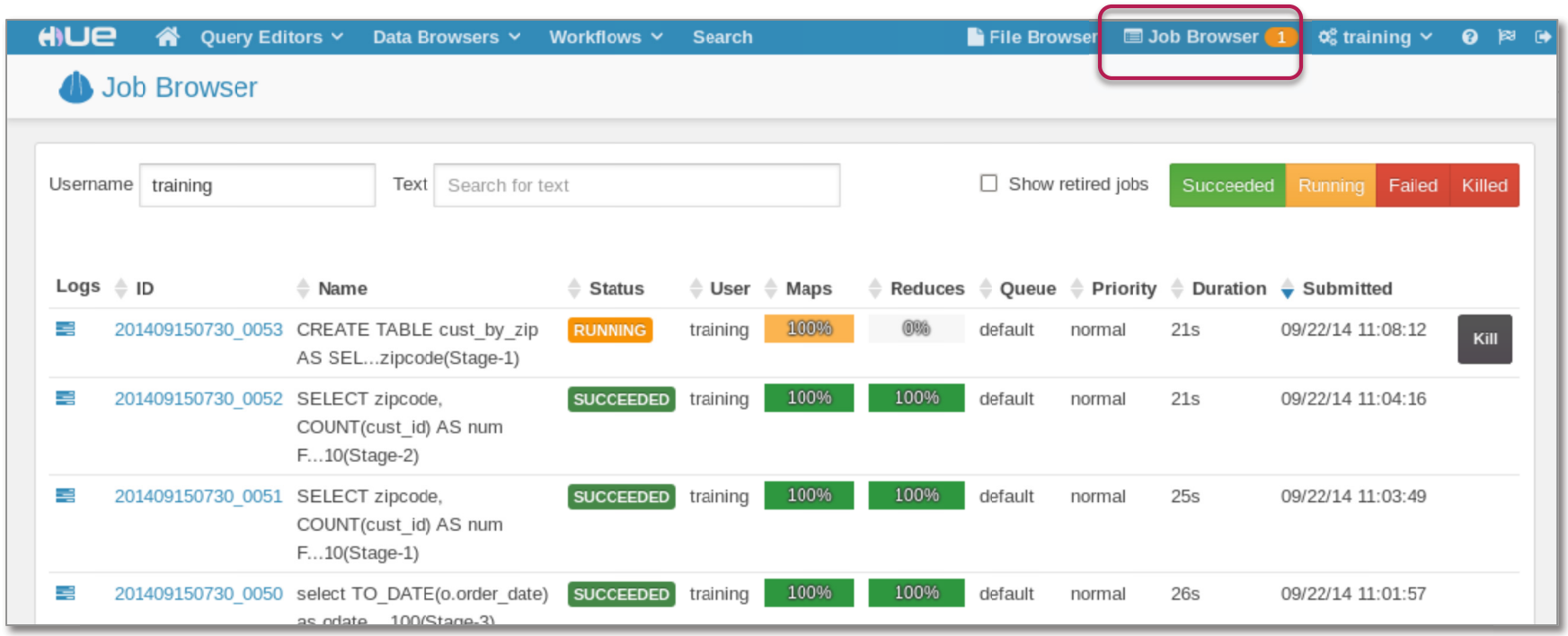
- **Hive also supports partial ordering via SORT BY**
  - Offers much better performance if global order isn't required

