Pig

Peter, Gang and Ronie

Introduction

• What is Pig?
  – An open-source high-level dataflow system
  – Provides a simple language for queries and data manipulation, Pig Latin, that is compiled into map-reduce jobs that are run on Hadoop
  – Pig Latin combines the high-level data manipulation constructs of SQL with the procedural programming of map-reduce

• Why is it important?
  – Companies and organizations like Yahoo, Google and Microsoft are collecting enormous data sets in the form of click streams, search logs, and web crawls
  – Some form of ad-hoc processing and analysis of all of this information is required

Existing Solutions

• Parallel database products (ex: Teradata)
  – Expensive at web scale
  – Data analysis programmers find the declarative SQL queries to be unnatural and restrictive

• Raw map-reduce
  – Complex n-stage dataflows are not supported; joins and related tasks require workarounds or custom implementations
  – Resulting code is difficult to reuse and maintain; shifts focus and attention away from data analysis

Language Features

• Several options for user-interaction
  – Interactive mode (console)
  – Batch mode (prepared script files containing Pig Latin commands)
  – Embedded mode (execute Pig Latin commands within a Java program)

• Built primarily for scan-centric workloads and read-only data analysis
  – Easily operates on both structured and schema-less, unstructured data
  – Transactional consistency and index-based lookups not required
  – Data curation and schema management can be overkill

• Flexible, fully nested data model

• Extensive UDF support
  – Currently must be written in Java
  – Can be written for filtering, grouping, per-tuple processing, loading and storing

Pig Latin vs. SQL

• Pig Latin is procedural (dataflow programming model)
  – Step-by-step query style is much cleaner and easier to write and follow than trying to wrap everything into a single block of SQL

Pig Latin vs. SQL (continued)

• Lazy evaluation (data not processed prior to STORE command)
• Data can be stored at any point during the pipeline
• An execution plan can be explicitly defined
  – No need to rely on the system to choose the desired plan via optimizer hints
• Pipeline splits are supported
  – SQL requires the join to run twice or materialized as an intermediate result


Data Model

- Supports four basic types
  - Atom: a simple atomic value (int, long, double, string)
    - ex: 'Peter'
  - Tuple: a sequence of fields that can be any of the data types
    - ex: ('Peter', 14)
  - Bag: a collection of tuples of potentially varying structures, can contain duplicates
    - ex: [('Peter'), ('Bob', (14, 21))]
  - Map: an associative array, the key must be a chararray but the value can be any type

Data Model (continued)

- By default Pig treats undeclared fields as bytearrays (collection of uninterpreted bytes)
- Can infer a field’s type based on:
  - Use of operators that expect a certain type of field
  - UDFs with a known or explicitly set return type
  - Schema information provided by a LOAD function or explicitly declared using an AS clause
- Type conversion is lazy

Pig Latin

- FOREACH-GENERATE (per-tuple processing)
  - It iterates over every input tuple in the bag, producing one output each, allowing efficient parallel implementation
  - Expressions within the GENERATE clause can take the form of the any of these expressions

Pig Latin (continued)

- LOAD / STORE
  - Default implementation expects/outputs to tab-delimited plain text file

Pig Latin (continued)

- (CO)GROUP vs. JOIN
  - (CO)GROUP takes advantage of nested data structure (combination of GROUP BY and JOIN)
  - User can choose to go through with cross-product for a join or perform aggregation on the nested bags

Compilation

- Pig Latin
  - LOAD / STORE
  - Other commands
  - Nested operations

- Compilation pipeline:
  - Pig Latin Programs
  - Logical Plan
  - Query Panel
  - Physical Plan
  - MapReduce Plan
  - Schema Checking
  - Logical Optimizer
  - Logical to Physical Translator
  - MapReduce Launcher
### Parsing
- Type checking with schema
- References verifying
- Logic plan generating
  - One-to-one fashion
  - Independent of execution platform
  - Limited optimization

