Apache Pig

1.What is Apache Pig

- Apache Pig is a high-level data flow platform for executing MapReduce programs of Hadoop.
- The language used for Pig is Pig Latin.
- The Pig scripts get internally converted to Map Reduce jobs and get executed on data stored in HDFS.
- Apart from that, Pig can also execute its job in Apache Tez or Apache Spark.
- Pig can handle any type of data, i.e., structured, semi-structured or unstructured and stores the corresponding results into Hadoop Data File System.
- Every task which can be achieved using PIG can also be achieved using java used in MapReduce.

2. Features of Apache Pig

1) Ease of programming

Writing complex java programs for map reduce is quite tough for non-programmers. Pig makes this process easy. In the Pig, the queries are converted to MapReduce internally.

2) Optimization opportunities

It is how tasks are encoded permits the system to optimize their execution automatically, allowing the user to focus on semantics rather than efficiency.

3) Extensibility

A user-defined function is written in which the user can write their logic to execute over the data set.

4) Flexible

It can easily handle structured as well as unstructured data.

5) In-built operators

It contains various type of operators such as sort, filter and joins.

3. Advantages of Apache Pig

- Less code - The Pig consumes less line of code to perform any operation.
- Reusability - The Pig code is flexible enough to reuse again.
- Nested data types - The Pig provides a useful concept of nested data types like tuple, bag, and map.

4. Pig Latin

The Pig Latin is a data flow language used by Apache Pig to analyze the data in Hadoop. It is a textual language that abstracts the programming from the Java MapReduce idiom into a notation.

5. Structure

- A Pig Latin program consists of a collection of statements.
- A statement can be thought of as an operation or a command.

  For example, a GROUP operation is a type of statement:

  grouped_records = **GROUP** records **BY** year;
- The command to list the files in a Hadoop filesystem is another example of a statement:

  ls /
- Statements are usually terminated with a semicolon
- The ls command, on the other hand, does not have to be terminated with a semicolon
- Grunt do not need the terminating semicolon
- Statements that have to be terminated with a semicolon can be split across multiple lines for readability:

  records = **LOAD** 'input/ncdc/micro-tab/sample.txt'
  AS (year:**chararray**, temperature:**int**, quality:**int**);
- Pig Latin has two forms of comments.
- Double hyphens are used for single-line comments.

  -- *My program*
  DUMP A; -- *What's in A?*
- C-style comments are more flexible since they delimit the beginning and end of the comment block with /* and */ markers. They can span lines or be embedded in a single line:

  /*
  * *Description of my program spanning*
  * *multiple lines.*
  */
  A = **LOAD** 'input/pig/join/A';
  B = **LOAD** 'input/pig/join/B';

```
C = JOIN A BY $0, /* ignored */ B BY $1;
DUMP C;
```

6. Statements

- 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 again the Pig Latin program from the first example:

```
-- 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;
```

6.1 Multiquery Execution

- In interactive mode, STORE acts like DUMP and will always trigger execution (this includes the run command), but in batch mode it will not (this includes the exec command).
- The reason for this is efficiency. In batch mode, Pig will parse the whole script to see whether there are any optimizations that could be made to limit the amount of data to be written to or read from disk.

Consider the following simple example:

```
A = LOAD 'input/pig/multiquery/A';
B = FILTER A BY $1 == 'banana';
```

C = **FILTER** A **BY** $1 != 'banana';

STORE B **INTO** 'output/b';

STORE **C** INTO **'output/c';**

6.2 Pig Latin relational operators

Category	Operator	Description
Loading and storing	LOAD	Loads data from the filesystem or other storage into a relation
	STORE	Saves a relation to the filesystem or other storage
	DUMP (\d)	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 to or from a relation
	MAPREDUCE	Runs a MapReduce job using a relation as input
	STREAM	Transforms a relation using an external program
	SAMPLE	Selects a random sample of a relation
	ASSERT	Ensures a condition is true for all rows in a relation; otherwise, fails
Grouping and joining	JOIN	Joins two or more relations
	COGROUP	Groups the data in two or more relations
	GROUP	Groups the data in a single relation
	CROSS	Creates the cross product of two or more relations
	CUBE	Creates aggregations for all combinations of specified columns in a relation
Sorting	ORDER	Sorts a relation by one or more fields
	RANK	Assign a rank to each tuple in a relation, optionally sorting by fields first
	LIMIT	Limits the size of a relation to a maximum number of tuples
Combining and splitting	UNION	Combines two or more relations into one
	SPLIT	Splits a relation into two or more relations

6.3 Pig Latin diagnostic operators

Operator (Shortcut)	Description
DESCRIBE (\de)	Prints a relation's schema
EXPLAIN (\e)	Prints the logical and physical plans
ILLUSTRATE (\i)	Shows a sample execution of the logical plan, using a generated subset of the input

6.4 Pig Latin macro and UDF statements

Statement	Description
REGISTER	Registers a JAR file with the Pig runtime
DEFINE	Creates an alias for a macro, UDF, streaming script, or command specification
IMPORT	Imports macros defined in a separate file into a script

6.5 Pig Latin commands

Category	Command	Description
Hadoop filesystem	cat	Prints the contents of one or more files
	cd	Changes the current directory
	copyFromLocal	Copies a local file or directory to a Hadoop filesystem
	copyToLocal	Copies a file or directory on a Hadoop filesystem to the local filesystem
	cp	Copies a file or directory to another directory
	fs	Accesses Hadoop's filesystem shell
	ls	Lists files
	mkdir	Creates a new directory
	mv	Moves a file or directory to another directory
	pwd	Prints the path of the current working directory
	rm	Deletes a file or directory
	rmf	Forcibly deletes a file or directory (does not fail if the file or directory does not exist)
Hadoop MapReduce	kill	Kills a MapReduce job

Category	Command	Description
Utility	clear	Clears the screen in Grunt
	exec	Runs a script in a new Grunt shell in batch mode
	help	Shows the available commands and options
	history	Prints the query statements run in the current Grunt session
	quit (\q)	Exits the interpreter
	run	Runs a script within the existing Grunt shell
	set	Sets Pig options and MapReduce job properties
	sh	Runs a shell command from within Grunt

7. Expressions

An expression is something that is evaluated to yield a value. Expressions can be used in Pig as a part of a statement containing a relational operator

Category	Expressions	Description	Examples
Constant	Literal	Constant value (see also the "Literal example" column in Table 16-6)	`1.0`, `'a'`
Field (by position)	`$n`	Field in position *n* (zero-based)	`$0`
Field (by name)	*f*	Field named *f*	`year`
Field (disambiguate)	*r*`::`*f*	Field named *f* from relation *r* after grouping or joining	`A::year`
Projection	*c*`.$`*n*, *c*`.`*f*	Field in container *c* (relation, bag, or tuple) by position, by name	`records.$0`, `records.year`
Map lookup	*m*`#`*k*	Value associated with key *k* in map *m*	`items#'Coat'`
Cast	`(`*t*`)` *f*	Cast of field *f* to type *t*	`(int) year`
Arithmetic	*x* `+` *y*, *x* `-` *y*	Addition, subtraction	`$1 + $2`, `$1 - $2`
	x `*` *y*, *x* `/` *y*	Multiplication, division	`$1 * $2`, `$1 / $2`
	x `%` *y*	Modulo, the remainder of *x* divided by *y*	`$1 % $2`
	`+`*x*, `-`*x*	Unary positive, negation	`+1`, `-1`
Conditional	*x* `?` *y* `:` *z*	Bincond/ternary; *y* if *x* evaluates to true, *z* otherwise	`quality == 0 ? 0 : 1`
	`CASE`	Multi-case conditional	`CASE q WHEN 0 THEN 'good' ELSE 'bad' END`

Category	Expressions	Description	Examples
Comparison	`x == y, x != y`	Equals, does not equal	`quality == 0, tempera ture != 9999`
	`x > y, x < y`	Greater than, less than	`quality > 0, quality < 10`
	`x >= y, x <= y`	Greater than or equal to, less than or equal to	`quality >= 1, quality <= 9`
	`x matches y`	Pattern matching with regular expression	`quality matches '[01459]'`
	`x is null`	Is null	`temperature is null`
	`x is not null`	Is not null	`temperature is not null`
Boolean	`x OR y`	Logical OR	`q == 0 OR q == 1`
	`x AND y`	Logical AND	`q == 0 AND r == 0`
	`NOT x`	Logical negation	`NOT q matches '[01459]'`
	`IN x`	Set membership	`q IN (0, 1, 4, 5, 9)`
Functional	*fn*`(`*f1*`,`*f2*`,...)`	Invocation of function *fn* on fields *f1*, *f2*, etc.	`isGood(quality)`
Flatten	`FLATTEN(`*f*`)`	Removal of a level of nesting from bags and tuples	`FLATTEN(group)`

