**TOPICS COVERED IN THIS SESSION:**

**Unit-V: Data Flow and Analysis**

**Pig:**

Installing and Running Pig - Execution Types - Running Pig Programs - Grunt - Pig Latin Editors - An Example - Generating Examples - Pig Latin - Structure - Statements - Expressions - Types – Schemas - Functions - Macros - User-Defined Functions - A Filter UDF - An Eval UDF - A Load UDF - Data Processing Operators - Loading and Storing Data - Filtering Data - Grouping and Joining Data - Sorting Data - Combining and Splitting Data - Pig in Practice - Parallelism - Anonymous Relations Parameter Substitution

**INTRODUCTION**

Apache Pig raises the level of abstraction for processing large datasets. MapReduce

allows the programmer to specify a map function followed by a reduce function but working out how to fit your data processing into this pattern can be a challenge. With Pig, the data structures are much richer, typically being multivalued and nested, and the 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 Hadoop 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 representation 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 in response to a half-dozen lines of Pig Latin issued 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 customizable: 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.

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. Download a stable release from *http://pig.apache.org/*

*releases.html*, and unpack the tarball in a suitable place on your workstation:

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

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

% **export PIG\_HOME=~/sw/pig-*x.y.z***

% **export PATH=$PATH:$PIG\_HOME/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.

**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.

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. We’ve used the -brief option to stop timestamps from being logged:

**% pig -brief**

**Logging error messages to: /Users/tom/pig\_1414246949680.log**

**Default bootup file /Users/tom/.pigbootup not found**

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. We’ve used the -brief option to stop timestamps from being logged:

**% pig -brief**

**Logging error messages to: /Users/tom/pig\_1414246949680.log**

**Default bootup file /Users/tom/.pigbootup not found**

Pig reports the filesystem (but not the YARN resource manager) that it has connected to. In MapReduce mode, you can optionally enable *auto-local mode* (by setting pig.auto.local.enabled to true), which is an optimization that runs small jobs locally if the input is less than 100 MB (set by pig.auto.local.input.maxbytes, default 100,000,000) and no more than one reducer is being used.

**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.

**Grunt**

Grunt has line-editing facilities like those found in GNU.

* Readline (used in the bash shell and many other command-line applications). For instance, the Ctrl-E key combination will move the cursor to the end of the line.
* Grunt remembers command history.
* Another handy feature is Grunt’s completion mechanism, which will try to complete Pig Latin keywords and functions when you press the Tab key.

**grunt> a = foreach b ge**

If you press the Tab key at this point, ge will expand to generate, a Pig Latin keyword:

**grunt> a = foreach b generate**

You can customize the completion tokens by creating a file named *autocomplete* and placing it on Pig’s classpath (such as in the *conf* directory in Pig’s *install* directory) or in the directory you invoked Grunt from. The file should have one token per line, and tokens must not contain any whitespace. Matching is case sensitive. It can be very handy to add commonly used file paths (especially because Pig does not perform filename completion) or the names of any user-defined functions you have created.

* You can get a list of commands using the help command.
* When you’ve finished your Grunt session, you can exit with the quit command, or the equivalent shortcut **\q.**

**Data Flow and Analysis**

**Pig:**

Installing and Running Pig - Execution Types - Running Pig Programs - Grunt - Pig Latin Editors - An Example - Generating Examples - Pig Latin - Structure - Statements - Expressions - Types – Schemas - Functions - Macros - User-Defined Functions - A Filter UDF - An Eval UDF - A Load UDF - Data Processing Operators - Loading and Storing Data - Filtering Data - Grouping and Joining Data - Sorting Data - Combining and Splitting Data - Pig in Practice - Parallelism - Anonymous Relations Parameter Substitution

**PIG LATIN EDITORS**

There are Pig Latin syntax highlighters available for a variety of editors, including

Eclipse, IntelliJ IDEA, Vim, Emacs, and TextMate.

**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. 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 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;

* **To explore what’s going on, Start up Grunt in local mode, and 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.

The result of the LOAD operator, and indeed any operator in Pig Latin, is a *relation*, which is just a set of tuples. A *tuple* is just like a row of data in a database table, with multiple fields in a particular order. In this example, the LOAD function produces a set of (year, temperature, quality) tuples that are present in the input file. We write a relation with one tuple per line, where tuples are represented as comma-separated items in parentheses:

(1950,0,1)

(1950,22,1)

(1950,-11,1)

(1949,111,1)

Relations are given names, or *aliases*, so they can be referred to. This relation is given the records alias. We can examine the contents of an alias using the DUMP operator:

grunt> DUMP records;

(1950,0,1)

(1950,22,1)

(1950,-11,1)

(1949,111,1)

(1949,78,1)

We can also see the structure of a relation—the relation’s *schema*—using the DESCRIBE operator on the relation’s alias:

grunt> DESCRIBE records;

records: {year: chararray,temperature: int,quality: int}

* **The third statement uses the GROUP function to group the records relation by the year field. Let’s use DUMP to see what it produces:**

grunt> **grouped\_records = GROUP filtered\_records BY year;**

grunt> **DUMP grouped\_records;**

(1949,{(1949,78,1),(1949,111,1)})

(1950,{(1950,-11,1),(1950,22,1),(1950,0,1)})

We now have two rows, or tuples: one for each year in the input data. The first field in each tuple is the field being grouped by (the year), and the second field has a bag of tuples for that year. A *bag* is just an unordered collection of tuples, which in Pig Latin is represented using curly braces.

