Search Engine Architecture

10. Big Data Processing Part Two
Today’s Agenda

- Making Hadoop more efficient
- Dataflow Languages
- Graph Processing
Hadoop is slow...
A Major Step Backwards?

- MapReduce is a step backward in database access:
  - Schemas are good
  - Separation of the schema from the application is good
  - High-level access languages are good
- MapReduce is poor implementation
  - Brute force and only brute force (no indexes, for example)
- MapReduce is not novel
- MapReduce is missing features
  - Bulk loader, indexing, updates, transactions...

Hadoop vs. Databases: Grep

\textbf{Figure 4:} Grep Task Results – 535MB/node Data Set

\textbf{Figure 5:} Grep Task Results – 1TB/cluster Data Set

\texttt{SELECT * FROM Data WHERE field LIKE ‘%XYZ%’;}

**Hadoop vs. Databases: Select**

**Figure 6: Selection Task Results**

```
SELECT pageURL, pageRank
FROM Rankings WHERE pageRank > X;
```

SELECT sourceIP, SUM(adRevenue) FROM UserVisits GROUP BY sourceIP;

Hadoop vs. Databases: Join

Figure 9: Join Task Results

“On the first day of logging the Facebook clickstream, more than 400 gigabytes of data was collected. The load, index, and aggregation processes for this data set really taxed the Oracle data warehouse. Even after significant tuning, we were unable to aggregate a day of clickstream data in less than 24 hours.”

Why?

Integer.parseInt
String.substring

Schemas are a good idea!

- Parsing fields out of flat text files is slow
- Schemas define a contract, decoupling logical from physical

Thrift

- Originally developed by Facebook, now an Apache project
- Provides a DDL with numerous language bindings
  - Compact binary encoding of typed structs
  - Fields can be marked as optional or required
  - Compiler automatically generates code for manipulating messages
- Provides RPC mechanisms for service definitions
- Alternatives include protobufs and Avro

struct Tweet {
  1: required i32 userId;
  2: required string userName;
  3: required string text;
  4: optional Location loc;
}

struct Location {
  1: required double latitude;
  2: required double longitude;
}
Dataflow Languages
Need for High-Level Languages

• Hadoop is great for large-data processing!
  • But writing Java programs for everything is verbose and slow
  • Data scientists don’t want to write Java
• Solution: develop higher-level data processing languages
  • Hive: HQL is like SQL
  • Pig: Pig Latin is a bit like Perl

Hive and Pig

- **Hive**: data warehousing application in Hadoop
  - Query language is HQL, variant of SQL
  - Tables stored on HDFS with different encodings
  - Developed by Facebook, now open source
- **Pig**: large-scale data processing system
  - Scripts are written in Pig Latin, a dataflow language
  - Programmer focuses on data transformations
  - Developed by Yahoo!, now open source
- **Common idea:**
  - Provide higher-level language to facilitate large-data processing
  - Higher-level language “compiles down” to Hadoop jobs

Hive: Example

- Hive looks similar to an SQL database
- Relational join on two tables:
  - Table of word counts from Shakespeare collection
  - Table of word counts from the bible

```sql
SELECT s.word, s.freq, k.freq FROM shakespeare s
JOIN bible k ON (s.word = k.word) WHERE s.freq >= 1 AND k.freq >= 1
ORDER BY s.freq DESC LIMIT 10;
```

<table>
<thead>
<tr>
<th></th>
<th>the</th>
<th>l</th>
<th>and</th>
<th>to</th>
<th>of</th>
<th>a</th>
<th>you</th>
<th>my</th>
<th>in</th>
<th>is</th>
</tr>
</thead>
<tbody>
<tr>
<td>count</td>
<td>25848</td>
<td>23031</td>
<td>19671</td>
<td>18038</td>
<td>16700</td>
<td>14170</td>
<td>12702</td>
<td>11297</td>
<td>10797</td>
<td>8882</td>
</tr>
<tr>
<td>freq</td>
<td>62394</td>
<td>8854</td>
<td>38985</td>
<td>13526</td>
<td>34654</td>
<td>8057</td>
<td>2720</td>
<td>4135</td>
<td>12445</td>
<td>6884</td>
</tr>
</tbody>
</table>

