Tutorial: High-Level Programming Languages
MapReduce Simplified

Pietro Michiardi

Eurecom
Overview

- **Raising the level of abstraction for processing large datasets**
  - Scalable Algorithm Design is complex using MapReduce
  - Code gets messy, redundant, difficult to re-use

- **Many alternatives exist, based on different principles**
  - Data-flow programming
  - SQL-like declarative programming
  - Additional operators (besides Map and Reduce)

- **Optimization is a hot research topic**
  - Based on traditional RDBMS optimizations
Topics covered

- Review foundations of relational algebra in light of MapReduce

- Hadoop PIG
  - Data-flow language, originated from Yahoo!
  - Internals
  - Optimizations

- Cascading + Scalding

- SPARK\(^1\)

\(^1\)This is an abuse: SPARK is an execution engine that replaces Hadoop, based on Reliable Distributed Datasets, that reside in memory. The programming model is MapReduce, using Scala.
Relational Algebra and MapReduce
Introduction

- **Disclaimer**
  - This is not a full course on Relational Algebra
  - Neither this is a course on SQL

- **Introduction to Relational Algebra, RDBMS and SQL**
  - Follow the video lectures of the Stanford class on RDBMS
    - http://www.db-class.org/
  - Note that you have to sign up for an account

- **Overview of this part**
  - Brief introduction to simplified relational algebra
  - Useful to understand Pig, Hive and HBase
Relational Algebra Operators

- There are a number of operations on data that fit well the relational algebra model
  - In traditional RDBMS, queries involve retrieval of small amounts of data
  - In this course, and in particular in this class, we should keep in mind the particular workload underlying MapReduce
    - Full scans of large amounts of data
    - Queries are not selective, they process all data

A review of some terminology

- A relation is a table
- Attributes are the column headers of the table
- The set of attributes of a relation is called a schema
  Example: $R(A_1, A_2, \ldots, A_n)$ indicates a relation called $R$ whose attributes are $A_1, A_2, \ldots, A_n$
Operators
Operators

Let’s start with an example

- Below, we have part of a relation called *Links* describing the structure of the Web
- There are two *attributes*: *From* and *To*
- A row, or *tuple*, of the relation is a pair of URLs, indicating the existence of a link between them
- The number of tuples in a real dataset is in the order of billions ($10^9$)

<table>
<thead>
<tr>
<th>From</th>
<th>To</th>
</tr>
</thead>
<tbody>
<tr>
<td>url1</td>
<td>url2</td>
</tr>
<tr>
<td>url1</td>
<td>url3</td>
</tr>
<tr>
<td>url2</td>
<td>url3</td>
</tr>
<tr>
<td>url2</td>
<td>url4</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Operators

- Relations (however big) can be stored in a distributed filesystem
  - If they don’t fit in a single machine, they’re broken into pieces (think HDFS)

- Next, we review and describe a set of relational algebra operators
  - Intuitive explanation of what they do
  - “Pseudo-code” of their implementation in/by MapReduce
Operators

- **Selection**: $\sigma_C(R)$
  - Apply condition $C$ to each tuple of relation $R$
  - Produce in output a relation containing only tuples that satisfy $C$

- **Projection**: $\pi_S(R)$
  - Given a subset $S$ of relation $R$ attributes
  - Produce in output a relation containing only tuples for the attributes in $S$

- **Union, Intersection and Difference**
  - Well known operators on sets
  - Apply to the set of tuples in two relations that have the same schema
  - Variations on the theme: work on bags
Operators

- **Natural join** $R \bowtie S$
  - Given two relations, *compare each pair of tuples*, one from each relation
  - If the tuples agree on all the attributes common to both schema → produce an output tuple that has components on each attribute
  - Otherwise produce nothing
  - *Join condition* can be on a subset of attributes

- **Let’s work with an example**
  - Recall the *Links* relation from previous slides
  - Query (or data processing job): *find the paths of length two in the Web*
Join Example

Informally, to satisfy the query we must:

- find the triples of URLs in the form \((u, v, w)\) such that there is a link from \(u\) to \(v\) and a link from \(v\) to \(w\)

Using the join operator

- Imagine we have two relations (with different schemas), and let’s try to apply the natural join operator
- There are two copies of \(\text{Links}: L_1(U_1, U_2)\) and \(L_2(U_2, U_3)\)
- Let’s compute \(L_1 \bowtie L_2\)
  - For each tuple \(t_1\) of \(L_1\) and each tuple \(t_2\) of \(L_2\), see if their \(U_2\) component are the same
  - If yes, then produce a tuple in output, with the schema \((U_1, U_2, U_3)\)
Join Example

- **What we have seen is called (to be precise) a self-join**
  - **Question**: How would you implement a self join in your favorite programming language?
  - **Question**: What is the time complexity of your algorithm?
  - **Question**: What is the space complexity of your algorithm?

- **To continue the example**
  - Say you are not interested in the entire two-hop path but just the start and end nodes
  - Then you do a projection and the notation would be: \( \pi_{U_1, U_3}(L_1 \Join L_2) \)
Grouping and Aggregation: $\gamma_X(R)$

- Given a relation $R$, partition its tuples according to their values in one set of attributes $G$
  - The set $G$ is called the grouping attributes
- Then, for each group, aggregate the values in certain other attributes
  - Aggregation functions: SUM, COUNT, AVG, MIN, MAX, ...

