16.2 PIG

The MAPREDUCE processing model is low-level. The computation of complex tasks with MAPREDUCE typically requires combining several jobs. Frequently used operations such as sort or group must be repeatedly introduced in applications as map/reduce functions, and integrated with more application specific operations. To design large-scale data processing applications, it would be definitely useful to dispose of a language that would save the burden of these low-level tasks while preserving the assets of MAPREDUCE. In some sense, this can be compared to introducing declarative languages such as SQL in databases, to facilitate the task of developing applications and thereby improve the productivity of application programmers.

To illustrate the use of high-level language primitives, we present the PIG environment and PIG (or PIGLATIN) language. In spite of sometimes clumsy ad hoc features, the language is in general quite adapted to standard large scale data processing tasks. Another advantage is that it can be tested with minimal installation overhead. PIG brings two important features with respect to the MAPREDUCE approach: (i) a richer data model, with nested data structures, and (ii) expressive data manipulation primitives that can be combined in *data flows* to obtain complex operations.

In brief, a PIG program takes as input a "bag" represented in a file. We will detail the bag data structure further, but it is a roughly speaking a *nested relation*, i.e., a relation where the entries may themselves be relations. A PIG program also produces a bag, either stored in a file or displayed on screen.

We begin with a short illustrative session, and then develop the data and processing model of PIG. The *Putting into Practice* chapter devoted to HADOOP gives practical hints and exercises to experiment with PIG.

16.2.1 A simple session

Consider the following simple example: given a file with a list of publications in a scientific journal, determine the average number of papers published each year. We use data coming from DBLP, a large collection of information on scientific publications, publicly available[2] in XML.

The PIG loader accepts a variety of input formats. We use here the default file format that it accepts. Each line of the file is interpreted as an entry (here a publication). Within a line, the attributes are separated by *tabs*. Suppose the input consists of the following lines:

```
2005    VLDB J. Model-based approximate querying in sensor networks.
1997    VLDB J. Dictionary-Based Order-Preserving String Compression.
2003    SIGMOD Record   Time management for new faculty.
2001    VLDB J. E-Services - Guest editorial.
2003    SIGMOD Record   Exposing undergraduate students to system internals.
1998    VLDB J. Integrating Reliable Memory in Databases.
1996    VLDB J. Query Processing and Optimization in Oracle Rdb
1996    VLDB J. A Complete Temporal Relational Algebra.
1994    SIGMOD Record   Data Modelling in the Large.
2002    SIGMOD Record   Data Mining: Concepts and Techniques - Book Review.
...
```

[2]*http://www.sigmod.org/dblp/db/index.html*

Each line gives the year a publication was published, the journal it was published in (e.g., the *VLDB Journal*) and its title.

Here is the complete PIG program that computes the average number of publications per year in SIGMOD RECORD.

```
-- Load records from the journal-small.txt file (tab separated)
articles = load '../../data/dblp/journal-small.txt'
    as (year: chararray, journal:chararray, title: chararray) ;
sr_articles = filter articles BY journal=='SIGMOD Record';
year_groups = group sr_articles by year;
avg_nb = foreach year_groups generate group, COUNT(sr_articles.title);
dump avg_nb;
```

When run on a sample file, the output may look as follows:

```
(1977,1)
(1981,7)
(1982,3)
(1983,1)
(1986,1)
...
```

The program is essentially a sequence of operations, each defining a temporary bag that can be used as input of the subsequent operations. It can be viewed as a flow of data transformation, that is linear in its simplest form but can more generally be an *acyclic workflow* (i.e., a directed acyclic graph).

We can run a step-by-step evaluation of this program with the *grunt* command interpreter to better figure out what is going on.

