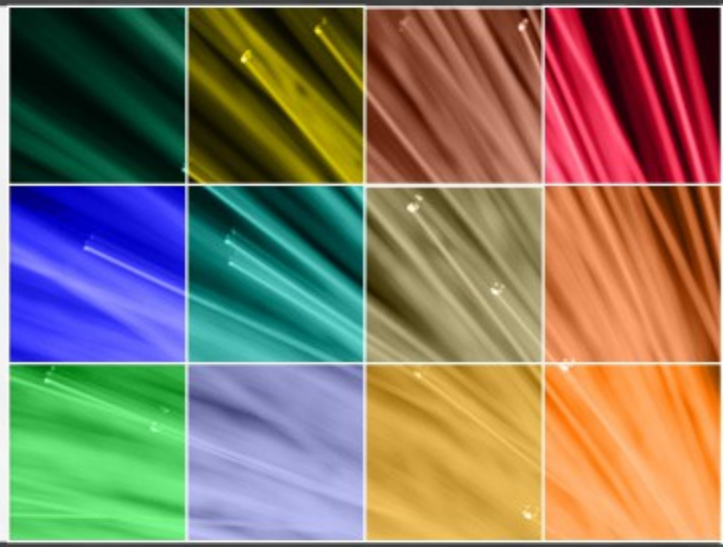
SYNTEL
Consider IT Done®

Introduction to Hive



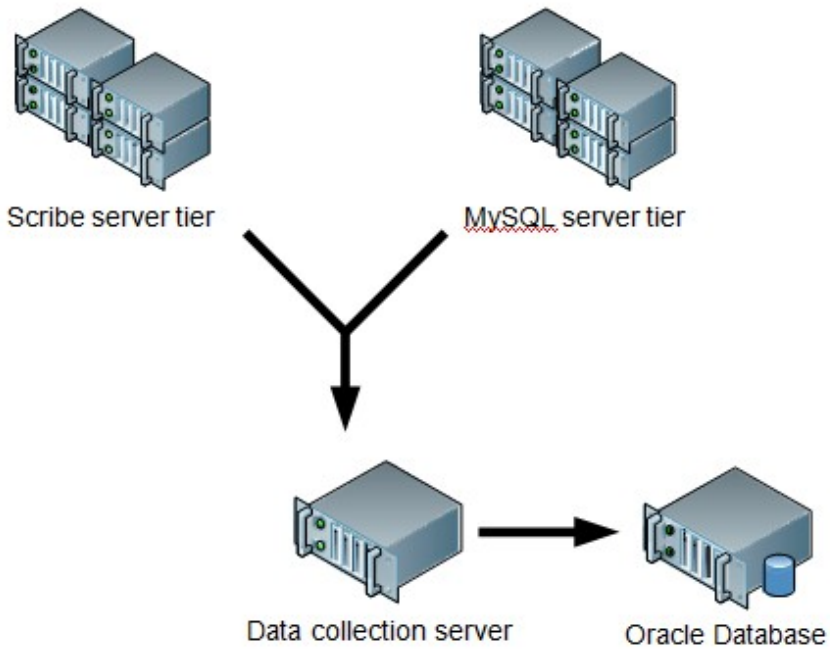
strategic offerings group

Outline

- Motivation
- Overview
- Data Model
- Working with Hive
- DDL Operations
- Custom mapper scripts
- User Defined Functions
- Wrap up & Conclusions

Background

- Started at Facebook
- Data was collected by nightly cron jobs into Oracle DB
- “ETL” via hand- coded python
- Grew from 10s of GBs (2006) to 1 TB/day new data (2007),now 10x that.



