

# CS595—BIG DATA TECHNOLOGIES

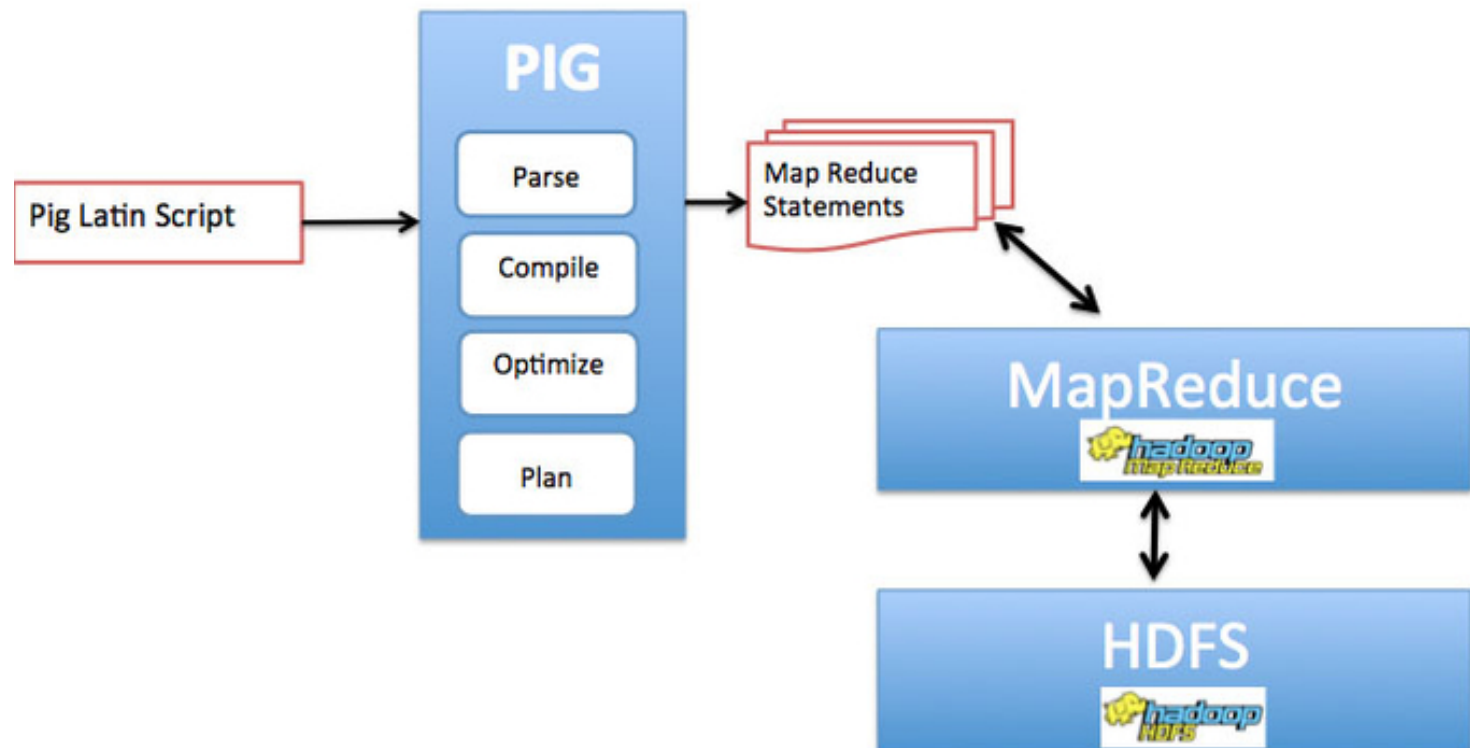
---

Module 05

Pig

# What is Pig?

- A the combination of a simple SQL-like scripting language called Pig Latin...
- Plus an engine that parses, optimizes, and automatically executes Pig Latin scripts as a series of MapReduce jobs on a Hadoop cluster



# What is Pig?

- Here is the word count program as a Pig script

```
-- Load input from the file named words.txt, and call the  
-- single field in the record 'line' or type chararray (string)
```

```
input = load 'words.txt' as (line: chararray);
```

```
-- TOKENIZE splits the line into a field for each word  
-- with the field (word) separators as spaces  
-- FLATTEN will take the collection of records returned by  
-- TOKENIZE and produce a separate record for each one,  
-- calling the single field in the record word.
```

```
words = foreach input generate FLATTEN(TOKENIZE(line, ' ')) as word;
```

```
-- Now group them together by each word.
```

```
grpds = group words by word;
```

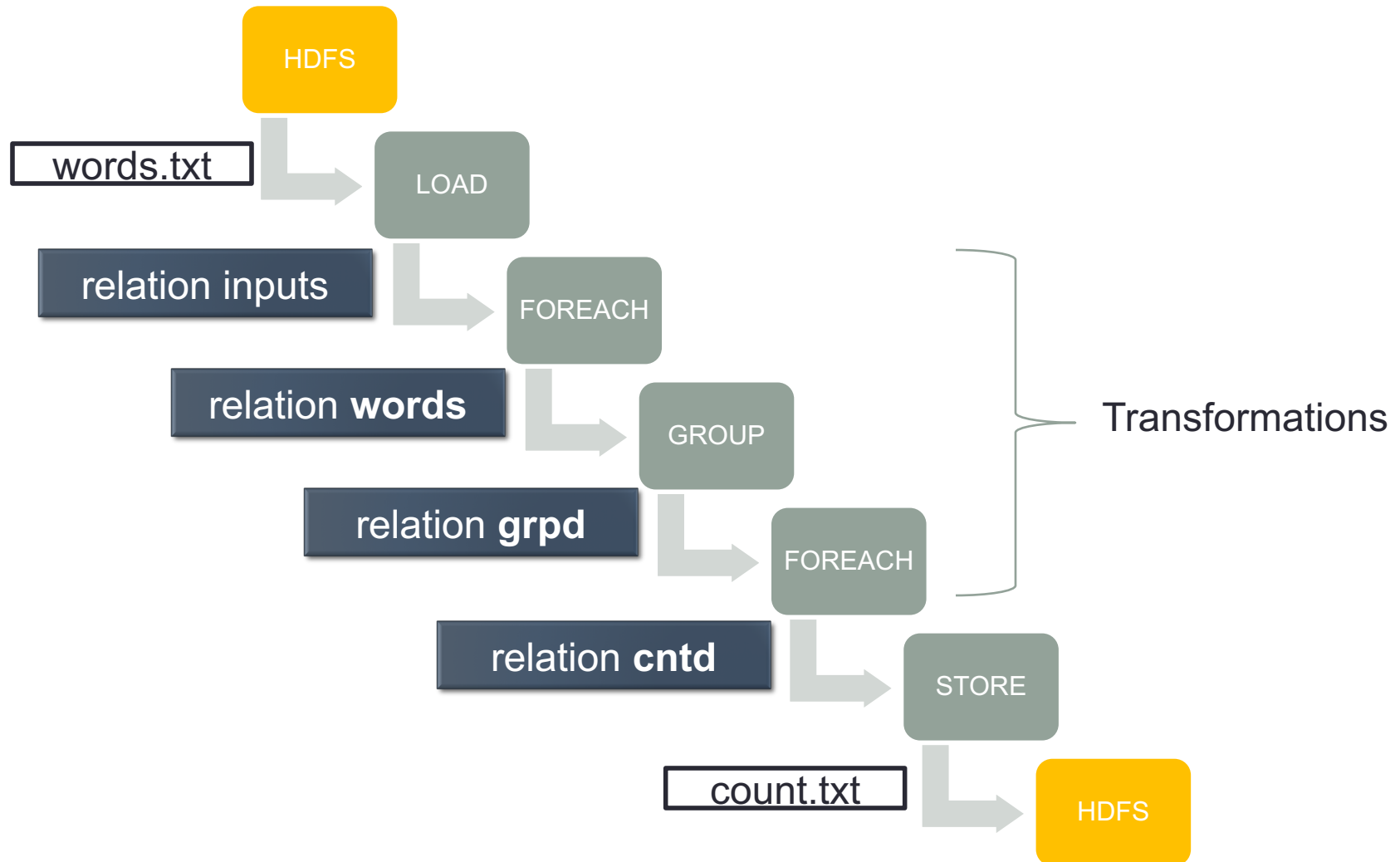
```
-- Count them.
```

```
cntds = foreach grpds generate group, COUNT(words);
```

```
-- Write out the results.
```

```
store cntds to 'count.txt'
```

