Module – 5 INTRODUCTIONS TO PIG AND HIVE

Pig raises the level of abstraction for processing large datasets. MapReduce allows you the programmer to specify a map function followed by a reduce function, but working out how to fit your data processing into this pattern, which often requires multiple MapReduce stages, can be a challenge. With Pig, the data structures are much richer, typically being multivalued and nested; and the set of transformations you can apply to the data are much more powerful they include joins, for example, which are not for the faint of heart in MapReduce.

Pig is made up of two pieces:

- The language used to express data flows, called *Pig Latin*.
- The execution environment to run Pig Latin programs. There are currently two environments: local execution in a single JVM and distributed execution on a Ha- doop cluster.

A Pig Latin program is made up of a series of operations, or transformations, that are applied to the input data to produce output. Taken as a whole, the operations describe a data flow, which the Pig execution environment translates into an executable repre- sentation and then runs. Under the covers, Pig turns the transformations into a series of MapReduce jobs, but as a programmer you are mostly unaware of this, which allows you to focus on the data rather than the nature of the execution.

Pig is a scripting language for exploring large datasets. One criticism of MapReduce is that the development cycle is very long. Writing the mappers and reducers, compiling and packaging the code, submitting the job(s), and retrieving the results is a time- consuming business, and even with Streaming, which removes the compile and package step, the experience is still involved. Pig's sweet spot is its ability to process terabytes of data simply by issuing a half-dozen lines of Pig Latin from the console. Indeed, it was created at Yahoo! to make it easier for researchers and engineers to mine the huge datasets there. Pig is very supportive of a programmer writing a query, since it provides several commands for introspecting the data structures in your program, as it is written. Even more useful, it can perform a sample run on a representative subset of your input data, so you can see whether there are errors in the processing before unleashing it on the full dataset.

Pig was designed to be extensible. Virtually all parts of the processing path are cus- tomizable: loading, storing, filtering, grouping, and joining can all be altered by user- defined functions (UDFs). These functions operate on Pig's nested data model, so they can integrate very deeply with Pig's operators. As another benefit, UDFs tend to be more reusable than the libraries developed for writing MapReduce programs.

Pig isn't suitable for all data processing tasks, however. Like MapReduce, it is designed for batch processing of data. If you want to perform a query that touches only a small amount of data in a large dataset, then Pig will not perform well, since it is set up to scan the whole dataset, or at least large portions of it.

In some cases, Pig doesn't perform as well as programs written in MapReduce. How- ever, the gap is narrowing with each release, as the Pig team implements sophisticated algorithms for implementing Pig's relational operators. It's fair to say that unless you are willing to invest a lot of effort optimizing Java MapReduce code, writing queries in Pig Latin will save you time.

INSTALLING AND RUNNING PIG

Pig runs as a client-side application. Even if you want to run Pig on a Hadoop cluster, there is nothing extra to install on the cluster: Pig launches jobs and interacts with HDFS (or other Hadoop filesystems) from your workstation.

Installation is straightforward. Java 6 is a prerequisite (and on Windows, you will need Cygwin). Download a stable release from *http://pig.apache.org/releases.html*, and unpack the tarball in a suitable place on your workstation:

% tar xzf pig-*x.y.z*.tar.gz

It's convenient to add Pig's binary directory to your command-line path. For example:

% export PIG_INSTALL=/home/tom/pig-*x.y.z*

% export PATH=$PATH:$PIG_INSTALL/bin

You also need to set the JAVA_HOME environment variable to point to a suitable Java installation.

Try typing pig -help to get usage instructions.

EXECUTION TYPES

Pig has two execution types or modes: local mode and MapReduce mode.

LOCAL MODE

In local mode, Pig runs in a single JVM and accesses the local filesystem. This mode is suitable only for small datasets and when trying out Pig.

The execution type is set using the -x or -exectype option. To run in local mode, set the option to local:

% pig -x local

grunt>

This starts Grunt, the Pig interactive shell, which is discussed in more detail shortly.

MAPREDUCE MODE

In MapReduce mode, Pig translates queries into MapReduce jobs and runs them on a Hadoop cluster. The cluster may be a pseudo- or fully distributed cluster. MapReduce mode (with a fully distributed cluster) is what you use when you want to run Pig on large datasets.

To use MapReduce mode, you first need to check that the version of Pig you down- loaded is compatible with the version of Hadoop you are using. Pig releases will only work against particular versions of Hadoop; this is documented in the release notes.

Pig honors the HADOOP_HOME environment variable for finding which Hadoop client to run. However if it is not set, Pig will use a bundled copy of the Hadoop libraries. Note that these may not match the version of Hadoop running on your cluster, so it is best to explicitly set HADOOP_HOME.

Next, you need to point Pig at the cluster's namenode and jobtracker. If the installation of Hadoop at HADOOP_HOME is already configured for this, then there is nothing more to do. Otherwise, you can set HADOOP_CONF_DIR to a directory containing the Hadoop site file (or files) that define fs.default.name and mapred.job.tracker.

