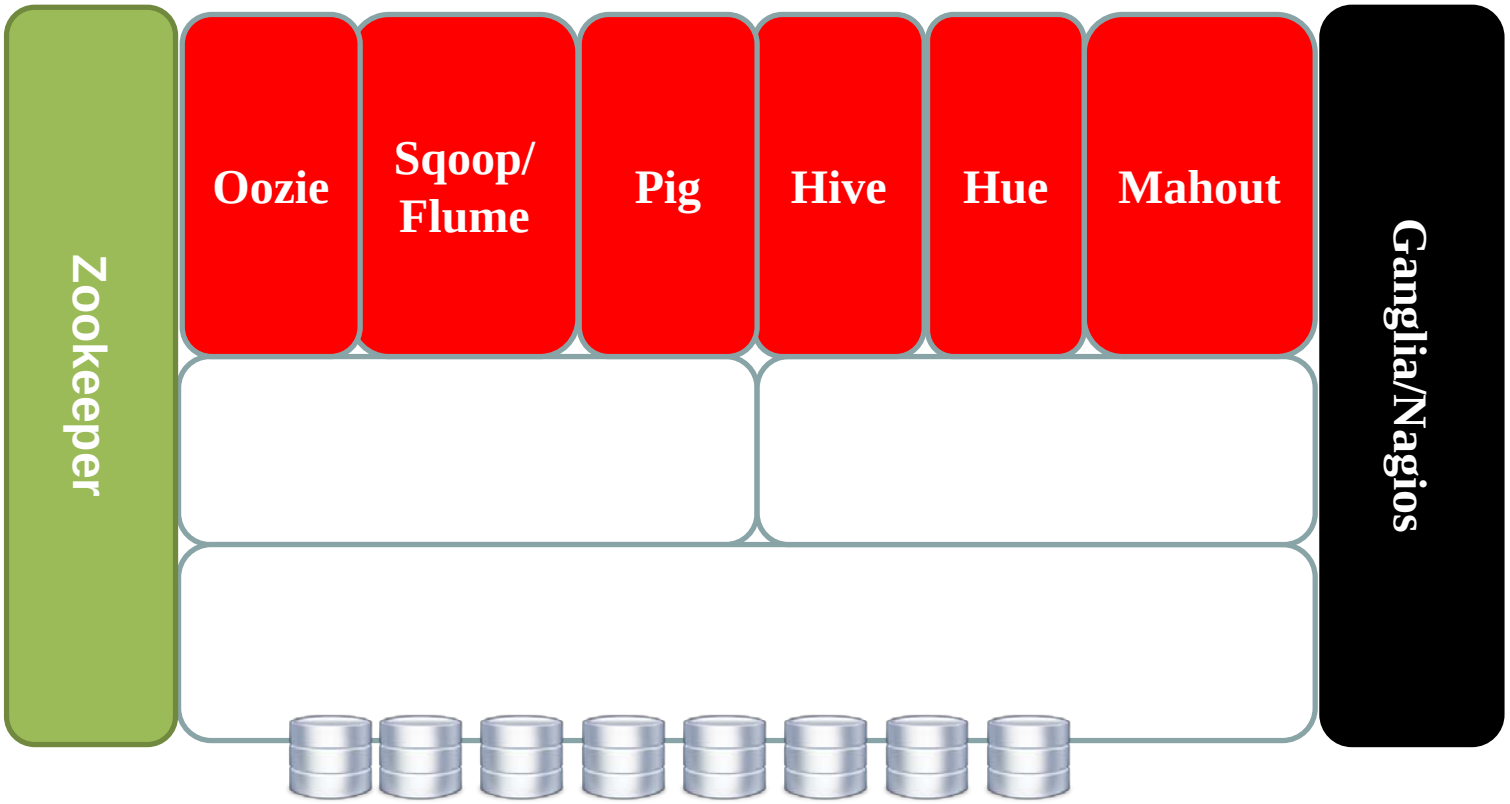
BIG DATA實務運算
APACHE PIG

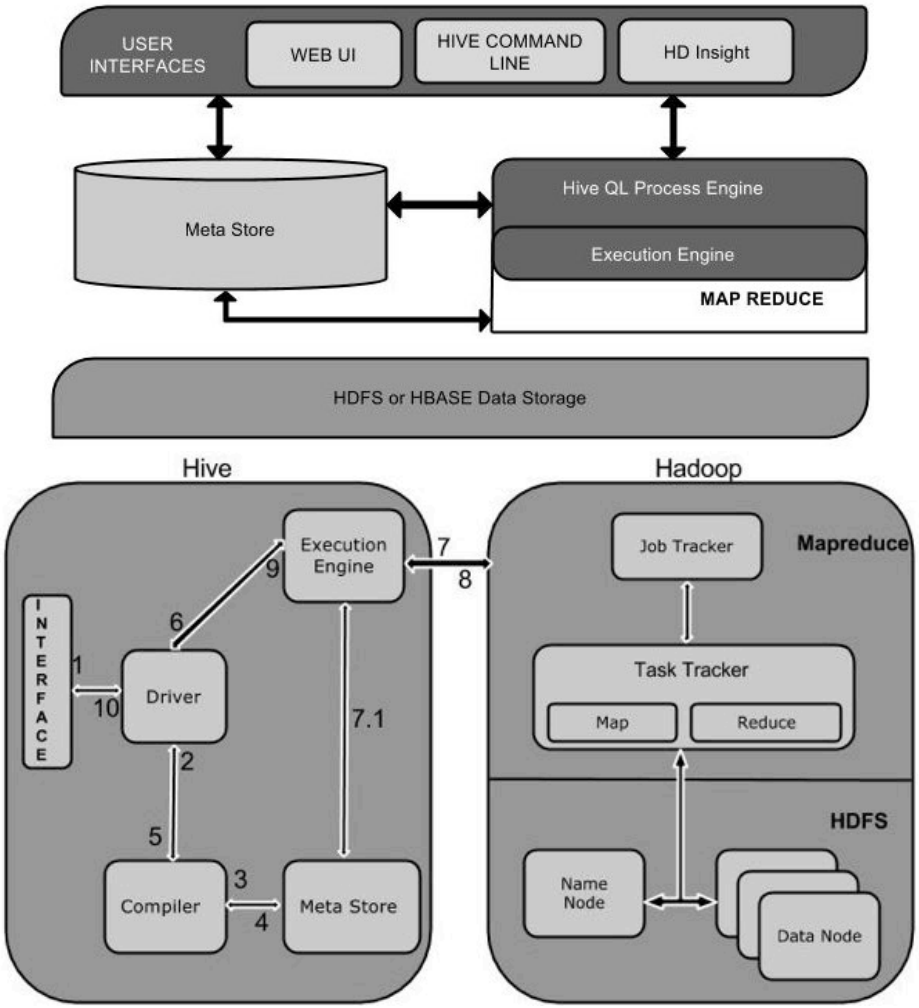
國立臺北科技大學資訊工程系郭忠義

Hadoop Family

- Open Source Software + Hardware Commodity



Hadoop Hive


USER INTERFACES
WEB UI
HIVE COMMAND LINE
HD Insight
Meta Store
Hive QL Process Engine
Execution Engine
MAP REDUCE
HDFS or HBASE Data Storage
Hive
Hadoop
Execution Engine
Job Tracker
Mapreduce
INTERFACE
Driver
Task Tracker
Map
Reduce
HDFS
Compiler
Meta Store
Name Node
Data Node
1
2
3
4
5
6
7
7.1
8
9
10

Apache Pig

- Consists of a high-level language for expressing data analysis, which is called “Pig Latin”
- Combined with the power of Hadoop and the MapReduce framework
- A platform for analyzing large data sets
- Pig Latin
  - 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

Example - Data Analysis Task

- Q : Find user who tend to visit “good” pages.

