Big Data Technology
Pig: Query Language atop Map-Reduce

Eshcar Hillel
Yahoo!

Ronny Lempel
Outbrain

*Based on slides by Edward Bortnikov & Ronny Lempel
Roadmap

- Previous class – MR Implementation

- This class – Query Languages atop MR
  - Beyond MR
  - SQL primitives and implementation
  - Pig – a procedural query language
Pig for Hadoop

http://hortonworks.com/hdp/

CS236620 Big Data Technology
Map-Reduce Critique

- Too low-level (only engineers can program)
- Hand-coding for many common operations
- Data flows extremely rigid (linear)

- This is exactly what SQL databases have been designed for!
- Can we reuse some of the good stuff?
The data is **structured**

Reflects Entities and Relationships
- ERD = Entity-Relationship Diagram

ERD captured by **schema**
- Data set = set of tables
- Table = set of tuples (rows)
- Row = set of items (columns, attributes)
  - Typically, uniquely addressable by primary key

**Relational Algebra**
- Theoretical foundation for relational databases
SQL (Structured Query Language)

- Declarative programming language
  - Focus on **what** you want, not **how** to retrieve
  - Flexible, Great for non-engineers
- Designed for relational databases
  - Select (read)
  - Insert/Update/Delete (write)
- Costs
  - Administration (schema management)
  - Query optimization (compiler or manual)
SQL Primitives

- **Project**
  
  ```sql
  SELECT url
  FROM pages
  ```

- **Filter (Select)**
  
  ```sql
  SELECT url
  FROM pages
  WHERE pagerank > 0.95
  ```
SQL Primitives

Join (⋈)

```
SELECT visits.user, pages.category
FROM visits, pages
WHERE visits.url = pages.url
```

```
SELECT p1.header, p2.header
FROM pages p1, pages p2, links l1, links l2
WHERE p1.url = l1.from AND p2.url = l2.to AND l1.id = l2.id
```
SQL Primitives

- **Aggregation**

  ```sql
  SELECT url, COUNT(url)
  FROM visits
  GROUP BY url
  HAVING COUNT(url) > 1000
  ```

- COUNT, SUM, AVG, STDDEV, MIN, MAX
SQL Primitives

- **Sort**

SELECT cookie, query
FROM querylog
WHERE
date < 1/1/2013
AND date > 1/12/2012
ORDER BY cookie, date
SQL Execution

- Query plan = compiler-generated program
- Operators + flow of control
- Operators receive and return tuple sets
SELECT url, COUNT(url) 
FROM visits 
GROUP BY url 
HAVING COUNT(url) > 1000
SELECT p1.header, p2.header
FROM   pages p1, pages p2, links l1, links l2
WHERE  
  p1.url = l1.from AND p2.url = l2.to AND  
  l1.id = l2.id
Dataflow Architectures

- Query plan = DAG
- Nodes = relational operators
- Links = queues

- Performance boosted through ...
  - Data parallelism
  - Compute parallelism
  - Pipelining
Query Optimization

- Multiple plans can be logically equivalent
  - Relation algebra allows operator re-ordering
  - Multiple operator implementations possible
- Goal: pick a plan that minimizes query latency
  - Minimize I/O, communication and computation
  - No-brainer optimizations: push the filters deep
  - Nontrivial optimizations: order of joins
- Challenge: exponential-size search space
SQL versus Map-Reduce

- SQL
  - Good for structured data
  - Rich declarative API (tool for analysts)
  - Compiler optimization, non-transparent to users

- Map-Reduce
  - Good for structured and unstructured data
  - Simple programmatic API (tool for engineers)
  - Users can optimize their jobs manually

- Can we enjoy the best of both worlds?
Implementation atop Map-Reduce

- Project, Filter – easy (how?)
- Sort – for free (not always required)
- Aggregation – mostly easy (how?)
  - How to handle TOP-K, AVG, STDDEV?

- Join?
  - Hint: use multiple Map inputs
Join over MR (1)

Select L.x, R.y
From L, R
Where L.key = R.key

CS236620 Big Data Technology
Join over MR (2)

- Typical situation: Big $\bowtie$ Small
  - E.g., Page Accesses (B) $\bowtie$ User Details (S)

- Idea: avoid reduce altogether
- Approach: replicate S to all mappers
  - Use shared files (distributed thru the MR cache)
  - At each mapper:
    - Store in RAM, hashed by join key
    - Scan the B-partition, compute match per record
Pig Latin

- Started at Yahoo! Research (2008)
- Procedural (control) yet high-level (simplicity)
- Operates directly on flat files
- Schemas optional (defined on the fly)
- Nested (non-normalized) data and operators
- Operators are customizable (UDF)
- Amenable to optimization
**Example Analysis Task**

**Find the top 10 most visited pages in each category**

<table>
<thead>
<tr>
<th>Visits</th>
<th></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>

<table>
<thead>
<tr>
<th>Urls</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Url</strong></td>
<td><strong>Category</strong></td>
</tr>
<tr>
<td>cnn.com</td>
<td>News</td>
</tr>
<tr>
<td>bbc.com</td>
<td>News</td>
</tr>
<tr>
<td>flickr.com</td>
<td>Photos</td>
</tr>
<tr>
<td>espn.com</td>
<td>Sports</td>
</tr>
</tbody>
</table>
Load Visits

Group by url

Foreach url generate count

Join by url

Group by category

Foreach category generate top(10)

UDF
In Pig Latin ...

visits = load '/data/visits' as (user, url, time);
gVisits = group visits by url;
visitCounts = foreach gVisits generate url, count(visits);

urlInfo = load '/data/urlInfo' as (url, category, pRank);
visitCounts = join visitCounts by url, urlInfo by url;

gCategories = group visitCounts by category;
topUrls = foreach gCategories generate top(visitCounts,10);

store topUrls into '/data/topUrls';
MR Implementation

Load Visits
Group by url
Foreach url generate count

Load Urls

Group by category

Join by url

Foreach category generate top(10)
Anything Else to Do?

- High-level languages improve MR’s **usability**
- Cannot address built-in **performance** problems

- MR designed for batch processing
  - E.g., daily production jobs, focused on throughput

- Not great for low-latency applications
  - High job setup overhead
  - A no-op 1K-task job runs for dozens of seconds in the absence of contention!
Summary – SQL atop MR

- Good for data warehouses, batch queries
  - The less legacy semantics, the better scalability

- Bad for interactive ad-hoc queries
  - Batch-oriented (high launch overhead)
  - Scan-oriented, lookups are expensive
  - Intermediate results materialized on DFS
  - Dataflow underexploited (limited pipelining)
Next Class

- Key-value store use case: user profile store
Further Reading

- A comparison of approaches to large-scale data analysis
  - Map Reduce: A Major Step Backwards
- Pig Latin: A Not-So-Foreign Language for Data Processing