# What is Pig?



# What is Pig?

- Pig Latin looks somewhat different from many of the programming languages you may have seen
- Pig Latin does not support control logic, that is, it does not have if, while, for or case statements
- This is because traditional procedural and object-oriented programming languages describe control flow, while data flow is a side effect of the program

# What is Pig?

- Enables data workers to write complex data processing logic (transformations) without knowing Java
- Appeals to developers already familiar with other scripting languages and with SQL
- It's much easier and faster to write than MapReduce, and it's easier to decipher, too
- Because the system automatically optimizes execution of MapReduce jobs, the user can focus on data semantics

# What is Pig Latin?

- Pig Latin is a data flow language
- This means it allows users to describe how data from one or more inputs should be read, transformed, and after that stored to one or more outputs in parallel
- These data flows can be simple linear flows, or complex workflows
  - That include points where multiple inputs are joined and where data is split into multiple streams to be processed by different operators
- To be mathematically precise, a Pig Latin script describes a directed acyclic graph (DAG)
  - Where the edges are data flows and the nodes are operators that process the data.

# What is Pig Latin?

- Pig Latin statements are the basic constructs you use to process data using Pig
- A Pig Latin statement is an operator that takes a relation as input and produces another relation as output...
- Except LOAD and STORE which read data from and write data to the file system
- Pig Latin scripts are generally organized as follows:
  - A LOAD statement to read data from the file system
  - A series of "transformation" statements to process the data
  - A STORE (or DUMP) statement to save (or display) results



# What is Pig Latin?

- After a cursory look, people often say that Pig Latin is a procedural version of SQL
- Although there are certainly similarities, there are more differences
  - SQL is a query language
  - Its focus is to allow users to form queries. It lets users describe what question they want answered, but not how they want it answered
  - In Pig Latin, on the other hand, the user describes exactly how to process the input data.
- Another major difference is that SQL is oriented around answering one question
  - When users want to do several data operations together, they must either write separate queries, storing the intermediate data into temporary tables...
  - Or use subqueries inside the query to do the earlier steps of the processing
  - However, many SQL users find subqueries confusing and difficult to form properly
  - Also, using subqueries creates an inside-out design where the first step in the data pipeline is the innermost query.
- Pig, however, is designed with a long series of data operations in mind
- There is no need to write the data pipeline in an inverted set of subqueries or to worry about storing data in temporary tables

# What is Pig Latin?

- Consider a case where a user wants to group one table on a key and then join it with a second table
- Because joins happen before grouping in a SQL query, this must be expressed either as a subquery or as two queries with the results stored in a temporary table

```
CREATE TEMP TABLE t1 AS  
SELECT customer, sum(purchase) AS total_purchases  
FROM transactions  
GROUP BY customer;
```

```
SELECT customer, total_purchases, zipcode  
FROM t1, customer_profile  
WHERE t1.customer = customer_profile.customer;
```

# What is Pig Latin?

- In Pig Latin, on the other hand, this looks like

```
-- Load the transactions file, group it by customer,  
-- and sum their total purchases  
txns    = load 'transactions' as (customer, purchase);  
grouped = group txns by customer;  
total   = foreach grouped generate group, SUM(txns.purchase) as tp;  
  
-- Load the customer_profile file  
profile = load 'customer_profile' as (customer, zipcode);  
  
-- Join the grouped and summed transactions and customer_profile data  
answer  = join total by group, profile by customer;  
  
-- Write the results to the screen  
dump answer;
```

# What is Pig Latin?

## Example

Suppose we have a table

urls: (url, category, pagerank)

Simple SQL query that finds,

For each sufficiently large category, the average pagerank of high pagerank urls in that category

```
SELECT category, Avg(pagerank)
FROM urls WHERE pagerank > 0.2
GROUP BY category HAVING COUNT(*) > 106
```

# What is Pig Latin?

## Example: Equivalent Pig Latin Program

```
good_urls = FILTER urls BY pagerank > 0.2;
```

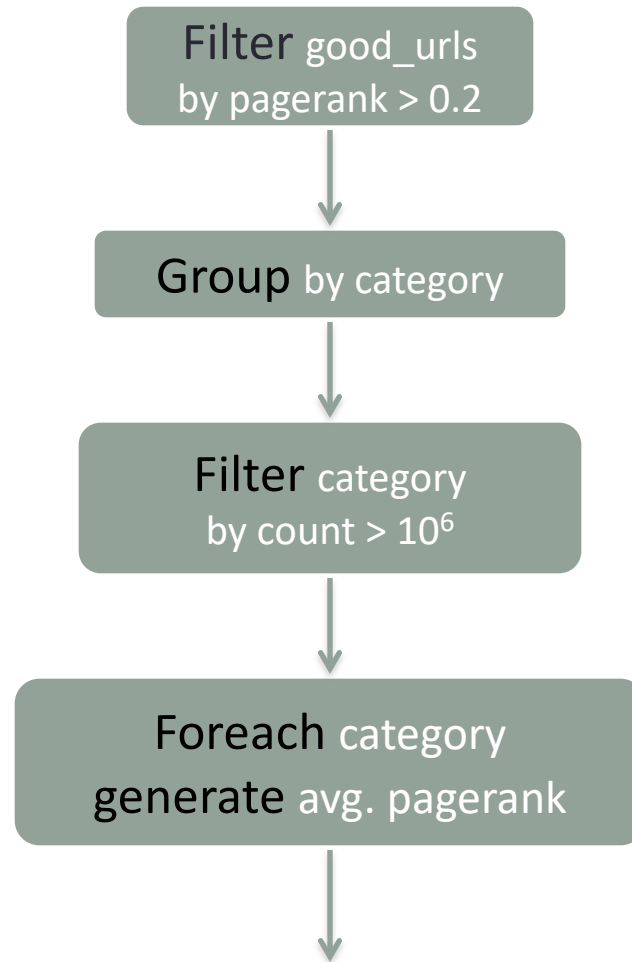
```
groups = GROUP good_urls BY category;
```

```
big_groups = FILTER groups BY  
                COUNT(good_urls) > 106 ;
```

```
output = FOREACH big_groups GENERATE  
                category, AVG(good_urls.pagerank);
```