Load and filter. The **load** operator produces as temporary result, a bag named `articles`. PIG disposes of a few atomic types (`int`, `chararray`, `bytearray`). To "inspect" a bag, the interpreter proposes two useful commands: **describe** outputs its type, and **illustrate** produces a sample of the relation's content.

```
grunt> DESCRIBE articles;
articles: {year: chararray,journal: chararray,title: chararray}

grunt> ILLUSTRATE articles;
---------------------------------------------------------------------------
| articles | year: chararray | journal: chararray | title: chararray      |
---------------------------------------------------------------------------
|          | 2003            | SIGMOD Record      | Call for Book Reviews.|
---------------------------------------------------------------------------
```

The file contains a bag of tuples, where the tuple attributes are distinguished by position. After loading, `articles` also contains a bag of tuples, but the tuple attributes are now distinguished by name.

The filter operation simply selects the elements satisfying certain conditions, pretty much like a relational selection.

Group. In the example, the bags resulting from the load or from the filter do not look different than standard relations. However, a difference is that they may have two identical elements. This would happen, in the example, if the file contains two identical lines. Note that this cannot happen in a relation that is a *set* of tuples. Bags allow the repetition of elements. Furthermore, like nested relations, PIG bags can be *nested*. The result of a **group** for instance is a nested bag. In the example, the group operation is used to create a bag with one element for each distinct year:

```
grunt> year_groups = GROUP sr_articles BY year;

grunt> describe year_groups;
year_groups: {group: chararray,
    sr_articles: {year: chararray,journal: chararray,title:chararray}}

grunt> illustrate year_groups;
 group: 1990
 sr_articles:
  {
   (1990, SIGMOD Record, An SQL-Based Query Language For Networks of Relations.),
   (1990, SIGMOD Record, New Hope on Data Models and Types.)
  }
```

PIG represents bags, nested or not, with curly braces `{}`. Observe the `year_groups` example provided by the **illustrate** command. Note that the grouping attribute is by convention named `group`. All the elements with the same year compose a nested bag.

Before detailing PIG, we summarize its main features essentially contrasting it with SQL:

- Bags in PIG allow repeated elements (therefore the term *bag*) unlike relations that are sets of elements.
- Bags in PIG allow nesting as in nested relations, but unlike classical relations.
- As we will see further, in the style of semistructured data, bags also allow further flexibility by not requiring any strict typing, i.e., by allowing heterogeneous collections.
- For processing, PIG is deliberately oriented toward batch transformations (from bags to bags) possibly in multiple steps. In this sense, it may be viewed as closer to a workflow engine than to an SQL processor.

Note that these design choices have clear motivations:

- The structure of a bag is flexible enough to capture the wide range of information typically found in large-scale data processing.
- The orientation toward read/write sequential data access patterns is, of course, motivated by the distributed query evaluation infrastructure targeted by PIG program, and (as we shall see) by the MAPREDUCE processing model.
- Because of the distributed processing, data elements should be processable independently from each other, to make parallel evaluation possible. So language primitives such as references or pointers are not offered. As a consequence, the language is not adapted to problems such as graph problems. (Note that such problems are notoriously difficult to parallelize.)

The rest of this section delves into a more detailed presentation of PIG's design and evaluation.

16.2.2 The data model

As shown by our simple session, a PIG *bag* is a bag of PIG tuples, i.e., a collection with possibly repeated elements. A PIG *tuple* consist of a sequence of values distinguished by their positions or a sequence of (attribute name, attribute value) pairs. Each value is either atomic or itself a bag.

To illustrate subtle aspects of nested representations, we briefly move away from the running example. Suppose that we obtain a nested bag (as a result of previous computations) of the form:

```
a :  { b : chararray, c : { c' : chararray }, d : { d' : chararray } }
```

Examples of tuples in this bag may be:

$$\langle a : \{ \quad \langle b:1, c:\{\langle c':2\rangle,\langle c':3\rangle\}, d:\{\langle d':2\rangle\}\rangle, \qquad \langle b:2, c:\emptyset, d:\{\langle d':2\rangle,\langle d':3\rangle\}\rangle \quad \}\rangle$$

Note that to represent the same bag in the relational model, we would need identifiers for tuples in the entire bag, and also for the tuples in the c and d bags. One could then use a relation over $b_{id}b$, one over $b_{id}c_{id}c$ and one over $b_{id}d_{id}d$:

b_{id}	b
i_1	1
i_2	2

b_{id}	c_{id}	c
i_1	j_1	2
i_1	j_3	3

b_{id}	d_{id}	d
i_1	j_2	2
i_2	j_4	2
i_2	j_5	3

Observe that an association between some b, c and d is obtained by sharing an `id`, and requires a join to be computed. The input and output of a single PIG operation would correspond to several first-normal-form relations[3]. Joins would be necessary to reconstruct the associations. In very large data sets, join processing is very likely to be a serious bottleneck.

