Introduction to Pig

Content developed and presented by:

Outline

- Motivation
- Background
- Components
- How it Works with Map Reduce
- Pig Latin by Example
- Wrap up & Conclusions

Motivation

- Map Reduce is very powerful, but:
  - It requires a Java programmer.
  - User has to re-invent common functionality (join, filter, etc.)

Pig Latin

- Pig provides a higher level language, Pig Latin, that:
  - Increases productivity. In one test
    - 10 lines of Pig Latin ≈ 200 lines of Java.
    - What took 4 hours to write in Java took 15 minutes in Pig Latin.
  - Opens the system to non-Java programmers.
  - Provides common operations like join, group, filter, sort.

Pig Engine

- Pig provides an execution engine atop Hadoop
  - Removes need for users to tune Hadoop for their needs.
  - Insulates users from changes in Hadoop interfaces.

Why a New Language?

- Pig Latin is a data flow language rather than procedural or declarative.
- User code and existing binaries can be included almost anywhere.
- Metadata not required, but used when available.
- Support for nested types.
- Operates on files in HDFS.

Background

- Yahoo! was the first big adopter of Hadoop.
- Hadoop gained popularity in the company quickly.
- Yahoo! Research developed Pig to address the need for a higher level language.
- Roughly 30% of Hadoop jobs run at Yahoo! are Pig jobs.

How Pig is Being Used

- Web log processing.
- Data processing for web search platforms.
- Ad hoc queries across large data sets.
- Rapid prototyping of algorithms for processing large data sets.

Accessing Pig

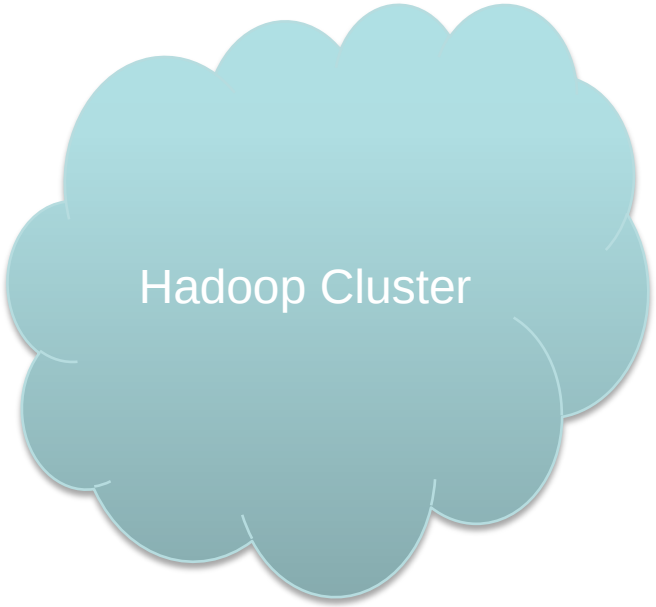
- Submit a script directly.
- Grunt, the pig shell.
- PigServer Java class, a JDBC like interface.
- PigPen, an eclipse plugin
  - Allows textual and graphical scripting.
  - Samples data and shows example data flow.

Components

Job executes on cluster

Pig resides on user machine

User machine



No need to install anything extra on your Hadoop cluster.

How It Works

Pig Latin

```
A = LOAD 'myfile'
    AS (x, y, z);
B = FILTER A by x> 0;
C = GROUP B BY x;
D = FOREACH A GENERATE
x, COUNT(B);
STORE D INTO 'output';
```

pig.jar:

- •parses
- •checks
- •optimizes
- •plans execution
- •submits jar to Hadoop
- •monitors job progress

Execution Plan

Map:
Filter

Reduce:
Count

Is it Fast?

- Pig team has developed a set of benchmark queries, Pig Mix http://wiki.apache.org/pig/PigMix
- Contains 12 queries covering most Pig features.
- Also wrote 12 Java programs to compare against MR performance.
- Release 0.2.0 is at 1.6x MR, proto 0.3.0 is at 1.4x.

Data Types

- Scalar types: int, long, double, chararray, bytearray.
- Complex types:
  - map: associative array.
  - tuple: ordered list of data, elements may be of any scalar or complex type.
  - bag: unordered collection of tuples.

