QUERYING HIVE TABLES

Querying Hive Tables

- **Writing Hive Queries**
- Views
- UDFs
- Join Optimizations
- Windowing and Analytics

Running Hive

- Hive queries can be submitted from any possible client mentioned before
- In Hive CLI (and Beeline), Hive commands can be run in two modes
  - Interactive
  - Batch
- The batch mode is by passing an '-e' option to pass a query to be executed, or an '-f' option to pass a file of valid Hive statements to be executed

```
$ hive -e "SELECT * FROM users WHERE city = '34'"
```

Running Hive

- Configurations can be passed
  - using the set statement,
  - Setting the hiveconf

```
$ hive --hiveconf <property=value>
```

- Variable substitution is possible, variables in the script in the form ${*variable_name*} can be set using

```
$ hive --hivevar <variable=value>
```

Running Hive

- Configurations can be passed
  - using the set statement,
  - Setting the hiveconf

```
$ hive --hiveconf <property=value>
```

- Variable substitution is possible, variables in the script in the form ${*variable_name*} can be set using

```
$ hive --hivevar <variable=value>
```


analyticscenter

Running Hive

- $HIVE_HOME/bin/.hiverc file is loaded every time the CLI is invoked

Writing Hive Queries

- HiveQL provides basic SQL-like operations, working on tables (and partitions), including:
  - Filtering records (WHERE clause)
  - Projecting Columns (SELECT clause)
  - Joining
  - Aggregating (and GROUP BY + agg, of course)
  - Running MapReduce jobs with custom scripts

Writing Hive Queries

- HiveQL provides basic SQL-like operations, working on tables (and partitions), including:
  - Joining on equality conditions

```
SELECT t1.c1, t2.c2
FROM t1 LEFT OUTER JOIN t2 ON (t1.c1 = t2.c1)
WHERE c3 RLIKE '^f.*r$'
```

```
SELECT t1.c1, t2.c2
FROM t1 JOIN t2 ON (t1.c1 = t2.c1)
WHERE c3 RLIKE '^f.*r$'
```

```
SELECT t1.c1, t2.c2
FROM t1 LEFT SEMI JOIN t2 ON (t1.c1 = t2.c1)
WHERE c3 RLIKE '^f.*r$'
```

Writing Hive Queries

- HiveQL provides basic SQL-like operations, working on tables (and partitions), including:
  - Aggregating (and GROUP BY + agg, of course)

```
SELECT COUNT(DISTINCT(c3)) from t
```

```
SELECT c1, SUM(c2) s from t GROUP BY c1 ORDER BY s;
```


analyticscenter

Writing Hive Queries

- HiveQL provides basic SQL-like operations, working on tables (and partitions), including:
  - Aggregates returning a composite type

```
SELECT ngrams(sentences(lower(c3)), 2, 100) from t
```

```
SELECT histogram_numeric(age) FROM users GROUP BY gender
```

Writing Hive Queries

- HiveQL provides basic SQL-like operations, working on tables (and partitions), including:
  - Running MapReduce jobs with custom scripts

```
FROM (
  FROM t1
  MAP t1.c1, t1.c2
  USING 'map_script'
  AS f1, f2
  CLUSTER BY f1) map_output

INSERT OVERWRITE TABLE reduce_output
  REDUCE map_output.f1, map_output.f2
  USING 'reduce_script'
  AS k1, k2;
```

Writing Hive Queries

- HiveQL provides basic SQL-like operations, working on tables (and partitions), including:
  - Notice the usage of the CLUSTER BY, which would group and sort the resulting map_output based on f1 (this is equivalent to shuffling&sorting in MapReduce)

```
FROM (
  FROM t1
  MAP t1.c1, t1.c2
  USING 'map_script'
  AS f1, f2
  CLUSTER BY f1) map_output

INSERT OVERWRITE TABLE reduce_output
  REDUCE map_output.f1, map_output.f2
  USING 'reduce_script'
  AS k1, k2;
```

Writing Hive Queries

- Hive supports sub-queries

```
SELECT *
FROM A
WHERE A.a IN (SELECT foo FROM B);
```

```
SELECT a
FROM T1
WHERE EXISTS (SELECT b FROM T2 WHERE T1.x = T2.y)
```