Visits

user	url	time
Amy	www.cnn.com	8:00
Amy	www.crap.com	8:05
Amy	www.myblog.com	10:00
Amy	www.flickr.com	10:05
Fred	cnn.com/index.htm	12:00

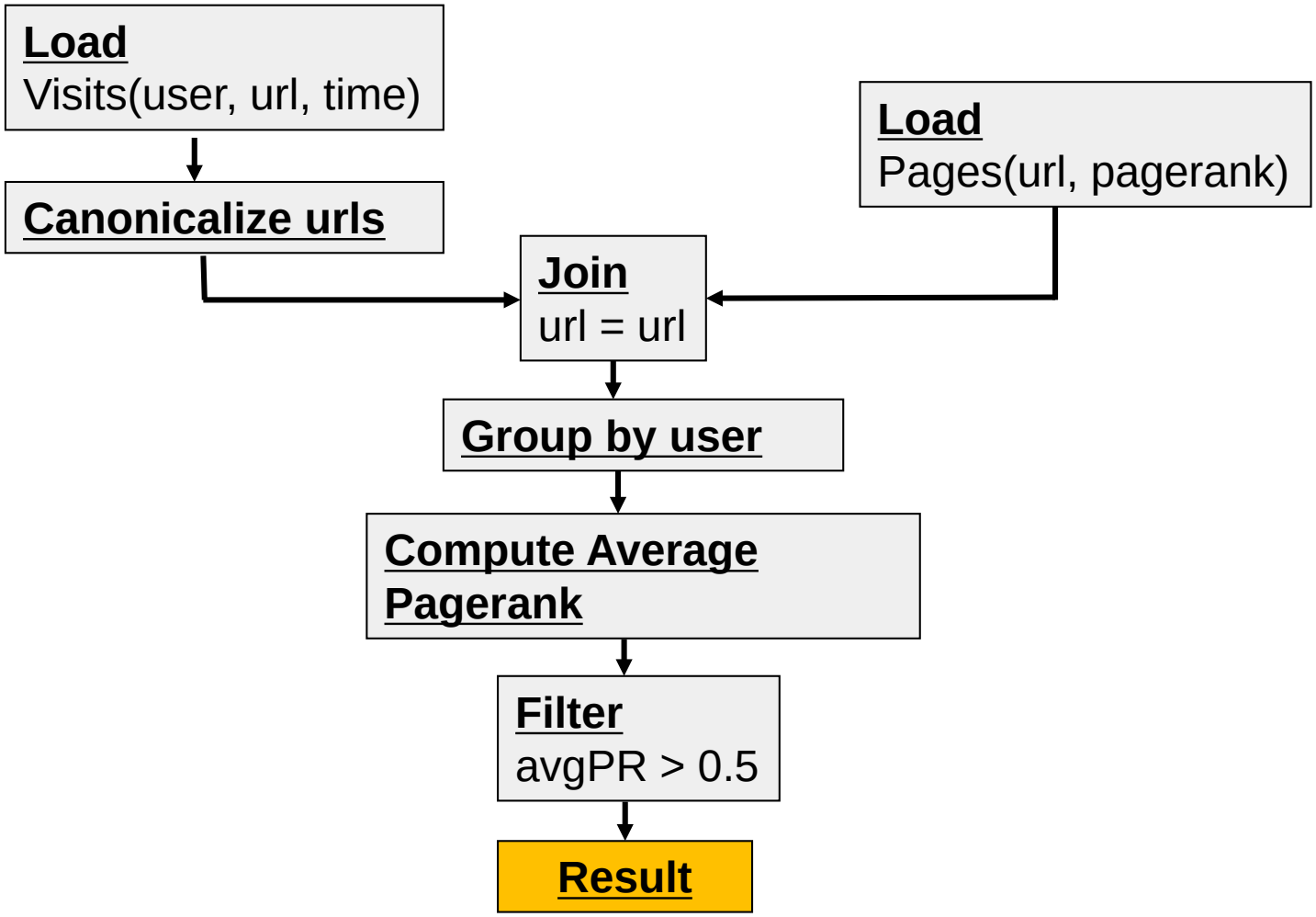
⋮

Pages

url	pagerank
www.cnn.com	0.9
www.flickr.com	0.9
www.myblog.com	0.7
www.crap.com	0.2

⋮

Conceptual Dataflow



By MapReduce

- Low-level programing.
  - Requiring more effort to understand and maintain
- Write join code yourself.
- Effort to manage multiple MapReduce jobs.

By Pig

```
Visits = load  ‘/data/visits’ as (user, url, time);
Visits = foreach Visits generate user, Canonicalize(url),
time;

Pages = load  ‘/data/pages’ as (url, pagerank);

VP = join  Visits by url, Pages by url;
UserVisits = group  VP by user;
UserPageranks = foreach UserVisits generate user,
AVG(VP.pagerank) as avgpr;
GoodUsers = filter  UserPageranks by avgpr > ‘0.5’;

 store  GoodUsers into '/data/good_users';
```

Typically a Pig script is **5%** of the code of native map/reduce written in about **5%** of the time.

Why Pig?

- Ease of programming.
  - script language(Pig Latin)
  - Pig not equals to SQL
- Increases productivity
  - 10 lines of Pig Latin ≈ 200 lines of Java.
  - What took 4 hours to write in Java took 15 minutes in Pig Latin.
  - Provide some common functionality (such as join, group, sort, etc)
  - Extensibility - UDF

More Comments

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.

Key Features

- Key type : field , tuple, and bag
- Schema : way to assign name & type of a value
- Operators : useful built-in operators
  - LOAD/STORE
  - GROUP/COGROUP
  - JOIN
  - FILTER
  - FOREACH
- Tools : DUMP & DESCRIBE

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

Running Pig

- You can run Pig (execute Pig Latin statements and Pig commands) using various modes.

	Local Mode	MapReduce Mode
Interactive Mode	Yes	Yes
Batch Mode	Yes	Yes

Execution Modes

- Local mode
  - Definition : all files are installed and run using your local host and file system.
  - How : using the -x flag (pig -x local).
- Mapreduce mode
  - Definition : access to a Hadoop cluster and HDFS installation.
  - How : the mode is the default mode; you can, but don't need to, specify it using the -x flag (pig OR pig -x mapreduce).