Hive Applications

- Log processing
- Text mining
- Document indexing
- Customer-facing business intelligence (e.g., Google Analytics)
- Predictive modeling, hypothesis testing

Hive Components

- Shell: allows interactive queries like MySQL shell connected to database
- - Also supports web and JDBC clients
- Driver: session handles,fetch,execute
- Compiler:parse,plan,optimize
- Execution engine: DAG of stages (M/R, HDFS, or metadata)
- Metastore: schema, location in HDFS,SerDe

Data Model

- **Tables**
  - **Typed columns (int, float, string, boolean)**
  - **Also, list: map (for JSON-like data)**
- **Partitions**
  - **e.g., to range-partition tables by date**
- **Buckets**
  - **Hash partitions within ranges (useful for sampling, join optimization)**

Metastore

- Database: namespace containing a set of tables
- Holds table definitions (column types, physical layout)
- Partition data
- Uses JPOX ORM for implementation; can be stored in Derby, MySQL, many other relational databases

Physical Layout

- Warehouse directory in HDFS
  - e.g., /user/hive/warehouse
- Tables stored in subdirectories of warehouse
  - Partitions form subdirectories of tables
- Actual data stored in flat files
  - Control char-delimited text, or SequenceFiles
  - With custom SerDe, can use arbitrary format

Starting the Hive shell

- Start a terminal and run

  $hive

- Should see a prompt like:

  hive>

Creating tables

- **hive> SHOW TABLES;**
- **hive> CREATE TABLE shakespeare (freq INT, word STRING) ROW FORMAT DELIMITED FIELDS TERMINATED BY „\t" STORED AS TEXTFILE;**
- **hive> DESCRIBE shakespeare;**

Generating Data

- Let’s get (word, frequency) data from the Shakespeare data set:

- $hadoop jar \$HADOOP_HOME/hadoop-*-examples.jar \grep input shakespeare_freq „\w+“

Loading data

- **Remove the MapReduce job logs:**

  $hadoop fs -rmr shakespeare_freq/_logs

- **Load dataset into Hive:**

  hive> LOAD DATA INPATH “shakespeare_freq” INTO TABLE shakespeare;

Selecting data

- **hive> SELECT * FROM shakespeare LIMIT 10;**

- **hive> SELECT * FROM shakespeare WHERE freq > 100 SORT BY freq ASC LIMIT 10;**

Most common frequency

- **hive> SELECT freq, COUNT(1) AS f2 FROM shakespeare GROUP BY freq SORT BY f2 DESC LIMIT 10;**

- **hive> EXPLAIN SELECT freq, COUNT(1) AS f2 FROM shakespeare GROUP BY freq SORT BY f2 DESC LIMIT 10;**

Joining tables

- A powerful feature of Hive is the ability to create queries that join tables together
- We have (freq, word) data for Shakespeare
- Can also calculate it for KJV
- Let's see what words show up a lot in both

Create the dataset

- $cd~/git/data
- $tar zxf bible.tar.gz
- $hadoop fs –put bible bible
-
- $hadoop jar \$HADOOP_HOME/hadoop-*-examples.jar \grep bible bible_freq „\w+“

Create the new table

- **hive> CREATE TABLE kjv (freq INT, word STRING) ROW FORMAT DELIMITED FIELDS TERMINATED BY „\t“ STORED AS TEXTFILE;**
- **hive> SHOW TABLES;**
- **hive> DESCRIBE kjv;**

Import data to Hive

- $ hadoop fs –rmr bible_freq/_logs
- hive> LOAD DATA INPATH “bible_freq” INTO TABLE kjv;

Create an intermediate table

- **hive> CREATE TABLE merged (word STRING, shake_f INT, kjv_f INT);**

Running the join

- hive> INSERT OVERWRITE TABLE merged SELECT s.word, s.freq, k.freq FROM shakespeare s JOIN kjv k ON (s.word = k.word) WHERE s.freq >= 1 AND k.freq >= 1;

- hive> SELECT * FROM merged LIMIT 20;

Most common intersections

- What words appeared most frequently in both corpuses?

- hive> SELECT word, shake_f, kjv_f, (shake_f +kjv_f) AS ss
  FROM merged SORT BY ss LIMIT 20;

Manipulating Tables

- DDL operations
  - SHOW TABLES
  - CREATE TABLE
  - ALTER TABLE
  - DROP TABLE
- Loading data
- Partitioning and bucketing

Creating Tables in Hive

- **Most straightforward:**

  CREATE TABLE foo(id INT,msg STRING);

- **Assumes default table layout**

  -Text files; fields terminated with ^A, lines terminated with \n

Changing Row Format

- Arbitrary field,record separators are possible. e.g., CSV format:

- CREATE TABLE foo(id INT, msg STRING)
  ROW FORMAT
  DELIMITED FIELDS TERMINATED BY “,” LINES TERMINATED BY “\n”;