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

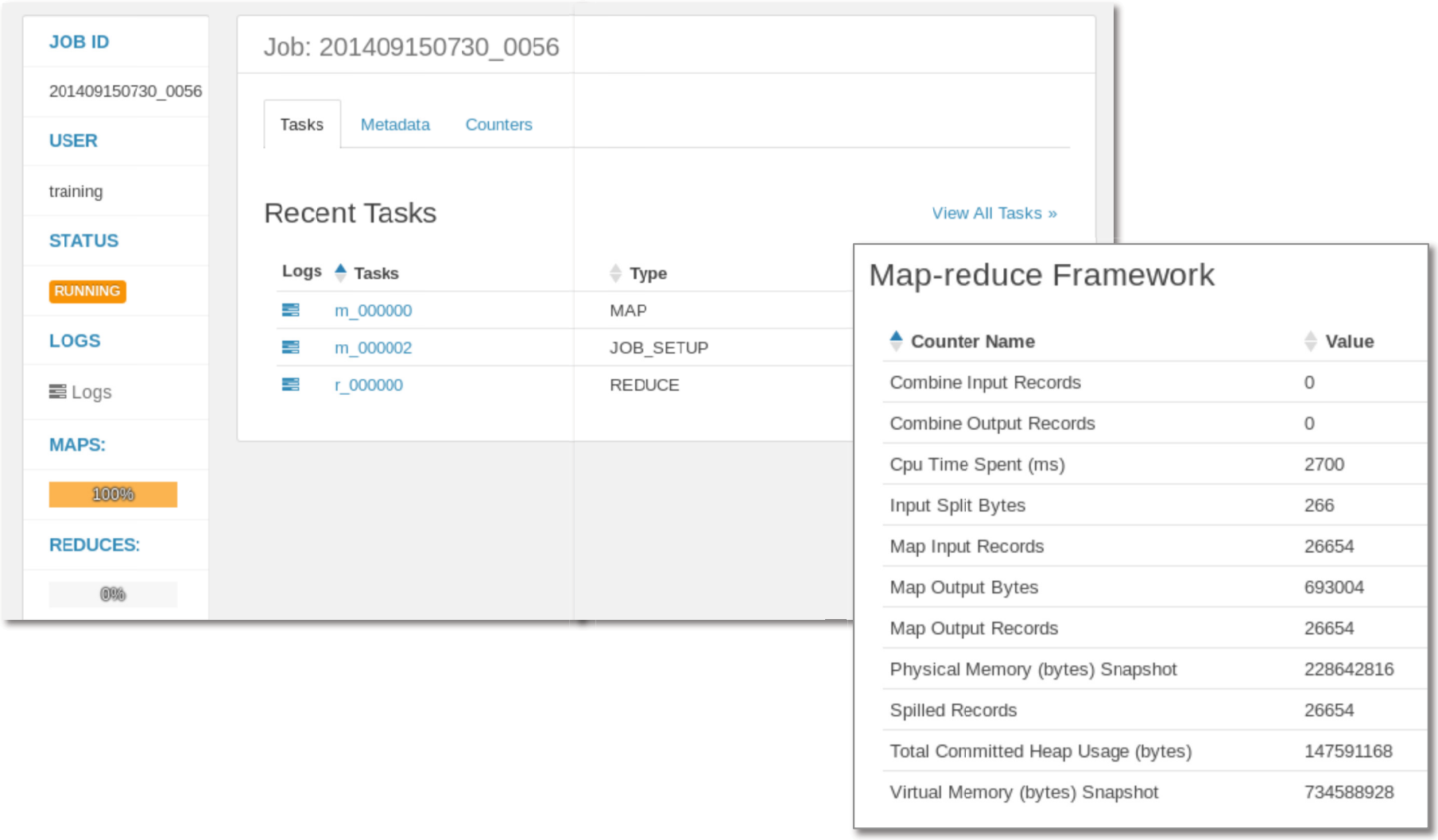


# Viewing a Job in Hue (1)

- **The Hue Job Browser will show you running and recent jobs**



# Viewing a Job in Hue (2)



# Chapter Topics

## Hive Optimization

- Understanding Query Performance
- **Controlling Job Execution**
- Bucketing
- Indexing Data
- Conclusion

# Parallel Execution

- **Stages in Hive's execution plan often lack dependencies**
  - This means they can be run in parallel
- **Hive supports parallel execution in such cases**
  - However, this feature is disabled by default
- **Enable this by setting the `hive.exec.parallel` property to `true`**

# Reducing Latency Through Local Execution

- **Running MapReduce jobs on the cluster has significant overhead**
  - Must divide work, assign tasks, start processes, collect results, etc.
  - Necessary to process large amounts of data in Hive
  - Possibly inefficient with small amount of data
- **Processing data locally can substantially speed up smaller jobs**
  - Local execution can substantially improve turnaround for small jobs
- **Use the Hive command shell**
  - An alternative HiveQL command line interface
  - Does not connect to Hive Server
  - Runs locally or submits job directly to the cluster
  - Mostly replaced by Beeline shell, but useful for local processing