8. Types

Category	Type	Description	Literal example
Boolean	`boolean`	True/false value	`true`
Numeric	`int`	32-bit signed integer	`1`
	`long`	64-bit signed integer	`1L`
	`float`	32-bit floating-point number	`1.0F`
	`double`	64-bit floating-point number	`1.0`
	`biginteger`	Arbitrary-precision integer	`'10000000000'`
	`bigdecimal`	Arbitrary-precision signed decimal number	`'0.1100010000000000000000001'`
Text	`chararray`	Character array in UTF-16 format	`'a'`
Binary	`bytearray`	Byte array	Not supported
Temporal	`datetime`	Date and time with time zone	Not supported, use `ToDate` built-in function
Complex	`tuple`	Sequence of fields of any type	`(1,'pomegranate')`
	`bag`	Unordered collection of tuples, possibly with duplicates	`{(1,'pomegranate'),(2)}`
	`map`	Set of key-value pairs; keys must be character arrays, but values may be any type	`['a'#'pomegranate']`

9. Schemas

A relation in Pig may have an associated schema, which gives the fields in the relation names and types. AS clause in a LOAD statement is used to attach a schema to a relation

```
grunt> records = LOAD 'input/ncdc/micro-tab/sample.txt'
>> AS (year:int, temperature:int, quality:int);
grunt> DESCRIBE records;
records: {year: int,temperature: int,quality: int}
```

It's possible to omit type declarations completely, too:

```
grunt> records = LOAD 'input/ncdc/micro-tab/sample.txt'
>> AS (year, temperature, quality);
grunt> DESCRIBE records;
records: {year: bytearray,temperature: bytearray,quality: bytearray}
```

In this case, we have specified only the names of the fields in the schema: year, temperature, and quality. The types default to bytearray, the most general type, representing a binary string. You don't need to specify types for every field; you can leave some to default to bytearray, as we have done for year in this declaration:

```
grunt> records = LOAD 'input/ncdc/micro-tab/sample.txt'
>> AS (year, temperature:int, quality:int);
grunt> DESCRIBE records;
records: {year: bytearray,temperature: int,quality: int}
```

However, if you specify a schema in this way, you do need to specify every field Fields in a relation with no schema can be referenced using only positional notation: $0 refers to the first field in a relation, $1 to the second, and so on. Their types default to bytearray:

```
grunt> projected_records = FOREACH records GENERATE $0, $1, $2;
grunt> DUMP projected_records;
(1950,0,1)
```

```
(1950,22,1)
(1950,-11,1)
(1949,111,1)
(1949,78,1)
grunt> DESCRIBE projected_records;
projected_records: {bytearray,bytearray,bytearray}
```

The schema is entirely optional and can be omitted by not specifying an AS clause:

```
grunt> records = LOAD 'input/ncdc/micro-tab/sample.txt';
grunt> DESCRIBE records;
Schema for records unknown.
```

10. Using Hive tables with HCatalog

A set of Pig queries over the same input data will often have the same schema repeated in each query. If the query processes a large number of fields, this repetition can become hard to maintain. HCatalog (which is a component of Hive) solves this problem by providing access to Hive's metastore, so that Pig queries can reference schemas by name, rather than specifying them in full each time.

```
% pig -useHCatalog
grunt> records = LOAD 'records' USING org.apache.hcatalog.pig.HCatLoader();
grunt> DESCRIBE records;
records: {year: chararray,temperature: int,quality: int}
grunt> DUMP records;
(1950,0,1)
(1950,22,1)
(1950,-11,1)
(1949,111,1)
(1949,78,1)
```

10.1 Validation and nulls

A SQL database will enforce the constraints in a table's schema at load time; for example, trying to load a string into a column that is declared to be a numeric type will fail.

Pig, if the value cannot be cast to the type declared in the schema, it will substitute a null value.Let's see how this works when we have the following input for the weather data, which has an "e" character in place of an integer:

1950 0 1
1950 22 1
1950 e 1
1949 111 1
1949 78 1

Pig handles the corrupt line by producing a null for the offending value, which is displayed as the absence of a value when dumped to screen

grunt> **records = LOAD 'input/ncdc/micro-tab/sample_corrupt.txt'**
>> **AS (year:chararray, temperature:int, quality:int);**
grunt> **DUMP records;**
(1950,0,1)
(1950,22,1)
(1950,,1)
(1949,111,1)
(1949,78,1)

Perhaps pull out all of the invalid records in one go so we can take action on them, perhaps by fixing our program (because they indicate that we have made a mistake) or by filtering them out (because the data is genuinely unusable):

grunt> **corrupt_records = FILTER records BY temperature is null;**
grunt> **DUMP corrupt_records;**

(1950,,1)

Another useful technique is to use the SPLIT operator to partition the data into "good"and "bad" relations, which can then be analyzed separately:

grunt> **SPLIT records INTO good_records IF temperature is not null,**

>> **bad_records OTHERWISE;**

grunt> **DUMP good_records;**

(1950,0,1)

(1950,22,1)

(1949,111,1)

(1949,78,1)

grunt> **DUMP bad_records;**

(1950,,1)

10.2 Schema merging

Some relational operators don't change the schema, so the relation produced by the LIMIT operator (which restricts a relation to a maximum number of tuples), for example, has the same schema as the relation it operates on. For other operators, the situation is more complicated. UNION, for example, combines two or more relations into one and tries to merge the input relations' schemas. If the schemas are incompatible, due to different types or number of fields,then the schema of the result of the UNION is unknown.

If you want to redefine the schema for a relation, you can use the FOREACH...GENERATE operator with AS clauses to define the schema for some or all of the fields of the input relation.

11. Functions

Functions in Pig come in four types:

Eval function

A function that takes one or more expressions and returns another expression. An example of a built-in eval function is MAX, which returns the maximum value of the entries in a bag. Some eval functions are *aggregate functions*, which means they operate on a bag of data to produce a scalar value; MAX is an example of an aggregate function.

Filter function

A special type of eval function that returns a logical Boolean result. As the name suggests, filter functions are used in the FILTER operator to remove unwanted rows.

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. Often, load and store functions are implemented by the same type. For example, PigStorage, which loads data from delimited text files, can store data in the same format.

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 {}.
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.

12. Macros

Macros provide a way to package reusable pieces of Pig Latin code from within Pig Latin itself. For example, we can extract the part of our Pig Latin program that performs grouping on a relation and then finds the maximum value in each group by defining a macro as follows:

```
DEFINE max_by_group(X, group_key, max_field) RETURNS Y {
A = GROUP $X by $group_key;
$Y = FOREACH A GENERATE group, MAX($X.$max_field);
};
```

The macro, called max_by_group, takes three parameters: a relation, X, and two field names, group_key and max_field. It returns a single relation, Y. Within the macro body, parameters and return aliases are referenced with a $ prefix, such as $X.

The macro is used as follows:

```
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);
max_temp = max_by_group(filtered_records, year, temperature);
DUMP max_temp
```

13. User-Defined Functions

13.1 A Filter UDF

Let's demonstrate by writing a filter function for filtering out weather records that do not have a temperature quality reading of satisfactory (or better). The idea is to change this line:

```
filtered_records = FILTER records BY temperature != 9999 AND
quality IN (0, 1, 4, 5, 9);
```

to

```
filtered_records = FILTER records BY temperature != 9999 AND isGood(quality);
```

FilterFunc UDF to remove records with unsatisfactory temperature quality readings

```
package com.hadoopbook.pig;
import java.io.IOException;
```

```
import java.util.ArrayList;
import java.util.List;
import org.apache.pig.FilterFunc;
import org.apache.pig.backend.executionengine.ExecException;
import org.apache.pig.data.DataType;
import org.apache.pig.data.Tuple;
import org.apache.pig.impl.logicalLayer.FrontendException;
public class IsGoodQuality extends FilterFunc {
@Override
public Boolean exec(Tuple tuple) throws IOException {
if (tuple == null || tuple.size() == 0) {
return false;
}
try {
Object object = tuple.get(0);
if (object == null) {
return false;
}
int i = (Integer) object;
return i == 0 || i == 1 || i == 4 || i == 5 || i == 9;
} catch (ExecException e) {
throw new IOException(e);
}
}
}
```

To use the new function, we first compile it and package it in a JAR file

```
grunt> REGISTER pig-examples.jar;
```

13.2 An Eval UDF

Consider the UDF in Example, which trims the leading and trailing whitespace from chararray values using the trim() method on java.lang.String.6

Example . An EvalFunc UDF to trim leading and trailing whitespace from chararray values

```
public class Trim extends PrimitiveEvalFunc<String, String> {
@Override
public String exec(String input) {
return input.trim();
}
}
```

The Trim UDF returns a string, which Pig translates as a chararray, as can be seen from the following session:

```
grunt> DUMP A;
( pomegranate)
(banana )
(apple)
( lychee )
grunt> DESCRIBE A;
A: {fruit: chararray}
grunt> B = FOREACH A GENERATE com.hadoopbook.pig.Trim(fruit);
grunt> DUMP B;
(pomegranate)
(banana)
(apple)
(lychee)
grunt> DESCRIBE B;
B: {chararray}
```

13.3 Dynamic invokers

Sometimes you want to use a function that is provided by a Java library, but without going to the effort of writing a UDF. Dynamic invokers allow you to do this by calling Java methods directly

from a Pig script.