In the notation, $X$ is a list of elements that can be:

- A grouping attribute
- An expression $\theta(A)$, where $\theta$ is one of the (five) aggregation functions and $A$ is an attribute NOT among the grouping attributes
Operators

- **Grouping and Aggregation:** $\gamma_X(R)$
  - The result of this operation is a relation with one tuple for each group
  - That tuple has a component for each of the grouping attributes, with the value common to tuples of that group
  - That tuple has another component for each aggregation, with the aggregate value for that group

- **Let’s work with an example**
  - Imagine that a social-networking site has a relation `Friends(User, Friend)`
  - The tuples are pairs $(a, b)$ such that $b$ is a friend of $a$
  - Query: compute the number of friends each member has
Grouping and Aggregation Example

How to satisfy the query

\[ \gamma_{User, \text{COUNT}(\text{Friend})} (\text{Friends}) \]
- This operation groups all the tuples by the value in their first component
- There is one group for each user
- Then, for each group, it counts the number of friends

Some details
- The \text{COUNT} operation applied to an attribute does not consider the values of that attribute
- In fact, it counts the number of tuples in the group
- In SQL, there is a “count distinct” operator that counts the number of different values
MapReduce implementation of (some) Relational Operators
Computing Selection

- In practice, selection does not need a full-blown MapReduce implementation
  - They can be implemented in the map portion alone
  - Actually, they could also be implemented in the reduce portion

- A MapReduce implementation of $\sigma_C(R)$
  - **Map:**
    - For each tuple $t$ in $R$, check if $t$ satisfies $C$
    - If so, emit a key/value pair $(t, t)$
  - **Reduce:**
    - Identity reducer
    - **Question:** single or multiple reducers?

- **NOTE:** the output is not exactly a relation
  - **WHY?**
Computing Projections

- Similar process to selection
  - But, projection may cause same tuple to appear several times

- A MapReduce implementation of $\pi_S(R)$
  
  **Map:**
  - For each tuple $t$ in $R$, construct a tuple $t'$ by eliminating those components whose attributes are not in $S$
  - Emit a key/value pair $(t', t')$

  **Reduce:**
  - For each key $t'$ produced by any of the Map tasks, fetch $t', [t', \cdots, t']$
  - Emit a key/value pair $(t', t')$

- NOTE: the reduce operation is duplicate elimination
  - This operation is associative and commutative, so it is possible to optimize MapReduce by using a Combiner in each mapper
Computing Unions

- Suppose relations $R$ and $S$ have the same schema
  - Map tasks will be assigned chunks from either $R$ or $S$
  - Mappers don’t do much, just pass by to reducers
  - Reducers do duplicate elimination

- A MapReduce implementation of union
  - **Map:**
    - For each tuple $t$ in $R$ or $S$, emit a key/value pair $(t, t)$
  - **Reduce:**
    - For each key $t$ there will be either one or two values
    - Emit $(t, t)$ in either case
Computing Intersections

- **Very similar to computing unions**
  - Suppose relations $R$ and $S$ have the same schema
  - The map function is the same (an identity mapper) as for union
  - The reduce function must produce a tuple only if both relations have that tuple

- **A MapReduce implementation of intersection**
  - **Map:** For each tuple $t$ in $R$ or $S$, emit a key/value pair $(t, t)$
  - **Reduce:** If key $t$ has value list $[t, t]$ then emit the key/value pair $(t, t)$
  - Otherwise, emit the key/value pair $(t, \text{NULL})$
Computing difference

- Assume we have two relations $R$ and $S$ with the same schema
  - The only way a tuple $t$ can appear in the output is if it is in $R$ but not in $S$
  - The map function can pass tuples from $R$ and $S$ to the reducer
  - NOTE: it must inform the reducer whether the tuple came from $R$ or $S$

A MapReduce implementation of difference

**Map:**
- For a tuple $t$ in $R$ emit a key/value pair $(t, 'R')$ and for a tuple $t$ in $S$, emit a key/value pair $(t, 'S')$

**Reduce:**
- For each key $t$, do the following:
  - If it is associated to '$R$', then emit $(t, t)$
  - If it is associated to ['$R$', '$S$'] or ['$S$', '$R$'], or ['$S$'], emit the key/value pair $(t, \text{NULL})$
Computing the natural Join

- **This topic is subject to continuous refinements**
  - There are many JOIN operators and many different implementations
  - We will see some of them in more detail in the Lab

- **Let’s look at two relations** $R(A, B)$ and $S(B, C)$
  - We must find tuples that agree on their $B$ components
  - We shall use the $B$-value of tuples from either relation as the key
  - The value will be the other component and the name of the relation
  - That way the reducer knows from which relation each tuple is coming from
Computing the natural Join

- **A MapReduce implementation of Natural Join**
  
  **Map:**
  - For each tuple \((a, b)\) of \(R\) emit the key/value pair \((b, ('R', a))\)
  - For each tuple \((b, c)\) of \(S\) emit the key/value pair \((b, ('S', c))\)

  **Reduce:**
  - Each key \(b\) will be associated to a list of pairs that are either \(('R', a)\) or \(('S', c)\)
  - Emit key/value pairs of the form
    \((b, [(a_1, b, c_1), (a_2, b, c_2), \cdots, (a_n, b, c_n)])\)

