Pig

Introduction

Pig is made up of Pig Latin (the language used to express data flows), and the execution environment to run Pig Latin programs. A Pig Latin program describes a data flow which the Pig execution environment translates into an executable representation and runs. Under the covers, Pig turns them into a series of MapReduce jobs. Pig is supportive of queries as it provides commands for introspecting the data structures in your program. It can also perform a sample run on a representative subset of your input data. Pig is also extensible: almost all parts of the processing path can be customized (loading, storing, filtering, grouping, ordering, joining) by user-defined functions (UDFs). Pig Latin is similar to SQL. The most significant difference is that Pig Latin is a data flow programming language, whereas SQL is a declarative programming language. Pig Latin is a step-by-step set of operations on an input relation, each step is a single transformation. SQL statements are a set of constraints that taken together defines the output. In many ways, Pig Latin figures out how to turn a declarative statement into a system of sets, much like a RDBMS query planner. RDBMS store data in tables with tightly predefined schema, Pig is more relaxed, schema can be defined at runtime although option. Pig does not support random reads or queries in the orders of tens of milliseconds, nor does it support random writes to update small portions of data; all writes are bul streaming writes, just like MapReduce.

Set-up and first run

Download and Install

Download a stable release from http://hadoop.apache.org/pig/releases.html and unpack the tarball in a suitable working directory:

```
% tar xzf pig-n.n.n.tar..gz
```

Add Pig's binary directory to command-line path:

```
% export PIG_INSTALL=/home/username/pig-n.n.n
% export PATH=$PATH:$PIG_INSTALL/bin
```

Set `JAVA_HOME` environment variable to a suitable Java library

Try typing `pig -help` to test if set-up is successful, usage instructions for pig will be shown

Execution Types

Pig has two execution modes: local and Hadoop

Local mode

Pig runs a single JVM and access the local filesystem. This mode is suitable for trying out pig on small datasets. The execution type is set using the -x or -exectype option:

```
% pig -x local
grunt>
```

This starts the Pig interactive shell, Grunt.

Hadoop mode

Pig translates queries into MapReduce jobs and runs them on a Hadoop cluster. Suitable to run on large datasets. The environment variable `PIG_HADOOP_VERSION` is used to tell Pig the version of Hadoop it is connecting to. For examples the following command will allow Pig to connect to any 0.18.x version of Hadoop:

```
% export PIG_HADOOP_VERSION=18
```

Next, point Pig at the cluster's namenode and jobtracker. If a Hadoop site file that defines `fs.default.name` and `mapred.job.tracker` already exists, add Hadoop's configuration directory to Pig's classpath:

```
% export PIG_CLASSPATH=$HADOOP_INSTALL/conf/
```

Alternatively, create a *pig.properties* file in the Pig's *conf* directory (may need to create this directory), and set two properties inside:

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

Once configured, launch Pig by setting the `-x` option to `mapreduce` or omit it entirely, as Hadoop mode is default. Console will print out the filesystem and jobtracker that it has connected to.

Running Pig Programs

There are three ways of executing Pig programs:

Script- Runs a script file that contains Pig commands. Use `-e` option to run a script specified as a string on the command line:

```
% pig script.pig
% pig -e script_goes_here
```

Grunt- Interactive shell for running Pig commands, Grunt is started when no file is specified for Pig to run:

```
%pig
grunt>
```

It is also possible to run Pig scripts from Grunt using `run` and `exec`:

```
grunt> run script.pig
grunt> exec script.pig
```

Embedded- Run Pig programs from Java, more details on the Pig wiki at http://wiki.apache.org/pig/EmbeddedPig

Grunt

Provides line-editing facilities (Ctrl-E moves cursor to end of line, up or down keys to recall lines in the history buffer). Completion mechanism, which will try to complete Pig Latin keywords and functions when Tab key is pressed:

```
grunt> a = FORE
```

(Tab key pressed)

```
grunt> a = FOREACH
```

Completion tokens are customizable by creating a file named *autocomplete* and placing it on Pig's classpath (such as *conf* directory). File should have one token per line, and no whitespaces.

`help` command lists commands.

`quit` command exits Grunt session.

