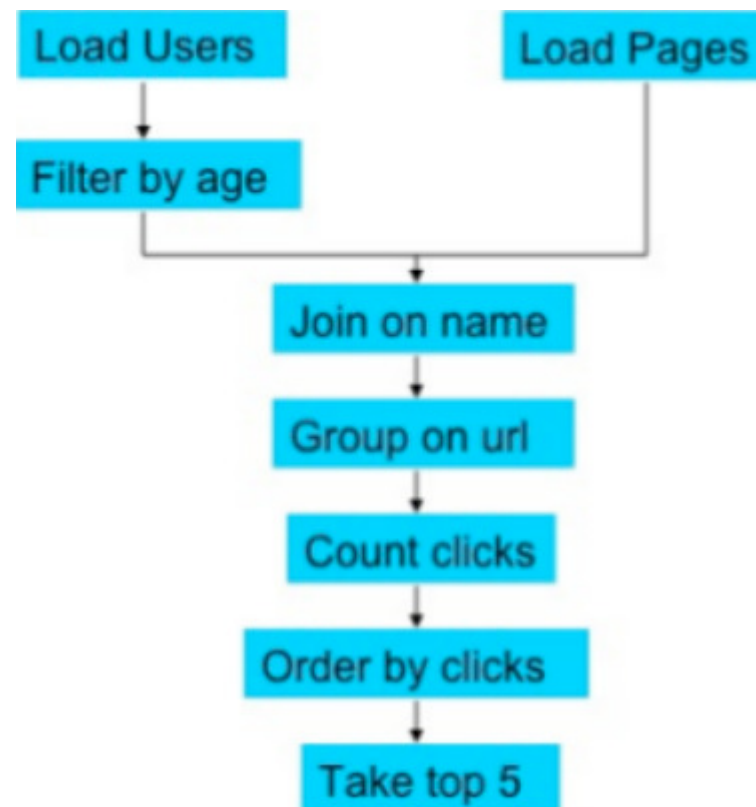


Why Pig?

- High level language
- Small learning curve
- Increases productivity
- Insulates you from complexity of MapReduce
 - ▣ Job configuration tuning
 - ▣ Mapper / Reducer optimization
 - ▣ Data re-use
 - ▣ Job Chains

Simple MapReduce Example

- Input: User profiles, page visits
- Output: the top 5 most visited pages by users aged 18-25



In Native Hadoop Code

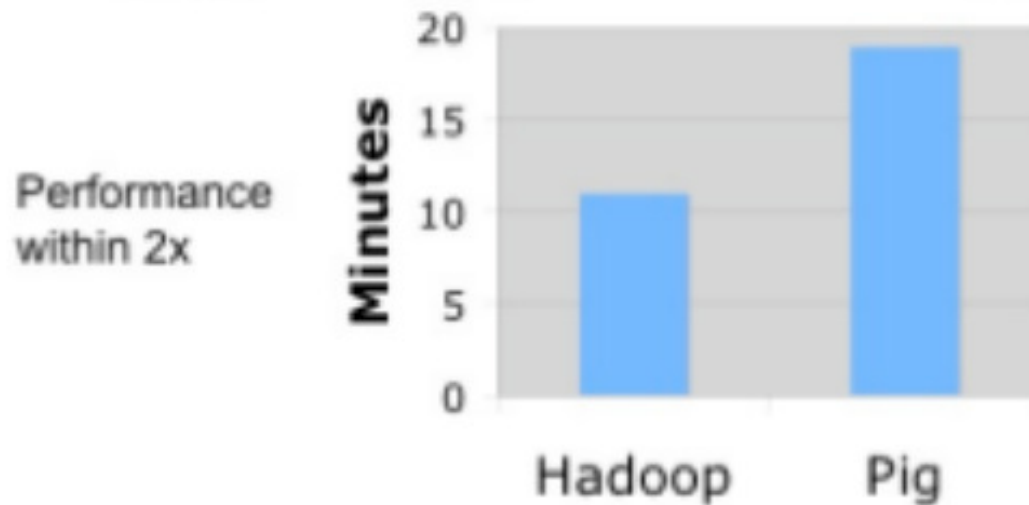
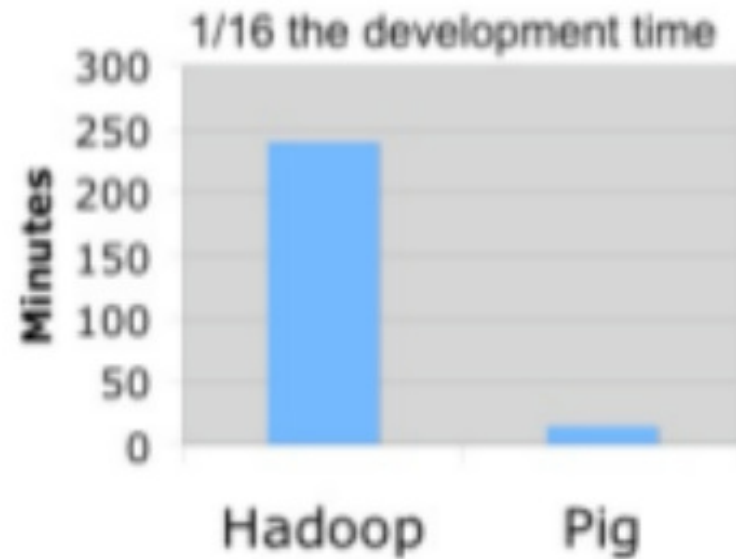
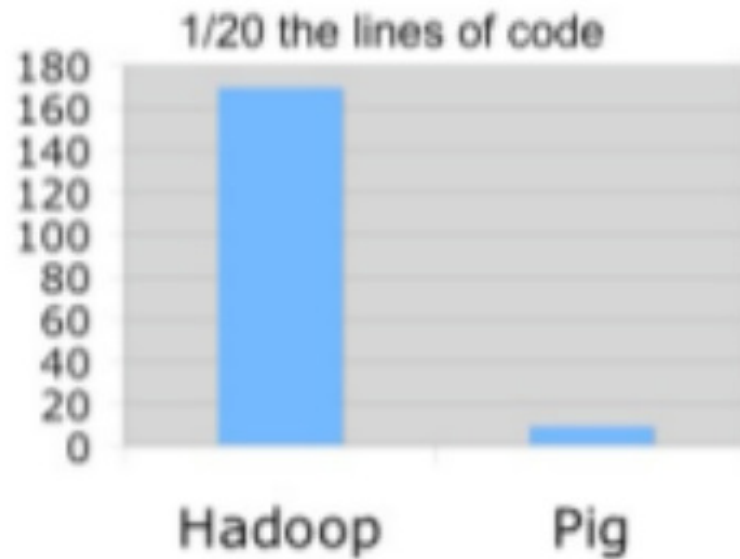
[illegible][illegible][illegible]