* **By grouping the data in this way, we have created a row per year, so now all that remains is to find the maximum temperature for the tuples in each bag. Before we do this, let’s understand the structure of the grouped\_records relation:**

grunt> **DESCRIBE grouped\_records;**

grouped\_records: {group: chararray,filtered\_records: {year: chararray,

temperature: int,quality: int}}

This tells us that the grouping field is given the alias group by Pig, and the second field is the same structure as the filtered\_records relation that was being grouped. With this information, we can try the fourth transformation:

grunt> **max\_temp = FOREACH grouped\_records GENERATE group,**

>> **MAX(filtered\_records.temperature);**

* **FOREACH processes every row to generate a derived set of rows, using a GENERATE clause to define the fields in each derived row.**

**In this example, the first field is group, which is just the year. The second field is a little more complex. The filtered\_records.temperature reference is to the temperature field of the filtered\_records bag in the grouped\_records relation. MAX is a built-in function for calculating the maximum value of fields in a bag. In this case, it calculates the maximum temperature for the fields in each filtered\_records bag. Let’s check the result:**

grunt> **DUMP max\_temp;**

(1949,111)

(1950,22)

**COMPARISON WITH DATABASES**

There are several differences between Pig and relational database management systems (RDBMSs).

* Pig Latin is a data flow programming language, whereas SQL is a declarative programming language.
* A Pig Latin program is a step-by-step set of operations on an input relation, in which each step is a single transformation. By contrast, SQL statements are a set of constraints that, taken together, define the output.
* RDBMSs store data in tables, with tightly predefined schemas. Pig is more relaxed about the data that it processes: you can define a schema at runtime, but it’s optional. It will operate on any source of tuples, text file with tab separated fields.
* There is no data import process to load the data into the RDBMS. In Pig, the data is loaded from the filesystem (usually HDFS) as the first step in the processing.
* Pig’s support, for complex and nested data structures, further differentiates it from SQL, which operates on flatter data structures. Also, Pig’s ability to use UDFs and streaming operators that are tightly integrated with the language and Pig’s nested data structures makes Pig Latin more customizable than most SQL dialects.
* RDBMSs have several features to support online, low-latency queries, such as transactions and indexes, that are absent in Pig. Pig does not support random reads or writes.

**Pig Latin**

The syntax and semantics of the Pig Latin programming language is described below:

* 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. statements or commands

for interactive use in Grunt do not need the terminating semicolon. It’s never an error to add a 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. 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';

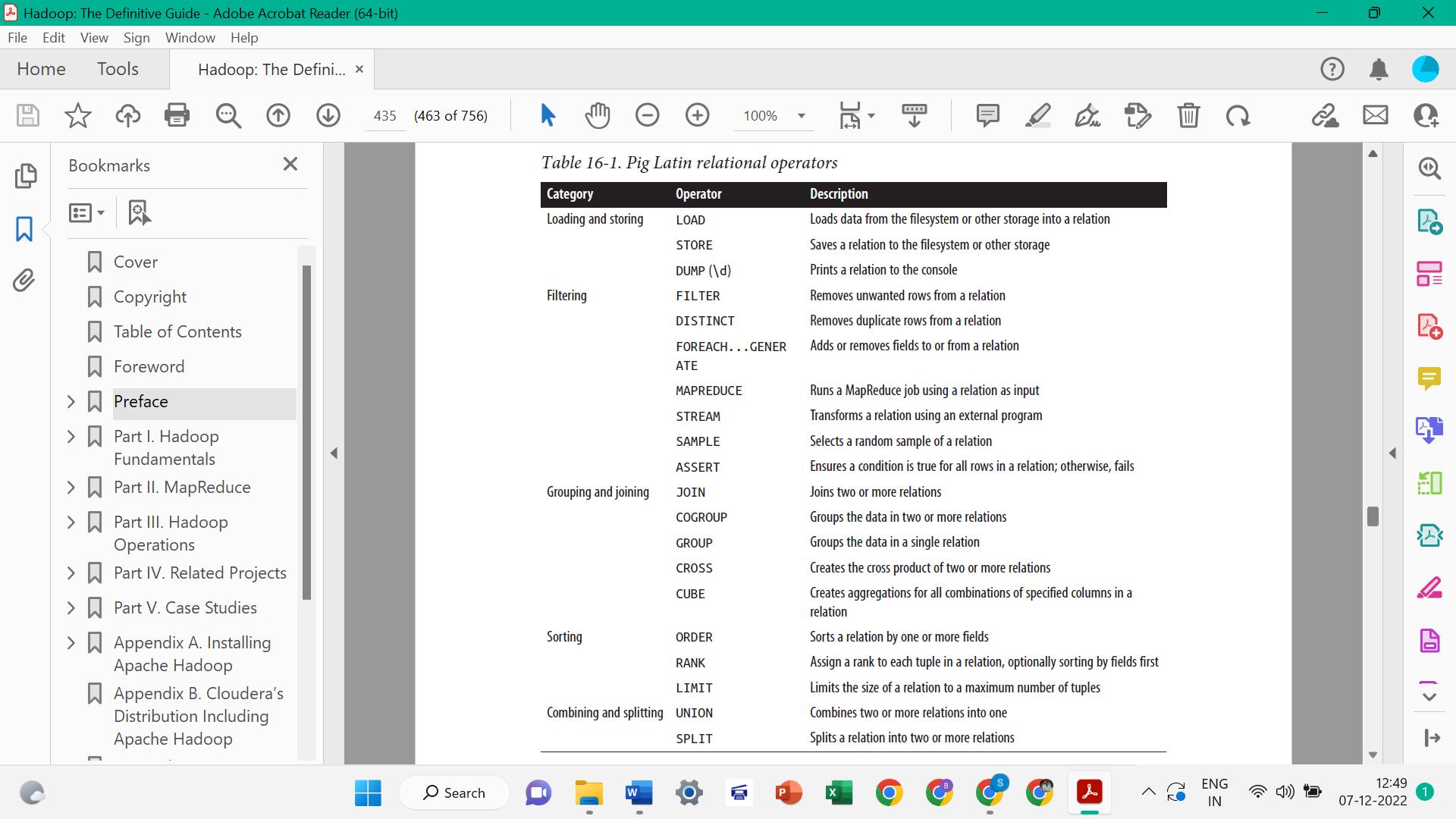
C = **JOIN** A **BY** $0, */\* ignored \*/* B **BY** $1;

**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).
* 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.

**STATEMENTS**

As a Pig Latin program is executed, each statement is parsed in turn. If there are syntax errors or other (semantic) problems, such as undefined aliases, the interpreter will halt and display an error message. The interpreter builds a logical plan for every relational operation, which forms the core of a Pig Latin program. The logical plan for the statement is added to the logical plan for the program so far, and then the interpreter moves on to the next statement.



Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

Description automatically generated

Graphical user interface, text, application

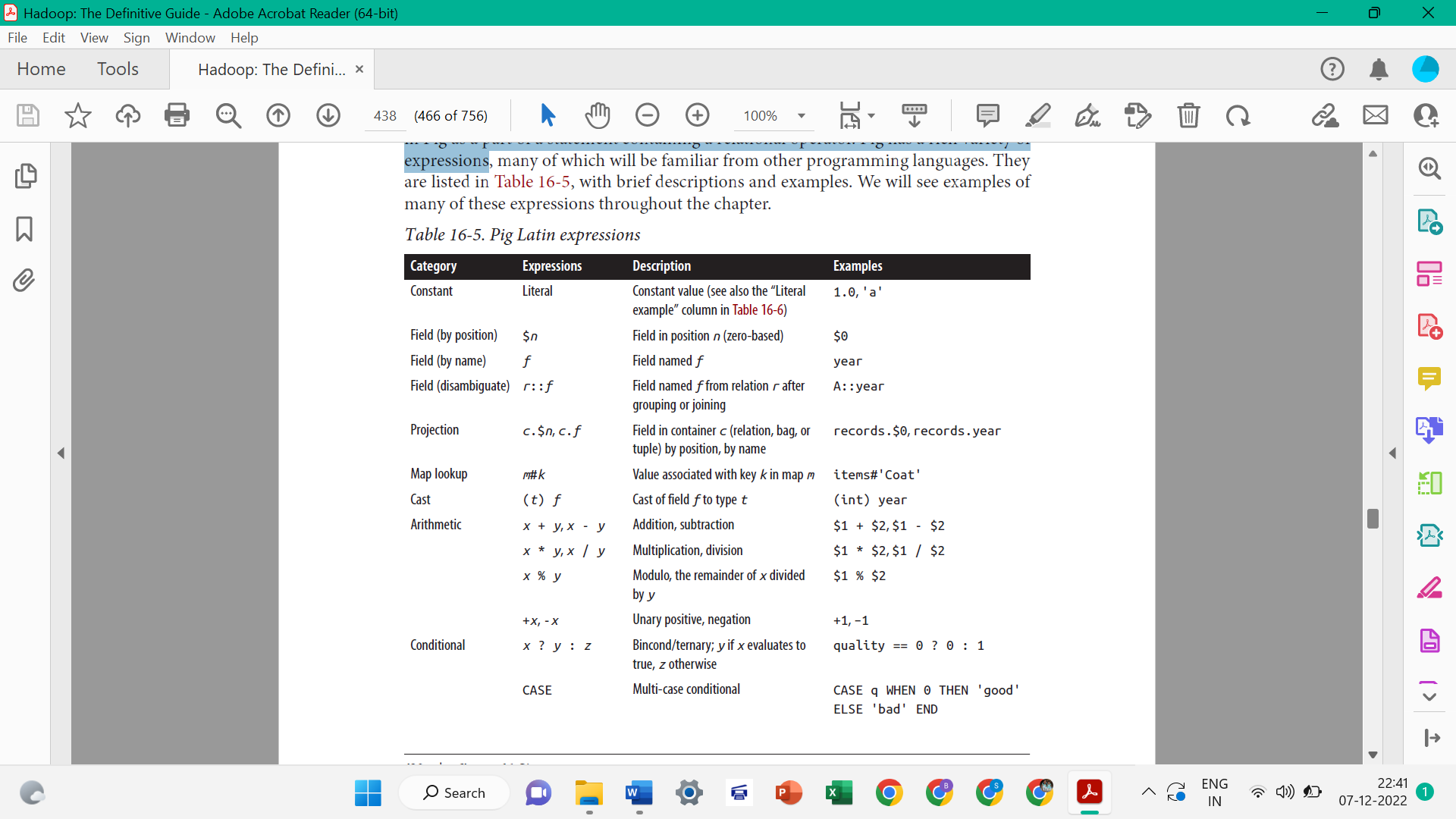
Description automatically generated

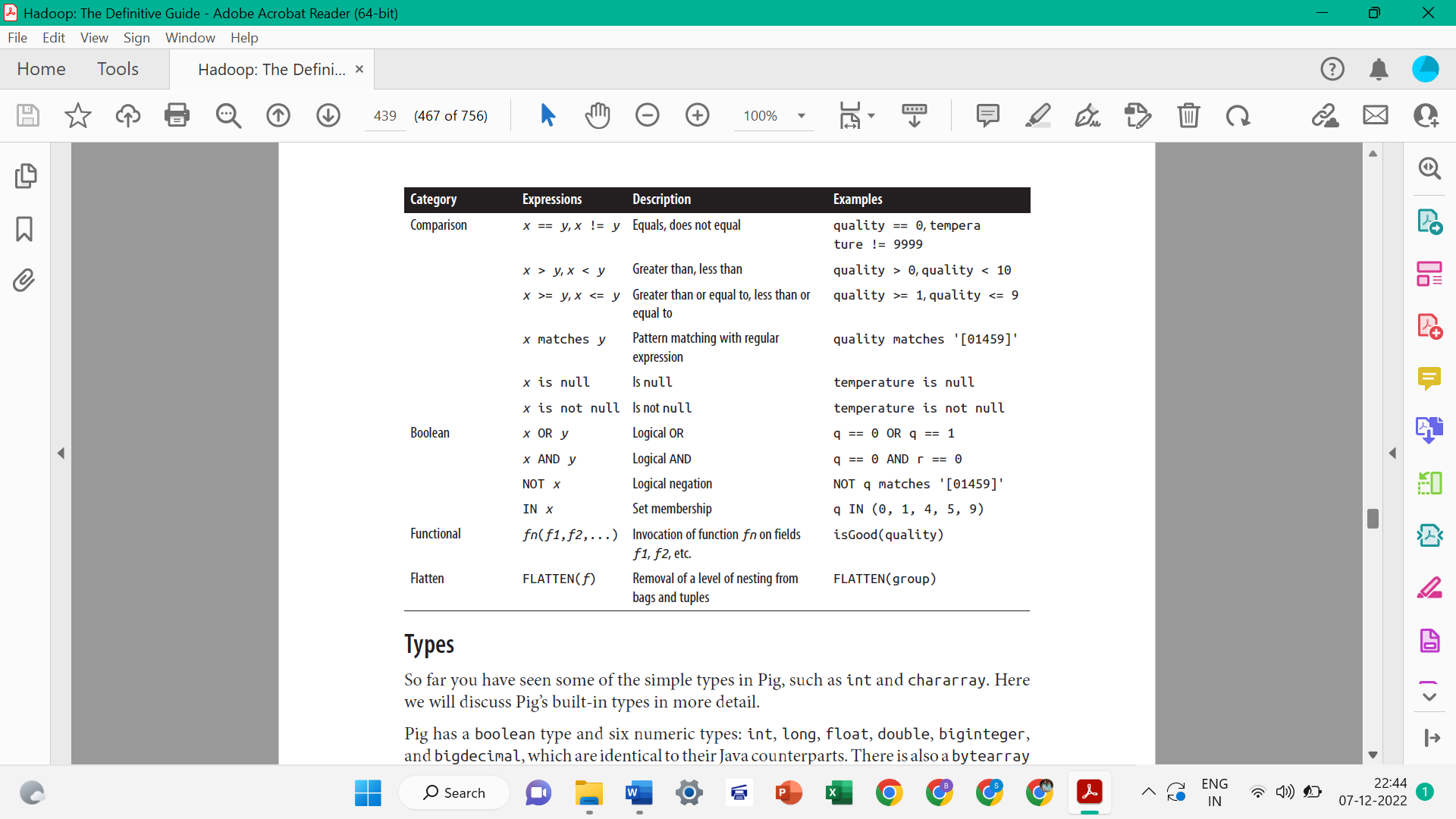
Text