Writing Hive Queries

- Hive supports sub-queries

```
SELECT col
FROM (
  SELECT a+b AS col
  FROM t1
) t2
```

```
SELECT t3.col
FROM (
  SELECT a+b AS col
  FROM t1
  UNION ALL
  SELECT     AS col
  FROM t2
) t3
```

Writing Hive Queries

- A Hive function can be a table-generating function
- This is usually used to flatten a column of a composite type

```
SELECT explode(ngrams(sentences(lower(c3)), 2, 100)) from t
```

Writing Hive Queries

- We usually want to join the ‘other’ columns to the table created by explode, but it is not allowed
- The trick is to use the LATERAL VIEW clause

Writing Hive Queries

```
SELECT city, conference
FROM
conferences LATERAL VIEW explode(confs) t2 AS conference;
```

conferences

city	confs
Istanbul	[‘SmartCon’, ‘DDD’]
NY	[‘Strata’, ‘IEEE’, ‘DS’]
Brussels	[‘Hadoop Summit’]

city	conference
Istanbul	‘SmartCon’
Istanbul	‘DDD’
NY	‘Strata’
NY	‘IEEE’
NY	‘DS’
Brussels	‘Hadoop Summit’

analyticscenter

Writing Hive Queries

- Here, the table created after explode is named t2 with a column called conference, and we joined this with the original table using the LATERAL VIEW CLAUSE

```
SELECT city, conference
FROM
conferences LATERAL VIEW explode(confs) t2 AS conference;
```


analyticscenter

Lab

Querying Hive Tables

Demo

Using Table Generating Functions

Querying Hive Tables

- Writing Hive Queries
- **Views**
- UDFs
- Join Optimizations
- Windowing and Analytics

Views

- We can create views in Hive, just like creating tables as select statements
- View is a logical object, which does not have an associated storage (they are not materialized)
- The rows of a view are only evaluated when the view is referenced in a query
- If the underlying table is dropped (or changed in an incompatible way), attempts to query the view would fail
- Views are read-only, they cannot be targets of LOAD/INSERT statements

Views

```
$ hive

hive> CREATE VIEW names
      AS
      SELECT DISTINCT(name) from users;

hive> OK
hive> SELECT COUNT(*) from names;
```

Querying Hive Tables

- Views Windowing and Analytics
- **UDFs**
- Join Optimizations
- Windowing and Analytics

UDFs

- UDFs are just the non-native versions of Hive's built-in functions
- They can be:
  - Standard UDFs
  - User Defined Aggregate Functions (UDAF)
  - User Defined Table-Generating Functions (UDTF)

UDFs

- To use a UDF within a Hive query, the containing Java Archive (jar) should be add to the classpath of all nodes making up the MapReduce cluster; and the function to be used should be created

```
$ hive

hive> add jar myjar.jar;
hive> CREATE FUNCTION myfunc AS 'fully.classified.MyClass'
```

- Alternatively

```
$ hive

hive> CREATE FUNCTION myfunc AS 'fully.classified.MyClass'
USING JAR 'hdfs://path/to/myjar'
```

UDFs

- Standard UDFs operate on one row at a time
- To define a UDF, the user writes a class that extends `org.apache.hadoop.hive.ql.exec.UDF` base class, adds and uses it in Hive

```
package com.example.hive.udf;

import org.apache.hadoop.hive.ql.exec.UDF;
import org.apache.hadoop.io.Text

public final class Lower extends UDF {
  public Text evaluate(final Text s) {
    if (s == null) { return null; }
    return new Text(s.toString().toLowerCase());
  }
}
```

UDFs

```
package com.example.hive.udf;

import org.apache.hadoop.hive.ql.exec.UDF;
import org.apache.hadoop.io.Text;

public final class Lower extends UDF {
  public Text evaluate(final Text s) {
    if (s == null) { return null; }
    return new Text(s.toString().toLowerCase());
  }
}
```

```
$ hive

hive> CREATE FUNCTION my_lower AS ‘Lower’ USING JAR ‘hdfs://
path/to/myjar’;

hive> SELECT my_lower(c1) FROM t1;
```

UDFs

- A standard UDF can also return an **`ArrayList`** for example, and then multiple columns can be projected from a column

UDTFs

- A User-Defined Table Generating Function generates multiple output rows for a single row (similar to **`explode(arr)`**)
- To define a UDTF, the user writes a class that extends `org.apache.hadoop.hive.ql.udf.generic.GenericUDTF` base class, adds and uses it in Hive

