Big Data
Session 4

Nick Kadochnikov

AGENDA

- Pig Latin
  - Pig scripting
  - Pig data types
  - Pig and HCatalog
  - Pig UDF

❖ Assignment 3: load data into Hadoop and create summary statistics using Pig

Pig : Building High-Level Dataflows over Map-Reduce

Introduction

- What is Pig?
  - An open-source high-level dataflow system
  - Provides a simple language for queries and data manipulation, Pig Latin, that is compiled into map-reduce jobs that are run on Hadoop
  - Pig Latin combines the high-level data manipulation constructs of SQL with the procedural programming of map-reduce

Adopted from: Peter, Gang and Ronie

Language Features

- Several options for user-interaction
  - Interactive mode (console)
  - Batch mode (prepared script files containing Pig Latin commands)
  - Embedded mode (execute Pig Latin commands within a Java program)
- Built primarily for scan-centric workloads and read-only data analysis
  - Easily operates on both structured and schema-less, unstructured data
  - Transactional consistency and index-based lookups not required
  - Data curation and schema management can be overkill
- Flexible, fully nested data model
- Extensive UDF support
  - Currently must be written in Java
  - Can be written for filtering, grouping, per-tuple processing, loading and storing

Adopted from: Peter, Gang and Ronie

Pig Latin vs. SQL

- Pig Latin is procedural (dataflow programming model)
  - Step-by-step query style is much cleaner and easier to write and follow than trying to wrap everything into a single block of SQL

```
insert into ValuableClicksPerDMA
select dma, count(*)
from geoinfo join (
                select name, ipaddr
                from users join clicks on (users.name = clicks.user)
                where value > 0;
            ) using ipaddr
group by dma;
```

```
Users                = load 'users' as (name, age, ipaddr);
Clicks               = load 'clicks' as (user, url, value);
ValuableClicks       = filter Clicks by value > 0;
UserClicks           = join Users by name, ValuableClicks by user;
Geoinfo              = load 'geoinfo' as (ipaddr, dma);
UserGeo              = join UserClicks by ipaddr, Geoinfo by ipaddr;
ByDMA                = group UserGeo by dma;
ValuableClicksPerDMA = foreach ByDMA generate group, COUNT(UserGeo);
store ValuableClicksPerDMA into 'ValuableClicksPerDMA';
```


Adopted from: Peter, Gang and Ronie

Source: http://developer.yahoo.net/blogs/hadoop/2010/01/comparing_pig_latin_and_sql_fo.html

Pig Latin vs. SQL (continued)

- Lazy evaluation (data not processed prior to STORE or DUMP command)
- Data can be stored at any point during the pipeline
- An execution plan can be explicitly defined

– No need to rely on the system to choose the desired plan via optimizer hints

- Pipeline splits are supported

– SQL requires the join to be run twice or materialized as an intermediate result

```
Users          = load 'users' as (name, age, gender, zip);
Purchases      = load 'purchases' as (user, purchase_price);
UserPurchases = join Users by name, Purchases by user;
GeoGroup       = group UserPurchases by zip;
GeoPurchase    = foreach GeoGroup generate group, SUM(UserPurchases.purchase_price) as sum;
ValuableGeos   = filter GeoPurchase by sum > 1000000;
store ValuableGeos into 'byzip';
DemoGroup      = group UserPurchases by (age, gender);
DemoPurchases = foreach DemoGroup generate group, SUM(UserPurchases.purchase_price) as sum;
ValuableDemos = filter DemoPurchases by sum > 100000000;
store ValuableDemos into 'byagegender';
```


Adopted from: Peter, Gang and Ronie

Source: http://developer.yahoo.net/blogs/hadoop/2010/01/comparing_pig_latin_and_sql_for_html

