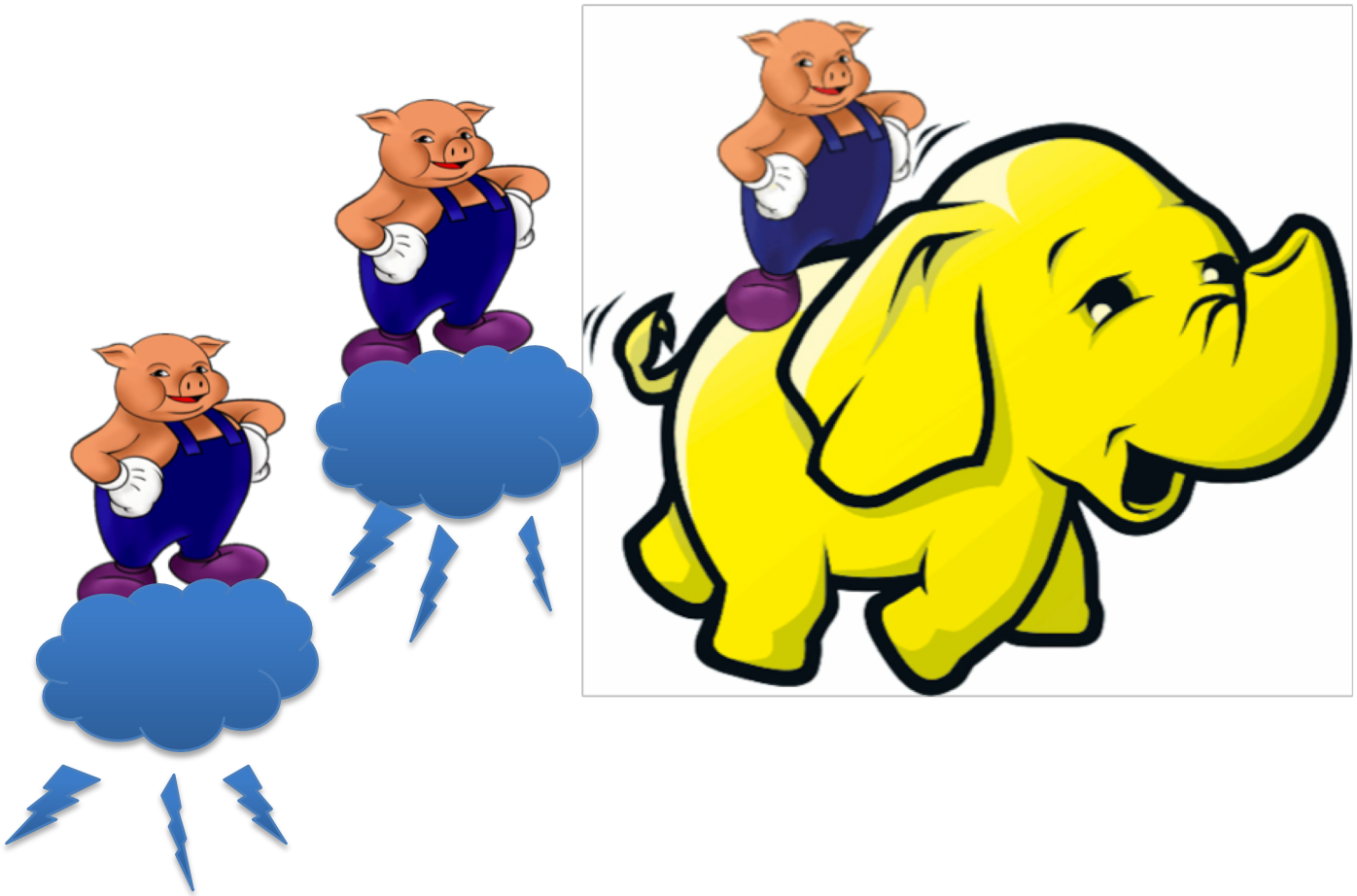
Introduction to Apache Pig



What is Apache Pig?

- Pig is a high level scripting language that is used with Hadoop, initially developed at **Yahoo!**. It uses both HDFS (read and write) and MapReduce (execute jobs). It provides an engine for executing data flows in parallel on Hadoop.
- You can do all the required data manipulations in Hadoop with Pig, such as many traditional data operations (join, sort, filter, etc.). It simplifies joining data and chaining jobs together.
- Pig can utilize the user defined functions (UDF) facility.
- Pig can invoke code written in many languages.
- Pig can be used as a component to build large and complex applications for solving real business problems.