UDTFs

- The `GenericUDTF is extended as follows:`
  - The user defines the schema of the returning table in the **`initialize`** method
  - The row generating logic is put into the **`process`** method, calling **`forward`** per row

UDTFs

```
//Standard Java syntax is not used –for brevity
package com.example.hive.udf;

import org.apache.hadoop.hive.ql.exec.UDF;
import org.apache.hadoop.io.Text;

public final class Words extends GenericUDTF {
  private StringObjectInspector stringOI = null
  public StructObjectInspector initialize (ObjectInspector[]
                                          argOIs){
  stringOI = (PrimitiveObjectInspector) args[0];
  fieldNames = new ArrayList<String>(1);
  fldOIs = new ArrayList<ObjectInspector>(1);
  fieldNames.add(“word”)
  fldOIs.add(PrimitiveObjectInspectorFactory.
           javaStringObjectInspector);
  return ObjectInspectorFactory.
         getStandardStructObjectInspector(fieldNames, fldOIs);
  }
  public void process(Object[] record) {...}
}
```

UDTFs

```
//Standard Java syntax is not used –for brevity
package com.example.hive.udf;

import org.apache.hadoop.hive.ql.exec.UDF;
import org.apache.hadoop.io.Text;

public final class Words extends GenericUDTF {
  private StringObjectInspector stringOI = null
  public StructObjectInspector initialize {...}

  public void process(Object[] record){
    document = (String)
    stringOI.getPrimitiveJavaObject(record[0]);

    String[] tokens = document.split(“\\s+”);
      for (String token : tokens) {
        forward(new Object[] {token});
      }
  }
}
```

UDAFs

- A User-Defined Aggregate Function generates a single value (still might be composite) from a collection of rows
- To define a UDAF, the user writes a class that extends `org.apache.hadoop.hive.ql.udf.generic.GenericUDAFResolver2` base class, adds and uses it in Hive
- Notice that a UDAF can run on all rows of a table
- Similar to the Pig's Algebraic interface, Hive UDAFs should define the behavior in:
  - Processing individual records
  - Combining local results
  - Merging combined results
- Hive then use this behavior to run MapReduce jobs from the UDAF code

UDAFs

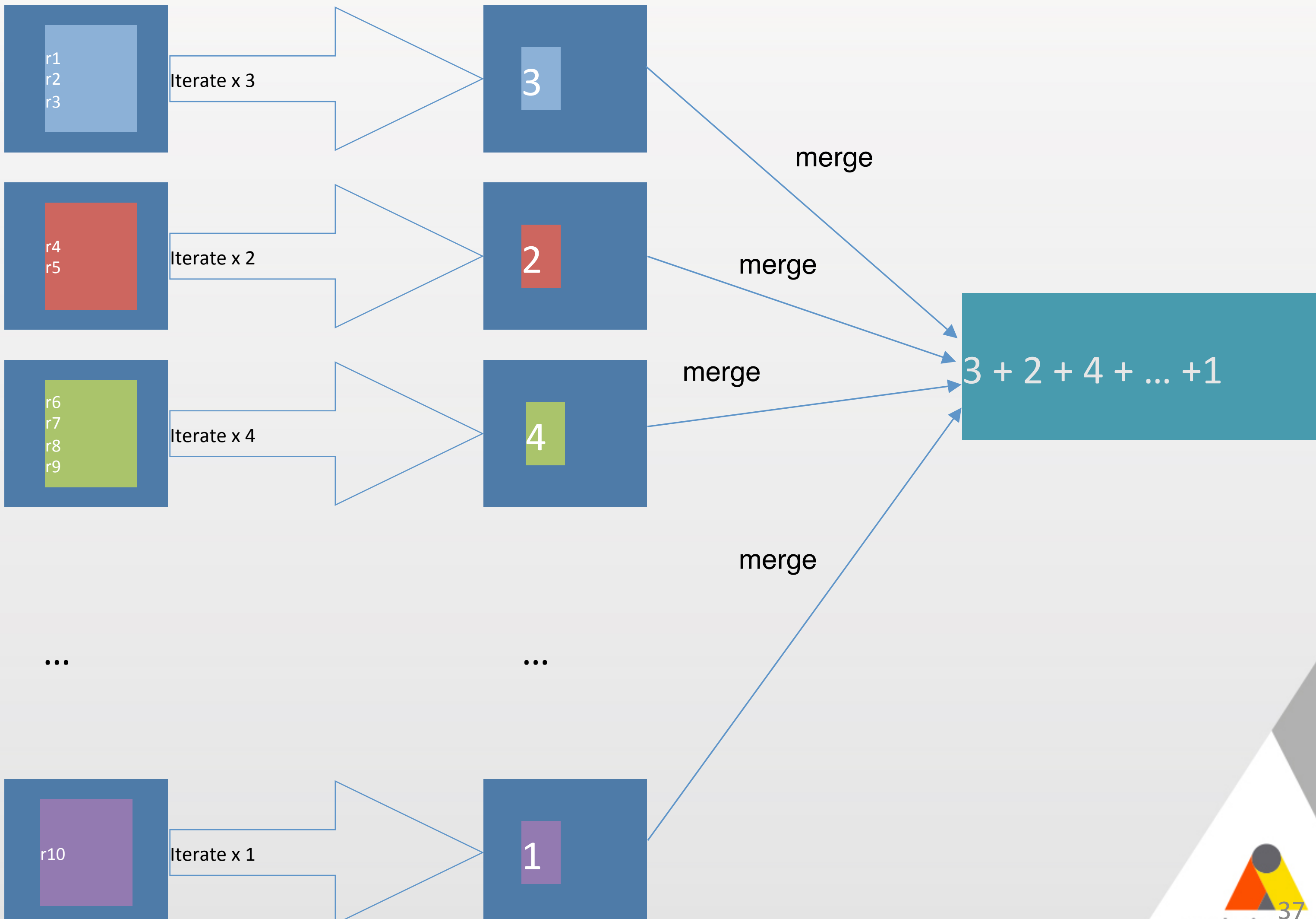
- We write a **`GenericUDAFEvaluator`** for defining a UDAF
- The **`GenericUDAFEvaluator`** class has **`init`**, **`getNewAggregationBuffer`**, **`iterate`**, **`terminatePartial`**, **`merge`**, and **`terminate`** methods for us to implement:
  - **`init`**: The initialization goes here (the result is initiated with a zero-value, and the output type information is returned)
  - **`getNewAggregationBuffer`**: This returns an object with a zero value as the aggregation estimate, new values would be accumulated into this object as they are processed