### Logic Plan
```
A=LOAD 'file1' AS (x, y, z);
B=LOAD 'file2' AS (t, u, v);
C=FILTER A by y > 0;
D=JOIN C BY x, B BY u;
E=GROUP D BY z;
F=FOREACH E GENERATE group, COUNT(D);
STORE F INTO 'output';
```

### Physical Plan
- 1:1 correspondence with most logical operators
- Except for:
  - DISTINCT
  - (CO)GROUP
  - JOIN
  - ORDER

### Physical Optimization
- Always use combiner for pre-aggregation
- Insert SPLIT to re-use intermediate result
- Early projection

### MapReduce Plan
- Determine MapReduce boundaries
  - GLOBAL REARRANGE
- Some operations are done by MapReduce framework
- Coalesce other operators into Map & Reduce stages
- Generate job jar file
Branching Plans

- Read the dataset once and process it in multiple ways
  - Good
    - Eliminate the cost to read it multiple times
  - Bad
    - Reduce the amount of memory for each stream

Physical plan execution

- Executing the portion of a physical plan within a Map or Reduce stage
  - Push vs. Pull (iterator) Model
    - Push
      - complicated API;
      - multiple threads needed

Physical plan execution (contd.)

- Pull
  - simple API; single thread

- Two drawbacks
  - **bag materialization** – “push” can control combiner within the operator
  - **branch point** – operators at branch point may face buffering issue

Nested programs example

```sql
 clicks = LOAD 'clicks'
 AS (userid, pageid, linkid, viewedat);
 byuser = GROUP clicks BY userid;
 result = FOREACH byuser {
    uniqPages = DISTINCT clicks.pageid;
    uniqLinks = DISTINCT clicks.linkid;
    GENERATE group, COUNT(uniqPages),
    COUNT(uniqLinks);
};
```
Memory management

- Java memory management
  NO low-level control over allocation and deallocation
- Intermediate results exceed available memory
- Memory manager: a list of Pig bags in a JVM
- Spill old bags and perform Garbage collection: when a new bag is added to the list; when the memory runs too low

New strategy (from Pig Manual)

- For Pig 0.6.0, the strategy for how Pig decides when to spill bags to disk is changed.
- In the past, Pig tried to figure out when an application was getting close to memory limit and then spill at that time.
- However, because Java does not include an accurate way to determine when to spill, Pig often ran out of memory.

New strategy (from Pig Manual)

- In the current version, allocate a fix amount of memory (10% of available memory by default) to store bags and spill to disk as soon as the memory limit is reached.
- This is very similar to how Hadoop decides when to spill data accumulated by the combiner. (Also with mapper output and reducer input!)

Streaming

- Allows data to be pushed through external executables
- Example:
  ```scala
  A = LOAD 'data';
  B = STREAM A THROUGH 'stream.pl -n 5';
  ```
- Due to asynchronous behavior of external executables, each STREAM operator will create two threads for feeding and consuming data from external executables.

Benchmark and Performance

- Pig Mix representative of jobs in Yahoo!
- Benchmark results

Pig problem

- Fragment-replicate; skewed; merge join
- User has to know when to use which join
- Because... Pig is domestic animal, does whatever you tell it to do.

- Alan Gates
Future work

• support more nested operators (current only FILTER, ORDER, DISTINCT)
• Better optimization order of execution, non-linear
• Metadata facility (better knowledge of data)
• Parallel job execution (inter-job, currently intra-job)
• Column-storage

Discussion

• The Good, the Bad, and the Pig
• Compare to LINQ/DryadLINQ? SCOPE?
• Use outside Yahoo!?
• Hybrid system! Local database for physical execution within a Map or Reduce stage.

Zebra (from Pig project page)

• Zebra is an access path library for reading and writing data in a column-oriented fashion. Zebra functions as an abstraction layer between your client application and data on the Hadoop Distributed File System (HDFS).