Execution Modes (Cont.)

- By Pig command :

```
/* local mode */
$ pig -x local ...
/* mapreduce mode */
$ pig ...
or
$ pig -x mapreduce ...
```

- By Java command :

```
/* local mode */
$ java -cp pig.jar org.apache.pig.Main -x local ...
/* mapreduce mode */
$ java -cp pig.jar org.apache.pig.Main ...
or
$ java -cp pig.jar org.apache.pig.Main -x mapreduce ...
```

Interactive Mode

- Definition : using the Grunt shell.
- How : Invoke the Grunt shell using the "**pig**" command and then enter your Pig Latin statements and Pig commands interactively at the command line.
- Example:

```
grunt> A = load 'passwd' using PigStorage(':');
grunt> B = foreach A generate $0 as id;
grunt> dump B;
```

Batch Mode

- Definition : run Pig in batch mode.
- How : Pig scripts + Pig command
- Example:

```
/* id.pig */

A = load 'passwd' using PigStorage(':');
B = foreach A generate $0 as id;
store B into ‘id.out’;
```

```
$ pig -x local id.pig
```

Batch Mode (Cont.)

- Pig scripts
  - Place Pig Latin statements and Pig commands in a single file.
  - Not required, but is good practice to identify the file using the ***.pig** extension.
  - Allows “parameter substitution”
  - Support comments
    - For single-line : use - -
    - For multi-line : use /* …. */

Utility Commands

- **exec** : Run a Pig script.
  - ✓ Syntax : exec [–param param_name = param_value] [–param_file file_name] [script]

```
grunt> cat myscript.pig
a = LOAD 'student' AS (name, age, gpa);
b = ORDER a BY name;

STORE b into '$out';

grunt> exec –param out=myoutput myscript.pig;
```

Utility Commands

- **run** : Run a Pig script.
  - ✓ Syntax : run [–param param_name = param_value] [–param_file file_name] script

```
grunt> cat myscript.pig
b = ORDER a BY name;
c = LIMIT b 10;

grunt> a = LOAD 'student' AS (name, age, gpa);
grunt> run myscript.pig
grunt> d = LIMIT c 3;
grunt> DUMP d;

(alice,20,2.47)
(alice,27,1.95)
```

Others

- File commands
  - cat, cd, copyFromLocal, copyToLocal, cp, ls, mkdir,mv, pwd, rm, rmf, exec, run
- Utility commands
  - help
  - quit
  - kill ***jobid***
  - set debug [on|off]
  - set job.name '***jobname***'

LEARNING PIG FUNDAMENTAL

Data Type – Simple Type

Type	Description	Example
int	Signed 32-bit integer	10
long	Signed 64-bit integer	10L
float	32-bit floating point	10.5F
double	64-bit floating point	10.5
chararray	Character array (string) in Unicode UTF-8 format	Hello world
bytearray	Byte array (blob)	
boolean	boolean	true/false (case insensitive)

Data Type – Complex Type

- A tuple just like a row with one or more fields.
  - Each field can be any data type
  - Any field may/may not have data
  - Syntax : (field [, field …])

Type	Description	Example
tuple	An ordered set of fields.	(20, true)

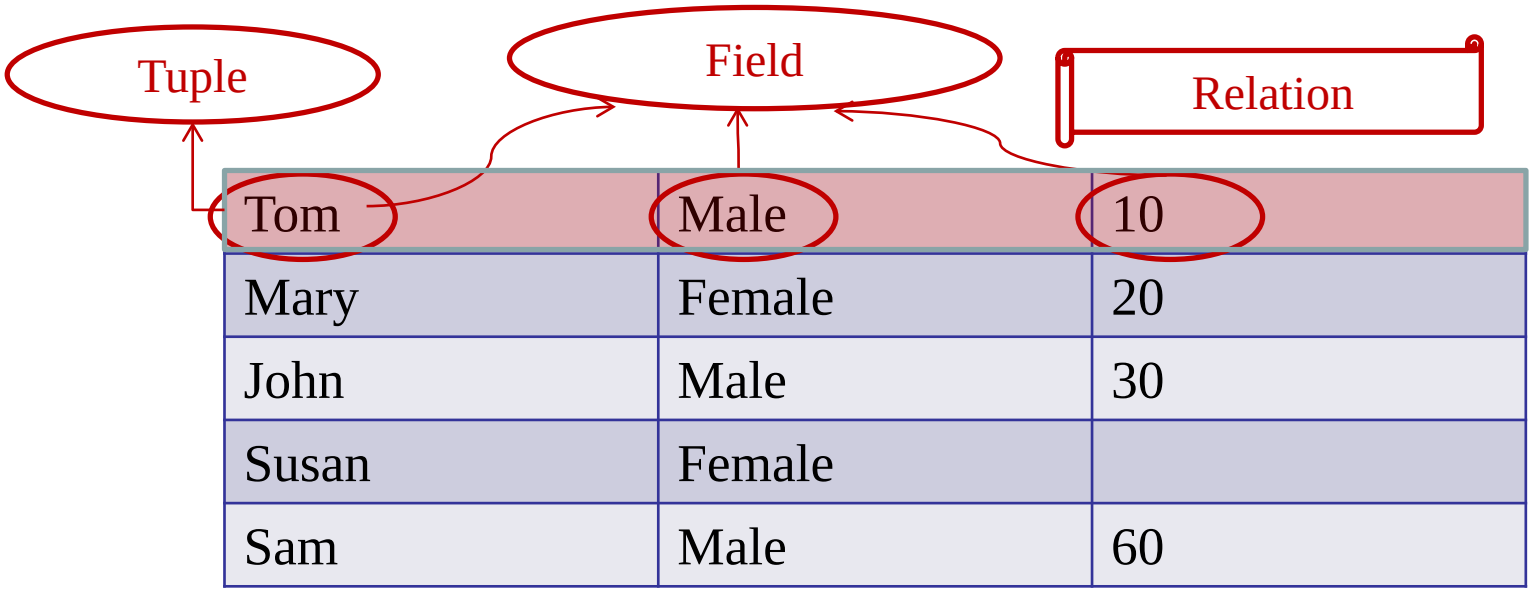
- A bag contains multiple tuples. (1~n)
  - Outer bag/Inner bag
  - Syntax : { tuple [, tuple …] }

Type	Description	Example
bag	An collection of tuples.	{(1, 2), (3, 4)}

Data Type – Complex Type

- Just like other language, key must be unique in a map.
  - Syntax : [key#value <, key#value …>]

Type	Description	Example
map	A set of key value pairs.	[open# trendmicro]



Data Type – Complex Type

- Conclusion?
  - A field is a piece of data.
  - A tuple is an ordered set of fields.
  - A bag is a collection of tuples.
  - A relation is a bag (more specifically, an outer bag).

PIG LATIN STATEMENT WORK WITH RELATIONS!!

Schemas

- Used as keyword "**AS**" clause.
- Defined with LOAD/STREAM/FOREACH operator.
- Enhance better parse-time error checking and more efficient code execution.
- **Optional** but **encourage** to use.

