


BIG DATA ANALYTICS

Hive

Querying Data | User Defined Functions

<https://www.youtube.com/c/RASINENIMADANAMOHANA>

Hive: *Outline*

- Introduction
- Installing Hive
 - The Hive Shell
- An Example
- Running Hive
- Comparison with Traditional Databases
- HiveQL
- Tables
- Querying Data 
- User-Defined Functions
 - Writing a User Defined Functions
 - Writing a User Defined Aggregate Function

HiveQL: Querying Data

In this topic, we discuss how to use various forms of the **SELECT** statement to retrieve data from **Hive**.

1. **Sorting and Aggregating**
2. MapReduce Scripts
3. Joins
4. Subqueries
5. Views

HiveQL: Querying Data

1. Sorting and Aggregating

- **Sorting data** in **Hive** can be achieved by using a standard **ORDER BY** clause.
- **ORDER BY** performs a **parallel total sort** of the **input**.
- When a **globally sorted** result is not required-and in many cases it isn't-we can use **Hive's** nonstandard extension, **SORT BY**, instead.
- **SORT BY** produces a **sorted file** per **reducer**.
- In some cases, we want to control which **reducer** a **particular row** goes to-typically so we can perform some **subsequent aggregation**. This is what **Hive's** **DISTRIBUTE BY** clause does.

HiveQL: Querying Data

1. Sorting and Aggregating

Example to sort the weather dataset by year and temperature, in such a way as to ensure that all the rows for a given year end up in the same reducer partition:

```
hive> FROM records2
> SELECT year, temperature
> DISTRIBUTE BY year
> SORT BY year ASC, temperature DESC;
1949 111
1949 78
1950 22
1950 0
1950 -11
```

- A follow-on query would be able to use the fact that each year's temperatures were grouped and sorted (in descending order) in the same file.
- If the columns for SORT BY and DISTRIBUTE BY are the same, we can use CLUSTER BY as a shorthand for specifying both.

HiveQL: Querying Data

In this topic, we discuss how to use various forms of the **SELECT** statement to **retrieve data** from **Hive**.

1. Sorting and Aggregating
2. MapReduce Scripts
3. Joins
4. Subqueries
5. Views

HiveQL: Querying Data

2. MapReduce Scripts

- Using an approach like **Hadoop Streaming**, the **TRANSFORM**, **MAP**, and **REDUCE** clauses make it possible to invoke an **external script** or **program** from **Hive**.

Example: Suppose we want to use a **script** to filter out rows that don't meet some condition, which **removes poor-quality readings**.

Program: **Python script** to filter out **poor-quality** weather records

```
#!/usr/bin/env python
import re
import sys
for line in sys.stdin:
    (year, temp, q) = line.strip().split()
    if (temp != "9999" and re.match("[01459]", q)):
        print "%s\t%s" % (year, temp)
```

HiveQL: Querying Data

2. MapReduce Scripts

We can use the **script** as follows:

```
hive> ADD FILE /Users/tom/book-workspace/hadoop-book/ch17-hive/  
src/main/python/is_good_quality.py;  
hive> FROM records2  
  > SELECT TRANSFORM(year, temperature, quality)  
  > USING 'is_good_quality.py'  
  > AS year, temperature;  
1950 0  
1950 22  
1950 -11  
1949 111  
1949 78
```

Before running the query, we need to register the script with Hive. This is so Hive knows to ship the file to the Hadoop cluster.

HiveQL: Querying Data

2. MapReduce Scripts

- The **query itself streams** the year, temperature, and quality fields as a **tab-separated line** to the `is_good_quality.py` script, and **parses** the **tab-separated output** into year and temperature fields to form the **output** of the query.
- This example has **no reducers**. If we use a **nested form** for the **query**, we can specify a **map** and a **reduce** function. This time we use the **MAP** and **REDUCE** keywords, but **SELECT TRANSFORM** in both cases would have the same result.

```
FROM (  
  FROM records2  
  MAP year, temperature, quality  
  USING 'is_good_quality.py'  
  AS year, temperature) map_output  
REDUCE year, temperature  
USING 'max_temperature_reduce.py'  
AS year, temperature;
```

HiveQL: Querying Data

In this topic, we discuss how to use various forms of the **SELECT** statement to **retrieve data** from **Hive**.