# What is Pig Latin?

## Example: Pig Latin Program Data Flow

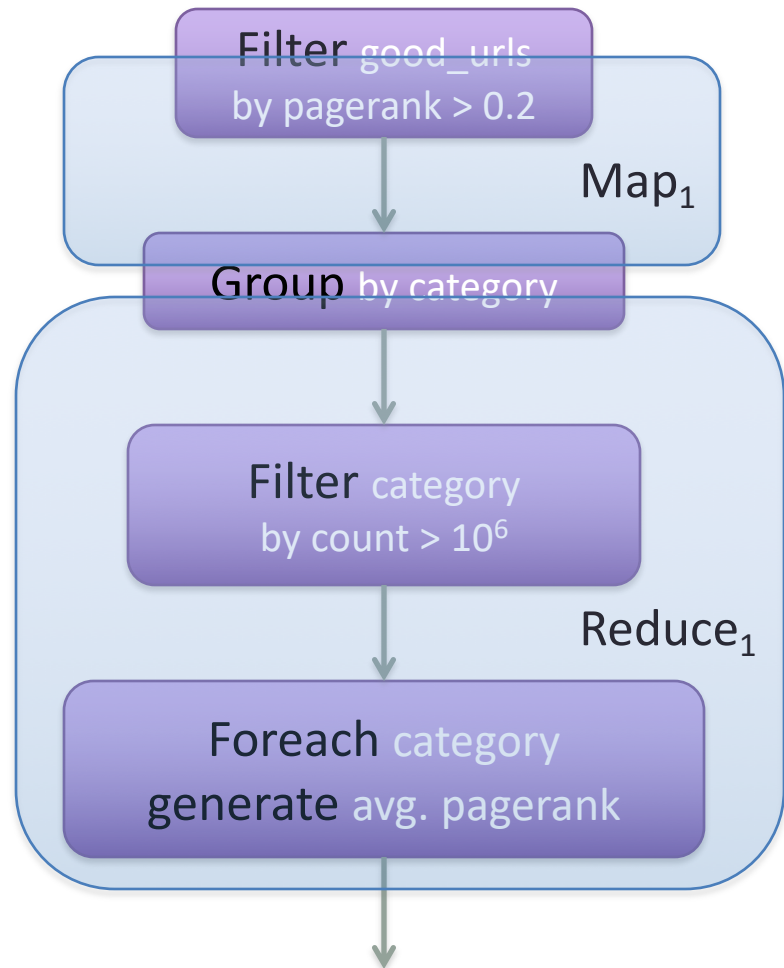


# What is Pig Latin?

## Example: Compilation into MapReduce

Every group or join operation forms a map-reduce boundary

Other operations pipelined into map and reduce phases



# Pig

## Sweet Spot Between SQL and MapReduce

- Pig offers users several advantages over using MapReduce directly
- Pig Latin provides all of the standard data-processing operations, such as join, filter, group by, order by, union, etc.
- MapReduce provides the group by operation directly (in the shuffle and reduce phases), and it provides the order by operation indirectly through the way it implements the grouping
- Filtering and projection can be implemented trivially in the map phase
- But other operators—particularly join—are not provided and must instead be written by the user.



# Pig

## Sweet Spot Between SQL and MapReduce

- Pig furnishes some complex, nontrivial implementations of these standard data operations.
- For example, because the number of records per key in a dataset is rarely evenly distributed, the data sent to the reducers is often skewed
- That is, one reducer may get 10 or more times the data as other reducers. Pig has join and order by operators that will handle this case and (in some cases) rebalance the reducers.
- But these took the Pig team months to write, and rewriting them in MapReduce would be time consuming.

# Pig

## Sweet Spot Between SQL and MapReduce

- In MapReduce, the data processing inside the map and reduce phases is opaque to the system
- This means that MapReduce has no opportunity to optimize or check the user's code
- Pig, on the other hand, can analyze a Pig Latin script and understand the data flow that the user is describing
- That means it can do early error checking and optimizations (can these two grouping operations be combined?)

# Pig

## Sweet Spot Between SQL and MapReduce

- MapReduce does not have a type system
- This is intentional, and it gives users the flexibility to use their own data types and serialization frameworks
- But the downside is that this further limits the system's ability to check users' code for errors both before and during runtime
- All of these points mean that Pig Latin is much lower-cost to write and maintain than Java code for MapReduce

# Pig

## Sweet Spot Between SQL and MapReduce

	SQL	Pig	MapReduce
<i>Programming style</i>	Large blocks of declarative constraints	➔	“Plug together pipes”
<i>Built-in data manipulations</i>	Group-by, Sort, Join, Filter, Aggregate, Top-k, etc...	←	Group-by, Sort
<i>Execution model</i>	Trust the query optimizer	➔	Simple, transparent
<i>Opportunities for automatic optimization</i>	Many	←	Few (logic buried in map() and reduce())
<i>Data Schema</i>	Must be known at table creation	➔	Not required, may be defined at runtime

# Pig Script

- The first line of this program loads the file *users*
- And declares that this data has two fields: name and age
- It assigns the name of Users to the input

**Users = load 'users' as (name, age);**

- The second line applies a filter to Users
- It allows records with an age between 18 and 25 inclusive
- All other records are discarded
- Now the data has records of users in the age range we are interested in
- The results of this filter are named Fltrd

**Fltrd = filter Users by age >= 18 and age <= 25;**

# Pig Script

- The second load statement loads *pages* and names it Pages. It declares its schema to have two fields, user and url.

```
Pages = load 'pages' as (user, url);
```

- The next line joins together Fltrd and Pages using Fltrd.name and Pages.user
- After this join we have found all the URLs each user has visited

```
Jnd = join Fltrd by name, Pages by user;
```

# Pig Script

- The next line collects records together by URL
- So for each value of url, there will be one record with a group of all records that have that value in the url field

**Grpd = group Jnd by url;**

- The next line then counts how many records are collected together for each URL
- After this line we now know, for each URL, how many times it was visited by users aged 18–25

**Smmd = foreach Grpd generate group, COUNT(Jnd) as clicks;**

# Pig Script

- The next thing to do is to sort this from most visits to least
- The following line sorts on the count value from the previous line and places it in descending order. Thus, the largest value will be first

**Srtd = order Smmd by clicks desc;**

- Next, we need only the top five pages, so the next line limits the sorted results to only five records

**Top5 = limit Srtd 5;**

- The results of this are then stored back to HDFS in the file *top5sites*

**store Top5 into 'top5sites';**



# Running Pig Interactively

- You can run Pig in interactive mode using the Grunt shell
  - Invoke the shell using the pig command and then enter your Pig Latin statements interactively at the command line
- 
- `$ pig -x mapreduce`
  - ... - Connecting to ...
  - `grunt>`
- 
- or
- 
- `$ pig`
  - ... - Connecting to ...
  - `grunt>`