Schema

- Legal format:
  - ✓ includes both the field name and field type
  - ✓ includes the field name only, lack field type
  - ✓ not to define a schema

```
A = LOAD 'data' AS (name:chararray, age:int, male:boolean);
```

```
A = LOAD ‘data’ AS (name, age, male);
```

```
A = LOAD ‘data’ ;
```

John	18	true
Mary	19	false
Bill	20	true
Joe	18	true

Schema – Example 2

(3,8,9) (mary,19)
(1,4,7) (john,18)
(2,5,8) (joe,18)

```
A = LOAD ‘data’ AS (F:tuple(f1:int, f2:int, f3:int),
T:tuple(t1:chararray, t2:int));
```

```
A = LOAD ‘data’ AS (F:(f1:int, f2:int, f3:int),
T:(t1:chararray, t2:int));
```

{(3,8,9)}
{(1,4,7)}
{(2,5,8)}

```
A = LOAD 'data' AS (B: bag {T: tuple(t1:int, t2:int, t3:int)});
```

```
A = LOAD 'data' AS (B: {T: (t1:int, t2:int, t3:int)});
```

Schema – other cases

- What if field name/field type is not defined?
  - ✓ Lacks field name : can only refer to the field using positional notation. ($0, $1…)
  - ✓ Lacks field type : value will be treated as bytearray. you can change the default type using the cast operators.

John	18	true
Mary	19	false
Bill	20	true
Joe	18	true

```
grunt> a = LOAD ‘data';
grunt> b = FOREACH  a GENERATE $0;
```

Standard Flow

- Loading Data
- Working with Data
- Storing Intermediate Results
- Storing Final Results
- Debugging Pig Latin

Loading Data

- Use the LOAD operator to load data from the file system.
- Default load function is “PigStorage”.
- Example :

```
LOAD  'data‘  [USING function]  [AS schema];
```

```
A = LOAD 'data' AS (name:chararray, age:int);
```

```
A = LOAD 'data' USING PigStorage(‘:’) AS (name, age);
```

Working with Data

- Transform data into what you want.
- Major operators:
  - FOREACH : work with columns of data
  - FILTER : work with tuples or rows of data
  - GROUP : group data in a single relation
  - JOIN : Performs an inner join of two or more relations based on common field values.
  - UNION : merge the contents of two or more relations
  - SPLIT : Partitions a relation into two or more relations

Storing Intermediate Results

- Default location is “/tmp”
- Could be configured by property “pig.temp.dir”

Storing Final Results

- Use the STORE operator to write output to the file system.
- Default store function is “PigStorage”.
- If the directory already exists, the STORE operation will fail. The output file will be named as “part-nnnnn”.
- Example :

```
STORE relation INTO 'directory' [USING function];
```

```
STORE  A  INTO  'myoutput‘  USING PigStorage ('*');
```

Debugging Pig Latin

- DUMP : display results to your terminal screen.**(useful)**
- DESCRIBE : review the schema of a relation. **(useful)**
- EXPLAIN : view the logical, physical, or map reduce execution plans to compute a relation.
- ILLUSTRATE : view the step-by-step execution of a series of statements.

A Complete Example

18	(John, Mary)
22	(Austin, Sunny)
78	(Richard, Alice)
87	(Johnny,)

I want to find couple(s) both alive and age > 75 ?

```
a = Load ‘data' AS (age:int, couple:(husband:chararray, wife:chararray))

b = FILTER a BY (age > 75) AND ((couple.husband is not null) AND
(couple.husband != '') AND (couple.wife is not null) AND (couple.wife != ''));

c = FOREACH b GENERATE couple AS lucky_couple:(old_man:chararray,
old_women:chararray);

STORE c INTO 'population';
```

Describe a ; ⇨ a: {age: int, couple: (husband: chararray, wife:chararray)}

A Complete Example

18	(John, Mary)
22	(Austin, Sunny)
78	(Richard, Alice)
87	(Johnny,)

```
a = Load ‘data' AS (age:int, couple:(husband:chararray, wife:chararray))
b = FILTER a BY (age > 75) AND ((couple.husband is not null) AND (couple.husband != '')
AND (couple.wife is not null) AND (couple.wife != ''));
c = FOREACH b GENERATE couple AS lucky_couple:(old_man:chararray,
old_women:chararray);
STORE c INTO 'population';
```

Dump a ;

(18,(John, Mary))
(22,(Austin, Sunny))
(78,(Richard, Alice))
(87,(Johnny,))

Dump b ; (78,(Richard, Alice))

Describe c ; c: {lucky_couple: (old_man: chararray,old_women: chararray)}

Folder ‘population’ (Richard, Alice)

OPERATOR

Arithmetic

Operator	Symbol
addition	+
subtraction	-
multiplication	*
division	/
modulo	%
bincond	?:

```
X = FOREACH A GENERATE f2, f1 + f2;
```

```
X = FOREACH A GENERATE f2, (f2==1?1:COUNT(B));
```

Boolean

Operator	Symbol
AND	AND
OR	OR
NOT	NOT

```
X = FILTER A BY (f1==8) OR (NOT (f2+f3 > f1));
```

Cast

Operator	Symbol
(Simple data type you want to cast to)	**(enclose data type in parentheses)**
to tuple	(tuple(data_type)) src_field
to bag	(bag{tuple(data_type)}) src_field
to map	(map[]) src_field

```
B = FOREACH A GENERATE (int)$0 + 1;
```

```
X = FOREACH B GENERATE group, (chararray)COUNT(A) AS total;
```

Cast

Operator	Symbol
(Simple data type you want to cast to)	(enclose data type in parentheses)
to tuple	(tuple(data_type)) src_field
to bag	**(bag{tuple(data_type)}) src_field**
to map	(map[]) src_field

(1,2,3)
(4,2,1)

```
A = LOAD 'data' AS fld;
B = FOREACH A GENERATE (tuple(int,int,float))fld;
```

{(1234L)}
{(5678L)}
{(9012)}

```
A = LOAD 'data' AS fld:bytearray;
B = FOREACH A GENERATE (bag{tuple(long)})fld;
```

[open#apache]
[apache#hadoop]
[hadoop#pig]

```
A = LOAD 'data' AS fld;
B = FOREACH A GENERATE ((map[])fld;
```

DESCRIBE A;

DESCRIBE B;

DUMP B;

Comparison

Operator	Symbol
equal	==
not equal	!=
greater than	>
greater than or equal to	>=
less than	<
less than or equal to	<=
pattern matching	matches

```
X = FILTER A BY (f2 == 'apache');
```

```
X = FILTER A BY (f1 matches '.*apache.*');
```

Type Construction

Operator	Symbol	Note
tuple constructor	()	Equivalent to **TOTUPLE**
bag constructor	{}	Equivalent to **TOBAG**
map constructor	[]	Equivalent to **TOMAP**