- Does not support “OPTIONALLY ENCLOSED BY” escape syntax

MapReduce Text Output

- CREATE TABLE foo(id INT, msg STRING) ROW FORMAT DELIMITED FIELDS TERMINATED BY „\t“ LINES TERMINATED BY „\n“;

SequenceFile Storage

- **CREATE TABLE foo(id INT, msg STRING) STORED AS SEQUENCEFILE;**

- **Can also explicitly say “STORED AS TEXTFILE”**

Data Types

- **Primitive types:**
  - **TINYINT**
  - **INT**
  - **BIGINT**
  - **BOOLEAN**
  - **DOUBLE**
  - **STRING**

- **Type constructors:**
  - **ARRAY < *primitive-type* >**
  - **MAP < *primitive-type, primitive-type* >**

Loading Data

- **HDFS-resident data:**

  LOAD DATA INPATH „mydata‟ [OVERWRITE] INTO TABLE foo;

- **Local filesystem data:**

  LOAD DATA LOCAL INPATH “mydata” INTO TABLE foo;

Subquery

- **Subquery syntax**

  SELECT ... FROM (subquery) name ...
- **Hive supports sub queries only in the FROM clause. The subquery has to be given a name because every table in a FROM clause must have a name.**
- **Columns in the subquery select list must have unique names. The columns in the subquery select list are available in the outer query just like columns of a table.**
- **The subquery can also be a query expression with UNION. Hive supports arbitrary levels of sub-queries.**

Subquery Example

- **Example with simple subquery:**

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

- **Example with subquery containing a UNION ALL:**

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

Aggregations

- In order to count the number of distinct users by gender one could write the following query:
  - SELECT pv_users.gender, count (DISTINCT pv_users.userid) FROM pv_users GROUP BY pv_users.gender;
- Multiple aggregations can be done at the same time, however, no two aggregations can have different DISTINCT columns
  - SELECT pv_users.gender, count(DISTINCT pv_users.userid), count(*), sum(DISTINCT pv_users.userid) FROM pv_users GROUP BY pv_users.gender;
- However, the following query is not allowed
  - SELECT pv_users.gender, count(DISTINCT pv_users.userid), count(DISTINCT pv_users.ip) FROM pv_users GROUP BY pv_users.gender;

Partitioning Data

- One or more partition columns may be specified:

  CREATE TABLE foo (id INT,msg STRING) PARTITIONED BY (dt STRING);

- Creates a subdirectory for each value of the partition column, e.g.:

  /user/hive/warehouse/foo/dt=2009-03-20/

- Queries with partition columns in WHERE clause can scan through only a subset of the data

Loading Data Into Partitions

- LOAD DATA INPATH „new_logs“ INTO TABLE mytable PARTITION(dt=2009-03-20);

Sampling Data

- May want to run query against fraction of available data

```
SELECT avg(cost) FROM purchases TABLESAMPLE (BUCKET 5
OUT OF 32 ON rand());
```

- Randomly distributes data into 32 buckets and picks data that falls into bucket #5

Bucketing Data In Advance

- TABLESAMPLE on rand() or non-bucket column causes full data scan
- Better: pre-bucket the data for sampling queries, and only scan pre-designed buckets

- CREATE TABLE purchases(id INT,cost DOUBLE, msg STRING) CLUSTERED BY id INTO 32 BUCKETS;

- Important: must remember to set mapred.reduce.tasks = 32 for ETL operations on this table.

Loading Data Into Buckets

- Bucketed tables require indirect load process
  - LOAD DATA INPATH… just moves files
- Example Table:

```
CREATE TABLE purchases(id INT, cost DOUBLE, msg STRING)
CLUSTERED BY id INTO 32 BUCKETS;
```

Populate the Loading Table

- CREATE TABLE purchaseload(id INT, cost DOUBLE, msg STRING);
- LOAD DATA INPATH “somepath” INTO TABLE purchaseload;

SELECT Into the Real Table

- set mapred.reduce.tasks = 32
  # this must ==number of buckets!

