Today's topics

• FAQs
• Pig Latin

FAQs

• Term project TP3 deadline
  • April 29 9PM
  • Late submission May 1 noon
• Term project presentation
  • May 1 4-6PM CSB130
  • May 3 4-5:50PM CSB130
• 10 minutes per team
• Presentation
• Q&A
• Transition

• Send your slides 2hrs before the class starts
• Final Exam: May 11 7:30-9:30AM CSB130
  • Guidelines are available on the course web page

This material is built based on,


Relation Names

• Pig Latin is a dataflow language
  • Each processing step results in a new data set or relation.
  ```
  input = load 'data'
  ```
  • input
    • Name of the relation
    • Results from loading the data set data
    • Relation names look like variables
      • But they are NOT.
Reusing the relation names

- It is possible to reuse relation names.

```plaintext
A = load 'NYSE_dividends' (exchange, symbol, date, dividends);
A = filter A by dividends > 0;
A = foreach A generate UPPER(symbol)
```

- This example creates a new relation called A repeatedly.
- It loses track of the old relations called A.
- **Not recommended**

Field Names

- Field (or column) in a relation

```plaintext
A = load 'NYSE_dividends' (exchange, symbol, date, dividends);
A = filter A by dividends > 0;
A = foreach A generate UPPER(symbol)
```

- `dividends, symbol, date ...`
- Looks like variables
- But you cannot assign values to them!

Preliminary Matters - 1

- Both relation and field names MUST start with an alphabetic character
- Then they can have 0 or more alphabetic, numeric, or `_` characters.
- All characters in the name must be ASCII
- Keyword in Pig Latin is NOT case-sensitive
- LOAD = load
- Field/relation names ARE case-sensitive

```plaintext
A = load 'foo'
a = load 'foo'
```

- Two different relations

Preliminary Matters - 2

- UDF names **ARE** case-sensitive
- `COUNT()` is not the same UDF as `count()`.
- Comments
- SQL-style
  ```plaintext
  --
  ``
- Java-style
  ```plaintext
  /*
  */
  ``
- ```plaintext
  A = load 'foo': -- this is a single-line comment
  */
  This is a multiline comment.
  */
  B = load /*a comment in the middle */'bar';
  ```

Input and Output: load

- Tab separated file store in HDFS
- Using relative path name

```plaintext
divs = load '/data/examples/NYSE_dividends'
divs = load 'hdfs://nn.acme.com/data/examples/NYSE_dividends'
```

- All of the relative path to the data file is based on your home directory of HDFS
- `users/yourlogin`
- Using complete path name

```plaintext
divs = load 'hdfs://nn.acme.com/data/examples/NYSE_dividends'
```

- Data stored in HBase
- Using loader for HBase

```plaintext
divs = load 'NYSE_dividends' using HBaseStorage()
```
PigStorage()
- Default data loader for PIG
- Indicating the separator
  
  `dive = load 'NYSE_dividends' using PigStorage(',');`
- Specifying the schema
  
  `dive = load 'NYSE_dividends' AS (exchange, symbol, date, dividends);`
- Loads data from file and directory

Input and Output: Store
- Pig stores your data on HDFS in a tab-delimited file using PigStorage()

```java
store processed into '/data/examples/processed';
store processed into 'hdfs://nn.acme.com/data/examples/processed';
```
- Pig stores your data to HBase

```java
store processed into 'processed' using HBaseStorage();
```

Relational Operations: foreach
- Pig's projection operator
- Takes a set of expressions
- Applies them to every record in the data pipeline
- Pass to the next operator

```java
A = load 'input' as (user:chararray, id:long, address:chararray, phone:chararray, preferences:map[]);
B = foreach A generate user, id;
```

```java
prices = load 'NYSE_daily' as (exchange, symbol, date, open, high, low, close, volume, adj_close);
gain = foreach prices generate close – open;
gain2 = foreach prices generate $6-$3;
```
- Expression in foreach
- `$` dollar sign + start from 0
- `alldata`: exchange, symbol, date, open, high, low, close, volume, adj_close
- `beginning`: exchange, symbol, date, open
- `middle`: open, high, low, close
- `end`: volume, adj_close
- Relation gain and gain2 will have the same values.
- Positional style reference is useful when the schema is unknown.
Relational Operations: \textit{foreach}