Description automatically generated

**Expressions**

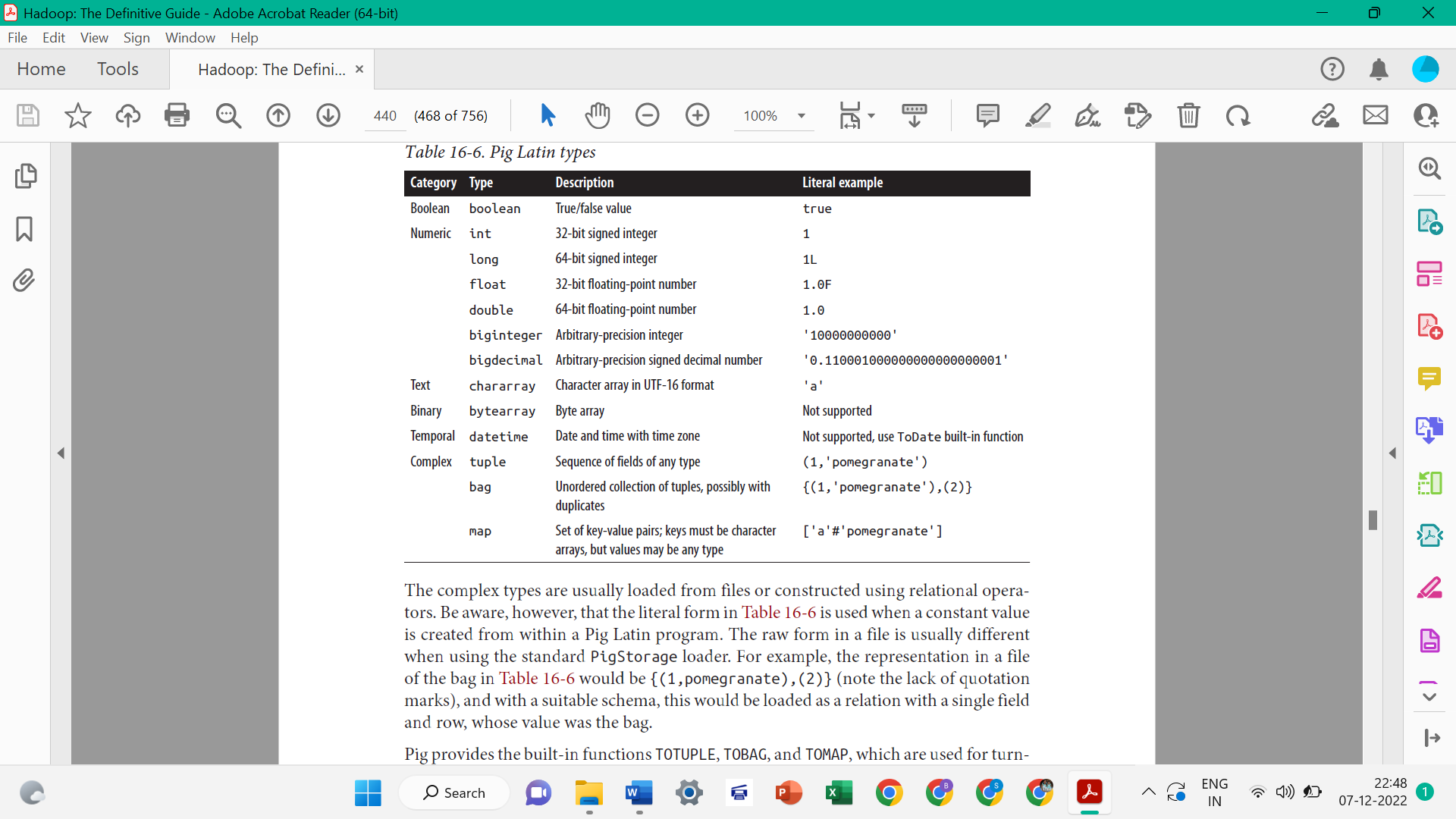
An expression is something that is evaluated to yield a value. Expressions can be used in Pig as a part of a statement containing a relational operator. Pig has a rich variety of expressions.