Pig Latin Editors

PigPen for Eclipse. Includes a Pig script text editor, example generator, button for running the script on a Hadoop cluster, operator graph window for visualizing data flow: http://wiki.apache.org/pig/PigPen

Pig Latin

Structure

A Pig Latin program consists of a collection of statements. A statement is an operation or command:

```
      records_by_year = GROUP records BY year;
      ls /
```

Statements are usually terminated with a semicolon. Statements that have to be terminated with a semicolon can span multiple lines for readability:

```
      records = LOAD 'input/salesdata.txt'
            AS (year:int, storelocation:chararray, salesfigures:int);
```

Commenting

```
      -- single-line comments from first hyphen till end of line
      DUMP A; --shows A
      /*
       * Comments spanning
       * multiple lines.
       */
      B = GROUP A /* embedded comment */ BY year
```

Keywords such as operators (LOAD, ILLUSTRATE), commands (cat, ls), expressions (matches, FLATTEN), and functions (DIFF, MAX) cannot be used as identifiers. Operators and commands are not case-sensitive to make interactive use more forgiving; aliases and functions 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 problems, the interpreter will halt and display and error. The interpreter builds a logical plan for each operation, and adds it to the logical plan for the program so far. No data processing takes place when the logical plan of the program is being built. When the interpreter sees a LOAD statement, it merely confirms the syntax is error-free and proceeds to add it to its logical plan. Similarly, no processing is done when the interpreter sees FOREACH... GENERATE statements or other operator commands, it validates each statement and adds them to the logical plan. The trigger for Pig to start processing is a DUMP or STORE statement, which causes the logical plan to be compiled into a physical plan and executed. In local mode, Pig will create a physical plan that runs in a single local JVM, whereas in Hadoop mode, Pig will create a series of MapReduce jobs.

Expressions

An expression is something that is evaluated to yield a value.

Constant	Literal	constant value	1.0, 'a'
Field(by position)	$n	Field in posn n (0-based)	$0
Field(by name)	f	Field name f	year
Projection	c.$n, c.f	Field in container c	records.$0, records.year
Map lookup	m#k	value associated with key k in map m	items#'coat'
Cast	(t) f	Cast of field f to type t	(int)year

Arithmetic	x+y, x-y	Add, subtract	$1 + $2
	x*y, x/y	Multiply, divide	
	x%y, +x, -x	Modulo, unary pos/neg	
Conditional	x ? y : z	y if x is true, z otherwise	quality == 0 ? 0 : 1
Comparison	x==y, x!=y	Equals, not equals	
	x>y, x<y	Gr than, less than	
	x>=y, x<=y	Gr eq than, less eq than	
	x matches y	Pattern matching with regular expression	quality matches ‘[01459]’
	x is null	is null	
	x is not null		
Boolean	x or y	Logical Or	q == 0 or q == 1
	x and y	Logical and	q == 0 and r == 1
	not x	Logical negation	not quality matches ‘[012]’
Functional	fn(f1,f2,...)	Invocation of function fn on fields f1, f2, etc.	isGood(quality)
Flatten	FLATTEN(f)	Removal of nesting from bags and tuples	FLATTEN(group)

Types

Simple atomic types representing numbers, binary and text

int	32-bit signed integer	1
long	64-bit signed integer	1L
float	32-bit floating-point number	1.0F
double	64-bit floating-point number	1.0
bytearray	Byte array	
chararray	Character array in UTF-16 format	‘a’

Complex types representing nested structures

tuple	Sequence of fields of any time	(1, ‘apple’)
bag	An unordered collection of tuples	{(1, ‘apple’),(2)}
map	A set of key-value pairs, keys must be atoms, values of any type	[1#’apple’]

Schemas

Use an AS clause in a LOAD statement to attach a schema to a relation:

```
grunt> records = LOAD ‘input/salesdata.txt’
>>   AS (year:int, storelocation:chararray, salesfigures:int);
grunt> DESCRIBE records;
records:{year:int, storelocation:chararray, salesfigures:int}
```

It is possible to omit type declarations completely, and the types of the fields will be declared to the default: bytearray. There is no need to specify types for every field:

```
.grunt> records = LOAD ‘input/salesdata.txt’
      >>   AS (year:int, storelocation, salesfigures);
```

There is no way to specify the type of a field without naming it.

Fields in a relation with no schema can be referenced only using positional notation: $0 refers to the first field in a relation, $1 to second and so on.