- FROM (FROM purchaseload SELECT id, cost, msg CLUSTER BY id) pl
- INSERT OVERWRITE TABLE purchases
- SELECT *;

Using Custom Mappers and Reducers

Hive has integration with Hadoop Streaming

- **Use streaming scripts; tab-delimited string I/O**

- **MAP**
- **REDUCE**
- **TRANSFORM**

Operators all take scripts as arguments

Preloading Mapper Scripts

- ***script-cmd*** **can be an arbitrary command**
  - **– e.g., /bin/cat**
- **If you have a custom mapper script, need to pre-load files to all nodes:**

  ADD FILE *mymapper.py*

Streaming example

```
FROM (
    FROM pv_users
    MAP     pv_users.userid, pv_users.date
    USING 'map_script' AS dt,uid CLUSTER BY dt) map_output

    INSERT OVERWRITE TABLE   pv_users_reduced REDUCE
map_output.dt, map_output.uid USING 'reduce_script'
    AS date, count;
```

Transform Example

```
INSERT OVERWRITE TABLE u_data_new
    SELECT
        TRANSFORM (userid, movieid, rating, unixtime)
        USING 'python weekday_mapper.py' AS (userid, movieid,
rating,weekday)
    FROM u_data;
```

Dropping Data

- **DROP TABLE *foo***
  -- delete a table
- **ALTER TABLE *foo* DROP PARTITION(*col=value*)**

Extending Tables

- **ALTER TABLE table_name ADD COLUMNS (col_name data_type [col_comment],**
  ...)

- **Can add non-partition columns to table**

Resorting Tables

Changing partitions, buckets for table requires full rescan

Create a new table to hold the new layout

Then for each partition:

INSERT OVERWRITE *new-table-name* PARTITION (*col=value*) SELECT * FROM *old-table* WHERE *col=value* CLUSTER BY *cluster-col*;

Hive User-defined function (UDF)

- Sometimes the query you want to write can't be expressed easily (or not at all) using the built-in functions that Hive provides. By writing user defined function (UDF),Hive makes it easy to plug in your own processing code and invokes it from hive query.
- UDFs have to be written in java. For other languages, you have to use SELECT TRANSFORM query, which allows you to stream data through a user-defined script.

Types Of UDF

There are three types of UDFs in Hive: regular UDF, UDAF and UDTF.
A UDF operates on single row and produces a single row as its output. Most functions, such as mathematical functions and string functions are of this type.

A UDAF works on multiple input rows and creates a single output row. Aggregate functions include such as COUNT and MAX.

A UDTF operates on a single row and produces multiple rows – a table – as output.

User-defined function (UDF)

- New UDF classes need to inherit from *org.apache.hadoop.hive.ql.exec.UDF* class. All UDF classes are required to implement one or more methods named "evaluate" which will be called by Hive. The following are some examples:
  - public int evaluate ();
  - public int evaluate (int a);
  - public double evaluate (int a, double b);
  - public String evaluate (String a, int b, String c);
- "evaluate" should never be a void method. However it can return "null" if needed.
- The evaluate() method is not defined by an interface since it may take an arbitrary number of arguments, of arbitrary types, and it may return a value of arbitrary type. Hive introspects the UDF to find the evaluate() method that matches the Hive function that was invoked.

User-defined function (UDF) example

```
import java.util.Date;
import java.text.DateFormat;

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

public class UnixTimeToDate extends UDF{
    public Text evaluate(Text text) {
        if(text == null) return null;
        long timestamp = Long.parseLong(text.toString());
        return new Text(toDate(timestamp));
    }

    private String toDate(long timestamp) {
        Date date = new Date (timestamp * 1000);
        return DateFormat.getInstance().format(date).toString();
    }
}

~
```

User-defined function (UDF) example

```
import org.apache.commons.lang.StringUtils;
import org.apache.hadoop.hive.ql.exec.UDF;
import org.apache.hadoop.io.Text;

public class Trim 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;
        }
}
```

User-defined function (UDF)

- **Hive supports Java primitives in UDFs (and a few other types like java.util.List and java.util.Map), so a signature**

  like: public String evaluate(String str) would work equally well. However, by using Text, we can take advantage of object reuse, which can bring efficiency savings, and so is to be preferred in general.