# Using the Hive Shell in Local Execution Mode

- **Start the Hive shell with the `hive` command**
- **Set environment for local processing**

```
$ hive

hive> SET mapred.job.tracker=local;
hive> SET mapred.local.dir=/home/training/tmpdata;

hive> SELECT zipcode, COUNT(cust_id) AS num
      FROM customers GROUP BY zipcode;
```

# Chapter Topics

## Hive Optimization

- Understanding Query Performance
- Controlling Job Execution
- **Bucketing**
- Indexing Data
- Conclusion

# What Is Bucketing?

- **Partitioning subdivides data by values in partitioned columns**
- **Bucketing data is another way of subdividing data**
  - Calculates hash code for values inserted into bucketed columns
  - Hash code used to assign new records to a “bucket”
- **Goal: distribute rows across a predefined number of buckets**
  - Useful for jobs which need random samples of data
  - Joins may be faster if all tables are bucketed on the join column

# Creating A Bucketed Table

- **Example of creating a table that supports bucketing**
  - Creates a table supporting 20 buckets based on `order_id` column
  - Each bucket should contain roughly 5% of the table's data

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

- **Column selected for bucketing should have well-distributed values**
  - Identifier columns are often a good choice

# Inserting Data Into A Bucketed Table

- **Bucketing isn't automatically enforced when inserting data**
- **Set the `hive.enforce.bucketing` property to `true`**
  - This sets the number of reducers to the number of buckets in the table definition

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

# Sampling Data From A Bucketed Table

- **Use the following syntax to sample data from a bucketed table:**
  - This example selects one of every ten records (10%)

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

- **It is possible to use `TABLESAMPLE` on a non-bucketed table**
  - However, this requires a full scan of the entire table

# Chapter Topics

## Hive Optimization

- Understanding Query Performance
- Controlling Job Execution
- Bucketing
- **Indexing Data**
- Conclusion

# Indexes in Hive

- **Tables in Hive also support indexes**
- **Similar to indexes in RDBMSs, but much more limited**
- **May improve performance for certain types of queries**
  - But maintaining them costs disk space and CPU time
- **Syntax to create an index:**

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

- **Handler class is the fully-qualified name of a Java class, such as:**
  - `org.apache.hadoop.hive.ql.index.compact.CompactIndexHandler`

# Viewing and Building Indexes in Hive

- **This command lists the indexes associated with the `orders` table**

```
SHOW FORMATTED INDEX ON orders;
```

- **Hive indexes are initially empty**
  - Building (and later rebuilding) indexes is a manual process
  - Use the `ALTER INDEX` command to rebuild an index
  - Caution: this can be a lengthy operation!

```
ALTER INDEX idx_orders_cust_id ON orders REBUILD;
```

# Chapter Topics

## Hive Optimization

- Understanding Query Performance
- Controlling Job Execution
- Bucketing
- Indexing Data
- **Conclusion**

# Essential Points

- **The EXPLAIN command shows a query's execution plan**
  - Understanding the execution plan helps you understand query performance
- **Local execution mode can significantly reduce query latency**
  - But only appropriate to use with small amounts of data
- **Bucketing subdivides a table's data**
  - Useful for jobs which need random samples of data
  - Joins may be faster if all tables are bucketed on the join column
- **Hive's indexing feature can boost performance for certain queries**
  - But it comes at the cost of increased disk and CPU usage

# Bibliography

**The following offer more information on topics discussed in this chapter**

- **Hive Manual for the `EXPLAIN` Command**
  - `http://tiny.cloudera.com/dac13a`
- **Hive Manual for Bucketed Tables**
  - `http://tiny.cloudera.com/dac13b`
- **Hive Manual for Indexes**
  - `http://tiny.cloudera.com/dac13c`

cloudera®
Ask Bigger Questions

# Extending Hive

## Chapter 16

# Course Chapters

| | |
|---|---|
| ▪ Introduction<br>▪ Hadoop Fundamentals | Course Introduction |
| ▪ Introduction to Pig<br>▪ Basic Data Analysis with Pig<br>▪ Processing Complex Data with Pig<br>▪ Multi-Dataset Operations with Pig<br>▪ Pig Troubleshooting and Optimization | Data ETL and Analysis With Pig |
| ▪ Introduction to Impala and Hive<br>▪ Querying With Impala and Hive<br>▪ Impala and Hive Data Management<br>▪ Data Storage and Performance | Introduction to Impala and Hive |
| ▪ Relational Data Analysis With Impala and Hive<br>▪ Working with Impala<br>▪ Analyzing Text and Complex Data with Hive<br>▪ Hive Optimization<br>▪ **Extending Hive** | **Data Analysis With Impala and Hive** |
| ▪ Choosing the Best Tool for the Job<br>▪ Conclusion | Course Conclusion |

