Introduction to Apache Hive

Pelle Jakovits

1. Oct, 2013, Tartu

Outline

- What is Hive
- Why Hive over MapReduce or Pig?
  - Advantages and disadvantages
- Running Hive
- HiveQL language
- Examples
- Internals
- Hive vs Pig
- Other Hadoop projects

Hive

- Data warehousing on top of Hadoop.
- Designed to
  - enable easy data summarization
  - ad-hoc querying
  - analysis of large volumes of data.
- HiveQL statements are automatically translated into MapReduce jobs

Advantages of Hive

- Higher level query language
  - Simplifies working with large amounts of data
- Lower learning curve than Pig or MapReduce
  - HiveQL is much closer to SQL than Pig
  - Less trial and error than Pig

Disadvantages

- Updating data is complicated
  - Mainly because of using HDFS
  - Can add records
  - Can overwrite partitions
- No real time access to data
  - Use other means like Hbase or Impala
- High latency

Running Hive

- We will look at it more closely in the practice session, but you can run hive from
  - Hive web interface
  - Hive shell
    - $HIVE_HOME/bin/hive for interactive shell
    - Or you can run queries directly: $HIVE_HOME/bin/hive -e ‘select a.col from tab1 a’
  - JDBC – Java Database Connectivity
    - "jdbc:hive://host:port/dbname„
  - Also possible to use hive directly in Python, C, C++, PHP

Query Processor

- Compiler
  - Parser
  - Type checking
  - Plan Generator
  - Task Generator
- Execution engine
  - Plan
  - Operators
  - UDF (UDAF's)

HiveQL

- Hive query language provides the basic SQL like operations.
- These operations are:
  - Ability to filter rows from a table using a where clause.
  - Ability to select certain columns from the table using a select clause.
  - Ability to do equi-joins between two tables.
  - Ability to evaluate aggregations on multiple "group by" columns for the data stored in a table.
  - Ability to store the results of a query into another table.
  - Ability to download the contents of a table to a local directory.
  - Ability to store the results of a query in a hadoop dfs directory.
  - Ability to manage tables and partitions (create, drop and alter).
  - Ability to use custom scripts in chosen language (for map/reduce).

Data units

- **Databases**: Namespaces that separate tables and other data units from naming confliction.
- **Tables:**
  - Homogeneous units of data which have the same schema.
  - Consists of specified columns accordingly to its schema
- **Partitions**:
  - Each Table can have one or more partition Keys which determines how data is stored.
  - Partitions allow the user to efficiently identify the rows that satisfy a certain criteria.
  - It is the user's job to guarantee the relationship between partition name and data!
  - Partitions are virtual columns, they are not part of the data, but are derived on load.
- **Buckets** (or **Clusters**):
  - Data in each partition may in turn be divided into Buckets based on the value of a hash function of some column of the Table.
  - For example the page_views table may be bucketed by userid to sample the data.

Types

- Types are associated with the columns in the tables. The following Primitive types are supported:
  - **Integers**
    - TINYINT - 1 byte integer
    - SMALLINT - 2 byte integer
    - INT - 4 byte integer
    - BIGINT - 8 byte integer
  - **Boolean type**
    - BOOLEAN
  - **Floating point numbers**
    - FLOAT
    - DOUBLE
  - **String type**
    - STRING

Complex types

- **Structs**
  - the elements within the type can be accessed using the DOT (.) notation. F
  - or example, for a column c of type STRUCT {a INT; b INT} the a field is accessed by the expression c.a
- **Maps** (key-value tuples)
  - The elements are accessed using ['element name'] notation.
  - For example in a map M comprising of a mapping from 'group' -> gid the gid value can be accessed using M['group']
- **Arrays** (indexable lists)
  - The elements in the array have to be in the same type.
  - Elements can be accessed using the [n] notation where n is an index (zero-based) into the array.
  - For example for an array A having the elements ['a', 'b', 'c'], A[1] retruns 'b'.

