CS523 - BDT
Big Data Technologies

Apache Pig

(Building High-Level Dataflows over Map-Reduce)

Need for High-Level Languages

- Hadoop MR is great for large-data processing, but writing Java programs for everything is verbose and slow – we are doing low-level language coding to perform low level operations.

- Development cycle of MR is very long and so for productivity, we need higher level tools.

- MR allows you, as a programmer, to specify a map function followed by a reduce function, but working out how to fit your data processing into this pattern, which often requires multiple MR stages, can be a challenge.

- Not everyone wants to (or can) write Java code!

- Solution: Get help from a few animals!

Need for High-Level Languages

- **Idea:**
  - Develop higher-level data processing languages to facilitate large-data processing
    - **Pig**: Pig Latin is a bit like Perl
    - **Hive**: HQL is like SQL

  - Higher-level language "compiles down" to MapReduce jobs

- So, the people like business analysts, data scientists, statisticians etc. who don't know Java and so cannot write MapReduce, now will be able to run their queries on the big data by using Pig or Hive.

Hadoop Ecosystem
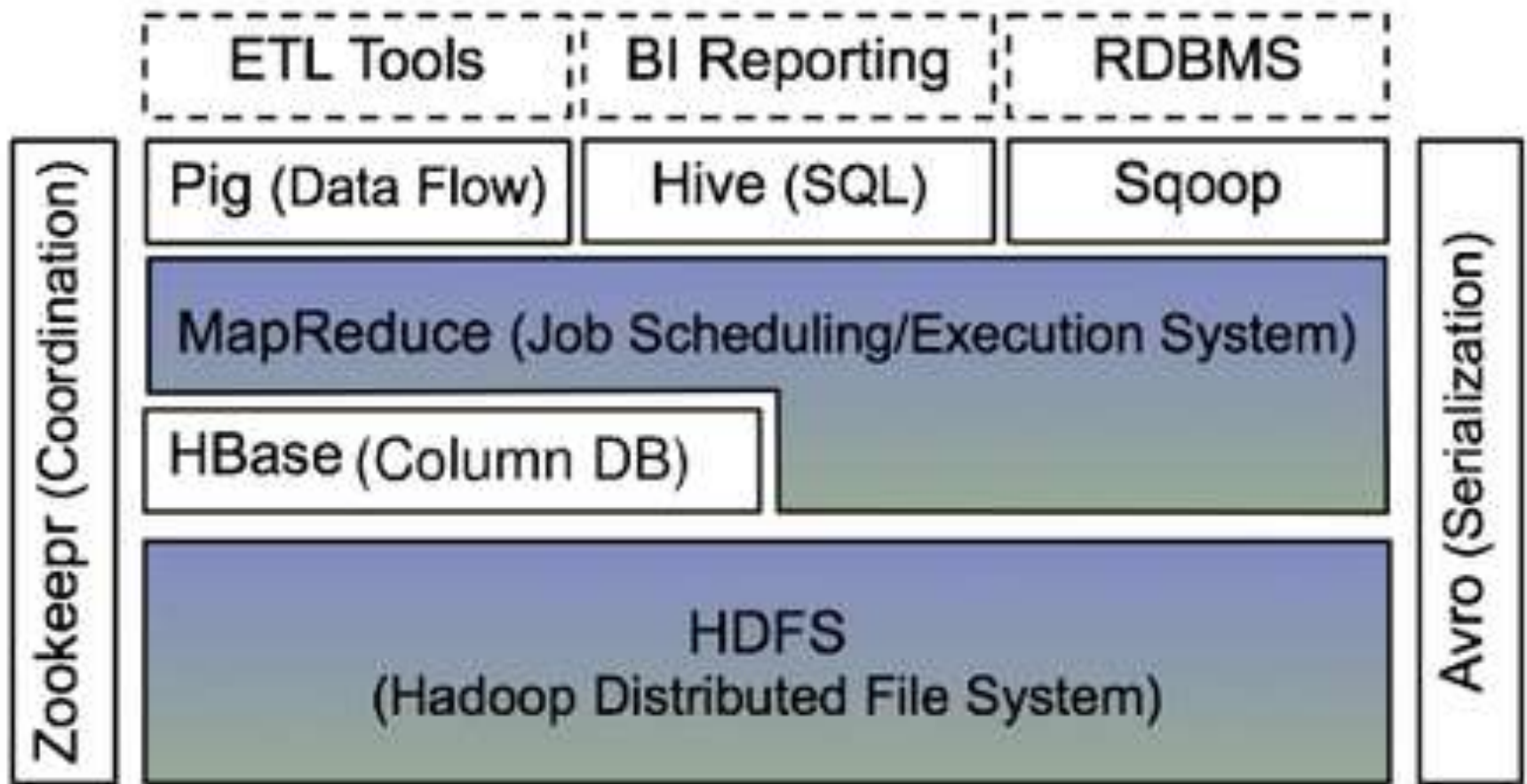
Apache Pig

- Large-scale data processing system in Hadoop

- Pig was developed by Yahoo in the year 2007 to cut down on the time required for development. (Pig became apache top level project in 2010.)

  - Roughly 1/3 of all Yahoo! internal jobs run in Pig

- Very easy to learn, read and write if you are familiar with SQL.

  - Pig's data flow language Pig Latin is a simple query algebra expressing data transformations and applying functions to records.

- Pig provides the users with a wide range of nested data types such as Maps, Tuples and Bags that are not present in MR along with some major data operations such as Ordering, Filtering, and Joins.

Pig and MapReduce

- Pig provides users with several advantages over using MapReduce directly.

- Pig Latin provides all of the standard data-processing operations, such as join, filter, group by, order by, distinct, union, etc.

- MapReduce provides the group by operation directly (that is what the shuffle plus reduce phases are), and it provides the order by operation indirectly through the way it implements the sorting.

- Filter 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 provides some complex, nontrivial implementations of these standard data operations.

Pig and MapReduce

- There is, of course, a cost to all this. It is possible to develop algorithms in MapReduce that cannot be done easily in Pig. And the developer gives up a level of control.

- A good engineer can always, given enough time, write code that will out perform a generic system.

- So for less common algorithms or extremely performance-sensitive ones, MapReduce is still the right choice.

- Basically this is the same situation as choosing to code in Java versus a scripting language such as Python. Java has more power, but due to its lower-level nature, it requires more development time than scripting languages.

- Developers will need to choose the right tool for each job.

Pig Components

Client machine

Hadoop Cluster

Pig Script → Pig Execution Engine → MapReduce → Hadoop Job

8
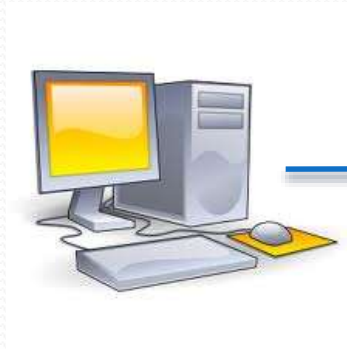
Pig Components

- **Pig Latin**
  - Command based language
  - Designed specifically for data transformation and flow expression
  - A *Pig Latin program* is made up of a series of operations, or transformations, that are applied to the i/p data to produce o/p.
- **Execution Engine**
  - The environment in which Pig Latin commands are executed
  - Currently there is support for Local and Hadoop modes
- **Pig Compiler (converts Pig Latin to MapReduce)**
  - Compiler strives to optimize execution
  - You automatically get optimization improvements with Pig updates

- Under the covers, Pig turns the transformations into a series of MapReduce jobs, but as a programmer you are mostly unaware of this, which allows you to focus on the data rather than the nature of the execution.