# Running Pig in Batch Mode

- You can run Pig in batch mode using Pig scripts and the "pig" command

```
pig id.pig
```

or

```
pig -x mapreduce id.pig
```

# Pig Scripts

## Best Practices

- Identify script files using the \*.pig extension
- You can include comments in Pig scripts:
  - For multi-line comments use `/* .... */`
  - For single-line comments use `--`

```
/* myscript.pig  
My script is simple.  
It includes three Pig Latin statements.  
*/
```

```
A = LOAD 'student' USING PigStorage()  
    AS (name:chararray, age:int, gpa:float); -- loading data  
B = FOREACH A GENERATE name; -- transforming data  
DUMP B; -- retrieving results
```

# Pig Data Types

- Before we take a look at the operators that Pig Latin provides, we first need to understand Pig's data model
- This includes Pig's data types, and how you can describe your data to Pig using schemas
- Pig's data types can be divided into two categories
  - Scalar types, which contain a single value
  - Complex types, which contain other types

# Pig Data Types

- The simple data types that pig supports are:
  - **int** : It is signed 32 bit integer
  - **long** : It is a 64 bit signed integer
  - **float** : It is a 32 bit floating point
  - **double** : It is a 63 bit floating ppoint
  - **chararray** : It is character array in unicode UTF-8 format. This corresponds to java's String object
  - **bytearray** : Used to represent bytes. It is the default data type.
    - If you don't specify a data type for a filed, then bytearray datatype is assigned for the field.
  - **boolean** : to represent true/false values

# Pig Data Types

- **Complex Types:** Pig supports three complex data types. They are listed below:
  - **Tuple** : An ordered set of fields. Tuple is represented by braces.
    - Example: (1,2)
  - **Bag** : A set of tuples is called a bag. Bag is represented by flower or curly braces.
    - Example: {(1,2),(3,4)}
  - **Map** : A set of key value pairs. Map is represented in a square brackets.
    - Example: [key#value] .
    - The # is used to separate key and value
- Pig allows nesting of complex data structures
  - Example: You can nest a tuple inside a Tuple, Bag or a Map

# Pig Data Types

- The formats for complex data types are shown here:
- Tuple
  - enclosed by (), items separated by ","
  - Non-empty tuple: (item1,item2,item3)
  - Empty tuple is valid: ()
- Bag
  - enclosed by {}, tuples separated by ","
  - Non-empty bag: {code}{(tuple1),(tuple2),(tuple3)}{code}
  - Empty bag is valid: {}
- Map
  - enclosed by [], items separated by ",", key and value separated by "#"
  - Non-empty map: [key1#value1,key2#value2]
  - Empty map is valid: []
- If load statement specifies a schema, Pig will convert the complex type according to schema
  - If conversion fails, the affected item will be null

# Schemas

- Schemas enable you to assign names to fields and declare types for fields
- The easiest way to communicate the schema of your data to Pig is to explicitly tell Pig what it is when you load the data

```
dividends = load 'NYSE_dividends' as  
(exchange:chararray, symbol:chararray, date:chararray, dividend:float);
```

- Pig now expects your data to have four fields
- If it has more, it will truncate the extra ones
- If it has less, it will pad the ends of the records with nulls



# Schemas

- Schema 1

A = LOAD 'input/A' as (name:chararray, age:int);

B = FILTER A BY age != 20;

- Schema 2

A = LOAD 'input/A' as (name:chararray, age:chararray);

B = FILTER A BY age != '20';

- No Schema

A = LOAD 'input/A' ;

B = FILTER A BY A.\$1 != '20';

# Schemas

- Schemas are optional but we encourage you to use them whenever possible
  - Type declarations result in better parse-time error checking and more efficient code execution
- Schemas for simple types and complex types can be used anywhere a schema definition is appropriate
- Schemas are defined with the LOAD and FOREACH operators using the AS clause
- If you define a schema using the LOAD operator, then it is the load function that enforces the schema

# Schemas

- You can define a schema that includes both the field name and field type  
`A = LOAD 'data' USING MyStorage() AS (name:chararray, age: int);`
- You can define a schema that includes the field name only; in this case, the field type defaults to bytearray
  - `A = LOAD 'data' USING MyStorage() AS (name, age);`
- You can choose not to define a schema; in this case, the field is unnamed and the field type defaults to bytearray
- If you assign a name to a field, you can refer to that field using the name or by positional notation
- If you don't assign a name to a field you can only refer to the field using positional notation
  - `G = GROUP A BY $0; -- field indexes are zero based`

# Schemas

- Once the schema of the source data is given, the schema of the intermediate relation will be induced by Pig

# Schemas

- Why schemas?
  - Scripts are more readable
  - Help system validate the input
- Similar to Database?
  - Yes. But schema here is optional
  - Schema is not fixed for a particular dataset, but changable

# Pig Scripts General Flow

- **Loading Data**

- Use the LOAD operator and the load/store functions to read data into Pig

- **Working with Data**

- Pig allows you to transform data in many ways. As a starting point, become familiar with these operators:
- Use the FILTER operator to work with tuples or rows of data
- Use the FOREACH operator to work with columns of data.
- Use the GROUP operator to group data in a single relation
- Use the inner JOIN and outer JOIN operators to group or join data in two or more relations.
- Use the UNION operator to merge the contents of two or more relations
- Use the SPLIT operator to partition the contents of a relation into multiple relations

# Pig Scripts General Flow

- **Storing Intermediate Results**

- Pig stores the intermediate data generated between MapReduce jobs in a temporary location on HDFS
- This location must already exist on HDFS prior to use
- This location can be configured using the `pig.temp.dir` property
- The property's default value is `"/tmp"`

- **Storing Final Results**

- Use the `STORE` operator and the `load/store` functions to write results to the file system
- During the testing/debugging phase of your implementation, you can use `DUMP` to display results to your terminal screen
- However, in a production environment you always want to use the `STORE` operator to save your results

# Input and Output

## LOAD

- Description
  - Loads data from the (HDFS) file system
- Syntax
  - `LOAD 'data' [USING function] [AS schema];`
- Usage
  - Use the LOAD operator to load data from the file system
  - If you specify a directory name, all the files in the directory are loaded
  - The load function assumes data of the type specified by the schema
  - If the data does not conform to the schema, depending on the loader, either a null value or an error is generated
  - PigStorage is the default load
    - For PigStorage the default field separator is the tab ('\t')
    - To use another separate write PigStorage('<separator>')



# Input and Output

## LOAD: Example #1

- Suppose we have a data file called myfile.txt. The fields are tab-delimited. The records are newline-separated

1 2 3

4 2 1

8 3 4

# Input and Output

## LOAD: Example #1

- The default load function, PigStorage, loads data from myfile.txt to form relation A
- Because no schema is specified, the fields are not named and all fields default to type bytearray

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

```
DUMP A;
```

```
(1,2,3)
```

```
(4,2,1)
```

```
(8,3,4)
```