1. Sorting and Aggregating
2. MapReduce Scripts
3. Joins
4. Subqueries
5. Views

HiveQL: Querying Data

3. Joins

- One of the **advantages** of using **Hive over** raw **MapReduce** is that **Hive** simplifies common tasks.
- For example, **joining tables** is much easier in **Hive** than in **MapReduce**, where it's quite complex.

Inner joins

- The **simplest** kind of **join** is the **inner join**, where each match in the **input tables** results in a **row** in the **output**.
- **Example:** Consider two small demonstration tables, **sales** (which lists the **names** of people and the **IDs** of the items they bought) and **things** (which lists the **item IDs** and their **names**):

HiveQL: Querying Data

3. Joins

Inner joins - Example

```
hive> SELECT * FROM sales;
```

```
Joe 2
```

```
Hank 4
```

```
Ali 0
```

```
Eve 3
```

```
Hank 2
```

```
hive> SELECT * FROM things;
```

```
2 Tie
```

```
4 Coat
```

```
3 Hat
```

```
1 Scarf
```

We can perform an inner join on the two tables as follows:

```
hive> SELECT sales.*, things.* FROM sales JOIN things ON (sales.id = things.id);
```

```
Joe 2 2 Tie
```

```
Hank 4 4 Coat
```

```
Eve 3 3 Hat
```

```
Hank 2 2 Tie
```

HiveQL: Querying Data

3. Joins

Inner joins - Example

- The table in the `FROM` clause (`sales`) is joined with the table in the `JOIN` clause (`things`), using the predicate in the `ON` clause. `Hive` only supports **equijoins**, which means that only **equality** can be used in the **join predicate**, which here matches on the **id** column in **both tables**.
- In `Hive`, we can **join on multiple columns** in the **join predicate** by specifying a **series of expressions**, separated by `AND` keywords. We can also join **more than two tables** by supplying additional `JOIN . . . ON . . .` clauses in the query. `Hive` is intelligent about trying to **minimize the number of MapReduce jobs** to perform the **joins**.

HiveQL: Querying Data

3. Joins

Inner joins - Example

- A single join is implemented as a single MapReduce job, but multiple joins can be performed in less than one MapReduce job per join if the same column is used in the join condition.
- We can see how many MapReduce jobs Hive will use for any particular query by prefixing it with the `EXPLAIN` keyword:

`EXPLAIN`

```
SELECT sales.*, things.* FROM sales JOIN things ON  
(sales.id = things.id);
```

HiveQL: Querying Data

3. Joins

Outer joins

- **Outer joins** allow us to find **nonmatches** in the **tables** being **joined**.

In the current example, when we performed an **inner join**, the **row** for **Ali** did not appear in the **output**, because the **ID** of the **item** she purchased was not present in the **things table**. If we change the **join type** to **LEFT OUTER JOIN**, the query will return a row for every row in the **left table (sales)**, even if there is no corresponding row in the table it is being joined to (**things**):

```
hive> SELECT sales.*, things.*
> FROM sales LEFT OUTER JOIN things ON (sales.id = things.id);

Joe 2 2 Tie
Hank 4 4 Coat
Ali 0 NULL NULL
Eve 3 3 Hat
Hank 2 2 Tie
```

HiveQL: Querying Data

3. Joins

Outer joins -Example

Notice that the row for Ali is now returned, and the columns from the things table are NULL because there is no match.

Hive also supports right outer joins, which reverses the roles of the tables relative to the left join.

In this case, all items from the things table are included, even those that weren't purchased by anyone (a scarf):

```
hive> SELECT sales.*, things.*  
> FROM sales RIGHT OUTER JOIN things ON (sales.id = things.id);
```

```
Joe 2 2 Tie  
Hank 2 2 Tie  
Hank 4 4 Coat  
Eve 3 3 Hat  
NULL NULL 1 Scarf
```


HiveQL: Querying Data

3. Joins

Outer joins -Example

Finally, there is a **full outer join**, where the **output** has a **row** for each row from **both tables** in the join:

```
hive> SELECT sales.*, things.*
> FROM sales FULL OUTER JOIN things ON (sales.id = things.id);
Ali 0 NULL NULL
NULL NULL 1 Scarf
Hank 2 2 Tie
Joe 2 2 Tie
Eve 3 3 Hat
Hank 4 4 Coat
```

HiveQL: Querying Data

3. Joins

Semi joins

Consider this **IN subquery**, which finds all the items in the **things** table that are in the **sales** table:

```
SELECT *  
FROM things  
WHERE things.id IN (SELECT id from sales);
```

We can also express it as follows:

```
hive> SELECT *  
      > FROM things LEFT SEMI JOIN sales ON (sales.id = things.id);  
2 Tie  
4 Coat  
3 Hat
```

HiveQL: Querying Data

3. Joins

Semi joins

There is a **restriction** that we must observe for **LEFT SEMI JOIN** queries: the **right table** (**sales**) may **appear only** in the **ON** clause.