Alternatively, you can set these two properties in the *pig.properties* file in Pig's *conf* directory (or the directory specified by PIG_CONF_DIR). Here's an example for a pseudo- distributed setup:

fs.default.name=hdfs://localhost/ mapred.job.tracker=localhost:8021

Once you have configured Pig to connect to a Hadoop cluster, you can launch Pig, setting the -x option to mapreduce, or omitting it entirely, as MapReduce mode is the default:

% **pig**

2012-01-18 20:23:05,764 [main] INFO org.apache.pig.Main - Logging error message s to: /private/tmp/pig_1326946985762.log

2012-01-18 20:23:06,009 [main] INFO org.apache.pig.backend.hadoop.executionengi ne.HExecutionEngine - Connecting to hadoop file system at: hdfs://localhost/ 2012-01-18 20:23:06,274 [main] INFO org.apache.pig.backend.hadoop.executionengi ne.HExecutionEngine - Connecting to map-reduce job tracker at: localhost:8021 grunt>

As you can see from the output, Pig reports the filesystem and jobtracker that it has connected to.

RUNNING PIG PROGRAMS

There are three ways of executing Pig programs, all of which work in both local and MapReduce mode:

Script

Pig can run a script file that contains Pig commands. For example, pig script.pig runs the commands in the local file *script.pig*. Alternatively, for very short scripts, you can use the -e option to run a script specified as a string on the command line.

Grunt

Grunt is an interactive shell for running Pig commands. Grunt is started when no file is specified for Pig to run, and the -e option is not used. It is also possible to run Pig scripts from within Grunt using run and exec.

Embedded

You can run Pig programs from Java using the PigServer class, much like you can use JDBC to run SQL programs from Java. For programmatic access to Grunt, use PigRunner.

Pig Latin Editors

PigPen is an Eclipse plug-in that provides an environment for developing Pig programs. It includes a Pig script text editor, an example generator (equivalent to the ILLUS- TRATE command), and a button for running the script on a Hadoop cluster. There is also an operator graph window, which shows a script in graph form, for visualizing the data flow. For full installation and usage instructions, please refer to the Pig wiki at *https://cwiki.apache.org/confluence/display/PIG/PigTools*.

There are also Pig Latin syntax highlighters for other editors, including Vim and Text- Mate. Details are available on the Pig wiki.

An Example

Let's look at a simple example by writing the program to calculate the maximum recorded temperature by year for the weather dataset in Pig Latin (just like we did using MapReduce. The complete program is only a few lines long:

```
-- 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 == 0 OR quality == 1 OR quality == 4 OR quality == 5 OR quality == 9);
grouped_records = GROUP filtered_records BY year;

max_temp = FOREACH grouped_records GENERATE group,
```

MAX(filtered_records.temperature);

DUMP max_temp;

To explore what's going on, we'll use Pig's Grunt interpreter, which allows us to enter lines and interact with the program to understand what it's doing. Start up Grunt in local mode, then enter the first line of the Pig script:

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

For simplicity, the program assumes that the input is tab-delimited text, with each line having just year, temperature, and quality fields. (Pig actually has more flexibility than this with regard to the input formats it accepts, as you'll see later.) This line describes the input data we want to process. The year:chararray notation describes the field's name and type; a chararray is like a Java string, and an int is like a Java int. The LOAD operator takes a URI argument; here we are just using a local file, but we could refer to an HDFS URI. The AS clause (which is optional) gives the fields names to make it convenient to refer to them in subsequent statements.

PIG LATIN

This section gives an informal description of the syntax and semantics of the Pig Latin programming language.[3] It is not meant to offer a complete reference to the language,[4] but there should be enough here for you to get a good understanding of Pig Latin's constructs.

STRUCTURE

A Pig Latin program consists of a collection of statements. A statement can be thought of as an operation, or a command.[5] For example, a GROUP operation is a type of statement:

The command to list the files in a Hadoop filesystem is another example of a statement:

ls /

Statements are usually terminated with a semicolon, as in the example of the GROUP statement. In fact, this is an example of a statement that must be terminated with a semicolon: it is a syntax error to omit it. The ls command, on the other hand, does not have to be terminated with a semicolon. As a general guideline, statements or com- mands for interactive use in

Grunt do not need the terminating semicolon. This group includes the interactive Hadoop commands, as well as the diagnostic operators like DESCRIBE. It's never an error to add a terminating semicolon, so if in doubt, it's sim- plest to add one.

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 single-line comments. Everything from the first hyphen to the end of the line is ignored by the Pig Latin interpreter:

```
-- 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;
```

Pig Latin has a list of keywords that have a special meaning in the language and cannot be used as identifiers. These include the operators (LOAD, ILLUSTRATE), commands (cat, ls), expressions (matches, FLATTEN), and functions (DIFF, MAX)—all of which are covered in the following sections.

Pig Latin has mixed rules on case sensitivity. Operators and commands are not case- sensitive (to make interactive use more forgiving); however, aliases and function names are case-sensitive.

TYPES

So far you have seen some of the simple types in Pig, such as int and chararray. Here we will discuss Pig's built-in types in more detail.

Pig hasfournumerictypes: int, long, float, anddouble, whichareidenticaltotheir Java counterparts. There is also a byte array type, like Java's byte array type for representing a blob of binary data, and char array, which, like java.lang. String, represents textual data in UTF-16 format, although it can be loaded or stored in UTF-8 format. Pig does not have types corresponding to Java's boolean, byte, short, or char primitive types. These are all easily represented using Pig's int type, orchar array forchar.

The numeric, textual, and binary types are simple atomic types. Pig Latin also has three complex types for representing nested structures: tuple, bag, and map.

The complex types are usually loaded from files or constructed using relational operators. Be aware, however, that the literal form is used when a constant value is created from within a

Pig Latin program. The raw form in a file is usually different when using the standard Pig

Storage loader. For example, the representation in a file of the bag would be {(1,pomegranate),(2)} (note the lack of quotes), and with a suitable schema, this would be loaded as a relation with a single field and row, whose value was the bag.