Data Model

- Supports four basic types
  - Atom: a simple atomic value (*int*, *long*, *double*, *string*)
    - ex: ‘Peter’
  - Tuple: a sequence of fields that can be any of the data types
    - ex: (‘Peter’, 14)
  - Bag: a collection of tuples of potentially varying structures, can contain duplicates
    - ex: {(‘Peter’), (‘Bob’, (14, 21))}
  - Map: an associative array, the key must be a *chararray* but the value can be any type

Adopted from: Peter, Gang and Ronie

Data Model (continued)

- By default Pig treats undeclared fields as *bytearrays* (collection of uninterpreted bytes)
- Can infer a field's type based on:
  - Use of operators that expect a certain type of field
  - UDFs with a known or explicitly set return type
  - Schema information provided by a LOAD function or explicitly declared using an AS clause
- Type conversion is lazy

Adopted from: Peter, Gang and Ronie

Pig Latin

- FOREACH-GENERATE (per-tuple processing)
  - Iterates over every input tuple in the bag, producing one output each, allowing efficient parallel implementation

```
expanded_queries = FOREACH queries GENERATE
                   userId, expandQuery(queryString);
```

  - Expressions within the GENERATE clause can take the form of the any of these expressions

$t = \left(\text{'alice'}, \left\{\begin{array}{l}(\text{'lakers', 1})\\(\text{'iPod', 2})\end{array}\right\}, [\text{'age'} \rightarrow 20]\right)$

Let fields of tuple `t` be called `f1, f2, f3`

Expression Type	Example	Value for `t`
Constant	`'bob'`	Independent of `t`
Field by position	`$0`	`'alice'`
Field by name	`f3`	$\left[\text{'age'} \rightarrow 20\right]$
Projection	`f2.$0`	$\left\{\begin{array}{l}(\text{'lakers'})\\(\text{'iPod'})\end{array}\right\}$
Map Lookup	`f3#'age'`	20
Function Evaluation	`SUM(f2.$1)`	`1 + 2 = 3`
Conditional Expression	`f3#'age'>18? 'adult':'minor'`	`'adult'`
Flattening	`FLATTEN(f2)`	`'lakers', 1` `'iPod', 2`

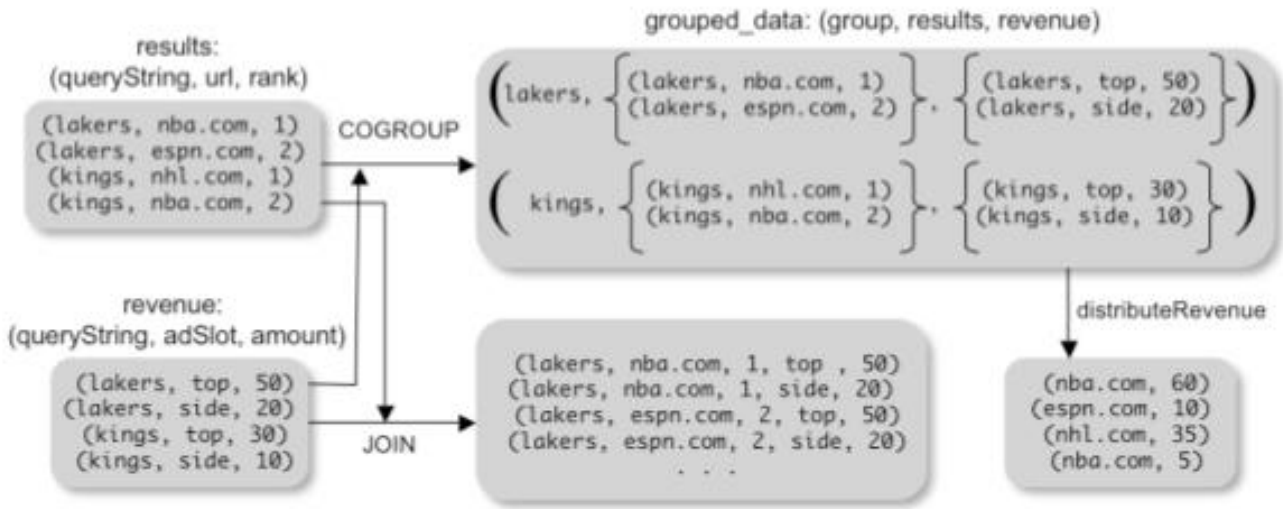
Adopted from: Peter, Gang and Ronie

Pig Latin (continued)