The following snippet shows how we could define and use a trim UDF that uses the Apache Commons Lang StringUtils class:

```
grunt> DEFINE trim InvokeForString('org.apache.commons.lang.StringUtils.trim',
>> 'String');
grunt> B = FOREACH A GENERATE trim(fruit);
grunt> DUMP B;
(pomegranate)
(banana)
(apple)
(lychee)
```

13.4 A Load UDF

A custom load function that can read plain-text column ranges as fields, very much like the Unix cut command. It is used as follows:

```
grunt> records = LOAD 'input/ncdc/micro/sample.txt'
>> USING com.hadoopbook.pig.CutLoadFunc('16-19,88-92,93-93')
>> AS (year:int, temperature:int, quality:int);
grunt> DUMP records;
(1950,0,1)
(1950,22,1)
(1950,-11,1)
(1949,111,1)
(1949,78,1)
```

The string passed to CutLoadFunc is the column specification; each comma-separated range defines a field, which is assigned a name and type in the AS clause.

A LoadFunc UDF to load tuple fields as column ranges

```
public CutLoadFunc(String cutPattern) {
  ranges = Range.parse(cutPattern);
}

@Override
public void setLocation(String location, Job job)
    throws IOException {
  FileInputFormat.setInputPaths(job, location);
}

@Override
public InputFormat getInputFormat() {
  return new TextInputFormat();
}

@Override
public void prepareToRead(RecordReader reader, PigSplit split) {
  this.reader = reader;
}

@Override
public Tuple getNext() throws IOException {
  try {
    if (!reader.nextKeyValue()) {
      return null;
    }
    Text value = (Text) reader.getCurrentValue();
    String line = value.toString();
    Tuple tuple = tupleFactory.newTuple(ranges.size());
    for (int i = 0; i < ranges.size(); i++) {
      Range range = ranges.get(i);
      if (range.getEnd() > line.length()) {
        LOG.warn(String.format(
            "Range end (%s) is longer than line length (%s)",
            range.getEnd(), line.length()));
```

//CutLoadFunc is constructed with a string that specifies the column ranges to use foreach field. The logic for parsing this string and creating a list of internal Range objects that encapsulates these ranges is contained in the Range class

// setLocation() on a **LoadFunc** to pass the input location to the loader.
Since CutLoadFunc uses a **TextInputFormat** to break the input into lines, we just pass the **location** to set the input path using a static method on **FileInputFormat**

getInputFormat() method to create a **RecordReader** for each **split**,
Pig passes each RecordReader to the **prepareToRead()** method of CutLoadFunc, which we store a reference to, so we can use it in the **getNext() method for iterating through the records**.

//when the line is shorter than the range asked for.
throw an exception

14. Data Processing Operators

14.1 Loading and Storing Data

Example of using PigStorage to store tuples as plain-text values separated by a colon character:

grunt> STORE A INTO 'out' USING PigStorage(':');

grunt> cat out

Joe:cherry:2

Ali:apple:3

Joe:banana:2

Eve:apple:7

14.2 Filtering Data

FOREACH...GENERATE

The FOREACH...GENERATE operator is used to act on every row in a relation. It can be used to remove fields or to generate new ones.

In this example, we do both:

grunt> **DUMP A;**

(Joe,cherry,2)

(Ali,apple,3)

(Joe,banana,2)

(Eve,apple,7)

grunt> **B = FOREACH A GENERATE $0, $2+1, 'Constant';**

grunt> **DUMP B;**

(Joe,3,Constant)

(Ali,4,Constant)

(Joe,3,Constant)

(Eve,8,Constant)

14.3 STREAM

- The STREAM operator allows you to transform data in a relation using an external program or script
- Unix cut command to extract the second field of each tuple in A

grunt> **C = STREAM A THROUGH `cut -f 2`;**

grunt> **DUMP C;**

(cherry)

(apple)

(banana)

(apple)

14.4 Grouping and Joining Data

JOIN (Inner join)

grunt> **DUMP A;**

(2,Tie)

(4,Coat)

(3,Hat)

(1,Scarf)

grunt> **DUMP B;**

(Joe,2)

(Hank,4)

(Ali,0)

(Eve,3)

(Hank,2)

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)

If one of the relations is small enough to fit in memory, you can use a special type of join called a fragment replicate join, which is implemented by distributing the small input to all the mappers and performing a map-side join. There is a special syntax for telling Pig to use a fragment replicate join:

grunt> C = JOIN A BY $0, B BY $1 USING 'replicated

Pig also supports outer joins using a syntax that is similar to SQL's

grunt> **C = JOIN A BY $0 LEFT OUTER, B BY $1;**

grunt> **DUMP C;**

(1,Scarf,,)

(2,Tie,Hank,2)

(2,Tie,Joe,2)

(3,Hat,Eve,3)

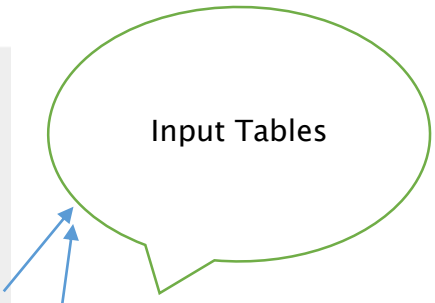
(4,Coat,Hank,4)

COGROUP

- The group operator is normally used with one relation, while the cogroup operator is used in statements involving two or more relations.

student_details.txt

```
001,Rajiv,Reddy,21,9848022337,Hyderabad
002,siddarth,Battacharya,22,9848022338,Kolkata
003,Rajesh,Khanna,22,9848022339,Delhi
004,Preethi,Agarwal,21,9848022330,Pune
005,Trupthi,Mohanthy,23,9848022336,Bhuwaneshwar
006,Archana,Mishra,23,9848022335,Chennai
007,Komal,Nayak,24,9848022334,trivendram
008,Bharathi,Nambiayar,24,9848022333,Chennai
```



employee_details.txt

```
001,Robin,22,newyork
002,BOB,23,Kolkata
003,Maya,23,Tokyo
004,Sara,25,London
005,David,23,Bhuwaneshwar
006,Maggy,22,Chennai
```

- **grunt> student_details=LOAD'hdfs://localhost:9000/pig_data/student_details.txt' USING PigStorage(',') as (id:int, firstname:chararray, lastname:chararray, age:int, phone:chararray, city:chararray);**
- **grunt>employee_details=LOAD'hdfs://localhost:9000/pig_data/employee_details.txt ' USING PigStorage(',')as (id:int, name:chararray, age:int, city:chararray);**
- **grunt> cogroup_data = COGROUP student_details by age, employee_details by age;**
- **grunt> Dump cogroup_data;**

Output:

- (21,{(4,Preethi,Agarwal,21,9848022330,Pune),(1,Rajiv,Reddy,21,9848022337,Hyderabad)}, { })
- (22,{(3,Rajesh,Khanna,22,9848022339,Delhi),(2,siddarth,Battacharya,22,9848022338,Kolkata) }, { (6,Maggy,22,Chennai),(1,Robin,22,newyork) })
- (23,{(6,Archana,Mishra,23,9848022335,Chennai),(5,Trupthi,Mohanthy,23,9848022336,Bhuwaneshwar)},{(5,David,23,Bhuwaneshwar),(3,Maya,23,Tokyo),(2,BOB,23,Kolkata)})
- (24,{(8,Bharathi,Nambiayar,24,9848022333,Chennai),(7,Komal,Nayak,24,9848022334,trivendram)}, { })
- (25,{ }, {(4,Sara,25,London)})

Cross Operator

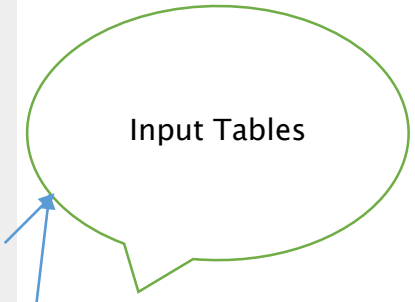
The CROSS operator computes the cross-product of two or more relations

customers.txt

```
1,Ramesh,32,Ahmedabad,2000.00
2,Khilan,25,Delhi,1500.00
3,kaushik,23,Kota,2000.00
4,Chaitali,25,Mumbai,6500.00
5,Hardik,27,Bhopal,8500.00
6,Komal,22,MP,4500.00
7,Muffy,24,Indore,10000.00
```

orders.txt

```
102,2009-10-08 00:00:00,3,3000
100,2009-10-08 00:00:00,3,1500
101,2009-11-20 00:00:00,2,1560
103,2008-05-20 00:00:00,4,2060
```



- grunt> cross_data = CROSS customers, orders;
- grunt> Dump cross_data;

Sample Output

```
(7,Muffy,24,Indore,10000,103,2008-05-20 00:00:00,4,2060)
(7,Muffy,24,Indore,10000,101,2009-11-20 00:00:00,2,1560)
(7,Muffy,24,Indore,10000,100,2009-10-08 00:00:00,3,1500)
(7,Muffy,24,Indore,10000,102,2009-10-08 00:00:00,3,3000)
(6,Komal,22,MP,4500,103,2008-05-20 00:00:00,4,2060)
(6,Komal,22,MP,4500,101,2009-11-20 00:00:00,2,1560)
(6,Komal,22,MP,4500,100,2009-10-08 00:00:00,3,1500)
(6,Komal,22,MP,4500,102,2009-10-08 00:00:00,3,3000)
(5,Hardik,27,Bhopal,8500,103,2008-05-20 00:00:00,4,2060)
(5,Hardik,27,Bhopal,8500,101,2009-11-20 00:00:00,2,1560)
(5,Hardik,27,Bhopal,8500,100,2009-10-08 00:00:00,3,1500)
(5,Hardik,27,Bhopal,8500,102,2009-10-08 00:00:00,3,3000)
```