As already mentioned, more flexibility is obtained by allowing heterogeneous tuples to cohabit in a same bag. More precisely, the number of attributes in a bag (and their types) may vary. This gives to the programmer much freedom to organize her dataflow by putting together results coming from different sources if necessary.

Returning to the running example, an intermediate structure created by our program (`year_groups`) represents tuples with an atomic `group` value (the year) and a nested `article` value containing the set of articles published that year.

Also, PIG bags introduce lots of flexibility by not imposing a strong typing. For instance, the following is a perfectly valid bag in PIG:

```
{
  (2005, {'SIGMOD Record', 'VLDB J.'}, {'article1', article2'} )
  (2003, 'SIGMOD Record', {'article1', article2'}, {'author1', 'author2'})
}
```

[3]A relation is in *first-normal-form*, 1NF for short, if each entry in the relation is atomic. Nested relations are also sometimes called not-first-normal-form relations.

This is essentially semistructured data, and can be related to the specificity of applications targeted by PIG. Input data sets often come from a non-structured source (log files, documents, email repositories) that does not comply to a rigid data model and needs to be organized and processed on the fly. Recall also that the application domain is typically that of data analysis: intermediate results are not meant to be persistent and they are not going to be used in transactions requiring stable and constrained structures.

PIG has a last data type to facilitate look-ups, namely *maps*. We mention it briefly. A map associates to a key, that is required to be a data atom, an arbitrary data value.

To summarize, every piece of data in PIG is one of the following four types:

- An *atom*, i.e., a simple atomic value.
- A *bag* of tuples (possibly heterogeneous and possibly with duplicates).
- A PIG *tuple*, i.e., a sequence of values.
- A PIG *map* from keys to values.

It should be clear that the model does not allow the definition of constraints commonly met in relational databases: key (primary key, foreign key), unicity, or any constraint that needs to be validated at the collection level. Thus, a collection can be partitioned at will, and each of its items can be manipulated independently from the others.

16.2.3 The operators

Table 16.1 gives the list of the main PIG operators operating on bags. The common characteristic of the unary operations is that they apply on a flow of tuples, that are independently processed one-at-a-time. The semantics of an operation applied to a tuple never depends on the previous or subsequent computations. Similarly, for binary operations: elementary operations are applied to a pair of tuples, one from each bag, independently from the other tuples in the two bags. This guarantees that the input data sets can be distributed and processed in parallel without affecting the result.

Operator	Description
foreach	Apply one or several expression(s) to each of the input tuples.
filter	Filter the input tuples with some criteria.
order	Order an input.
distinct	Remove duplicates from an input.
cogroup	Associate two related groups from distinct inputs.
cross	Cross product of two inputs.
join	Join of two inputs.
union	Union of two inputs (possibly heterogeneous, unlike in SQL).

Table 16.1: List of PIG operators

We illustrate some important features with examples applied to the following tiny data file *webdam-books.txt*. Each line contains a publication date, a book title and the name of an author.

```
1995    Foundations of Databases Abiteboul
1995    Foundations of Databases Hull
1995    Foundations of Databases Vianu
2010    Web Data Management Abiteboul
2010    Web Data Management Manolescu
2010    Web Data Management Rigaux
2010    Web Data Management Rousset
2010    Web Data Management Senellart
```

```
-- Load records from the webdam-books.txt file (tab separated)
books = load '../../data/dblp/webdam-books.txt'
    as (year: int, title: chararray, author: chararray) ;
group_auth = group books by title;
authors = foreach group_auth generate group, books.author;
dump authors;
```

Figure 16.4: Example of **group** and **foreach**

The first example (Figure 16.4) shows a combination of **group** and **foreach** to obtain a bag with one tuple for each book, and a nested list of the authors.