Find Excite Logs

- We'll be using some logs from excite for our first few examples.
- The data is in the virtual machine:

  `~/git/data/excite-small.log`

Examples

- Start a terminal and run

```
$ cd ~/git/data
$ pig -x local
```

- Should see a prompt like:

```
grunt>
```

Aggregation

Let's count the number of times each user appears in the excite data set.

```
log  = LOAD 'excite-small.log'
        AS (user, timestamp, query);
grpd = GROUP log BY user;
cntd = FOREACH grpd GENERATE group, COUNT(log);
STORE cntd INTO 'output';
```

Results:

```
002BB5A52580A8ED        18
005BD9CD3AC6BB38        18
...
```

A Closer Look…

- What's going on here?

```
log  = LOAD 'excite-small.log'
        AS (user, timestamp, query);
grpd = GROUP log BY user;
cntd = FOREACH grpd GENERATE group,
  COUNT(log);
STORE cntd INTO 'output';
```

Datasets and Aliases

- Each statement defines a new *dataset*, possibly in terms of existing datasets.
- Each dataset is immutable.
- Datasets can be given *aliases* to use later.

```
log  = LOAD 'excite-small.log'
  AS (user, timestamp, query);
```

LOAD Returns a Bag

- LOAD statements return a *bag* of *tuples.* A bag is an ordered set of values. Each tuple has multiple elements, which can be referenced *positionally* or *by name.*

```
log = LOAD 'excite-small.log'
  AS (user, timestamp, query);
```

Bags and FOREACH

- The FOREACH… GENERATE statement iterates over the members of a bag.

```
usernames = FOREACH log
   GENERATE user;
```

- The result of a FOREACH is another bag.
- Elements are named as in the input bag.

Positional Reference

- The following creates identical output data

```
usernames = FOREACH log
    GENERATE $1;
```

- …But the elements of **`usernames`** aren't named "**`user`**" – unless you do this:

```
usernames = FOREACH log
    GENERATE $1 as user;
```

Grouping

- In Pig grouping is separate operation from applying aggregate functions.
- In the earlier example, the output of group statement is (key, bag), where key is the group key and bag contains a tuple for every record with that key.
- For example:

```
alan 1
bob  9    =>  alan, {(alan, 1),(alan, 3)}
alan 3        bob, {(bob, 9)}
```

Grouping and Types

- **`grpd = GROUP log BY user;`**
- GROUP BY makes an output bag containing tuples, containing more bags

- In: BagOf(user, query, time)
- Out: BagOf(**`group`**, BagOf(user, query, time) named log)
- The grouping item is always named "**`group`**"

Filtering

Now let's apply a filter to the groups so that we only get the high frequency users.

```
log   = LOAD 'excite-small.log'
          AS (user, time, query);
grpd  = GROUP log BY user;
cntd  = FOREACH grpd GENERATE
          group, COUNT(log) AS cnt;
fltrd = FILTER cntd BY cnt > 50;
STORE fltrd INTO 'output';
```

Results:

```
0B294E3062F036C3        61
128315306CE647F6        78
7D286B5592D83BBE        59
```

Ordering

Sort the high frequency users by frequency.

```
log   = LOAD 'excite-small.log'
          AS (user, time, query);
grpd  = GROUP log BY user;
cntd  = FOREACH grpd GENERATE
          group, COUNT(log) AS cnt;
fltrd = FILTER cntd BY cnt > 50;
srtd  = ORDER fltrd BY cnt;
STORE srtd INTO 'output';
```

Results:

```
7D286B5592D83BBE        59
0B294E3062F036C3        61
128315306CE647F6        78
```

Prepare Data For Joins

Run a pig job to prepare the Shakespare and King James Bible data for the following examples:

```
A = load 'bible';
B = foreach A generate
    flatten(TOKENIZE((chararray)$0)) as word;
C = filter B by word matches '\\w+';
D = group C by word;
E = foreach D generate COUNT(C), group;
store E into 'bible_freq';
```

Repeat for Shakespeare, changing ‘bible’ to ‘input’ in line 1 and ‘bible_freq’ to ‘shakespeare_freq’ in line 7.