User-defined function (UDF)

- To use the UDF in Hive, we need to package the compiled Java class in a JAR file and register the jar file with Hive:

```
[sb25634@hadoop02 ~]$ hive
Hive history file=/tmp/sb25634/hive_job_log_sb25634_201205091708_1049444344.txt
hive> add jar UnixTimeToDate.jar;
Added UnixTimeToDate.jar to class path
Added resource: UnixTimeToDate.jar
```

- You can confirm using list jar command in hive.

```
hive> list jars;
UnixTimeToDate.jar
```

- Once hive is started up with your jars in the classpath, the final step is to register your function:

```
hive> create temporary function MyDateFunction as 'UnixTimeToDate';
OK
Time taken: 0.252 seconds
```

User-defined function (UDF)

- The TEMPORARY keyword highlights the fact that UDFs are only defined for the duration of the Hive session (they are not persisted in the metastore). This means you need to add the JAR file, and define the function at the beginning of each script or session.
- Now you can start using it.

```
hive> select MyDateFunction(unix_timestamp()) from users limit 1;
Total MapReduce jobs = 1
Launching Job 1 out of 1
Number of reduce tasks is set to 0 since there's no reduce operator
Starting Job = job_201203142100_0561, Tracking URL = http://hadoop1:50030/jobdetails.jsp?jobid=job_201203142100_0561
Kill Command = /usr/lib/hadoop/bin/hadoop job  -Dmapred.job.tracker=192.168.102.75:8021 -kill job_201203142100_0561
2012-05-09 17:14:11,075 Stage-1 map = 0%,  reduce = 0%
2012-05-09 17:14:14,106 Stage-1 map = 100%,  reduce = 0%
2012-05-09 17:14:16,122 Stage-1 map = 100%,  reduce = 100%
Ended Job = job_201203142100_0561
OK
5/9/12 10:29 PM
Time taken: 14.018 seconds
```

User-defined Aggregate function (UDAF)

- A UDAF works on multiple input rows and creates a single output row. Aggregate functions include such as COUNT and MAX.
- New UDAF classes need to inherit from *org.apache.hadoop.hive.ql.exec.UDAF* class.
- It contains one or more nested static classes implementing org.apache.hadoop.hive.ql.exec.UDAFEvaluator.

User-defined Aggregate function (UDAF)

- **An evaluator must implement five methods.**
  - **init () method, which reset the status of the aggregation function.**
  - **iterate()**

The iterate() method is called every time there is a new value to be aggregated. The evaluator should update its internal state with the result of performing the aggregation. The arguments that iterate() takes correspond to those in the Hive function from which it was called.

  - **terminatePartial()**

The terminatePartial() method is called when Hive wants a result for the partial aggregation. The method must return an object that encapsulates the state of the aggregation.

User-defined Aggregate function (UDAF)