Why Pig?

- Otherwise need to know Java (or another suitable language) to code Map/Reduce.
- Writing native Map/Reduce is time-consuming and not easy.
- It is difficult to make abstractions in Java as in Pig.
- Java is very verbose (400 lines of Java code → 30 lines of Pig script).
- Join operations are very difficult to write in Java and other languages.
- Chaining together MapReduce jobs in Java can be tedious.
- Optimized Pig benefits the users.

A Pig Application Example: ETL

- ETL: A transaction model that describes how a process extracts data from a source, transforms it according to a rule set and then loads it into a **data store**.
- Pig can ingest data from files, streams or other sources using the User Defined Functions (UDFs).
- The UDF feature allows passing the data to more complex algorithms for the transform.
- It can then perform select, iteration, and other transforms over the data.
- Finally Pig can store the results into the Hadoop Data File System.

Pig Case Sensitivity

- **Keywords** in **Pig Latin** are **not case-sensitive**:

  LOAD is equivalent to load.

- **Relations** and **field names** are case-sensitive:

  A = load 'dividends';

  is not equivalent to

  a = load 'dividends';

- **UDF names** are also case-sensitive:

  COUNT is not the same UDF as count.

Three Pig Elements

- **Pig Latin**
  - It is the scripting language of Pig, a dataflow language.
  - It requires no metadata or schema.
  - Its statements are translated into a series of MapReduce jobs.
- **Grunt**
  - The interactive shell.
- **Piggybank**
  - A shared repository of user defined functions (UDFs).

Grunt

- **Grunt** is Pig's interactive shell, which enables users to enter Pig Latin interactively and provides a shell for users to interact with HDFS.
- To enter Grunt, type: **pig** and it shows **grunt>**
- Pig will not start executing the Pig Latin entered until it sees either a **store** or **dump** statement.
- Grunt does basic syntax and semantic checking while you type.
- If you do make a mistake while entering a line of Pig Latin, you can reenter the line and Pig will take the last instance of the line entered.
- To exit Grunt you can type **quit** or enter **Ctrl-D**.

HDFS Commands in Grunt

- Grunt's other major use is to act as a shell for HDFS. All hadoop fs shell commands are available and are accessed using the keyword **fs**. The dash(-) used in the hadoop fs is also required:

  grunt>**fs -ls**

- The following commands can be issued directly without **fs** and **-**
  - **cat *filename*** (can be applied to a directory for all the files)
  - **copyFromLocal *localfile hdfsfile*** (done serially not in parallel)
  - **copyToLocal *hdfsfile localfile*** (done serially not in parallel)
  - **rmr *filename*** (equivalent to **rm -r** in Unix)
- A number of the commands come directly from Unix/Linux shells and will operate in ways that are familiar: **chmod**, **cd**, **cp**, **cat**, **ls**, **mkdir**, **mv**, **pwd**, **rm**, etc.

Controlling Pig from Grunt

Grunt also provides commands for controlling Pig and MapReduce.

- **kill** *jobid*

  Kill the MapReduce job associated with *jobid*.

- **exec** [[-param *param_name* = *param_value*]] [[-param_file *filename*]] *pig_script*

  Execute the Pig Latin script, *pig_script*. Relation names defined in *pig_script* are not imported into Grunt.

- **run** [[-param *param_name* = *param_value*]] [[-param_file *filename*]] *pig_script*

  Execute the Pig Latin script, *pig_script,* in the current Grunt shell. Thus all relation names referenced in *pig_script* are available to Grunt, and the commands in *pig_script* are accessible via the shell history.

Pig's Data Model

- Pig data types:
  - scalar types: containing a single value
  - complex types: containing other types
- Scalar types:
  - int store 4-byte signed integer
  - long long integer, 8-byte signed integer, e.g., 5000000000L
  - float 4-byte floating point number, e.g., 3.14f, 6.022e23f
  - double 8-byte double-precision floating-point number
  - chararray a string or character array, expressed as string literals with single quotes, e.g., 'Cornell'.
  - bytearray a blob or array of bytes
- Complex types:
  - Map
  - Tuple
  - Bag

Pig Complex Data Types

- **Map**: a collection of key value pairs, e.g.,
  ['firstName'#'John', 'lastName'#'Smith', 'id'#1234567]
- **Tuple**: a fixed-length, ordered collection of Pig data elements. Tuples are divided into fields, with each field containing one data element. These elements can be of any type—they do not all need to be the same type.