- (CO)GROUP vs. JOIN
  - COGROUP takes advantage of nested data structure (combination of GROUP BY and JOIN)
  - User can choose to go through with cross-product for a join or perform aggregation on the nested bags

```
grouped_data = COGROUP results BY queryString,
               revenue BY queryString;

join_result = JOIN results BY queryString,
              revenue BY queryString;
```

```
temp_var = COGROUP results BY queryString,
           revenue BY queryString;
join_result = FOREACH temp_var GENERATE
              FLATTEN(results), FLATTEN(revenue);
```



Adopted from: Peter, Gang and Ronie

Pig Latin (continued)

- LOAD / STORE
  - Default implementation expects/outputs to tab-delimited plain text file

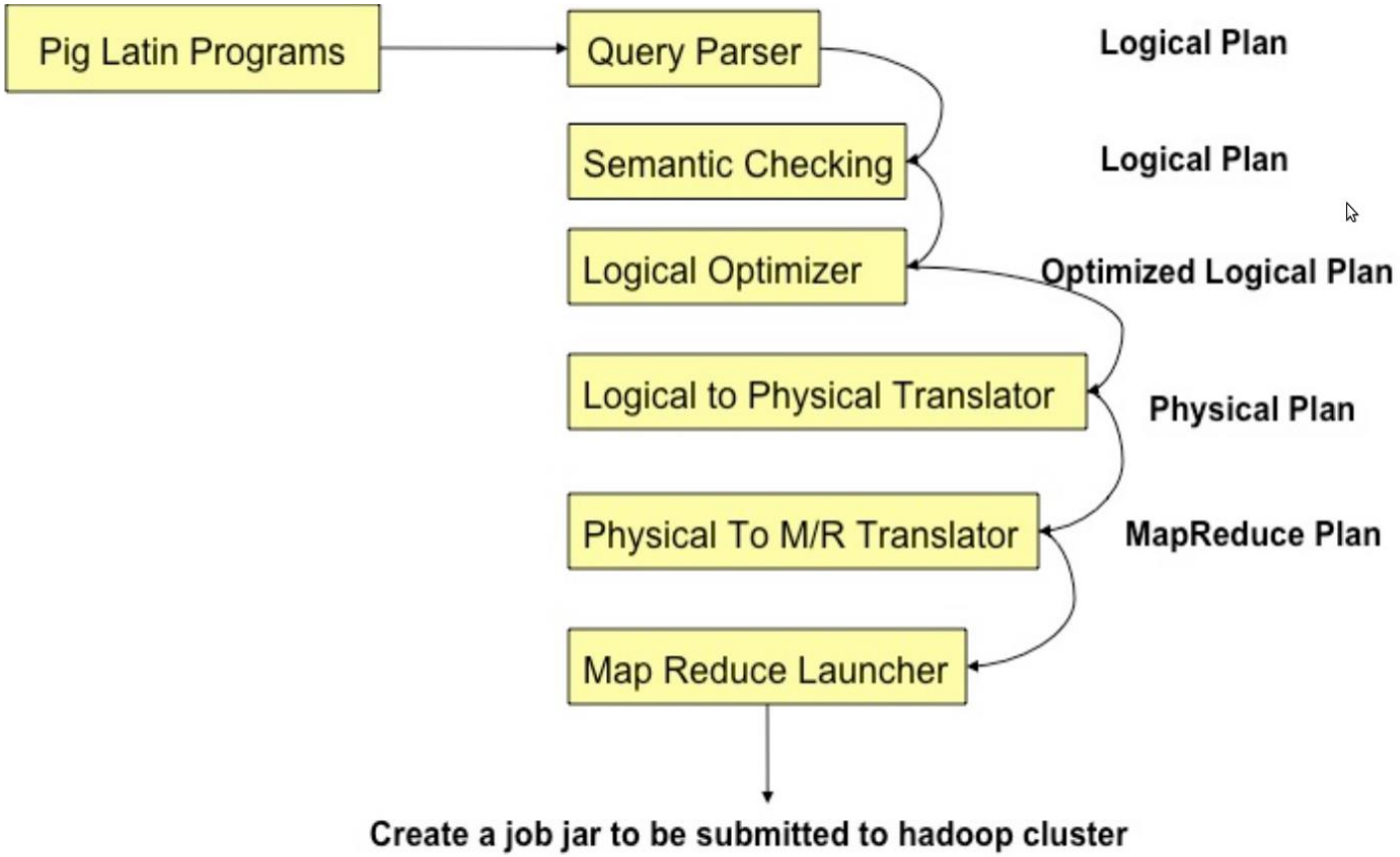
```
queries = LOAD 'query_log.txt'
          USING myLoad()
          AS (userId, queryString, timestamp);
```

```
STORE query_revenues INTO 'myoutput'
      USING myStore();
```