# Input and Output

## LOAD: Example #2

- In this example a schema is specified using the AS keyword
- You use the DESCRIBE operator to view the schema

```
A = LOAD 'myfile.txt' USING PigStorage('\t') AS (f1:int, f2:int, f3:int);
```

```
DESCRIBE A;
```

```
a: {f1: int,f2: int,f3: int}
```

# Input and Output STORE

- Description
  - Saves results to the (HDFS) file system
- Syntax
  - STORE alias INTO 'directory' [USING function];
- Usage
  - Use the STORE operator to execute Pig Latin statements and save results to the file system
  - PigStorage is the default store function and does not need to be specified
  - To debug scripts during development, you can use DUMP to check intermediate results

# Input and Output

## STORE: Example

- In this example data is stored using PigStorage and the asterisk character (\*) as the field delimiter

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

```
DUMP A;
```

```
(1,2,3)
```

```
(4,2,1)
```

```
(8,3,4)
```

```
(4,3,3)
```

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

```
CAT myoutput; -- prints the content of a file to the screen
```

```
1*2*3
```

```
4*2*1
```

```
8*3*4
```

```
4*3*3
```

# Debugging

- Pig Latin provides operators that can help you debug your Pig Latin statements:
  - Use the DUMP operator to display results to your terminal screen
  - Use the DESCRIBE operator to review the schema of a relation
  - Use the EXPLAIN operator to view the logical, physical, or map reduce execution plans to compute a relation
  - Use the ILLUSTRATE operator to view the step-by-step execution of a series of statements

# Debugging DUMP

- Describe
  - Dumps or displays results to screen
- Syntax
  - DUMP alias;
- Usage
  - Use the DUMP operator to execute Pig Latin statements and display the results to your screen
  - DUMP is meant for interactive mode; statements are executed immediately and the results are not saved

# Debugging

## DUMP: Example #1

- In this example a dump is performed after each statement.

```
A = LOAD 'student' AS (name:chararray, age:int, gpa:float);
```

```
DUMP A;  
(John,18,4.0F)  
(Mary,19,3.7F)  
(Bill,20,3.9F)  
(Joe,22,3.8F)  
(Jill,20,4.0F)
```

```
B = FILTER A BY name matches 'J.+';
```

```
DUMP B;  
(John,18,4.0F)  
(Joe,22,3.8F)  
(Jill,20,4.0F)
```



# Debugging

## DUMP: Example #2

- In this example we use the limit operator to restrict the number of records the dump operator displays

```
A = LOAD 'student' AS (name:chararray, age:int, gpa:float);
```

```
DUMP A;  
(John,18,4.0F)  
(Mary,19,3.7F)  
(Bill,20,3.9F)  
(Joe,22,3.8F)  
(Jill,20,4.0F)
```

```
B = LIMIT A 2;
```

```
DUMP B;  
(John,18,4.0F)  
(Joe,22,3.8F)
```

# Debugging

## DESCRIBE

- Description
  - Returns the schema of a relation
- Syntax
  - DESCRIBE alias;
- Usage
  - Use the DESCRIBE operator to view the schema of a relation
  - You can view outer relations as well as relations defined in a nested FOREACH statement

# Debugging DESCRIBE

- In this example a schema is specified using the AS clause. If all data conforms to the schema, Pig will use the assigned types

```
A = LOAD 'student' AS (name:chararray, age:int, gpa:float);  
B = FILTER A BY name matches 'J.+';  
C = GROUP B BY name;  
D = FOREACH C GENERATE COUNT(B.age);
```

```
DESCRIBE A;
```

```
A: {name: chararray,age: int,gpa: float}
```

```
DESCRIBE B;
```

```
B: {name: chararray,age: int,gpa: float}
```

```
DESCRIBE C;
```

```
C: {group: chararray,B: {(name: chararray,age: int,gpa: float)}}
```

```
DESCRIBE D;
```

```
D: {long}
```

# Debugging

## DESCRIBE

- This example shows how to view the schema of a nested relation using the :: operator.

```
A = LOAD 'studentab10k' AS (name, age, gpa);
```

```
B = GROUP A BY name;
```

```
C = FOREACH B {
```

```
    D = DISTINCT A.age;
```

```
    GENERATE COUNT(D), group;}
```

```
DESCRIBE C::D;
```

```
D: {age: bytearray}
```

# Debugging

## ILLUSTRATE

- Description
  - Displays a step-by-step execution of a sequence of statements
- Syntax
  - `ILLUSTRATE {alias | -script scriptfile};`
- Usage
  - Use the ILLUSTRATE operator to review how data is transformed through a sequence of Pig Latin statements.
  - Allows you to test your programs on small datasets and get faster turnaround times
  - Is based on an example generator that works by retrieving a small sample of the input data and then propagating this data through the pipeline
  - However, some operators, such as JOIN and FILTER, can eliminate tuples from the data - and this could result in no data following through the pipeline
  - To address this issue, the algorithm will automatically generate example data, in near real-time
  - So, you might see data propagating through the pipeline that was not found in the original input data
  - But this data changes nothing and ensures that you will be able to examine the semantics of your Pig Latin statements

# Debugging

## ILLUSTRATE: Example #1

- This example demonstrates how to use ILLUSTRATE with a relation
- Note that the LOAD statement must include a schema (the AS clause)

```
visits = LOAD 'visits.txt' USING PigStorage(',')  
        AS (user:chararray, url:chararray, timestamp:chararray);  
DUMP visits;
```

```
(Amy,yahoo.com,19990421)  
(Fred,harvard.edu,19991104)  
(Amy,cnn.com,20070218)  
(Frank,nba.com,20070305)  
(Fred,berkeley.edu,20071204)  
(Fred,stanford.edu,20071206)
```

```
recent_visits = FILTER visits BY timestamp >= '20071201';  
user_visits = GROUP recent_visits BY user;  
num_user_visits = FOREACH user_visits GENERATE group, COUNT(recent_visits);  
DUMP num_user_visits;
```

```
(Fred,2)
```

# Debugging

## ILLUSTRATE: Example #1

ILLUSTRATE num\_user\_visits;

visits	user: chararray	url: chararray	timestamp: chararray
	Fred	berkeley.edu	20071204
	Fred	stanford.edu	20071206
	Frank	nba.com	20070305

recent_visits	user: chararray	url: chararray	timestamp: chararray
	Fred	berkeley.edu	20071204
	Fred	stanford.edu	20071206

# Debugging

## ILLUSTRATE: Example #1

user_visits	group: chararray	recent_visits: bag({user: chararray,url: chararray,timestamp: chararray})
-----		
	Fred	{(Fred, berkeley.edu, 20071204), (Fred, stanford.edu, 20071206)}

num_user_visits	group: chararray	long
	Fred	2



# Relational Operations

## Overview

- Main tools Pig Latin provides to operate on your data
- Allows you to transform it by sorting, grouping, joining, projecting, and filtering

# Relational Operations

## FILTER

- **Description**  
Selects tuples from a relation based on some condition
- **Syntax**  
alias = FILTER alias BY expression;
- **Usage**  
Use the FILTER operator to select rows of data meeting a certain criteria