  ('2019-03-27', 'Error', 404, 'Page not found')

- **Bag**: an unordered collection of tuples, e.g.,

  {
  ('2019-3-27', 'Error', 404, 'Page not found'),
  ('2019-3-27', 'Info', 200, 'OK')
  }

Note: All these complex types can contain data of any types, including other complex data types. Bag is the one type in Pig that is not required to fit into memory.

Pig Nulls

- In Pig Latin nulls are implemented using the SQL definition of null. A null data element means the value is **unknown** or **non-existent**. This might be because the data is missing, an error occurred in processing it, etc.
- Nulls can be used as constant expressions in place of expressions of any type, but Null values are viral for all arithmetic operators, e.g., x + null = null for all values of x.

Pig Schemas

- If a schema is available (e.g., HCatalog, JSON), Pig will make use of it, but if not, Pig will still process the data, making the best guesses it can, based on how the script treats the data.
- The easiest way is to explicitly specify the data types when you load the data:

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

Pig now expects four fields. If it has more, Pig will truncate the extra ones. If it has less, it will pad the end of the record with nulls. This makes Pig load operation flexible.

- If you specify a schema without giving explicit data types, then bytearray is assumed:

```
dividends = load 'dividends' as (exchange, symbol, date,
    dividend);
```

Declare the Schema of Complex Types

You do not have to declare the schema of complex types, but you can if you want to.

For example:

- If your data has a tuple in it, you can declare that field to be a tuple without specifying the fields it contains.
- You can also declare that field to be a tuple that has three columns, all of which are integers.

Specify Each Data Type Inside a Schema Declaration

Data Type	Syntax	Example
int	int	as (a:int)
long	long	as (a:long)
float	float	as (a:float)
double	double	as (a:double)
chararray	chararray	as (a:chararray)
bytearray	bytearray	as (a:bytearray)
map	map[] or map[*type*], where *type* is any valid type, declaring all values in the map to be of this type.	(a:map[], b:map[int])
tuple	tuple() or tuple(*list_of_fields*), where *list_of_fields* is a comma-separated list of field declarations.	as (a:tuple(), b:tuple(x:int, y:int))
bag	bag{} or bag{t:(*list_of_fields*)}, where *list_of_fields* is a comma-separated list of field declarations. Note that the tuple inside the bag must have a name (here t), even though you will never be able to access that tuple t directly.	as (a:bag{}, b:bag{t:(x:int, y:int)})

Data Type Casts

- Pig can **explicitly** convert one data type to another one based on the cast specification. The syntax for casts in Pig is the same as in Java—the type name in parentheses before the value:

  (int)bat#'baseOnBalls' - **(int)**bat#'ibbs'

- Pig can also insert casts **implicitly**. For example:

  daily = load 'NYSE_daily' as (…, close:float, volume:int, …);
  rough = foreach daily generate **volume*close**;

  Pig changes volume*close to **(float)volume*close** to do the calculation without losing precision.

- In general, Pig always widens types to fit when it needs to insert these implicit casts. Thus,
  - int and long →long
  - int or long and float → float
  - int, long, or float and double → double

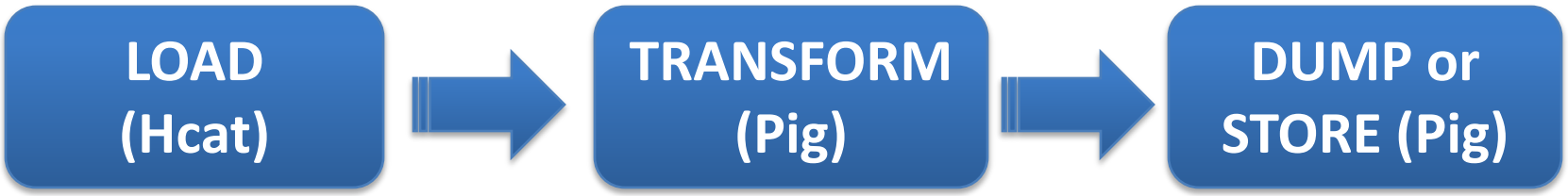
  However, there are no **implicit** casts between numeric types and chararrays or other types.