Pig Execution Engine
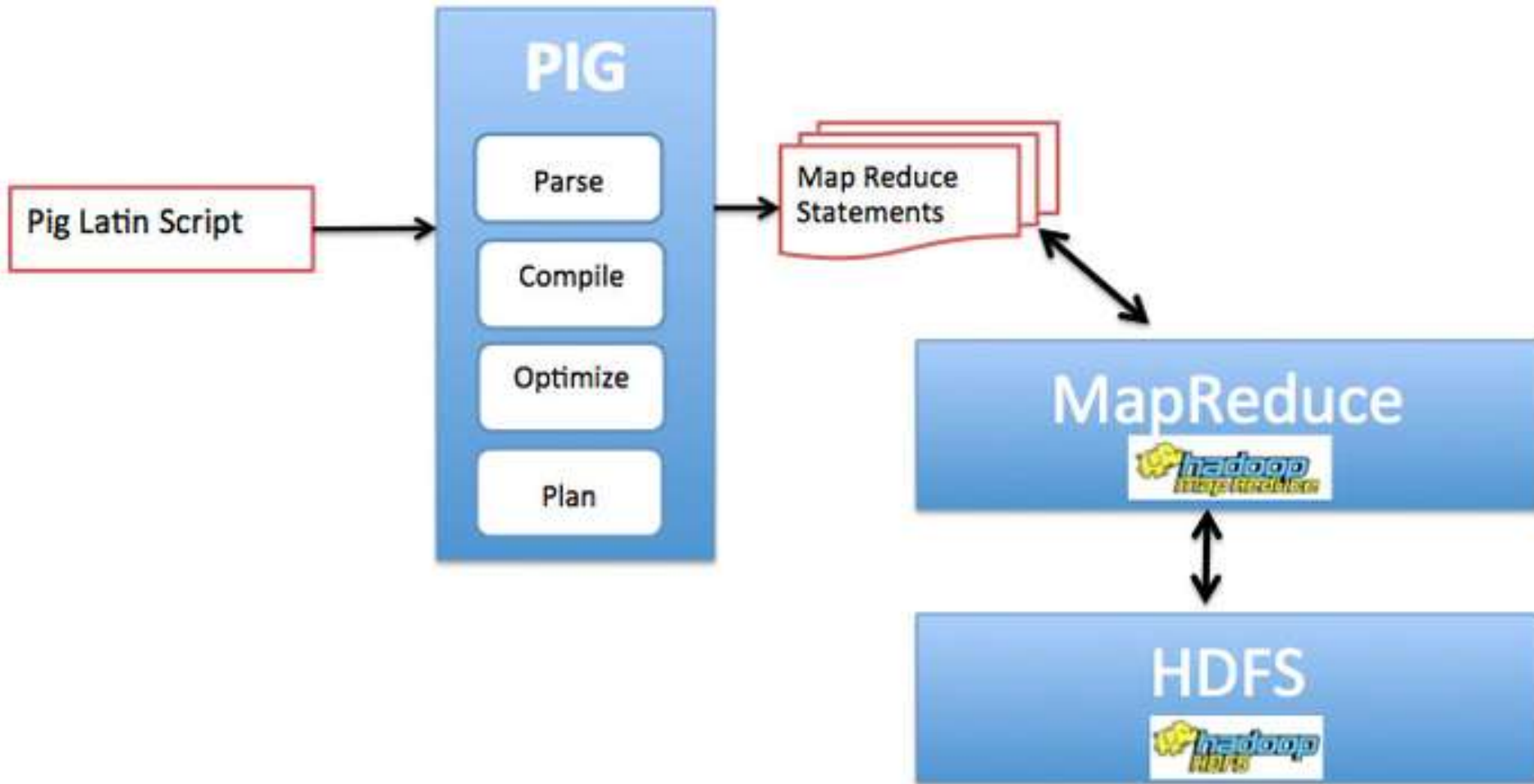
Pig Execution Modes

1. **Local**

    - Executes in a single JVM

    - All files are installed and run using your local host and file system

    - Does not involve a real Hadoop cluster

    - Great for development, experimentation and prototyping

    - Specify local mode using the -x flag

    ```
    $ pig -x local
    grunt>
    ```

Pig Execution Modes

2. **Hadoop Mode (MapReduce mode)**

- Default mode

- Access to a Hadoop cluster and HDFS installation

- Pig renders Pig Latin into MapReduce jobs and executes them on the cluster

- Can execute against pseudo distributed or fully-distributed Hadoop installation

```
$ pig -x mapreduce
grunt>
```

OR

```
$ pig
grunt>
```

Running Pig

- **Script (Batch mode)**
  - Execute commands written in a file
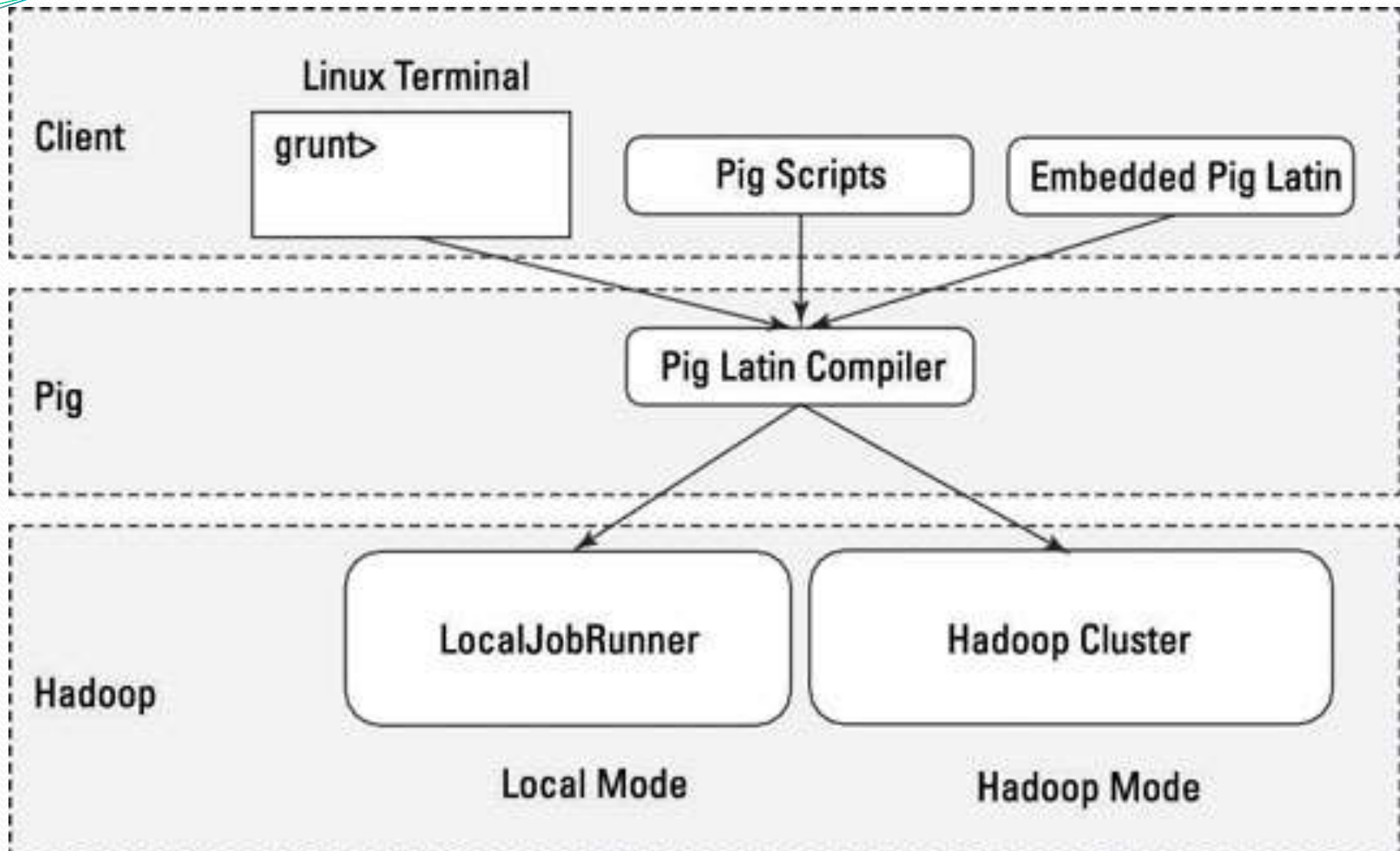 - $ pig scriptFile.pig