- Other commands
  - FILTER, ORDER, DISTINCT, CROSS, UNION
- Nested operations
  - FILTER, ORDER and DISTINCT can be nested within a FOREACH statement to process nested bags within tuples

Adopted from: Peter, Gang and Ronie

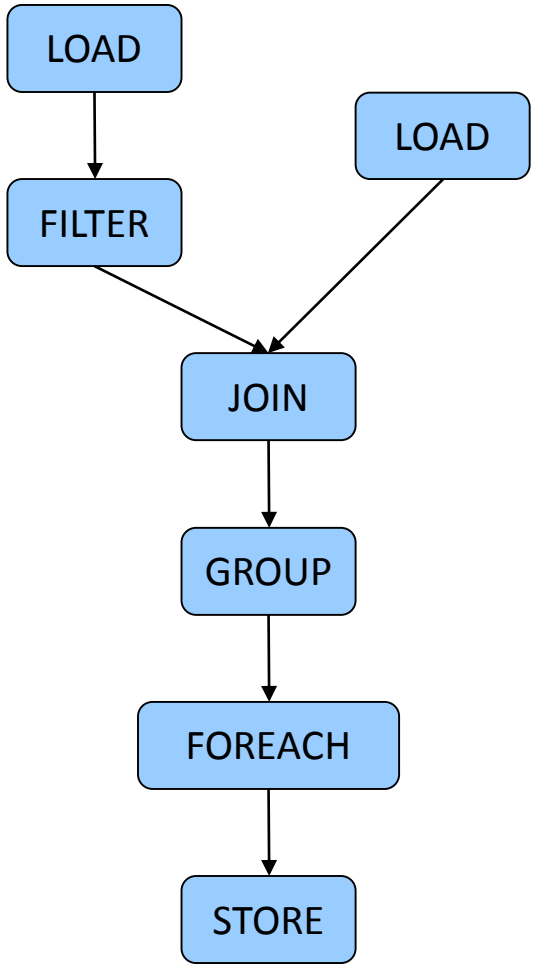
COMPILATION



Adopted from: Peter, Gang and Ronie

Logical Plan

```
A=LOAD 'file1' AS (x, y, z);

B=LOAD 'file2' AS (t, u, v);

C=FILTER A by y > 0;

D=JOIN C BY x, B BY u;

E=GROUP D BY z;

F=FOREACH E GENERATE
      group, COUNT(D);

STORE F INTO 'output';
```