- **NOTES**
  - **Question:** what if the MapReduce framework wouldn’t implement the distributed (and sorted) group by?
  - In general, for \(n\) tuples in relation \(R\) and \(m\) tuples in relation \(S\) all with a common \(B\)-value, then we end up with \(nm\) tuples in the result
  - If all tuples of both relations have the same \(B\)-value, then we’re computing the **cartesian product**
Let $R(A, B, C)$ be a relation to which we apply $\gamma_{A, \theta(B)}(R)$

- The map operation prepares the grouping
- The grouping is done by the framework
- The reducer computes the aggregation
- Simplifying assumptions: one grouping attribute and one aggregation function

MapReduce implementation of $\gamma_{A, \theta(B)}(R)$

**Map:**
- For each tuple $(a, b, c)$ emit the key/value pair $(a, b)$

**Reduce:**
- Each key $a$ represents a group
- Apply $\theta$ to the list $[b_1, b_2, \cdots, b_n]$
- Emit the key/value pair $(a, x)$ where $x = \theta([b_1, b_2, \cdots, b_n])$
Hadoop PIG
Introduction

- Collection and analysis of enormous datasets is at the heart of innovation in many organizations
  - E.g.: web crawls, search logs, click streams

- Manual inspection before batch processing
  - Very often engineers look for exploitable trends in their data to drive the design of more sophisticated techniques
  - This is difficult to do in practice, given the sheer size of the datasets

- The MapReduce model has its own limitations
  - One input
  - Two-stage, two operators
  - Rigid data-flow
MapReduce limitations

- **Very often tricky workarounds are required**\(^2\)
  - This is very often exemplified by the difficulty in performing **JOIN** operations

- **Custom code required even for basic operations**
  - Projection and Filtering need to be “rewritten” for each job

→ Code is difficult to reuse and maintain
→ Semantics of the analysis task are obscured
→ Optimizations are difficult due to opacity of **Map** and **Reduce**

\(^2\)The term workaround should not only be intended as negative.
Use Cases

Rollup aggregates

- Compute aggregates against user activity logs, web crawls, etc.
  - Example: compute the frequency of search terms aggregated over days, weeks, month
  - Example: compute frequency of search terms aggregated over geographical location, based on IP addresses

Requirements

- Successive aggregations
- Joins followed by aggregations

Pig vs. OLAP systems

- Datasets are too big
- Data curation is too costly
Use Cases

Temporal Analysis

- **Study how search query distributions change over time**
  - Correlation of search queries from two distinct time periods (groups)
  - Custom processing of the queries in each correlation group

- **Pig supports operators that minimize memory footprint**
  - Instead, in a RDBMS such operations typically involve *JOINS* over very large datasets that do not fit in memory and thus become slow
Use Cases

Session Analysis

- Study sequences of page views and clicks

- Example of typical aggregates
  - Average length of user session
  - Number of links clicked by a user before leaving a website
  - Click pattern variations in time

- Pig supports advanced data structures, and UDFs
Pig Latin

- Pig Latin, a high-level programming language developed at Yahoo!
  - Combines the best of both declarative and imperative worlds
    - High-level declarative querying in the spirit of SQL
    - Low-level, procedural programming á la MapReduce

- Pig Latin features
  - Multi-valued, nested data structures instead of flat tables
  - Powerful data transformations primitives, including joins

- Pig Latin program
  - Made up of a series of operations (or transformations)
  - Each operation is applied to input data and produce output data
  → A Pig Latin program describes a data flow
Example 1

Pig Latin premiere

- Assume we have the following table:

```
urls: (url, category, pagerank)
```

- Where:
  - `url`: is the url of a web page
  - `category`: corresponds to a pre-defined category for the web page
  - `pagerank`: is the numerical value of the pagerank associated to a web page

→ Find, for each sufficiently large category, the average page rank of high-pagerank urls in that category
Example 1

SQL

SELECT category, AVG(pagerank)
FROM urls WHERE pagerank > 0.2
GROUP BY category HAVING COUNT(*) > $10^6$
Example 1

Pig Latin

good_urls = FILTER urls BY pagerank > 0.2;
groups = GROUP good_urls BY category;
big_groups = FILTER groups BY COUNT(good_urls) > 10^6;
output = FOREACH big_groups GENERATE
category, AVG(good_urls.pagerank);
Pig Execution environment

- **How do we go from Pig Latin to MapReduce?**
  - The Pig system is in charge of this
  - Complex execution environment that interacts with Hadoop MapReduce
  - The programmer focuses on the data and analysis

- **Pig Compiler**
  - Pig Latin operators are translated into MapReduce code
  - **NOTE**: in some cases, hand-written MapReduce code performs better

- **Pig Optimizer**
  - Pig Latin data flows undergo an (automatic) optimization phase
  - These optimizations are borrowed from the RDBMS community
Pig and Pig Latin

- **Pig is not a RDBMS!**
  - This means it is not suitable for all data processing tasks

- **Designed for batch processing**
  - Of course, since it compiles to MapReduce
  - Of course, since data is materialized as files on HDFS

- **NOT designed for random access**
  - Query selectivity does not match that of a RDBMS
  - Full-scans oriented!
Comparison with RDBMS

- It may seem that Pig Latin is similar to SQL
  - We’ll see several examples, operators, etc. that resemble SQL statements

- Data-flow vs. declarative programming language
  - Data-flow:
    - Step-by-step set of operations
    - Each operation is a single transformation
  - Declarative:
    - Set of constraints
    - Applied together to an input to generate output