It **cannot be referenced** in a **SELECT** expression, for example.

HiveQL: Querying Data

3. Joins

Map joins

Consider the **original** inner join again:

```
SELECT sales.*, things.*
```

```
FROM sales JOIN things ON (sales.id = things.id);
```

- If **one table** is **small enough** to fit in memory, as **things** is here, **Hive** can **load** it into **memory** to **perform the join** in each of the mappers. This is called a **map join**.
- The **job** to **execute this query** has **no reducers**, so this query would **not work** for a **RIGHT** or **FULL OUTER JOIN**, since **absence of matching** can be detected only in an **aggregating (reduce)** step across all the inputs.

HiveQL: Querying Data

3. Joins

Map joins

- Map joins can take advantage of **bucketed tables**, since a **mapper** working on a **bucket** of the **left table** needs to **load** only the **corresponding buckets** of the **right table** to **perform the join**.
- The **syntax** for the **join** is the **same** as for the **in-memory case** shown earlier; however, we also need to **enable** the **optimization** with the following:

```
SET hive.optimize.bucketmapjoin=true;
```

HiveQL: Querying Data

In this topic, we discuss how to use various forms of the **SELECT** statement to **retrieve data** from **Hive**.

1. Sorting and Aggregating
2. MapReduce Scripts
3. Joins
4. **Subqueries**
5. Views

HiveQL: Querying Data

4. Subqueries

- A **subquery** is a **SELECT** statement that is **embedded** in another **SQL** statement.
- **Hive** has **limited support** for **subqueries**, permitting a **subquery** in the **FROM** clause of a **SELECT** statement, or in the **WHERE** clause in certain cases.
- The following query finds the **mean maximum temperature** for **every year** and **weather station**:

```
SELECT station, year, AVG(max_temperature)
FROM (
    SELECT station, year, MAX(temperature) AS max_temperature
    FROM records2
    WHERE temperature != 9999 AND quality IN (0, 1, 4, 5, 9)
    GROUP BY station, year
) mt
GROUP BY station, year;
```

HiveQL: Querying Data

4. Subqueries

- The **FROM subquery** is used to find the maximum temperature for each station/date combination, and then the **outer query** uses the **AVG** aggregate function to find the average of the maximum temperature readings for each station/date combination.
- The **outer query** accesses the **results of the subquery** like it does a **table**, which is why the **subquery** must be given an **alias** (`mt`). The **columns of the subquery** have to be given **unique names** so that the **outer query** can refer to them.

HiveQL: Querying Data

In this topic, we discuss how to use various forms of the **SELECT** statement to **retrieve data** from **Hive**.

1. Sorting and Aggregating
2. MapReduce Scripts
3. Joins
4. Subqueries
5. Views

HiveQL: Querying Data

4. Views

What is a View?

- A “virtual table” created by a `SELECT` statement
- Presents data differently from its storage format
- Used for simplifying or aggregating data and restricting access

How Views Work:

- Not stored on disk; executed when queried
- Can be materialized manually by creating a new table if needed

HiveQL: Querying Data

4. Views

Creating Views:

1. View for Valid Records:

```
CREATE VIEW valid_records AS
SELECT *
FROM records2
WHERE temperature != 9999 AND quality IN (0, 1, 4, 5, 9);
```

2. View for Max Temperatures

```
CREATE VIEW max_temperatures (station, year, max_temperature)
AS
SELECT station, year, MAX(temperature)
FROM valid_records
GROUP BY station, year;
```

HiveQL: Querying Data

4. Views

Using Views:

Query example:

```
SELECT station, year, AVG(max_temperature)
FROM max_temperatures
GROUP BY station, year;
```

- Views are read-only
- Use `SHOW TABLES` to list **views** and `DESCRIBE EXTENDED view_name` for details
- **Hive** optimizes queries involving **views** to match queries without views

Hive: *Outline*

- Introduction
- Installing Hive
 - The Hive Shell
- An Example
- Running Hive
- Comparison with Traditional Databases
- HiveQL
- Tables
- Querying Data
- User-Defined Functions 
 - Writing a User Defined Functions
 - Writing a User Defined Aggregate Function

HiveQL: User-Defined Functions

- Extend **Hive's** functionality when **built-in functions** aren't enough.
- By allowing us to write a **user-defined function (UDF)**, **Hive** makes it easy to plug-in our own processing code and invoke it from a **Hive query**.

