Summary: Pig Latin: A Not-So-Foreign Language for Data Processing

1. INTRODUCTION

Authors have developed a new language called Pig Latin that combines the best of both worlds: high-level declarative querying in the spirit of SQL, and low-level, procedural programming a la map-reduce.

Example 1. Suppose we have a table urls: (url, category, pagerank). The following is a simple SQL query that finds, for each sufficiently large category, the average pagerank of high-pagerank urls in that category.

```
SELECT category, AVG(pagerank) FROM urls WHERE pagerank > 0.2 GROUP BY category HAVING
COUNT(*) > 10
```

An equivalent Pig Latin program is the following. (Pig Latin is described in detail in Section 3; a detailed understanding of the language is not required to follow this example.)

```
good_urls = FILTER urls BY pagerank > 0.2;
groups = GROUP good_urls BY category;
big_groups = FILTER groups BY COUNT(good_urls)>10;
output = FOREACH big_groups GENERATE category, AVG(good_urls.pagerank);
```

As evident from the above example, a Pig Latin program is a sequence of steps, much like in a programming language, each of which carries out a single data transformation. This characteristic is immediately appealing to many programmers. At the same time, the transformations carried out in each step are fairly high-level, e.g., filtering, grouping, and aggregation, much like in SQL. The use of such high-level primitives renders low-level manipulations (as required in map-reduce) unnecessary. Pig Latin has several other unconventional features that are important for our setting of casual ad-hoc data analysis by programmers. These features include support for a flexible, fully nested data model, extensive support for user defined functions, and the ability to operate over plain input files without any schema information. Pig Latin also comes with a novel debugging environment that is especially useful when dealing with enormous data sets.

2. FEATURES AND MOTIVATION

2.1 Dataflow Language

The use of high-level, relational-algebra-style primitives, e.g., GROUP, FILTER, allows traditional database optimizations to be carried out, in cases where the system and/or user have enough confidence that query optimization can succeed. For example, suppose one is interested in the set of urls of pages that are classified as spam, but have a high pagerank score. In Pig Latin, one can write:

```
spam_urls = FILTER urls BY isSpam(url);
culprit_urls = FILTER spam_urls BY pagerank > 0.8;
```

where culprit_urls contains the final set of urls that the user is interested in.

2.2 Quick Start and Interoperability

Pig is designed to support ad-hoc data analysis. If a user has a data file obtained, say, from a dump of the search engine logs, she can run Pig Latin queries over it directly. She need only provide a function that gives Pig the ability to parse the content of the file into tuples. There is no need to go through a time-consuming data import process prior to running queries, as in conventional database management systems.

2.3 Nested Data Model

Pig Latin has a flexible, fully nested data model, and allows complex, non-atomic data types such as set, map, and tuple to occur as fields of a table.

2.4 UDFs as First-Class Citizens

To accommodate specialized data processing tasks, Pig Latin has extensive support for user-defined functions (UDFs). Essentially all aspects of processing in Pig Latin including grouping, filtering, joining, and per-tuple processing can be customized through the use of UDFs. The input and output of UDFs in Pig Latin follow our flexible, fully nested data model. Consequently, a UDF to be used in Pig Latin can take non-atomic parameters as input, and also output non-atomic values.

2.5 Parallelism Required

Since Pig Latin is geared toward processing web-scale data, it does not make sense to consider non-parallel evaluation. Consequently, we have only included in Pig Latin a small set of carefully chosen primitives that can be easily parallelized. Language primitives that do not lend themselves to efficient parallel evaluation (e.g., non-equi-joins, correlated subqueries) have been deliberately excluded.

2.7 Debugging Environment

Pig comes with a novel interactive debugging environment that generates a concise example data table illustrating the output of each step of the user's program. The example data is carefully chosen to resemble the real data as far as possible and also to fully illustrate the semantics of the program. Moreover, the example data is automatically adjusted as the program evolves.

3. PIG LATIN

3.1 Data Model

Pig has a rich, yet simple data model consisting of the following four types:

- Atom: An atom contains a simple atomic value such as a string or a number, e.g., `alice'.
- Tuple: A tuple is a sequence of fields, each of which can be any of the data types, e.g., (`alice', `lakers').
- Bag: A bag is a collection of tuples with possible duplicates.
- Map: A map is a collection of data items, where each item has an associated key through which it can be looked up.

3.2 Specifying Input Data: LOAD

The first step in a Pig Latin program is to specify what the input data files are, and how the file contents are to be deserialized, i.e., converted into Pig's data model. An input file is assumed to contain a sequence of tuples, i.e., a bag. This step is carried out by the LOAD command. For example,

```
queries = LOAD `query_log.txt'
USING myLoad()
AS (userId, queryString, timestamp);
```

3.3 Per-tuple Processing: FOREACH

Once input data _le(s) have been specified through LOAD, one can specify the processing that needs to be carried out on the data. One of the basic operations is that of applying some processing to every tuple of a data set. This is achieved through the FOREACH command. For example,

```
expanded_queries = FOREACH queries GENERATE
userId, expandQuery(queryString);
```

3.4 Discarding Unwanted Data: FILTER

Another very common operation is to retain only some subset of the data that is of interest, while discarding the rest. This operation is done by the FILTER command. For example, to get rid of bot traffic in the bag queries:

```
real_queries = FILTER queries BY userId neq `bot';
```

3.5 Getting Related Data Together: COGROUP

Per-tuple processing only takes us so far. It is often necessary to group together tuples from one or more data sets, that are related in some way, so that they can subsequently be processed together. This grouping operation is done by the COGROUP command. For example, suppose we have two data sets that we have specified through a LOAD command:

```
results: (queryString, url, position)
revenue: (queryString, adSlot, amount)
```

3.6 JOIN in Pig Latin

Not all users need the exibility o_ered by COGROUP. In many cases, all that is required is a regular equi-join. Thus, Pig Latin provides a JOIN keyword for equi-joins. For example,

```
join_result = JOIN results BY queryString,
revenue BY queryString;
```

3.7 Other Commands

Pig Latin has a number of other commands that are very similar to their SQL counterparts. These are:

1. UNION: Returns the union of two or more bags.
2. CROSS: Returns the cross product of two or more bags.
3. ORDER: Orders a bag by the specified field(s).
4. DISTINCT: Eliminates duplicate tuples in a bag. This command is just a shortcut for grouping the bag by all fields, and then projecting out the groups.

3.8 Asking for Output: STORE

The user can ask for the result of a Pig Latin expression sequence to be materialized to a file, by issuing the STORE command, e.g.,

```
STORE query_revenues INTO `myoutput' USING myStore();
```

4. FUTURE WORK

There are a number of promising directions that are yet unexplored in the context of the Pig system.

- “Safe" optimizer: One of the arguments that we have put forward in the favor of Pig Latin is that due to its procedural nature, users have tighter control over how their programs are executed.
- User interfaces: The productivity obtained by Pig can be enhanced through the right user interfaces. Pig Pen is a first step in that direction.
- External functions: Pig programs currently run as Java map-reduce jobs, and consequently only support UDFs written in Java.
- Unified environment: Pig Latin does not have control structures like loops and conditionals. If those are needed, Pig Latin, just like SQL, can be embedded in Java with a JDBC-style interface.