# Relational Operations

## FILTER: Example #1

- Suppose we have relation A

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

```
DUMP A;
```

```
(1,2,3)
```

```
(4,2,1)
```

```
(8,3,4)
```

```
(4,3,3)
```

```
(7,2,5)
```

```
(8,4,3).
```

# Relational Operations

## FILTER: Example #1

- In this example the condition states that if the third field equals 3, then include the tuple with relation X

```
X = FILTER A BY f3 == 3;
```

```
DUMP X;
```

```
(1,2,3)
```

```
(4,3,3)
```

```
(8,4,3)
```

# Relational Operations

## FILTER: Example #2

- In this example the condition states
  - If the first field equals 8
  - Or if the sum of fields f2 and f3 is not greater than first field
  - Then include the tuple in relation X

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

DUMP X;

(4,2,1)

(8,3,4)

(7,2,5)

(8,4,3)

# Relational Operations

## FILTER: More Details

- Allows you to select which records will be retained in your data pipeline
- A 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 you expect...
- Including `==` to test equality and `!=`, `>`, `>=`, `<`, and `<=`
- These comparators can be used on any scalar data type
- `==` and `!=` can be applied to maps and tuples
- To use these with two tuples, both tuples must have either the same schema or no schema
- None of the equality operators can be applied to bags

# Relational Operations

## FILTER: More Details

- You can test whether the value of a scalar field is within a set of values using the IN operator

```
divs = load 'NYSE_dividends' as (exchange:chararray, symbol:chararray,  
    date:chararray, dividends:float);  
cme_ctb_cht = filter divs by symbol in ('CME', 'CTB', 'CHT');
```

- For chararrays, users can test to see whether the chararray matches a regular expression

```
divs = load 'NYSE_dividends' as (exchange:chararray,  
    symbol:chararray, date:chararray, dividends:float);  
startswithcm = filter divs by symbol matches 'CM.*';
```

# Relational Operations

## FOREACH

- Description
  - Generates data transformations based on columns of data
- Syntax
  - FOREACH alias GENERATE expression [AS schema] [expression [AS schema]....];
- Usage
  - Use the FOREACH...GENERATE operation to work with columns of data



# Relational Operations

## FOREACH: Example #1

- In this example two fields from relation A are projected to form relation X

```
X = FOREACH A GENERATE a1, a2;
```

```
DUMP X;
```

```
(1,2)
```

```
(4,2)
```

```
(8,3)
```

```
(4,3)
```

```
(7,2)
```

```
(8,4)
```

# Relational Operations

## FOREACH: More Details

- Takes a set of expressions and applies them to every record in the data pipeline—hence the name foreach
- From these expressions it generates new records to send down the pipeline to the next operator
- For those familiar with database terminology, it is Pig's projection operator
- The following code loads an entire record, but then removes all but user and id fields from the record

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

```
B = foreach A generate user, id;
```

# Relational Operations

## FOREACH: More Details

- foreach supports a list of expressions
- The simplest are constants and field references
- The syntax for constants has already been discussed
- Field references can be by name (as shown in the preceding example) or by position
- Positional references are preceded by a \$ (dollar sign) and start from zero

```
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;
```

- The relations gain and gain2 will contain the same values
- Positional-style references are useful where the schema is unknown or undeclared

# Relational Operations

## FOREACH: More Details

- Standard arithmetic operators for integers and floating-point numbers are supported
  - $+$  for addition
  - $-$  for subtraction
  - $*$  for multiplication
  - $/$  for division
  - For integers the modulo operator  $\%$  is supported
- These operators return values of their own type, so  $5/2$  is 2, and  $5.0/2.0$  is 2.5
- The unary negative operator  $(-)$  is also supported for both integers and floating-point numbers
- Pig Latin obeys the standard mathematical precedence rules
- Null values are viral for all arithmetic operators
  - That is,  $x + \text{null} = \text{null}$  for all values of  $x$

# Relational Operations

## FOREACH: : More Details

- To extract data from complex types, use the projection operators
- For maps this is # (the pound sign or hash), followed by the name of the key as a string

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

- The value associated with a key may be of any type
- If you reference a key that does not exist in the map, the result is a null

# Relational Operations

## FOREACH: More Details

- Tuple projection is done with `.`, the dot operator.
- As with top-level records, the field can be referenced by name (if you have a schema for the tuple) or by position

```
A = load 'input' as (t:tuple(x:int, y:int));
```

```
B = foreach A generate t.x, t.$1;
```

- Referencing a field name that does not exist in the tuple will produce an error
- Referencing a nonexistent positional field in the tuple will return null

# Relational Operations

## FOREACH: More Details

- Bags do not guarantee that their tuples are stored in any order
- So allowing a projection of the tuple inside the bag would not be meaningful
- Instead, when you project fields in a bag, you are creating a new bag with only those fields

```
A = load 'input' as (b:bag{t:(x:int, y:int)});
```

```
B = foreach A generate b.x;
```

- This will produce a new bag whose tuples have only the field x in them

# Relational Operations

## FOREACH: More Details

- You can project multiple fields in a bag by surrounding the fields with parentheses and separating them by commas

```
A = load 'input' as (b:bag{t:(x:int, y:int)});
```

```
B = foreach A generate b.(x, y);
```



# Relational Operations

## FOREACH: More Details

- The distinction that `b.x` is a bag and not a scalar value has consequences
- Consider the following Pig Latin, which will not work

```
A = load 'foo' as (x:chararray, y:int, z:int);
```

```
B = group A by x; -- produces bag A containing all records for a given value of x
```

```
C = foreach B generate SUM(A.y + A.z);
```

- This will produce an error because `A.y` and `A.z` are bag and the *addition operator is not defined on bags*
- The correct way to do this calculation in Pig Latin is

```
A = load 'foo' as (x:chararray, y:int, z:int);
```

```
A1 = foreach A generate x, y + z as yz;
```

```
B = group A1 by x;
```

```
C = foreach B generate SUM(A1.yz)
```

# Relational Operations

## FOREACH: More Details

- User-defined functions (UDFs) can be invoked in foreach statements.
- These are called *evaluation functions*, or *eval funcs*
- Because they are part of a foreach statement, UDFs take one record at a time and produce one output
- Either the input or the output can be a bag, so this one record can contain a bag of records

```
divs = load 'NYSE_dividends' as (exchange, symbol, date, dividends);  
--make sure all strings are uppercase  
upped = foreach divs generate UPPER(symbol) as symbol, dividends;  
grpds = group upped by symbol; --output a bag upped for each value of  
symbol  
--take a bag of integers, and produce one result for each group  
sums = foreach grpds generate group, SUM(upped.dividends);
```

# Relational Operations

## FOREACH: More Details

- The result of each foreach statement is a new tuple, usually with a different schema than the tuple that was input to foreach
- Pig can almost always infer the data types of the fields in this schema from the foreach statement
- But it cannot always infer the names of those fields
- For fields that are simple projections with no other operators applied, Pig keeps the same names as before

```
divs = load 'NYSE_dividends' as (exchange:chararray, symbol:chararray,  
                                date:chararray, dividends:float);
```

```
sym = foreach divs generate symbol;  
describe sym;
```

```
sym: {symbol: chararray}
```