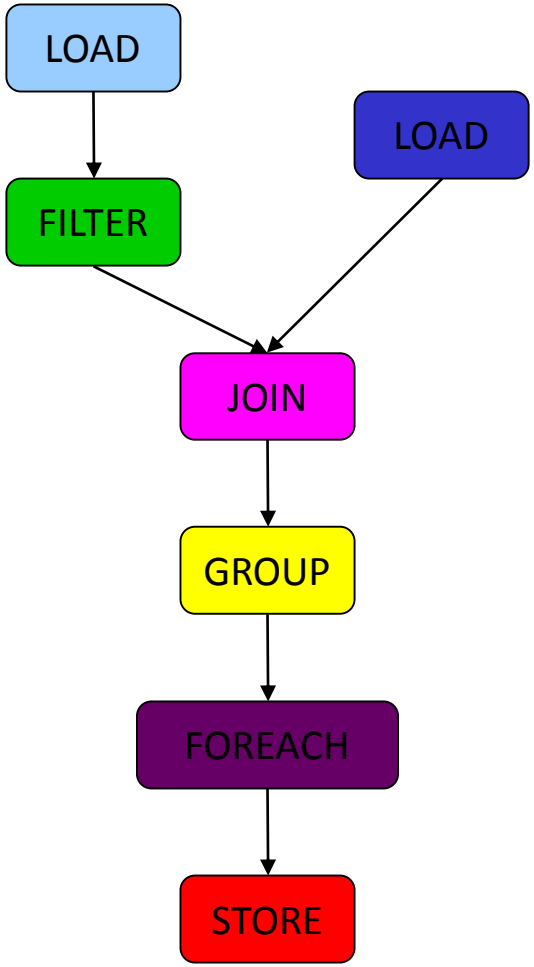


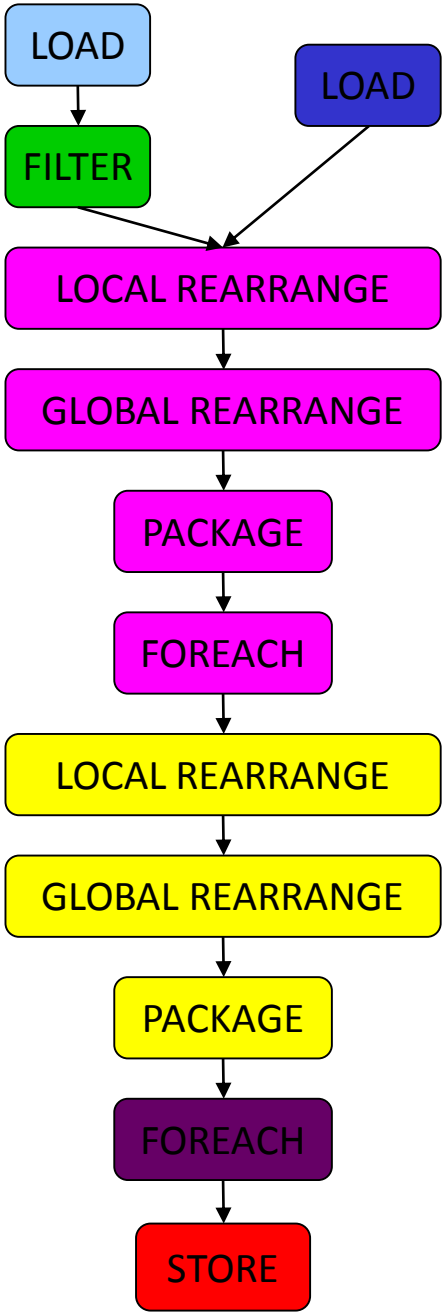
Adopted from: Peter, Gang and Ronie

Physical Plan

- 1:1 correspondence with most logical operators
- Except for:
  - DISTINCT
  - (CO)GROUP
  - JOIN
  - ORDER