Types

All of Pig Latin’s types are listed below:



**Schemas**

1. A relation in Pig may have an associated schema, which gives the fields in the relation names and types.

For example:

**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}**

Pig’s flexibility in the degree to which schemas are declared contrasts with schemas in traditional SQL databases, which are declared before the data is loaded into the system. Pig is designed for analysing plain input files with no associated type information, so it is quite natural to choose types for fields later than you would with an RDBMS.

1. 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}**

The types default to bytearray, the most general type, representing a binary string.

1. 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}**

1. 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.**

***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.***

**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. In Pig, if the value cannot be cast to the type declared in the schema, it will substitute a null value.

**For example:**

**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 (and also when saved using**

**STORE):**

**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)**

**Pig produces a warning for the invalid field (not shown here) but does not halt its**

**processing.**

**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.

***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.

Graphical user interface, text, application, email

Description automatically generated

Graphical user interface, text

Description automatically generated

**Other libraries**

If the function you need is not available, you can write your own user-defined function

(or UDF). Piggy Bank, a library of Pig functions shared by the Pig community and distributed as a part of Pig. Apache DataFu is another rich library of Pig UDFs. In addition to general utility functions, it includes functions for computing basic statistics, performing sampling and estimation, hashing, and working with web data.

**User-Defined Functions**

Pig makes it easy to define and use user-defined functions. UDFs can be written in Java, Python, JavaScript, Ruby, or Groovy, all of which are run using the Java Scripting API.

**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);

***Filter UDFs are all subclasses of FilterFunc, which itself is a subclass of EvalFunc.***

A 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 IsGood 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. Then we tell Pig about the JAR file with the REGISTER operator, which is given the local path to the filename (and is *not* enclosed in quotes):

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

Finally, we can invoke the function:

grunt> **filtered\_records = FILTER records BY temperature != 9999 AND**

>> **com.hadoopbook.pig.IsGood (quality);**

We can shorten the function name by defining an alias, using the DEFINE operator:

**grunt> DEFINE isGood com.hadoopbook.pig.IsGood ();**

**grunt> filtered\_records = FILTER records BY temperature != 9999 AND**

**>> isGood(quality);**

**Leveraging types**

The filter works when the quality field is declared to be of type int, but if the type

information is absent, the UDF fails! This happens because the field is the default type,

bytearray, represented by the DataByteArray class. Because DataByteArray is not an

Integer, the cast fails.

The obvious way to fix this is to convert the field to an integer in the exec() method.

However, there is a better way, which is to tell Pig the types of the fields that the function expects. The getArgToFuncMapping() method on EvalFunc is provided for precisely this reason.