Group Operator

The GROUP operator is used to group the data in one or more relations. It collects the data having the same key

student_details.txt

```
001,Rajiv,Reddy,21,9848022337,Hyderabad
002,siddarth,Battacharya,22,9848022338,Kolkata
003,Rajesh,Khanna,22,9848022339,Delhi
004,Preethi,Agarwal,21,9848022330,Pune
005,Trupthi,Mohanthy,23,9848022336,Bhuwaneshwar
006,Archana,Mishra,23,9848022335,Chennai
007,Komal,Nayak,24,9848022334,trivendram
008,Bharathi,Nambiayar,24,9848022333,Chennai
```

INPUT TABLE

- grunt> group_data = GROUP student_details by age;
- grunt> Dump group_data;

OUTPUT

```
(21,{(4,Preethi,Agarwal,21,9848022330,Pune),(1,Rajiv,Reddy,21,9848022337,Hydera bad)})
(22,{(3,Rajesh,Khanna,22,9848022339,Delhi),(2,siddarth,Battacharya,22,984802233 8,Kolkata)})
(23,{(6,Archana,Mishra,23,9848022335,Chennai),(5,Trupthi,Mohanthy,23,9848022336
,Bhuwaneshwar)})
(24,{(8,Bharathi,Nambiayar,24,9848022333,Chennai),(7,Komal,Nayak,24,9848022334, trivendram)})
```

15. Sorting

Order By

The ORDER BY operator is used to display the contents of a relation in a sorted order based on one or more fields

INPUT TABLE

student_details.txt

```
001,Rajiv,Reddy,21,9848022337,Hyderabad
002,siddarth,Battacharya,22,9848022338,Kolkata
003,Rajesh,Khanna,22,9848022339,Delhi
004,Preethi,Agarwal,21,9848022330,Pune
005,Trupthi,Mohanthy,23,9848022336,Bhuwaneshwar
006,Archana,Mishra,23,9848022335,Chennai
007,Komal,Nayak,24,9848022334,trivendram
008,Bharathi,Nambiayar,24,9848022333,Chennai
```

grunt> order_by_data = ORDER student_details BY age DESC;

• Output

(8,Bharathi,Nambiayar,24,9848022333,Chennai)
(7,Komal,Nayak,24,9848022334,trivendram)
(6,Archana,Mishra,23,9848022335,Chennai)
(5,Trupthi,Mohanthy,23,9848022336,Bhuwaneshwar)
(3,Rajesh,Khanna,22,9848022339,Delhi)
(2,siddarth,Battacharya,22,9848022338,Kolkata)
(4,Preethi,Agarwal,21,9848022330,Pune)
(1,Rajiv,Reddy,21,9848022337,Hyderabad)

Limit Operator

The LIMIT operator is used to get a limited number of tuples from a relation**.**

grunt> limit_data = LIMIT student_details 4;

output:

INPUT TABLE

```
(1,Rajiv,Reddy,21,9848022337,Hyderabad)
(2,siddarth,Battacharya,22,9848022338,Kolkata)
(3,Rajesh,Khanna,22,9848022339,Delhi)
(4,Preethi,Agarwal,21,9848022330,Pune)
```

16. Combining and Splitting

- **Union Operator**

  To perform UNION operation on two relations, their columns and domains must be identical

INPUT TABLES

Student_data1.txt

```
001,Rajiv,Reddy,9848022337,Hyderabad
002,siddarth,Battacharya,9848022338,Kolkata
003,Rajesh,Khanna,9848022339,Delhi
004,Preethi,Agarwal,9848022330,Pune
005,Trupthi,Mohanthy,9848022336,Bhuwaneshwar
006,Archana,Mishra,9848022335,Chennai.
```

Student_data2.txt

```
7,Komal,Nayak,9848022334,trivendram.
8,Bharathi,Nambiayar,9848022333,Chennai.
```

```
grunt> student1 = LOAD 'hdfs://localhost:9000/pig_data/student_data1.txt' USING PigStorage(',')
        as (id:int, firstname:chararray, lastname:chararray, phone:chararray, city:chararray);

grunt> student2 = LOAD 'hdfs://localhost:9000/pig_data/student_data2.txt' USING PigStorage(',')
        as (id:int, firstname:chararray, lastname:chararray, phone:chararray, city:chararray);
```

- grunt> student = UNION student1, student2;

- Output

```
(1,Rajiv,Reddy,9848022337,Hyderabad)
(2,siddarth,Battacharya,9848022338,Kolkata)
(3,Rajesh,Khanna,9848022339,Delhi)
(4,Preethi,Agarwal,9848022330,Pune)
(5,Trupthi,Mohanthy,9848022336,Bhuwaneshwar)
(6,Archana,Mishra,9848022335,Chennai)
(7,Komal,Nayak,9848022334,trivendram)
(8,Bharathi,Nambiayar,9848022333,Chennai)
```

Split Operator

Split Operator

- The **SPLIT** operator is used to split a relation into two or more relations.

SPLIT student_details into student_details1 if age<23, student_details2 if (22<age and age>25);

INPUT TABLE

student_details.txt

```
001,Rajiv,Reddy,21,9848022337,Hyderabad
002,siddarth,Battacharya,22,9848022338,Kolkata
003,Rajesh,Khanna,22,9848022339,Delhi
004,Preethi,Agarwal,21,9848022330,Pune
005,Trupthi,Mohanthy,23,9848022336,Bhuwaneshwar
006,Archana,Mishra,23,9848022335,Chennai
007,Komal,Nayak,24,9848022334,trivendram
008,Bharathi,Nambiayar,24,9848022333,Chennai
```

OUTPUT

```
grunt> Dump student_details1;
(1,Rajiv,Reddy,21,9848022337,Hyderabad)
(2,siddarth,Battacharya,22,9848022338,Kolkata)
(3,Rajesh,Khanna,22,9848022339,Delhi)
(4,Preethi,Agarwal,21,9848022330,Pune)

grunt> Dump student_details2;
(5,Trupthi,Mohanthy,23,9848022336,Bhuwaneshwar)
(6,Archana,Mishra,23,9848022335,Chennai)
(7,Komal,Nayak,24,9848022334,trivendram)
(8,Bharathi,Nambiayar,24,9848022333,Chennai)
```

17. Pig in Practice

- Some practical techniques that are worth knowing about when you are developing and running Pig programs.

Parallelism

- By default, Pig sets the number of reducers by looking at the size of the input and using one reducer per 1 GB of input, up to a maximum of 999 reducers.
- You can override these parameters by setting pig.exec.reducers .bytes.per.reducer (the default is 1,000,000,000 bytes) and pig.exec.reducers.max (the default is 999).
- To explicitly set the number of reducers you want for each job, you can use a PARALLEL clause for operators that run in the reduce phase.
- These include all the grouping and joining operators (GROUP, COGROUP, JOIN, CROSS), as well as DISTINCT and ORDER.
- The following line sets the number of reducers to 30 for the GROUP:

```
grouped_records = GROUP records BY year PARALLEL 30;
```

- The number of map tasks is set by the size of the input (with one map per HDFS block) and is not affected by the PARALLEL clause

Anonymous Relations

- You usually apply a diagnostic operator like DUMP or DESCRIBE to the most recently defined relation.
- Similarly, it can be tiresome to have to come up with a name for each relation when using the interpreter.
- Pig allows you to use the special syntax => to create a relation with no alias, which can only be referred to with @.

```
grunt> => LOAD 'input/ncdc/micro-tab/sample.txt';
grunt> DUMP @
(1950,0,1)
(1950,22,1)
(1950,-11,1)
```

(1949,111,1)

(1949,78,1)

Parameter Substitution

- If you have a Pig script that you run on a regular basis, it's quite common to want to be able to run the same script with different parameters.
- Pig supports *parameter substitution*, where parameters in the script are substituted with values supplied at runtime.
- Parameters are denoted by identifiers prefixed with a $ character

```
-- max_temp_param.pig
records = LOAD '$input' AS (year:chararray, temperature:int, quality:int);
filtered_records = FILTER records BY temperature != 9999 AND
quality IN (0, 1, 4, 5, 9);
grouped_records = GROUP filtered_records BY year;
max_temp = FOREACH grouped_records GENERATE group,
MAX(filtered_records.temperature);
STORE max_temp into '$output';
```

Parameters can be specified when launching Pig using the -param option, once for each parameter:

```
% pig -param input=/user/tom/input/ncdc/micro-tab/sample.txt \
> -param output=/tmp/out \
> ch16-pig/src/main/pig/max_temp_param.pi
```

Dynamic parameters

- For parameters that are supplied using the -param option, it is easy to make the value dynamic by running a command or script.
- Many Unix shells support command substitution for a command enclosed in backticks, and we can use this to make the output directory date-based:

```
% pig -param input=/user/tom/input/ncdc/micro-tab/sample.txt \
> -param output=/tmp/`date "+%Y-%m-%d"`/out \
```

> ch16-pig/src/main/pig/max_temp_param.pig

HIVE

18. HIVE

Hive is a data warehouse infrastructure tool to process structured data in Hadoop. It resides on top of Hadoop to summarize big data, and makes querying and analyzing easy.
Initially hive was developed by Jeff Hammerbacher at facebook, later the apache software foundation took it up and developed it further as an open source under the apache hive.