Validation and nulls

Trying to load a string into a column declared to be a numeric type will fail. If the value cannot be cast to the type declared in the schema, Pig will substitute it with a null value. It is displayed as an absence of value when DUMP on console. We can pull out invalid records by a number of ways:

```
    grunt> corrupt_records = FILTER records BY location is null;
    grunt> SPLIT records INTO good_records IF id is not null,
    >> bad_records if id is null;
```

And count them by grouping the records:

```
    grunt> grouprecords = GROUP corrupt_records ALL;
    grunt> all_grouped = FOREACH grouprecords GENERATE group,
    >> COUNT(corrupt_records);
```

Schema merging

Pig can figure out the resulting schema based on the input schema. In cases like LIMIT, the schema does not change from input to output. For UNION, if the schemas are incompatible, the schema of the result is unknown.
The schema of a relation can be described by using the DESCRIBE operator. Use FOREACH... GENERATE... AS to redefine schemas.

Functions

Five types of functions
Eval function- Takes one or more expressions and returns another expression, e.g. MAX

AVG	Calculates the average (mean) value of entries in a bag
CONCAT	Concatenates two bytearrays/chararrays together
COUNT	Counts the number of entries in a bag
DIFF	Calculates the set different in two bags
MAX	Calculates the maximum value of entries in a bag
MIN	Calculates the minimum value of entries in a bag
SIZE	Calculates the size of a type, no. of chars for chararrays, no. of bytes of byte arrays, number of entries for containers
SUM	Calculates the sum of values of entries in a bag
TOKENIZE	Tokenizes a character array into a bag of its constituent words

Filter function- Special type of eval function that returns a logical boolean result, e.g. IsEmpty
Comparison function- Function that impose an ordering on a pair of tuples
Load function- Specifies how to load data into a relation from external
Store function- Specifies how to save contents of relation into external storage

PigStorage	Loads or stores relations using a field-delimited text format. Each line is broken into fields using a configurable field delimiter (default is tab). Default storage mode when none specified.
BinStorage	Loads or stores relations from or to binary files. Use Hadoop Writable objects.
BinaryStorage	Loads or stores relations containing only single-field tuples

	with value of type bytearray.
TextLoader	Loads relations from a plain-text format. Each line corresponds to a tuple whose single field is the line of text.
PigDump	Stores relations by writing toString() representation of tuples, one per line.

User-Defined Functions

UDFs are written in Java. They help make Pig scripts more concise and encapsulates the logic in one place so it can easily be reused elsewhere

Filter UDF

To change this line:

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

Into:

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

All filter functions are subclasses of `FilterFunc`, which itself is a subclass of `EvalFunc`. `EvalFunc` looks like this:

```
      public abstract class EvalFunc<T> {
            public abstract T exec(Tuple input) throws IOException;
      }
```

`EvalFunc` ’s only abstract method, `exec()`, takes a tuple and returns a single value, the (parameterized) type `T`. For `FilterFunc`, `T` is `Boolean`, so the method should return true only for those tuples that should not be filtered out.

For the quality filter, write a class `IsGoodQuality`, that extends `FilterFunc` and implements the `exec()` method:

```
      public class IsGoodQuality extends FilterFunc {
        @Override
        public Boolean exec(Tuple tuple) throws IOException {
// ensures the tuple is not empty before filter
          if ( tuple == null || tuple.size() == 0) {
            return fase;
          }
          try {
// the Tuple class is a list of objects with associated types
// we are concerned with the first field of the tuple
            Object object = tuple.get(0);
              if (object == null) {
                return false;
              }
// if its not null, cast it and check it, returning true
// if it matches the specified readings
              int i = (Integer) object;
              return i == 0 || i == 1 || i == 4 || i == 5 ||
                i == 9;
          } catch (ExecException e) {
            throw new IOException(e);
          }
        }
      }
```

Compile the code, package it into a JAR file, and register the JAR file into Pig:

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

Invoke the function:

```
grunt> filtered_records = FILTER records BY temperature != 9999
>>    AND IsGoodQuality(quality);
```

When running in distributed Hadoop mode, Pig will ensure that the JAR files get shipped to the cluster. For the UDF above, Pig looks for a class with the name IsGoodQuality, which it finds in the pig.jar that we registered. We can shorten the function name by defining an alias:

```
grunt> DEFINE isGood com.hadoop.pig.IsGoodQuality();
grunt> filtered_records = FILTER records BY temperature != 9999
>>    AND isGood(quality);
```