- **Grunt (Interactive mode)**
  - Interactive Shell for executing Pig Commands
  - Started when script file is NOT provided
  - Can execute scripts from Grunt via run or exec commands

- **Embedded**
  - Execute Pig commands using *PigServer* class
    - Just like JDBC to execute SQL
  - Can have programmatic access to Grunt via *PigRunner* class
  - Pig UDFs

Running Pig

Grunt Shell

- Grunt is Pig's interactive shell. It enables users to enter Pig Latin interactively and provides a shell for users to interact with HDFS.

- "Pig" for entering into Grunt shell and "Quit" or "Ctrl-D" for exiting the Grunt Shell

- Grunt has line-editing facilities like Ctrl-E will move the cursor to the end of the line, etc.

- Grunt supports most of the file system commands

```
grunt>fs -ls

grunt>cat filename

grunt>copyFromLocal localfile hdfsfile

grunt>copyToLocal hdfsfile localfile

grunt>rmr filename
```

Pig Latin

- Pig Latin is a data flow language in which each processing step results in a new data set, or relation.
- The program consists of a collection of statements. A statement can be thought of as an operation or a command.
- Statements are usually terminated with a semicolon.
- Pig Latin has two forms of comments. Double hyphens (--) are used for single-line comments and C-style comments (/* */) for multiline comments.

```
/*
* Description of my program spanning
* multiple lines.
*/
A = LOAD 'input/pig/join/A'; -- What's in A?
B = LOAD 'input/pig/join/B';
C = JOIN A BY $0, /* ignored */ B BY $1;
DUMP C;
```

Pig Latin

- Pig Latin has a list of keywords that have a special meaning in the language and cannot be used as identifiers.

- These include the operators (LOAD, ILLUSTRATE), commands (cat, ls), expressions (matches, FLATTEN), and functions (DIFF, MAX).

➢ There is no need to be concerned with map, shuffle, and reduce phases when using Pig. It will manage decomposing the operators in your script into the appropriate MapReduce phases.

Conventions and Case Sensitivity

- **Case Sensitive**
  - Alias names
  - Pig Latin Functions
- **Case Insensitive**
  - Pig Latin Keywords

Alias
Case Sensitive

Function
Case Sensitive

```
counts = FOREACH charGroup GENERATE group, COUNT(c);
```

Alias
Case Sensitive

Keywords
Case Insensitive

- **General conventions**
  - Upper case is a system keyword
  - Lowercase is something that you provide

18

Pig Latin Example

```
students = LOAD 'students.txt' AS (id, name, gpa);

distinction = FILTER students BY gpa > 3.5;

rankings = ORDER distinction BY gpa DESC;

STORE rankings INTO 'students_with_distinction';
```

- Typically, Pig would load a dataset in a **bag** and then creates new bags by modifying the existing ones.

- In the example above, the *students.txt* file is loaded in *students* bag, then a new bag called *distinction* is created for students whose gpa is above 3.5.

- Later a new bag *rankings* is created and finally the results are stored in a file called *students_with_distinction.*

Pig Latin Building Blocks

- **Building blocks**
  - Field (atomic value) – piece of data
  - Tuple – ordered set of fields, represented with "(" and ")"
    `(10.4, 5, word, 4, field1)`
  - Bag – collection of tuples, represented with "{" and "}"
    - `{ (10.4, 5, word, 4, field1), (this, 1, blah) }`

- **Similar to Relational Database**
  - Bag is a table in the database
  - Tuple is a row in a table
  - Bags do not require that all tuples contain the same number of fields (Unlike relational table)
    - If Pig tries to access a field that does not exist, a null value is substituted.

Pig Latin Simple Data Types

Simple Type	Description
int	Signed 32-bit integer
long	Signed 64-bit integer
float	32-bit floating point
double	64-bit floating point
chararray	Character array (string) in Unicode UTF-8 format
bytearray	Byte array (blob)
boolean	boolean

Pig Latin Complex Data Types

- **<u>Tuple</u>**: Sequence of fields of any type
  - enclosed by ( ), items separated by ","
  - Empty tuple is also valid: ( )
  - Example: (1,'pomegranate')

- **<u>Bag</u>**: Unordered collection of tuples, possibly with duplicates
  - enclosed by { }, tuples separated by ","
  - Empty bag is also valid: { }
  - Example: {(1,'pomegranate'),(2)}

- **<u>Map</u>**: Set of key-value pairs; keys must be character arrays, but values may be any type
  - enclosed by [ ], items separated by " , ", key and value separated by "#"
  - Non-empty map: [key1#value1,key2#value2]
  - Empty map is also valid: [ ]
  - Example: ['a'#'pomegranate']

Categories of Functions in Pig

- *Eval function*
  - A function that takes one or more expressions and returns another expression. E.g. MAX
- *Filter function*
  - A special type of Eval function that returns a logical Boolean result. E.g. IsEmpty
- *Load function*
  - A function that specifies how to load data into a relation from external storage.
- *Store function*
  - A function that specifies how to save the contents of a relation to external storage.

Built-in Functions in Pig Latin

Category	Function	Description
Eval	AVG	Calculates the average (mean) value of entries in a bag.
	CONCAT	Concatenates byte arrays or character arrays together.
	COUNT	Calculates the number of non-null entries in a bag.
	COUNT_STAR	Calculates the number of entries in a bag, including those that are null.
	DIFF	Calculates the set difference of two bags. If the two arguments are not bags, returns a bag containing both if they are equal; otherwise, returns an empty bag.
	MAX	Calculates the maximum value of entries in a bag.
	MIN	Calculates the minimum value of entries in a bag.
	SIZE	Calculates the size of a type. The size of numeric types is always 1; for character arrays, it is the number of characters; for byte arrays, the number of bytes; and for containers (tuple, bag, map), it is the number of entries.
	SUM	Calculates the sum of the values of entries in a bag.
	TOBAG	Converts one or more expressions to individual tuples, which are then put in a bag. A synonym for ().
	TOKENIZE	Tokenizes a character array into a bag of its constituent words.
	TOMAP	Converts an even number of expressions to a map of key-value pairs. A synonym for [].
	TOP	Calculates the top n tuples in a bag.
	TOTUPLE	Converts one or more expressions to a tuple. A synonym for {}.

Built-in Functions in Pig Latin

Category	Function	Description
Filter	IsEmpty	Tests whether a bag or map is empty.
Load/Store	PigStorage	Loads or stores relations using a field-delimited text format. Each line is broken into fields using a configurable field delimiter (defaults to a tab character) to be stored in the tuple's fields. It is the default storage when none is specified.[a]
	TextLoader	Loads relations from a plain-text format. Each line corresponds to a tuple whose single field is the line of text.
	JsonLoader, JsonStorage	Loads or stores relations from or to a (Pig-defined) JSON format. Each tuple is stored on one line.
	AvroStorage	Loads or stores relations from or to Avro datafiles.
	ParquetLoader, ParquetStorer	Loads or stores relations from or to Parquet files.
	OrcStorage	Loads or stores relations from or to Hive ORCFiles.
	HBaseStorage	Loads or stores relations from or to HBase tables.