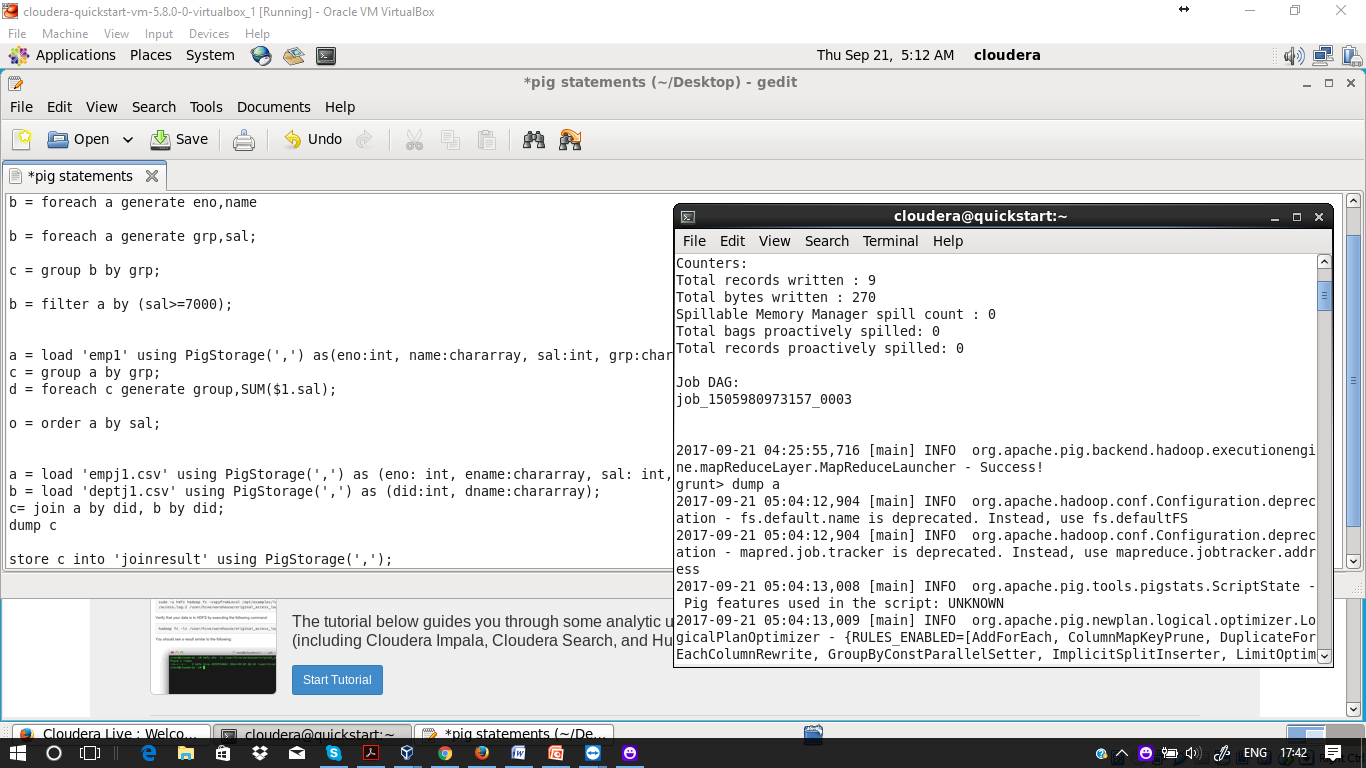
**Some Operators**

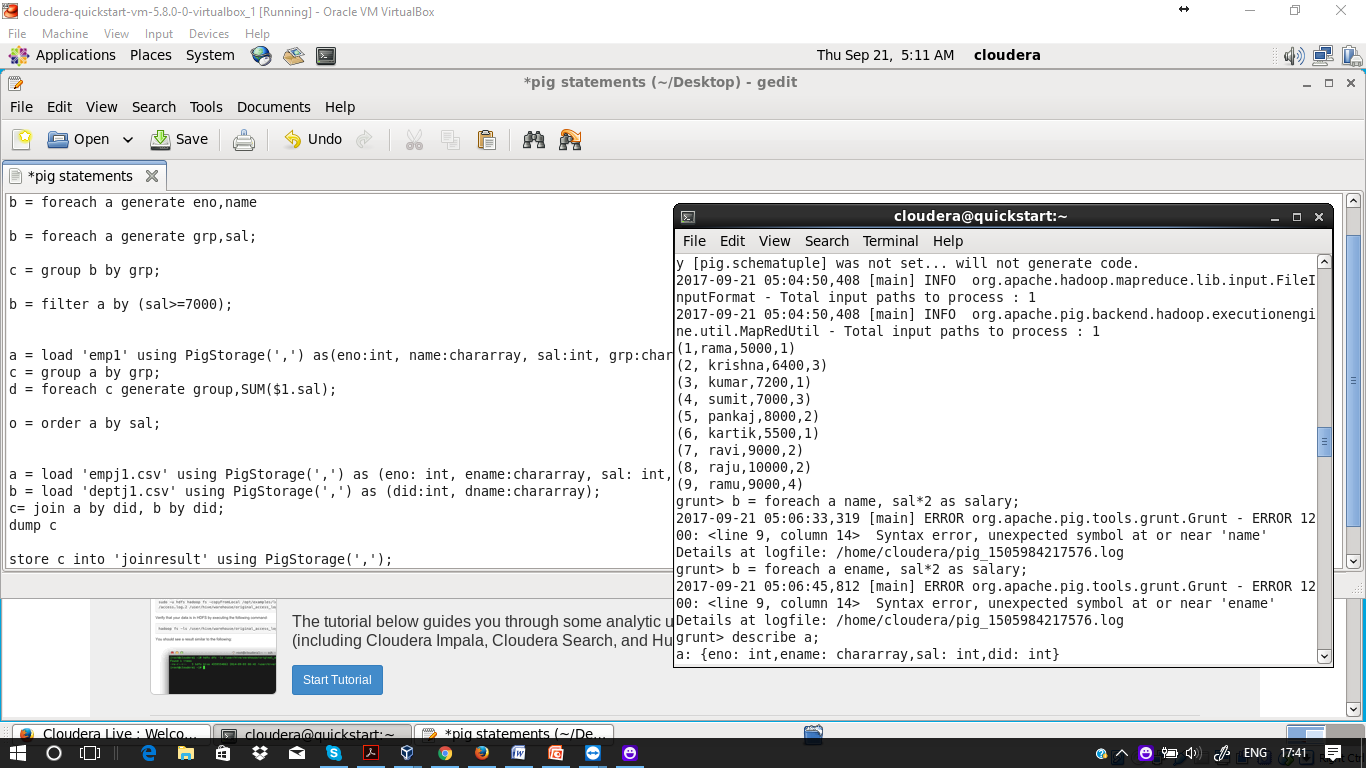
**foreach**

Apply the given operation on each record of the relation and return a new relation.