analyticscenter

UDAFs

- We write a **`GenericUDAFEvaluator`** for defining a UDAF
- The **`GenericUDAFEvaluator`** class has **`init`**, **`getNewAggregationBuffer`**, **`iterate`**, **`terminatePartial`**, **`merge`**, and **`terminate`** methods for us to implement:
  - **`iterate`**: Here, what we would have done normally in a map-call is defined (This becomes the Mapper#map) and accumulated into the aggregation buffer
  - **`terminatePartial`**: This is called after records of a split are processed (for each InputSplit), to prepare the partial outputs (an aggregation buffer is converted to an object, might contain a combine logic)
  - **`merge`**: This is where the reduce logic goes, partial results are merged into the current aggregate

UDAFs


r1
r2
r3
Iterate x 3
3
merge
r4
r5
Iterate x 2
2
merge
r6
r7
r8
r9
Iterate x 4
4
merge
3 + 2 + 4 + … +1
merge
…
…
r10
Iterate x 1
1

Querying Hive Tables

- Writing Hive Queries
- Views
- UDFs
- **Join Optimizations**
- Windowing and Analytics

Join Optimizations

- As defined in Big Data processing concepts, in a MapReduce-like environment, join operations can greatly be optimized depending on:
  - Size of the data sets
  - How data sets are partitioned

Join Optimizations

- Hive supports multiple join implementations
  - Standard shuffle joins
  - Map joins (broadcast joins)
  - Bucket map joins
  - Bucket sort merge map joins
  - Skew joins

Join Optimizations: Map Joins

- Map joins in Hive are broadcast joins, that is, when one of the tables is small
  - The small table is replicated in each node
  - Every input split of the large table is joined with the small table locally (map-side)
  - No reduce job is required

Join Optimizations: Map Joins

- To tell Hive to run Map Join, we can give Hive a hint:

```
SELECT /*+ MAPJOIN(small_table) */
large_table.*, small_table.*
FROM
large_table JOIN small_table
ON (large_table.c1 = small_table.c1)
```

- Alternatively, we set **`hive.auto.convert.join`** configuration parameter **`true`**

```
set hive.auto.convert.join=true;
SELECT large_table.*, small_table.*
FROM
large_table JOIN small_table
ON (large_table.c1 = small_table.c1)
```