In Pig

```
users = LOAD 'users' AS (name, age);
users = FILTER users BY age >= 18 AND age <= 25;
pages = LOAD 'pages' AS (user, url);
joined = JOIN users BY name, pages BY user;
grouped = group JOINED BY url;
summed = FOREACH grouped GENERATE group, COUNT(joined) AS clicks;
sorted = ORDER summed BY clicks DESC;
top5 = LIMIT sorted 5;
STORE top5 INTO '/data/top5sites';
```

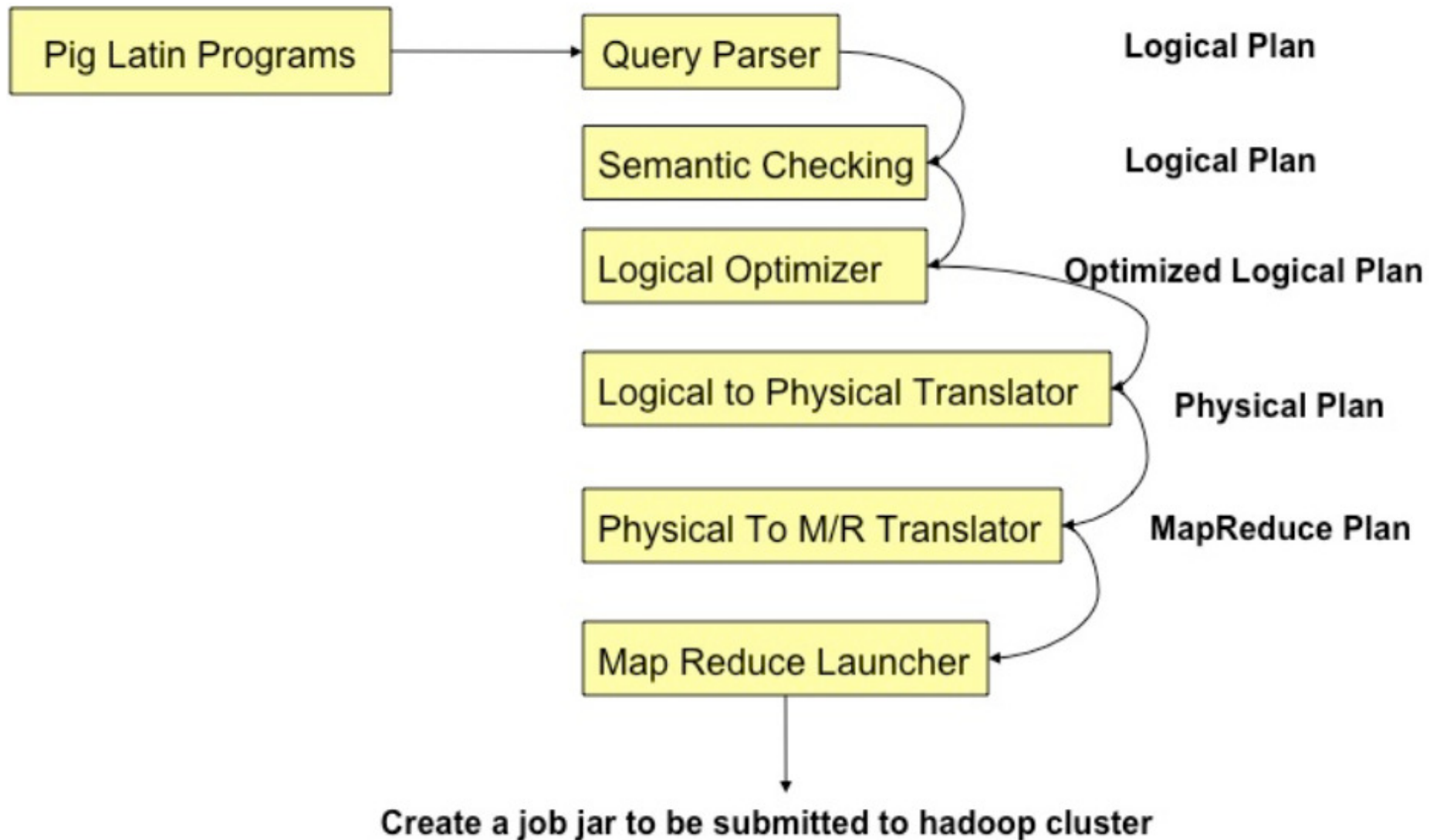


COMPARISONS



- Significantly fewer lines of code
- Considerable less development time
- Reasonably close to optimal performance

Under the Hood



How Pig differs from MapReduce

- ❑ Pig provides users with several advantages over using MapReduce directly.
- ❑ Pig Latin provides all of the standard data-processing operations, such as join, filter, group by, order by, union, etc.
- ❑ MapReduce provides the group by operation directly (that is what the shuffle plus reduce phases are), and it provides the order by operation indirectly through the way it implements the grouping.
- ❑ Filter and projection can be implemented trivially in the map phase.
- ❑ But other operators, particularly join, are not provided and must instead be written by the user.
- ❑ Pig provides some complex, nontrivial implementations of these standard data operations.
- ❑ For example, because the number of records per key in a dataset is rarely evenly distributed, the data sent to the reducers is often skewed.
- ❑ That is, one reducer will get 10 or more times the data than other reducers.
- ❑ Pig has join and order by operators that will handle this case and (in some cases) rebalance the reducers.
- ❑ But these took the Pig team months to write, and rewriting these in MapReduce would be time consuming.
- ❑ In MapReduce, the data processing inside the map and reduce phases is opaque to the system.
- ❑ This means that MapReduce has no opportunity to optimize or check the user's code.
- ❑ Pig, on the other hand, can analyze a Pig Latin script and understand the data flow that the user is describing.
- ❑ That means it can do early error checking (did the user try to add a string field to an integer field?) and optimizations (can these two grouping operations be combined?).
- ❑ MapReduce does not have a type system.
- ❑ This is intentional, and it gives users the flexibility to use their own data types and serialization frameworks.