Pig Supports the Following Type Casts

	To int	To long	To float	To double	To chararray
From int		Yes.	Yes.	Yes.	Yes.
From long	Yes. Any values greater than 2^{31} or less than -2^{31} will be truncated.		Yes.	Yes.	Yes.
From float	Yes. Values will be truncated to int values.	Yes. Values will be truncated to long values.		Yes.	Yes.
From double	Yes. Values will be truncated to int values.	Yes. Values will be truncated to long values.	Yes. Values with precision beyond what float can represent will be truncated.		Yes.
From chararray	Yes. Chararrays with nonnumeric characters result in null.	Yes. Chararrays with nonnumeric characters result in null.	Yes. Chararrays with nonnumeric characters result in null.	Yes. Chararrays with nonnumeric characters result in null.	

Introduction to Pig Latin

- Pig allows users to describe how data from one or more inputs should be read, processed, and then stored to one or more outputs in parallel.
- When Pig Latin data flow works with Hcatalog (Hcat), there is no need for Pig to handle the data location and metadata since Hcat holds this information. The data flow looks like the following:



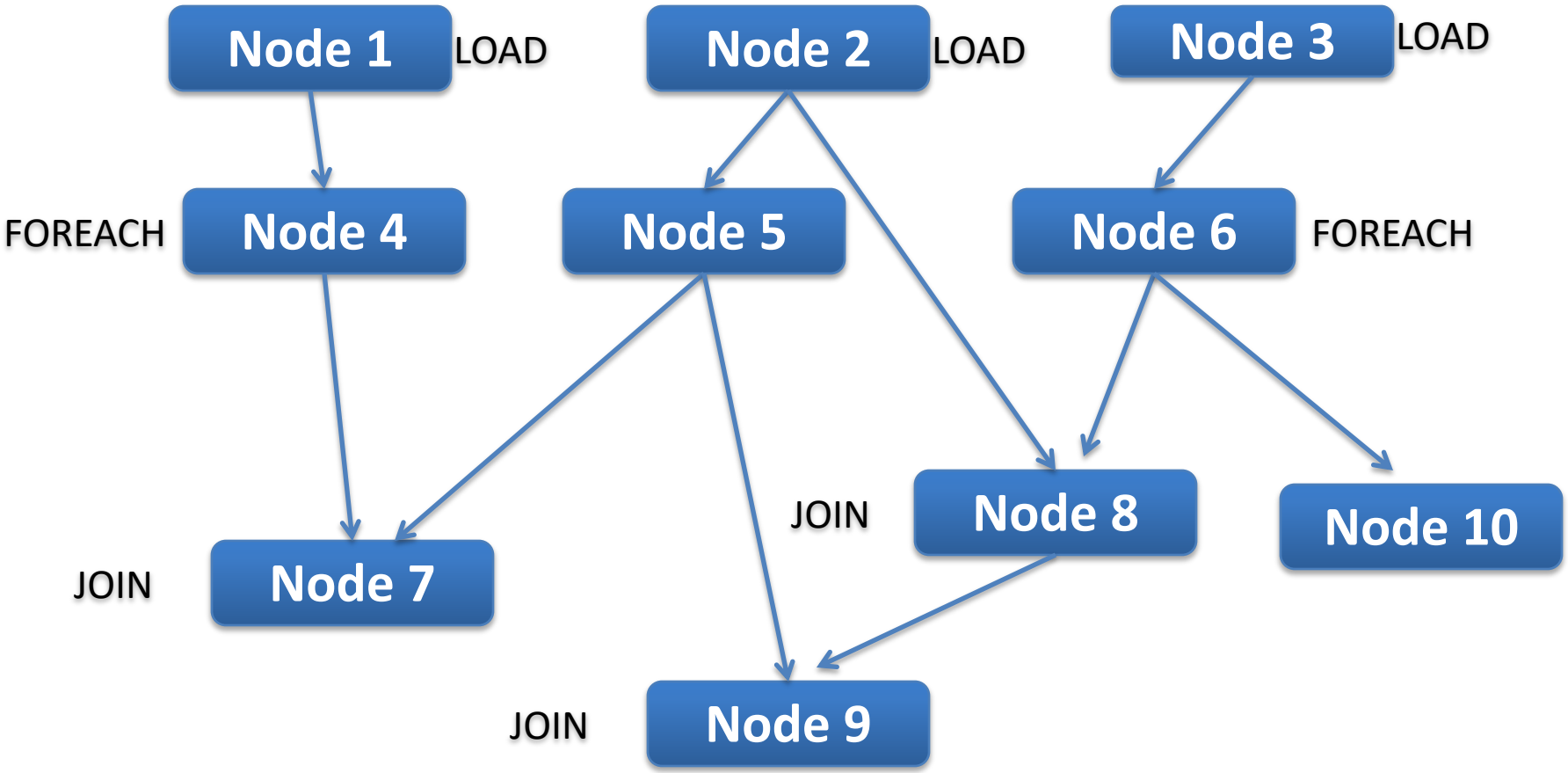
Read data from the file system

Manipulate the data

Output/display data to the screen or store at a location

Pig Latin Structured Processing Flow

A Pig Latin script describes a **directed acyclic graph (DAG)**. The edges are data flows and the nodes are operators that process the data.

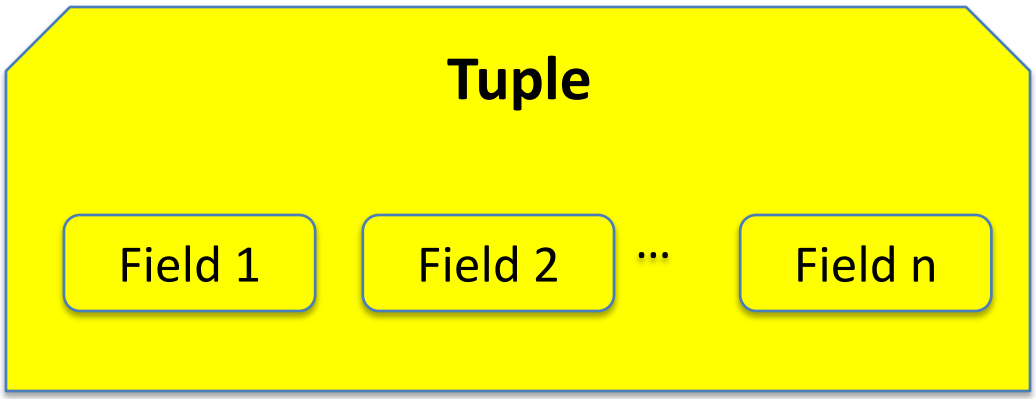


Steps to Transform a Pig Script to MapReduce Jobs

- **Pig interpreter** processes each Pig script statement.
- The statement is added to a logical plan if the statement is valid.
- Use the DUMP or STORE command to execute the logical plan in MapReduce.

Pig Relations

- Pig **relations** are similar to tables in a relational database.
- A relation in Pig is called a **bag**.