# Relational Operations

## FOREACH: More Details

- The result of each foreach statement is a new tuple, usually with a different schema than the tuple that was input to foreach
- Pig can almost always infer the data types of the fields in this schema from the foreach statement
- But it cannot always infer the names of those fields
- For fields that are simple projections with no other operators applied, Pig keeps the same names as before

```
divs = load 'NYSE_dividends' as (exchange:chararray, symbol:chararray,  
                                date:chararray, dividends:float);
```

```
sym = foreach divs generate symbol;  
describe sym;
```

```
sym: {symbol: chararray}
```

# Relational Operations

## FOREACH: More Details

- Once any expression beyond simple projection is applied, Pig does not assign a name to the field
- If you do not explicitly assign a name, the field will be nameless and will be addressable only via a positional parameter; for example, \$0
- You can assign a name with the *as* clause

```
divs    = load 'NYSE_dividends' as (exchange:chararray, symbol:chararray,  
                                   date:chararray, dividends:float);
```

```
in_cents = foreach divs generate dividends * 100.0 as dividend,  
                             dividends * 100.0;
```

```
describe in_cents;
```

```
in_cents: {dividend: double,double}
```

- The second field is unnamed since we didn't assign a name to it.

# Relational Operators

## GROUP

- Description
  - Groups the data in one or more relations
- Syntax
  - GROUP alias ALL;
  - GROUP alias BY expression;
- Usage
  - The GROUP operator groups together tuples that have the same group key (key field).
  - The key field will be a tuple if the group key has more than one field, otherwise it will be the same type as that of the group key.
  - The result of a GROUP operation is a relation that includes one tuple per group. This tuple contains two fields
  - The first field is named "group" (do not confuse this with the GROUP operator) and is the same type as the group key
  - The second field takes the name of the original relation and is type bag.
  - The GROUP and JOIN operators perform similar functions. GROUP creates a nested set of output tuples while JOIN creates a flat set of output tuples

# Relational Operations

## GROUP: Example #1

- Suppose we have relation A

```
A = load 'student' AS (name:chararray,age:int,gpa:float);
```

```
DESCRIBE A;
```

```
A: {name: chararray,age: int,gpa: float}
```

```
DUMP A;
```

```
(John,18,4.0F)
```

```
(Mary,19,3.8F)
```

```
(Bill,20,3.9F)
```

```
(Joe,18,3.8F)
```

# Relational Operations

## GROUP: Example #1

- Now, suppose we group relation A on field "age" for form relation B

```
B = GROUP A BY age;
```

```
DESCRIBE B;
```

```
B: {group: int, A: {name: chararray, age: int, gpa: float}}
```

```
DUMP B;
```

```
(18, {(John, 18, 4.0F), (Joe, 18, 3.8F)})
```

```
(19, {(Mary, 19, 3.8F)})
```

```
(20, {(Bill, 20, 3.9F)})
```



# Relational Operations

## GROUP: More Details

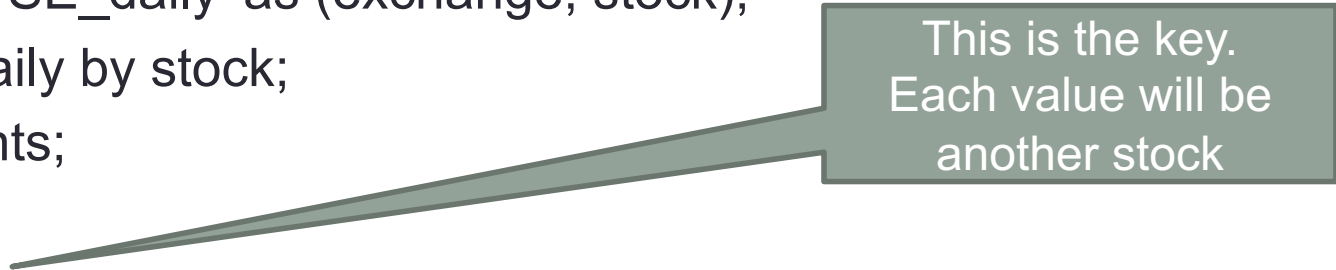
- Collects together records with the same key
- It shares syntax with SQL but the grouping operator in Pig Latin is different
- In SQL the group by clause creates a group that must feed directly into one or more aggregate functions
- In Pig Latin there is no direct connection between group and aggregate functions
- Instead, group does exactly what it says: collects all records with the same value for the provided key together into a bag
- You can then pass this to an aggregate function if you want, or do other things with it

# Relational Operations

## GROUP: More Details

- For example,

```
daily = load 'NYSE_daily' as (exchange, stock);  
grp = group daily by stock;  
describe in_cents;
```



This is the key.  
Each value will be  
another stock

```
grp: {group: bytearray, daily: {exchange: bytearray, stock: bytearray}}
```

- The records coming out of the group by statement have two fields: the key and the bag of collected records
- The key field is named group
- The bag is named for the alias that was grouped
- For each record in the group, the entire record (including the key) is in the bag

# Relational Operations

## GROUP: More Details

- You can also group on multiple keys
- But the keys must be surrounded by parentheses
- The resulting records still have two fields
- In this case, the group field is a tuple with a field for each key:

```
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;
```

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

# Relational Operations

## DISTINCT

- Description
  - Removes duplicate tuples in a relation
- Syntax
  - DISTINCT alias;
- Usage
  - Use the DISTINCT operator to remove duplicate tuples in a relation
  - DISTINCT does not preserve the original order of the contents (to eliminate duplicates, Pig must first sort the data)

# Relational Operations

## DISTINCT: Example

- Suppose we have relation A.

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

```
DUMP A;
```

```
(8,3,4)
```

```
(1,2,3)
```

```
(4,3,3)
```

```
(4,3,3)
```

```
(1,2,3)
```

- In this example all duplicate tuples are removed.

```
X = DISTINCT A;
```

```
DUMP X;
```

```
(1,2,3)
```

```
(4,3,3)
```

```
(8,3,4)
```

# Relational Operations

## JOIN (inner)

- Description
  - Performs an inner join of two or more relations based on common field values
- Syntax
  - JOIN alias BY {expression|('expression [, expression ...]')} (, alias BY {expression|('expression [, expression ...]')} ...)
- Usage
  - Use the JOIN operator to perform an inner, equijoin 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.
  - The GROUP and JOIN operators perform similar functions
  - GROUP creates a nested set of output tuples while JOIN creates a flat set of output tuples.