Pig provides built-in functions TOTUPLE, TOBAG and TOMAP, which are used for turning expressions into tuples, bags and maps.

Although relations and bags are conceptually the same (an unordered collection of tuples), in practice Pig treats them slightly differently. A relation is a top-level construct, whereas a bag has to be contained in a relation. Normally, you don't have to worry about this, but there are a few restrictions that can trip up the uninitiated. For example, it's not possible to create a relation from a bag literal. So the following statement fails:

```
A = {(1,2),(3,4)}; -- Error
```

The simplest workaround in this case is to load the data from a file using the LOAD statement.

As another example, you can't treat a relation like a bag and project a field into a new relation ($0 refers to the first field of A, using the positional notation):

```
B = A.$0;
```

Instead, you have to use a relational operator to turn the relation A into relation B:

```
B = FOREACH A GENERATE $0;
```

It's possible that a future version of Pig Latin will remove these inconsistencies and treat relations and bags in the same way.

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. Furthermore, many aggregate functions are *algebraic*, which means that the result of the function may be calculated incrementally. In MapReduce terms, algebraic functions make use of the combiner and are much more efficient to calculate MAX is an algebraic function, whereas a function to calculate the median of a collectionof values is an example of a function that is not algebraic

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. They can also be used in other relational operators that take boolean con- ditions and, in general, expressions using boolean or conditional expressions. An example of a built-in filter function is IsEmpty, which tests whether a bag or a map contains any items.

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.

Pig comes with a collection of built-in functions. The complete list of built-in functions, which includes a large number of standard math and string functions, can be found in the documentation for each Pig release.

If the function you need is not available, you can write your own. Before you do that, however, have a look in the *Piggy Bank*, a repository of Pig functions shared by the Pig community. For example, there are load and store functions in the Piggy Bank for Avro data files, CSV files, Hive RCFiles, Sequence Files, and XML files. The Pig website has instructions on how to browse and obtain the Piggy Bank functions. If the Piggy Bank doesn't have what you need, you can write your own function (and if it is sufficiently general, you might consider contributing it to the Piggy Bank so that others can benefit from it, too). These are known as *user-defined functions*, or UDFs.

DATA PROCESSING OPERATORS

Loading and Storing Data

Throughout this chapter, we have seen how to load data from external storage for processing in Pig. Storing the results is straightforward, too. Here's an 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
```

Filtering Data

Once you have some data loaded into a relation, the next step is often to filter it to remove the data that you are not interested in. By filtering early in the processing pipe- line, you minimize the amount of data flowing through the system, which can improve efficiency.

FOREACH...GENERATE

We have already seen how to remove rows from a relation using the FILTER operator with simple expressions and a UDF. 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)

Here we have created a new relation B with three fields. Its first field is a projection of the first field ($0) of A. B's second field is the third field of A ($2) with one added to it. B's third field is a constant field (every row in B has the same third field) with the chararray value Constant.

The FOREACH...GENERATE operator has a nested form to support more complex processing. In the following example, we compute various statistics for the weather dataset:

```
-- year_stats.pig

REGISTER pig-examples.jar;

DEFINE isGood com.hadoopbook.pig.IsGoodQuality(); records = LOAD
'input/ncdc/all/19{1,2,3,4,5}0*'

USING com.hadoopbook.pig.CutLoadFunc('5-10,11-15,16-19,88-92,93-93')

AS (usaf:chararray, wban:chararray, year:int, temperature:int, quality:int);
grouped_records = GROUP records BY year PARALLEL 30;

year_stats = FOREACH grouped_records { uniq_stations = DISTINCT records.usaf;
good_records = FILTER records BY isGood(quality);

GENERATE FLATTEN(group), COUNT(uniq_stations) AS station_count,
```

COUNT(good_records) AS good_record_count, COUNT(records) AS record_count;

}

DUMP year_stats;

Using the cut UDF we developed earlier, we load various fields from the input dataset into the records relation. Next we group records by year. Notice the PARALLEL key- word for setting the number of reducers to use; this is vital when running on a cluster. Then we process each group using a nested FOREACH...GENERATE operator. The first nested statement creates a relation for the distinct USAF identifiers for stations using the DISTINCT operator. The second nested statement creates a relation for the

records with "good" readings using the FILTER operator and a UDF. The final nested statement is a GENERATE statement (a nested FOREACH...GENERATE must always have a GENERATE statement as the last nested statement) that generates the summary fields of interest using the grouped records, as well as the relations created in the nested block.

Running it on a few years of data, we get the following:

(1920,8L,8595L,8595L) (1950,1988L,8635452L,8641353L) (1930,121L,89245L,89262L) (1910,7L,7650L,7650L) (1940,732L,1052333L,1052976L)

The fields are year, number of unique stations, total number of good readings, and total number of readings. We can see how the number of weather stations and readings grew over time.

STREAM

The STREAM operator allows you to transform data in a relation using an external program or script. It is named by analogy with Hadoop Streaming, which provides a similar capability for MapReduce.

STREAM can use built-in commands with arguments. Here is an example that uses the Unix cut command to extract the second field of each tuple in A. Note that the com- mand and its arguments are enclosed in backticks:

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

grunt> **DUMP C;** (cherry) (apple) (banana) (apple)

The STREAM operator uses PigStorage to serialize and deserialize relations to and from the program's standard input and output streams. Tuples in A are converted to tab- delimited lines that are passed to the script. The output of the script is read one line at a time and split on tabs

to create new tuples for the output relation C. You can provide a custom serializer and deserializer, which implement PigToStream and StreamToPig respectively (both in the org.apache.pig package), using the DEFINE command.

Pig streaming is most powerful when you write custom processing scripts. The follow- ing Python script filters out bad weather records:

```
#!/usr/bin/env python

import re import sys
```

```
for line in sys.stdin:

(year, temp, q) = line.strip().split()

if (temp != "9999" and re.match("[01459]", q)):

print "%s\t%s" % (year, temp)
```

To use the script, you need to ship it to the cluster. This is achieved via a DEFINE clause, which also creates an alias for the STREAM command. The STREAM statement can then refer to the alias, as the following Pig script shows:

```
-- max_temp_filter_stream.pig

DEFINE is_good_quality `is_good_quality.py`

SHIP ('ch11/src/main/python/is_good_quality.py'); records = LOAD 'input/ncdc/micro-tab/sample.txt'

AS (year:chararray, temperature:int, quality:int); filtered_records = STREAMrecords THROUGH is_good_quality

AS (year:chararray, temperature:int); grouped_records = GROUP filtered_records BY year; max_temp = FOREACH grouped_records GENERATE group,

MAX(filtered_records.temperature); DUMP max_temp;
```

Grouping and Joining Data

Joining datasets in MapReduce takes some work on the part of the, whereas Pig has very good built-in support for join operations, making it much more approachable. Since the large datasets that are suitable for analysis by Pig (and MapReduce in general) are not normalized, joins are used more infrequently in Pig than they are in SQL.

JOIN

Let's look at an example of an inner join. Consider the relations A and B: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,Joe,2)

(2,Tie,Hank,2)

(3,Hat,Eve,3)

(4,Coat,Hank,4)

This is a classic inner join, where each match between the two relations corresponds to a row in the result. (It's actually an equijoin since the join predicate is equality.) The result's fields are made up of all the fields of all the input relations.

You should use the general join operator if all the relations being joined are too large to fit in memory. If one of the relations is small enough to fit in memory, there is 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 using an in-memory lookup table against the (fragmented) larger relation. There is a special syntax for telling Pig to use a fragment replicate join:[8]

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

The first relation must be the large one, followed by one or more small ones (all of which mustfit in memory).

Pig also supports outer joins using a syntax that is similar to SQL's. For example:grunt>

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

grunt> **DUMP C;**

(1,Scarf,,)

(2,Tie,Joe,2)

(2,Tie,Hank,2)

(3,Hat,Eve,3)
(4,Coat,Hank,4)

COGROUP

JOIN always gives a flat structure: a set of tuples. The COGROUP statement is similar to JOIN, but creates a nested set of output tuples. This can be useful if you want to exploit the structure in subsequent statements:

grunt> D = COGROUP A BY $0, B BY $1;

grunt> **DUMP D;**

(0,{},{(Ali,0)})

(1,{(1,Scarf)},{}) (2,{(2,Tie)},{(Joe,2),(Hank,2)})(3,{(3,Hat)},{(Eve,3)})

(4,{(4,Coat)},{(Hank,4)})

COGROUP generates a tuple for each unique grouping key. The first field of each tuple is the key, and the remaining fields are bags of tuples from the relations with a matching key. The first bag contains the matching tuples from relation A with the same key. Similarly, the second bag contains the matching tuples from relation B with the same key.

If for a particular key a relation has no matching key, then the bag for that relation is empty. For example, since no one has bought a scarf (with ID 1), the second bag in the tuple for that row is empty. This is an example of an outer join, which is the default type for COGROUP. It can be made explicit using the OUTER keyword, making this COGROUP statement the same as the previous one:

D = COGROUP A BY $0 **OUTER**, B BY $1 **OUTER;**

You can suppress rows with empty bags by using the INNER keyword, which gives the COGROUP inner join semantics. The INNER keyword is applied per relation, so the following only suppresses rows when relation A has no match (dropping the unknown product 0 here):

grunt> E = COGROUP A BY $0 INNER, B BY $1;

grunt> **DUMP E;** (1,{(1,Scarf)},{}) (2,{(2,Tie)},{(Joe,2),(Hank,2)})

(3,{(3,Hat)},{(Eve,3)})

(4,{(4,Coat)},{(Hank,4)})

We can flatten this structure to discover who bought each of the items in relation A:grunt>

F = FOREACH E GENERATE FLATTEN(A), B.$0;

grunt> **DUMP F;** (1,Scarf,{}) (2,Tie,{(Joe),(Hank)})

(3,Hat,{(Eve)})

(4,Coat,{(Hank)})

Using a combination of COGROUP, INNER, and FLATTEN (which removes nesting) it's possible to simulate an (inner) JOIN:

grunt> G = COGROUP A BY $0 INNER, B BY $1 INNER;

grunt> **H = FOREACH G GENERATE FLATTEN($1), FLATTEN($2);**

grunt> **DUMP H;**

(2,Tie,Joe,2)

(2,Tie,Hank,2)

(3,Hat,Eve,3)

(4,Coat,Hank,4)

This gives the same result as JOIN A BY $0, B BY $1.

If the join key is composed of several fields, you can specify them all in the BY clauses of the JOIN or COGROUP statement. Make sure that the number of fields in each BY clause is the same.

Here's another example of a join in Pig, in a script for calculating the maximum tem- perature for every station over a time period controlled by the input:

-- max_temp_station_name.pig REGISTER pig-examples.jar;

DEFINE isGood com.hadoopbook.pig.IsGoodQuality();

stations = LOAD 'input/ncdc/metadata/stations-fixed-width.txt' USING com.hadoopbook.pig.CutLoadFunc('1-6,8-12,14-42')

AS (usaf:chararray, wban:chararray, name:chararray);

trimmed_stations = FOREACH stations GENERATE usaf, wban, com.hadoopbook.pig.Trim(name);

records = LOAD 'input/ncdc/all/191*'

USING com.hadoopbook.pig.CutLoadFunc('5-10,11-15,88-92,93-93') AS

(usaf:chararray, wban:chararray, temperature:int, quality:int);

filtered_records = FILTER records BY temperature != 9999 AND isGood(quality); grouped_records = GROUP filtered_records BY (usaf, wban) PARALLEL 30; max_temp = FOREACH grouped_records GENERATE FLATTEN(group),

MAX(filtered_records.temperature);

max_temp_named = JOIN max_temp BY (usaf, wban), trimmed_stations BY (usaf,wban)

PARALLEL 30;

max_temp_result = FOREACH max_temp_named GENERATE $0, $1, $5, $2; STORE

max_temp_result INTO 'max_temp_by_station';

We use the cut UDF we developed earlier to load one relation holding the station IDs (USAF and WBAN identifiers) and names, and one relation holding all the weather records, keyed by station ID. We group the filtered weather records by station ID and aggregate by maximum temperature, before joining with the stations. Finally, we project out the fields we want in the final result: USAF, WBAN, station name, maxi- mum temperature.

This query could be made more efficient by using a fragment replicate join, as the station metadata is small.

CROSS

Pig Latin includes the cross-product operator (also known as the cartesian product), which joins every tuple in a relation with every tuple in a second relation (and with every tuple in further relations if supplied). The size of the output is the product of the size of the inputs, potentially making the output very large:

grunt> **I = CROSS A, B;**

grunt> **DUMP I;**

(2,Tie,Joe,2)

(2,Tie,Hank,4)

(2,Tie,Ali,0)

(2,Tie,Eve,3)

(2,Tie,Hank,2)

(4,Coat,Joe,2)

(4,Coat,Hank,4)

(4,Coat,Ali,0)

(4,Coat,Eve,3)
(4,Coat,Hank,2)

(3,Hat,Joe,2)

(3,Hat,Hank,4)

(3,Hat,Ali,0)

(3,Hat,Eve,3)

(3,Hat,Hank,2)

(1,Scarf,Joe,2)

(1,Scarf,Hank,4)

(1,Scarf,Ali,0)

(1,Scarf,Eve,3)

(1,Scarf,Hank,2)