- A bag is a collection of unordered tuples, which correspond to the rows in a table. Unlike a database table, these tuples are not required to have the same number of fields, nor the fields at the same position (or the column) have the same data type.
- A tuple represents an ordered set of fields.
- A field is a piece of data and it has a name (must start with a character).



Pig Relation Name: Alias

- In **input = load 'data'**, input is the name of the relation that results from loading the dataset, data.
- A relation name is referred to as an **alias**. Relation names look like variables, but they are not. Once made, it is permanent.
- It is possible to reuse relation names; for example, this is legitimate:

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

However, it is not recommended. It looks here as if you are reassigning A, but really you are creating new relations called A, losing track of the old relations called A.

Pig Latin Comments

Pig Latin has two types of **comment operators**:

- SQL-style single-line comments (**--**)

  A = load 'foo'; **--**this is a single-line comment

- Java-style multiline comments (**/* */**).

```
/*
This Pig script is to …
.
.
.
*/

B = load /* a comment in the middle */'bar';
```

Input and Output

- **The Load Statement:** It is to specify input. By default, load looks for data on HDFS in a **tab-delimited** file using the default load function **PigStorage().** For example,

  div = load '/data/examples/dividends';

  It looks for a file called **dividends** in the **/data/examples** directory.

- You can use relative path names. By default, your Pig jobs will run in your home directory on HDFS, /user/*yourlogin* (in our case, it is **/user/root**).

- You can also specify a full URL for the path, for example,

  hdfs://nnode.acme.com/data/examples/dividends

  to read the file from the HDFS instance that has nnode.acme.com as a NameNode.