- Arithmetic operators
- Integers and floating-point numbers
  - \(+\), \(-\), \(*\), \(/\)
  - Return value of their own type
  - \(5/2 = 2\)
  - \(5/2.0 = 2.5\)
  - \(x + \text{null} = ?\)
  - For integers the modulo operator \(\%\) is supported

- Binary condition operator
  - \(2 == 2 \rightarrow 4\)
  - \(2 == 3 \rightarrow 4\)
  - \(\text{null} == 2 \rightarrow \text{null}\)
  - \(2 == 2 \rightarrow \text{null}\) type error: both values must be the same type

Using \textit{foreach} for the complex types

- Extract data from complex types
  - Use the projection operators
- 
  \(\#\) for maps

\[
\text{bball} = \text{load} \ 'baseball' \ \text{as} \ (\text{name:chararray}, \ \text{team:chararray}, \ \text{position:bag\{t: (p:chararray)}), \ \text{bat:map\{\}}); \\
\text{avgs} = \text{foreach bball} \ \text{generate} \ \text{bat\#'batting_average'};
\]

- \textit{for} tuples

\[
\begin{align*}
\text{A} & = \text{load} \ 'input' \ \text{as} \ (t:tuple(x:int, y:int)); \\
\text{B} & = \text{foreach A} \ \text{generate} \ t.x, t.y;
\end{align*}
\]

filter

- Select records to be retained in your data pipeline
- Filter contains a predicate
  - If that predicate evaluates to true for a given record, that record will be passed down the pipeline
  - Otherwise, it will not
- Predicates can contain the equality operators
  - \(==, !=, <, >, \geq\) and \(\leq\)
  - For any scalar types
  - \(\text{in}\) and \(\text{like}\) can be used for maps and tuples

Using Boolean operators in filters

\[
\begin{align*}
\text{divs} & = \text{load} \ 'NYSE_dividends' \ \text{as} \ (\text{exchange:chararray}, \ \text{symbol:chararray}, \ \text{date:chararray}, \ \text{dividends:float}); \\
\text{startswithcm} & = \text{filter divs by} \ \text{symbol matches} \ 'CM.*'; \\
\text{Notstartswithcm} & = \text{filter divs by} \ \text{not symbol matches} \ 'CM.*';
\end{align*}
\]
Using null in filters

- “X == null”
  - Does not result in false or true
  - Results null
  - Filters will not pass the data through
- records: 2, null, and 4
  - filter (x == 2): which record will be passed to the next operation of the data pipeline?
  - filter x != 4: which record will be passed to the next operation of the data pipeline?
- Finding values that are not null?
  - Use, is not null

Examples

- in Pig Latin
  daily = load 'NYSE_daily' as (exchange, stock);
grpd = group daily by stock;
cnt = foreach grpd generate group, COUNT(daily);

- in Pig Latin
  daily = load 'NYSE_daily' as (exchange, stock);
grpd = group daily by stock;
store grpd into 'by_group'

Output of group

- The key and the bag of collected records

  -- in Pig Latin
  daily = load 'NYSE_daily' as (exchange, stock);
grpd = group daily by stock;
store grpd into 'by_group'

  Grpd: {group: (exchange:bytearray, stock:bytearray), daily:{exchange:bytearray, stock:bytearray, date:bytearray, dividends:bytearray}}

grouping with multiple keys

- in Pig Latin
  daily = load 'NYSE_daily' as (exchange, stock, date, dividends);
grpd = group daily by (exchange, stock);
avg = foreach grpd generate group, AVG(daily.dividends);
describe grpd;

  -- output
  grpd: (group: (exchange:bytearray, stock:bytearray), daily: (exchange:bytearray, stock:bytearray, date:bytearray, dividends:bytearray))

Skewed results

- We cannot expect that the number of values per key is distributed evenly over the reducers.
  - E.g. group the Index of web pages by the base URL.
  - Certain values such as google.com will appear far more frequently.
  - Some reducers get far more data than others.
  - Your MapReduce job will not finish until all your reducers have finished.
order by

- The order statement sorts your data
- Provides total order
  - Not only in each partition

```sql
daily = load 'NYSE_daily' as (exchange, stock, date, symbol);
bydate = order daily by date;
Bydatensymbol = order daily by date, symbol;
```

- Order causes your data pipeline to go through a reduce phase.