Type Construction

Operator	Symbol	Note
tuple constructor	()	Equivalent to **TOTUPLE**
bag constructor	{}	Equivalent to **TOBAG**
map constructor	[]	Equivalent to **TOMAP**

```
Joe   20
Amy  22
Leo   18
```

```
a = LOAD 'students' as (name:chararray, age:int);
b = FOREACH a GENERATE (name, age);
STORE b INTO ‘results’;
```

```
(Joe,20)
(Amy,22)
(Leo,18)
```

```
Joe  20   3.5
Amy  22   3.2
Leo  18   2.1
```

```
a = LOAD ‘data' AS (name:chararray, age:int, gpa:float);
b = FOREACH a GENERATE {(name, age)},
{name, age};
STORE b INTO ‘results’;
```

```
{(Joe ,20)}
{(Joe ),(20)}
{(Amy,22)}
{(Amy),(22)}
{(Leo,18)}
{(Leo),(18)}
```

```
Joe  20   3.5
Amy  22   3.2
Leo  18   2.1
```

```
a = LOAD 'students' AS (name:chararray, age:int, gpa:float);
b = FOREACH a GENERATE [name, gpa];
STORE b INTO ‘results’;
```

```
[Joe#3.5]
[Amy#3.2]
[Leo#2.1]
```

Null

Operator	Symbol
is null	is null
is not null	is not null

```
X = FILTER A BY f1 is not null;
```

LOAD

- Load data from the file system
- Default load function is PigStorage, could choose to loaded by other customized UDF.

LOAD 'data' [USING function] [AS schema];

```
A = LOAD 'myfile.txt' USING PigStorage('\t');

A = LOAD 'myfile.txt' AS (f1:int, f2:int, f3:int);
```

STORE

- Save result to the file system.
- Default store function is PigStorage, could be customized by UDF.
- About the output directory , if the directory already exist , the operation will fail.

STORE ‘alias’ INTO ‘directory’ [USING function]

```
STORE A INTO 'myOutput' USING PigStorage ('*');
```

GROUP

- Groups the data in one or more relations.
- The result of a GROUP operation is a relation that includes one tuple per group. The tuple contains 2 fields:
  - first field : named group, the same type as the group key
  - second field : whose name is the same with original relation, and type is a bag.

alias = GROUP alias {ALL | BY expression} [,alias ALL | BY expression …]

John 18 4.0F Mary 19 3.8F Bill 20 3.9F Joe 18 3.8F

```
a = LOAD ‘data' AS (name:chararray, age:int, gpa:float);
b = GROUP a BY age;
c = FOREACH b GENERATE group, COUNT(a);
```

DESCRIBE b;

DUMP b;

DUMP c;

COGROUP

- Basically, the GROUP and COGROUP operators are identical.
  - For readability GROUP is used in statements involving one relation.
  - COGROUP is used in statements involving two or more relations.

alias = COGROUP alias {ALL | BY expression} [,alias ALL | BY expression …]

```
Alice   turtle Alice
goldfish Alice   cat
Bob   dog
Bob   cat
```

```
Cindy   Alice Mark
Alice Paul   Bob
Paul   Jane
```

```
a = LOAD 'data1' AS (owner:chararray,pet:chararray);
b = LOAD 'data2' AS (friend1:chararray,friend2:chararray);
x = COGROUP a BY owner, b BY friend2;
```

DESCRIBE x;

DUMP x;

FILTER

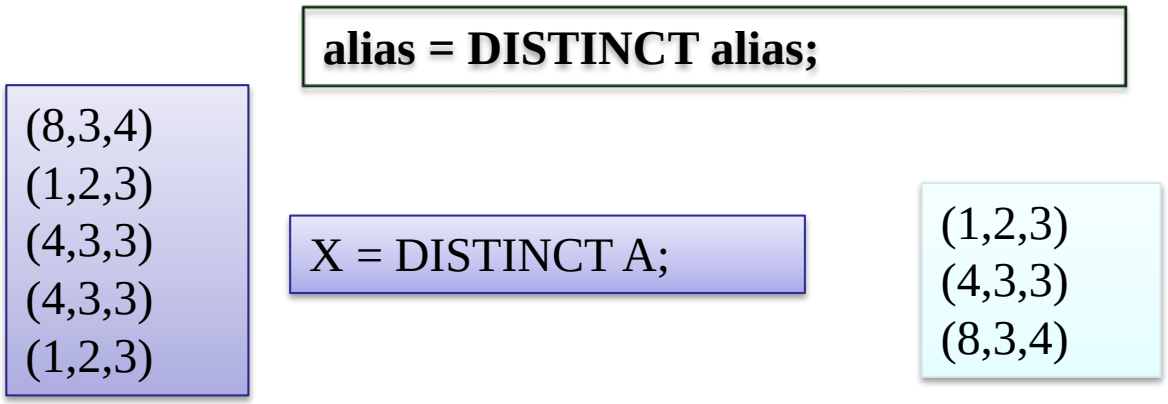
- Selects tuples from a relation based on some condition.

alias = FILTER alias BY expression;

```
X = FILTER  A  BY (f1 == 8)  OR  (NOT (f2+f3 > f1));
```

DISTINCT

- Removes duplicate tuples in a relation.
- You can **not** use DISTINCT on a subset of fields; to do this, use FOREACH and a nested block to first select the fields and then apply DISTINCT.



ORDER BY

- Sorts a relation based on one or more fields.
- Currently, supporting ordering on fields with simple types or by tuple designator (*). You **cannot** order on fields with complex types or by expressions.

alias = ORDER alias BY {*[ASC|DESC]} | field[ASC|DESC] [,field[ASC|DESC] …]

```
x = ORDER a BY a3 DESC;
```

JOIN (Inner Join)

- Performs an inner join of two or more relations based on common field values.
- Inner joins ignore null keys, so it makes sense to filter them out before the join.

alias = JOIN alias BY field (, alias BY field);

1	2	3
4	2	1
8	3	4
4	3	3
7	2	5
8	4	3

2	4
8	9
1	3
2	7
2	9
4	6
4	9

```
a = LOAD 'data1' AS (a1:int,a2:int,a3:int);
b = LOAD 'data2' AS (b1:int,b2:int);
x = JOIN a BY a1, b BY b1;
```

DESCRIBE x;

DUMP x;

FLATTEN

- Un-nests tuple/bag - changes the structure of it.

FLATTEN (Tuple/Bag)

```
18   (John, Mary)
22   (Austin, Sunny)
78   (Richard, Alice)
87   (Johnny,)
```

```
a = Load ‘data' AS (age:int, couple:(husband:chararray, wife:chararray));
b = FOREACH a GENERATE age, couple;
c = FOREACH a GENERATE age,
FLATTEN(couple);
```

DESCRIBE b;

DUMP b;

DESCRIBE c;

DUMP c;

```
18   (John, Mary)
22    (Austin, Sunny)
22    (Tom, Rosa)
78   (Richard, Alice)
87   (Johnny,)
```

```
a = Load ‘data' AS (age:int, couple:(husband:chararray,
wife:chararray));
b = GROUP a BY age;
c = FOREACH b GENERATE FLATTEN(a);
```

UNION

- Computes the union of two or more relations.
- Does **not** eliminate duplicate tuples.
- Does not ensure (as databases do) that all tuples adhere to the same schema – the simplest case should be “ensure that the tuples in the input relations have the same schema”

1	2	3
4	5	6
7	8	9

1	2	3
1	3	5
2	4	6

UNION alias, alias [,alias…];

```
a = LOAD 'data1' AS (a1:int,a2:int,a3:int);
b = LOAD 'data2' AS (b1:int,b2:int, b3:int);
x = UNION a, b;
```

DESCRIBE x;

DUMP x;

SPLIT

- Partitions a relation into two or more relations, Result :
  - A tuple may be assigned to more than one relation.
  - A tuple may not be assigned to any relation.