Join Optimizations: Bucket Map Joins

- Consider the following join scenario:
  - All tables are bucketed on the join columns
  - A table's bucket size is a multiple of the other's (e.g. one table has 10 buckets, the other has 20 buckets)
  - If one mapper is run for each bucket of one of the tables, and only the required buckets of the other table are replicated to the relevant mappers, a map join can still be performed
  - This is called a bucket map join in Hive

Join Optimizations: Bucket Map Joins

- To tell Hive to run Bucket Map Join, we can give Hive a hint:

```
SELECT /*+ MAPJOIN(t2) */
t1.*, t2.*
FROM
t1 JOIN t2
ON (t1.c1 = t2.c1)
```

- Alternatively, we set `hive.optimize.bucketmapjoin` configuration parameter `true`

```
set hive.optimize.bucketmapjoin=true;
SELECT t1.*, t2.*
FROM
t1 JOIN t2
ON (t1.c1 = t2.c1)
```


analyticscenter

Join Optimizations: Sort Merge Bucket Map Joins

- If in the bucketed table scenario, the data in the buckets are also sorted by the join column, the map join can still be performed, and more efficiently by exploiting the fact that data are sorted
- Join is performed simply by merging corresponding buckets

Join Optimizations: Sort Merge Bucket Map Joins

- In addition to the **`hive.optimize.bucketmapjoin`**, we set **`hive.optimize.bucketmapjoin.sortmerge`** configuration parameter **`true`**, and the **`hive.input.format`** parameter **`org.apache.hive.ql.io.BucketizedIHiveInputFormat`**

```
set hive.optimize.bucketmapjoin=true;
set hive.optimize.bucketmapjoin.sortmerge=true;
set hive.input.format=org.apache.hive.ql.io.BucketizedHiveInputFormat;

SELECT t1.*, t2.*
FROM
t1 JOIN t2
ON (t1.c1 = t2.c1)
```

Querying Hive Tables

- Writing Hive Queries
- Views
- UDFs
- Join Optimizations
- **Windowing and Analytics**

Windowing and Analytics

- OVER a window specification, Hive supports
  - Standard Aggregations such as COUNT
  - LEAD, LAG, and FIRST_VALUE, LAST_VALUE
  - Analytics Functions
    - RANK
    - ROW_NUMBER
    - DENSE_RANK
    - CUME_DIST
    - PERCENT_RANK
    - NTILE

analyticscenter

Windowing and Analytics

- A window is specified by
  - A PARTITION BY (essentially flat version of GROUP BY) statement
  - PARTITION BY might not be necessary depending on the function used over the window
- ORDER BY can be run within a window
- The window size can be restricted with the **preceding** and **following** rows specification:

```
ROWS BETWEEN (CURRENT ROW)|(UNBOUNDED|<num>) PRECEDING
     AND     (CURRENT ROW)|(UNBOUNDED|<num>) FOLLOWING
```

Windowing and Analytics

- Example queries with windowing functions:

```
SELECT t.*, LEAD(c2)
OVER(ORDER BY c1)
FROM t;
```

```
SELECT t.*, LAG(c2, 3)
OVER(ORDER BY c1)
FROM t;
```

```
SELECT t.*, RANK()
OVER (PARTITION BY c1 ORDER BY c2) FROM t
```

```
SELECT *, SUM(amount)
OVER(PARTITION BY year(date) ORDER BY date
ROWS UNBOUNDED PRECEDING AND CURRENT ROW)
FROM sales;
```

Windowing and Analytics

```
SELECT *,
SUM(amount) OVER(PARTITION BY year(date) ORDER BY date
ROWS UNBOUNDED PRECEDING AND CURRENT ROW) AS s,
RANK() OVER(PARTITION BY year(date) ORDER BY date) AS r
FROM sales;
```

sales

date	amount
“2014-12-10”	10
“2014-11-20”	23
“2013-10-01”	5
“2013-04-20”	10
“2015-01-01”	33
“2014-07-01”	11

date	amount	s	r
“2013-04-20”	10	10	1
“2013-10-01”	5	15	2
“2014-07-01”	11	11	1
“2014-11-20”	23	34	2
“2014-12-10”	10	44	3
“2015-01-01”	33	33	1

Demo

Using Windowing Functions

Querying Hive Tables

End of Chapter