- **merge()**

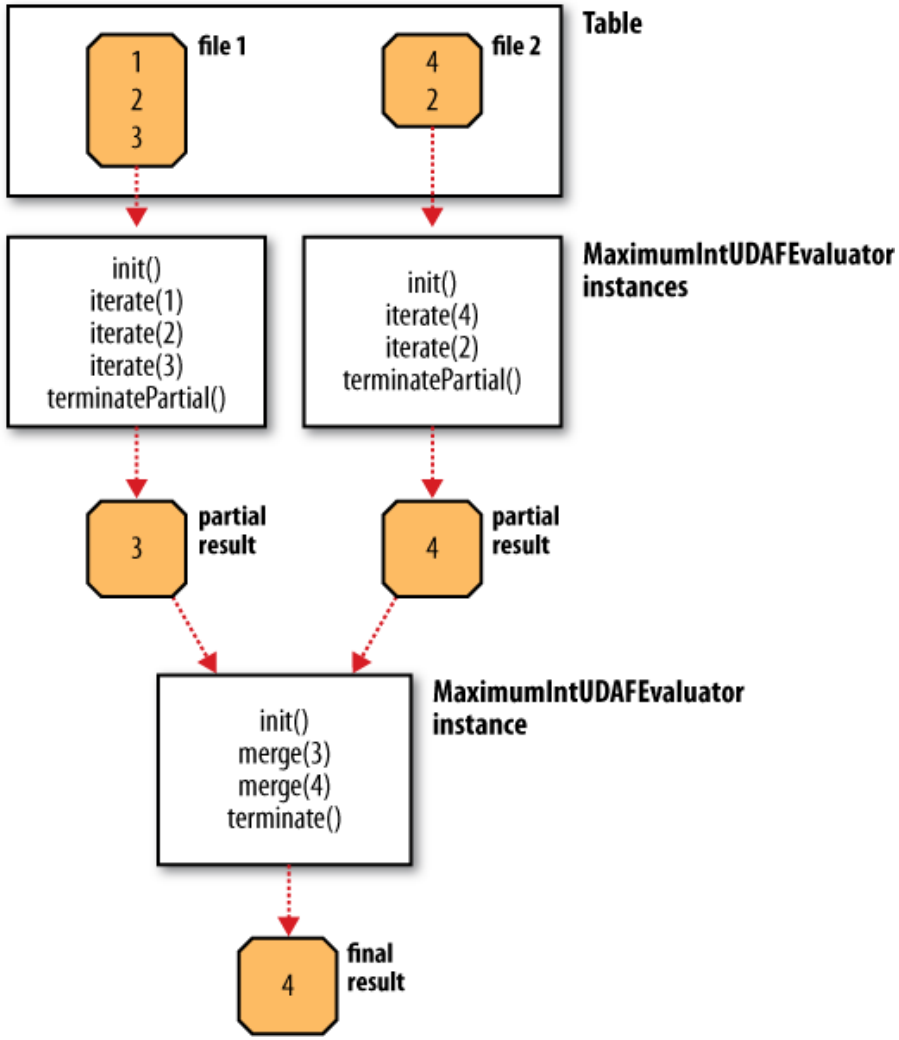
  The merge() method is called when Hive decides to combine one partial aggregation with another. The method takes a single object whose type must correspond to the return type of the terminatePartial() method. The method should implement the logic to combine the evaluator's state with the state of the partial aggregation.

User-defined Aggregate function (UDAF)

- **terminate()**

  The terminate() method is called when the final result of the aggregation is needed.

DATA FLOW WITH PARTIAL RESULTS FOR A UDAF



User-defined Aggregate function (UDAF) example

```
import org.apache.hadoop.hive.ql.exec.UDAF;
import org.apache.hadoop.hive.ql.exec.UDAFEvaluator;

public final class UDAFExampleAvg extends UDAF {


   public static class UDAFAvgState
   {

     private long mCount;

     private double mSum;

   }

   public static class UDAFExampleAvgEvaluator implements UDAFEvaluator {

     UDAFAvgState state;

     public UDAFExampleAvgEvaluator()
     {
         super();
         state = new UDAFAvgState();
         init();
     }

     public void init()
     {
       state.mSum = 0;
       state.mCount = 0;
     }

     public boolean iterate(Double o)
     {
      if (o != null)
      {
          state.mSum += o;
          state.mCount++;
       }
        return true;
     }
```

User-defined Aggregate function (UDAF) Example

```
    public UDAFAvgState terminatePartial()
    {
              return state.mCount == 0 ? null : state;
    }

    public boolean merge(UDAFAvgState o)
    {
        if (o != null)
        {
                  state.mSum += o.mSum;
                  state.mCount += o.mCount;
        }
        return true;
     }

     public Double terminate()
     {
              return state.mCount == 0 ? null : Double.valueOf(state.mSum/state.mCount);
     }

}

   private UDAFExampleAvg()
   {
   }

}
```

Creating Sample Table

- Before using UDAF, let's create a table and load it with some data to test UDAFs.
- Create a file which will contain numbers of type double.

```
10.12
23.44
12.33
45.66
57.67
74.97
95.36
85.35
10.34
55.567
```

- Start hive CLI using hive command.
- Create table using following Hive DDL query.

```
hive> create table UDFDemo
    > (val double
    > )
    > ROW FORMAT DELIMITED
    > LOCATION '/user/sb25634/UDFExample';
OK
Time taken: 4.572 seconds
```