Built in functions

- Round(number), floor(number), ceil(number)
- Rand(seed)
- Concat(string), substr(string, length)
- Upper(string), lower(string), trim(string)
- Yea(date), Month(date), Day (date)
- Count(*), sum(col), avg(col), max(col), min(col)

Create Table

```
CREATE TABLE page_view(viewTime INT, userid BIGINT,
                page_url STRING, referrer_url STRING,
                ip STRING COMMENT 'IP Address of the User')
COMMENT 'This is the page view table'
PARTITIONED BY(dt STRING, country STRING)
ROW FORMAT DELIMITED
        FIELDS TERMINATED BY '1'
STORED AS SEQUENCEFILE;
```

Load Data

- There are multiple ways to load data into Hive tables.
- The user can create an external table that points to a specified location within HDFS.
- The user can copy a file into the specified location in HDFS and create a table pointing to this location with all the relevant row format information.
- Once this is done, the user can transform the data and insert them into any other Hive table.

Load example

- CREATE EXTERNAL TABLE page_view_stg(viewTime INT, userid BIGINT,
- page_url STRING, referrer_url STRING,
- ip STRING COMMENT 'IP Address of the User',
- country STRING COMMENT 'country of origination')
- COMMENT 'This is the staging page view table'
- ROW FORMAT DELIMITED FIELDS TERMINATED BY '44' LINES TERMINATED BY '12'
- STORED AS TEXTFILE
- LOCATION '/user/data/staging/page_view';

Load example

- hadoop dfs -put /tmp/pv_2008-06-08.txt /user/data/staging/page_view

- FROM page_view_stg pvs
- INSERT OVERWRITE TABLE page_view PARTITION(dt='2008-06-08', country='US')
- SELECT pvs.viewTime, pvs.userid, pvs.page_url, pvs.referrer_url, null, null, pvs.ip
- WHERE pvs.country = 'US';

Loading and storing data

- Loading data directly:

LOAD DATA LOCAL INPATH /tmp/pv_2008-06-08_us.txt INTO TABLE page_view PARTITION(date='2008-06-08', country='US')

Storing locally

- To write the output into a local disk, for example to load it into an excel spreadsheet later:

```
INSERT OVERWRITE LOCAL DIRECTORY
'/tmp/pv_gender_sum'
SELECT pv_gender_sum.*
FROM pv_gender_sum;
```

Display functions

- SHOW TABLES;
  - Lists all tables
- SHOW PARTITIONS page_view;
  - Lists partitions on a specific table
- DESCRIBE EXTENDED page_view;
  - To show all metadata
  - Mainly for debugging

INSERT

```
INSERT OVERWRITE TABLE xyz_com_page_views
SELECT page_views.*
FROM page_views
WHERE page_views.date >= '2008-03-01' AND
page_views.date <= '2008-03-31' AND
    page_views.referrer_url like '%xyz.com';
```

Multiple table/file inserts

- The output can be sent into multiple tables or even to hadoop dfs files (which can then be manipulated using hdfs utilities).
- If along with the gender breakdown, one needed to find the breakdown of unique page views by age, one could accomplish that with the following query:

```
FROM pv_users
INSERT OVERWRITE TABLE pv_gender_sum
   SELECT pv_users.gender, count_distinct(pv_users.userid)
   GROUP BY pv_users.gender
INSERT OVERWRITE DIRECTORY '/user/data/tmp/pv_age_sum'
   SELECT pv_users.age, count_distinct(pv_users.userid)
   GROUP BY pv_users.age;
```

- The first insert clause sends the results of the first group by to a Hive table while the second one sends the results to a hadoop dfs files.