- ❑ The downside is that this further limits the system's ability to check users' code for error before and during runtime.

Pig for SQL Users

SQL	Pig Latin
SELECT * FROM users;	users = LOAD '/hdfs/users' USING PigStorage ('\\t') AS (name:chararray, age:int, weight:int);
SELECT * FROM users where weight < 150;	skinnyUsers = FILTER users BY weight < 150;
SELECT name, age FROM users where weight < 150;	skinnyUserNames = FOREACH skinnyUsers GENERATE name, age;
SELECT name, SUM(orderAmount) FROM orders GROUP BY name...	A = GROUP orders BY name; B = FOREACH A GENERATE \$0 AS name, SUM(\$1.orderAmount) AS orderTotal;
...HAVING SUM(orderAmount) > 500...	C = FILTER B BY orderTotal > 500;
...ORDER BY name ASC;	D = ORDER C BY name ASC;
SELECT DISTINCT name FROM users;	names = FOREACH users GENERATE name; uniqueNames = DISTINCT names;
SELECT name, COUNT(DISTINCT age) FROM users GROUP BY name;	usersByName = GROUP users BY name; numAgesByName = FOREACH usersByName { ages = DISTINCT users.age; GENERATE FLATTEN(group), COUNT(ages); }

Group then join in SQL



```
CREATE TEMP TABLE t1 AS  
SELECT customer, sum(purchase) AS total_purchases FROM transactions  
GROUP BY customer;
```

```
SELECT customer, total_purchases, zipcode FROM t1, customer_profile  
WHERE t1.customer = customer_profile.customer;
```

Group then join in Pig Latin

-- Load the transactions file, group it by customer, and sum their total purchases

```
txns = load 'transactions' as (customer, purchase);
```

```
grouped = group txns by customer;
```

```
total = foreach grouped generate group, SUM(txns.purchase) as tp;
```

-- Load the customer_profile file

```
profile = load 'customer_profile' as (customer, zipcode);
```

-- join the grouped and summed transactions and customer_profile data

```
answer = join total by group, profile by customer;
```

-- Write the results to the screen

```
dump answer;
```

Real World Pig Script

- "Aggregate yesterday's API web server logs by client and function call."

```
logs = LOAD 'api.log' using PigStorage('\t') AS (type, date, ipAddress, sessionId, clientId, apiMethod);
methods = FILTER logs BY type == 'INFO ';
methods = FOREACH methods GENERATE type, date, clientId, class, method;
methods = GROUP methods BY (clientId, class, method);
methodStats = FOREACH methods GENERATE $0.clientId, $0.class, $0.method, COUNT($1) as methodCount;
STORE methodStats to '/stats/$date/api/apiUsageByClient'
```