FLATTENing Sets

```
B = foreach A generate
  flatten(TOKENIZE((chararray)$0))
  as word;
```

- "TOKENIZE" returns a new bag for each input; "flatten" eliminates bag nesting
- A: { string, string, string…}
- After tokenize: {{string, string…} {string…}}
- After flatten { string, string… }
- flatten "as word": { word, word, … }

Join

We can use join to find words that are in both the King James version of the Bible and Shakespeare's collected works.

```
bard   = LOAD 'shakespeare_freq' AS (freq, word);
kjv    = LOAD 'bible_freq' AS (freq, word);
inboth = JOIN bard BY word, kjv BY word;
STORE inboth INTO 'output';
```

Results:

```
2       Abide   1       Abide
2       Abraham 111     Abraham
3  …
```

Inboth is: BagOf((freq, word) named bard, (freq, word) named kjv)

Anti-Join

Find words that are in the Bible that are not in Shakespeare.

```
bard   = LOAD 'shakespeare_freq' AS (freq, word);
kjv    = LOAD 'bible_freq' AS (freq, word);
grpd   = COGROUP bard BY word, kjv BY word;
nobard = FILTER grpd BY COUNT(bard) == 0;
out    = FOREACH nobard GENERATE FLATTEN(kjv);
STORE out INTO 'output';
```

Results:

```
1       Abimael
22 Abimelech
...
```

Cogrouping

- Cogrouping is a generalization of grouping.
- Keys for two (or more) inputs are collected.
- In previous slide, output of group statement is (key, bag1, bag2), where key is the group key, bag1 contains a tuple for every record in the first input with that key, and bag2 contains a tuple for every record in the second input with that key.

Cogrouping Continued

- For example:

```
Input 1:                    Input 2:
alan 1                      alan 5
bob  9                      john 3
alan 3

Output:
alan, {(alan, 1),(alan, 3)}, {(alan, 5)}
bob, {(bob, 9),}, {}
john, {}, {(john, 3)}
```

Using Types

- By default Pig treats data as un-typed.
- User can declare types of data at load time.

```
log = LOAD 'shakespeare_freq'
          AS (freq:int, word: chararray);
```

- If data type is not declared but script treats value as a certain type, Pig will assume it is of that type and cast it.

```
log      = LOAD 'shakespeare_freq' AS (freq, word);
weighted = FOREACH log
           GENERATE freq * 100; --freq cast to int
```

Custom Load & Store

- By default Pig assumes data is tab separated UTF-8.
- User can set a different delimiter.
- If you have a different format, you use another load/store function to handle de/serialization of the data:

```
A = LOAD 'data.json'
    USING PigJsonLoader();
…
STORE INTO 'output.json'
    USING PigJsonLoader();
```

User Defined Functions

- For logic that cannot be done in Pig.
- Can be used to do column transformation, filtering, ordering, custom aggregation.
- For example, you want to write custom logic to do user session analysis:

```
log  = LOAD 'excite-small.log'
        AS (user, time, query);
grpd = GROUP log BY user;
cntd = FOREACH grpd GENERATE
        group, SessionAnalysis(log);
STORE cntd INTO 'output';
```

Nested Operations

- A FOREACH can apply a set of operators to every tuple in turn.
- Using the previous session analysis example, assume the UDF requires input to be sorted by time:

```
log  = LOAD 'excite-small.log'
         AS (user, time, query);
grpd = GROUP log BY user;
cntd = FOREACH grpd {
   srtd = ORDER log BY time;
   GENERATE group, SessionAnalysis(srtd);
}
STORE cntd INTO 'output';
```

Split

- Data flow need not be linear:

```
A = LOAD 'data';
B = GROUP A BY $0;
…
C = GROUP A by $1;
…
```

- Split can be used explicitly:

```
A = LOAD 'data';
SPLIT A INTO NEG IF $0 < 0, POS IF $0 > 0;
```

- `B = FOREACH NEG GENERATE …`

```
…
```

Pig Commands

Pig Command	What it does
load	Read data from file system.
store	Write data to file system.
foreach	Apply expression to each record and output one or more records.
filter	Apply predicate and remove records that do not return true.
group/cogroup	Collect records with the same key from one or more inputs.
join	Join two or more inputs based on a key.
order	Sort records based on a key.
distinct	Remove duplicate records.
union	Merge two data sets.
split	Split data into 2 or more sets, based on filter conditions.
stream	Send all records through a user provided binary.
dump	Write output to stdout.
limit	Limit the number of records.

Project Status

- Open source, Apache 2.0 license.
- Official subproject of Apache Hadoop.
- 10 committers (8 from Yahoo!)
- Version 0.2.0 released April 2009.

Conclusions

- Opens up the power of Map Reduce.
- Provides common data processing operations.
- Supports rapid iteration of ad-hoc queries.

Content developed and presented by: