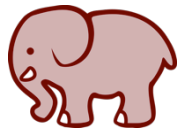
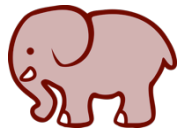
Apache Hadoop – A Developer's Course

Fundamentals of PIG



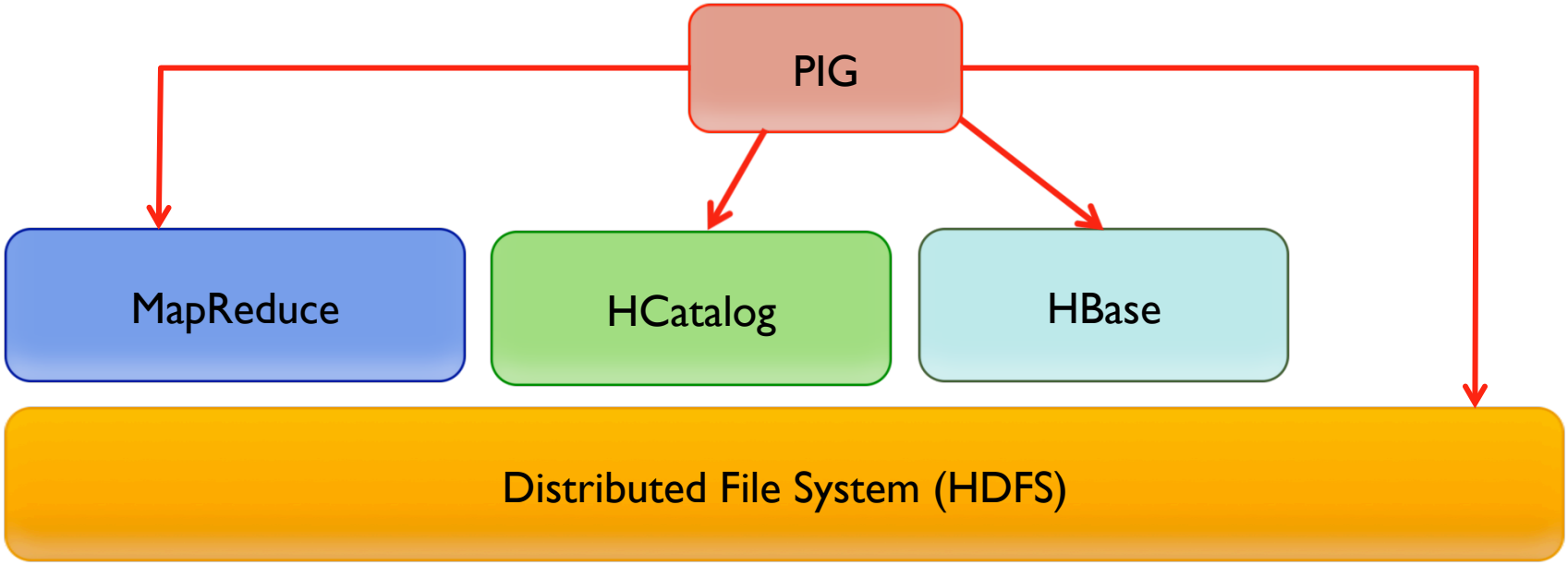
PIG – What is it?

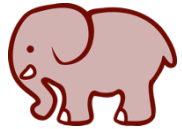
- Invented at Yahoo – to simplify data analysis on HDFS
- Used to implement MapReduce jobs with few lines of PIG Latin
  - Without use of a traditional (lengthy) MapReduce program
- Can query large datasets sitting on HDFS
- Has two main components
  - An SQL-like data processing language called Pig Latin
  - A compiler that compiles and runs Pig Latin scripts
- Mostly used at Yahoo – even now



PIG in the EcoSystem

- Runs on Hadoop utilizing both HDFS and MapReduce
- Reads and writes files from HDFS by default
- Stores intermediate data among MapReduce jobs





Running PIG

- A PIG Latin script executes in three modes:

1. **MapReduce**: the code executes as a MapReduce application on a Hadoop cluster (default)

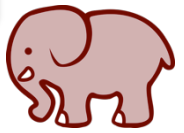
```
$ pig mylatinscript.pig
```

2. **Local**: the code executes locally in a single JVM using a local text file (development use case)

```
$ pig –x local mylatinscript.pig
```

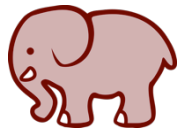
3. **Interactive**: PIG commands are entered manually at a command prompt known as Grunt shell

```
$ pig
grunt>
```



Understanding PIG Execution

- PIG Latin is a "data flow language"
- During execution each statement is processed by the PIG interpreter
- If a statement is valid, it gets added to a logical plan built by the interpreter
- The steps in the logical plan do not actually execute until a DUMP or STORE command is received



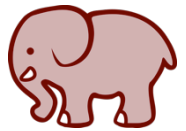
PIG Example

- Step 1: Build a logical plan
- Step 2: Build the logical plan into a physical plan
- Step 3: Running the physical plan into a MapReduce job

```
logevents = LOAD 'input/logdata.log' AS (date, level, code,
message);

severe = FILTER logevents BY (level == 'severe' AND code >=
500);

severegroup = GROUP severe BY code;

STORE severegroup INTO 'output/severeevents';
```

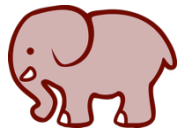
Logical Plan

Create and run physical plan



Apache Hadoop – A Developer's Course

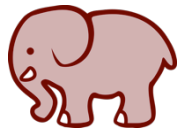
Grunt



Interactive PIG session

- PIG stores command history and allows editing within Grunt shell
- Command completion facilitated by TAB key
- Enter quit to exit shell

```
$ pig
grunt> F = LOAD 'input/logfile';
grunt> DUMP F;

(output of F appears here)
```



PIG command options

- Full listing of PIG commands

```
$ pig –h
grunt> help
```

- Execute a PIG script

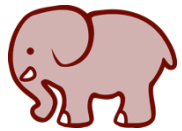
```
$ pig –e scriptname
```

- Specify a parameter

```
$ pig –p key1=value1 –p key2=value2
```

- List PIG properties to be used, if set by user

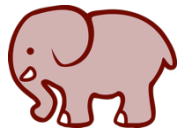
```
$ pig –h properties
```



Grunt HDFS commands

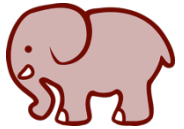
```
$ pig
grunt> fs –ls filename
grunt> fs –cat filename
grunt> fs –copyFromLocal file_local file_hdfs
grunt> fs –copyToLocal file_hdfs file_local_copy
grunt> fs –rm filename
grunt> fs –mkdir dirname
grunt> fs –mv fromDir/filename toDir/filename
grunt> quit
$
```

- Grunt acts as a shell for HDFS
- HDFS commands inside grunt> have a lot of resemblance with 'hadoop fs' commands



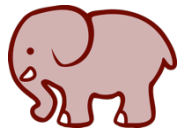
Lab : Get familiar with Pig commands

- Run Pig commands using Grunt shell



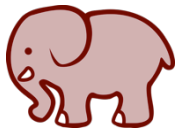
Apache Hadoop – A Developer's Course

Pig – Fundamentals



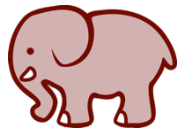
Pig Data Model

- 6 Scalar Types
  - int: An integer
  - long: A long number
  - float: A float number
  - double: A double number
  - chararray: A string or character array: \n, \t, \uxxxx all can be included
  - bytearray: A blob or array of bytes, represents `DataByteArray` or `byte[]`
- 3 Complex Types
  - Tuple: fixed length ordered collection of Pig data elements. Contains *fields*. Analogous to a row in SQL. Allows association of a schema for data validation.
    - e.g. (G, 78, John, 95432)
  - Bag: Unordered collection of tuples. Need not fit in memory. Can spill to local disk. Limited to size of local disk. Can become very big.
    - e.g. {(G, 78, John, 95432), (H, 65, Paul, 91302)}
  - Map: Collection of key-value pairs. key=chararray, value=any data element.
    - e.g. [name#Samantha, age#24]



Relations

- Pig Latin statements work with relations – similar to a table in a relational database where tuples in bag correspond to rows in a table
- Unlike rows in a table
  - Tuples in Pig relation do not have to contain same number of fields
  - Fields don't need to have the same data type
- Relations are like a bag of tuples



How to define a relation

- To load data from a file, use the LOAD command, e.g.

```
grunt> employee = LOAD 'employee.csv';
```

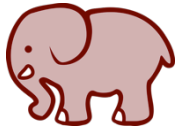
- Use the AS clause to define a schema for the relation, e.g.

```
grunt> employee = LOAD 'employee.csv' AS (name:
chararray, age: int, salary: double, city: chararray);
```

- Use the DESCRIBE command to view the schema of a relation, e.g.

```
grunt> DESCRIBE employee;
```

- A field's data type defaults to bytearray, if not specified



Using schema in a relation

- Suppose we have the following in a relation:

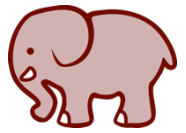
```
Bob, 21, 70500.00, San Antonio
Paul, 34, 81400.00, Austin
Erica, 47, 96200.00, Dallas
Samantha, 41, 68940.00, Houston
Ashley, 22, 67000.00, Dallas
```

- The data above represents name, age, salary and city – which can be loaded as follows:

```
grunt> employees = LOAD 'employees.csv' USING
PigStorage(',') AS (name: chararray, age: int, salary: double,
city: chararray);
```

- To see the schema use the DESCRIBE command as follows:

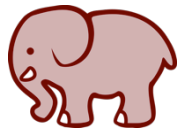
```
grunt> DESCRIBE employees;
employees: (name: chararray, age: int, salary: double, city:
chararray)
```



Relations using a schema

- Using a schema:
  - Allows you to refer to the values of a relation by the name of the field in the schema
  - For instance, FILTER command can refer to the field by name, e.g.

```
grunt> employees = LOAD 'employee.csv' USING
PigStorage(',') AS (name: chararray, age: int, salary: double,
city: chararray);
grunt> younghires = FILTER employees BY age <= 25;
```



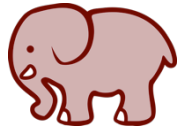
Relations without using a schema

- Not using a schema:
  - Pig loads the data anyway (because "pigs eat anything"), but schema is unknown, e.g.

```
grunt> employees = LOAD 'employees.csv' USING
PigStorage(',');
grunt> DESCRIBE employees;
Schema for employees is unknown.
```

  - A field is referenced by its position within relation
    - $0 is the first field, $1 is the second, and so on, e.g.

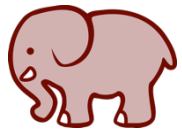
```
grunt> younghires = FILTER employees BY $1 <= 25;
grunt> DUMP younghires;
(Bob, 21, 70500.0, San Antonio)
(Ashley, 22, 67000.0, Dallas)
```



The FILTER operator

- Selects records to retain the data in the pipeline
- Use with keyword BY
- Contains a predicate evaluated to be true or false

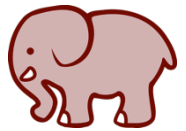
```
grunt> employees = LOAD 'employees.csv' USING
PigStorage(',') AS (name: chararray, age: int, salary: double,
city: chararray);
grunt> younghires = FILTER employees BY age <= 25;
grunt> DUMP younghires;
(Bob, 21, 70500.0, San Antonio)
(Ashley, 22, 67000.0, Dallas)
```



The DUMP command

- Used to display data on your screen
- Each record displayed as a tuple surrounded by ()
- Each bag is surrounded by { }
- Useful for
  - Debugging
  - Prototyping
  - Looking at results of quick ad-hoc jobs
- Use LIMIT to limit the number of tuples (rows) to display

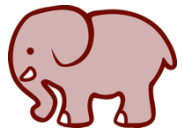
```
grunt> employees = LOAD 'employees.csv' USING
PigStorage(',') AS (name: chararray, age: int, salary: double,
city: chararray);
grunt> DUMP employees;
(Bob, 21, 70500.0, San Antonio)
(Paul, 34, 81400.0, Austin)
...
```



The ORDER BY command

- Sorts a relation based on one or more fields
- Must indicate the key or set of keys by which you wish to order your data

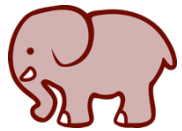
```
grunt> employees = LOAD 'employees.csv' USING
PigStorage(',') AS (name: chararray, age: int, salary: double,
city: chararray);
grunt> employees_by_city = ORDER employees BY city;
grunt> DUMP employees_by_city;
(Paul,34,81400.0, Austin)
(Erica,47,96200.0, Dallas)
(Ashley,22,67000.0, Dallas)
(Samantha,41,68940.0, Houston)
(Bob,21,70500.0, San Antonio)
```



The LIMIT operator

- Limits the number of output tuples
- Syntax is:
  - result = LIMIT alias n
  - where n is the number of tuples to show

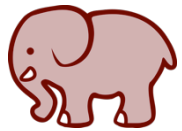
```
grunt> employees = LOAD 'employees.csv' USING
PigStorage(',') AS (name: chararray, age: int, salary: double,
city: chararray);
grunt> bycity = GROUP employees by city;
grunt> limited = LIMIT bycity 3;
grunt> DUMP limited;
( Austin,{(Paul,34,81400.0, Austin)})
( Dallas,{(Erica,47,96200.0, Dallas),(Ashley,22,67000.0,
Dallas)})
( Houston,{(Samantha,41,68940.0, Houston)})
```



Using PARALLEL

- PARALLEL allows you to specify the number of reducers
- Can be attached to any relational operator in Pig Latin
- It controls only reduce-side parallelism
- It works with the following operators:
  - GROUP, ORDER, DISTINCT, JOIN, LIMIT, COGROUP
- It can also be set system-wide by using
  - set default_parallel 10;

```
grunt> employees = LOAD 'employees.csv' USING
PigStorage(',') AS (name: chararray, age: int, salary: double,
city: chararray);
grunt> bycity = GROUP employees by city PARALLEL 10;
```



Parameter Substitution

- To specify information at runtime, one can pass parameters from command line
- Parameters are noted with a "$" in Pig Latin

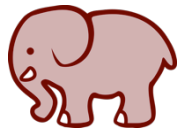
```
employees = LOAD '$input' AS (name: chararray, age: int,
salary: double, city: chararray);
limitedData = LIMIT data $size;
DUMP limitedData;
```

- When invoking this script, pass argument values as follows:

```
$ pig –param input=employees.csv –param size=4 myscript.pig
```

OR

```
$ pig
grunt> exec –param input=employees.csv –param size=4 myscript.pig
```



Dealing with Nulls in Pig

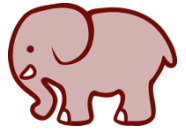
- Data elements may be null
  - Which is interpreted as "undefined"
- LOAD automatically inserts "null" for missing or invalid fields, e.g.

```
grunt> employees = LOAD 'employees.csv' USING
PigStorage(',') AS (name: chararray, age: int, salary: double,
city: chararray);
```

```
Bob, 21, 70500.00, San Antonio
Paul, , 81400.00, Austin
Erica, , 96200.00, Dallas
Samantha, forty, 68940.00, Houston
Ashley, 22, 67000.00, Dallas
```

is stored as

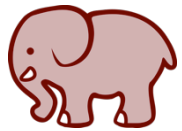
```
Bob, 21, 70500.0, San Antonio
Paul, null, 81400.0, Austin
Erica, 47, 96200.0, Dallas
Samantha, null, 68940.0, Houston
Ashley, 22, 67000.0, Dallas
```



The GROUP operator

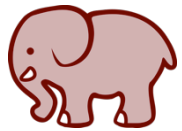
- The GROUP operator groups tuples together based on the specified way
- General usage is:
  - *alias2* = GROUP *alias1* BY *expression;* where
    - *alias1* = existing relation that you want to group
    - *alias2* = the new relation to be created after grouping
    - *expression* = a tuple expression that is the key you want to group by

```
grunt> employees = LOAD 'employee.csv' USING PigStorage(',')
AS (name: chararray, age: int, salary: double, city: chararray);
grunt> a = GROUP employees BY salary;
grunt> DESCRIBE a;
a: {group: double,employees: {(name: chararray,age: int,salary:
double,city: chararray)}}
```



The JOIN operator

- The JOIN operator performs an inner join on two or more relations based on common field values
- JOIN syntax is:
  - *alias2* = JOIN *alias1* by *expression1*, alias BY *expression2*, ...; where
    - alias1 = an existing relation
    - expression1 = a field of the relation
  - Result of JOIN is a flat set of tuples

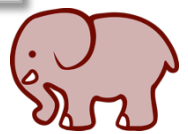


JOIN Example

- Add a new data file for each employee, with name and phone numbers.
- Join the two data files as follows:

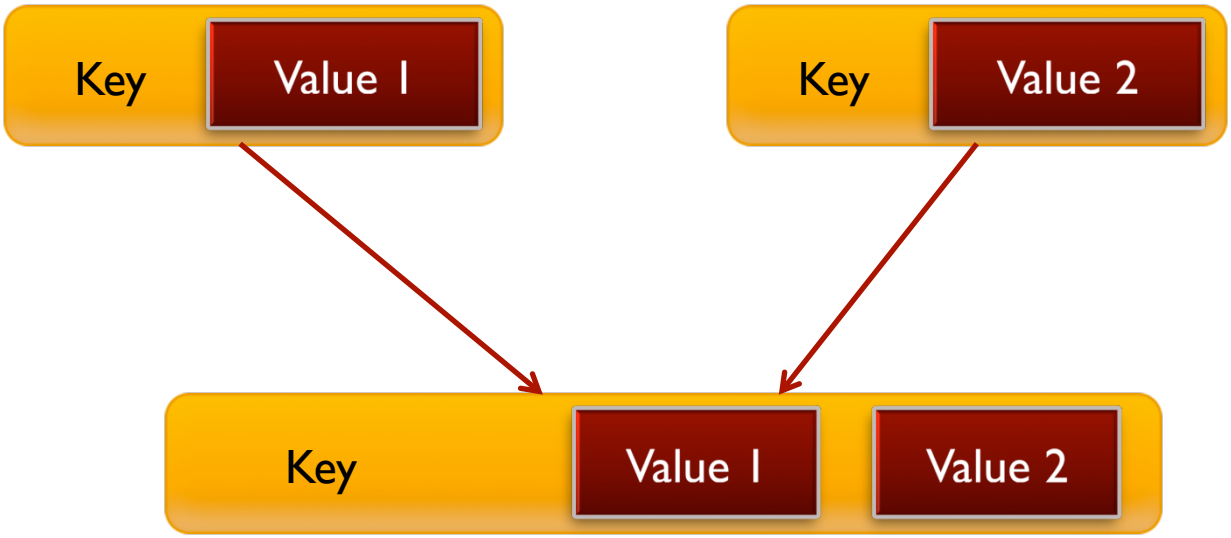
```
Bob, 210-443-3224
Paul, 512-732-9473
Erica, 214-789-4693
Samantha, 713-797-4793
Ashley, 214-549-8743
```

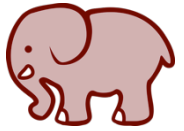
```
grunt> employees = LOAD 'employee.csv' USING PigStorage(',')
AS (name: chararray, age: int, salary: double, city: chararray);
grunt> phones = LOAD 'phones.csv' USING PigStorage(',') AS
(name: chararray, phone: chararray);
grunt> emp = JOIN employees BY name, phones by name;
grunt> DESCRIBE emp;
emp: emp: {employees::name: chararray,employees::age:
int,employees::salary: double,employees::city:
chararray,phones::name: chararray,phones::phone: chararray}
grunt> DUMP emp;
(Bob,21,70500.0, San Antonio,Bob, 210-443-3224)
(Paul,34,81400.0, Austin,Paul, 512-732-9473)
(Erica,47,96200.0, Dallas,Erica, 214-789-4693)
(Ashley,22,67000.0, Dallas,Ashley, 214-549-8743)
(Samantha,41,68940.0, Houston,Samantha, 713-797-4793)
```



The COGROUP Operator

- Collects records based on a key
- Pig brings together bags associated with a key
- Requires a reduce phase because it collects records with like keys together (partitioner's hash() imitates group by)
- Results in a record with a key and a bag for each input

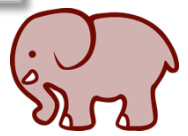




COGROUP example

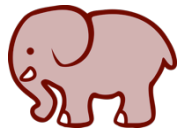
- Example to load two datasets and form a COGROUP by name

```
grunt> employees = LOAD 'employee.csv' USING PigStorage(',')
AS (name: chararray, age: int, salary: double, city: chararray);
grunt> phones = LOAD 'phones.csv' USING PigStorage(',') AS
(name: chararray, phone: chararray);
grunt> emp = COGROUP employees BY name, phones by name;
grunt> DESCRIBE emp;
emp: {group: chararray,employees: {(name: chararray,age:
int,salary: double,city: chararray)},phones: {(name:
chararray,phone: chararray)}}
grunt> DUMP emp;
(Bob,{(Bob,21,70500.0, San Antonio)},{(Bob, 210-443-3224)})
(Paul,{(Paul,34,81400.0, Austin)},{(Paul, 512-732-9473)})
(Erica,{(Erica,47,96200.0, Dallas)},{(Erica, 214-789-4693)})
(Ashley,{(Ashley,22,67000.0, Dallas)},{(Ashley,
214-549-8743)})
(Samantha,{(Samantha,41,68940.0, Houston)},{(Samantha,
713-797-4793)})
```



The FOREACH operator

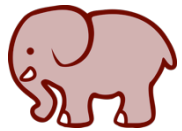
- The FOREACH operator transforms data into a new relation based on the columns of the data
- FOREACH syntax is:
  - *alias2* = FOREACH *alias1* GENERATE *expression*
    - *alias1* = an existing relation
    - *expression* = an expression that determines the output



FOREACH example

- FOREACH creates a bag of tuples, e.g.

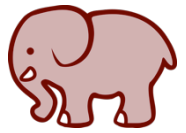
```
grunt> employees = LOAD 'employee.csv' USING PigStorage(',')
AS (name: chararray, age: int, salary: double, city: chararray);
grunt> salaries = FOREACH employees GENERATE salary, city;
grunt> DESCRIBE salaries;
grunt> DUMP salaries;
(70500.0, San Antonio)
(81400.0, Austin)
(96200.0, Dallas)
(68940.0, Houston)
(67000.0, Dallas)
```



FOREACH with grouping example

- FOREACH can be combined with grouping to compute aggregates
- Example below finds the total salary for each city

```
grunt> employees = LOAD 'employee.csv' USING PigStorage(',')
AS (name: chararray, age: int, salary: double, city: chararray);
grunt> salaries = FOREACH employees GENERATE salary, city;
grunt> bycity = GROUP salaries BY city;
grunt> salarysum = FOREACH bycity GENERATE group,
SUM(salaries.salary);
grunt> DESCRIBE salarysum;
salarysum: {group: chararray,double}
grunt> DUMP salarysum;
( Austin,81400.0)
( Dallas,163200.0)
( Houston,68940.0)
( San Antonio,70500.0)
```

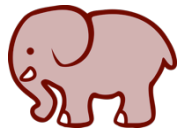


FLATTEN Operator

- Successive operations often produce bags of bags
- FLATTEN removes one level of nesting

```
grunt> employeeskills = LOAD 'skills.csv' USING PigStorage(',')
AS (name: chararray, experience: int, skills:bag{ t:(p:
chararray)});
grunt> skills = FOREACH employeeskills GENERATE name,
FLATTEN(skills) AS skillset;
grunt> byskills = GROUP skills BY skillset;
grunt> DESCRIBE byskills;
grunt> DUMP byskills;
```

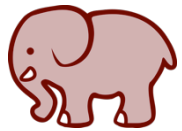
EXAMPLE IS INCORRECT !!!!



The SAMPLE operator

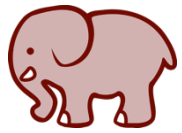
- It is a simple way to sample your data
- Reads through all data and returns a percentage of the rows
- Expressed as a double value between 0 and 1
  - e.g. 0.3 means 30%

```
grunt> employees = LOAD 'employee.csv' USING PigStorage(',')
AS (name: chararray, age: int, salary: double, city: chararray);
grunt> sampleset = SAMPLE employees 0.3;
grunt> DUMP sampleset;
(Bob,21,70500.0, San Antonio)
(Erica,47,96200.0, Dallas)
```



Debugging Pig scripts

- Use EXPLAIN, ILLUSTRATE and DESCRIBE to debug
- First use local cluster to test your scripts before running it in the cluster
  - No waiting for a slot
  - Logs appear on your screen rather than on a remote task node
  - A debugger can be attached to the process since it is local
  - May be slower – does not matter for debugging

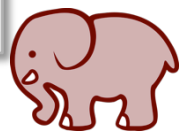


Using EXPLAIN to debug

- EXPLAIN shows how Pig compiles a script into MapReduce jobs
- It provides Logical and Physical execution details
  - It produces a logical plan
  - Show logical operators that Pig will use to execute the script
  - Flow of the chart displayed is bottom-to-top

```
grunt> employees = LOAD 'employee.csv' USING PigStorage(',') AS
(name: chararray, age: int, salary: double, city: chararray);
grunt> salaries = FOREACH employees GENERATE salary, city;
grunt> bycity = GROUP salaries BY city;
grunt> EXPLAIN bycity;
```

```
#-----------------------------------------------
# New Logical Plan:
#-----------------------------------------------
bycity: (Name: LOStore Schema: group#14:chararray,salaries#19:bag{#26:tuple(salary#13:double,city#14:chararray)})ColumnPrune:InputUids=[19, 14]ColumnPrune:OutputUids=[19, 14]
|
|---bycity: (Name: LOCogroup Schema: group#14:chararray,salaries#19:bag{#26:tuple(salary#13:double,city#14:chararray)})ColumnPrune:InputUids=[13, 14]ColumnPrune:OutputUids=[19, 14]
    |   |
    |   city:(Name: Project Type: chararray Uid: 14 Input: 0 Column: 1)
    |
    |---salaries: (Name: LOForEach Schema: salary#13:double,city#14:chararray)
        |   |
        |   (Name: LOGenerate[false,false] Schema: salary#13:double,city#14:chararray)
        |   |   |
        |   |   (Name: Cast Type: double Uid: 13)
        |   |   |
        |   |   |---salary:(Name: Project Type: bytearray Uid: 13 Input: 0 Column: (*))
        |   |   |
        |   |   (Name: Cast Type: chararray Uid: 14)
        |   |   |
        |   |   |---city:(Name: Project Type: bytearray Uid: 14 Input: 1 Column: (*))
        |   |
        |   |---(Name: LOInnerLoad[0] Schema: salary#13:bytearray)
        |   |
        |   |---(Name: LOInnerLoad[1] Schema: city#14:bytearray)
        |
        |---employees: (Name: LOLoad Schema: salary#13:bytearray,city#14:bytearray)ColumnPrune:RequiredColumns=[2, 3]ColumnPrune:InputUids=[13, 14]ColumnPrune:OutputUids=[13,
14]RequiredFields:[2, 3]
```

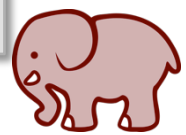


Using ILLUSTRATE to debug

- ILLUSTRATE takes a sample of your data and and runs it through a script
  - It ensures that for every operator there is data which will pass through it some which will not
  - If needed, it will manufacture data that looks like the data entered and changes what it needs
- It displays the step-by-step transformations of the data
- Only works when a schema is defined for a relation

```
grunt> ILLUSTRATE bycity;
```

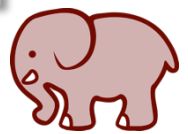
```
-------------------------------------------------------------------------------------
| employees    | name:chararray   | age:int   | salary:double    | city:chararray   |
-------------------------------------------------------------------------------------
|            | Bob              | 21        | 70500.0          | San Antonio      |
|            | Bob              | 21        | 70500.0          | San Antonio      |
-------------------------------------------------------------------------------------
-------------------------------------------------
| salaries    | salary:double    | city:chararray   |
-------------------------------------------------
|           | 70500.0          | San Antonio      |
|           | 70500.0          | San Antonio      |
-------------------------------------------------
-------------------------------------------------------------------------------------------
| bycity    | group:chararray   | salaries:bag{:tuple(salary:double,city:chararray)}          |
-------------------------------------------------------------------------------------------
|         | San Antonio       | {(70500.0, San Antonio), (70500.0, San Antonio)}          |
-------------------------------------------------------------------------------------------
```



User defined functions (UDF)

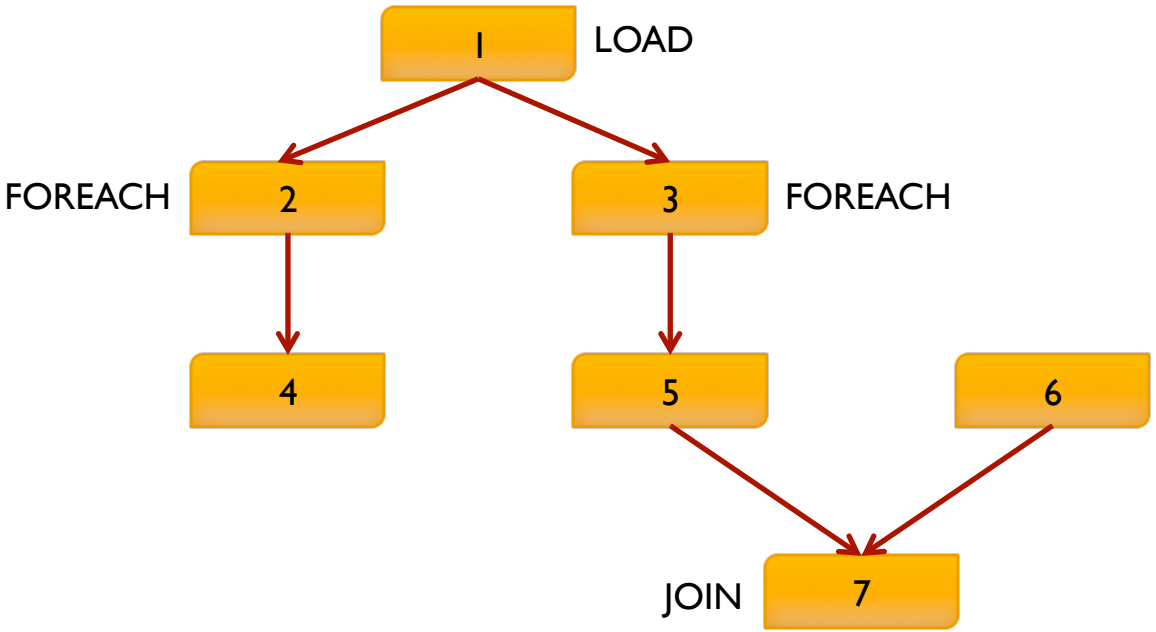
- Users can combine Pig operators with their own User Defined Functions
- There exists a collection of user-contributed UDFs in a Piggybank
  - Packaged and released with Pig http://pig.apache.org/docs/r0.8.1/api/
- UDFs are written in Java and implemented as Java classes in jar files
  - Simply let Pig know about your Jar file

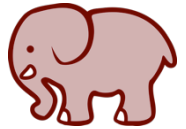
```
grunt> REGISTER '/usr/lib/pig/piggybank.jar'
grunt> employees = LOAD 'employee.csv' USING PigStorage(',')
AS (name: chararray, age: int, salary: double, city: chararray);
grunt> backwards = FOREACH employees GENERATE
org.apache.pig.piggybank.evaluation.string.Reverse(city);
(oinotnA naS )
(nitsuA )
(sallaD )
(notsuoH )
(sallaD )
```



Structured Processing Flow in Pig Latin

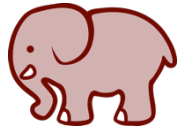
- Pig Latin describes a Directed Acyclic Graph (DAG) where the edges are data-flows and the nodes are operators





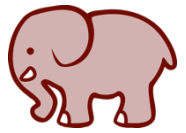
Lab : Pig Data Operations

- Get familiar with relations, schemas, JOIN, FILTER, EXPLAIN and ILLUSTRATE commands



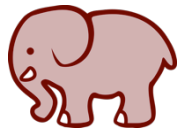
Apache Hadoop – A Developer's Course

Pig Latin



Apache Hadoop – A Developer's Course

Introduction to Hive

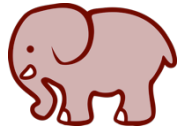


Hive Introduction

- Hive is a data-warehousing layer built on top of Apache Hadoop

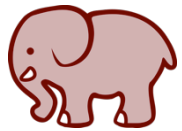
- Hive allows you to define a structure for your unstructured big data, simplifying the process of performing analysis and queries by introducing a familiar SQL-like language called HiveQL

- Hive is for data analysts who are familiar with SQL and need to do ad-hoc queries, summarization and data analysis on their HDFS data



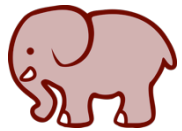
What Hive is not

- It is not relational database
- Hive uses a database to store metadata, but the data that Hive processes is stored in HDFS
- Hive is not designed for online transaction processing and does not offer real-time queries and row-level updates



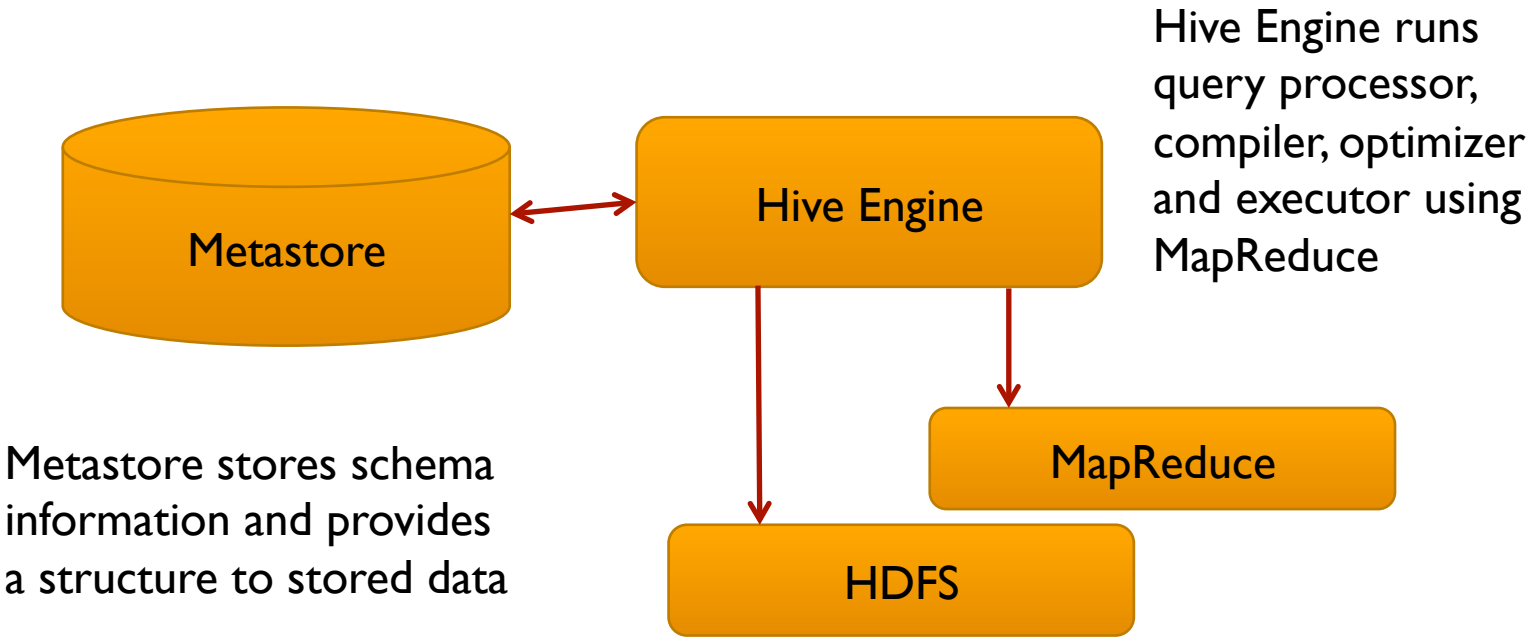
Pig vs. Hive

- When to use Hive
  - when you want to query the data
  - when you need answers to specific questions
  - if you are familiar with SQL
- When to use Pig
  - for ETL (Extract Transform Load) operations
  - pre-processing your data for easier analysis
  - when you have a long series of steps to perform in the analysis
- Hive and Pig both work well together – and many businesses do it



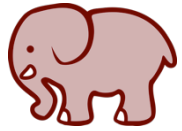
Hive Basics

- Data warehousing package built on top of Hadoop
- System for querying and manipulating structured data which
  - Uses MapReduce for execution
  - Uses HDFS (or Hbase) for storage



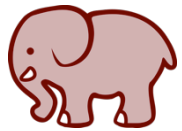
Hive Engine runs query processor, compiler, optimizer and executor using MapReduce

Metastore stores schema information and provides a structure to stored data



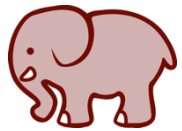
What is a Hive Table?

- Hive table consists of
  - Data: typically a file or a group of files in HDFS
  - Schema: in the form of metadata stored in a relational database
- Schema and data are separate
  - A schema can be defined for existing data
  - Data can be added or removed independently
  - Hive can be "pointed" at existing data
- You have to define a schema if you have existing data in HDFS what you want to use in Hive

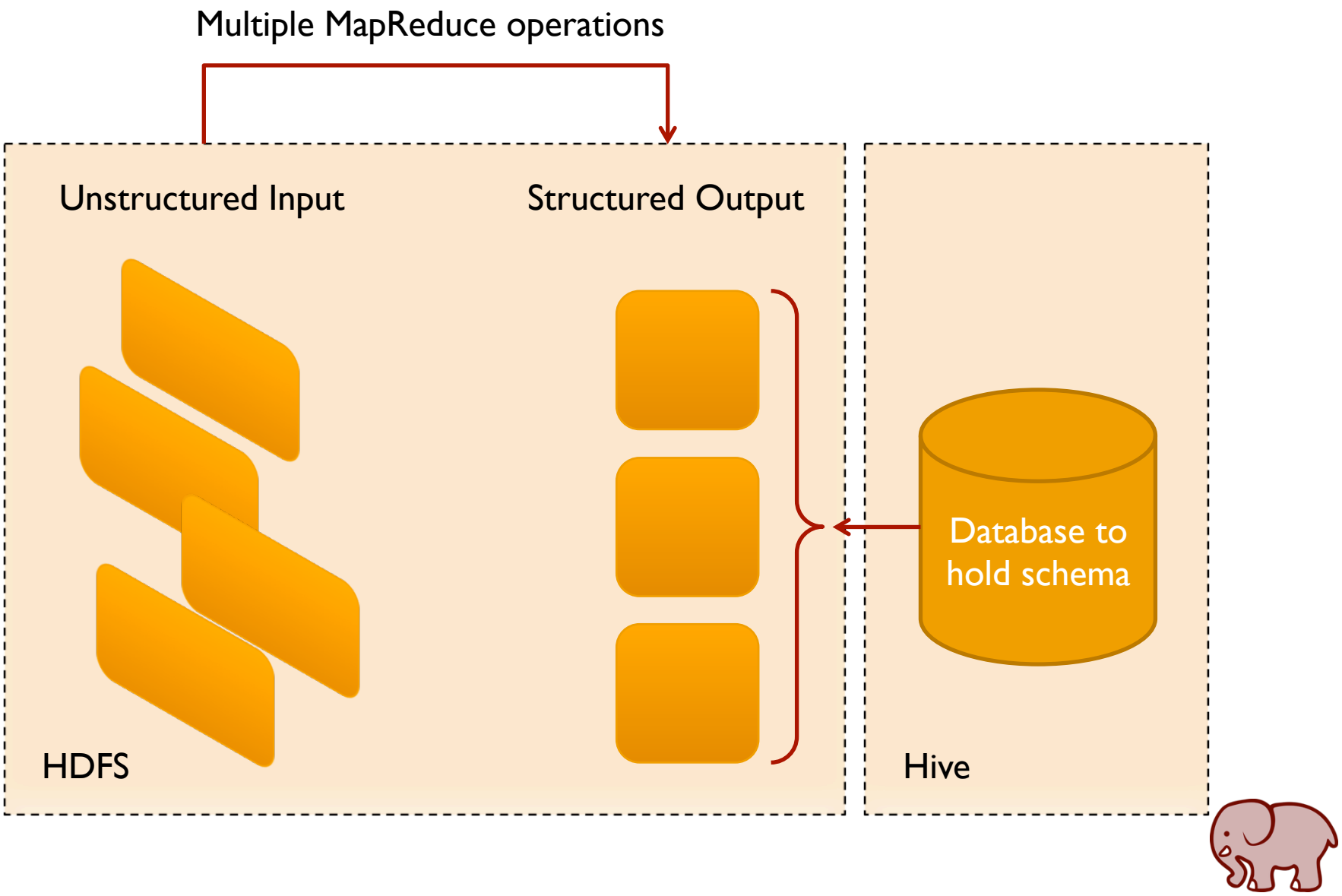


HiveQL

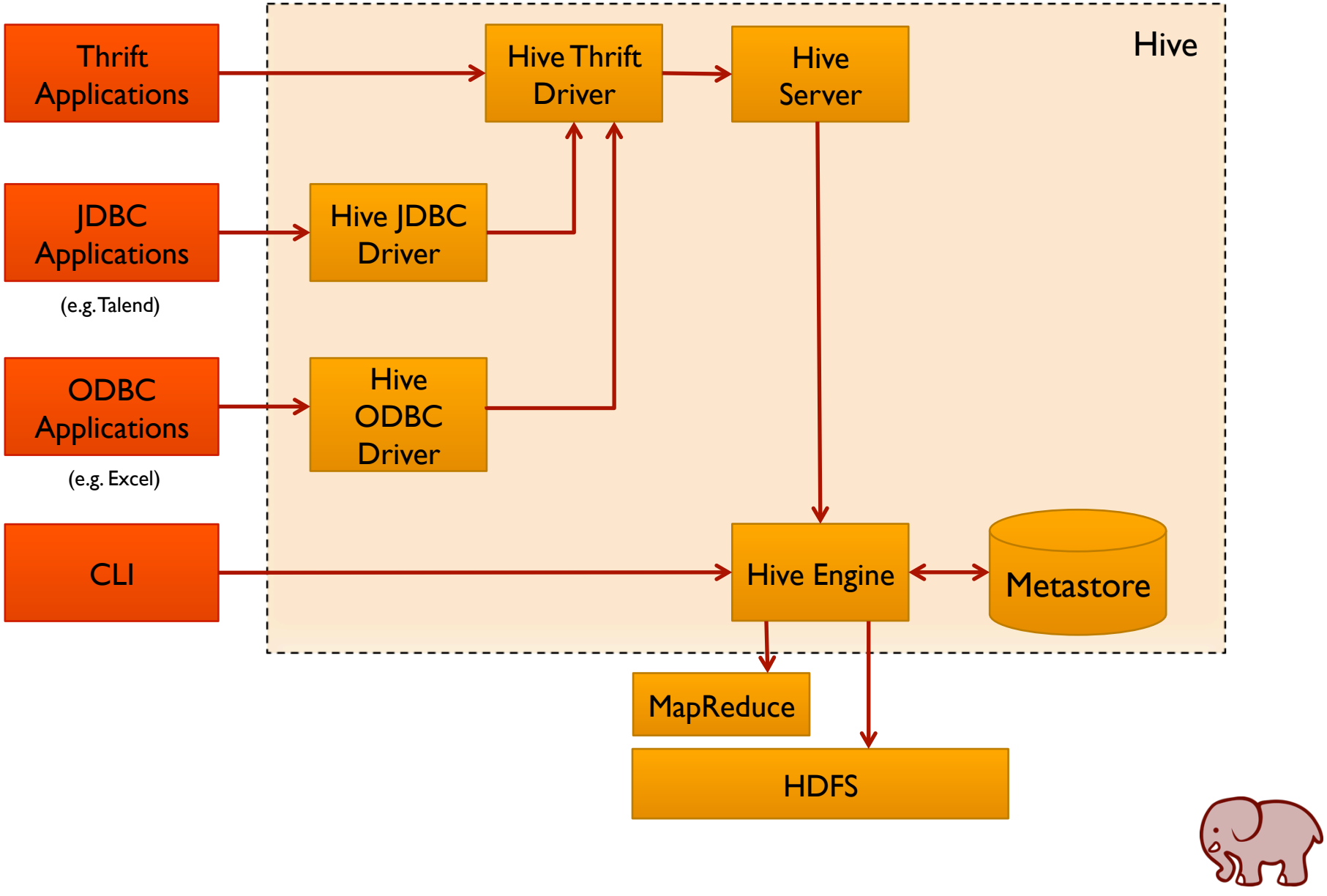
- Hive's SQL-like language is called HiveQL
- Is uses familiar relational database concepts such as tables, rows, columns and schema
- Designed to work with structured data
- Converts SQL queries to MapReduce jobs
- Supports uses such as:
  - Ad-hoc queries
  - Summarization
  - Data Analysis



Typical Hive use-case

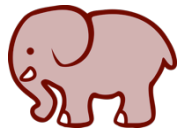


Hive Architecture



Apache Hadoop – A Developer's Course

Hive Shell and HiveQL



Running jobs with Hive Shell

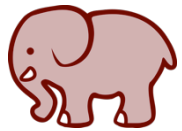
- Hive shell is the primary way to interact with Hive

```
$ hive
hive>
```

- Hive can also be invoked in batch mode

```
$ hive -f myhivescript.q
```

  - Use -S to only display results



Common Hive examples

- At terminal enter

```
$ hive
```

- List all properties and values

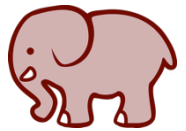
```
hive> set -v
```

- List and describe tables

```
hive> show tables ;
hive> describe <tablename> ;
hive> describe extended <tablename> ;
```

- List and describe functions

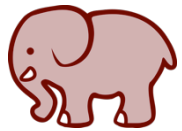
```
hive> show functions ;
hive> describe function <functionname> ;
```



Querying data in Hive

- Selecting data

```
hive> SELECT * FROM employees;
hive> SELECT * FROM employees WHERE salary > 40000 SORT
BY city;
```



HiveQL

- HiveQL is similar to other SQLs
- User does not need to know MapReduce
- HiveQL is based on SQL-92 specification
- Supports multi-table inserts via your code

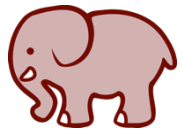


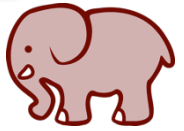
Table Operations

- Defining a table

```
hive> CREATE TABLE employees
(name STRING, age int, salary double, city STRING)
ROW FORMAT DELIMITED
FIELDS TERMINATED BY ','
STORED AS TEXTFILE;
```

- ROW FORMAT is a Hive-unique command which indicates that each row is comma-delimited text
- HiveQL statements are terminated by a semi-colon ';'
- Other table operations

```
SHOW TABLES
CREATE TABLE
ALTER TABLE
DROP TABLE
```



Managing Tables

- Seeing current tables

```
hive> SHOW TABLES;
```

- Checking the schema

```
hive> DESCRIBE employees;
```

- Changing the table name

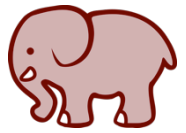
```
hive> ALTER TABLE employees RENAME to employeedata;
```

- Adding a column

```
hive> ALTER TABLE employees ADD COLUMNS (zip STRING);
```

- Dropping a partition

```
hive> ALTER TABLE employees DROP PARTITION (age > 50);
```

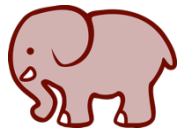


Loading data into Hive

- Use LOAD DATA to import data into a Hive table
- To load data indicate the path of the data

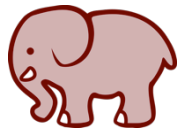
```
hive> LOAD DATA LOCAL INPATH 'input/employee.csv' INTO
TABLE employees;
```

- Files are loaded as-is without modification
- Default location is /user/hive/warehouse
- Hive can read all files in a directory
- Use keyword OVERWRITE to write over a file of same name
- Schema is checked when data is queried
  - If row does not match schema, data is read as NULL



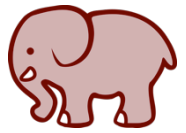
Apache Hadoop – A Developer's Course

Hive Operators and Functions



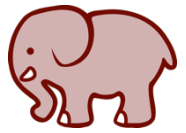
Apache Hadoop – A Developer's Course

HCatalog



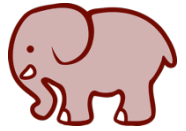
Apache Hadoop – A Developer's Course

HBase



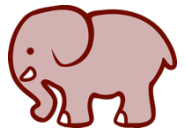
Apache Hadoop – A Developer's Course

HBase Architecture and API



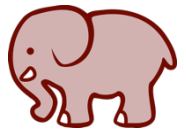
Apache Hadoop – A Developer's Course

Impala



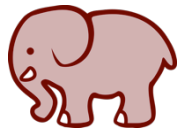
Apache Hadoop – A Developer's Course

Hadoop Web Services



HDFS Command Examples

```
$ hadoop fs -ls /user/john
$ hadoop fs -lsr
$ hadoop fs -mkdir input
$ hadoop fs -put ~/training/commands.txt notes
$ hadoop fs -chmod 777 notes/command.txt
$ hadoop fs -cat notes/commands.txt | more
$ hadoop fs -rm notes/*.txt
```



Uploading and Retrieving files on HDFS

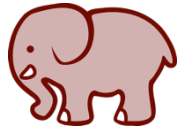
- Uploading

```
$ hadoop fs –put filenameSource filenameDestination
$ hadoop fs –put filename directoryName/fileName
```

```
$ hadoop fs –put foo bar
$ hadoop fs –put foo directoryName/fileName
$ hadoop fs –lsr directoryName
```

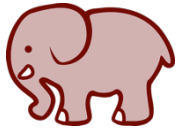
- Retrieving

```
$ hadoop fs –cat foo
$ hadoop fs –get foo LocalFoo
$ hadoop fs –copyToLocal foo LocalFoo
$ hadoop fs –rmr directory/file
```