Input and Output (cont'd)

- **The Load Function:** a function for loading data of different formats or sources since in practice most of the data will not be in tab-separated text files. For example,
  - to load your data from HBase, use the loader for HBase:

    ```
    divs = load 'dividends' using HBaseStorage();
    ```

  - to read comma-separated text data, use PigStorage() with an argument to indicate which character to use as a separator (here, comma):

    ```
    divs = load 'NYSE_dividends' using PigStorage(',');
    ```

Input and Output (cont'd)

- **The as clause:** The load statement also can have an as clause, which allows you to specify the schema of the data you are loading, for example:

```
divs = load 'dividends' as
(exchange, symbol, date, dividends);
```

Input and Output (cont'd)

- **The Store Statement:** to write out somewhere the data that have been processed. In many ways it is the mirror image of the load statement.
- By default, Pig stores your data on HDFS in a tab-delimited file using PigStorage():

  store processed **into** '/data/examples/processed';
- You can specify a different store function (e.g., HBaseStorage()) with a using clause:

  store processed **into** 'processed' using HBaseStorage();
- You can also pass arguments to your store function, e.g., storing your data as comma-separated text data; PigStorage() takes an argument to indicate which character to use as a separator:

  store processed **into** 'processed' using PigStorage(',');

(Note: When writing to a filesystem, "processed" will be a directory with part files rather than a single file.)

Input and Output (cont'd)

- **The Dump Statement:** It directs the output of your script to your computer screen, for example:

```
dump processed;
```

Most cases： store data

During debugging／prototyping／ad hoc jobs：
display data

Some Important Pig Keywords (Relational Operators) Working on Relations

Relational operators are the main tools Pig Latin provides to operate on your data. They allow you to transform it by sorting, grouping, joining, projecting, and filtering.

- FOREACH
- FILTER
- GROUP
- ORDER
- DISTINCT
- LIMIT
- JOIN
- SAMPLE
- PARALLEL

The FOREACH Keyword

- The **foreach** keyword takes a set of expressions and applies them to every record in the data pipeline. From these expressions it **generates** new records to send down the pipeline to the next operator. For example, the following code loads an entire record, but then removes all but the **user** and **id** fields from each record:

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

- Another example, which extracts the **age** and **salary** fields from each record:

```
e1 = LOAD ' /ipig/input/file1' USING PigStorage(',') AS
  (name:chararray, age:int, zip:int, salary:double);
f= FOREACH e1 GENERATE age, salary;
```

Field References in FOREACH

- **Reference by field name:**

```
prices = load 'NYSE_daily.txt' as (exchange, symbol, date, open,
            high, low, close, volume, adj_close);
gain1 = foreach prices generate close - open;
```

- **Positional references**: preceded by a dollar sign ($) and starts from zero. It is useful when the schema is unknown or undeclared.

```
gain2 = foreach prices generate $6 - $3;
```

- **Refer to all fields using an asterisk (*).**
- **Refer to ranges of fields using two periods (..).**

```
beginning = foreach prices generate ..open; -- produces exchange,
                                               symbol, date, open
middle = foreach prices generate open..close; -- produces open, high,
                                                  low, close
end = foreach prices generate volume..; -- produces volume, adj_close
```

Binary Condition Operator (*Bincond*) in FOREACH

10 == 10 ? 1 : 4 **--returns 1**

A Boolean test

The value to return if the test is true

The value to return if the test is false

- If the test returns null, bincond returns null.
- Both value arguments of the bincond must return the same type

2 == 3 ? 1 : 4 --returns 4
null == 2 ? 1 : 4 -- returns null

2 == 2 ? 1 : 'fred' -- type error; both values must be of the same type

Extract Data From Complex Types (in FOREACH)

- Use the projection operators to extract data:
  - For **maps**, use 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';
```

  - For **tuples**, use the dot operator (.):

```
A = load 'input' as (t:tuple(x:int, y:int));
B = foreach A generate t.x, t.$1;
```

  - For **bags**, the dot operator (.) is used but the projection is not as straightforward as maps and tuples since bags do not guarantee that their tuples are stored in any order. When you project fields in a bag, you are creating a new bag with only those fields.

Projection for Bag Types (in FOREACH)

- Produce a single field in bag. It produces a new bag whose tuples have only one field in them. In the following example b.x produced is a bag and not a scalar value.

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

- 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);
```