# Extending Hive

**In this chapter, you will learn**

- **What role SerDes play in Hive**
- **How to use a custom SerDe**
- **How to use `TRANSFORM` for custom record processing**
- **How to add support for a User-Defined Function (UDF)**
- **How to use variable substitution**

# Chapter Topics



- **SerDes**
- Data Transformation with Custom Scripts
- User-Defined Functions
- Parameterized Queries
- Conclusion
- Hands-On Exercise: Data Transformation with Hive

# Hive SerDes

- **Hive uses a SerDe for reading and writing records**
  - Stands for "Serializer / Deserializer"
  - SerDes control the row format of the table
  - Specified, sometimes implicitly, when table is created
- **Hive ships with many SerDes, including:**

| Name | Reads and Writes Records |
|---|---|
| `LazySimpleSerDe` | Using specified field delimiters (default) |
| `RegexSerDe` | Based on supplied patterns |
| `ColumnarSerDe` | Using the columnar format needed by RCFile |
| `HBaseSerDe` | Using an HBase table |

# Recap: Creating a Table with Regex SerDe

Input Data

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

Using SerDe

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

Resulting Table

| event_date | event_time | event_type | phone_num | 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 |

# Adding a Custom SerDe to Hive

- **Hive also allows writing custom SerDes using its Java API**
  - There are many open source SerDes on the Web
  - Writing your own is seldom necessary
- **We will now explain how to add a custom SerDe to Hive**
  - It reads and writes records in CSV format
  - Using JAR file from `http://tiny.cloudera.com/dac14a`

# Adding a JAR File to Hive

- **You must register external libraries before using them**
  - Ensures Hive can find the library (JAR file) at runtime

```
ADD JAR hdfs:/dualcore/scripts/csv-serde-1.0.jar;
```

- **Remains in effect only during the current Beeline session**
  - Your system manager can add the JAR permanently

# Using the SerDe in Hive

Input Data

```
1,Gigabux,gigabux@example.com
2,"ACME Distribution Co.",acme@example.com
3,"Bitmonkey, Inc.",bmi@example.com
```

Specify SerDe

```
CREATE TABLE vendors
    (id INT,
     name STRING,
     email STRING)
  ROW FORMAT SERDE 'com.bizo.hive.serde.csv.CSVSerde';
```

Resulting Table

| id | name | email |
|---|---|---|
| 1 | Gigabux | gigabux@example.com |
| 2 | ACME Distribution Co. | acme@example.com |
| 3 | Bitmonkey, Inc. | bmi@example.com |

# Chapter Topics



- SerDes
- **Data Transformation with Custom Scripts**
- User-Defined Functions
- Parameterized Queries
- Conclusion
- Hands-On Exercise: Data Transformation with Hive

# Using TRANSFORM to Process Data Using External Scripts

- **You are not limited to manipulating data exclusively in HiveQL**
  - Hive allows you to transform data through external scripts or programs
  - These can be written in nearly any language
- **This is done with HiveQL's `TRANSFORM . . . USING` construct**
  - One or more fields are supplied as arguments to `TRANSFORM()`
  - Copy the script to HDFS, and identify it with `USING`
    - It receives each record, processes it, and returns the result

```
SELECT TRANSFORM(product_name, price)
  USING 'hdfs:/myscripts/tax_calculator.py'
  FROM products;
```

# Data Input and Output with TRANSFORM

- **Your external program will receive one record per line on standard input**
  - Each field in the supplied record will be a tab-separated string
  - NULL values are converted to the literal string \N
- **You may need to convert values to appropriate types within your program**
  - For example, converting to numeric types for calculations
- **Your program must return tab-delimited fields on standard output**
  - Output fields can optionally be named and cast using the syntax below

```
SELECT TRANSFORM(product_name, price)
  USING 'hdfs:/myscripts/tax_calculator.py'
  AS (item_name STRING, tax INT)
  FROM products;
```