**Example:**Graphical user interface, text, application

Description automatically generated

****

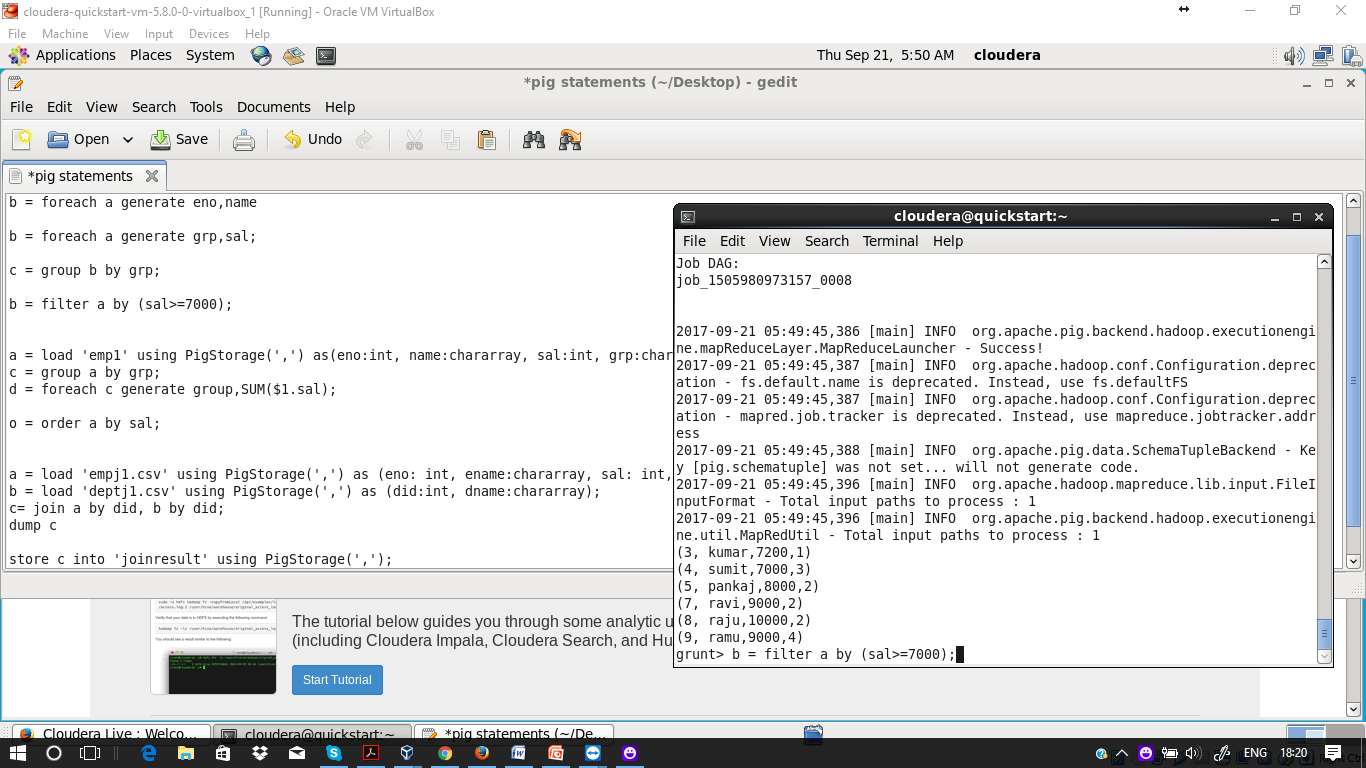
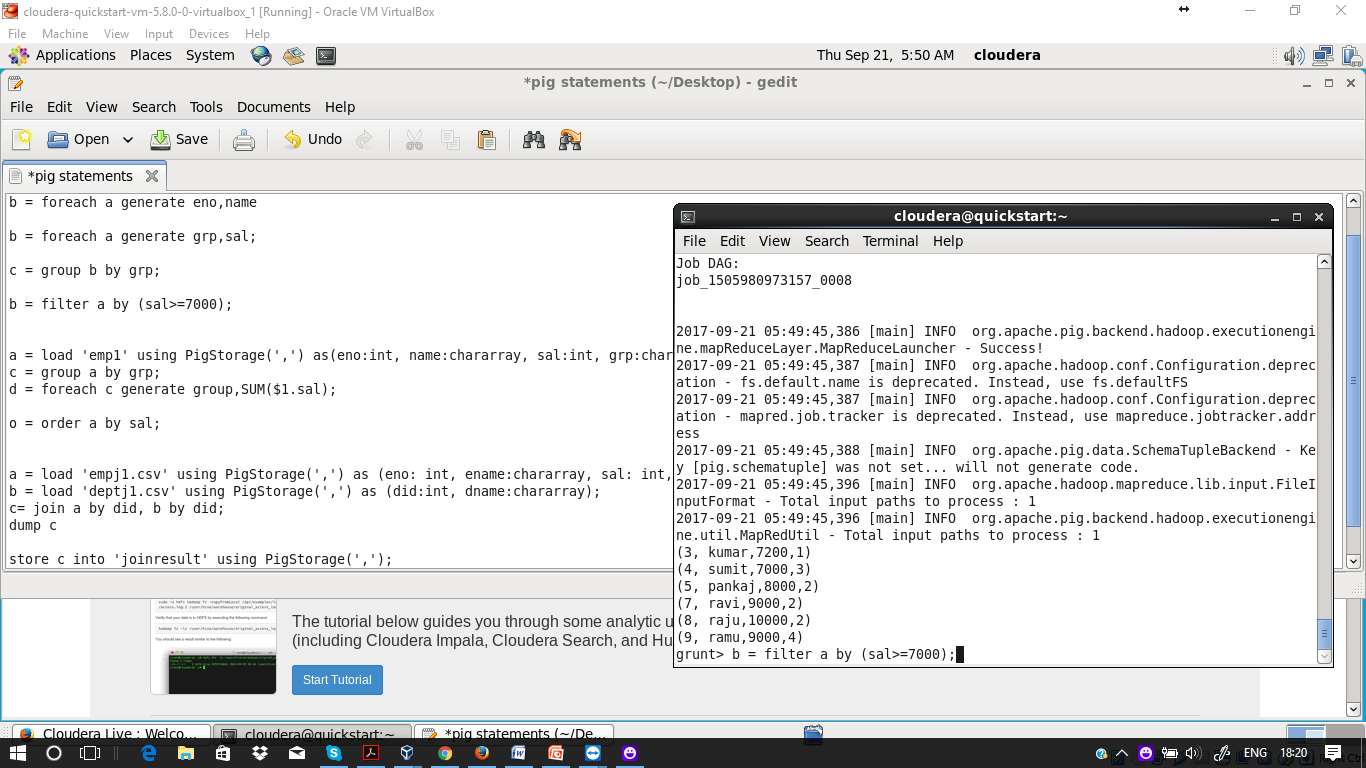
**** Graphical user interface, text, application