# Relational Operations

## JOIN (inner): Example

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

`DUMP A;`

`(1,2,3)`

`(4,2,1)`

`(8,3,4)`

`(4,3,3)`

`(7,2,5)`

`(8,4,3)`

`B = LOAD 'data2' AS (b1:int,b2:int);`

`DUMP B;`

`(2,4)`

`(8,9)`

`(1,3)`

`(2,7)`

`(2,9)`

`(4,6)`

`(4,9)`

# Relational Operations

## JOIN (inner): Example

- In this example relations A and B are joined by their first fields

```
X = JOIN A BY a1, B BY b1;
```

```
DUMP X;
```

```
(1,2,3,1,3)
```

```
(4,2,1,4,6)
```

```
(4,3,3,4,6)
```

```
(4,2,1,4,9)
```

```
(4,3,3,4,9)
```

```
(8,3,4,8,9)
```

```
(8,4,3,8,9)
```



# Relational Operations

## JOIN (outer)

- Description
  - Performs an outer join of two relations based on common field values
- Syntax
  - JOIN left-alias BY left-alias-column [LEFT|RIGHT|FULL] [OUTER], right-alias BY right-alias-column
- Usage
  - Use the JOIN operator with the corresponding keywords to perform left, right, or full outer joins
  - The keyword OUTER is optional for outer joins; the keywords LEFT, RIGHT and FULL will imply left outer, right outer and full outer joins respectively when OUTER is omitted
  - Outer joins will only work provided the relations which need to produce nulls (in the case of non-matching keys) have schemas.
  - Outer joins will only work for two-way joins; to perform a multi-way outer join, you will need to perform multiple two-way outer joins

# Relational Operations

## JOIN (outer): Example

- This example shows a left outer join

```
A = LOAD 'a.txt' AS (n:chararray, a:int);
```

```
B = LOAD 'b.txt' AS (n:chararray, m:chararray);
```

```
C = JOIN A by $0 LEFT OUTER, B BY $0;
```

# Relational Operations

## LIMIT

- Description
  - Limits the number of output tuples
- Syntax
  - LIMIT alias n;
- Description
  - Use the LIMIT operator to limit the number of output tuples.
  - If the specified number of output tuples is equal to or exceeds the number of tuples in the relation, all tuples in the relation are returned.
  - If the specified number of output tuples is less than the number of tuples in the relation, then n tuples are returned
  - There is no guarantee which n tuples will be returned, and the tuples that are returned can change from one run to the next
  - A particular set of tuples can be requested using the ORDER operator followed by LIMIT.

# Relational Operations

## LIMIT: Example #1

- Suppose we have relation A

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

```
DUMP A;
```

```
(1,2,3)
```

```
(4,2,1)
```

```
(8,3,4)
```

```
(4,3,3)
```

```
(7,2,5)
```

```
(8,4,3)
```

# Relational Operations

## LIMIT: Example #1

- In this example output is limited to 3 tuples. Note that there is no guarantee which three tuples will be output

```
X = LIMIT A 3;
```

```
DUMP X;
```

```
(1,2,3)
```

```
(4,3,3)
```

```
(7,2,5)
```

# Relational Operations

## LIMIT: Example #2

- In this example the ORDER operator is used to order the tuples and the LIMIT operator is used to output the first three tuples
  - `B = ORDER A BY f1 DESC, f2 ASC;`
  - `DUMP B;`
    - (8,3,4)
    - (8,4,3)
    - (7,2,5)
    - (4,2,1)
    - (4,3,3)
    - (1,2,3)
  - `X = LIMIT B 3;`
  - `DUMP X;`
    - (8,3,4)
    - (8,4,3)
    - (7,2,5)

# Relational Operations

## ORDER ... BY

- Description
  - Sorts a relation based on one or more fields
- Syntax
  - ORDER alias BY { \* [ASC|DESC] | field\_alias [ASC|DESC] [, field\_alias [ASC|DESC] ...] } [PARALLEL n];
- Usage
  - In Pig, relations are unordered
  - If you order relation A to produce relation X (X = ORDER A BY \* DESC;) relations A and X still contain the same data.
  - If you retrieve relation X (DUMP X;) the data is guaranteed to be in the order you specified (descending).
  - However, if you further process relation X (Y = FILTER X BY \$0 > 1;) there is no guarantee that the data will be processed in the order you originally specified (descending)

# Relational Operations

## ORDER ... BY

- Pig currently supports ordering on fields with simple types or by tuple designator (\*)
- You cannot order on fields with complex types or by expressions

```
A = LOAD 'mydata' AS (x: int, y: map[]);
```

```
B = ORDER A BY x; -- this is allowed because x is a simple type
```

```
B = ORDER A BY y; -- this is not allowed because y is a complex type
```

```
B = ORDER A BY y#'id'; -- this is not allowed because y#'id' is an expression
```



# Relational Operations

## ORDER ... BY: Example

- Suppose we have relation A

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

```
DUMP A;
```

```
(1,2,3)
```

```
(4,2,1)
```

```
(8,3,4)
```

```
(4,3,3)
```

```
(7,2,5)
```

```
(8,4,3)
```

# Relational Operations

## ORDER ... BY: Example

- In this example relation A is sorted by the third field, f3 in descending order
- Note that the order of the three tuples ending in 3 can vary

```
X = ORDER A BY a3 DESC;
```

```
DUMP X;
```

```
(7,2,5)
```

```
(8,3,4)
```

```
(1,2,3)
```

```
(4,3,3)
```

```
(8,4,3)
```

```
(4,2,1)
```

# Relational Operations

## SAMPLE

- Description
  - Selects a random sample of data based on the specified sample size
- Syntax
  - SAMPLE alias size;
- Usage
  - Use the SAMPLE operator to select a random data sample with the stated sample size
  - SAMPLE is a probabilistic operator; there is no guarantee the exact same number of tuples will be returned for a particular sample size each time the operator is used
  - 'size' is either...
    - A constant, range 0 to 1 (for example, enter 0.1 for 10%)
    - A scalar used in an expression
    - The expression can consist of constants or scalars; it cannot contain any columns from the input relation

# Relational Operations

## SAMPLE: Example

- In this example relation X will contain 1% of the data in relation A

```
A = LOAD 'data' AS (f1:int,f2:int,f3:int);
```

```
X = SAMPLE A 0.01;
```

- In this example, a scalar expression is used (it will sample approximately 1000 records from the input)

```
a = load 'a.txt';
```

```
b = group a all;
```

```
c = foreach b generate COUNT(a) as num_rows;
```

```
e = sample a 1000/c.num_rows;
```

# Relational Operations

## SPLIT

- Description
  - Partitions a relation into two or more relations
- Syntax
  - SPLIT alias INTO alias IF expression, alias IF expression [, alias IF expression ...] [, alias OTHERWISE];
- Usage
  - Use the SPLIT operator to partition the contents of a relation into two or more relations based on some expression
  - Depending on the conditions stated in the expression:
    - A tuple may be assigned to more than one relation.
    - A tuple may not be assigned to any relation.

# Relational Operations

## SPLIT

- In this example relation A is split into three relations, X, Y, and Z.

```
A = LOAD 'data' AS (f1:int,f2:int,f3:int);
```

```
DUMP A;  
(1,2,3)  
(4,5,6)  
(7,8,9)
```

```
SPLIT A INTO X IF f1<7, Y IF f2==5, Z IF (f3<6 OR f3>6);
```

```
DUMP X;  
(1,2,3)  
(4,5,6)
```

```
DUMP Y;  
(4,5,6)
```

```
DUMP Z;  
(1,2,3)  
(7,8,9)
```