# Hive TRANSFORM Example (1)

- **Here is a complete example of using TRANSFORM in Hive**
  - Our Perl script parses an e-mail address, determines to which country it corresponds, and then returns an appropriate greeting
  - Here's a sample of the input data

**employees table**

| fname | email |
|---|---|
| Antoine | antoin@example.fr |
| Kai | kai@example.de |
| Pedro | pedro@example.mx |

  - Here's the corresponding HiveQL code

```
SELECT TRANSFORM(fname, email)
  USING 'hdfs:/dualcore/scripts/greeting.pl'
  AS greeting
  FROM employees;
```

# Hive TRANSFORM Example (2)

- **The Perl script for this example is shown below**
  - A complete explanation of this script follows on the next few slides

```
#!/usr/bin/env perl

%greetings = ('de' => 'Hallo',
              'fr' => 'Bonjour',
              'mx' => 'Hola');

while (<STDIN>) {
   ($name, $email) = split /\t/;
   ($suffix) = $email =~ /\.([a-z]+)/;
   $greeting = $greetings{$suffix};
   $greeting = 'Hello' unless defined($greeting);
   print "$greeting $name\n";
}
```

# Hive TRANSFORM Example (3)

```
#!/usr/bin/env perl

%greetings = ('de' => 'Hallo',
              'fr' => 'Bonjour',
              'mx' => 'Hola');

while (<
    ($nam
    ($suf
    $gree
    $gree
    print
}
```

The first line tells the system to use the Perl interpreter when running this script.

We define our greetings in the next line using an associative array keyed by the country code we'll extract from the e-mail address.

# Hive TRANSFORM Example (4)

```
#!/usr/bin/env perl

%greeti

while (<STDIN>) {
    ($name, $email) = split /\t/;
    ($suffix) = $email =~ /\.([a-z]+)$/;
    $greeting = $greetings{$suffix};
    $greeting = 'Hello' unless defined($greeting);
    print "$greeting $name\n";
}
```

We read each record from standard input within the loop, and then split them into fields based on tab characters.

# Hive TRANSFORM Example (5)

```
#!/usr/bin/env perl

%greetings = ('de' => 'Hallo',
              'fr' => 'Bonjour',
              'mx' => 'Hola');

while (<STDIN>) {
   ($name, $email) = split /\t/;
   ($suffix) = $email =~ /\.([a-z]+)$/;
   $greeting = $greetings{$suffix};
   $greeting = 'Hello' unless defined($greeting);

}
```

We extract the country code from the e-mail address (the pattern matches any letters following the final dot). We use that to look up a greeting, but default to 'Hello' if we didn't find one.

# Hive TRANSFORM Example (6)

```
#!/usr/bin/env perl

%greetings = ('de' => 'Hallo',
              'fr' => 'Bonjour',

wh
    $greeting = 'Hello' unless defined($greeting);
    print "$greeting $name\n";
}
```

Finally, we return our greeting as a single field by printing this value to standard output. If we had multiple fields, we'd simply separate each by tab characters when printing them here.

# Hive TRANSFORM Example (7)

- **Finally, here's the result of our transformation**

```
SELECT TRANSFORM(fname, email)
  USING 'hdfs:/dualcore/scripts/greeting.pl'
  AS greeting
  FROM employees;

Bonjour Antoine
Hallo Kai
Hola Pedro
```

# Using Scripts in a Secure Cluster

- **Note for sites using Sentry for security:**
  - Because of security risks, TRANSFORM is not allowed in clusters secured by Sentry
  - Instead use Hadoop Streaming instead of Hive to invoke the script

# Chapter Topics



- SerDes
- Data Transformation with Custom Scripts
- **User-Defined Functions**
- Parameterized Queries
- Conclusion
- Hands-On Exercise: Data Transformation with Hive

# Overview of User-Defined Functions (UDFs)

- **User-Defined Functions (UDFs) are custom functions**
  - Invoked with the same syntax as built-in functions

```
SELECT CALC_SHIPPING_COST(order_id, 'OVERNIGHT')
  FROM orders WHERE order_id = 5742354;
```

- **There are three types of UDFs in Hive**
  - Standard UDFs
  - User-Defined Aggregate Functions (UDAFs)
  - User-Defined Table Functions (UDTFs)