Types of UDFs:

1. UDF (Regular):

- Operates on a **single row**, produces a **single row**.
- **Example:** Mathematical functions like **ABS()**, **ROUND()**.

2. UDAF (User-Defined Aggregate Function):

- Processes **multiple rows**, returns a **single row**.
- **Example:** Aggregate functions like **COUNT()**, **MAX()**.

3. UDTF (User-Defined Table-Generating Function):

- Takes **one row**, returns **multiple rows (table output)**.
- **Example:** **Explode** function to split arrays into individual rows.

HiveQL: User-Defined Functions

Example: UDTF with `explode()`:

1. Input Data:

Consider a **table** with a **single column**, `x`, which contains **arrays of strings**.

```
CREATE TABLE arrays (x ARRAY<STRING>)
```

```
ROW FORMAT DELIMITED
```

```
FIELDS TERMINATED BY '\001'
```

```
COLLECTION ITEMS TERMINATED BY '\002';
```

Notice that the **ROW FORMAT** clause specifies that the entries in the array are **delimited** by **Ctrl-B** characters.

The **example file** that we are going to load has the following contents, where **^B** is a representation of the **Ctrl-B** character to make it suitable for printing:

```
a^Bb
```

```
c^Bd^Be
```

HiveQL: User-Defined Functions

Example: UDTF with explode():

2. Query:

After running a `LOAD DATA` command, the following query confirms that the data was loaded correctly:

```
hive> SELECT * FROM arrays;
```

```
["a", "b"]
```

```
["c", "d", "e"]
```


HiveQL: User-Defined Functions

Example: UDTF with `explode()`:

2. Query:

Next, we can use the `explode` UDTF to transform this table. This function releases a row for each entry in the array, so in this case the type of the output column `y` is `STRING`. The result is that the table is flattened into **five rows**:

```
hive> SELECT explode(x) AS y FROM arrays;  
a  
b  
c  
d  
e
```

`SELECT` statements using UDTFs have **some restrictions** (e.g., they cannot retrieve additional column expressions), which make them less useful in practice. For this reason, **Hive** supports `LATERAL VIEW` queries, which are more powerful.

HiveQL: User-Defined Functions

Writing a UDF:

- To illustrate the process of **writing** and **using** a UDF, we'll write a simple UDF to **trim characters** from the **ends of strings**.
- **Hive** already has a **built-in function** called **trim**, so we'll call ours **strip**.
- The **code** for the **Strip Java class** is shown in below **Example**.

HiveQL: User-Defined Functions

Writing a UDF:

Example: *A UDF for stripping characters from the ends of strings*

```
package com.hadoopbook.hive;
import org.apache.commons.lang.StringUtils;
import org.apache.hadoop.hive ql.exec.UDF;
import org.apache.hadoop.io.Text;
public class Strip extends UDF {
    private Text result = new Text();
    public Text evaluate(Text str) {
        if (str == null) {return null;}
        result.set(StringUtils.strip(str.toString()));
        return result;
    }
    public Text evaluate(Text str, String stripChars) {
        if (str == null) return null;
        result.set(StringUtils.strip(str.toString(), stripChars));
        return result;
    }
}
```

HiveQL: User-Defined Functions

Writing a UDF:

A UDF must satisfy the following two properties:

- A UDF must be a subclass of `org.apache.hadoop.hive.q1.exec.UDF`.
- A UDF must implement at least one `evaluate()` method.

Usage in Hive:

- To use the UDF in **Hive**, we first need to package the compiled **Java class** in a **JAR** file.
- Next, we **register** the **function** in the **metastore** and give it a name using the **CREATE FUNCTION** statement:

```
CREATE FUNCTION strip AS 'com.hadoopbook.hive.Strip'  
USING JAR '/path/to/hive-examples.jar';
```

HiveQL: User-Defined Functions

Writing a UDF:

Usage in Hive: cont'd.