→ With Pig Latin it’s like working at the query planner
Comparison with RDBMS

- **RDBMS store data in tables**
  - Schemas are predefined and strict
  - Tables are flat

- **Pig and Pig Latin work on more complex data structures**
  - Schema can be defined at run-time for readability
  - *Pigs eat anything!*
  - UDF and streaming together with nested data structures make Pig and Pig Latin more flexible
Features and Motivations
Features and Motivations

- **Design goals of Pig and Pig Latin**
  - Appealing to programmers for performing ad-hoc analysis of data
  - Number of features that go beyond those of traditional RDBMS

- **Next: overview of salient features**
  - There will be a dedicated set of slides to optimizations later on
Dataflow Language

- A Pig Latin program specifies a series of steps
  - Each step is a single, high level data transformation
  - Stylistically different from SQL

- With reference to Example 1
  - The programmer supply an order in which each operation will be done

- Consider the following snippet

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

- Data flow optimizations
  - Explicit sequences of operations can be overridden
  - Use of high-level, relational-algebra-style primitives (GROUP, FILTER,...) allows using traditional RDBMS optimization techniques

→ NOTE: it is necessary to check whether such optimizations are beneficial or not, by hand

- Pig Latin allows Pig to perform optimizations that would otherwise by a tedious manual exercise if done at the MapReduce level
Quick Start and Interoperability

- **Data I/O is greatly simplified in Pig**
  - No need to curate, bulk import, parse, apply schema, create indexes that traditional RDBMS require
  - Standard and ad-hoc “readers” and “writers” facilitate the task of ingesting and producing data in arbitrary formats

- **Pig can work with a wide range of other tools**

- **Why RDBMS have stringent requirements?**
  - To enable transactional consistency guarantees
  - To enable efficient point lookup (using physical indexes)
  - To enable data curation on behalf of the user
  - To enable other users figuring out what the data is, by studying the schema
Quick Start and Interoperability

**Why is Pig so flexible?**
- Supports read-only workloads
- Supports scan-only workloads (no lookups)
  → No need for transactions nor indexes

**Why data curation is not required?**
- Very often, Pig is used for ad-hoc data analysis
- Work on temporary datasets, then throw them
  → Curation is an overkill

**Schemas are optional**
- Can apply one on the fly, at runtime
- Can refer to fields using positional notation
  
  ```
  E.g.: good_urls = FILTER urls BY $2 > 0.2
  ```
Nested Data Model

- Easier for “programmers” to think of nested data structures
  - E.g.: capture information about positional occurrences of terms in a collection of documents
  - `Map<documnetId, Set<positions> >`

- Instead, RDBMS allows only flat tables
  - Only atomic fields as columns
  - Require **normalization**
  - From the example above: need to create two tables
    - `term_info: (termId, termString, ...)`
    - `position_info: (termId, documentId, position)`
  - Occurrence information obtained by joining on `termId`, and grouping on `termId`, `documentId`
Nested Data Model

- Fully nested data model (see also later in the presentation)
  - Allows complex, non-atomic data types
  - E.g.: set, map, tuple

- Advantages of a nested data model
  - More natural than normalization
  - Data is often already stored in a nested fashion on disk
    - E.g.: a web crawler outputs for each crawled url, the set of outlinks
    - Separating this in normalized form imply use of joins, which is an overkill for web-scale data
  - Nested data allows to have an algebraic language
    - E.g.: each tuple output by GROUP has one non-atomic field, a nested set of tuples from the same group
  - Nested data makes life easy when writing UDFs
User Defined Functions

- **Custom processing is often predominant**
  - E.g.: users may be interested in performing natural language stemming of a search term, or tagging urls as spam

- **All commands of Pig Latin can be customized**
  - Grouping, filtering, joining, per-tuple processing

- **UDFs support the nested data model**
  - Input and output can be non-atomic
Example 2

- **Continues from Example 1**
  - Assume we want to find for each category, the top 10 urls according to pagerank

  ```pig
  groups = GROUP urls BY category;
  output = FOREACH groups GENERATE category, top10(urls);
  ```

  - `top10()` is a UDF that accepts a set of urls (for each group at a time)
  - it outputs a set containing the top 10 urls by pagerank for that group
  - final output contains non-atomic fields
User Defined Functions

- **UDFs can be used in all Pig Latin constructs**

- **Instead, in SQL, there are restrictions**
  - Only scalar functions can be used in `SELECT` clauses
  - Only set-valued functions can appear in the `FROM` clause
  - Aggregation functions can only be applied to `GROUP BY` or `PARTITION BY`

- **UDFs can be written in Java, Python and Javascript**
  - With streaming, we can use also C/C++, Python, ...

---

3 As of Pig 0.8.1 and later. We will use version 0.10.0 or more.
Handling parallel execution

- Pig and Pig Latin are geared towards parallel processing
  - Of course, the underlying execution engine is MapReduce

- Pig Latin primitives are chosen such that they can be easily parallelized
  - Non-equi joins, correlated sub-queries,... are not directly supported

- Users may specify parallelization parameters at run time
  - Question: Can you specify the number of maps?
  - Question: Can you specify the number of reducers?
Pig Latin
Introduction

- **Not a complete reference to the Pig Latin language**: refer to [1]
  - Here we cover some interesting aspects

- **The focus here is on some language primitives**
  - Optimizations are treated separately
  - How they can be implemented is covered later

- **Examples are taken from [2, 3]**
Data Model

- **Supports four types**
  - *Atom*: contains a simple atomic value as a string or a number, e.g., ‘alice’
  
  - *Tuple*: sequence of *fields*, each can be of any data type, e.g.,
    (‘alice’, ‘lakers’)

  - *Bag*: collection of tuples with possible duplicates. Flexible schema, no need to have the same number and type of fields
    
    \[
    \begin{align*}
    \{ & \text{('alice', 'lakers')} \\
    \{ & \text{('alice', ('iPod', 'apple'))} \\
    \end{align*}
    \]

    The example shows that tuples can be nested
Data Model

- **Supports four types**
  - *Map*: collection of data items, where each item has an associated key for lookup. The schema, as with bags, is flexible.
    - **NOTE**: keys are required to be data atoms, for efficient lookup.

\[
\begin{align*}
\text{‘fan of’} & \rightarrow \left\{ \text{‘lakers’}, \text{‘iPod’} \right\} \\
\text{‘age’} & \rightarrow 20
\end{align*}
\]

- The key ‘fan of’ is mapped to a bag containing two tuples
- The key ‘age’ is mapped to an atom

- Maps are useful to model datasets in which schema may be dynamic (over time)
Structure

- **Pig latin programs are a sequence of steps**
  - Can use an interactive shell (called `grunt`)
  - Can feed them as a “script”

- **Comments**
  - In line: with double hyphens (--)
  - C-style for longer comments (/* ... */)

- **Reserved keywords**
  - List of keywords that can’t be used as identifiers
  - Same old story as for any language
Statements

- As a Pig Latin program is executed, each statement is *parsed*
  - The interpreter builds a logical plan for every relational operation
  - The logical plan of each statement is added to that of the program so far
  - Then the interpreter moves on to the next statement

- **IMPORTANT:** No data processing takes place during construction of logical plan
  - When the interpreter sees the first line of a program, it confirms that it is syntactically and semantically correct
  - Then it adds it to the logical plan
  - It does not even check the existence of files, for data load operations
Statements

→ It makes no sense to start any processing until the whole flow is defined
  ▶ Indeed, there are several optimizations that could make a program more efficient (e.g., by avoiding to operate on some data that later on is going to be filtered)

● The trigger for Pig to start execution are the DUMP and STORE statements
  ▶ It is only at this point that the logical plan is compiled into a physical plan

● How the physical plan is built
  ▶ Pig prepares a series of MapReduce jobs
    ★ In Local mode, these are run locally on the JVM
    ★ In MapReduce mode, the jobs are sent to the Hadoop Cluster
  ▶ IMPORTANT: The command EXPLAIN can be used to show the MapReduce plan
Statements

Multi-query execution

- There is a difference between **dump** and **store**
  - Apart from diagnosis, and interactive mode, in batch mode **store** allows for program/job optimizations

- Main optimization objective: minimize I/O
  - Consider the following example:
    ```pig
    A = LOAD 'input/pig/multiquery/A';
    B = FILTER A BY $1 == 'banana';
    C = FILTER A BY $1 != 'banana';
    STORE B INTO 'output/b';
    STORE C INTO 'output/c';
    ```
Statements

Multi-query execution

- **In the example, relations B and C are both derived from A**
  - Naively, this means that at the first `STORE` operator the input should be read
  - Then, at the second `STORE` operator, the input should be read again

- **Pig will run this as a single MapReduce job**
  - Relation A is going to be read only once
  - Then, each relation B and C will be written to the output
Expressions

- An expression is something that is evaluated to yield a value
  - Lookup on [3] for documentation

\[
\begin{align*}
  t &= (\text{'alice'}, \{ ('lakers', 1), ('iPod', 2) \}, ['age' \rightarrow 20]) \\
\end{align*}
\]

Let fields of tuple \( t \) be called \( f_1, f_2, f_3 \)

<table>
<thead>
<tr>
<th>Expression Type</th>
<th>Example</th>
<th>Value for ( t )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant</td>
<td>'bob'</td>
<td>Independent of ( t )</td>
</tr>
<tr>
<td>Field by position</td>
<td>$0</td>
<td>'alice'</td>
</tr>
<tr>
<td>Field by name</td>
<td>( f_3 )</td>
<td>'age' \rightarrow 20</td>
</tr>
<tr>
<td>Projection</td>
<td>( f_2 . 0 )</td>
<td>{ ('lakers') }</td>
</tr>
<tr>
<td></td>
<td></td>
<td>{ ('iPod') }</td>
</tr>
<tr>
<td>Map Lookup</td>
<td>( f_3 # 'age' )</td>
<td>20</td>
</tr>
<tr>
<td>Function Evaluation</td>
<td>( \text{SUM}(f_2 . 1) )</td>
<td>1 + 2 = 3</td>
</tr>
<tr>
<td>Conditional Expression</td>
<td>( f_3 # 'age' \text{'} \geq 18? )</td>
<td>'adult'</td>
</tr>
<tr>
<td></td>
<td>'adult' : 'minor'</td>
<td></td>
</tr>
<tr>
<td>Flattening</td>
<td>( \text{FLATTEN}(f_2) )</td>
<td>'lakers', 1 \ 'iPod', 2</td>
</tr>
</tbody>
</table>
Schemas

- **A relation in Pig may have an associated schema**
  - This is optional
  - A schema gives the fields in the relations names and types
  - Use the command `DESCRIBE` to reveal the schema in use for a relation