The operator **foreach** applies some expressions to the attributes of each input tuple. PIG provides a number a predefined expressions (projection/flattening of nested sets, arithmetic functions, conditional expressions), and allows User Defined Functions (UDF) as well. In the example, a *projection* expressed as `books.authors` is applied to the nested set result of the **group** operator. The final `authors` nested bag is:

```
(Foundations of Databases,
   {(Abiteboul),(Hull),(Vianu)})
(Web Data Management,
   {(Abiteboul),(Manolescu),(Rigaux),(Rousset),(Senellart)})
```

The **flatten** expression serves to unnest a nested attribute.

```
-- Take the 'authors' bag and flatten the nested set
flattened = foreach authors generate group, flatten(author);
```

Applied to the nested bag computed earlier, **flatten** yields a relation in 1NF:

```
(Foundations of Databases,Abiteboul)
(Foundations of Databases,Hull)
(Foundations of Databases,Vianu)
(Web Data Management,Abiteboul)
(Web Data Management,Manolescu)
(Web Data Management,Rigaux)
(Web Data Management,Rousset)
(Web Data Management,Senellart)
```

The **cogroup** operator collects related information from different sources and gathers them as separate nested sets. Suppose for instance that we also have the following file *webdam-publishers.txt*:

```
Fundations of Databases Addison-Wesley  USA
Fundations of Databases Vuibert France
Web Data Management     Cambridge University Press      USA
```

We can run a PIG program that associates the set of authors and the set of publishers for each book (Figure 16.5).

```
--- Load records from the webdam-publishers.txt file
publishers = load '../../data/dblp/webdam-publishers.txt'
    as (title: chararray, publisher: chararray) ;
cogrouped = cogroup flattened by group, publishers by title;
```

Figure 16.5: Illustration of the **cogroup** operator

The result (limited to *Foundations of databases*) is the following.

```
(Foundations of Databases,
  { (Foundations of Databases,Abiteboul),
    (Foundations of Databases,Hull),
    (Foundations of Databases,Vianu)
  },
  {(Foundations of Databases,Addison-Wesley),
   (Foundations of Databases,Vuibert)
  }
)
```

The result of a **cogroup** evaluation contains one tuple for each group with three attributes. The first one (named `group`) is the identifier of the group, the second and third attributes being nested bags with, respectively, tuples associated to the identifier in the first input bag, and tuples associated to the identifier in the second one. Cogrouping is close to joining the two (or more) inputs on their common identifier, that can be expressed as follows:

```
-- Take the 'flattened' bag, join with 'publishers'
joined = join flattened by group, publishers by title;
```

The structure of the result is however different than the one obtained with **cogroup**.

```
(Foundations of Databases,Abiteboul,Fundations of Databases,Addison-Wesley)
(Foundations of Databases,Abiteboul,Fundations of Databases,Vuibert)
(Foundations of Databases,Hull,Fundations of Databases,Addison-Wesley)
(Foundations of Databases,Hull,Fundations of Databases,Vuibert)
(Foundations of Databases,Vianu,Fundations of Databases,Addison-Wesley)
(Foundations of Databases,Vianu,Fundations of Databases,Vuibert)
```

In this example, it makes sense to apply **cogroup** because the (nested) set of authors and the (nested) set of publishers are independent, and it may be worth considering them as separate bags. The **join** applies a cross product of these sets right away which may lead to more complicated data processing later.

The difference between **cogroup** and **join** is an illustration of the expressiveness brought by the nested data model. The relational join operator must deliver flat tuples, and intermediate states of the result cannot be kept as first class citizen of the data model, although this could sometimes be useful from a data processing point of view. As another illustration, consider the standard SQL **group by** operator in relational databases. It operates in two, non-breakable steps that correspond to a PIG **group**, yielding a nested set, followed by a **foreach**, applying an aggregation function. The following example is a PIG program that computes a 1NF relation with the number of authors for each book.

```
-- Load records from the webdam-books.txt file (tab separated)
books = load 'webdam-books.txt'
    as (year: int, title: chararray, author: chararray) ;
group_auth = group books by title;
authors = foreach group_auth generate group, COUNT(books.author);
dump authors;
```

The possible downside of this modeling flexibility is that the size of a tuple is unbounded: it can contain arbitrarily large nested bags. This may limit the parallel execution (the extreme situation is a bag with only one tuple and very large nested bags), and force some operators to flush their input or output tuple to the disk if the main memory is exhausted.

16.2.4 Using MAPREDUCE to optimize PIG programs

The starting point of this optimization is that a combination of **group** and **foreach** operators of PIG can be almost directly translated into a program using MAPREDUCE. In that sense, a MAPREDUCE job may be viewed as a group-by operator over large scale data with build-in parallelism, fault tolerance and load balancing features. The MAP phase produces grouping keys for each tuple. The shuffle phase of MAPREDUCE puts these keys together in intermediate pairs (akin to the nested bags, result of the PIG **group**). Finally, the REDUCE phase provides an aggregation mechanism to cluster intermediate pairs. This observation is at the core of using a MAPREDUCE environment as a support for the execution of PIG programs.

Basically, each **(co)group** operator in the PIG data flow yields a MAPREDUCE tasks that incorporates the evaluation of PIG operators surrounding the **(co)group**. As previously explained, a *join*, can be obtained using a **cogroup** followed by a flattening of the inner nested bags. So, joins can also benefit from the MAPREDUCE environment.

To conclude, we illustrate such a MAPREDUCE evaluation with two of the examples previously discussed.

Example: group and foreach. In a first example, we use the program given in Figure 16.4, page 345. Following the classical query evaluation mechanism, the compilation transforms this program through several abstraction levels. Three levels are here represented. The "logical" level directly represents the dataflow process. At this point, some limited reorganization may take place. For instance, a **filter** operator should be "pushed" as near as possible to the **load** to decrease the amount of data that needs to be processed.

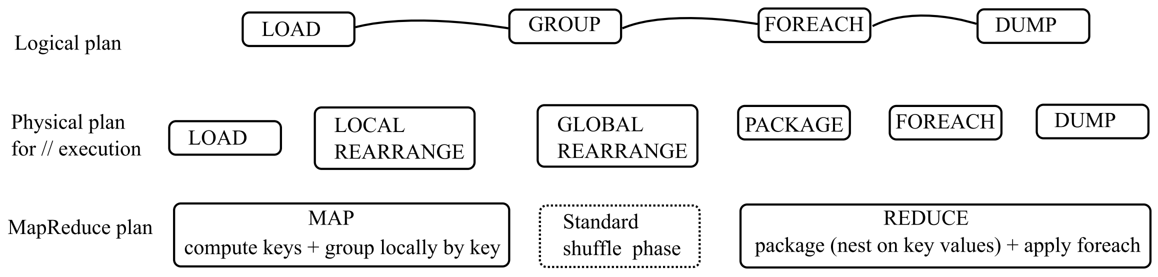

Figure 16.6: Compilation of a PIG program in MAPREDUCE

The second level represents the sequence of physical operations that need to be executed in a parallel query processing environment. PIG targets several parallel execution models, and this intermediate level provides the means to describe and manipulate a physical plan independently from a specific infrastructure.

The blocks in the physical plan introduce some new operators, namely REARRANGE (LOCAL and GLOBAL), and PACKAGE. REARRANGE denotes a physical operator that groups tuples with the same key, via either hashing or sorting. The distinction between LOCAL and GLOBAL stems from the parallelization context. The LOCAL operator takes place on a single node, whereas the GLOBAL operator needs to collect and arrange tuples initially affected to many nodes. The algorithms that implement these variants may therefore be quite different.