Loading data in sample table

- **Load data in table using load data command as shown below.**

```
hive> load data local inpath '/home/sb25634/Demo' into table UDFDemo;
Copying data from file:/home/sb25634/Demo
Copying file: file:/home/sb25634/Demo
Loading data to table default.udfdemo
OK
Time taken: 0.317 seconds
```

```
hive> select * from UDFDemo;
OK
10.12
23.44
12.33
45.66
57.67
74.97
95.36
85.35
10.34
55.56
Time taken: 7.31 seconds
```

User-defined Aggregate function (UDAF)

```
hive> add jar My_Average_UDAF.jar;
Added My_Average_UDAF.jar to class path
Added resource: My_Average_UDAF.jar
hive> create temporary function My_Average as 'UDAFExampleAvg';
OK
Time taken: 0.248 seconds
```

As the java class for UDAF has nested classes, make sure you add all these classes in jar file.

```
hive> select My_Average(val) from UDFDemo;
Total MapReduce jobs = 1
Launching Job 1 out of 1
Number of reduce tasks determined at compile time: 1
In order to change the average load for a reducer (in bytes):
  set hive.exec.reducers.bytes.per.reducer=<number>
In order to limit the maximum number of reducers:
  set hive.exec.reducers.max=<number>
In order to set a constant number of reducers:
  set mapred.reduce.tasks=<number>
Starting Job = job_201203142100_0591, Tracking URL = http://hadoop1:50030/jobdetails.jsp?jobid=job_201203142100_0591
Kill Command = /usr/lib/hadoop/bin/hadoop job  -Dmapred.job.tracker=192.168.102.75:8021 -kill job_201203142100_0591
2012-05-11 23:13:15,732 Stage-1 map = 0%,  reduce = 0%
2012-05-11 23:13:17,772 Stage-1 map = 100%,  reduce = 0%
2012-05-11 23:13:25,836 Stage-1 map = 100%,  reduce = 100%
Ended Job = job_201203142100_0591
OK
47.08
Time taken: 30.324 seconds
```

User-defined Table-Generating function (UDTF)

- New UDTF classes need to extend from org.apache.hadoop.hive.ql.udf.generic.GenericUDTF class. (There is no plain UDTF class.)
- We need to implement three methods: initialize, process, and close. To emit output, we call the forward method

User-defined Table-Generating function (UDTF)

- **The initialize method:**
  - **This method will be called exactly once per instance. In addition to performing any custom initialization logic you may need, it is responsible for verifying the input types and specifying the output types.**
  - **Hive uses a system of ObjectInspectors to both describe types and to convert Objects into more specific types. For our tokenize, we want a single String as input, so we'll check that the input ObjectInspector[] array contains a single PrimitiveObjectInspector of the STRING category. If anything is wrong, we throw a UDFArgumentException with a suitable error message**

User-defined Table-Generating function (UDTF)

- **The process method:**

  This method is where the heavy lifting occurs. This method gets called for each row of the input.

- **The close method:**

  This method allows us to do any post-processing cleanup

UDTF

```
import java.util.ArrayList;
import java.util.List;

import org.apache.hadoop.hive.ql.metadata.HiveException;
import org.apache.hadoop.hive.ql.udf.generic.GenericUDTF;
import org.apache.hadoop.hive.serde2.objectinspector.StructObjectInspector;
import org.apache.hadoop.hive.ql.exec.UDFArgumentException;
import org.apache.hadoop.hive.serde2.objectinspector.ObjectInspector;
import org.apache.hadoop.hive.serde2.objectinspector.ObjectInspectorFactory;
import org.apache.hadoop.hive.serde2.objectinspector.primitive.PrimitiveObjectInspectorFactory;
import org.apache.hadoop.hive.serde2.objectinspector.PrimitiveObjectInspector;

public class TokenizerUDTF extends GenericUDTF
{
    PrimitiveObjectInspector stringOI;

    public StructObjectInspector initialize(ObjectInspector[] args) throws UDFArgumentException
    {
        if (args.length != 1)
        {
            throw new UDFArgumentException("tokenize() takes exactly one argument");
        }

        if (args[0].getCategory() != ObjectInspector.Category.PRIMITIVE&&
            ((PrimitiveObjectInspector) args[0]).getPrimitiveCategory() !=
            PrimitiveObjectInspector.PrimitiveCategory.STRING)
         {
            throw new UDFArgumentException("tokenize() takes a string as a parameter");
         }
```