- **Schema declaration is flexible but reuse is awkward**
  - A set of queries over the same input data will often have the same schema
  - This is sometimes hard to maintain (unlike HIVE) as there is no external components to maintain this association

**HINT::** You can write a UDF function to perform a personalized load operation which encapsulates the schema
Validation and nulls

- Pig does not have the same power to enforce constraints on schema at load time as a RDBMS
  - If a value cannot be cast to a type declared in the schema, then it will be set to a null value
  - This also happens for corrupt files

- A useful technique to partition input data to discern good and bad records
  - Use the SPLIT operator
    
    SPLIT records INTO good_records IF temperature is not null, bad_records IF temperature is NULL;
Other relevant information

- **Schema merging**
  - How schema are propagated to new relations?

- **Functions**
  - Look up on the web for *Piggy Bank*

- **User-Defined Functions**
  - Use [3] for an introduction to designing UDFs
Data Processing Operators

Loading and storing data

- **The first step in a Pig Latin program is to load data**
  - What input files are
  - How the file contents are to be deserialized
  - An input file is assumed to contain a sequence of tuples

- **Data loading is done with the LOAD command**

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

Loading and storing data

- The example above specifies the following:
  - The input file is `query_log.txt`
  - The input file should be converted into tuples using the custom `myLoad` deserializer
  - The loaded tuples have three fields, specified by the schema

- Optional parts
  - `USING` clause is optional: if not specified, the input file is assumed to be plain text, tab-delimited
  - `AS` clause is optional: if not specified, must refer to fields by position instead of by name
Data Processing Operators

Loading and storing data

- **Return value of the `LOAD` command**
  - Handle to a bag
  - This can be used by subsequent commands
    → bag handles are only logical
    → no file is actually read!

- **The command to write output to disk is `STORE`**
  - It has similar semantics to the `LOAD` command
Data Processing Operators

Per-tuple processing: Filtering data

- Once you have some data loaded into a relation, the next step is to filter it
  - This is done, e.g., to remove unwanted data
  - **HINT:** By filtering early in the processing pipeline, you minimize the amount of data flowing through the system

- A basic operation is to apply some processing over every tuple of a data set
  - This is achieved with the `FOREACH` command
    ```
    expanded_queries = FOREACH queries GENERATE
    userId, expandQuery(queryString);
    ```
Data Processing Operators

Per-tuple processing: Filtering data

- **Comments on the example above:**
  - Each tuple of the bag queries should be processed **independently**
  - The second field of the output is the result of a UDF

- **Semantics of the `FOREACH` command**
  - There can be no dependence between the processing of different input tuples
  - This allows for an efficient parallel implementation

- **Semantics of the `GENERATE` clause**
  - Followed by a list of expressions
  - Also *flattering* is allowed
    - This is done to eliminate nesting in data
    - Allows to make output data independent for further parallel processing
    - Useful to store data on disk
Data Processing Operators

Per-tuple processing: Discarding unwanted data

- A common operation is to retain a portion of the input data
  - This is done with the `FILTER` command
    ```
    real_queries = FILTER queries BY userId neq 'bot';
    ```

- Filtering conditions involve a combination of expressions
  - Comparison operators
  - Logical connectors
  - UDF
Data Processing Operators

Per-tuple processing: Streaming data

- **The STREAM operator allows transforming data in a relation using an external program or script**
  - This is possible because Hadoop MapReduce supports “streaming”
  - Example:
    \[ C = \text{STREAM} \ A \ \text{THROUGH} \ \text{‘cut -f 2’}; \]
    which use the Unix `cut` command to extract the second field of each tuple in \( A \)

- **The STREAM operator uses PigStorage to serialize and deserialize relations to and from stdin/stdout**
  - Can also provide a custom serializer/deserializer
  - Works well with python
Data Processing Operators

Getting related data together

- It is often necessary to *group* together tuples from one or more data sets
  - We will explore several nuances of “grouping”

- The first grouping operation we study is given by the COGROUP command

Example: Assume we have loaded two relations

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

- results contains, for different query strings, the urls shown as search results, and the positions at which they where shown
- revenue contains, for different query strings, and different advertisement slots, the average amount of revenue
Data Processing Operators

Getting related data together

- Suppose we want to group together all search results data and revenue data for the same query string

```pig
grouped_data = COGROUP results BY queryString, revenue BY queryString;
```

results:
- (queryString, url, rank)
- (lakers, nba.com, 1)
- (lakers, espn.com, 2)
- (kings, nhl.com, 1)
- (kings, nba.com, 2)

revenue:
- (queryString, adSlot, amount)
- (lakers, top, 50)
- (lakers, side, 20)
- (kings, top, 30)
- (kings, side, 10)

```pig
(lakers, {lakers, nba.com, 1}, {lakers, espn.com, 2}),
{lakers, top, 50}, {lakers, side, 20})
(kings, {kings, nhl.com, 1}, {kings, nba.com, 2}),
{kings, top, 30}, {kings, side, 10})
```

```pig
distributeRevenue
(nba.com, 60)
(esp.com, 10)
(nhl.com, 35)
(nba.com, 5)
```
Data Processing Operators

The **COGROUP** command

- **Output of a COGROUP contains one tuple for each group**
  - First field \((\text{group})\) is the group identifier (the value of the \text{queryString})
  - Each of the next fields is a bag, one for each group being co-grouped

- **Grouping can be performed according to UDFs**

- **Next: why COGROUP when you can use JOINS?**
Data Processing Operators

**COGROUP vs JOIN**

- **JOIN vs. COGROUP**
  - Their are equivalent: $\text{JOIN} = \text{COGROUP}$ followed by a cross product of the tuples in the nested bags.

- **Example 3:** Suppose we try to attribute search revenue to search-results urls → compute monetary worth of each url
  
  grouped_data = COGROUP results BY queryString, revenue BY queryString;
  url_revenues = FOREACH grouped_data GENERATE FLATTEN(distrubteRevenue(results, revenue));

  - Where $\text{distrubteRevenue}$ is a UDF that accepts search results and revenue information for each query string, and outputs a bag of urls and revenue attributed to them.
Data Processing Operators

COGROUP vs JOIN

- More details on the UDF distribute Revenue
  - Attributes revenue from the top slot entirely to the first search result
  - The revenue from the side slot may be equally split among all results

- Let’s see how to do the same with a JOIN
  - JOIN the tables results and revenues by queryString
  - GROUP BY queryString
  - Apply a custom aggregation function

- What happens behind the scenes
  - During the join, the system computes the cross product of the search and revenue information
  - Then the custom aggregation needs to undo this cross product, because the UDF specifically requires so
Data Processing Operators

**COGROUP in details**

- **The COGROUP statement conforms to an algebraic language**
  - The operator carries out only the operation of grouping together tuples into nested bags
  - The user can decide whether to apply a (custom) aggregation on those tuples or to cross-product them and obtain a join

- **It is thanks to the nested data model that COGROUP is an independent operation**
  - Implementation details are tricky
  - Groups can be very large (and are redundant)
Data Processing Operators

A special case of COGROUP: the GROUP operator

- Sometimes, we want to operate on a single dataset
  - This is when you use the GROUP operator

Let’s continue from Example 3:

- Assume we want to find the total revenue for each query string. This writes as:
  
  ```pig
  grouped_revenue = GROUP revenue BY queryString;
  query_revenue = FOREACH grouped_revenue GENERATE queryString, SUM(revenue.amount) AS totalRevenue;
  ```

- Note that `revenue.amount` refers to a projection of the nested bag in the tuples of `grouped_revenue`
Data Processing Operators

JOIN in Pig Latin

- In many cases, the typical operation on two or more datasets amounts to an equi-join
  - IMPORTANT NOTE: large datasets that are suitable to be analyzed with Pig (and MapReduce) are generally not normalized
  - JOINs are used more infrequently in Pig Latin than they are in SQL

- The syntax of a JOIN

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

- This is a classic inner join (actually an equi join), where each match between the two relations corresponds to a row in the `join_result`
Data Processing Operators

JOIN in Pig Latin

- JOINs lend themselves to optimization opportunities
  - We will work on this in the laboratory

- Assume we join two datasets, one of which is considerably smaller than the other
  - For instance, suppose a dataset fits in memory

- Fragment replicate join
  - Syntax: append the clause `USING "replicated"` to a JOIN statement
  - Uses a distributed cache available in Hadoop
  - All mappers will have a copy of the small input
  - This is a Map-side join
Data Processing Operators

MapReduce in Pig Latin

- It is trivial to express MapReduce programs in Pig Latin
  - This is achieved using `GROUP` and `FOREACH` statements
  - A map function operates on one input tuple at a time and outputs a bag of key-value pairs
  - The reduce function operates on all values for a key at a time to produce the final result

Example

```pig
map_result = FOREACH input GENERATE FLATTEN(map(*));
key_groups = GROUP map_results BY $0;
output = FOREACH key_groups GENERATE reduce(*);
```

- where `map()` and `reduce()` are UDF
Introduction

- Pig Latin Programs are compiled into MapReduce jobs, and executed using Hadoop
- How to build a logical plan for a Pig Latin program
- How to compile the logical plan into a physical plan of MapReduce jobs
- How to avoid resource exhaustion
Building a Logical Plan

- As clients issue Pig Latin commands (interactive or batch mode)
  - The Pig interpreter parses the commands
  - Then it verifies validity of input files and bags (variables)
    - E.g.: if the command is `c = COGROUP a BY . . . , b BY . . . ;`, it verifies if `a` and `b` have already been defined

- Pig builds a **logical plan** for every bag
  - When a new bag is defined by a command, the new logical plan is a combination of the plans for the input and that of the current command
Building a Logical Plan

- No processing is carried out when constructing the logical plans
  - Processing is triggered only by \texttt{STORE} or \texttt{DUMP}
  - At that point, the logical plan is compiled to a physical plan

- \textbf{Lazy execution model}
  - Allows in-memory pipelining
  - File reordering
  - Various optimizations from the traditional RDBMS world

- \textbf{Pig is (potentially) platform independent}
  - Parsing and logical plan construction are platform oblivious
  - Only the compiler is specific to Hadoop
Building the Physical Plan

Compilation of a logical plan into a physical plan is “simple”

- MapReduce primitives allow a parallel `GROUP BY`
  - Map assigns keys for grouping
  - Reduce process a group at a time (actually in parallel)

How the compiler works

- Converts each `(CO)GROUP` command in the logical plan into distinct MapReduce jobs
- *Map function* for `(CO)GROUP` command $C$ initially assigns keys to tuples based on the `BY` clause(s) of $C$
- *Reduce function* is initially a no-op
Building the Physical Plan