Skew reducing for order

- Skew of the values in data is very common with order
- Pig balances the output across reducers
  - Samples the input of the order statement to get an estimate of the key distribution
  - Builds a partitioner that produces a balanced total order
- Example
  - Sampled data (a,b,e,e,e,e,e,m,q,r,z) and three reducers
  - Partitioner decides:
    - a, b go to reducer 1
    - e goes to reducer 2
    - m, q, r, z go to reducer 3

distinct

- It removes duplicate records
- Works only on entire records.
- Distinct forces a reduce phase
- Distinct uses combiner to remove any duplicate records it can delete in the map phase
- Equivalent to select distinct x in SQL

```sql
daily = load 'NYSE_daily' as (exchange, stock, date, dividends);
divq = distinct daily;
```

join

- Join is one of the popular and core pieces of data processing.
- Join selects records from one input to put together with records from another input.
  - Indicating key for each input
  - If those keys are equal, two rows are joined.

```sql
daily = load 'NYSE_daily' as (exchange, stock, date, adj_close);
divs = load 'NYSE_dividends' as (exchange, symbol, date, dividends);
jnd = join daily by symbol, divs by symbol;
```

join with multiple keys

- Join with multiple keys:
  - Same number of keys
  - Compatible types
  - Implicit cast can be applied

```sql
daily = load 'NYSE_daily' as (exchange, symbol, stock, date, adj_close);
divs = load 'NYSE_dividends' as (exchange, symbol, date, dividends);
jnd = join daily by (symbol, date), divs by (symbol, date);
```

Results of join

- Join preserves the names of the fields
- Join prepends the name of the relation the field came from
  - Uses `::`

```sql
jnd: {daily::exchange:bytearray,
daily::symbol:bytearray, daily::date:bytearray,
daily::open:bytearray, daily::high:bytearray,
daily::low:bytearray, daily::close:bytearray,
daily::volume:bytearray, daily::adj_close:bytearray,
div::exchange:bytearray, div::symbol:bytearray,
div::date:bytearray, div::dividends:bytearray}
```
Outer join

- left/right outer join
  - Records from the left/right relation will be included even when they do not have a match on the right/left side.
- full outer join
  - Records from both relations will be included even when they do not have a match.
- Outer joins are supported only when Pig knows the schema of the data on the left or right side(s)

```pig
daily = load 'NYSE_daily' as (exchange, symbol, stock, date, adj_close);
divs = load 'NYSE_dividends' as (exchange, symbol, date, dividends);
jnd = join daily by (symbol, date) left outer, divs by (symbol, date);
```

Processing join

- Map phase annotates each record with which input it came from
- Pig collects records and cross product between the records from left and right relations.
  - Order the records based on the annotation
  - Load left first to memory
  - Then check items from right relation
  - Plan your join based on the size of relations.

```
pig
```

limit

- Limited number of results

```pig
divs = load 'NYSE_dividends' as (exchange, symbol, date, dividends);
first10 = limit divs 10;
```

Processing limit

- Limit causes an additional reduce phase
  - Collect data and count them
  - Optimizing process
    - Limit the output of each map
    - Apply the limit in the reducer again

sample

- Generates a sample of your data
- Reads through all of your data and returns only a percentage of rows.

```pig
divs = load 'NYSE_dividends' as (exchange, symbol, date, dividends);
some = sample divs 0.1;
```

parallel

- Pig prefers that you tell it how parallel to be.
- The parallel clause can be attached to any relational operator
- It controls only the reduce phase
  - group, order, distinct, join, limit, cogroup and cross

```pig
divs = load 'NYSE_dividends' as (exchange, symbol, date, dividends);
bysymbol = group divs by symbol parallel 10;
```