Pig Job Performance

- "Find the most commonly used desktop browser, mobile browser, operating system, email client, and geographic location for every contact."
 - ▣ 150 line Pig Latin script
 - ▣ Runs daily on 6 node computation cluster
 - ▣ Processes ~1B rows of raw tracking data in 40 minutes, doing multiple groups and joins via 16 chained MapReduce jobs with 2100 mappers
 - ▣ Output: ~40M rows of contact attributes


User Defined Functions (UDF)

To perform more complex operations on fields Written in java, compiled into a jar, loaded into your Pig script at runtime.

```
package myudfs;
import java.io.IOException;
import org.apache.pig.EvalFunc;
import org.apache.pig.data.Tuple;
import org.apache.pig.impl.util.WrappedIOException;

public class UPPER extends EvalFunc<String>
{
    public String exec(Tuple input) throws IOException
    {
        if (input == null || input.size() == 0)
            return null;
        try{
            String str = (String)input.get(0);
            return str.toUpperCase();
        }catch(Exception e) {
            throw WrappedIOException.wrap("Caught exception processing input row ", e);
        }
    }
}
```

Making use of your UDF in a Pig Script:



```
REGISTER myudfs.jar;  
students = LOAD 'student_data' AS (name: chararray, age: int, gpa: float);  
upperNames = FOREACH students GENERATE myudfs.UPPER(name);  
DUMP upperNames;
```

UDF Pitfalls

- ❑ UDFs are limited; can only operate on fields, not on groups of fields.
- ❑ A given UDF can only return a single data type (integer / float / chararray /etc).
- ❑ To build a jar file that contains all available UDFs, follow these steps:
 - ❑ Checkout UDF code: `svn co http://svn.apache.org/repos/asf/pig/trunk/contrib/piggybank`
 - ❑ Add pig.jar to your ClassPath: `export CLASSPATH=$CLASSPATH:/path/to/pig.jar`
 - ❑ Build the jar file: `cd trunk/contrib/piggybank/java; run "ant"`
- ❑ This will generate piggybank.jar in the same directory.
- ❑ You must build piggybank in order to read UDF documentation - run "ant javadoc" from directory trunk/contrib/piggybank/java.
- ❑ The documentation is generated in directory trunk/contrib/piggybank/java/build/javadoc.
- ❑ How to compile a custom UDF isn't obvious.
- ❑ After writing your UDF, you must place your java code in an appropriate directory inside a checkout of the piggybank code and build the piggybank jar with ant.

Common Pig Pitfalls

- Trying to match pig version with hadoop / hbase versions.
- There is very little documentation on what is compatible with what.
- A few snippets from the mailing list:
 - “Are you using Pig 8 distribution or Pig 8 from svn?
 - ▣ You want the latter (soon-to-be-Pig 0.8.1)”
- “Please upgrade your pig version to the latest in the 0.8 branch.
- The 0.8 release is not compatible with 0.20+ versions of hbase; we bumped up the support in 0.8.1, which is nearing release.
- Cloudera's latest CDH3 GA might have these patches (it was just released today) but CDH3B4 didn't.”

Pitfalls

- ❑ Bugs in older versions of pig requiring you to register jars.
- ❑ Indicated by MapReduce job failure due to `java.lang.ClassNotFoundException`:
- ❑ I finally resolved the problem by manually registering jars:
- ❑ `REGISTER /path/to/pig_0.8/lib/google-collections-1.0.jar;`
- ❑ `REGISTER /path/to/pig_0.8/lib/hbase-0.20.3-1.cloudera.jar;`
- ❑ `REGISTER /path/to/pig_0.8/lib/zookeeper-hbase-1329.jar`
- ❑ From the mailing list: “If you are using Hbase 0.91 and Pig 0.8.1, the `hbaseStorage` code in Pig is supposed to auto-register the hbase, zookeeper, and google-collections jars, so you won't have to do that.”
- ❑ No more registering jars, though they do need to be on your classpath.

Obscure Pig Pitfalls



- ❑ HBaseLoader bug requiring disabling input splits.
- ❑ Pig versions prior to 0.8.1 will only load a single HBase region unless you disable input splits.
- ❑ Fix via: SET pig.splitCombination 'false';

Obscure Pig Pitfalls



```
visitors = LOAD 'hbase://tracking' USING HBaseStorage( 'open:browser  
open:ip open:os open:createdDate') as (browser:chararray, ipAddress:  
chararray, os:chararray, createdDate:chararray);
```

Resulted in:

```
java.lang.RuntimeException: Failed to create DataStorage at org.  
apache.pig.backend.hadoop.datastorage.HDataStorage.init(HDataStorage.java:75)
```

Caused by:

```
Call to hadoopMaster failed on java.io.EOFException
```

Pig Setup

Requirements

Mandatory

Unix and Windows users need the following:

- **Hadoop 0.20.2, 020.203, 020.204, 0.20.205, 1.0.0, 1.0.1, or 0.23.0, 0.23.1** - <http://hadoop.apache.org/common/releases.html> (You can run Pig with different versions of Hadoop by setting HADOOP_HOME to point to the directory where you have installed Hadoop. If you do not set HADOOP_HOME, by default Pig will run with the embedded version, currently Hadoop 1.0.0.)
- **Java 1.6** - <http://java.sun.com/javase/downloads/index.jsp> (set JAVA_HOME to the root of your Java installation)