SPLIT alias INTO alias IF expression, alias IF expression [, alias IF expression …] [, alias OTHERWISE];

(1,2,3)
(4,5,6)
(7,8,9)

```
a = LOAD 'data1' AS (a1:int,a2:int,a3:int);

SPLIT a INTO x IF f1==7, y IF (f1 < 5 AND f2 < 6), z OTHERWISE;
```

DUMP x;

DUMP y;

DUMP z;

FOREACH

alias = FOREACH alias GENERATE expression [AS schema] [expression [AS schema]….];

- Generates data transformations based on columns of data.

1	2	3
4	5	6
7	8	9

```
a = LOAD 'data' AS (a1:int,a2:int,a3:int);

x = FOREACH a GENERATE a1+a2 AS f1:int, (a3==3?
TOTUPLE(1,1) : TOTUPLE (0,0));
```

DUMP x;

FOREACH

- Could work with nested.
  - FOREACH statements can be nested to two levels only.
  - Valid nested op : CROSS, DISTINCT, FILTER, FOREACH, LIMIT, and ORDER BY
  - The GENERATE keyword must be the last statement within the nested block.

```
MARY,22,female
RICHARD,26,male
AUSTIN,28,male
JOHN,52,male
TOM,40,male
JIJI,38,female
```

```
a = Load ‘data' USING PigStorage(',') AS (name:chararray, age:int, gender:chararray);
b = GROUP a BY gender;
c = FOREACH b
 {
     c0 = FILTER a BY (age > 20) AND (age < 30);
     GENERATE group, COUNT(c0) AS young;
 }
```

DESCRIBE b;

DUMP b;

DESCRIBE c;

DUMP c;

FOREACH

MARY,female,LA
RICHARD,male,Taipei
AUSTIN,male,Taipei
TOM,male,Taichiung
JOHN,male
JIJI,female

DESCRIBE c;

DUMP c;

```
a = LOAD ‘data' USING PigStorage(',') AS (name:chararray,
gender:chararray, location:chararray);
b = GROUP a BY gender;
c = FOREACH b
{
   c0 = FILTER a BY (location is not null) AND (location != '');
   c1 = ORDER c0 BY name;
   GENERATE group, c1;
}
```

Exercise

- Goal : Familiar with Pig Latin basic usage, includes:
  - Be able to run a Pig script.
  - Be able to load/store data using Pig.
  - Be able to parse data using Pig.
  - Be able to applying simple operation on data.

Built-in Functions

- **Don't** need to be registered, and qualified when they are used

Eval	Math	String
AVG	ABS	INDEXOF
CONCAT	ACOS	LAST_INDEX_OF
COUNT	ASIN	LCFIRST
COUNT_STAR	CBRT	UCFIRST
DIFF	CEIL	LOWER
ISEMPTY	COS	UPPER
MAX	COSH	REPLACE
MIN	EXP	SUBSTRING
SIZE	FLOOR	TRIM
SUM	LOG	REGEX_EXTRACT
TOKENIZE	LOG10	REGEX_EXTRACT_ALL

Built-in Functions

Name	Syntax	Description
TOTUPLE	TOTUPLE(expression [, expression ...])	Converts one or more expressions to type tuple.
TOMAP	TOMAP(key-expression, value-expression [, key-expression, value-expression ...])	Converts key/value expression pairs into a map
TOBAG	TOBAG(expression [, expression ...])	Converts one or more expressions to type bag
TOP	TOP(topN,column,relation)	Returns the top-n tuples from a bag of tuples.

Built-in Function : COUNT

- Computes the number of elements in a bag.
  - Requiring a preceding GROUP ALL statement for global counts or a GROUP BY statement for group counts.
  - It will ignore nulls. If you want to include NULL values in the count computation, use **COUNT_STAR**

1	2	3
8	3	4
7	2	5
8	4	3
	1	1
	1	1

```
a = LOAD 'data' AS (f1:int, f2:int, f3:int);
b = GROUP a BY f1;
x1 = FOREACH b GENERATE COUNT(a);
x2 = FOREACH b GENERATE COUNT_STAR(a);
```

DUMP x1;

DUMP x2;

Built-in Function : SUM

- Computes the sum of the numeric values in a single-column bag.
  - Requiring a preceding GROUP ALL statement for global sums and a GROUP BY statement for group sums.

```
Alice,turtle,1
Alice,goldfish,5
Alice,cat,2
Bob,dog,2
Bob,cat,2
```

```
a = LOAD 'data' USING  PigStorage(‘,’) AS
(owner:chararray,pet_type:chararray,pet_count:int);
b = GROUP a BY owner;
x = FOREACH b GENERATE group,
SUM(a.pet_num);
```

DUMP x;

Built-in Function: Load/Store

- PigStorage
- TextLoader
- JsonLoader/JsonStorage
- (Others)

What is UDF?

- "User Defined Function"
  - could be implemented by Java/Python/Javascript/Ruby. (The most extensive support is provided for Java)
- Types
  - ✓ Eval Function
  - ✓ Load/Store Function

Pig Type	Java Class
bytearray	DataByteArray
chararray	String
int	Integer
long	Long
float	Float
double	Double
tuple	Tuple
bag	DataBag
map	Map<Object, Object>

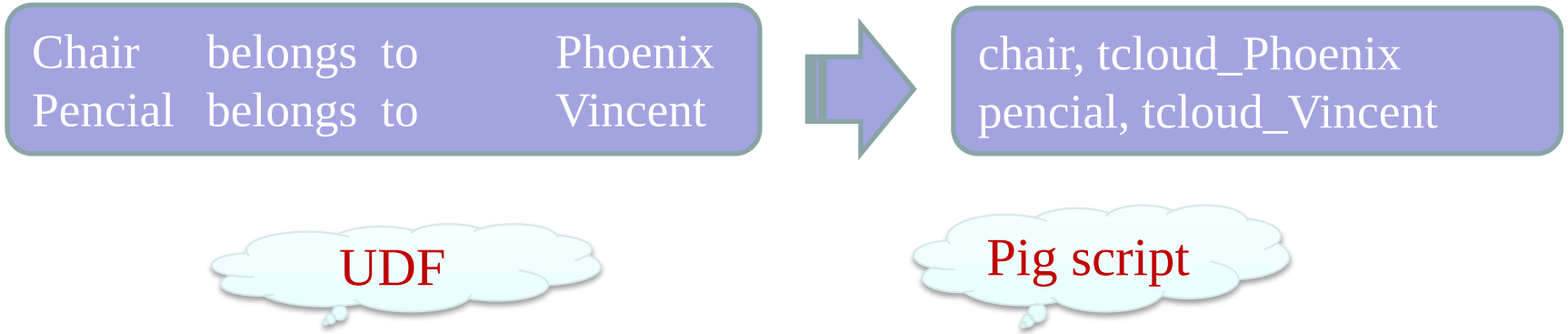
Usage

- Compile **pig.jar** first
- Register UDF jar in your pig script
- Using the UDF with full name (package + class name)
- Example