Hive is not

- A relational database
- A design for online transaction processing(OLTP)
- A language for real time queries and row level updates

Features of hive

- It stores schema in a database and processed data into HDFS
- It is designed for OLAP
- It provides SQL type language for querying called HiveQL or HQL
- It is familiar, fast, scalable, and extensible

19. Hive Shell

The shell is the primary way that we will interact with Hive, by issuing commands in *HiveQL* . HiveQL is Hive's query language, a dialect of SQL. It is heavily influenced by MySQL, so if you are familiar with MySQL you should feel at home using Hive.

When starting Hive for the first time, we can check that it is working by listing its tables: there should be none. The command must be terminated witha semicolon to tell Hive to execute it:

Ex: hive> **SHOW TABLES;**

OK

Time taken: 10.425 seconds

20. Features of shell

It is possible to run the hive shell in non-interactive mode. The –f option runsthe commands in the specified file, ex: *script.q*,

% **hive -f script.q**

For short scripts, you can use the -e option to specify the commands inline, inwhich case the final semicolon is not required:

% **hive -e 'SELECT * FROM dummy'**

Hive history file=/tmp/tom/hive_job_log_tom_201005042112_1906486281.txtOK

X

Time taken: 4.734 seconds

In both interactive and non-interactive mode, Hive will print information to standard error —such as the time taken to run a query, to surpress these messages using the –s option it shows only the result of the query.

% **hive -S -e 'SELECT * FROM dummy'**

X

Other useful Hive shell features include the ability to run commands on the host operating system by using a! Prefix to the command and the ability to access Hadoop file systems using the dfs command

21. Hive Services

The Hive shell is only one of several services that you can run using the hive command.
You can specify the service to run using the --service option. Type hive –servicehelp to get a list of available service names; the most useful are described below.

- Cli: The command line interface to Hive (the shell). This is the default service.

a) Hiveserver: Runs Hive as a server exposing a Thrift service, enabling access from a range of

clients written in different languages. Applications using the Thrift, JDBC, and ODBC connectors need to run a Hive server to communicate with Hive. Set the HIVE_PORT environment variable to specify the port the server will listen on(defaults to 10,000).

- Hwi: The Hive Web Interface. A simple web interface that can be used as an alternative to the CLI without having to install any client software

b) Jar: The Hive equivalent to hadoop jar, a convenient way to run Java applications that includes both Hadoop and Hive classes on the classpath.

c) Metastore: Using this service, it is possible to run the metastore as a standalone (remote) process.

22. Hive Clients

Hive as a server (hive --service hiveserver), then there are a number ofdifferent mechanisms for connecting to it from applications.

The relationship between Hive clients and Hive services is illustrated in Figure

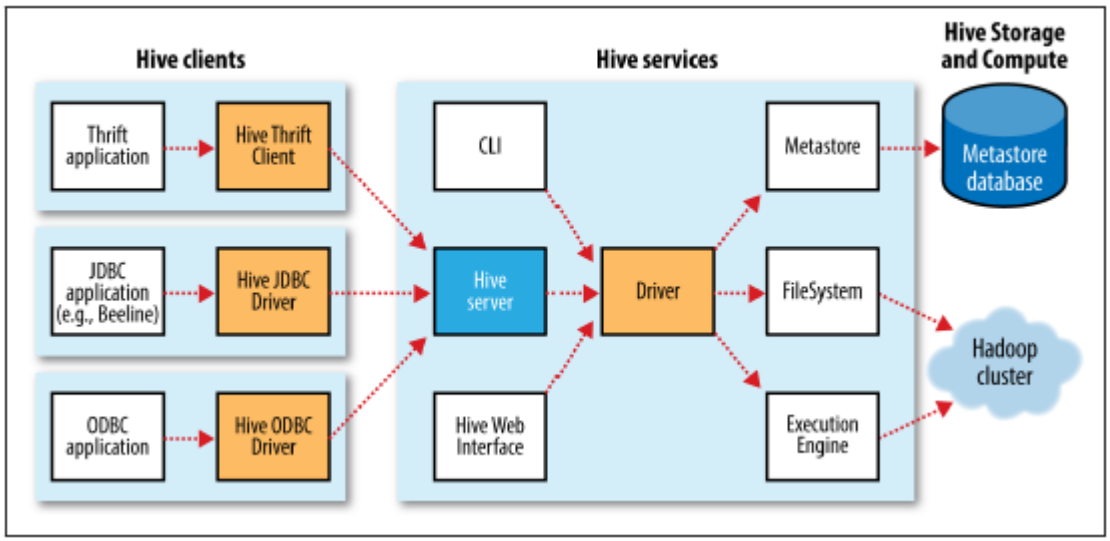


Fig: Hive architecture

- ***Thrift Client***

The Hive Thrift Client makes it easy to run Hive commands from a wide range of programming languages. Thrift bindings for Hive are available for C++, Java, PHP, Python, and Ruby.

- ***JDBC Driver***

Hive provides a Type 4 (pure Java) JDBC driver, defined in the class org.apache.hadoop.hive.jdbc.HiveDriver. When configured with a JDBC URI of the form jdbc:hive://*host*:*port*/*dbname*, a Java application will connect to aHive server running in a separate process at the given host and port

- ***ODBC Driver***

The Hive ODBC Driver allows applications that support the ODBC protocol toconnect to Hive.

23. The metastore

- The *metastore* is the central repository of Hive metadata.
- The metastore is divided into two pieces: a service and the backingstore for the data. There are 3 different metastore configurations

a) Embedded metastore

- By default, the metastore service runs in the same JVM as the Hive service and contains an embedded Derby database instance backed by the local disk. This is called the *embedded metastore* configuration
- A simple way to get started with Hive
- Only one embedded Derby database can access the database files on diskat any one time, which means you can only have one Hive session
- If we open another session it attempts to open a connection to the metastore i.e., trying to start a second session gives the error
  Error: Failed to start database 'metastore_db'

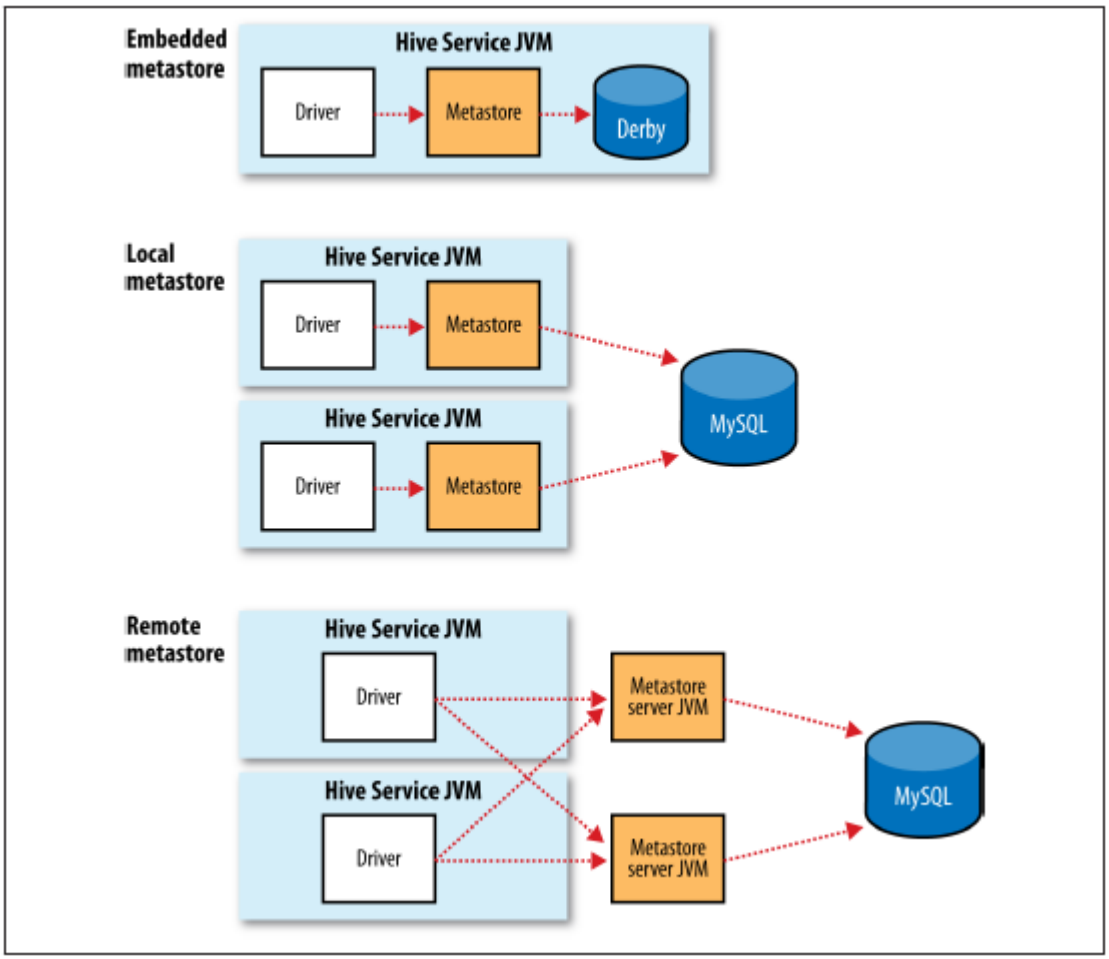


Figure : Metastore configurations

b) local metastore

- To support multiple sessions (and therefore multiple users) is to use a standalone database.
- This configuration is referred to as a *local metastore* , since the metastore service still runs in the same process as the Hive service, but connects to a database running in a separate process, either on the same machine or on a remote machine.

c) Remote metastore

- Where one or more metastore servers run in separate processes tothe Hive service.
- This brings better manageability and security, since the database tier can be

completely firewalled off, and the clients no longer needthe database credentials.