The FILTER Keyword

- The **filter** statement allows you to select which records will be retained in your data pipeline. It contains a **predicate**. If the predicate evaluates to true for a given record, that record will be passed down the pipeline; otherwise, it will not.
- The operators: ==, !=, >, >=, <, <= (for all scalar data types)
- == and != can be applied to maps and tuples, but none of them can be applied to bags.
- Some examples:
  - logevents = LOAD 'input/my.log' AS (date:chararray, level:chararray, code:int, message:chararray);
    severe = **FILTER** logevents **BY** (level == 'severe' AND code >= 500);
  - divs = load 'dividends' as (exchange:chararray, symbol:chararray, date:chararray, dividends:float);
    startswithCM = **filter** divs **by** symbol **matches** 'CM.*';
  - dividendNotNull = **filter** divs **by** dividends **is not null**;
  - dividendIsNull = **filter** divs **by** dividends **is null**;

The GROUP Keyword

- The **group** statement collects together records with the same key. In Pig there is no direct connection between group and aggregate functions. Instead, it collects all records with the same value for the provided key together into a bag, and then passes it to an aggregate function if you want or do something else.
- The records coming out of the **group** statement have two fields, the **key,** named **group**, and the **bag** of collected records, named for the alias that was grouped. The bag has the same schema as alias.
- An example: pass the bag to an aggregate function

```
daily = load 'NYSE_daily' as (exchange:chararry, stock:bytearray);
grpd = group daily by stock;
cnt = foreach grpd generate group, COUNT(daily);
```

The GROUP Keyword (cont'd)

- An example: store the bag output

```
daily = load 'NYSE_daily' as (exchange, stock);
grpd = group daily by stock;
store grpd into 'by_group';
```

- Describe the group result:

```
describe grpd;
```

grpd: {group: bytearray,daily: {exchange: bytearray, stock: bytearray}}

- Another example: filter, group and then store the bag output

```
logevents = LOAD 'input/my.log' AS (date:chararray,
    level:chararray, code:int, message:chararray);
severe = FILTER logevents BY (level == 'severe' AND code >= 500);
grouped = GROUP severe BY code;
STORE grouped INTO 'output/severeevents';
```

The GROUP Keyword (cont'd)

- Calculate average dividends for each stock symbol.

```
divs = load '/NYSE_dividends.txt' as (exchange:chararray,
        symbol:chararray, date:chararray, dividends:float);
grpd = group divs by symbol;
avgdiv = foreach grpd generate group, AVG(divs.dividends);
dump avgdiv;
```

```
The Job job_1428719433416_0063 has been started successfully.
You can always go back to Query History for results after the run.

(CA,0.03999999910593033)
(CB,0.3499999940395355)
(CE,0.03999999910593033)
(CF,0.10000000149011612)
(CI,0.03999999910593033)
(CL,0.4299999997019768)
(CM,0.777999997138977)
(CP,0.2212500013411045)
(CR,0.20000000298023224)
(CS,0.09600000083446503)
    ⋮
```

The ORDER Keyword

- The **order** statement sorts the data, producing a total order of the output data. Total order means that not only is the data sorted in each partition, it is also guaranteed that all records in partition n are less than all records in partition n - 1 for all n. Nulls are always the smallest.
- An example:

```
employees = LOAD 'pig/input/file1' USING PigStorage(',') AS
                    (name:chararray, age:int, zip:int, salary:double);
sorted = ORDER employees BY salary;
```

- Sort descendingly by appending **desc** to a key in the sort. In order statements with multiple keys, desc applies only to the key it immediately follows.

```
daily = load 'NYSE_daily' as (exchange:chararray, symbol:chararray,
date:chararray, open:float, high:float, low:float, close:float,
volume:int, adj_close:float);
byclose = order daily by close desc, open;
dump byclose;
```

The DISTINCT Keyword

- The **distinct** statement removes duplicate records. It works only on entire records, not on individual fields.

```
-- find a distinct list of ticker symbols for each exchange
daily = load 'NYSE_daily' as (exchange:chararray,
        symbol:chararray);
uniq = distinct daily;
```

The LIMIT Keyword

- The limit statement returns a limited number line of the results.
- Pig does not guarantee the order in which records are produced. Thus, the example script below could return different results every time.
- Putting an **order** immediately before the limit will guarantee that the same results are returned every time.

```
employees = LOAD 'pig/input/file1' USING PigStorage(',') AS
        (name:chararray, age:int, zip:int, salary:double);
agegroup = GROUP employees BY age;
h= LIMIT agegroup 100;
```

This example here will return at most 100 lines (if your input has less than 100 lines total, it will return them all).