Leveraging types

This filter works when the quality field is declared to be type `int`. When the type is absent, quality is the default type `bytearray`, represented by `DataByteArray` class, and because `DataByteArray` is not an Integer, cast fails. One way to fix this is to convert the field to an integer in the exec() method. But a better way to fix this is to tell Pig the types of the fields that the function expects. The `getArgToFuncMapping()` method on `EvalFunc` is provided for precisely this reason. We can override it to tell Pig that the first field should be an integer.

```
@Override
public List<FuncSpec> getArgToFuncMapping() throws
  FrontendException {
    List<FuncSpec> funcSpecs = new ArrayList<FuncSpec>();
      funcSpecs.add(new FuncSpec(this.getClass().getName(),
        new Schema(new Schema.FieldSchema(null,
          DataType.INTEGER))));
  return funcSpecs;
}
```

This method returns a `FuncSpec` object corresponding to each of the fields of the tuple that is passed to the `exec()` method. Here there is a single field, and we construct and anyonymous `FieldSchema` (the name is passed as `null`, since Pig ignores the name when doing type conversion). The type is specified using the `INTEGER` constant on Pig's `DataType` class.

With this function, Pig will attempt to convert the argument passed to the function into an integer. If the field cannot be converted, then a `null` is passed for the field. An `exec()` method always returns false if the field is a `null`.

Final program with the new functions:

```
REGISTER pig.jar;
DEFINE isGood com.hadoop.pig.IsGoodQuality();
records = LOAD ‘input/sample.txt’
  AS (year:chararray, temperature:int, quality:int);
filtered_records = FILTER records BY temperature != 9999
  AND isGood(quality);
…
```

Eval UDF

When writing an eval function, consider what the output schema looks like. In this:

```
b = FOREACH a GENERATE udf($0);
```

If `udf` creates a tuple with scalar fields, Pig can determine B’s schema through reflection. For complex types such as bags, tuples, or maps, Pig needs more help: implement the `outputSchema()` method to give Pig information about the output schema.

Load UDF

A custom load function that can read plain-text column ranges as fields:

```
grunt> records = LOAD ‘input/sample.txt’
>>  USING CutLoadFunc(‘16-19,88-92,93-93’)
>>  AS (year:int, temperature:int, quality:int);
grunt> DUMP records;
(1950,0,1)
(1950,22,1)
(1950,-11,1)
(1949,111,1)
(1945,78,1)
```

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

```
public class CutLoadFunc extends Utf8StorageConverter implements LoadFunc {
  private static final Log LOG =
    LogFactory.getLog(CutLoadFunc.class);
  private static final Charset UTF8 = Charset.forName(“UTF-8”);
  private static final byte RECORD_DELIMITER = (byte) ‘\n’;
  private TupleFactory tupleFactory =
    TupleFactory.getInstance();
  private BufferedPositionedInputStream in;
  private long end = Long.MAX_VALUE;
  private List<Range> ranges;

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

  @Override
  public void bindTo(String fileName,
    BufferedPositionInputStream in,
    long offset, long end) throws IOException {
    this.in = in;
    this.end = end;
    // throw away the first (partial) record - it will be picked up by
    // another instance
    if (offset != 0) {
      getNext();
    }
  }

  @Override
  public Tuple getNext() throws IOException {
    if (in == null || in.getPosition() > end) {
      return null;
    }
    String line;
    while ((line = in.readLine(UTF8, RECORD_DELIMITER))!=null) {
      Tuple tuple = tupleFactory.newTuple(ranges.size());
      for (int i = 0; i < ranges.size(); i++) {
        try {
          Range range = ranges.get(i);
          if (range.getEnd() > line.length()) {
            LOG.warn(String.format(
              "Range end (%s) is longer than line length (%s)",
              range.getEnd(), line.length()));
            continue;
          }
          tuple.set(i, new
            DataByteArray(range.getSubString(line)));
        } catch (ExecException e) {
        throw new IOException(e);
        }
      }
      return tuple;
    }
    return null;
  }

  @Override
  public void fieldsToRead(Schema schema) {
    // cannot use this information to optimize, so ignore it
  }

  @Override
  public Schema determineSchema(String fileName,
    ExecType execType, DataStorage storage) throws IOException {
    // cannot determine schema in general
    return null;
  }
}
```

When Pig is using Hadoop, data loading takes place in the mapper, so it is important that input can be split into portions that are independently handled by each mapper. In Pig, load functions are asked to load a portion of the input, specified as a byte range. The start and end offsets are typically not on record boundaries since they correspond

to HDFS blocks, so on initialization, in the `bindTo()` method, the load function needs to find the start of the next record boundary. The Pig runtime calls `getNext()` repeatedly until it gets past the byte range it was asked to load. At this point, it returns `null` to signal that there is not more tuples to be read.
`CutLoadFunc` is constructed with a string that specifies the column ranges to use for each field. The logic for parsing this string and creating a list of internal `Range` objects that encapsulates these ranges is contained in the `Range` class.
In `CutLoadFunc` 's `bindTo()` method, we find the next record boundary. If it is at the beginning of the stream, we know the stream is positioned at the first record boundary, otherwise we call getNext() to discard the first partial line, since it will be handled in full by another instance of `CutLoadFunc` -- the one whose byte range immediately proceeds the current one.
The role of the `getNext()` implementation is to turn lines of the input file into `Tuple` objects. It does this by means of a `TupleFactory`, a Pig class for creating `Tuple` instances. The `newTuple()` method creates a new tuple with the required number of fields, which is just the number of `Range` classes, and the fields are populated using substrings of the line, which are determined by the `Range` objects.
Using a schema
If the user has specified a schema, then the fields need converting to the relevant types. However, this is performed lazily by Pig, hence the loader should always construct tuples of type bytearray, using the `DataByteArray` type. `CutLoadFunc` extends Pig's `Utf8StorageConverter`, which provides standard conversions between UTF-8 encoded data and Pig data types. In some cases the load function itself can determine the schema, for example self-describing data like XMLor JSON, we could create a schema for Pig by looking at the data. The load function can also determine the schema in other ways, for example via an external file. It will then need to provide an implementaion of `determineSchema()` that returns a schema. However if the user supplies a schema in the `AS` clause of `LOAD`, it takes precedence over the schema returned by the load function's `determineSchema()`.
For `CutLoadFunc`, `determineSchema()` is `null`, so there is a schema only if the user supplies one.

Data Processing Operators

Loading and Storing Data

Using PigStorage to store tuples as plain-text values separated by a colon character:

```
grunt> STORE a INTO 'out' USING PigStorage(':');
grunt> cat out
hello:five:1
bye:three:2
```

PigStorage	Loads or stores relations using a field-delimited text format. Each line is broken into fields using a configurable field delimiter (default is tab). Default storage mode when none specified.
BinStorage	Loads or stores relations from or to binary files. Use Hadoop Writable objects.
BinaryStorage	Loads or stores relations containing only single-field tuples with value of type bytearray.
TextLoader	Loads relations from a plain-text format. Each line corresponds to a tuple whose single field is the line of text.
PigDump	Stores relations by writing toString() representation of tuples, one per line.

Filtering Data

Remove data that is not interesting. By filtering early in the processing pipeline, you can minimize the amount of data flowing through the system.

<u>FILTER .. BY</u>

Selects tuples from relation based on a certain condition. Remove rows.

```
alias = FILTER alias BY expression;
DUMP a;
(1,2,3)
(1,2,2)
(1,2,4)
b = FILTER a BY ($2 < 3) OR ($0 + $1 = $2);
DUMP b;
(1,2,2)
(1,2,3)
```

<u>FOREACH .. GENERATE</u>

Acts of every row in a relation, generates data transformation based on columns of data

Projection: `b = FOREACH a GENERATE $0, $1+$2;`

Nested projection: `c = FOREACH b GENERATE group, a.$0;`

Schema: `b = FOREACH a GENERATE $0+$1 AS sum:int;`

Applying functions: `c = FOREACH b GENERATE group, SUM(a.a1);`

Flatten: `c = FOREACH b GENERATE group, FLATTEN(A);`

Supports nested form for more complex processing

Grouping and Joining Data

JOIN
A classic inner join whereby each match between two relations corresponds to a row in the result. The result's fields are made up of all the fields of all the input relations.

```
      alias = JOIN alias BY {expression..} (, alias BY {expressions...}
)
      DUMP a;
      (2, tie)
      (4, coat)
      (3, hat)
      DUMP b;
      (Joe, 2)
      (Hank, 4)
      (Jerry, 2)
      (Mary, 0)
      c = JOIN a BY $0, b BY $1;
      DUMP c;
      (2, tie, Joe, 2)
      (2, tie, Jerry, 2)
      (4, coat, Hank, 4)
```

COGROUP (OUTER)
Joining will always give a flat structure, a set of tuples. COGROUP creates a nested set of output tuples.

```
      d = COGROUP a BY $0, b BY $1;
      dump d;
      (0, {}, {(Mary,0)})
      (2, {(2,tie)}, {(Joe,2),(Jerry,2)})
      (3, {(3,hat)}, {})
      (4, {(4,coat)},{(Hank,4)})
```

COGROUP generates a tuple for each unique grouping key. The first field of each generated tuple is the key, and the remaining fields are bags of tuples from the relations with a matching key. The first bag contains matching tuples from relation a with the same key while the second contains matching tuples from relation b with the same key. If for a particular key a relation has no matching key, the bag is empty. This is an example of an OUTER join, which is default for COGROUP
COGROUP (INNER)
Suppress rows with empty bags by using the INNER keyword. The INNER keyword is applied per relation. The following suppresses rows when relation a has no match (or it can be seen in a different way: relation a forms the backbone of the results, hence if b has a match for a key which a does not, the tuple is not formed):

```
      e = COGROUP a BY $0, b BY $1;
      DUMP e;
      (2, {(2,tie)}, {(Joe,2),(Jerry,2)})
      (3, {(3,hat)}, {})
      (4, {(4,coat)},{(Hank,4)})
```

CROSS

Cross product (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).
GROUP
Groups data in a single relation, creates a relation whose first field is the grouping field, and 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):
```
f = GROUP a BY SIZE($1);
DUMP f;
(3L, {(2,tie),(3,hat)})
(4L, {(4,coat)}
```
Special grouping operations: ALL and ANY. ALL groups all the tuples in a single group. The common use of this is to count the numbers of tuples in a relation. The ANY keyword is used to group the tuples in a relation randomly, which can be useful for sampling:
```
g = GROUP a BY ANY
DUMP g;
(all, {(2, tie),(4, coat),(3, hat)})
```

Sorting Data
Consider relation data:
```
DUMP data;
(2,3)
(1,2)
(2,4)
```
The order in which the rows are processed, displayed via DUMP, stored via STORED is not guaranteed to be the same every time. The ORDER operator can be used to impose an order on the relation. The default sort order compares fields of the same type using natural orderings, and different types are given an arbitrary, but deterministic, order (a tuple is always “less than” a bag, for example). A different ordering can be imposed with a USING clause that specifies a UDF that extends Pig’s `ComparisonFunc` class. Below sorts data by $0 in a ascending order, and $1 in a descending order:
```
data_sorted = ORDER data BY $0, $1 DESC;
DUMP data_sorted;
(1,2)
(2,4)
(2,3)
```
If we process `data_sorted` further via GENERATE or other similar expression the order is not guaranteed to be the same. Hence we usually order the data just before retrieving the output.
We can use LIMIT to limit the number of results. Usually LIMIT will select any n tuples from a relation, but when use immediately after ORDER, the order is retained:
```
data_sorted_limit = LIMIT data_sorted 2;
DUMP data_sorted_limit;
(1,2)
(2,4)
```
Combining and Splitting Data

Union

The UNION statement is used to combine relations.

```
	DUMP bdata;
	(z,x,8)
	(w,y,1)
	union_data = UNION data, bdata;
	DUMP union_data;
	(1,2)
	(2,3)
	(z,x,8)
	(2,4)
	(w,y,1)
```

Order of tuples in union_data is undefined since relations are unordered. Union of two relations with different schema or with different number of fields is possible. Pig will attempt to merge the schema from the relations but when they are incompatible, the resultant union relation has no schema.

Split

The SPLIT operator partitions a relation into two or more relations:

```
	SPLIT data INTO small_data IF $1 < 3, big_data IF $1 >= 3;
	DUMP small_data;
	(1,2)
	DUMP big_data;
	(2,4)
	(2,3)
```