24. Comparison with traditional databases Schema on Read versus Schema on Write

- In a traditional database, a table's schema is enforced at data load time.
- Hive does not verify the data when it is loaded but it is verifiedwhen the query is issued
- Traditional db takes longer time to load data
    - Schema on read makes for a very fastinitial load **Query time performance** :
    - Schema on write makes query time performance faster
    - Schema on read makes longer time for query execution

Transactions:

Hive doesnot support transactions .

Hive does not support updates(or deletes) .

Indexes:

Release 0.7.0 introduced indexes, which can speed up queries

Locking:

Release 0.7.0 introduces table and partitional level locking in hive. Locks are managed transparently by zookeeper.

25. Hive QL

- HiveQL is Hive's SQL dialect .
- It does not provide the full features of SQL -92 language constructs.
- The main difference between HiveQL and SQL are *Table : A high -level comparison ofSQL and HiveQL*

Feature	SQL	HiveQL
Updates	UPDATE, INSERT, DELETE	UPDATE, INSERT, DELETE
Transactions	Supported	Limited support
Indexes	Supported	Supported
Data types	Integral, floating-point, fixed-point, text and binary strings, temporal	Boolean, integral, floating-point, fixed-point, text and binary strings, temporal, array, map, struct
Functions	Hundreds of built-in functions	Hundreds of built-in functions
Multitable inserts	Not supported	Supported
CREATE TABLE...AS SELECT	Not valid SQL-92, but found in some databases	Supported
SELECT	SQL-92	SQL-92. SORT BY for partial ordering, LIMIT to limit number of rows returned
Joins	SQL-92, or variants (join tables in the FROM clause, join condition in the WHERE clause)	Inner joins, outer joins, semi joins, map joins, cross joins
Subqueries	In any clause (correlated or noncorrelated)	In the FROM, WHERE, or HAVING clauses (uncorrelated subqueries not supported)
Views	Updatable (materialized or nonmaterialized)	Read-only (materialized views not supported)

26. HBasics

HBase is a distributed column-oriented database built on top of HDFS. HBase is the Hadoop application to use when you require real-time read/write random-access to very large datasets. Apache HBase is an open-source, distributed, versioned, non-relational database modeled after Google's Bigtable: A Distributed Storage System for Structured Data.

Features

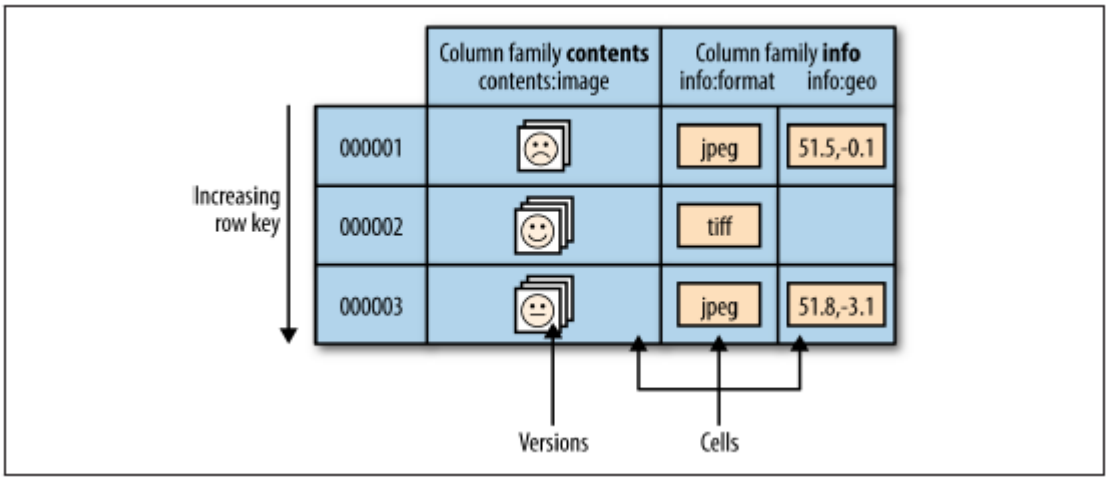
- Linear and modular scalability.
- Strictly consistent reads and writes.

- Automatic and configurable sharding of tables
- Automatic failover support between RegionServers.
- Convenient base classes for backing Hadoop MapReduce jobs with Apache HBase tables.
- Easy to use Java API for client access.
- Block cache and Bloom Filters for real-time queries.
- Query predicate push down via server side Filters.
- Thrift gateway and a REST-ful Web service that supports XML, Protobuf, and binary data encoding options
- Extensible jruby-based (JIRB) shell.

27. Concepts

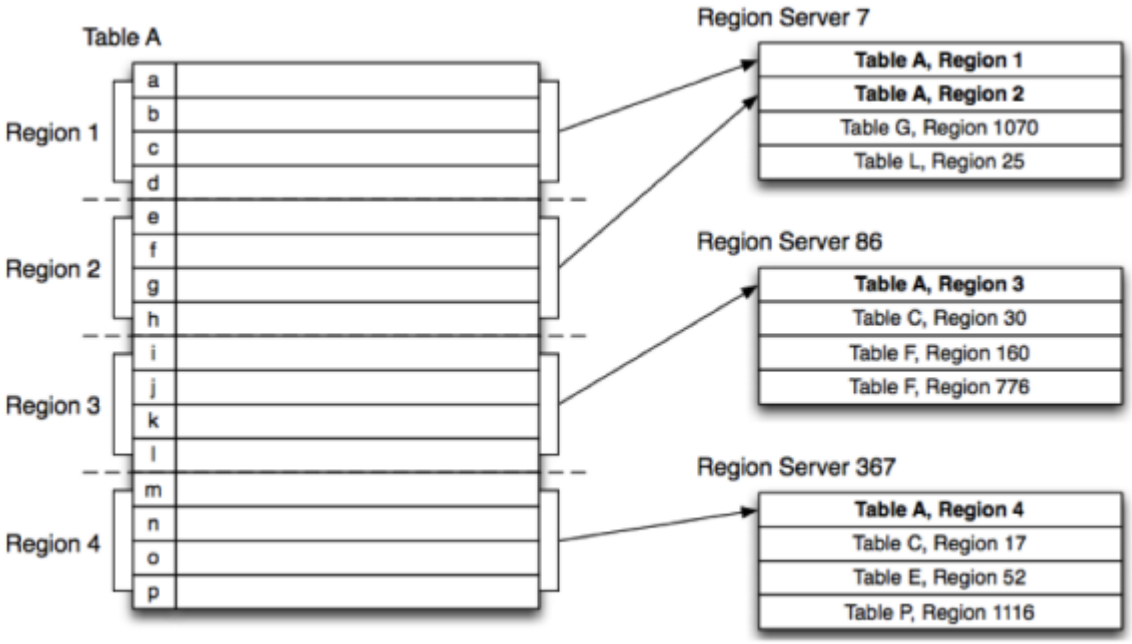
Whirlwind Tour of the Data Model

- Tables are made of rows and columns.
- Table cells—the intersection of row and column coordinates—are versioned.
- By default, their version is a timestamp auto-assigned by HBase at the time of cell insertion.
- A cell's content is an uninterpreted array of bytes
- Table row keys are also byte arrays, so theoretically anything can serve as a row key.
- Table rows are sorted by row key, aka the table's primary key
- Row columns are grouped into column families.
- All column family members have a common prefix, so, for example, the columns info:format and info:geo are both members of the info column family, whereas contents:image belongs to the contents family.
- A table's column families must be specified up front as part of the table schema definition, but new column family members can be added on demand.



Regions

Tables are automatically partitioned horizontally by HBase into regions. Each region comprises a subset of a table's rows. A region is denoted by the table it belongs to



Locking

Row updates are atomic, no matter how many row columns constitute the row-level transaction. This keeps the locking model simple.

28. Implementation

Components of Apache HBase Architecture

HBase architecture has 3 important components- HMaster, Region Server and ZooKeeper.

HMaster

HBase HMaster is a lightweight process that assigns regions to region servers in the Hadoop cluster for load balancing. Responsibilities of HMaster –

- Manages and Monitors the Hadoop Cluster
- Performs Administration (Interface for creating, updating and deleting tables.)
- Controlling the failover
- DDL operations are handled by the HMaster
- Whenever a client wants to change the schema and change any of the metadata operations, HMaster is responsible for all these operations.