- When using Hive locally, a local file path is sufficient, but on a cluster we should copy the JAR file into HDFS and use an HDFS URI in the USING JAR clause.
- The UDF is now ready to be used, just like a built-in function:

```
hive> SELECT strip(' bee ') FROM dummy;
```

```
bee
```

```
hive> SELECT strip('banana', 'ab') FROM dummy;
```

```
nan
```

Notice that the UDF's name is not case sensitive:

```
hive> SELECT STRIP(' bee ') FROM dummy;
```

```
bee
```

If we want to remove the function, use the DROP FUNCTION statement:

```
DROP FUNCTION strip;
```

HiveQL: User-Defined Functions

Writing a UDAF (User-Defined Aggregate Function):

- An **aggregate function** is more difficult to write than a regular UDF.
- **Values** are **aggregated** in **chunks** (potentially across many tasks), so the implementation has to be capable of combining partial aggregations into a final result.
- The **code** to achieve this is best explained by **example**, so let's look at the implementation of a **simple UDAF** for **calculating the maximum of a collection of integers**.

HiveQL: User-Defined Functions

Writing a UDAF (User-Defined Aggregate Function):

Example: A *UDAF* for calculating the maximum of a collection of integers

```
import org.apache.hadoop.hive.ql.exec.UDAF;
import org.apache.hadoop.hive.ql.exec.UDAFEvaluator;
import org.apache.hadoop.io.IntWritable;
public class Maximum extends UDAF {
    public static class MaximumIntUDAFEvaluator implements UDAFEvaluator
    {
        private IntWritable result;
        public void init() {
            result = null;
        }
    }
}
```

HiveQL: User-Defined Functions

Writing a UDAF (User-Defined Aggregate Function):

Example: A *UDAF* for calculating the maximum of a collection of integers (cont'd.)

```
public boolean iterate(IntWritable value) {
    if (value == null) return true;
    if (result == null) result = new IntWritable(value.get());
    else result.set(Math.max(result.get(), value.get()));
    return true;
}
public IntWritable terminatePartial() {
    return result;
}
public boolean merge(IntWritable other) {
    return iterate(other);
}
public IntWritable terminate() {
    return result;
}
}
```


HiveQL: User-Defined Functions

Writing a UDAF (User-Defined Aggregate Function):

Example: A *UDAF* for calculating the maximum of a collection of integers (cont'd.)

- The class structure is slightly different from the one for UDFs.
- A UDAF must be a subclass of `org.apache.hadoop.hive.q1.exec.UDAF` (note the "A" in UDAF) and contain one or more nested static classes implementing `org.apache.hadoop.hive.q1.exec.UDAFEvaluator`.
- In this example, there is a **single nested class**, `MaximumIntUDAFEvaluator`, but we could add more evaluators, such as `MaximumLongUDAFEvaluator`, `MaximumFloatUDAFEvaluator`, and so on, to provide overloaded forms of the UDAF for **finding** the **maximum** of a **collection** of **longs**, **floats**, and so on.

HiveQL: User-Defined Functions

Writing a UDAF (User-Defined Aggregate Function):

Example: A *UDAF* for calculating the maximum of a collection of integers (*cont'd.*)

An **evaluator** must implement **five methods**, described in turn here (the flow is illustrated in **below Figure**):