Pig Latin Relational Operators

Category	Operator	Description
Loading and Storing	LOAD	Loads data from the file system.
	STORE	Saves a relation to the file system or other storage.
	DUMP	Prints a relation to the console.
Filtering	FILTER	Removes unwanted rows from a relation.
	DISTINCT	Removes duplicate rows from a relation.
	FOREACH….GENERATE	Adds or removes fields from a relation.
	STREAM	Transforms a relation using an external program.
Grouping and Joining	JOIN	Joins two or more relations.
	GROUP	Groups the data in a single relation.
	COGROUP	Groups the data in two or more relations.
	CROSS	Creates the cross product of two or more relations.
Sorting	ORDER	Sorts a relation by one or more fields.
	LIMIT	Limits the size of a relation to a maximum number of tuples.
Combining & Splitting	UNION	Combines two or more relations into one.
	SPLIT	Splits a relation into two or more relations.

Statements in Pig Latin

- When a Pig Latin program is executed, each statement is parsed in turn. If there are syntax errors or other (semantic) problems, such as undefined aliases, the interpreter will halt and display an error message.

- The interpreter builds a logical plan for every relational operation, which forms the core of a Pig Latin program.

- The logical plan for the statement is added to the logical plan for the program so far, and then the interpreter moves on to the next statement.

- It's important to note that no data processing takes place while the logical plan of the program is being constructed.

- For example, consider the Pig Latin program on the next slide.

```
-- max_temp.pig: Finds the maximum temperature by year
records = LOAD 'input/ncdc/micro-tab/sample.txt'
  AS (year:chararray, temperature:int, quality:int);
filtered_records = FILTER records BY temperature != 9999 AND
  quality IN (0, 1, 4, 5, 9);
grouped_records = GROUP filtered_records BY year;
max_temp = FOREACH grouped_records GENERATE group,
  MAX(filtered_records.temperature);
DUMP max_temp;
```

- When the Pig Latin interpreter sees the first line containing the LOAD statement, it confirms that it is syntactically and semantically correct and adds it to the logical plan, but it does not load the data from the file (or even check whether the file exists).

- Indeed, where would it load it? Into memory?

Schemas

- Fields are given some names through the use of a schema and fields may be referenced by their names.

Three possibilities:
- Define a schema which includes both field name and data type
- Define a schema which includes only field name
  - Default data type will be bytearray
- Need not define any kind of schema
  - Fields are un-named
  - Fields data type will be bytearray

Field References

- A field can also be referenced by position.

- This is accomplished by specifying a $ and a number.

- Field numbers start with zero. So with a schema of *(name: chararray, age:int, salary:float),* the age field could be referenced by position as $1.

- If a schema is not specified when data is loaded, then the fields can only be referenced by position.

Simple Pig Latin Example

```
$ pig                                          Start Grunt with default
grunt> cat cs523/Examples/pig/test.txt         MapReduce mode
a 1
d 4                                             Load contents of text files
c 9                                             into a Bag named records
k 6
grunt> records = LOAD 'cs523/Examples/pig/test.txt' AS
(letter:chararray, count:int);
grunt> dump records;                            Display records bag to
...                                             the screen
org.apache.pig.backend.hadoop.executionengine.mapReduceL
ayer.MapReduceLauncher - 50% complete
...
(a,1)
(d,4)                                           Results of the bag named records
(c,9)                                           are printed to the screen
(k,6)
grunt>
```

32

LOAD Command

- **LOAD** operator is used to read data from a source and place that data into a relation (bag of tuples).

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

- **data – name of the directory or file**
  - Must be in single quotes
- **USING – specifies the load function to use**
- **AS – assign a schema to incoming data**
  - Assigns names to fields
  - Declares types for fields

Available LOAD Functions

- Pig supplies many load functions:

  - **PigStorage, TextLoader, BinStorage, JsonLoader, AvroStorage, HBaseStorage etc.**

- You can also code your own load function if your data is in some proprietary format not supported by the supplied load functions.

- The default load function is *PigStorage*. This reads data that is in a delimited format with the default delimiter being the tab character.

  - If some other delimiting character is to be used, then that character is supplied in single quotes.

- The *TextLoader* reads in a **line of text** and this line of text is placed **into a single tuple**.

LOAD Command Example

Data

```
records = LOAD 'cs523/Examples/pig/log.txt'
USING PigStorage(',')
AS (userId:chararray, timestamp:long, query:chararray);
```

Schema

User selected Load Function, there are a lot of choices or you can implement your own

DUMP and STORE statements

- **No action is taken until DUMP or STORE commands are encountered**
  - Pig will parse, validate and analyze statements but will not execute them
- **DUMP – displays the results to the screen**
- **STORE – saves results (typically to a file)**

Nothing is executed; Pig will optimize this entire chunk of script

```
records = LOAD
'cs523/Examples/pig/test.txt' as
(letter:chararray, count:int);
...
...
...
...
DUMP final_bag;           The fun begins here
```

STORE Command for Large Data

- Hadoop data is usually quite large and it doesn't make sense to print it on the screen.

- The common pattern is to persist results to Hadoop (HDFS, HBase)

- This is done with STORE command.

```
STORE records INTO 'output' USING PigStorage('*');
```

- In this example PigStorage stores the contents of "records" into files with fields that are delimited with an asterisk (*).

- The STORE function specifies that the files will be located in a directory named "output" and that the files will be named *part-nnnnn*.
  (for example, part-m-00000 or part-r-00000).

Limiting Output

- For information and debugging purposes you can print a small sub-set of the output to the screen.

```
grunt> records = LOAD 'cs523/Examples/pig/log.txt' AS
(userId:chararray, timestamp:long, query:chararray);

grunt> toPrint = LIMIT records 5;

grunt> DUMP toPrint;
```

Only 5 records will be displayed

Pig Latin - Diagnostic Tools

- **Dump Operator-** It is used to display the output of pig Latin statements on the screen, so that developers can debug the code.

- **Display the structure of the Bag**

  ```
  grunt> DESCRIBE <bag_name>;
  ```

- **Display Execution Plan**

  ```
  grunt> EXPLAIN <bag_name>;
  ```

  - Produces Various reports
    - Logical Plan
    - MapReduce Plan (physical plan) – which shows how the physical operators are grouped into MR jobs. This is a good way to find out how many MR jobs Pig will run for your query.

- **Illustrate how Pig engine transforms the data**
  - ```
    grunt> ILLUSTRATE <bag_name>;
    ```

39

ILUSTRATE Command

The **illustrate** operator gives you the step-by-step execution of a sequence of statements.

```
grunt> chars = LOAD 'cs523/Examples/pig/b.txt'
                    AS (letter:chararray);

grunt> g = GROUP chars BY letter;

grunt> ILLUSTRATE g;
```

```
-------------------------------------------
| chars     | letter:chararray     |
-------------------------------------------
|           | k                    |
|           | k                    |
-------------------------------------------

-------------------------------------------------------------------------------------
| g      | group:chararray     | chars:bag{:tuple(letter:chararray)}
|
-------------------------------------------------------------------------------------

|        | k                   | {(k), (k)}
|
-------------------------------------------------------------------------------------
```

Inner Bag

Outer Bag

Pig Latin – GROUP BY

```
grunt> chars = LOAD 'cs523/Examples/pig/b.txt' AS (letter:chararray);
grunt> describe chars;
chars: {letter: chararray}
grunt> dump chars;
(a)
(k)
...
...
(k)
(c)
grunt> charGroup = GROUP chars BY letter;
grunt> describe charGroup;
charGroup: {group: chararray, chars: {(letter: chararray)}}
grunt> dump charGroup;
(a,{(a),(a),(a)})
(c,{(c),(c)})
(i,{(i),(i),(i)})
(k,{(k),(k),(k),(k)})
(l,{(l),(l)})
```

Creates a new bag with element named group and element named chars

The chars bag is grouped by "letter"; therefore 'group' element will contain unique values

'chars' element is a bag itself and contains all tuples from 'chars' bag that match the value from 'letter'

Pig Latin – GROUP BY

- The GROUP BY statement collects records with the same key together into a bag.

- 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.

Inner vs. Outer Bag

```
grunt> chars = LOAD 'cs523/Examples/pig/b.txt' AS
(letter:chararray);
```

```
grunt> charGroup = GROUP chars by letter;
```

```
grunt> describe charGroup;
```

charGroup: {group: chararray,chars: {(letter:
chararray)}}

```
grunt> dump charGroup;
```

(a,{(a),(a),(a)})

(c,{(c),(c)})

(i,{(i),(i),(i)})

(k,{(k),(k),(k),(k)})

(l,{(l),(l)})

Inner Bag

Outer Bag

Grouping on Multiple Columns

```
➢ daily = LOAD 'NYSE_daily' AS (exchange, stock, date,
                                              dividends);
➢ grpd = GROUP daily BY (exchange, stock);
➢ DESCRIBE grpd;
grpd: {group: (exchange:bytearray, stock:bytearray),
             daily: {exchange:bytearray, stock:bytearray,
                   date: bytearray, dividends:bytearray}}
```

Pig Latin – FOREACH GENERATE

FOREACH <bag> GENERATE <data>

- This operation is used to apply transformation to each element in the bag, so that respective action is performed to generate new data items.

- **FOREACH** 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 those familiar with database terminology, it is Pig's projection operator.

```
grunt> result = FOREACH bag GENERATE f1;
```

FOREACH Example

```
grunt>records=LOAD 'data/a.txt' AS (c:chararray, i:int);
grunt>dump records;
(a,1)
(d,4)
(c,9)
(k,6)
grunt> counts = FOREACH records GENERATE i;
grunt> dump counts;
(1)
(4)
(9)
(6)
```

For each row emit 'i' field

FOREACH with Functions

FOREACH B GENERATE group, FUNCTION(A);

```
grunt> chars = LOAD 'data/b.txt' AS (c:chararray);
grunt> charGroup = GROUP chars by c;
grunt> dump charGroup;
(a,{(a),(a),(a)})
(c,{(c),(c)})
(i,{(i),(i),(i)})
(k,{(k),(k),(k),(k)})
grunt> describe charGroup;
charGroup: {group:chararray, chars:{(c: chararray)}}
grunt> counts = FOREACH charGroup GENERATE group, COUNT(chars);
grunt> dump counts;
(a,3)
(c,2)
(i,3)
(k,4)
```

For each row in 'charGroup' bag, emit group field and count the number of items in 'chars' bag

More Examples of FOREACH

➢ 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;

➢ 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

➢ A = LOAD 'input' AS (t:tuple(x:int, y:int));

➢ B = FOREACH A GENERATE t.x, t.$1;

GROUP ALL

- You can also use **ALL** to group together all of the records in your pipeline into one single record.

- The record coming out of GROUP ALL has the *chararray* literal **all** as a key.

- Usually this does not matter because you will pass the bag directly to an aggregate function such as COUNT. But if you plan to store the record or use it for another purpose, you might want to project out the artificial key first.

Example of GROUP ALL

```
grunt> dump charGroup;
(a,{(a),(a),(a)})
(c,{(c),(c)})
(i,{(i),(i),(i)})
(k,{(k),(k),(k),(k)})
(l,{(l),(l)})

grunt> g = GROUP charGroup ALL;
grunt> dump g;
(all,{(l,{(l),(l)}),(k,{(k),(k),(k),(k)}),(i,{(i),(i),(i)}),(c,{(c),
                    (c)}),(a,{(a),(a),(a)})})

grunt> DESCRIBE g;
g: {group:chararray, charGroup: {(group:chararray, chars:{(letter:
                                  chararray)})}}

grunt> cnt = FOREACH g GENERATE COUNT(charGroup);
(5)
```

Pig Latin – ORDER BY

- The *ORDER BY* statement sorts your data for you, producing a total order of your output data.

- Total order means that not only is the data sorted in each partition of your data, it is also guaranteed that all records in partition *n* are less than all records in partition *n - 1* for all *n*.

- When your data is stored on HDFS, where each partition is a part file, this means that cat will output your data in order.

```
daily = load 'NYSE_daily' as (exchange:chararray,
        symbol:chararray, date:chararray, open:float,
          high:float, low:float, close:float, volume:int,
            adj_close:float);
bydate = ORDER daily BY date;
bydatensymbol = ORDER daily BY date, symbol;
```

Pig Latin – ORDER BY

- It is also possible to reverse the order of the sort by appending DESC to a key in the sort.

- In ORDER BY statements with multiple keys, DESC applies only to the key it immediately follows. Other keys will still be sorted in ascending order:

```
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; -- open still sorted in ascending
                                              order
```

TOKENIZE Function

- **Splits a string into tokens and outputs as a bag of tokens**
  - Separators are: space, double quotes("), comma(,), parenthesis(()), star(*) etc.

  `{(chararray)} TOKENIZE(chararray input)`

  - *Parameter:*
    - *source*: the chararray to split
  - *Returns:*
    - *input* split on whitespace, with each resulting value being placed in its own tuple and all tuples placed in the bag

TOKENIZE Function

```
grunt>linesOfText = LOAD 'data/c.txt' AS
                                  (line:chararray);

grunt>dump linesOfText;
(this is a line of text)
(yet another line of text)
(third line of words)
grunt>tokenBag = FOREACH linesOfText GENERATE
                                  TOKENIZE(line);

grunt>dump tokenBag;
({(this),(is),(a),(line),(of),(text)})
({(yet),(another),(line),(of),(text)})
({(third),(line),(of),(words)})
grunt> describe tokenBag;
tokenBag:{bag_of_tokenTuples:{tuple_of_tokens:
                        (token: chararray)}}
```

Split each row line by space and return a bag of tokens

Each row is a bag of words produced by TOKENIZE function

54

FLATTEN Operator

- Sometimes there is data in a tuple or a bag and if we want to remove the level of nesting from that data, then FLATTEN can be used.

- FLATTEN un-nests bags and tuples.

- FLATTEN is not a function, it's an operator and it re-arranges output.

- For tuples, the FLATTEN operator will substitute the fields of a tuple in place of a tuple, whereas un-nesting bags is a little complex because it requires creating new tuples.

FLATTEN Operator

```
grunt> dump tokenBag;
({(this),(is),(a),(line),(of),(text)})
({(yet),(another),(line),(of),(text)})
({(third),(line),(of),(words)})

grunt>flatBag = FOREACH tokenBag GENERATE flatten($0);
grunt> dump flatBag;
(this)
(is)
(a)
···
...
(text)
(third)
(line)
(of)
(words)
```

Nested structure:
bag of
bags of tuples

Each row is flatten
resulting in a
bag of simple tokens

Elements in a bag can
be referenced by index

FILTER..BY Operator

- FILTER selects tuples from a relation based on some condition.

- FILTER is commonly used to select the data that you want; or, conversely, to filter out (remove) the data you don't want.

```
New_alias = FILTER alias BY expression;
```

- Use the FILTER operator to work with tuples or rows of data. If you want to work with columns of data, use the FOREACH...GENERATE operation.

FILTER Operator Example

- A = LOAD '/home/cloudera/cs523/Examples/pig/a.txt' USING PigStorage (',') 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)

- **X = FILTER A BY a3 == 3;**
- DUMP X;

(1,2,3)

(4,3,3)

(8,4,3)

FILTER Operator Example

- A = LOAD '/home/cloudera/cs523/Examples/pig/a.txt' USING PigStorage (',') 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)

- **X = FILTER A BY (a1 == 8) OR (NOT (a2+a3 > a1));**
- DUMP X;

(4,2,1)

(8,3,4)

(7,2,5)

(8,4,3)

59

Pig Latin - DISTINCT

- The distinct statement is very simple. It removes duplicate records. It works only on entire records, not on individual fields:

```
-- find a distinct list of ticker symbols for each exchange
-- This load will truncate the records, picking up just the first two
   fields.
daily = LOAD 'NYSE_daily' AS (exchange:chararray,
                                  symbol:chararray);
uniq = DISTINCT daily;
```

- Because it needs to collect *like* records together in order to determine whether they are duplicates, distinct forces a reduce phase. It does make use of the combiner to remove any duplicate records it can delete in the map phase.

60

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.

Example using GROUP BY - GENERATE:

```
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;
```

Example using DISTINCT:

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

JOIN Operator

- Joins two datasets together by a common attribute.

- Joining datasets in MapReduce takes some work on the part of the programmer, whereas Pig has very good built-in support for join operations.

- By default, Join operator always performs an inner join (equi join in relational algebra) - Only those records from both tables are generated that have identical values in the joined columns.

JOIN Operator Example

- Consider the relations A and B:

```
grunt> DUMP A;
(2,Tie)
(3,Hat)
(1,Scarf)
(4,Coat)
```

```
grunt> DUMP B;
(Hank,2)
(Joe,2)
(Hank,4)
(Ali,0)
(Eve,3)
```

- We can join the two relations on the numerical (identity) field in each:

```
grunt> C = JOIN A BY $0, B BY $1;
grunt> DUMP C;
(2,Tie,Hank,2)
(2,Tie,Joe,2)
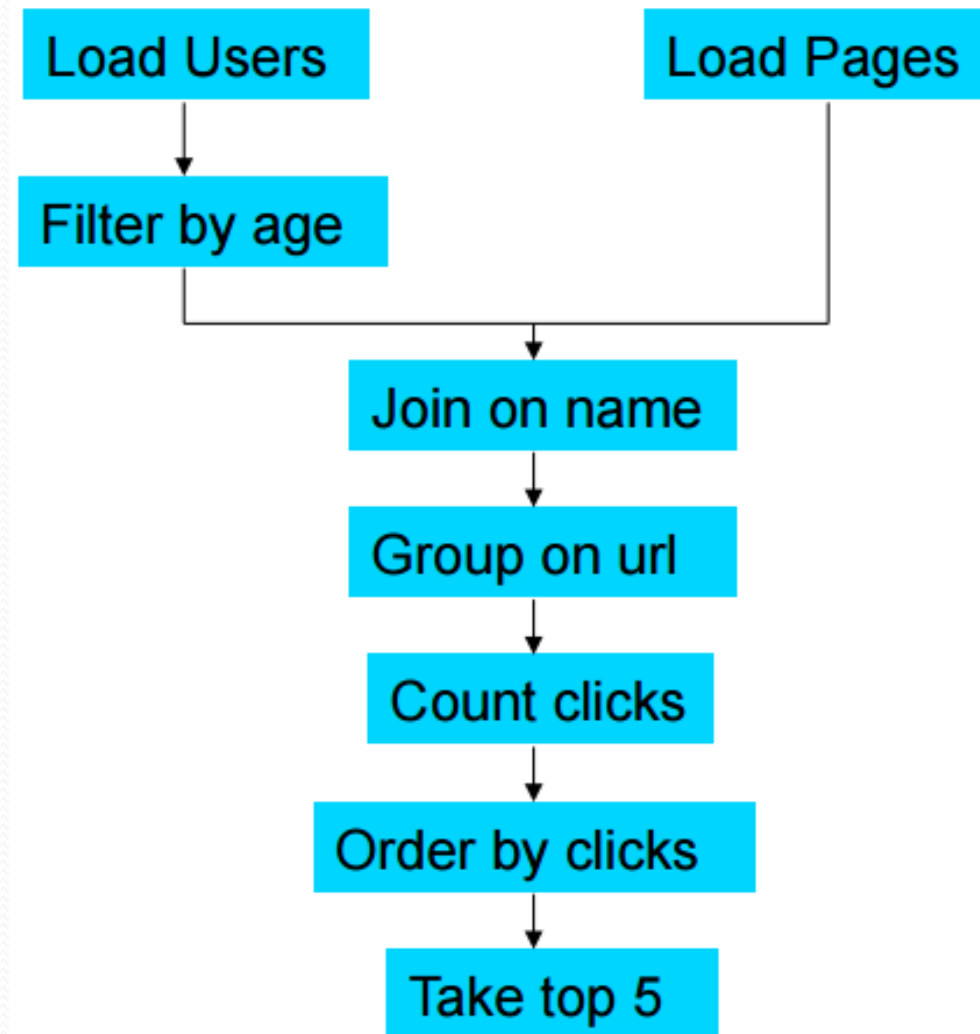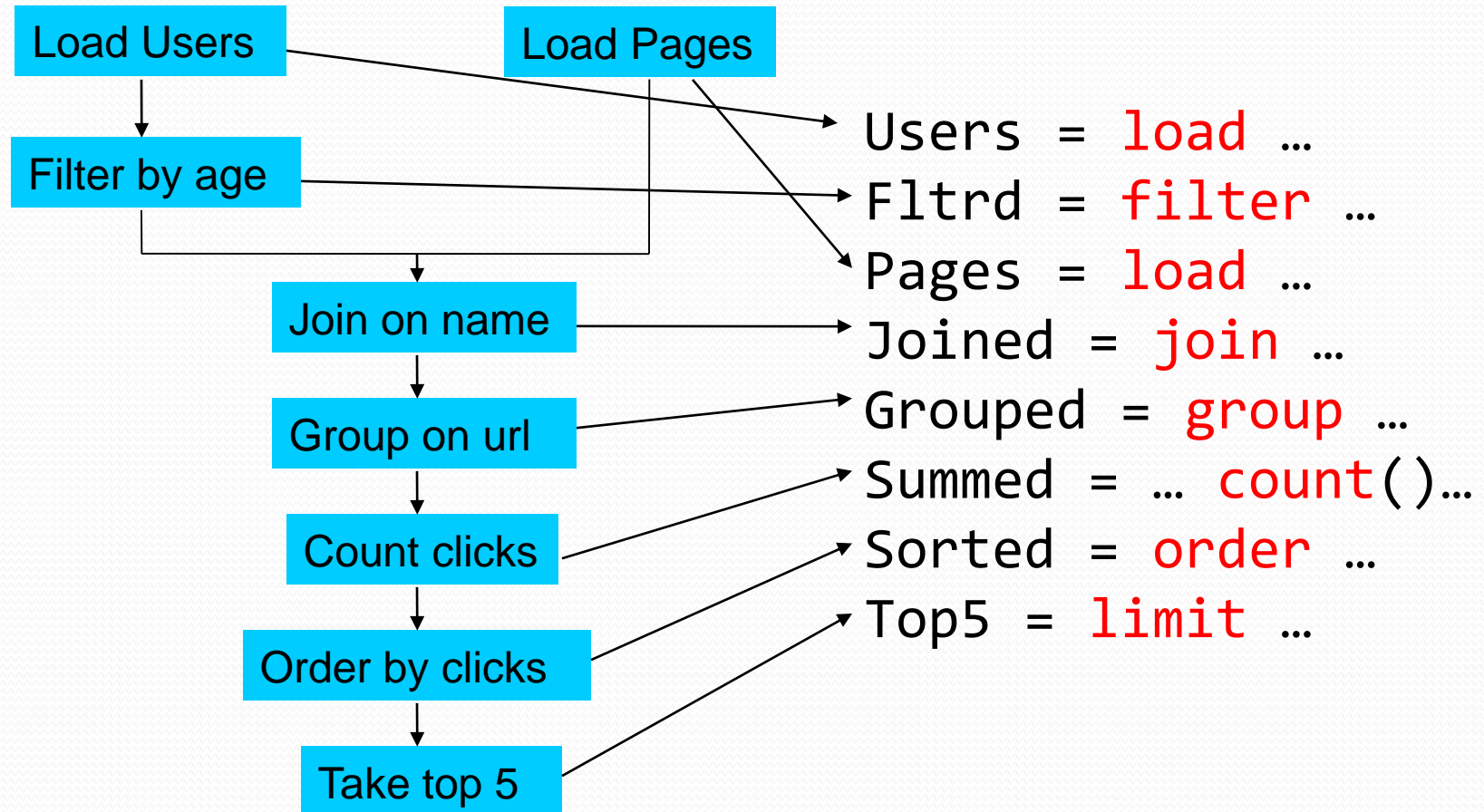(3,Hat,Eve,3)
(4,Coat,Hank,4)
```

JOIN Operator Example

- **Problem Statement:**
Suppose you have *user* data in one file (users.csv (name,age)), *website* data in another file (pages.csv (user, url)), and you need to find the top 5 most visited sites by users aged 18 - 25.

Ease of Translation


```
Users = load …
Fltrd = filter …
Pages = load …
Joined = join …
Grouped = group …
Summed = … count()…
Sorted = order …
Top5 = limit …
```

Diagram blocks:
Load Users → Filter by age
Load Pages
Join on name
Group on url
Count clicks
Order by clicks
Take top 5

Ease of Translation

Every group or join operation forms a map-reduce boundary

Job 1

Job 2

Job 3

```
Users = load …
Fltrd = filter …
Pages = load …
Joined = join …
Grouped = group …
Summed = … count()…
Sorted = order …
Top5 = limit …
```

Load Users
Load Pages
Filter by age
Join on name
Group on url
Count clicks
Order by clicks
Take top 5
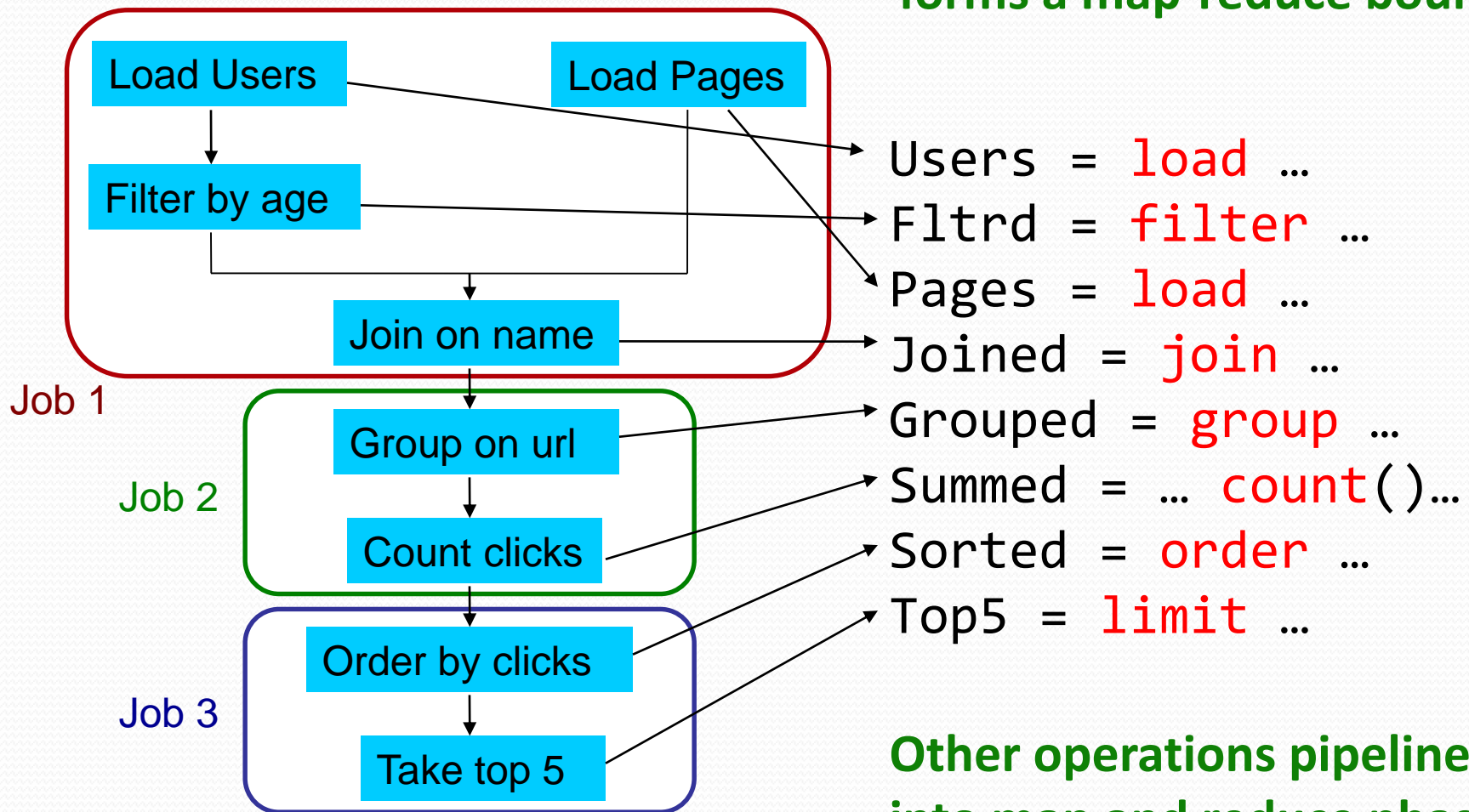
Other operations pipelined into map and reduce phases

In MapReduce


```
import java.io.IOException;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

import org.apache.hadoop.fs.Path;
import org.apache.hadoop.io.LongWritable;
import org.apache.hadoop.io.Text;
import org.apache.hadoop.io.Writable;
import org.apache.hadoop.io.WritableComparable;
import org.apache.hadoop.mapred.FileInputFormat;
import org.apache.hadoop.mapred.FileOutputFormat;
import org.apache.hadoop.mapred.JobConf;
import org.apache.hadoop.mapred.KeyValueTextInputFormat;
import org.apache.hadoop.mapred.Mapper;
import org.apache.hadoop.mapred.MapReduceBase;
import org.apache.hadoop.mapred.OutputCollector;
import org.apache.hadoop.mapred.RecordReader;
import org.apache.hadoop.mapred.Reducer;
import org.apache.hadoop.mapred.Reporter;
import org.apache.hadoop.mapred.SequenceFileInputFormat;
import org.apache.hadoop.mapred.SequenceFileOutputFormat;
import org.apache.hadoop.mapred.TextInputFormat;
import org.apache.hadoop.mapred.jobcontrol.Job;
import org.apache.hadoop.mapred.jobcontrol.JobControl;
import org.apache.hadoop.mapred.lib.IdentityMapper;

public class MRExample {
    public static class LoadPages extends MapReduceBase
        implements Mapper<LongWritable, Text, Text, Text> {

        public void map(LongWritable k, Text val,
                OutputCollector<Text, Text> oc,
                Reporter reporter) throws IOException {
            // Pull the key out
            String line = val.toString();
            int firstComma = line.indexOf(',');
            String key = line.substring(0, firstComma);
            String value = line.substring(firstComma + 1);
            Text outKey = new Text(key);
            // Prepend an index to the value so we know which file
            // it came from.
            Text outVal = new Text("1" + value);
            oc.collect(outKey, outVal);
        }
    }
    public static class LoadAndFilterUsers extends MapReduceBase
        implements Mapper<LongWritable, Text, Text, Text> {

        public void map(LongWritable k, Text val,
                OutputCollector<Text, Text> oc,
                Reporter reporter) throws IOException {
            // Pull the key out
            String line = val.toString();
            int firstComma = line.indexOf(',');
            String value = line.substring(firstComma + 1);
            int age = Integer.parseInt(value);
            if (age < 18 || age > 25) return;
            String key = line.substring(0, firstComma);
            Text outKey = new Text(key);
            // Prepend an index to the value so we know which file
            // it came from.
            Text outVal = new Text("2" + value);
            oc.collect(outKey, outVal);
        }
    }
    public static class Join extends MapReduceBase
        implements Reducer<Text, Text, Text, Text> {

        public void reduce(Text key,
                Iterator<Text> iter,
                OutputCollector<Text, Text> oc,
                Reporter reporter) throws IOException {
            // For each value, figure out which file it's from and
store it
            // accordingly.
            List<String> first = new ArrayList<String>();
            List<String> second = new ArrayList<String>();

            while (iter.hasNext()) {
                Text t = iter.next();
                String value = t.toString();
                if (value.charAt(0) == '1')
first.add(value.substring(1));
                else second.add(value.substring(1));
```

```
                reporter.setStatus("OK");
            }

            // Do the cross product and collect the values
            for (String s1 : first) {
                for (String s2 : second) {
                    String outval = key + "," + s1 + "," + s2;
                    oc.collect(null, new Text(outval));
                    reporter.setStatus("OK");
                }
            }
        }
    }
    public static class LoadJoined extends MapReduceBase
        implements Mapper<Text, Text, Text, LongWritable> {

        public void map(
                Text k,
                Text val,
                OutputCollector<Text, LongWritable> oc,
                Reporter reporter) throws IOException {
            // Find the url
            String line = val.toString();
            int firstComma = line.indexOf(',');
            int secondComma = line.indexOf(',', firstComma);
            String key = line.substring(firstComma, secondComma);
            // drop the rest of the record, I don't need it anymore,
            // just pass a 1 for the combiner/reducer to sum instead.
            Text outKey = new Text(key);
            oc.collect(outKey, new LongWritable(1L));
        }
    }
    public static class ReduceUrls extends MapReduceBase
        implements Reducer<Text, LongWritable, WritableComparable,
Writable> {

        public void reduce(
                Text key,
                Iterator<LongWritable> iter,
                OutputCollector<WritableComparable, Writable> oc,
                Reporter reporter) throws IOException {
            // Add up all the values we see

            long sum = 0;
            while (iter.hasNext()) {
                sum += iter.next().get();
                reporter.setStatus("OK");
            }

            oc.collect(key, new LongWritable(sum));
        }
    }
    public static class LoadClicks extends MapReduceBase
        implements Mapper<WritableComparable, Writable, LongWritable,
Text> {

        public void map(
                WritableComparable key,
                Writable val,
                OutputCollector<LongWritable, Text> oc,
                Reporter reporter) throws IOException {
            oc.collect((LongWritable)val, (Text)key);
        }
    }
    public static class LimitClicks extends MapReduceBase
        implements Reducer<LongWritable, Text, LongWritable, Text> {

        int count = 0;
        public void reduce(
                LongWritable key,
                Iterator<Text> iter,
                OutputCollector<LongWritable, Text> oc,
                Reporter reporter) throws IOException {

            // Only output the first 100 records
            while (count < 100 && iter.hasNext()) {
                oc.collect(key, iter.next());
                count++;
            }
        }
    }
    public static void main(String[] args) throws IOException {
        JobConf lp = new JobConf(MRExample.class);
        lp.setJobName("Load Pages");
        lp.setInputFormat(TextInputFormat.class);
```

```
        lp.setOutputKeyClass(Text.class);
        lp.setOutputValueClass(Text.class);
        lp.setMapperClass(LoadPages.class);
        FileInputFormat.addInputPath(lp, new
Path("/user/gates/pages"));
        FileOutputFormat.setOutputPath(lp,
            new Path("/user/gates/tmp/indexed_pages"));
        lp.setNumReduceTasks(0);
        Job loadPages = new Job(lp);

        JobConf lfu = new JobConf(MRExample.class);
        lfu.setJobName("Load and Filter Users");
        lfu.setInputFormat(TextInputFormat.class);
        lfu.setOutputKeyClass(Text.class);
        lfu.setOutputValueClass(Text.class);
        lfu.setMapperClass(LoadAndFilterUsers.class);
        FileInputFormat.addInputPath(lfu, new
Path("/user/gates/users"));
        FileOutputFormat.setOutputPath(lfu,
            new Path("/user/gates/tmp/filtered_users"));
        lfu.setNumReduceTasks(0);
        Job loadUsers = new Job(lfu);

        JobConf join = new JobConf(MRExample.class);
        join.setJobName("Join Users and Pages");
        join.setInputFormat(KeyValueTextInputFormat.class);
        join.setOutputKeyClass(Text.class);
        join.setOutputValueClass(Text.class);
        join.setMapperClass(IdentityMapper.class);
        join.setReducerClass(Join.class);
        FileInputFormat.addInputPath(join, new
Path("/user/gates/tmp/indexed_pages"));
        FileInputFormat.addInputPath(join, new
Path("/user/gates/tmp/filtered_users"));
        FileOutputFormat.setOutputPath(join, new
Path("/user/gates/tmp/joined"));
        join.setNumReduceTasks(50);
        Job joinJob = new Job(join);
        joinJob.addDependingJob(loadPages);
        joinJob.addDependingJob(loadUsers);

        JobConf group = new JobConf(MRExample.class);
        group.setJobName("Group URLs");
        group.setInputFormat(KeyValueTextInputFormat.class);
        group.setOutputKeyClass(Text.class);
        group.setOutputValueClass(LongWritable.class);
        group.setOutputFormat(SequenceFileOutputFormat.class);
        group.setMapperClass(LoadJoined.class);
        group.setCombinerClass(ReduceUrls.class);
        group.setReducerClass(ReduceUrls.class);
        FileInputFormat.addInputPath(group, new
Path("/user/gates/tmp/joined"));
        FileOutputFormat.setOutputPath(group, new
Path("/user/gates/tmp/grouped"));
        group.setNumReduceTasks(50);
        Job groupJob = new Job(group);
        groupJob.addDependingJob(joinJob);

        JobConf top100 = new JobConf(MRExample.class);
        top100.setJobName("Top 100 sites");
        top100.setInputFormat(SequenceFileInputFormat.class);
        top100.setOutputKeyClass(LongWritable.class);
        top100.setOutputValueClass(Text.class);
        top100.setOutputFormat(SequenceFileOutputFormat.class);
        top100.setMapperClass(LoadClicks.class);
        top100.setCombinerClass(LimitClicks.class);
        top100.setReducerClass(LimitClicks.class);
        FileInputFormat.addInputPath(top100, new
Path("/user/gates/tmp/grouped"));
        FileOutputFormat.setOutputPath(top100, new
Path("/user/gates/top100sitesforusers18to25"));
        top100.setNumReduceTasks(1);
        Job limit = new Job(top100);
        limit.addDependingJob(groupJob);

        JobControl jc = new JobControl("Find top 100 sites for users
18 to 25");
        jc.addJob(loadPages);
        jc.addJob(loadUsers);
        jc.addJob(joinJob);
        jc.addJob(groupJob);
        jc.addJob(limit);
        jc.run();
    }
}
```

170 lines of code, 4 hours to write

In Pig → 9 lines of code, 15 minutes to write