- **MapReduce boundary is the COGROUP command**
  - The sequence of FILTER and FOREACH from the LOAD to the first COGROUP $C_1$ are pushed in the Map function
  - The commands in later COGROUP commands $C_i$ and $C_{i+1}$ can be pushed into:
    - the Reduce function of $C_i$
    - the Map function of $C_{i+1}$
Building the Physical Plan

- Pig optimization for the physical plan
  - Among the two options outlined above, the first is preferred
  - Indeed, grouping is often followed by aggregation
    → reduces the amount of data to be materialized between jobs

- **COGROUP** command with more than one input dataset
  - Map function appends an extra field to each tuple to identify the dataset
  - Reduce function decodes this information and inserts tuple in the appropriate nested bags for each group
Building the Physical Plan

- **How parallelism is achieved**
  - For **LOAD** this is inherited by operating over HDFS
  - For **FILTER** and **FOREACH**, this is automatic thanks to MapReduce framework
  - For **(CO)GROUP** uses the **SHUFFLE** phase

- **A note on the ORDER command**
  - Translated in two MapReduce jobs
  - First job: **Samples the input** to determine quantiles of the sort key
  - Second job: Range partitions the input according to quantiles, followed by sorting in the reduce phase

- **Known overheads due to MapReduce inflexibility**
  - Data materialization between jobs
  - Multiple inputs are not supported well
Efficiency measures

- **(CO) GROUP** command place tuples of the same group in nested bags
  - Bag materialization (I/O) can be avoided
  - This is important also due to memory constraints
  - Distributive or algebraic aggregation facilitate this task

- **What is an algebraic function?**
  - Function that can be structured as a tree of sub-functions
  - Each leaf sub-function operates over a subset of the input data
  - If nodes in the tree achieve data reduction, then the system can reduce materialization
  - Examples: COUNT, SUM, MIN, MAX, AVERAGE, ...
Efficiency measures

- Pig compiler uses the **combiner** function of Hadoop
  - A special API for algebraic UDF is available

- There are cases in which **(CO) GROUP** is inefficient
  - This happens with non-algebraic functions
  - Nested bags can be spilled to disk
  - Pig provides a **disk-resident bag implementation**
    - Features external sort algorithms
    - Features duplicates elimination
Debugging
Introduction

- The process of creating Pig Latin programs is generally iterative
  - The user makes an initial stab
  - The stab is executed
  - The user inspects the output check correctness
  - If not, revise the program and repeat the process

- This iterative process can be inefficient
  - The sheer size of data volumes hinders this kind of experimentation
  - Need to create a side dataset that is a small sample of the original one

- Sampling can be problematic
  - Example: consider an equi-join on relations $A(x, y)$ and $B(x, z)$ on attribute $x$
  - If there are many distinct values of $x$, it is highly probable that a small sample of $A$ and $B$ will not contain matching $x$ values
  - Empty result
Welcome Pig Pen

- **Pig comes with a debugging environment, Pig Pen**
  - It creates a side dataset automatically
  - This is done in a manner that avoids sampling problems
  - The side dataset must be tailored to the user program

- **Sandbox Dataset**
  - Takes as input a Pig Latin program $P$
    - This is a sequence of $n$ commands
    - Each command consumes one or more input bags and produces one output bag
  - The output is a set of *example bags* $\{B_1, B_2, \ldots, B_n\}$
    - Each output example bag corresponds to the output of each command in $P$
  - The output set of example bags need to be **consistent**
    - The output of each operator needs to be that obtained with the input example bag
Properties of the Sandbox Dataset

There are three primary objectives in selecting a sandbox dataset

- **Realism**: the sandbox should be a subset of the actual dataset. If this is not possible, individual values should be the ones in the actual dataset
- **Conciseness**: the example bags should be as small as possible
- **Completeness**: the example bags should collectively illustrate the key semantics of each command

Overview of the procedure to generate the sandbox

- Take small random samples of the original data
- Synthesize additional data tuples to improve completeness
- When possible use real data values on synthetic tuples
- Apply a pruning pass to eliminate redundant example tuples and improve conciseness
Optimizations
Introduction

- Pig implements several optimizations
  - Most of them are derived from traditional works in RDBMS
  - Logical vs. Physical optimizations
Single-program Optimizations

- **Logical optimizations: query plan**
  - Early projection
  - Early filtering
  - Operator rewrites

- **Physical optimization: execution plan**
  - Mapping of logical operations to MapReduce
  - Splitting logical operations in multiple physical ones
  - Join execution strategies
Cross-program Optimizations

- Popular tables
  - Web crawls
  - Search query log

- Popular transformations
  - Eliminate spam
  - Group pages by host
  - Join web crawl with search log

- GOAL: minimize redundant work
Cross-program Optimizations

- **Concurrent work sharing**
  - Execute related Pig Latin programs together to perform common work only once
  - This is difficult to achieve: scheduling, “sharability”

- **Non-concurrent work sharing**
  - Re-use I/O or CPU work done by one program, later in time
  - This is difficult to achieve: caching, replication
Work-Sharing Techniques
Work-Sharing Techniques
Work-Sharing Techniques
References I

[1] Pig wiki.

Pig latin: A not-so-foreign language for data processing.

Hadoop, The Definitive Guide.
O’Reilly, Yahoo, 2010.