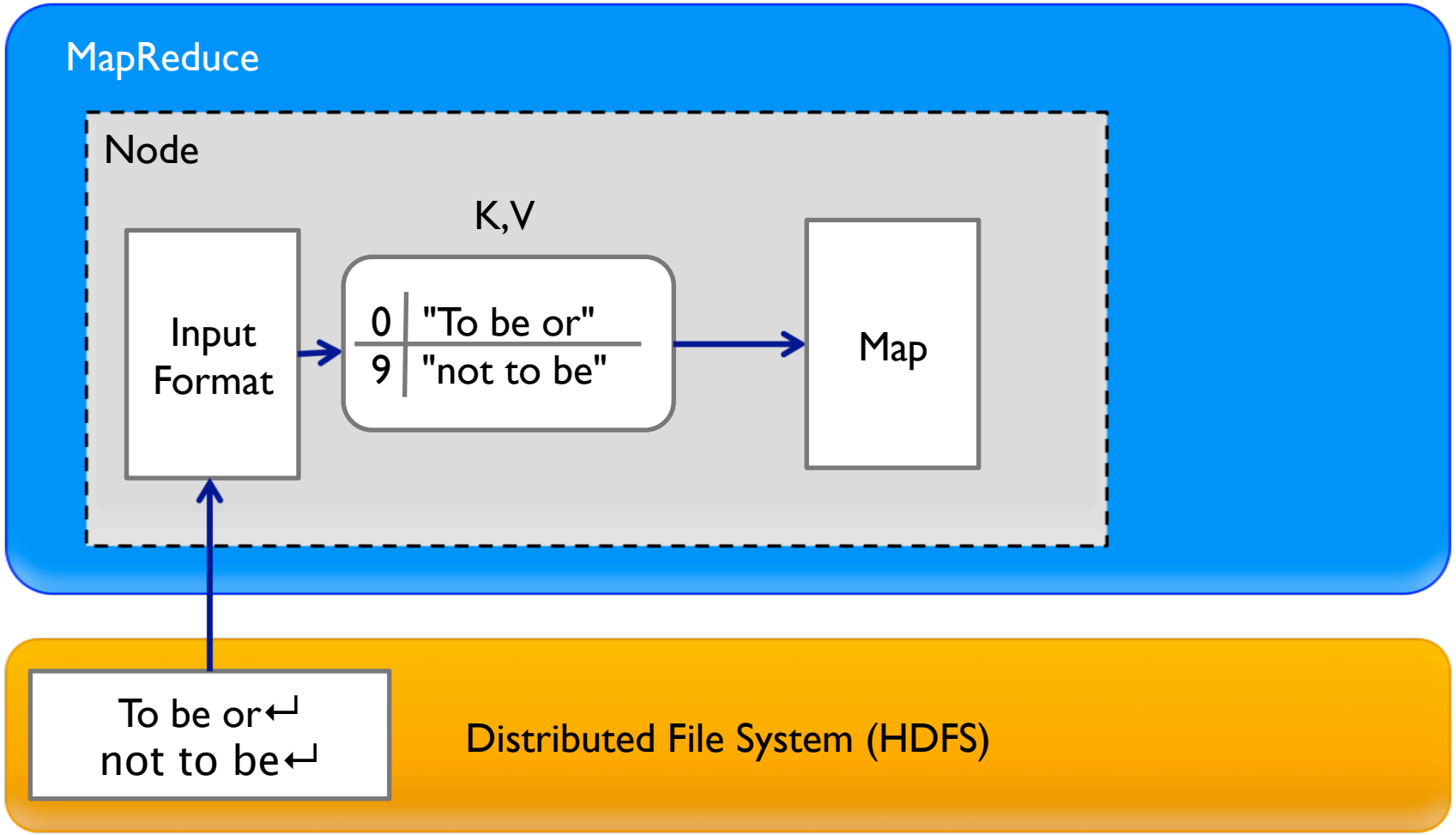


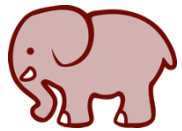
Lab 2: Get familiar with HDFS commands

- Practice basic HDFS file-system commands

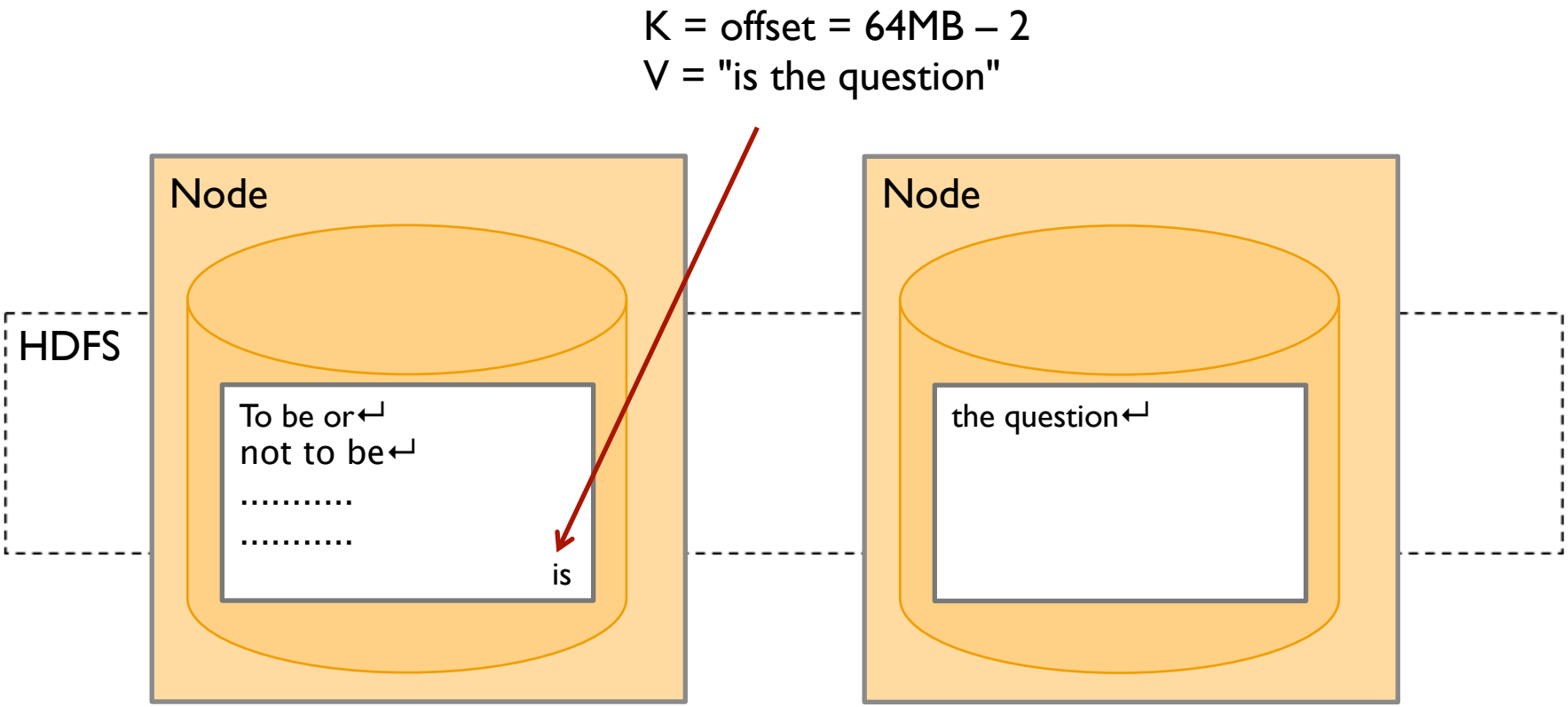


TextInputFormat

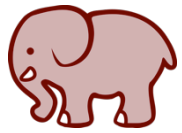




TextInputFormat

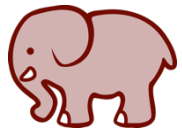


"Value" of a "key" can span over multiple blocks



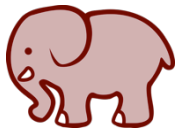
FileInputFormat

- FileInputFormat used for large files
  - Larger than HDFS block
- Split size
  - Controlled by Hadoop properties
  - Can be overridden by Application
- Hadoop is optimized by
  - decreasing the number of files and
  - making size of files larger

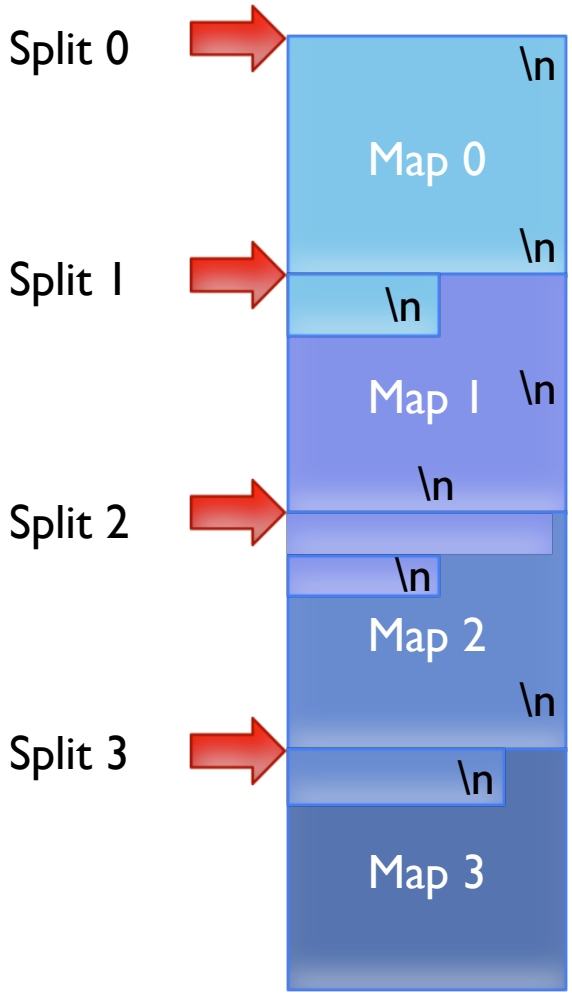


InputSplits and HDFS

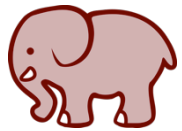
- Split is a portion of the data to be processed by a single map process in parallel
- Split = single block of data
  - records may straddle blocks
  - tasks often need to access a small portion of a record from a block on another node
  - Use larger blocks for a split to:
    - Increase performance due to more data for a mapper
    - A mapper has setup/teardown time that is constant
    - A small split means less work for the constant setup time
- `JobClient` computes input splits



InputSplit



- Usually crosses boundaries
  - of Mappers
  - of HDFS blocks

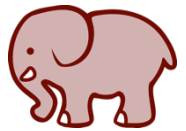


RecordReader

- The RecordReader breaks the data into key/value pairs for input to the Mapper

```
public interface RecordReader<K, V> {
  /* Reads the next key/value pair from the input for processing. */
  boolean next(K key, V value) throws IOException;

  /* Create an object of the appropriate type to be used as a key. */
  K createKey();

  /* Create an object of the appropriate type to be used as a value. */
  V createValue();

  /* Returns the current position in the input. */
  long getPos() throws IOException;

  /* Close this {@link InputSplit} to future operations. */
  public void close() throws IOException;

  /* How much of the input has the {@link RecordReader} consumed i.e.
     has been processed by? */
  float getProgress() throws IOException;
}
```

API docs: http://hadoop.apache.org/docs/r1.2.0/api/org/apache/hadoop/mapred/RecordReader.html



RecordReader – split vs. record