When dealing with large datasets, you should try to avoid operations that generate intermediate representations that are quadratic (or worse) in size. Computing the cross- product of the whole input dataset is rarely needed, if ever.

For example, at first blush one might expect that calculating pairwise document simi- larity in a corpus of documents would require every document pair to be generated befor e calculating their similarity. However, if one starts with the insight that most document pairs have a similarity score of zero (that is, they are unrelated), then we can find a way to a better algorithm.

In this case, the key idea is to focus on the entities that we are using to calculate similarity (terms in a document, for example) and make them the center of the algorithm. In practice, we also remove terms that don't help discriminate between documents (stop- words), and this reduces the problem space still further. Using this technique to analyze a set of roughly one million
(10^6) documents generates in the order of one billion

(10^9) intermediate pairs,[9] rather than the one trillion (10^{12}) produced by the naive approach

(generating the cross-product of the input) or the approach with no stopword removal. **GROUP**

Although COGROUP groups the data in two or more relations, the GROUP statement groups the data in a single relation. GROUP supports grouping by more than equality of keys: you can use an expression or user-defined function as the group key. For ex- ample, consider the following relation A:
grunt> **DUMP A;** (Joe,cherry) (Ali,apple) (Joe,banana) (Eve,apple)Let's

group by the number of characters in the second field:

grunt> **B = GROUP A BY SIZE($1);**

grunt> **DUMP B;**

(5,{(Ali,apple),(Eve,apple)})

(6,{(Joe,cherry),(Joe,banana)})

GROUP creates a relation whose first field is the grouping field, which is given the alias group. The second field is a bag containing the grouped fields with the same schema as the original relation (in this case, A).

There are also two special grouping operations: ALL and ANY. ALL groups all the tuples in a relation in a single group, as if the GROUP function was a constant:

grunt> **C = GROUP A ALL;**

grunt> **DUMP C;**

(all,{(Joe,cherry),(Ali,apple),(Joe,banana),(Eve,apple)})

Note that there is no BY in this form of the GROUP statement. The ALL grouping is commonly used to count the number of tuples in a relation.
The ANY keyword is used to group the tuples in a relation randomly, which can be useful for sampling.

Sorting Data

Relations are unordered in Pig. Consider a relation A:

grunt> **DUMP A;**

(2,3)

(1,2)

(2,4)

There is no guarantee which order the rows will be processed in. In particular, when retrieving the contents of A using DUMP or STORE, the rows may be written in any order. If you want to impose an order on the output, you can use the ORDER operator to sort a relation by one or more fields. The default sort order compares fields of the same type using the natural ordering

and different types are given an arbitrary, but deterministic, ordering (a tuple is always "less than" a bag, for example).

The following example sorts A by the first field in ascending order and by the second field in descending order:

grunt> **B = ORDER A BY $0, $1 DESC;**

grunt> **DUMP B;**

(1,2)

(2,4)

(2,3)

Any further processing on a sorted relation is not guaranteed to retain its order. For example:grunt> **C**

= FOREACH B GENERATE *;

Even though relation C has the same contents as relation B, its tuples may be emitted in any order by a DUMP or a STORE. It is for this reason that it is usual to perform the ORDER operation just before retrieving the output.

The LIMIT statement is useful for limiting the number of results, as a quick and dirty way to get a sample of a relation; prototyping (the ILLUSTRATE command) should be preferred for generating more representative samples of the data. It can be used imme- diately after the ORDER statement to retrieve the first *n* tuples. Usually, LIMIT will select any *n* tuples from a relation, but when used immediately after an ORDER state- ment, the order is retained (in an exception to the rule that processing a relation does not retain its order):

grunt> **D = LIMIT B 2;**

grunt> **DUMP D;**

(1,2)

(2,4)

If the limit is greater than the number of tuples in the relation, all tuples are returned (so LIMIT has no effect).

Using LIMIT can improve the performance of a query because Pig tries to apply the limit as early as possible in the processing pipeline, to minimize the amount of data that needs to be processed. For this reason, you should always use LIMIT if you are not interested in the entire output.

Combining and Splitting Data

Sometimes you have several relations that you would like to combine into one. For this, the UNION statement is used. For example:

grunt> **DUMP A;**

(2,3)

(1,2)

(2,4)

grunt> **DUMP B;**

(z,x,8)

(w,y,1)

grunt> **C = UNION A, B;**

grunt> **DUMP C;**

(2,3)

(1,2)

(2,4)

(z,x,8)

(w,y,1)

C is the union of relations A and B, and since relations are unordered, the order of the tuplesin C is undefined. Also, it's possible to form the union of two relations with different schemas or with different numbers of fields, as we have done here. Pig attempts to merge the schemas fromthe relations that UNION is operating on. In this case, they are incompatible, so C has no schema:

grunt> **DESCRIBE A;** A: {f0: int,f1: int} grunt> **DESCRIBE B;**B: {f0:

chararray,f1: chararray,f2: int} grunt> **DESCRIBE C;** Schema for C unknown.

If the output relation has no schema, your script needs to be able to handle tuples that vary inthe number of fields and/or types.

The SPLIT operator is the opposite of UNION; it partitions a relation into two or morerelations..

Pig in Practice

There are some practical techniques that are worth knowing about when you are developing and running Pig programs. This section covers some of them.

Parallelism

When running in MapReduce mode it's important that the degree of parallelism matches the size of the dataset. By default, Pig will sets the number of reducers by looking at the size of the input, and using one reducer per 1GB of input, up to a max- imum of 999 reducers.You can override these parameters by setting pig.exec.reduc ers.bytes.per.reducer (the default is 1000000000 bytes) and pig.exec.reducers.max (default 999).

To explictly 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 followingline sets the number of reducers to 30 for the GROUP:

grouped_records = GROUP records BY year PARALLEL 30;

Alternatively, you can set the default_parallel option, and it will take effect for all subsequent jobs:

grunt> **set default_parallel 30**

A good setting for the number of reduce tasks is slightly fewer than the number of reduce slotsin the cluster.

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.

Parameter Substitution

If you have a Pig script that you run on a regular basis, then it's quite common to want to be able to run the same script with different parameters. For example, a script that runs daily may use the date to determine which input files it runs over. 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; for example, $input and
$output are used in the following script to specify the input and output paths:

```
-- max_temp_param.pig

records = LOAD '$input' AS (year:chararray, temperature:int, quality:int);filtered_records =
FILTER records BY temperature != 9999 AND

(quality == 0 OR quality == 1 OR quality == 4 OR quality == 5 OR quality == 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, one for eachparameter:

```
% pig -param input=/user/tom/input/ncdc/micro-tab/sample.txt \
> -param output=/tmp/out \
> ch11/src/main/pig/max_temp_param.pig
```

You can also put parameters in a file and pass them to Pig using the -param_file option. For example, we can achieve the same result as the previous command by placing the parameter definitions in a file:

```
# Input file

input=/user/tom/input/ncdc/micro-tab/sample.txt # Output file

output=/tmp/out
```

The *pig* invocation then becomes:

```
% pig -param_file ch11/src/main/pig/max_temp_param.param \
```

- **ch11/src/main/pig/max_temp_param.pig**

You can specify multiple parameter files using -param_file repeatedly. You can also use a combination of -param and -param_file options, and if any parameter is defined in both a parameter file and on the command line, the last value on the command line takes precedence.

HIVE

In "Information Platforms and the Rise of the Data Scientist,"[1] Jeff Hammerbacher describes Information Platforms as "the locus of their organization's efforts to ingest, process, and generate information," and how they "serve to accelerate the process of learning from empirical data."

One of the biggest ingredients in the Information Platform built by Jeff's team at Face- book was Hive, a framework for data warehousing on top of Hadoop. Hive grew from a need to manage and learn from the huge volumes of data that Facebook was producing every day from its burgeoning social network. After trying a few different systems, the team chose Hadoop for storage and processing, since it was cost-effective and met their scalability needs.[2]

Hive was created to make it possible for analysts with strong SQL skills (but meager Java programming skills) to run queries on the huge volumes of data that Facebook stored in HDFS. Today, Hive is a successful Apache project used by many organizations as a general-purpose, scalable data processing platform.

Of course, SQL isn't ideal for every big data problem—it's not a good fit for building complex machine learning algorithms, for example—but it's great for many analyses, and it has the huge advantage of being very well known in the industry. What's more, SQL is the *lingua franca* in business intelligence tools (ODBC is a common bridge, for example), so Hive is well placed to integrate with these products.