example_toTcloudStr_pig.txt

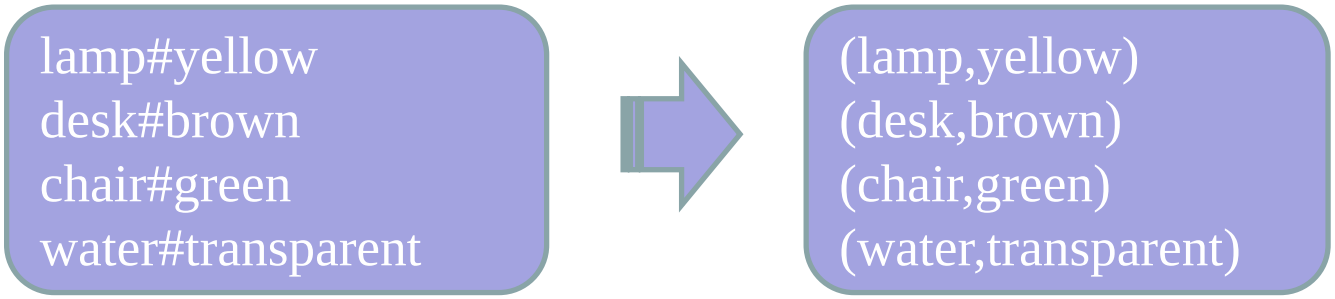
All UDF origins from…

- **EvalFunc<T>**
  - public abstract T **exec**(Tuple input) throws IOException
  - public Schema **outputSchema**(Schema input)
  - public List<FuncSpec> **getArgToFuncMapping**() throws FrontendException



UDF – Eval function (2)

- Extends **EvalFunc<T>**
- Example:

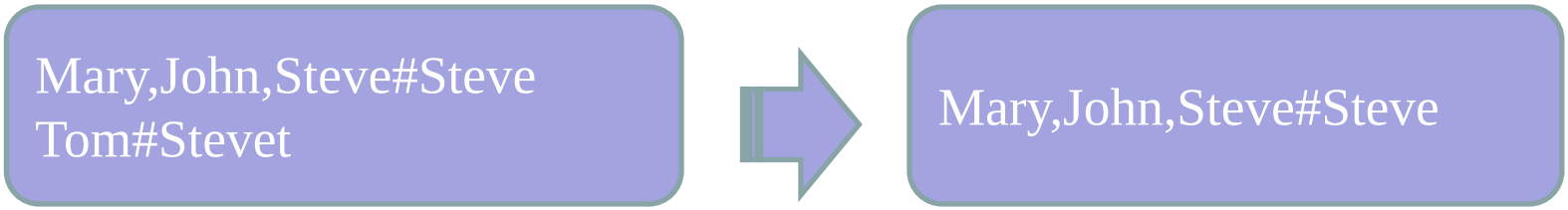


UDF

Pig script

UDF – Filter function

- Extends **FilterFunc**
- Example :

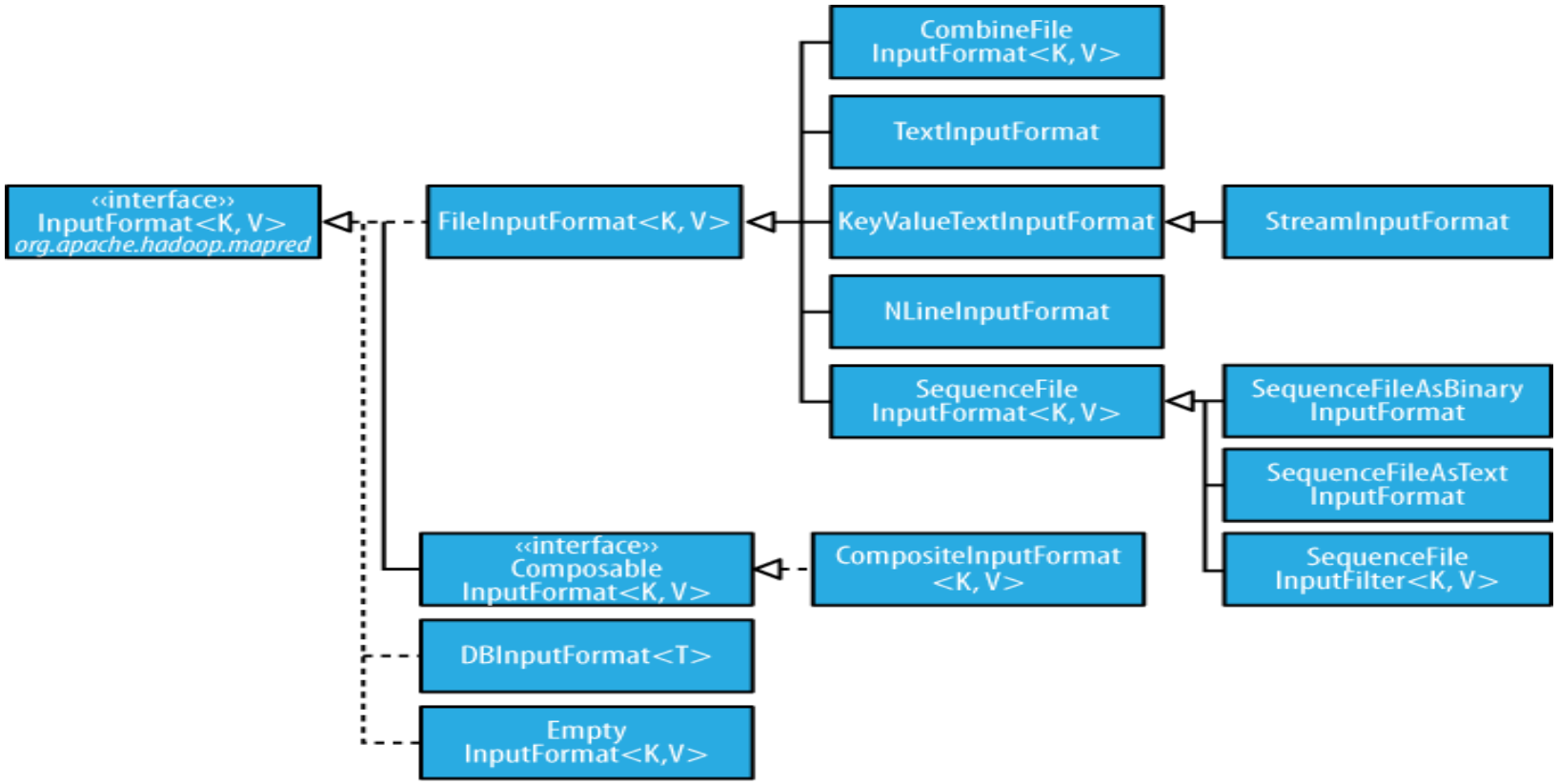


UDF

Pig script

Load/Store Function

- Basic class is LoadFunc/StoreFunc
- Aligned with Hadoop's InputFormat and OutputFormat