Adopted from: Peter, Gang and Ronie



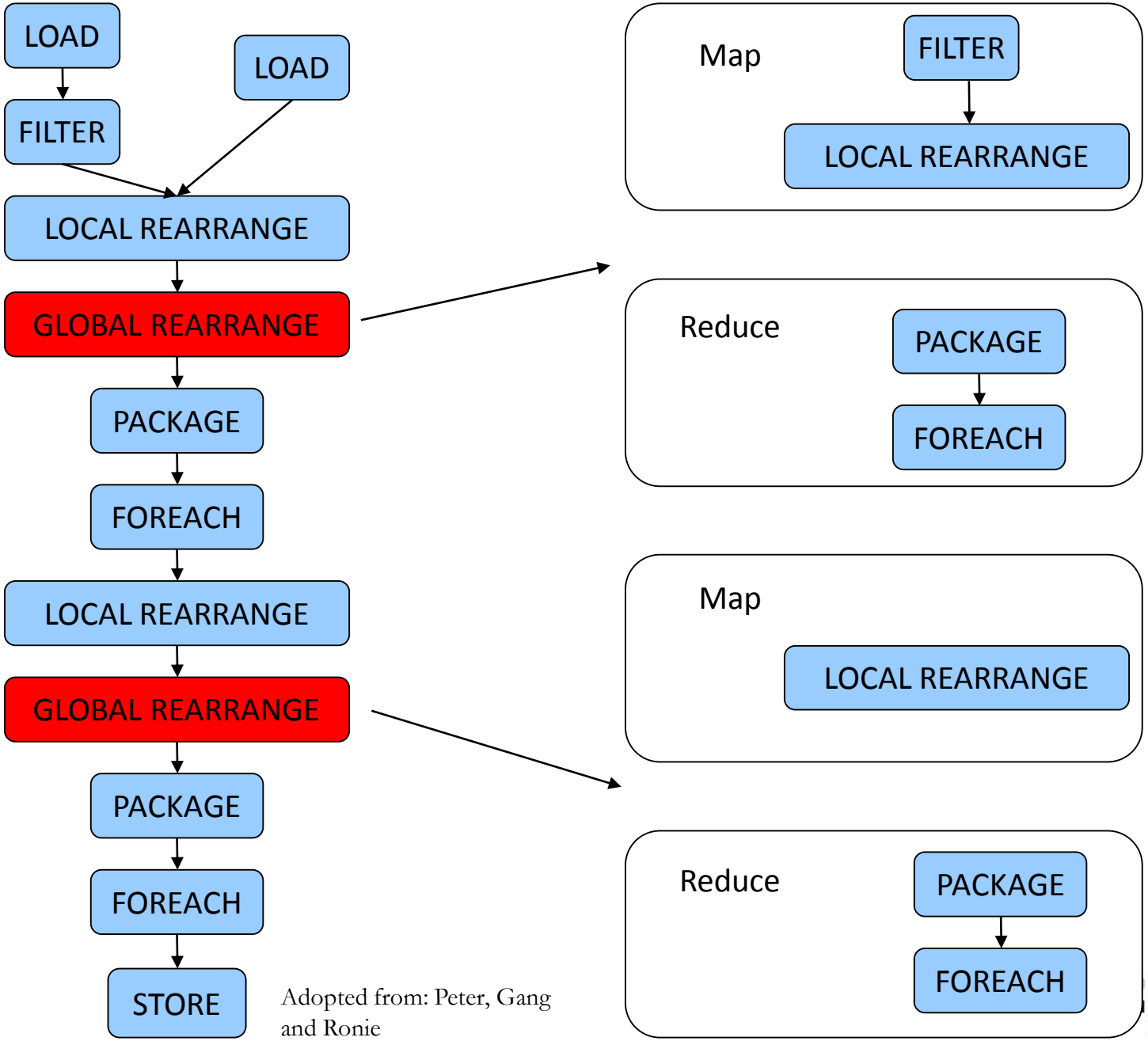


Adopted from: Peter, Gang and Ronie

MapReduce Plan

- Determine MapReduce boundaries
  - GLOBAL REARRANGE
- Some operations are done by MapReduce framework
- Coalesce other operators into Map & Reduce stages
- Generate job jar file