Region Server

These are the worker nodes which handle read, write, update, and delete requests from clients. Region Server process, runs on every node in the hadoop cluster. Region Server runs on HDFS DataNode and consists of the following components –

- Block Cache – This is the read cache. Most frequently read data is stored in the read cache and whenever the block cache is full, recently used data is evicted.
- MemStore- This is the write cache and stores new data that is not yet written to the disk. Every column family in a region has a MemStore.
- Write Ahead Log (WAL) is a file that stores new data that is not persisted to permanent storage.
- HFile is the actual storage file that stores the rows as sorted key values on a disk.

Zookeeper

ZooKeeper service keeps track of all the region servers that are there in an HBase cluster- tracking information about how many region servers are there and which region servers are holding which DataNode.

Various services that Zookeeper provides include –

- Establishing client communication with region servers.
- Tracking server failure and network partitions.
- Maintain Configuration Information

- Provides ephemeral nodes, which represent different region servers

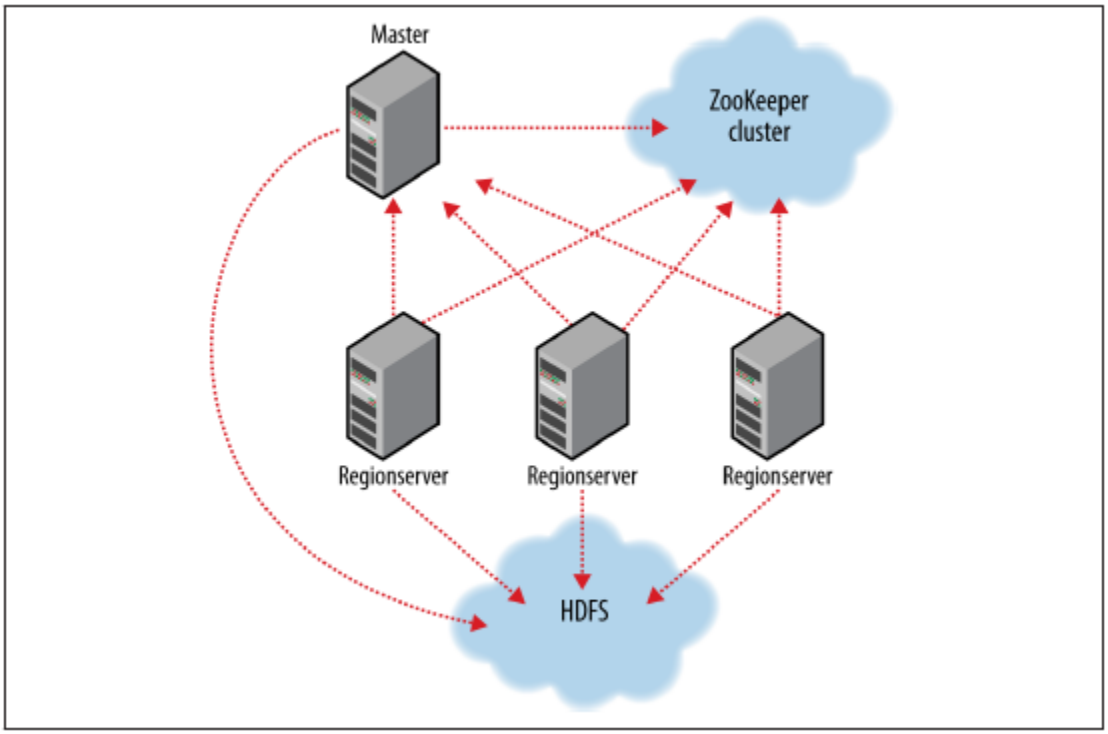


Figure. HBase cluster members

29. Clients

There are a number of client options for interacting with an HBase cluster.

Java

HBase, like Hadoop, is written in Java. Example shows the Java version of how you would do the shell operation.

```
public class ExampleClient {
public static void main(String[] args) throws IOException {
Configuration config = HBaseConfiguration.create();
// Create table
HBaseAdmin admin = new HBaseAdmin(config);
try {
```

```
TableName tableName = TableName.valueOf("test");
HTableDescriptor htd = new HTableDescriptor(tableName);
HColumnDescriptor hcd = new HColumnDescriptor("data");
htd.addFamily(hcd);
admin.createTable(htd);
HTableDescriptor[] tables = admin.listTables();
if (tables.length != 1 &&
Bytes.equals(tableName.getName(), tables[0].getTableName().getName())) {
throw new IOException("Failed create of table");
}
        // Run some operations -- three puts, a get, and a scan -- against the table.
HTable table = new HTable(config, tableName);
try {
for (int i = 1; i <= 3; i++) {
byte[] row = Bytes.toBytes("row" + i);
Put put = new Put(row);
byte[] columnFamily = Bytes.toBytes("data");
byte[] qualifier = Bytes.toBytes(String.valueOf(i));
byte[] value = Bytes.toBytes("value" + i);
put.add(columnFamily, qualifier, value);
table.put(put);
}
Get get = new Get(Bytes.toBytes("row1"));
Result result = table.get(get);
System.out.println("Get: " + result);
Scan scan = new Scan();
ResultScanner scanner = table.getScanner(scan);
try {
for (Result scannerResult : scanner) {
System.out.println("Scan: " + scannerResult);
}
```

```
} finally {
scanner.close();
}
// Disable then drop the table
admin.disableTable(tableName);
admin.deleteTable(tableName);
} finally {
table.close();
}
} finally {
admin.close();
}
}
        }
```

Program Explanation

- The HBaseConfiguration class to create a Configuration object.
- It will return a Configuration that has read the HBase configuration from the hbasesite.
- xml and hbase-default.xml files found on the program's classpath.
- This Configuration is subsequently used to create instances of HBaseAdmin and HTable. HBaseAdmin is used for administering your HBase cluster, specifically for adding and dropping tables.
- HTable is used to access a specific table.
- The Configuration instance points these classes at the cluster the code is to work against.
- To create a table, we need to create an instance of HBaseAdmin and then ask it to create
- The table named test with a single column family named data.
- In our example, our table schema is the default.
- We could use methods on HTableDescriptor and HColumn Descriptor to change the table schema.
- Next, the code asserts the table was actually created, and throws an exception if it wasn't.

- To operate on a table, we will need an instance of HTable, which we construct by passing
- It our Configuration instance and the name of the table.
- We then create Put objects in a loop to insert data into the table.
- Each Put puts a single cell value of valuen into a row named rown on the column named data:n, where n is from 1 to 3.
- The column name is specified in two parts: the column family name, and the column family qualifier.
- The code makes liberal use of HBase's Bytes utility class (found in the org.apache.ha
- doop.hbase.util package) to convert identifiers and values to the byte arrays that
- HBase requires. Next, we create a Get object to retrieve and print the first row that we added.
- Then we use a Scan object to scan over the table, printing out what we find.
- At the end of the program, we clean up by first disabling the table and then deleting it
- (recall that a table must be disabled before it can be dropped).

MapReduce

The TableInputFormat class makes splits on region boundaries so maps are handed a single region to work on. The TableOutputFormat will write the result of the reduce into HBase. The SimpleRowCounter class in Example runs a map task to count rows using TableInputFormat.

Example . A MapReduce application to count the number of rows in an HBase table

```
public class SimpleRowCounter extends Configured implements Tool {
static class RowCounterMapper extends TableMapper<ImmutableBytesWritable, Result> {
public static enum Counters { ROWS }
@Override
public void map(ImmutableBytesWritable row, Result value, Context context) {
context.getCounter(Counters.ROWS).increment(1);
}
}
@Override
public int run(String[] args) throws Exception {
```

```
if (args.length != 1) {
System.err.println("Usage: SimpleRowCounter <tablename>");
return -1;
}
String tableName = args[0];
Scan scan = new Scan();
scan.setFilter(new FirstKeyOnlyFilter());
Job job = new Job(getConf(), getClass().getSimpleName());
job.setJarByClass(getClass());
TableMapReduceUtil.initTableMapperJob(tableName, scan,
RowCounterMapper.class, ImmutableBytesWritable.class, Result.class, job);
job.setNumReduceTasks(0);
job.setOutputFormatClass(NullOutputFormat.class);
return job.waitForCompletion(true) ? 0 : 1;
}
public static void main(String[] args) throws Exception {
int exitCode = ToolRunner.run(HBaseConfiguration.create(),
new SimpleRowCounter(), args);
System.exit(exitCode);
}
}
```

Program Explanation

- *The RowCounterMapper nested class is a subclass of the HBase TableMapper abstract class, a specialization of org.apache.hadoop.mapreduce.Mapper that sets the map input types passed by TableInputFormat.*
- *Input keys are ImmutableBytesWritable objects (row keys), and values are Result objects (row results from a scan).*
- *Since this job counts rows and does not emit any output from the map, we just increment Counters.*
- *ROWS by 1 for every row we see.*

- *In the run() method, we create a scan object that is used to configure the job by invoking*
- *the TableMapReduceUtil.initTableMapJob() utility method, which, among other things (such as setting the map class to use), sets the input format to TableInputFormat*
- *Notice how we set a filter, an instance of FirstKeyOnlyFilter, on the scan.*
- *This filter instructs the server to short-circuit when running server-side, populating the Result object in the mapper with only the first cell in each row.*
- *Since the mapper ignores the*
- *cell values, this is a useful optimization*

30. Loading Data

There are a relatively small number of stations, so their static data is easily inserted using any of the available interfaces. The example code includes a Java application for doing this, which is run as follows:

% **hbase HBaseStationImporter input/ncdc/metadata/stations-fixed-width.txt**

However, let's assume that there are billions of individual observations to be loaded. This kind of import is normally an extremely complex and long-running database operation, but MapReduce and HBase's distribution model allow us to make full use of the cluster. We'll copy the raw input data onto HDFS, and then run a MapReduce job that can read the input and write to HBase

Example . A MapReduce application to import temperature data from HDFS into an HBase table

```
public class HBaseTemperatureImporter extends Configured implements Tool {
static class HBaseTemperatureMapper<K> extends Mapper<LongWritable, Text, K, Put> {
private NcdcRecordParser parser = new NcdcRecordParser();
@Override
public void map(LongWritable key, Text value, Context context) throws
IOException, InterruptedException {
parser.parse(value.toString());
if (parser.isValidTemperature()) {
byte[] rowKey = RowKeyConverter.makeObservationRowKey(parser.getStationId(),
```

```
p = new Put(rowKey);
p.add(HBaseTemperatureQuery.DATA_COLUMNFAMILY,
HBaseTemperatureQuery.AIRTEMP_QUALIFIER,
Bytes.toBytes(parser.getAirTemperature()));
context.write(null, p);
}
}
}
@Override
public int run(String[] args) throws Exception {
if (args.length != 1) {
System.err.println("Usage: HBaseTemperatureImporter <input>");
return -1;
}
Job job = new Job(getConf(), getClass().getSimpleName());
job.setJarByClass(getClass());
FileInputFormat.addInputPath(job, new Path(args[0]));
job.getConfiguration().set(TableOutputFormat.OUTPUT_TABLE, "observations");
job.setMapperClass(HBaseTemperatureMapper.class);
job.setNumReduceTasks(0);
job.setOutputFormatClass(TableOutputFormat.class);
return job.waitForCompletion(true) ? 0 : 1;
}
public static void main(String[] args) throws Exception {
int exitCode = ToolRunner.run(HBaseConfiguration.create(),
new HBaseTemperatureImporter(), args);
System.exit(exitCode);
}
}
```

The row key for each observation is created in the makeObservationRowKey() method on RowKeyConverter from the station ID and observation time:

```
public class RowKeyConverter {
private static final int STATION_ID_LENGTH = 12;
* @return A row key whose format is: <station_id> <reverse_order_timestamp>
*/
public static byte[] makeObservationRowKey(String stationId,
long observationTime) {
byte[] row = new byte[STATION_ID_LENGTH + Bytes.SIZEOF_LONG];
Bytes.putBytes(row, 0, Bytes.toBytes(stationId), 0, STATION_ID_LENGTH);
long reverseOrderTimestamp = Long.MAX_VALUE - observationTime;
Bytes.putLong(row, STATION_ID_LENGTH, reverseOrderTimestamp);
return row;
}
}
```

Explanation

- The conversion takes advantage of the fact that the station ID is a fixed-length ASCII string.
- Like in the earlier example, we use HBase's Bytes class for converting between byte arrays and common Java types.
- The putBytes() and putLong() methods are used to fill the station ID and timestamp portions of the key at the relevant offsets in the byte array.
- The job is configured in the run() method to use HBase's TableOutputFormat.
- The table to write to must be specified by setting the TableOutputFormat.OUTPUT_TABLE property in the job configuration.

31. Web Queries

- To implement the online query application, we will use the HBase Java API directly.
- Here it becomes clear how important your choice of schema and storage format is.

Station queries

The simplest query will be to get the static station information. This is a single row lookup, performed using a get() operation. This type of query is simple in a traditional database, but HBase gives you additional control and flexibility. Using the info family as a key-value dictionary (column names as keys, column values as values), the code from HBaseStationQuery looks like this:

```
static final byte[] INFO_COLUMNFAMILY = Bytes.toBytes("info");
static final byte[] NAME_QUALIFIER = Bytes.toBytes("name");
static final byte[] LOCATION_QUALIFIER = Bytes.toBytes("location");
static final byte[] DESCRIPTION_QUALIFIER = Bytes.toBytes("description");
public Map<String, String> getStationInfo(HTable table, String stationId)
throws IOException {
Get get = new Get(Bytes.toBytes(stationId));
get.addFamily(INFO_COLUMNFAMILY);
Result res = table.get(get);
if (res == null) {
return null;
}
Map<String, String> resultMap = new LinkedHashMap<String, String>();
resultMap.put("name", getValue(res, INFO_COLUMNFAMILY, NAME_QUALIFIER));
resultMap.put("location", getValue(res, INFO_COLUMNFAMILY,
LOCATION_QUALIFIER));
resultMap.put("description", getValue(res, INFO_COLUMNFAMILY,
DESCRIPTION_QUALIFIER));
return resultMap;
}
private static String getValue(Result res, byte[] cf, byte[] qualifier) {
byte[] value = res.getValue(cf, qualifier);
return value == null? "": Bytes.toString(value);
}
```

Explanation

- In this example, getStationInfo() takes an HTable instance and a station ID.
- To get the station info, we use get(), passing a Get instance configured to retrieve all the column values for the row identified by the station ID in the defined column family, INFO_COLUMNFAMILY.
- The get() results are returned in a Result.
- It contains the row, and you can fetch cell values by stipulating the column cell you want.
- The getStationInfo() method converts the Result into a more friendly Map of String keys and values.

Problems to Ponder

1. HIVE word count program

Step 1. Create a table in hive

hive> create table hiveword (comments string);

Step 2. Load data from the sample file

hive> load data local inpath '/home/cloudera/Desktop/hiveword.txt' into table hiveword;

Step 3. Convert comments into an array of strings

hive> select split (comments, ' ') from hiveword;

OK

["This","is","BDA","LAB"]

["WE","ARE","WORKING","IN","HIVE",""]

["BDA","LAB"]

Time taken: 0.099 seconds, Fetched: 3 row(s)

Step 4. Use table generation udf

//Now time to return multiple row from the above array of string, and for that we have built in table generation UDTF explode function.//

hive> select explode(split(comments, ' ')) from hiveword;

OK

This

is

BDA

LAB

WE

ARE

WORKING

IN

HIVE

BDA

LAB

Time taken: 0.104 seconds, Fetched: 12 row(s)

Step 5 final step

```
hive> SELECT word,count(*) from (select explode(split(comments, ' ')) as word from hiveword)
    w
   > GROUP BY word;
```

Output:

```
Query ID = cloudera_20230503074242_3bdf4293-17b4-4798-bd4f-ae5588c40bc4
Total jobs = 1
Launching Job 1 out of 1
Number of reduce tasks not specified. Estimated from input data size: 1
In order to change the average load for a reducer (in bytes):
  set hive.exec.reducers.bytes.per.reducer=<number>
In order to limit the maximum number of reducers:
  set hive.exec.reducers.max=<number>
In order to set a constant number of reducers:
  set mapreduce.job.reduces=<number>
Starting Job = job_1683122816047_0001, Tracking URL =
    http://quickstart.cloudera:8088/proxy/application_1683122816047_0001/
Kill Command = /usr/lib/hadoop/bin/hadoop job  -kill job_1683122816047_0001
Hadoop job information for Stage-1: number of mappers: 1; number of reducers: 1
2023-05-03 07:42:37,655 Stage-1 map = 0%,  reduce = 0%
2023-05-03 07:42:57,664 Stage-1 map = 100%,  reduce = 0%, Cumulative CPU 3.14 sec
2023-05-03 07:43:18,077 Stage-1 map = 100%,  reduce = 100%, Cumulative CPU 6.36 sec
MapReduce Total cumulative CPU time: 6 seconds 360 msec
Ended Job = job_1683122816047_0001
MapReduce Jobs Launched:
Stage-Stage-1: Map: 1  Reduce: 1   Cumulative CPU: 6.36 sec   HDFS Read: 8386 HDFS Write:
    60 SUCCESS
```

Total MapReduce CPU Time Spent: 6 seconds 360 msec

OK

1

ARE 1

BDA 2

HIVE 1

IN 1

LAB 2

This 1

WE 1

WORKING 1

is 1

Time taken: 76.882 seconds, Fetched: 10 row(s)

2. Word count program for Pig

Step1: Use Load statement to load the data into a relation .As keyword used to declare column names, as we dont have any columns, we declared only one column named line.

grunt> input1 = LOAD '/home/cloudera/Desktop/pigword.txt' as (line:Chararray);

Step 2&3:

2. Convert the Sentence into words.

The data we have is in sentences. So we have to convert that data into words using TOKENIZE Function.

3.Convert Column into Rows

We have to convert every line of data into multiple rows ,for this we have function called FLATTEN in pig.

grunt> Words = FOREACH input1 GENERATE FLATTEN(TOKENIZE(line,' '))AS word;

4. Apply GROUP BY

We have to count each word occurance, for that we have to group all the words.

grunt> Grouped =GROUP Words BY word;

5. Generate word count

grunt> wordcount = FOREACH Grouped GENERATE group, COUNT(Words);

6.Print the word count on console using Dump.

grunt> DUMP wordcount;

OUTPUT:

(Mary, had, a, little, lamb,,1)
(its, fleece, was, white, as, snow.,1)