This chapter is an introduction to using Hive. It assumes that you have working knowl- edge of SQL and general database architecture; as we go through Hive's features, we'll often compare them to the equivalent in a traditional RDBMS.

Installing Hive

In normal use, Hive runs on your workstation and converts your SQL query into a series of MapReduce jobs for execution on a Hadoop cluster. Hive organizes data into tables, which provide a means for attaching structure to data stored in HDFS. Metadata— such as table schemas—is stored in a database called the *metastore*.

When starting out with Hive, it is convenient to run the metastore on your local ma- chine. In this configuration, which is the default, the Hive table definitions that you create will be local to your machine, so you can't share them with other users. We'll see how to configure a shared remote metastore, which is the norm in production environments.

Installation of Hive is straightforward. Java 6 is a prerequisite; and on Windows, you will need Cygwin, too. You also need to have the same version of Hadoop installed locally that your cluster is running.[3] Of course, you may choose to run Hadoop locally, either in standalone or pseudo-distributed mode, while getting started with Hive. These options are all covered in Appendix A.

Download a release at *http://hive.apache.org/releases.html*, and unpack the tarball in asuitable place on your workstation:

% **tar xzf hive-*x.y.z*-dev.tar.gz**

It's handy to put Hive on your path to make it easy to launch:

% export HIVE_INSTALL=/home/tom/hive-*x.y.z*-dev

% **export PATH=$PATH:$HIVE_INSTALL/bin**

Now type hive to launch the Hive shell:

% **hive** hive>

Running Hive

In this section, we look at some more practical aspects of running Hive, including how to set up Hive to run against a Hadoop cluster and a shared metastore. In doing so, we'll see Hive's architecture in some detail.

Configuring Hive

Hive is configured using an XML configuration file like Hadoop's. The file is called *hive- site.xml* and is located in Hive's *conf* directory. This file is where you can set prop- erties that you want to set every time you run Hive. The same directory contains *hive- default.xml*, which documents the properties that Hive exposes and their default values.

You can override the configuration directory that Hive looks for in *hive-site.xml* by passing the --config option to the *hive* command:

% hive --config /Users/tom/dev/hive-conf

Note that this option specifies the containing directory, not *hive-site.xml* itself. It can be useful if you have multiple site files—for different clusters, say—that you switch between on a regular basis. Alternatively, you can set the HIVE_CONF_DIR environment variable to the configuration directory, for the same effect.

The *hive-site.xml* is a natural place to put the cluster connection details: you can specify the filesystem and jobtracker using the usual Hadoop properties, fs.default. name and mapred.job.tracker (see Appendix A for more details on configuring Hadoop). If not set, they default to the local filesystem and the local (in-process) job runner—just like they do in Hadoop—which is very handy when trying out Hive on small trial datasets. Metastore configuration settings are com- monly found in *hive-site.xml*, too.

Hive also permits you to set properties on a per-session basis, by passing the

-hiveconf option to the *hive* command. For example, the following command sets the cluster (to a pseudo-distributed cluster) for the duration of the session:

```
% hive -hiveconf fs.default.name=localhost -hiveconf
mapred.job.tracker=localhost:8021
```

If you plan to have more than one Hive user sharing a Hadoop cluster, then you need to make the directories that Hive uses writable by all users. The following commands will create the directories and set their permissions appropriately:

```
% hadoop fs -mkdir /tmp
```

% hadoop fs -chmod a+w /tmp

```
% hadoop fs -mkdir /user/hive/warehouse
```

% hadoop fs -chmod a+w /user/hive/warehouse

If all users are in the same group, then permissions g+w are sufficient on the warehouse directory.

You can change settings from within a session, too, using the SET command. This is useful for changing Hive or MapReduce job settings for a particular query. For example, the following command ensures buckets are populated according to the table definition.

```
hive> SET hive.enforce.bucketing=true;
```

To see the current value of any property, use SET with just the property name:hive>

SET hive.enforce.bucketing;

hive.enforce.bucketing=true

By itself, SET will list all the properties (and their values) set by Hive. Note that the list will not include Hadoop defaults, unless they have been explicitly overridden in one of the ways covered in this section. Use SET -v to list all the properties in the system, including Hadoop defaults.

There is a precedence hierarchy to setting properties. In the following list, lower num- bers take precedence over higher numbers:

1. The Hive SET command
2. The command line -hiveconf option
3. *hive-site.xml*
4. *hive-default.xml*
5. *hadoop-site.xml* (or, equivalently, *core-site.xml*, *hdfs-site.xml*, and *mapred- site.xml*)
6. *hadoop-default.xml* (or, equivalently, *core-default.xml*, *hdfs-default.xml*, and *mapred-default.xml*)

DATA TYPES

Hive supports both primitive and complex data types. Primitives include numeric, boolean, string, and timestamp types. The complex data types include arrays, maps, and structs.. Note that the literals shown are those used from within HiveQL; they are not the serialized formused in the table's storage format.

Primitive types

Hive's primitive types correspond roughly to Java's, although some names are influ- enced by MySQL's type names (some of which, in turn, overlap with SQL-92). There are four signed integral types: TINYINT, SMALLINT, INT, and BIGINT, which are equivalent to Java's byte, short, int, and long primitive types, respectively; they are 1-byte, 2-byte, 4-byte, and 8-byte signed integers.

Hive's floating-point types, FLOAT and DOUBLE, correspond to Java's float and double, which are 32-bit and 64-bit floating point numbers. Unlike some databases, there is no option to control the number of significant digits or decimal places stored for floating point values.