Windows users also need to install Cygwin and the Perl package: <http://www.cygwin.com/>

Optional

- **Python 2.5** - <http://jython.org/downloads.html> (when using Python UDFs or embedding Pig in Python)
- **JavaScript 1.7** - https://developer.mozilla.org/en/Rhino_downloads_archive and <http://mirrors.ibiblio.org/pub/mirrors/maven2/rhino/js/> (when using JavaScript UDFs or embedding Pig in JavaScript)
- **JRuby 1.6.7** - <http://www.jruby.org/download> (when using JRuby UDFs)
- **Groovy (groovy-all) 1.8.6** - <http://groovy.codehaus.org/Download> or directly on a maven repo <http://mirrors.ibiblio.org/pub/mirrors/maven2/org/codehaus/groovy/groovy-all/1.8.6/> (when using Groovy UDFs or embedding Pig in Groovy)
- **Ant 1.7** - <http://ant.apache.org/> (for builds)
- **JUnit 4.5** - <http://junit.sourceforge.net/> (for unit tests)

Download Pig

To get a Pig distribution, do the following:

1. Download a recent stable release from one of the Apache Download Mirrors (see [Pig Releases](#)).
2. Unpack the downloaded Pig distribution, and then note the following:
 - The Pig script file, `pig`, is located in the `bin` directory (`/pig-n.n.n/bin/pig`). The Pig environment variables are described in the Pig script file.
 - The Pig properties file, `pig.properties`, is located in the `conf` directory (`/pig-n.n.n/conf/pig.properties`). You can specify an alternate location using the `PIG_CONF_DIR` environment variable.
3. Add `/pig-n.n.n/bin` to your path. Use `export` (`bash`, `sh`, `ksh`) or `setenv` (`tcsh`, `csh`). For example:

```
$ export PATH=/<my-path-to-pig>/pig-n.n.n/bin:$PATH
```
4. Test the Pig installation with this simple command: `$ pig -help`

Build Pig

To build pig, do the following:

1. Check out the Pig code from SVN: `svn co http://svn.apache.org/repos/asf/pig/trunk`
2. Build the code from the top directory: `ant`
If the build is successful, you should see the `pig.jar` file created in that directory.
3. Validate the `pig.jar` by running a unit test: `ant test`

Running Pig

You can run Pig (execute Pig Latin statements and Pig commands) using various modes.

	Local Mode	Mapreduce Mode
Interactive Mode	yes	yes
Batch Mode	yes	yes

Execution Modes

Pig has two execution modes or exectypes:

- **Local Mode** - To run Pig in local mode, you need access to a single machine; all files are installed and run using your local host and file system. Specify local mode using the -x flag (pig -x local). Note that local mode does not support parallel mapper execution with Hadoop 0.20.x and 1.0.0. This is because the LocalJobRunner of these Hadoop versions is not thread-safe.
- **Mapreduce Mode** - To run Pig in mapreduce mode, you need access to a Hadoop cluster and HDFS installation. Mapreduce mode is the default mode; you can, *but don't need to*, specify it using the -x flag (pig OR pig -x mapreduce).

You can run Pig in either mode using the "pig" command (the bin/pig Perl script) or the "java" command (java -cp pig.jar ...).

Examples

This example shows how to run Pig in local and mapreduce mode using the pig command.

```
/* local mode */  
$ pig -x local ...
```

```
/* mapreduce mode */  
$ pig ...  
or  
$ pig -x mapreduce ...
```

This example shows how to run Pig in local and mapreduce mode using the java command.

```
/* local mode */  
$ java -cp pig.jar org.apache.pig.Main -x local ...
```

```
/* mapreduce mode */  
$ java -cp pig.jar org.apache.pig.Main ...  
or  
$ java -cp pig.jar org.apache.pig.Main -x mapreduce ...
```

Interactive Mode

You can run Pig in interactive mode using the Grunt shell. Invoke the Grunt shell using the "pig" command (as shown below) and then enter your Pig Latin statements and Pig commands interactively at the command line.

Example

These Pig Latin statements extract all user IDs from the /etc/passwd file. First, copy the /etc/passwd file to your local working directory. Next, invoke the Grunt shell by typing the "pig" command (in local or hadoop mode). Then, enter the Pig Latin statements interactively at the grunt prompt (be sure to include the semicolon after each statement). The DUMP operator will display the results to your terminal screen.

```
grunt> A = load 'passwd' using PigStorage(':');
grunt> B = foreach A generate $0 as id;
grunt> dump B;
```

Local Mode

```
$ pig -x local
... - Connecting to ...
grunt>
```

Mapreduce Mode

```
$ pig -x mapreduce
... - Connecting to ...
grunt>
```

or

```
$ pig
... - Connecting to ...
grunt>
```


Batch Mode

You can run Pig in batch mode using [Pig scripts](#) and the "pig" command (in local or hadoop mode).