UDTF

```
        stringOI = (PrimitiveObjectInspector) args[0];

        List fieldNames = new ArrayList<String>(2);
        List fieldOIs = new ArrayList(2);
        fieldNames.add("word");
        fieldNames.add("cnt");
        fieldOIs.add(PrimitiveObjectInspectorFactory.javaStringObjectInspector);
        fieldOIs.add(PrimitiveObjectInspectorFactory.javaIntObjectInspector);

        return ObjectInspectorFactory.getStandardStructObjectInspector(fieldNames, fieldOIs);
    }

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

        if (document == null)
        {
            return;
        }

        String[] tokens = document.split("\\s+");

        for (String token : tokens)
        {
            forward(new Object[] { token, Integer.valueOf(1)});
        }
    }
    public void close() throws HiveException { }

}
```

User-defined Table-Generating function (UDTF)

```
hive> add jar TokenizerUDTF.jar;
Added TokenizerUDTF.jar to class path
Added resource: TokenizerUDTF.jar
```

```
hive> create temporary function tokenize as 'TokenizerUDTF';
OK
Time taken: 0.0080 seconds
```

```
hive> select tokenize(content) as (word,count) from docs;
Total MapReduce jobs = 1
Launching Job 1 out of 1
Number of reduce tasks is set to 0 since there's no reduce operator
Starting Job = job_201203142100_0592, Tracking URL = http://hadoop1:50030/jobdetails.jsp?jobid=job_201203142100_0592
Kill Command = /usr/lib/hadoop/bin/hadoop job  -Dmapred.job.tracker=192.168.102.75:8021 -kill job_201203142100_0592
2012-05-14 15:47:15,726 Stage-1 map = 0%,  reduce = 0%
2012-05-14 15:47:18,754 Stage-1 map = 100%,  reduce = 0%
2012-05-14 15:47:20,774 Stage-1 map = 100%,  reduce = 100%
Ended Job = job_201203142100_0592
OK
A       1
UDTF    1
operates        1
on      1
multiple        1
input   1
rows    1
and     1
produces        1
multiple        1
rows    1
a       1
table   1
as      1
output. 1
A       1
UDTF    1
operates        1
on      1
multiple        1
input   1
rows    1
and     1
produces        1
multiple        1
rows    1
a       1
table   1
as      1
output. 1
Time taken: 10.264 seconds
```

Project Status

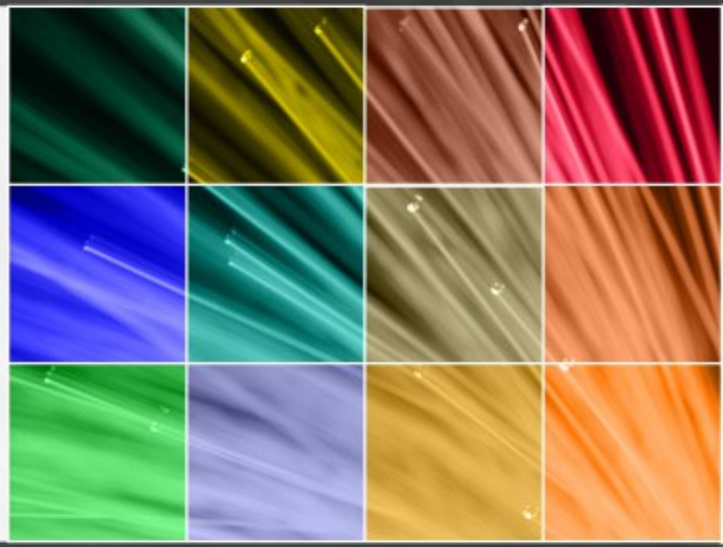
- Open source, Apache 2.0 license
- Official subproject of Apache Hadoop
- Several committers
- Current version is 0.5.0

Related work

- Parallel databases: Gamma, Bubba, Volcano
- Google: Sawzall
- Yahoo: Pig
- IBM Research: JAQL
- Microsoft: DryadLINQ, SCOPE
- Greenplum: YAML MapReduce
- Aster Data: In-database MapReduce
- Business.com: CloudBase

Syntel
Consider IT Done®

Thank You



strategic offerings group