Description automatically generated

**Syntax: *Foreach relation generate columns/expression***





**Example**:

Graphical user interface, text

Description automatically generatedGraphical user interface, text

Description automatically generatedGraphical user interface, text, application

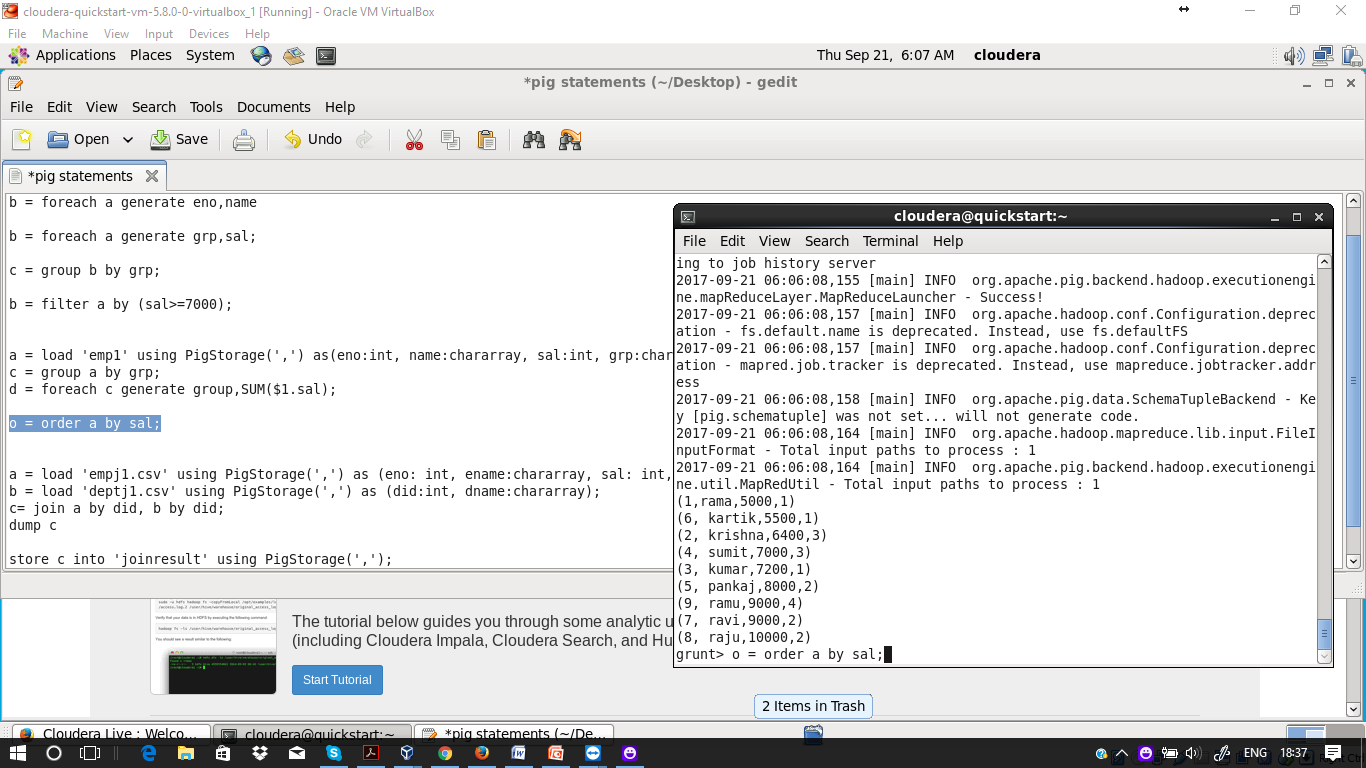
Description automatically generatedGraphical user interface, text, application

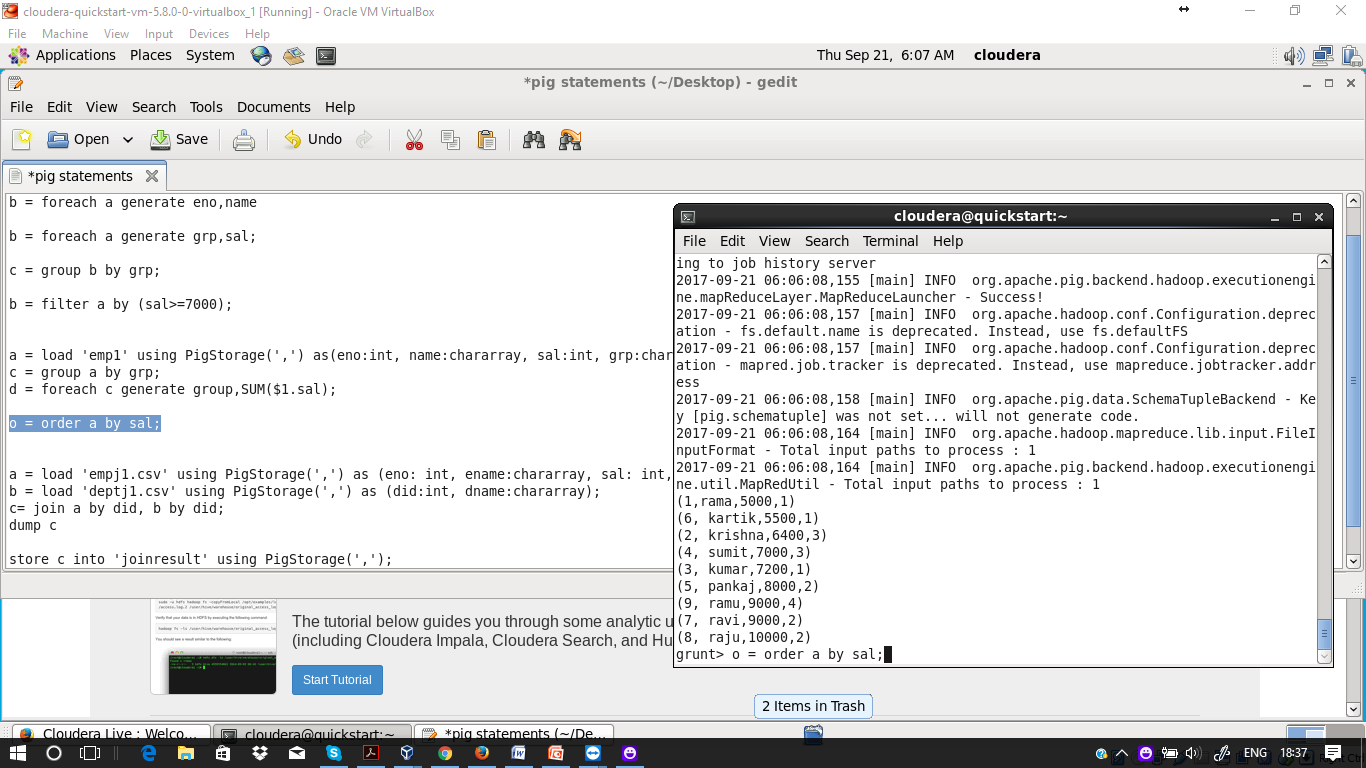
Description automatically generated

**Example**:



**Example**:





Graphical user interface, text

Description automatically generatedGraphical user interface, text

Description automatically generated

**Example**:

