Cloud Programming

Hyun Duk Kim and Chia–Chi Lin
January 16, 2010
Agenda

- Introduction
- Pig Latin
- DryadLINQ
- Comparison between Pig Latin and DryadLINQ
- Wave computing
- Related work
- Discussion
Huge Amount of data analysis
Especially web service companies
→ Need of parallel/distributed system

Parallel DB
→ Expensive at web scale, Limited SQL
Background

- Map/Reduce
  - More procedural programming model.
  - Popular cloud computing environment
- Emergence of parallel computing tools
  - Ease of programming
    - User can just submit tasks in the specific form, then tools execute them in distributed manner.
    - Ex. Hadoop, Dryad, ...
Limitations of Hadoop/Dryad

### Power of programming
- Too low-level, Rigid
- Hard to maintain, Hard to reuse code
- Re-implement common queries
- Poor debugging environment

### Optimization across jobs
- Redundant computing
- Load imbalance
- Success rate vs. Window size

---

Pig Latin, DryadLINQ

Wave Computing
Pig Latin:
A Not-So-Foreign Language for Data Processing
SIGMOD’08

Christopher Olston, Benjamin Reed, Utkarsh Srivastava,
Ravi Kumar, and Andrew Tomkins
Pig Latin

Declarative SQL

Pig Latin

Low-level, Procedural Map/reduce
Example

- Find the users who tend to visit high-pagerank pages

<table>
<thead>
<tr>
<th>SQL</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT user FROM visits, user WHERE avgpr &gt; 0.6 IN ( SELECT user, AVG(pagerank) )</td>
</tr>
<tr>
<td>... one nested SQL query</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Pig Latin</th>
</tr>
</thead>
<tbody>
<tr>
<td>V_p = JOIN visits BY url, pages BY url;</td>
</tr>
<tr>
<td>Users = GROUP v_p BY user;</td>
</tr>
<tr>
<td>Useravg = FOREACH users GENERATE group, AVG(v_p.pagerank) AS avgpr;</td>
</tr>
<tr>
<td>Answer = FILTER useravg BY avgpr &gt; ‘0.5’;</td>
</tr>
<tr>
<td>... sequence of commands</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Java Map/Reduce</th>
</tr>
</thead>
<tbody>
<tr>
<td>public static class Map extends MapReduceBase implements Mapper&lt;LongWritable, Text, Text, IntWritable&gt; {</td>
</tr>
<tr>
<td>... more than 100 lines</td>
</tr>
</tbody>
</table>
Pig

- Execution engine on atop Hadoop
- Open source project
- Mainly developing/using in Yahoo
Find the users who tend to visit high-pagerank pages

<table>
<thead>
<tr>
<th>Visits</th>
<th>URL Info</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>User</strong></td>
<td><strong>URL</strong></td>
</tr>
<tr>
<td>Amy</td>
<td>cnn.com</td>
</tr>
<tr>
<td>Amy</td>
<td>bbc.com</td>
</tr>
<tr>
<td>Amy</td>
<td>flickr.com</td>
</tr>
<tr>
<td>Fred</td>
<td>cnn.com</td>
</tr>
</tbody>
</table>
visits = LOAD 'visits.txt' AS (user, url, time);
pages = LOAD 'pages.txt' AS (url, pagerank);

v_p = JOIN visits BY url, pages BY url;
users = GROUP v_p BY user;
useravg = FOREACH users
    GENERATE group, AVG(v_p.pagerank) AS avgpr;
answer = FILTER useravg BY avgpr > '0.5';
In Pig Latin

visits = LOAD 'visits.txt' AS (user, url, time);
pages = LOAD 'pages.txt' AS (url, pagerank);

visits: (Amy, cnn.com, 8am)
(Amy, frogs.com, 9am)
(Fred, snails.com, 11am)
pages: (cnn.com, 0.8)
(frogs.com, 0.8)
(snails.com, 0.3)
visits = LOAD 'visits.txt' AS (user, url, time);

pages = LOAD 'pages.txt' AS (url, pagerank);

v_p = JOIN visits BY url, pages BY url;

visits: (Amy, cnn.com, 8am)
      (Amy, frogs.com, 9am)
      (Fred, snails.com, 11am)

pages: (cnn.com, 0.8)
      (frogs.com, 0.8)
      (snails.com, 0.3)

v_p: (Amy, cnn.com, 8am, cnn.com, 0.8)
     (Amy, frogs.com, 9am, frogs.com, 0.8)
     (Fred, snails.com, 11am, snails.com, 0.3)
In Pig Latin

visits = LOAD ‘visits.txt’ AS (user, url, time);
pages = LOAD ‘pages.txt’ AS (url, pagerank);

v_p = JOIN visits BY url, pages BY url;

users = GROUP v_p BY user;

v_p: (Amy, cnn.com, 8am, cnn.com, 0.8)
    (Amy, frogs.com, 9am, frogs.com, 0.8)
    (Fred, snails.com, 11am, snails.com, 0.3)

users: (Amy, { (Amy, cnn.com, 8am, cnn.com, 0.8)
    (Amy, frogs.com, 9am, frogs.com, 0.8) })
    (Fred, { (Fred, snails.com, 11am, snails.com, 0.3) })
In Pig Latin

visits = LOAD 'visits.txt' AS (user, url, time);

pages = LOAD 'pages.txt' AS (url, pagerank);

v_p = JOIN visits BY url, pages BY url;

users = GROUP v_p BY user;

v_p: (Amy, cnn.com, 8am, cnn.com, 0.8)
    (Amy, frogs.com, 9am, frogs.com, 0.8)
    (Fred, snails.com, 11am, snails.com, 0.3)

users: (Amy, { (Amy, cnn.com, 8am, cnn.com, 0.8)
               (Amy, frogs.com, 9am, frogs.com, 0.8) } )
     (Fred, { (Fred, snails.com, 11am, snails.com, 0.3) } )
visits  = LOAD ‘visits.txt’ AS (user, url, time);
pages  = LOAD ‘pages.txt’ AS (url, pagerank);

v_p    = JOIN visits BY url, pages BY url;
users  = GROUP v_p BY user;
useravg = FOREACH users
          GENERATE group, AVG(v_p.pagerank) AS avgpr;

users:  (Amy, { (Amy, cnn.com, 8am, cnn.com, 0.8)
               (Amy, frogs.com, 9am, frogs.com, 0.8) } )
        (Fred, { (Fred, snails.com, 11am, snails.com, 0.3) } )

useravg: (Amy, 0.8)
          (Fred, 0.3)
In Pig Latin

visits = LOAD 'visits.txt' AS (user, url, time);
pages = LOAD 'pages.txt' AS (url, pagerank);

v_p = JOIN visits BY url, pages BY url;
users = GROUP v_p BY user;
useravg = FOREACH users
   GENERATE group, AVG(v_p.pagerank) AS avgpr;

users: (Amy, { (Amy, cnn.com, 8am, cnn.com, 0.8)
         (Amy, frogs.com, 9am, frogs.com, 0.8) })
      (Fred, { (Fred, snails.com, 11am, snails.com, 0.3) })

useravg: (Amy, 0.8)
         (Fred, 0.3)

Can use any UDFs
In Pig Latin

visits = LOAD 'visits.txt' AS (user, url, time);

pages = LOAD 'pages.txt' AS (url, pagerank);

v_p = JOIN visits BY url, pages BY url;

users = GROUP v_p BY user;

useravg = FOREACH users
          GENERATE group, AVG(v_p.pagerank) AS avgpr;

answer = FILTER useravg BY avgpr > '0.5';

useravg: (Amy, 0.8)
          (Fred, 0.3)

answer: (Amy, 0.8)
Data Flow

Load visits

Load pages

Join by url

Group by user

Foreach category
generate avg

...
Compilation into Map–Reduce

Every group or join operation forms a map-reduce boundary
Other operations pipelined into map and reduce phases
Data Model

- Atom
  - 'alice'
- Tuple
  - ('alice', 'lakers')
- Bag
  - ('alice', 'lakers')
- Map
  - [ 'age' -> 20 ]
- Nested Data Model
  - (Amy, { (Amy, cnn.com, 8am, cnn.com, 0.8)
    (Amy, frogs.com, 9am, frogs.com, 0.8) })
Pig Latin Command

- Specifying Input Data: LOAD
- Per-tuple Processing: FOREACH
- Discarding Unwanted Data: FILTER
- Getting Related Data Together: COGROUP
- Other Commands
  - UNION, CROSS, ORDER, DISTINCT
- Asking for Output: STORE

Very Similar to SQL commands
Debugging Environment

- Pig Pen
Future Work

- “Safe” optimizer
  - Performs only high-confidence rewrites
- User interface
  - Boxes and arrows UI
  - Promote collaboration, sharing code fragments and UDFs
- External functions
  - Provide UDF packages
- Unified environment
  - Use loops, conditionals of host language
Why Pig?

- Implementation productivity
  - 10 lines of Pig Latin = 200 lines of Java M/R
  - 15 minutes to write in Pig Latin = 4 hours Java M/R
- Provide common operations like join, group, filter, sort
- Open to non-Java programmers
Why not Pig?

- Slower speed
  - Code converting overload
  - Not task-specific optimization
- Not flexible for special operation
  - Implementing UDF takes time
- Not SQL
  - Weaker functions
  - Need additional effort to convert existing SQL query system to the distributed system with Pig
Discussion

- Should Pig Latin have all the SQL features?
- Is Pig really easier than Hadoop MapReduce Programming for whom does not know SQL?
DryadLINQ:
A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language

OSDI’08 (Awarded Best Paper)

Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Ulfar Erlingsson, Pradeep Kumar Gunda, and Jon Currey
Is Pig + Hadoop enough?

- Obviously, Microsoft does not think so
- But, why?
  - Hadoop employs the MapReduce programming model
  - “…… aims for simplicity at the expense of generality and performance …..” [1]

Dryad

- Directed–acyclic graph (DAG)
  - Flexible
  - Permits efficient execution plans for many algorithms
- However, it is oftentimes infeasible to specify the DAG by hand

- Directed--acyclic graph (DAG)
- Flexible
- Permits efficient execution plans for many algorithms
- However, it is oftentimes infeasible to specify the DAG by hand

New jobs

Job \(_1\): \(v_{11}, v_{12}, \ldots\)
Job \(_2\): \(v_{21}, v_{22}, \ldots\)
Job \(_3\): \(\ldots\)

cluster

scheduler
What is missing?

DryadLINQ provides automatic query plan generation
Dryad provides automatic distributed execution
Dryad + LINQ = DryadLINQ

LINQ expression

```csharp
var docs = DryadLinq.GetTable<Doc>("file://docs.txt");
var words = docs.SelectMany(doc => doc.words);
var groups = words.GroupBy(word => word);
var counts = groups.Select(g => new WordCount(g.Key, g.Count()));

counts.ToDryadTable("counts.txt");
```

Dryad execution

```
var docs = DryadLinq.GetTable<Doc>("file://docs.txt");
var words = docs.SelectMany(doc => doc.words);
var groups = words.GroupBy(word => word);
var counts = groups.Select(g => new WordCount(g.Key, g.Count()));

counts.ToDryadTable("counts.txt");
```
DryadLINQ System Architecture

Client machine

.NET program

ToTable

Query Expr

Distributed query plan

DryadLINQ

Invoke

DryadTable

foreach

.Net Objects

Data center

Query

Vertex code

Input Tables

JM

Dryad Execution

Output Tables

foreach

Results
Static Optimizations

- Pipelining
  - Executing multiple operators in a single process
- Removing redundancy
  - Remove unnecessary partitioning steps
- Eager aggregation
  - Moving down-stream aggregations in front of partitioning operators
- I/O reduction
  - TCP-pipe and in-memory FIFO channels
  - Compresses data before performing a partitioning
Optimization Example – OrderBy

Static Optimization

OrderBy

Deterministic Sampling

Histogram

Data Distribution

Merge

Sort
Dynamic Optimizations
Evaluation – Terasort

- TeraByte Sort (Indy): 10 billion 100-Byte records with 10-Byte key
Q18 from the Sloan Digital Sky Survey database: three-way Join over two input tables containing 11.8 GBytes and 41.8 GBytes of data, respectively
Conclusion and Discussion

- DryadLINQ is an elegant programming environment combining the benefits of LINQ with the power of Dryad
- Supports multiple languages including C#, VB, and F#
- Leverages other systems that use the same constructs such as PLINQ, LINQ-to-SQL, and LINQ-to-Object
- Clean separation of Dryad and DryadLINQ
Conclusion and Discussion

- Directed-acyclic graph provides generality but also brings complexity
- Dynamic optimizations on concurrent jobs
- Debugging, analyzing, and monitoring
Comparison between Pig Latin and DryadLINQ
## Comparison

<table>
<thead>
<tr>
<th></th>
<th>Pig Latin</th>
<th>DryadLINQ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base System</td>
<td>Hadoop (HDFS)</td>
<td>Dryad</td>
</tr>
<tr>
<td>Main Contributor</td>
<td>Yahoo, Open Source</td>
<td>Microsoft (Internal)</td>
</tr>
<tr>
<td>Programming</td>
<td>Imperative</td>
<td>Imperative &amp; Declarative</td>
</tr>
<tr>
<td>Model Structure</td>
<td>Sequence of Map/Reduce</td>
<td>Directed Acyclic graph</td>
</tr>
<tr>
<td>Development environment</td>
<td>Mainly linux, Some eclipse plug-in</td>
<td>Windows, Visual Studio</td>
</tr>
<tr>
<td>Main Language</td>
<td>Java</td>
<td>C#</td>
</tr>
<tr>
<td>Compared to SQL</td>
<td>Similar</td>
<td>Very similar</td>
</tr>
</tbody>
</table>

- Both enable users to use parallel computing tool more conveniently
- But, slower speed than original system
  → Need for consideration in speed improvement
Wave Computing in the Cloud
HotOS’09

Bingsheng He, Mao Yang, Zhenyu Guo, Rishan Chen, Wei Lin, Bing Su, Hongyi Wang, and Lidong Zhou
What do we have right now?

- **Execution plans**
  - Dryad and Hadoop
- **High-level languages**
  - DryadLINQ and Pig Latin
- **Optimizations for performance and resource utilization in both dimensions for a single job**
- **However, regarding optimization, there are still something left ……**
Can you identify the inefficiency?
More Examples

Every day:

- Extract
  - Filter: “Chinese”
  - Compute Top Ten
  - ...

Every week:

- Extract
  - Filter: “Chinese”
  - Compute Top Ten
  - ...

Common computation on per-day log (Ideally)
What do the statistics say?

- **Normalized Total I/O**
  - Current Production System: 46% of Ideal System

- **Pie Charts**
  - Redundant I/O on input data: 33%
  - Distinct I/O: 67%
  - Common computation steps: 30%
  - Other computation steps: 70%

*(Results from simulation)*
The Wave Model

- **Streams**
  - Append-only files and partitioned on multiple machines

- **Query series**
  - Recurrent computations on a stream, with each performed on one or more stream segments
The Wave Model

Individual query series

Jumbo queries

Time
Opportunities

- Enabling prediction
  - Input and output data
  - Computation complexity of custom functions
  - Execution environment

- Wave optimizations
  - Shared scan and computation
  - Query decomposition, planning, and scheduling

- Waves into the cloud
Logical optimization (computation sharing) reduces the total I/O by 12.3%.

Full optimization (computation + data sharing) reduces the total I/O by 42.3%.
Logical optimization reduces the total machine time by 30.5%
Full optimization reduces the total machine time by 42.0%
Conclusion and Discussion

- Wave computing introduces a new processing model that can potentially unlock the full power of data-intensive distributed computing
- Identifies computation and I/O redundancy
- Enables optimizations from other fields such as database
Conclusion and Discussion

- Feasibility of the model
  - Could we apply the model directly to community clouds?

- More opportunities
  - Caching/reusing intermediate results
Related Work
Related Work

- **Map-reduce-merge**
  - Map-reduce does not support processing multiple related heterogeneous datasets (Joins)
  - Add Merge phase after reduce

- **Hadoop Streaming**
  - Want to use existing executables or other languages
  - Allows to create map/reduce using any executable or script

- **Hbase**
  - Slow in random, realtime read/write access to Big Data
  - Distributed column-oriented store model like Google’s Bigtable for hadoop.
Related Work

- Hive
  - A data warehouse infrastructure that provides data summarization and ad hoc querying

- Zookeeper
  - A high-performance coordinate service for distributed applications
Thank You
References
References

- Pig Latin: A Not–So–Foreign Language for Data Processing, C. Olston et al., SIGMOD 2008 (Yahoo!)
- Cloudera Pig Tutorial
  http://www.cloudera.com/videos/introduction_to_pig
- Dryad: Distributed Data–Parallel Programs from Sequential Building Blocks, M. Isard et al., EuroSys 2007
- DryadLINQ: A System for General–Purpose Distributed Data–Parallel Computing Using a High–Level Language, Y. Yu et al., OSDI 2008
References

- Wave Computing in the Cloud, B. He et al., HotOS 2009
- Comet: Batched Stream Processing in Data Intensive Distributed Computing, B. He et al., Technical Report 2009
More Discussion
Features

- Dataflow Language
- Nested Data Model
- Nested Operation
- Support UDF (User Defined Function)
- Parallelism Required
- Debugging Environment
Motivation

- Limitation of map/reduce
  - Difficulty in programming
    - Too low-level, Rigid
    - Hard to maintain, Hard to reuse code
    - Common queries that are difficult to program
    - Poor debugging environment
    - → Pig–Latin, DryadLINQ

  - Performance issues
    - Redundancy
    - Load Imbalance
    - Success Rate vs. Window size
    - → Wave computing
Find the average pagerank of high-pagerank urls for each sufficiently large category.

**SQL**
```
SELECT category, AVG(pagerank)
FROM urls WHERE pagerank > 0.2
GROUP BY category HAVING COUNT(*) > 106
```

**Pig Latin**
```
good_urls = FILTER urls BY pagerank > 0.2;
groups = GROUP good_urls BY category;
big_groups = FILTER groups BY COUNT(good_urls) > 106;
output = FOREACH big_groups
    GENERATE category, AVG(good_urls.pagerank);
```

**Java Map/Reduce**
```
public static class Map extends MapReduceBase implements Mapper<LongWritable, Text, Text, IntWritable> {
    ... more than 100 lines
```