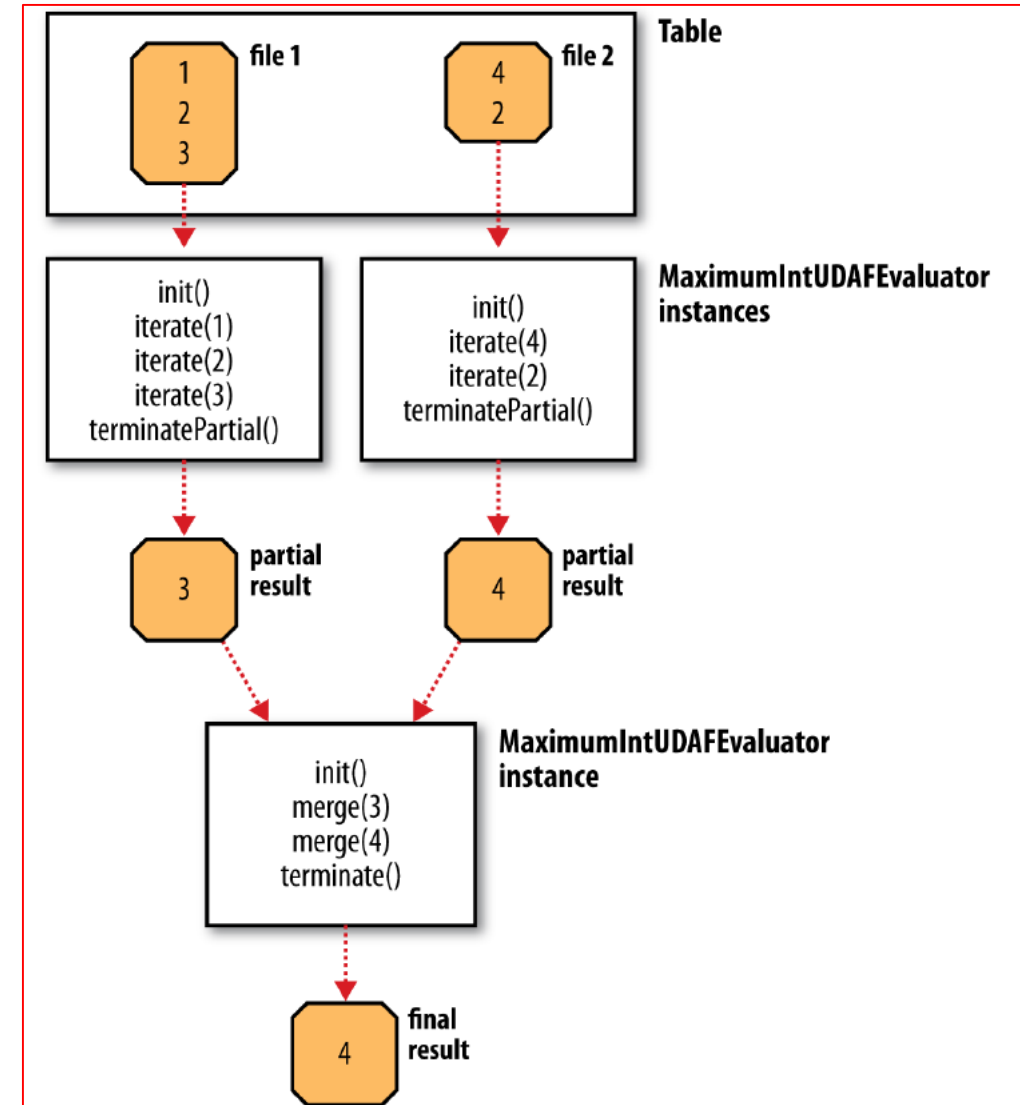


Figure. Data flow with partial results for a UDAF

HiveQL: User-Defined Functions

Writing a UDAF (User-Defined Aggregate Function):

Example: A *UDAF* for calculating the maximum of a collection of integers (cont'd.)

1. `init()` Method:

Initializes the evaluator and sets the internal state.

For `MaximumIntUDAFEvaluator`, the internal state (`IntWritable` object) is initialized to `null`, indicating no values have been aggregated yet.

2. `iterate()` Method:

Called for each new value to be aggregated. It updates the evaluator's state based on the aggregation logic. If the input is not `null`, the state is updated to the maximum of the current state and the new value. Always returns `true` to signify the input was valid.

HiveQL: User-Defined Functions

Writing a UDAF (User-Defined Aggregate Function):

Example: A *UDAF* for calculating the maximum of a collection of integers (cont'd.)

3. terminatePartial() Method:

Returns the current state during a partial aggregation, encapsulated as an `IntWritable` object, which holds either the maximum value seen so far or `null` if no values have been processed.

4. merge() Method:

Combines partial aggregations by delegating to the `iterate()` method, ensuring the internal state is updated to reflect the combination of both states.

5. terminate() Method:

Returns the final result of the aggregation, which is the maximum value encountered or `null` if no valid values were processed.

HiveQL: User-Defined Functions

Writing a UDAF (User-Defined Aggregate Function):

Example: A *UDAF* for calculating the maximum of a collection of integers (cont'd.)

Usage of UDAF `Maximum`:

```
hive> CREATE TEMPORARY FUNCTION maximum AS  
'com.hadoopbook.hive.Maximum';
```

```
hive> SELECT maximum(temperature) FROM records;
```

```
111
```