UDF – LoadFunc

- Extends LoadFunc
  - getInputFormat
  - prepareToRead
  - setLocation
  - getNext
- Example

More about UDF…

- Schema
- Error handling
  - WrappedIOException (deprecated)
- Function overloading

function_overloading.txt

- Reporting progress
  - Protected data variabe in Class EvalFunc : reporter.progress();

BEST PRACTICE

Pig Advance

STORE vs. DUMP

```
a = LOAD 'input' AS (x, y, z);
b = FILTER a BY x > 5;
DUMP b;
c = FOREACH b GENERATE y, z;
STORE c INTO 'output';
```

```
a = LOAD 'input' AS (x, y, z); b =
FILTER a BY x > 5; STORE b INTO
'output1';
c = FOREACH b GENERATE y, z;
STORE c INTO 'output2';
```

When using **DUMP** : The first job will execute A > B > DUMP while the second job will execute A > B > C > STORE.

Implicit Dependencies

- Description : If a script has dependencies on the execution order outside of what Pig knows about, execution may fail.
- How : To ensure that the right execution order is enforced, add the exec statement.

```
...
STORE a INTO 'out1';
b = LOAD 'data2';
c = FOREACH b GENERATE MYUDF($0,'out1');
STORE c INTO 'out2';
```

```
...
STORE a INTO 'out1';
EXEC;
b = LOAD 'data2';
c = FOREACH b GENERATE MYUDF($0,'out1');
STORE c INTO 'out2';
```

Use Optimization

- Pig supports various optimization rules, default setting is turn on.

```
B = FILTER A BY NOT (NOT(a0 > 5) OR a > 10);
```

```
B = FILTER A BY a0 > 5 AND a <= 10;
```

Use Types

- If types are not specified in the load statement, Pig assumes the type of double for numeric computations.
- Benefit of using types
  - Help with speed of arithmetic computation
  - Early error detection

```
A = load 'myfile' as (t, u, v);
B = foreach A generate t + u;
```

```
A = load 'myfile' as (t: int, u: int, v);
B = foreach A generate t + u;
```

Project Early and Often

- Pig does not (yet) determine when a field is no longer needed and drop the field from the row.
- Goal : reduce the amount of data being carried through the map and reduce phases by pig.

```
A = load 'myfile' as (t, u, v);
B = load 'myotherfile' as (x, y, z);
C = join A by t, B by x;
D = group C by u;
E = foreach D generate group, COUNT($1);
```

```
A = load 'myfile' as (t, u, v);
A1 = foreach A generate t, u;
B = load 'myotherfile' as (x, y, z);
B1 = foreach B generate x;
C = join A1 by t, B1 by x;
C1 = foreach C generate t, u;
D = group C1 by u;
E = foreach D generate group, COUNT($1);
```

Filter Early and Often

- Goal : reduce the amount of data being carried through the map and reduce phases by pig.

```
A = load 'myfile' as (t, u, v);
B = load 'myotherfile' as (x, y, z);
C = join A by t, B by x;
D = group C by u;
E = foreach D generate group, COUNT($1);
F = filter E by C.t == 1;
```

```
A = load 'myfile' as (t, u, v);
B = load 'myotherfile' as (x, y, z); C = filter A by t == 1;
D = join C by t, B by x;
E = group D by u;
F = foreach E generate group, COUNT($1);
```

Reduce Your Operator Pipeline

- Balance between readability and performance.

```
A = load 'data' as (in: map[]);
-- get key out of the map
B = foreach A generate in#'k1' as k1, in#'k2' as k2;
 -- concatenate the keys
C = foreach B generate CONCAT(k1, k2);
```

```
A = load 'data' as (in: map[]);
-- concatenate the keys from the map
B = foreach A generate CONCAT(in#'k1', in#'k2');
```

Drop Nulls Before a Join

- In one test where the key was null 7% of the time and the data was spread across 200 reducers, we saw a about a 10x speed up in the query by adding the early filters.

```
A = load 'myfile' as (t, u, v);
B = load 'myotherfile' as (x, y, z);
A1 = filter A by t is not null;
B1 = filter B by x is not null;
C = join A1 by t, B1 by x;
```

Use the LIMIT Operator

- Often you are not interested in the entire output but rather a sample or top results. In such cases, using LIMIT to gain much better performance

```
A = load 'myfile' as (t, u, v);
B = limit A 500;
```

```
-- Get top results:
A = load 'myfile' as (t, u, v);
B = order A by t;
C = limit B 500;
```

Prefer DISTINCT over GROUP BY/GENERATE

- To extract unique values from a column in a relation you can use DISTINCT or GROUP BY/GENERATE. DISTINCT is the preferred method; it is faster and more efficient.

```
A = LOAD 'myfile' AS (t, u, v);
B = FOREACH A GENERATE u;
C = GROUP B BY u;
D = FOREACH C GENERATE
group AS uniquekey;
DUMP D;
```

```
A = LOAD 'myfile' AS (t, u, v);
B = FOREACH A GENERATE u;
C = DISTINCT B;
DUMP C;
```

Big Data 時代

- 網路瀏覽
- 電信通訊
- 金融交易
- 工廠機台狀態日誌

Pig Latin + UDF = Easily To Analyze (Big) Data !

Big Data Analyze–以電信業為例

- Data
  - 姓名、性別、電話、住址、年齡、費率設定、通話時間
- Business (商業價值) -日常營運
  - Q1 : 客戶總人數?
  - Q2 : 客戶男女人數各多少?
  - Q3 : 各費率客戶人數各是多少?
  - Q4 : 列出花蓮(Hualien)地區每位客戶的通話記錄
  - Q5 : 統計每位客戶的總通話時數
  - Q6 : 列出每位客戶於深夜時段(23:00~02:00)的通話記錄

  - 註 : Q4~Q6答案, 請按客戶姓名排序

Big Data Analyze–以電信業為例

- Business (商業價值) –市場分析
  - Q1 :手機持有率最高的前三名縣市?
  - Q2 :用戶選擇高費率(798/999/1288/1388)的比例? (高費率人數/總人數)
  - Q3 :若某用戶有超過3通通話時間大於60分鐘的通聯記錄，且使用低費率(<798)者,請列出他的費率以及相關的長時間(>60分鐘)通話紀錄
  - Q4 :列出各區域總通話時數
  - Q5 :選出各區域總通話時數最高的前3名客戶予以特殊獎勵