Adopted from: Peter, Gang and Ronie



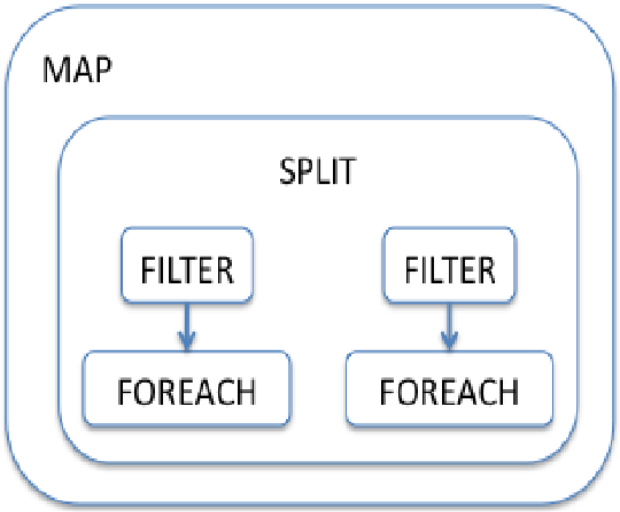
Adopted from: Peter, Gang and Ronie

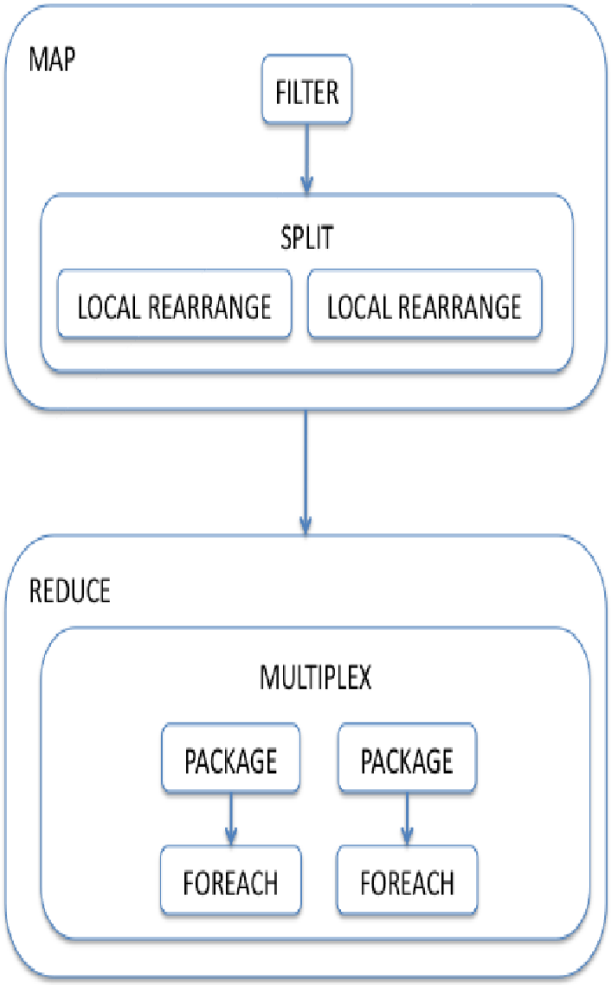
Branching Plans

- Read the dataset once and process it in multiple ways
- Good
  - Eliminate the cost to read it multiple times
- Bad
  - Reduce the amount of memory for each stream

Adopted from: Peter, Gang and Ronie

BRANCHING PLANS





Adopted from: Peter, Gang and Ronie

Interactive Shell

- Pig interactive shell is called "grunt"
  - Type **pig** in your telnet command
  - CTRL-D to exit

To run interactive shell with Hive tables:

- In RCC environment you don't need to specify anything additional. Depending on the environment you might need to explicitly declare "-useHCatalog"
  - pig –useHCatalog
  - pig -useHCatalog pigscript.txt

CASE SENSITIVITY:

- Case sensitivity:
  - Keywords in Pig Latin are not case-sensitive; for example, LOAD is equivalent to load.
  - But relation and field names are. So A = load 'foo'; is not equivalent to a = load 'foo';.
  - UDF names are also case-sensitive, thus COUNT is not the same UDF as count.