The JOIN Keyword

- The **join** statement is one of the workhorses of data processing; it selects records from one input to put together with records from another input. This is done by indicating keys for each input. When those keys are equal, the two rows are joined. Records for which no match is found are dropped.
- **join** preserves the names of the fields of the inputs passed to it.

```
e1 = LOAD 'pig/input/file1' USING PigStorage(',') AS
     (name:chararray, age:int, zip:int, salary:double);
e2 = LOAD 'pig/input/file2' USING PigStorage(',') AS
     (name:chararray, phone:chararray);
e3 = JOIN e1 BY name, e2 BY name;
DESCRIBE e3;
DUMP e3;
```

The JOIN Keyword (cont'd)

- You can also join on multiple keys:
  - must have the same number of keys;
  - must be of the same or compatible types (meaning that an implicit cast can be inserted).

```
daily = load 'NYSE_daily' as (exchange, symbol, date, open, high,
            low, close, volume, adj_close);
divs = load 'NYSE_dividends' as (exchange, symbol, date,
            dividends);
jnd = join daily by (symbol, date), divs by (symbol, date);
```

The JOIN Keyword (cont'd)

- Pig also supports **outer joins**. In outer joins, records that do not have a match on the other side are included, with null values being filled in for the missing fields. Outer joins can be **left**, **right**, or **full**.

```
daily = load 'NYSE_daily' as (exchange, symbol, date, open,
        high,low, close, volume, adj_close);
divs = load 'NYSE_dividends' as (exchange, symbol, date,
        dividends);
jnd = join daily by (symbol, date) left outer, divs by
        (symbol, date);
```

The SAMPLE Keyword (cont'd)

- The **sample** statement offers a simple way to get a sample of your data. It reads through all of the data but returns only a percentage of rows. What percentage it returns is expressed as a double value, between 0 and 1.

```
divs = load 'NYSE_dividends';
tenPercent = sample divs 0.1;
```

So, the above example samples 10% of the the divs relation.

The PARALLEL Keyword

- One of Pig's core claims is that it provides a language for parallel data processing. To do this, it provides the **parallel** clause.
- The parallel clause can be attached to any relational operator in Pig Latin. However, it controls only reduce-side parallelism.

```
daily = load 'NYSE_daily' as (exchange, symbol, date, open,
        high, low, close, volume, adj_close);
bysymbl = group daily by symbol parallel 10;
```

In this example, parallel will cause the MapReduce job spawned by Pig to have 10 reducers.

The PARALLEL Keyword (cont'd)

- The parallel clauses apply only to the statements to which they are attached; they do not carry through the script.

```
daily = load 'NYSE_daily' as (exchange, symbol,
        date, open, high, low, close, volume, adj_close);
bysymbl = group daily by symbol parallel 10;
average = foreach bysymbl generate group,
        AVG(daily.close) as avg;
sorted = order average by avg desc parallel 2;
```

The PARALLEL Keyword (cont'd)

- You can set a script-wide value for parallel using the **set command**. For example, all MapReduce jobs will be done with 50 reducers:

```
set default_parallel 50;
daily = load 'NYSE_daily' as (exchange, symbol,
            date, open, high, low, close, volume, adj_close);
bysymbl = group daily by symbol;
average = foreach bysymbl generate group,
            AVG(daily.close) as avg;
sorted = order average by avg desc;
```

- When a default parallel level is set, you can still add a parallel clause to any statement to override the default value. Thus it can be helpful to set a default value as a base to use in most cases, and specifically add a parallel clause only when you have an operator that needs a different value.

Pig Debugging Operators

- **ILLUSTRATE** to view the step-by-step execution of a series of statements.

  ILLUSTRATE alias;

- **EXPLAIN** to view the logical, physical, or map/reduce execution plans to compute a relation.

  EXPLAIN [–script pigscript] [–out path] […] alias;

- **DESCRIBE** to review the schema of a relation.

  DESCRIBE alias;

- Some tips for debugging,
  - Use the local mode to test a script before running it in the cluster.
  - It is slow but there is no waiting for a slot.
  - The logs for your operations appear on the screen rather than on a remote task node.