Hive supports a BOOLEAN type for storing true and false values.

There is a single Hive data type for storing text, STRING, which is a variable-length character string. Hive's STRING type is like VARCHAR in other databases, although there is no declaration of the maximum number of characters to store with STRING. (The theoretical maximum size STRING that may be stored is 2GB, although in practice it may be inefficient to materialize such large values. Sqoop has large object support.

The BINARY data type is for storing variable-length binary data.

The TIMESTAMP data type stores timestamps with nanosecond precision. Hive comes with UDFs for converting between Hive timestamps, Unix timestamps (seconds since the Unix epoch), and strings, which makes most common date operations tractable. TIMESTAMP does not encapsulate a timezone, however the to_utc_timestamp and from_utc_timestamp functions make it possible to do timezone conversions.

Conversions

Primitive types form a hierarchy, which dictates the implicit type conversions that Hive will perform. For example, a TINYINT will be converted to an INT, if an expression ex-

pects an INT; however, the reverse conversion will not occur and Hive will return an error unless the CAST operator is used.

The implicit conversion rules can be summarized as follows. Any integral numeric type can be implicitly converted to a wider type. All the integral numeric types, FLOAT, and (perhaps surprisingly) STRING can be implicitly converted to DOUBLE. TINYINT, SMALL INT, and INT can all be converted to FLOAT. BOOLEAN types cannot be converted to any other type.

You can perform explicit type conversion using CAST. For example, CAST('1' AS INT) will convert the string '1' to the integer value 1. If the cast fails—as it does in CAST('X' AS INT), for example—then the expression returns NULL.

Complex types

Hive has three complex types: ARRAY, MAP, and STRUCT. ARRAY and MAP are like their namesakes in Java, while a STRUCT is a record type which encapsulates a set of named fields. Complex types permit an arbitrary level of nesting. Complex type declarations must specify the type of the fields in the collection, using an angled bracket notation, as illustrated in this table definition which has three columns, one for each complex type:

```
CREATE TABLE complex ( col1 ARRAY<INT>,col2

MAP<STRING, INT>,

col3 STRUCT<a:STRING, b:INT, c:DOUBLE>

);
```

If we load the table with one row of data for ARRAY, MAP, and STRUCT shown in the “Literal examples” then the following query demonstrates the field accessor operators for each type:

hive> SELECT col1[0], col2['b'], col3.c FROM complex;OPERATORS

AND FUNCTIONS

The usual set of SQL operators is provided by Hive: relational operators (such as x = 'a' for testing equality, x IS NULL for testing nullity, x LIKE 'a%' for pattern matching), arithmetic operators (such as x + 1 for addition), and logical operators (such as x OR y for logical OR). The operators match those in MySQL, which deviates from SQL-92 since || is logical OR, not string concatenation. Use the concat function for the latter in both MySQL and Hive.

Hive comes with a large number of built-in functions—too many to list here—divided into categories including mathematical and statistical functions, string functions, date functions (for operating on string representations of dates), conditional functions, aggregate functions, and functions for working with

XML (using the xpath function) and JSON.

You can retrieve a list of functions from the Hive shell by typing SHOW FUNCTIONS.[6] To get brief usage instructions for a particular function, use the DESCRIBE command:

hive> **DESCRIBE FUNCTION length;** length(str) -

Returns the length of str**USER-DEFINED**

FUNCTIONS

Sometimes the query you want to write can't be expressed easily (or at all) using the built-in functions that Hive provides. By writing a *user-defined function* (UDF), Hive makes it easy to plug in your own processing code and invoke it from a Hive query.

UDFs have to be written in Java, the language that Hive itself is written in. For other languages, consider using a SELECT TRANSFORM query, which allows you to stream data through a user-defined script.

There are three types of UDF in Hive: (regular) UDFs, UDAFs (user-defined aggregate functions), and UDTFs (user-defined table-generating functions). They differ in the numbers of rows that they accept as input and produce as output:

- A UDF operates on a single row and produces a single row as its output. Most functions,such as mathematical functions and string functions, are of this type.
- A UDAF works on multiple input rows and creates a single output row. Aggregate functions include such functions as COUNT and MAX.
- A UDTF operates on a single row and produces multiple rows—a table—as output.

Table-generating functions are less well known than the other two types, so let's look at an example. Consider a table with a single column, x, which contains arrays of strings. It's instructive to take a slight detour to see how the table is defined and populated:

CREATE TABLE arrays (x ARRAY<STRING>) ROW FORMAT DELIMITEDFIELDS

TERMINATED BY '\001'

COLLECTION ITEMS TERMINATED BY '\002';

Notice that the ROW FORMAT clause specifies that the entries in the array are delimited by Control-B characters. The example file that we are going to load has the following contents, where ^B is a representation of the Control-B character to make it suitable for printing:a^Bb

c^Bd^Be

After running a LOAD DATA command, the following query confirms that the data wasloaded correctly:

hive > **SELECT * FROM arrays;**

["a","b"] ["c","d","e"]

Next, we can use the explode UDTF to transform this table. This function emits a row for each entry in the array, so in this case the type of the output column y is STRING. The result is that the table is flattened into five rows:

```
hive > SELECT explode(x) AS y FROM arrays;
```

a b c d e

SELECT statements using UDTFs have some restrictions (such as not being able to re- trieve additional column expressions), which make them less useful in practice. For this reason, Hive supports LATERAL VIEW queries, which are more powerful.