PACKAGE relates to the PIG data model. Once a set of tuples sharing the same key are put together by a REARRANGE, a nested bag can be created and associated with the key value to form the typical nested structure produced by the **(co)group** operation. Expressions in the **foreach** operator can then be applied.

The lower level in Figure 16.4 shows the MAPREDUCE execution of this physical plan. There is only one MAPREDUCE job, and the physical execution proceeds as follows:

1. MAP generates the key of the input tuples (in general, this operation may involve the application of one or several functions), and groups the tuples associated to given key in intermediate pairs;

2. the GLOBAL REARRANGE operator is natively supported by the MAPREDUCE framework: recall that intermediate pairs that hash to a same value are assigned to a single Reducer, that performs a merge to "arrange" the tuples with a common key together;

3. the PACKAGE physical operator is implemented as part of the *reduce()* function, that takes care of applying any expression required by the **foreach** loop.

Example: join and group. Our second example involves a **join** followed by a **group**. It returns the number of publishers of Victor Vianu. Note that one might want to remove duplicates from the answer; this is left as an exercise.

Figure 16.8 shows the execution of this program using two MAPREDUCE jobs. The first one carries out the join. Both inputs (books and publishers) are loaded, filtered, sorted on the title,

```
-- Load records from the webdam-books.txt file (tab separated)
books = load '../../data/dblp/webdam-books.txt'
    as (year: int, title: chararray, author: chararray) ;
-- Keep only books from Victor Vianu
vianu = filter books by author == 'Vianu';
--- Load records from the webdam-publishers.txt file
publishers = load '../../data/dblp/webdam-publishers.txt'
    as (title: chararray, publisher: chararray) ;
-- Join on the book title
joined = join vianu by title, publishers by title;
-- Now, group on the author name
grouped = group joined by vianu::author;
-- Finally count the publishers (nb: we should remove duplicates!)
count = foreach grouped generate group, COUNT(joined.publisher);
```

Figure 16.7: A complex PIG program with **join** and **group**

tagged with their provenance, and stored in intermediate pairs (MAP phase). Specifically, the *map()* function receives rows:

1. either from the `books` input with year, title, and author.

2. or from the `publishers` input with title and publisher. again recording provenance.

Each row records its provenance, either `books` or `publishers`.

These intermediate pairs are sorted during the shuffle phase, and submitted to the *reduce()* function. For each key (title), this function must take the set of authors (known by their provenance), the set of publishers (idem), and compute their cross product that constitutes a part of the join result. This output can then be transmitted to the next MAPREDUCE job in charge of executing the **group**.

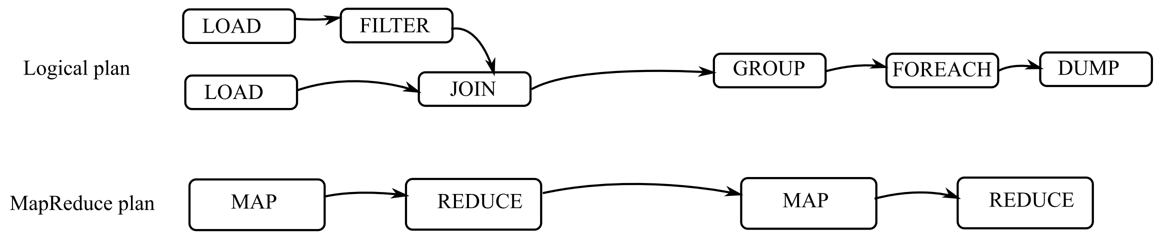

Figure 16.8: A multi-jobs MAPREDUCE execution

Clearly, this complex query would require an important amount of work with MAPREDUCE programming, whereas it is here fulfilled by a few PIG instructions. The advantage is more related to the software engineering process than to the efficiency of the result. the Due to the rather straighforward strategy applied by the PIG evaluator, early performance reports show that PIG execution is, not surprisingly, slightly worse than the equivalent MAPREDUCE direct implementation. This is notably due to the overhead introduced by the translation mechanism. The next section mentions alternative approaches that pursue similar goal that PIG.