Example

The Pig Latin statements in the Pig script (id.pig) extract all user IDs from the /etc/passwd file. First, copy the /etc/passwd file to your local working directory. Next, run the Pig script from the command line (using local or mapreduce mode). The STORE operator will write the results to a file (id.out).

```
/* id.pig */  
  
A = load 'passwd' using PigStorage(':'); -- load the passwd file  
B = foreach A generate $0 as id; -- extract the user IDs  
store B into 'id.out'; -- write the results to a file name id.out
```

Local Mode

```
$ pig -x local id.pig
```

Mapreduce Mode

```
$ pig id.pig  
or  
$ pig -x mapreduce id.pig
```

Pig Scripts

Use Pig scripts to place Pig Latin statements and Pig commands in a single file. While not required, it is good practice to identify the file using the *.pig extension.

You can run Pig scripts from the command line and from the Grunt shell (see the [run](#) and [exec](#) commands).

Pig scripts allow you to pass values to parameters using [parameter substitution](#).

Comments in Scripts

You can include comments in Pig scripts:

- For multi-line comments use `/* */`
- For single-line comments use `--`

```
/* myscript.pig
My script is simple.
It includes three Pig Latin statements.
*/

A = LOAD 'student' USING PigStorage() AS (name:chararray, age:int, gpa:float);
B = FOREACH A GENERATE name; -- transforming data
DUMP B; -- retrieving results
```

Scripts and Distributed File Systems

Pig supports running scripts (and Jar files) that are stored in HDFS, Amazon S3, and other distributed file systems. The script's full location URI is required (see [REGISTER](#) for information about Jar files). For example, to run a Pig script on HDFS, do the following:

```
$ pig hdfs://nn.mydomain.com:9020/myscripts/script.pig
```

Pig Latin Statements

Pig Latin statements are the basic constructs you use to process data using Pig. A Pig Latin statement is an operator that takes a [relation](#) as input and produces another relation as output. (This definition applies to all Pig Latin operators except LOAD and STORE which read data from and write data to the file system.) Pig Latin statements may include [expressions](#) and [schemas](#). Pig Latin statements can span multiple lines and must end with a semi-colon (;). By default, Pig Latin statements are processed using [multi-query execution](#).

Pig Latin statements are generally organized as follows:

- A LOAD statement to read data from the file system.
- A series of "transformation" statements to process the data.
- A DUMP statement to view results or a STORE statement to save the results.

Note that a DUMP or STORE statement is required to generate output.

- In this example Pig will validate, but not execute, the LOAD and FOREACH statements.

```
A = LOAD 'student' USING PigStorage() AS (name:chararray, age:int, gpa:float);  
B = FOREACH A GENERATE name;
```

- In this example, Pig will validate and then execute the LOAD, FOREACH, and DUMP statements.

```
A = LOAD 'student' USING PigStorage() AS (name:chararray, age:int, gpa:float);  
B = FOREACH A GENERATE name;  
DUMP B;  
(John)  
(Mary)  
(Bill)  
(Joe)
```

Loading Data

Use the [LOAD](#) operator and the [load/store functions](#) to read data into Pig (PigStorage is the default load function).

Working with Data

Pig allows you to transform data in many ways. As a starting point, become familiar with these operators:

- Use the [FILTER](#) operator to work with tuples or rows of data. Use the [FOREACH](#) operator to work with columns of data.
- Use the [GROUP](#) operator to group data in a single relation. Use the [COGROUP](#), [inner JOIN](#), and [outer JOIN](#) operators to group or join data in two or more relations.
- Use the [UNION](#) operator to merge the contents of two or more relations. Use the [SPLIT](#) operator to partition the contents of a relation into multiple relations.

Storing Intermediate Results

Pig stores the intermediate data generated between MapReduce jobs in a temporary location on HDFS. This location must already exist on HDFS prior to use. This location can be configured using the `pig.temp.dir` property. The property's default value is `"/tmp"` which is the same as the hardcoded location in Pig 0.7.0 and earlier versions.

Storing Final Results

Use the [STORE](#) operator and the [load/store functions](#) to write results to the file system (PigStorage is the default store function).

Note: During the testing/debugging phase of your implementation, you can use `DUMP` to display results to your terminal screen. However, in a production environment you always want to use the `STORE` operator to save your results (see [Store vs. Dump](#)).

Debugging Pig Latin

Pig Latin provides operators that can help you debug your Pig Latin statements:

- Use the [DUMP](#) operator to display results to your terminal screen.
- Use the [DESCRIBE](#) operator to review the schema of a relation.
- Use the [EXPLAIN](#) operator to view the logical, physical, or map reduce execution plans to compute a relation.
- Use the [ILLUSTRATE](#) operator to view the step-by-step execution of a series of statements.

Shortcuts for Debugging Operators

Pig provides shortcuts for the frequently used debugging operators (DUMP, DESCRIBE, EXPLAIN, ILLUSTRATE). These shortcuts can be used in Grunt shell or within pig scripts. Following are the shortcuts supported by pig

- \d alias - shortcut for [DUMP](#) operator. If alias is ignored last defined alias will be used.
- \de alias - shortcut for [DESCRIBE](#) operator. If alias is ignored last defined alias will be used.
- \e alias - shortcut for [EXPLAIN](#) operator. If alias is ignored last defined alias will be used.
- \i alias - shortcut for [ILLUSTRATE](#) operator. If alias is ignored last defined alias will be used.
- \q - To quit grunt shell

Pig Properties

Pig supports a number of Java properties that you can use to customize Pig behavior. You can retrieve a list of the properties using the [help properties](#) command. All of these properties are optional; none are required.