COMMENTS

- A = load 'foo'; --this is a single-line comment
- /*
- * This is a multiline comment.
- */
- B = load /* a comment in the middle */'bar';

PIG LOAD OPERATOR

- The LOAD operator is used to read data from a source and place that data into a relation.
- Pig supplies four such load functions and you can supply your own as well.
  - PigStorage. This default loader reads data that is in a delimited format with the default delimiter being the tab character (another delimiting character can be used as well - supplied in single quotes).
  - The TextLoader reads in a line of text and this line of text is placed into a single tuple.
  - The BinStorage function is used to read data in a machine readable format. This is used by Pig internally to store temporary data.
  - There is also a JsonLoader. This loads data that is in a standard JSON format.

LOAD INTO PIG

- Loading tab-delimited file

A = load '/datadir/datafile' using PigStorage('\t');

  - will read data from a file, datafile located in the /datadir.

A = load '/datadir/datafile' using PigStorage(',') as (f1:int,f2:chararray, f3:float);

  - would read a comma delimited file

- If you do not supply a schema for the PigStorage function, then the fields are not named and all fields default to bytearray for their data type.
- The JsonLoader function requires a schema.

FIELDS REFERENCE AND HIVE

- Can read entire directory or from Hive or HBASE table
  - Don't need to specify the schema then (will read columns from Hive)
- Fields are assigned names through the use of a schema and fields may be referenced by their names
  - Or by their position specifying a $ and a number, where field numbers start with zero.
  - With a schema of (name: chararray, age:int, salary:float), the *age* field could be reference by position as $1
  - Or by the name age
- If a schema is not specified when data is loaded, then the fields can only be referenced by position.
- Both fields and relations are case sensitive

Pig Data Types and Operators

- http://www.qubole.com/pig-function-cheat-sheet/

Output and Result Storage

- The STORE operator saves results to a file. It uses built-in functions, similar to the LOAD operator, to determine how the output data is to be formatted.
  - PigStorage the default format and the tab character is the default delimiter.
  - BinStorage is used by Pig for temporary files.
  - PigDump stores the data as tuples in a humanreadable UTF-8 format.
  - JsonStorage writes data in a JSON format.
- The format of the STORE operator is STORE alias into '<directory' [using function()];
- The specified directory will be created. If it already exits, then the operation will fail. Files written into the directory will be of the format partnnnnn.
  - STORE B INTO '/user/kadochnikov/Books/BX_Book_Ratings_01';
  - When writing to a filesystem, processed will be a directory with part files rather than a single file. Number of part files depends on the parallelism of the last job before the store. If it has reduces, it will be determined by the parallel level set for that job. If it is a map-only job, it will be determined by the number of maps, which is controlled by Hadoop and not Pig.
- DUMP operator writes the results to the console, similar to "print"

FILTER AND ORDER BY

- The FILTER operator selects records based upon a specified criteria
  - Similar to where in SQL

- The ORDER BY operator sorts a relation on one or more fields.
  - Default is ASC, you can specify DESC
  - Multiple comma separated fields may be specified, each specifying their own sort sequence.

GROUPING AND FOREACH

- COGROUP/GROUP
  - Similar to Group By in SQL
  - The GROUP and COGROUP operators are actually the same operator but, by convention, the COGROUP operator is used when grouping multiple relations at the same time.
- FOREACH
  - Generates transformation of data for each row as specified

Explicit Field Order

my_data = LOAD 'file.tsv' AS (field1:int, field2:chararray, field3:float, field4:int);

- FOREACH my_data GENERATE $0, $2;
  - (field1, field3)
- FOREACH my_data GENERATE field2..field4;
  - (field2, field3, field4)
- FOREACH my_data GENERATE field2..;
  - (field2, field3, field4)
- FOREACH my_data GENERATE *;
  - (field1, field2, field3, field4)

DISTINCT, UNION, JOIN

- All similar to SQL

Thank You!