# Developing Hive UDFs

- **Hive UDFs are written in Java**
  - Currently no support for writing UDFs in other languages
  - Using `TRANSFORM` may be an alternative to UDFs
- **Open source UDFs are plentiful on the Web**
- **There are three steps for using a UDF in Hive**
  1. Copy the function's JAR file to HDFS
  2. Register the function
  3. Use the function in your query

**Attention Java Developers**
Cloudera now offers a free e-learning module "Writing UDFs for Hive"

`http://tiny.cloudera.com/dac14b`

# Example: Using a UDF in Hive (1)

- **Our example UDF was compiled from sources found in GitHub**
  - Popular Web site for many open source software projects
  - Project URL: `http://tiny.cloudera.com/dac14e`
- **We compiled the source and packaged it into a JAR file**
  - We have included a copy of it on your VM
- **Our example shows the `DATE_FORMAT` UDF in that JAR file**
  - Allows great flexibility in formatting date fields in output

# Example: Using a UDF in Hive (2)

- **First, copy the JAR file**
  - Same step as with a custom SerDe

```
$ hdfs dfs -put date-format-udf.jar /myscripts
```

- **Next, register the function and assign an alias**
  - The quoted value is the fully-qualified Java class for the UDF

```
CREATE TEMPORARY FUNCTION DATE_FORMAT
  AS 'com.nexr.platform.hive.udf.UDFDateFormat'
  USING JAR 'hdfs:/myscripts/date-format-udf.jar';
```

# Example: Using a UDF in Hive (3)

- **You may then use the function in your query**

```
SELECT order_date FROM orders LIMIT 1;
2011-12-06 10:03:35

SELECT DATE_FORMAT(order_date, 'dd-MMM-yyyy')
  FROM orders LIMIT 1;
06-Dec-2011

SELECT DATE_FORMAT(order_date, 'dd/mm/yy')
  FROM orders LIMIT 1;
06/12/11

SELECT DATE_FORMAT(order_date, 'EEEE, MMM d, yyyy')
  FROM orders LIMIT 1;
Tuesday, Dec 6, 2011
```

# Chapter Topics



- SerDes
- Data Transformation with Custom Scripts
- User-Defined Functions
- **Parameterized Queries**
- Conclusion
- Hands-On Exercise: Data Transformation with Hive

# Hive Variables (1)

- **Hive supports variable substitution**
  - Swaps a placeholder with a variable's literal value at run time
  - Variable names are case-sensitive
- **Set a named variable equal to some value:**

```
SET -v state=CA;
```

- **To use the variable's value in a HiveQL query:**

```
SELECT * FROM employees
  WHERE STATE = '${hivevar:state}';
```

# Hive Variables (2)

- **You can set variables when you invoke Beeline from the command line**
  - Eases repetitive queries by reducing need to modify HiveQL
- **For example, imagine that we have the following in `state.hql`**

```
SELECT COUNT(DISTINCT emp_id) FROM employees
  WHERE state = '${hivevar:state}';
```

- **This makes creating per-state reports easy:**

```
$ beeline -u … -hivevar state=CA -f state.hql
$ beeline -u … -hivevar state=NY -f state.hql
```

# Chapter Topics



- SerDes
- Data Transformation with Custom Scripts
- User-Defined Functions
- Parameterized Queries
- **Conclusion**
- Hands-on Exercise: Data Transformation with Hive

# Essential Points

- **SerDes govern how Hive reads and writes a table's records**
  - Specified (or defaulted) when creating a table
- **`TRANSFORM` processes records using an external program**
  - This can be written in nearly any language
- **UDFs are User-Defined Functions**
  - Custom logic that can be invoked just like built-in functions
- **Hive substitutes variable placeholders with literal values you assign**
  - This is done when you execute the query
  - Especially helpful with repetitive queries

# Chapter Topics



- SerDes
- Data Transformation with Custom Scripts
- User-Defined Functions
- Parameterized Queries
- Conclusion
- **Hands-On Exercise: Data Transformation with Hive**

# Hands-On Exercise: Data Transformation with Hive

- **In this Hands-On Exercise, you will use a Hive transform script and a UDF (User-Defined Function) together to estimate shipping costs on abandoned orders**
- **Please refer to the Hands-On Exercise Manual for instructions**

cloudera®
Ask Bigger Questions

# Choosing the Best Tool for the Job

Chapter 17