To specify Pig properties use one of these mechanisms:

- The pig.properties file (add the directory that contains the pig.properties file to the classpath)
- The -D command line option and a Pig property (pig -Dpig.tmpfilecompression=true)
- The -P command line option and a properties file (pig -P mypig.properties)
- The [set](#) command (set pig.exec.nocombiner true)

Note: The properties file uses standard Java property file format.

The following precedence order is supported: pig.properties < -D Pig property < -P properties file < set command. This means that if the same property is provided using the -D command line option as well as the -P command line option (properties file), the value of the property in the properties file will take precedence.

To specify Hadoop properties you can use the same mechanisms:

- Hadoop configuration files (include pig-cluster-hadoop-site.xml)
- The -D command line option and a Hadoop property (pig -Dmapreduce.task.profile=true)
- The -P command line option and a property file (pig -P property_file)
- The [set](#) command (set mapred.map.tasks.speculative.execution false)

The same precedence holds: Hadoop configuration files < -D Hadoop property < -P properties_file < set command.

Hadoop properties are not interpreted by Pig but are passed directly to Hadoop. Any Hadoop property can be passed this way.

All properties that Pig collects, including Hadoop properties, are available to any UDF via the UDFContext object. To get access to the properties, you can call the getJobConf method.

Table 10.2 Data read/write operators in Pig Latin

LOAD	<pre>alias = LOAD 'file' [USING function] [AS schema];</pre> <p>Load data from a file into a relation. Uses the PigStorage load function as default unless specified otherwise with the USING option. The data can be given a schema using the AS option.</p>
LIMIT	<pre>alias = LIMIT alias n;</pre> <p>Limit the number of tuples to n. When used right after <code>alias</code> was processed by an ORDER operator, LIMIT returns the first n tuples. Otherwise there's no guarantee which tuples are returned. The LIMIT operator defies categorization because it's certainly not a read/write operator but it's not a true relational operator either. We include it here for the practical reason that a reader looking up the DUMP operator, explained later, will remember to use the LIMIT operator right before it.</p>
DUMP	<pre>DUMP alias;</pre> <p>Display the content of a relation. Use mainly for debugging. The relation should be small enough for printing on screen. You can apply the LIMIT operation on an alias to make sure it's small enough for display.</p>
STORE	<pre>STORE alias INTO 'directory' [USING function];</pre> <p>Store data from a relation into a directory. The directory must not exist when this command is executed. Pig will create the directory and store the relation in files named <code>part-nnnnn</code> in it. Uses the PigStorage store function as default unless specified otherwise with the USING option.</p>

Table 10.3 Diagnostic operators in Pig Latin

DESCRIBE	DESCRIBE alias; Display the schema of a relation.
EXPLAIN	EXPLAIN [-out path] [-brief] [-dot] [-param ...] [-param_file ...] alias; Display the execution plan used to compute a relation. When used with a script name, for example, EXPLAIN myscript.pig, it will show the execution plan of the script.
ILLUSTRATE	ILLUSTRATE alias; Display step-by-step how data is transformed, starting with a load command, to arrive at the resulting relation. To keep the display and processing manageable, only a (not completely random) sample of the input data is used to simulate the execution. In the unfortunate case where none of Pig's initial sample will survive the script to generate meaningful data, Pig will "fake" some similar initial data that will survive to generate data for alias. For example, consider these operations: <pre>A = LOAD 'student.data' as (name, age); B = FILTER A by age > 18; ILLUSTRATE B;</pre> If every tuple Pig samples for A happens to have age less than or equal to 18, B is empty and not much is "illustrated." Instead Pig will construct for A some tuples with age greater than 18. This way B won't be an empty relation and users can see how the script works. In order for ILLUSTRATE to work, the load command in the first step must include a schema. The subsequent transformations must not include the LIMIT or SPLIT operators, or the nested FOREACH operator, or the use of the map data type (to be explained in section 10.5.1).

Table 10.7 Built-in functions in Pig Latin

AVG	Calculate the average of numeric values in a single-column bag.
CONCAT	Concatenate two strings (<code>chararray</code>) or two <code>bytearrays</code> .
COUNT	Calculate the number of tuples in a bag. See <code>SIZE</code> for other data types.
DIFF	Compare two fields in a tuple. If the two fields are bags, it will return tuples that are in one bag but not the other. If the two fields are values, it will emit tuples where the values don't match.
MAX	Calculate the maximum value in a single-column bag. The column must be a numeric type or a <code>chararray</code> .
MIN	Calculate the minimum value in a single-column bag. The column must be a numeric type or a <code>chararray</code> .
SIZE	Calculate the number of elements. For a bag it counts the number of tuples. For a tuple it counts the number of elements. For a <code>chararray</code> it counts the number of characters. For a <code>bytearray</code> it counts the number of bytes. For numeric scalars it always returns 1.
SUM	Calculate the sum of numeric values in a single-column bag.
TOKENIZE	Split a string (<code>chararray</code>) into a bag of words (each word is a tuple in the bag). Word separators are space, double quote ("), comma, parentheses, and asterisk (*).
IsEmpty	Check if a bag or map is empty.