10 reducers
Using different join implementations

1. Join a small and a large datasets

- Lookup in a smaller input
- Translate a US ZIP code to state and city

At most 100,000 zip codes

Your client information table

Join a small and large datasets

- Forcing a reduce phase that will sort your big file and zip code translation file
- Load zip code file into the memory of each machine, and join by streaming through the large file
- looking up each record in the zip code file

Fragment-replicate join

- You fragment one file and replicate the other

Fragment-replicate join

- Fragment-replicate join supports only inner and left outer joins
- Loads the replicated input into Hadoop’s distributed cache.
- You preload a file onto the local disk of nodes
- Saves time when opening multiple files from the remote machines
- Multiple map tasks are located on the same machine
- Files in the distributed cache are shared between those instances

Processing Fragment-replicate join

- Pig runs a map-only MapReduce job to preprocess the file and get it ready for loading into the distributed cache.
- Any filter or foreach jobs between load and join
- Join will be done in a second map-only job.
Using different join implementations
2. Joining Skewed Data

Using different join implementations
2. Joining Skewed Data

Joining skewed data

- The number of records per key
- Building a map of the Web and joining by the domain of the URL
  - e.g. google.com
- If there are few keys that have much more records than other keys
  - Pig provides skew join

Skew join

- First MapReduce job
  - Sampling one input for the join
  - Identifies any keys that have so many records that it will not fit into memory
- Second MapReduce job
  - Join
  - Except for the keys identified in sampling, it does a standard join
  - Keys identified in the sampling stage
  - Split across the appropriate number of reducers

Skew join

- Cities in users
  - 20 in Barcelona
  - 300,000 in New York
  - 3,500 in Portland
- 75,000 records can fit into memory on each reducer

Skew join

users = load 'users' as (name:chararray,
city:chararray details:chararray);
cinfo = load 'cityinfo' as (city:chararray,
population:int, info:chararray);
join = join cinfo by city, users by city using 'skewed';

- New York would be identified as a key that needed to be split across reducers
  - Others will be treated as in a default join
- Records from cityinfo with New York as a key will be duplicated and sent to reducers
  - Note that the second input (here, users) is the one that will be sampled
  - The first input (here, cinfo) will be replicated across reducers
- This algorithm addresses skew in only one input

Skew join

users = load 'users' as (name:chararray,
city:chararray details:chararray);
cinfo = load 'cityinfo' as (city:chararray,
population:int, info:chararray);
join = join cinfo by city, users by city using 'skewed';

- New York would be identified as a key that cannot fit into one memory
- Records from users with New York as the key are split between two reducers
- Records from city with New York as a key are duplicated and sent to both of reducers.
What if both inputs have skew?

- This algorithm will still work
  - It will be slow
- Before Pig introduced skew join in v.0.4
  - If data was skewed on both sides, the job could not be performed.
  - It was not possible to fit all the records for the high-cardinality key values in memory for either side

Skew join can be done on inner or outer joins
- Multiway joins should be broken into a series of joins

Why don't we use skew join all of the time?
- Small performance penalty
  - Input must be sampled
  - This adds ~5% to the latency

Using different join implementations
3. Merge Join

Joining sorted data
- Common database join (sort-merge join)
  - Sort both inputs on the join key
  - Walk through both inputs together
  - Join
- MapReduce join
  - Requires a full MapReduce job
  - Not more efficient than the default join

What if the input data is already sorted?

Processing merge join
- Run a MapReduce job to sample data from NYSE_dividends_sorted
  - Builds an index
  - Identifies the join keys
- Second MapReduce job
  - Takes the first input, 'NYSE_daily_sorted'
  - Looks for the last entry that is less than its value of symbol
  - Opens NYSE_dividends_sorted at the corresponding block for that entry
  - (CA,1), (CHY,2), (CP,3) the first symbol in a given map's input split of NYSE_daily_sorted was CJA,
  - Open the block 2 of NYSE_dividends_sorted.
  - Find matching and collect data

Without reduce phase: more efficient than a default join