JOIN

- LEFT OUTER, RIGHT OUTER or FULL OUTER
- Can join more than 2 tables at once
- It is best to put the largest table on the rightmost side of the join to get the best performance.

```
INSERT OVERWRITE TABLE pv_users
SELECT pv.*, u.gender, u.age
FROM user u JOIN page_view pv ON (pv.userid =
u.id)
WHERE pv.date = '2008-03-03';
```

Aggregations

```
INSERT OVERWRITE TABLE pv_gender_agg
SELECT pv_users.gender, count(DISTINCT
pv_users.userid), count(*), sum(DISTINCT
pv_users.userid)
FROM pv_users
GROUP BY pv_users.gender;
```

Sampling

```
INSERT OVERWRITE TABLE
pv_gender_sum_sample

SELECT pv_gender_sum.*

FROM pv_gender_sum TABLESAMPLE(BUCKET 3
OUT OF 32);
```

Union

```
INSERT OVERWRITE TABLE actions_users
SELECT u.id, actions.date
FROM (
   SELECT av.uid AS uid
   FROM action_video av
   WHERE av.date = '2008-06-03'

   UNION ALL

   SELECT ac.uid AS uid
   FROM action_comment ac
   WHERE ac.date = '2008-06-03'
   ) actions JOIN users u ON(u.id = actions.uid);
```

Running custom mapreduce

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

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

Map Example

```
import sys
import datetime

for line in sys.stdin:
  line = line.strip()
  userid, unixtime = line.split('\t')
  weekday =
datetime.datetime.fromtimestamp(float(unixtime)).
isoweekday()
  print ','.join([userid, str(weekday)])
```

Co-group

```
FROM (
    FROM (
        FROM action_video av
        SELECT av.uid AS uid, av.id AS id, av.date AS date
      UNION ALL
        FROM action_comment ac
        SELECT ac.uid AS uid, ac.id AS id, ac.date AS date
    ) union_actions
    SELECT union_actions.uid, union_actions.id, union_actions.date
    CLUSTER BY union_actions.uid) map

 INSERT OVERWRITE TABLE actions_reduced
    SELECT TRANSFORM(map.uid, map.id, map.date) USING 'reduce_script' AS
(uid, id, reduced_val);
```

UDF’s

Java UDF

```
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());
  }
}
```

Using UDF

- create temporary function my_lower as 'com.example.hive.udf.Lower';
- hive> select my_lower(title), sum(freq) from titles group by my_lower(title);

Hive disadvantages

- Same disadvantages as MapReduce and Pig
  - Slow start-up and clean-up of MapReduce jobs
    - It takes time for Hadoop to schedule MR jobs
  - Not suitable for interactive OLAP Analytics
    - When results are expected in < 1 sec
- Designed for querying and not data transformation
  - Limitations of the SQL language
  - Tasks like co-grouping can get complicated

Pig vs Hive

	Pig	Hive
Purpose	Data transformation	Ad-Hoc querying
Language	Something similiar to SQL	SQL-like
Difficulty	Medium (Trial-and-error)	Low to medium
Schemas	Yes (implicit)	Yes (explicit)
Join (Distributed)	Yes	Yes
Shell	Yes	Yes
Streaming	Yes	Yes
Web interface	No	Yes
Partitions	No	Yes
UDF’s	Yes	Yes

Pig vs Hive

- SQL might not be the perfect language for expressing data transformation commands
- Pig
  - Mainly for data transformations and processing
  - Unstructured data
- Hive
  - Mainly for warehousing and querying data
  - Structured data

More Hadoop projects

- **Hbase**
  - Open-source distributed database ontop of HDFS
  - Hbase tables only use a single key
  - Tuned for real-time access to data
- **Cloudera Impala™**
  - Simplified, real time queries over HDFS
  - Bypass job schedulling, and remove everything else that makes MR slow.

Big Picture

- Store large amounts of data to HDFS
- Process raw data: Pig
- Build schema using Hive
- Data querries: Hive
- Real time access
  - access to data with Hbase
  - real time queries with Impala

Is Hadoop enough?

- Why use Hadoop for large scale data processing?
  - It is becoming a de facto standard in Big Data
  - Collaboration among Top Companies instead of vendor tool lock-in.
    - Amazon, Apache, Facebook, Yahoo!, etc all contribute to open source Hadoop
  - There are tools from setting up Hadoop cluster in minutes and importing data from relational databases to setting up workflows of MR, Pig and Hive.

Thats All

- This week`s practice session
  - Processing data with Hive
  - Similiar exercise as last two weeks, this time using Hive
- Next lecture: NoSQL
  - Stepping away from Hadoop to give you a broader view