Table 10.8 Relational operators in Pig Latin

SPLIT	<p><code>SPLIT alias INTO alias IF expression, alias IF expression [, alias IF expression ...];</code></p> <p>Splits a relation into two or more relations, based on the given Boolean expressions. Note that a tuple can be assigned to more than one relation, or to none at all.</p>
UNION	<p><code>alias = UNION alias, alias, [, alias ...]</code></p> <p>Creates the union of two or more relations. Note that</p> <ul style="list-style-type: none">■ As with any relation, there's no guarantee to the order of tuples■ Doesn't require the relations to have the same schema or even the same number of fields■ Doesn't remove duplicate tuples
FILTER	<p><code>alias = FILTER alias BY expression;</code></p> <p>Selects tuples based on Boolean expression. Used to select tuples that you want or remove tuples that you don't want.</p>
DISTINCT	<p><code>alias = DISTINCT alias [PARALLEL n];</code></p> <p>Remove duplicate tuples.</p>
SAMPLE	<p><code>alias = SAMPLE alias factor;</code></p> <p>Randomly sample a relation. The sampling factor is given in <code>factor</code>. For example, a 1% sample of data in relation <code>large_data</code> is</p> <p><code>small_data = SAMPLE large_data 0.01;</code></p> <p>The operation is probabilistic in such a way that the size of <code>small_data</code> will not be exactly 1% of <code>large_data</code>, and there's no guarantee the operation will return the same number of tuples each time.</p>
FOREACH	<p><code>alias = FOREACH alias GENERATE expression [,expression ...] [AS schema];</code></p> <p>Loop through each tuple and generate new tuple(s). Usually applied to transform columns of data, such as adding or deleting fields.</p> <p>One can optionally specify a schema for the output relation; for example, naming new fields.</p>

Table 10.8 Relational operators in Pig Latin (continued)

FOREACH (nested)	<pre>alias = FOREACH nested_alias { alias = nested_op; [alias = nested_op; ...] GENERATE expression [, expression ...]; };</pre> <p>Loop through each tuple in <code>nested_alias</code> and generate new tuple(s). At least one of the fields of <code>nested_alias</code> should be a bag. <code>DISTINCT</code>, <code>FILTER</code>, <code>LIMIT</code>, <code>ORDER</code>, and <code>SAMPLE</code> are allowed operations in <code>nested_op</code> to operate on the inner bag(s).</p>
JOIN	<pre>alias = JOIN alias BY field_alias, alias BY field_alias [, alias BY field_alias _] [USING "replicated"] [PARALLEL n];</pre> <p>Compute inner join of two or more relations based on common field values. When using the <code>replicated</code> option, Pig stores all relations after the first one in memory for faster processing. You have to ensure that all those smaller relations together are indeed small enough to fit in memory.</p> <p>Under <code>JOIN</code>, when the input relations are flat, the output relation is also flat. In addition, the number of fields in the output relation is the sum of the number of fields in the input relations, and the output relation's schema is a concatenation of the input relations' schemas.</p>
GROUP	<pre>alias = GROUP alias { [ALL] [BY ([field_alias [, field_alias]] * [expression]]) [PARALLEL n];</pre> <p>Within a single relation, group together tuples with the same group key. Usually the group key is one or more fields, but it can also be the entire tuple (*) or an expression. One can also use <code>GROUP alias ALL</code> to group all tuples into one group.</p> <p>The output relation has two fields with autogenerated names. The first field is always named "group" and it has the same type as the group key. The second field takes the name of the input relation and is a bag type. The schema for the bag is the same as the schema for the input relation.</p>

COGROUP	<pre>alias = COGROUP alias BY field_alias [INNER OUTER] , alias BY field_alias [INNER OUTER] [PARALLEL n];</pre> <p>Group tuples from two or more relations, based on common group values.</p> <p>The output relation will have a tuple for each unique group value. Each tuple will have the group value as its first field. The second field is a bag containing tuples from the first input relation with matching group value. Ditto for the third field of the output tuple.</p> <p>In the default OUTER join semantic, all group values appearing in any input relation are represented in the output relation. If an input relation doesn't have any tuple with a particular group value, it will have an empty bag in the corresponding output tuple. If the INNER option is set for a relation, then only group values that exist in that input relation are allowed in the output relation. There can't be an empty bag for that relation in the output.</p> <p>You can group on multiple fields. For this, you have to specify the fields in a comma-separated list enclosed by parentheses for <code>field_alias</code>.</p> <p>COGROUP (with INNER) and JOIN are similar except that COGROUP generates nested output tuples.</p>
CROSS	<pre>alias = CROSS alias, alias [, alias ...] [PARALLEL n];</pre> <p>Compute the (flat) cross-product of two or more relations. This is an expensive operation and you should avoid it as far as possible.</p>
ORDER	<pre>alias = ORDER alias BY { * [ASC DESC] field_alias [ASC DESC] [, field_alias [ASC DESC] ...] } [PARALLEL n];</pre> <p>Sort a relation based on one or more fields. If you retrieve the relation right after the ORDER operation (by DUMP or STORE), it's guaranteed to be in the desired sorted order. Further processing (FILTER, DISTINCT, etc.) may destroy the ordering.</p>
STREAM	<pre>alias = STREAM alias [, alias ...] THROUGH ('command' cmd_alias) [AS schema] ;</pre> <p>Process a relation with an external script.</p>