Hive: Behind the Scenes

SELECT s.word, s.freq, k.freq FROM shakespeare s
JOIN bible k ON (s.word = k.word) WHERE s.freq >= 1 AND k.freq >= 1
ORDER BY s.freq DESC LIMIT 10;

(Abstract Syntax Tree)

(one or more of MapReduce jobs)

Hive: Behind the Scenes

STAGE DEPENDENCIES:
Stage-1 is a root stage
Stage-2 depends on stages: Stage-1
Stage-0 is a root stage

STAGE PLANS:
Stage: Stage-1
Map Reduce
Alias -> Map Operator Tree:
s
| TableScan               |
| alias: s               |
| Filter Operator        |
| predicate:             |
| expr: (freq >= 1)     |
| type: boolean          |
Reduce Output Operator
key expressions:
expr: word
  type: string
sort order: +
Map-reduce partition columns:
  expr: word
type: string
tag: 0
value expressions:
  expr: freq
type: int
expr: word
type: string

k
| TableScan               |
| alias: k               |
| Filter Operator        |
| predicate:             |
| expr: (freq >= 1)     |
| type: boolean          |
Reduce Output Operator
key expressions:
expr: word
  type: string
sort order: +
Map-reduce partition columns:
  expr: word
type: string
tag: 1
value expressions:
  expr: freq
type: int

Reduce Operator Tree:
Join Operator
  condition map:
    Inner Join 0 to 1
    condition expressions:
      0 (VALUE._col0) [VALUE._col1]
      1 (VALUE._col0)
  outputColumnNames: _col0, _col1, _col2
Filter Operator
  predicate:
    expr: (_col0 >= 1) and (_col2 >= 1)
type: boolean
Select Operator
  expressions:
    expr: _col1
type: string
    expr: _col0
type: int
    expr: _col2
type: int
  outputColumnNames: _col0, _col1, _col2
File Output Operator
  compressed: false
GlobalTableId: 0
table:
  input format: org.apache.hadoop.mapred.TextInputFormat
  output format: org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat

Stage: Stage-2
Map Reduce
Alias -> Map Operator Tree:
hdfs://localhost:8022/tmp/hive-training/364214370/10002
Reduce Output Operator
key expressions:
  expr: _col1
type: int
sort order: -
tag: -1
value expressions:
  expr: _col0
type: string
  expr: _col1
type: int
  expr: _col2
type: int
Reduce Operator Tree:
Extract
Limit
File Output Operator
  compressed: false
GlobalTableId: 0
table:
  input format: org.apache.hadoop.mapred.TextInputFormat
  output format: org.apache.hadoop.hive.ql.io.HiveIgnoreKeyTextOutputFormat

Stage: Stage-0
Fetch Operator
  limit: 10

Hive Architecture

Hive Implementation

• Metastore holds metadata
  • Databases, tables
  • Schemas (field names, field types, etc.)
  • Permission information (roles and users)
• Hive data stored in HDFS
  • Tables in directories
  • Partitions of tables in sub-directories
  • Actual data in files

Pig!

Pig: Example

Task: Find the top 10 most visited pages in each category

<table>
<thead>
<tr>
<th>User</th>
<th>Url</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Amy</td>
<td>cnn.com</td>
<td>8:00</td>
</tr>
<tr>
<td>Amy</td>
<td>bbc.com</td>
<td>10:00</td>
</tr>
<tr>
<td>Amy</td>
<td>flickr.com</td>
<td>10:05</td>
</tr>
<tr>
<td>Fred</td>
<td>cnn.com</td>
<td>12:00</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Url</th>
<th>Category</th>
<th>PageRank</th>
</tr>
</thead>
<tbody>
<tr>
<td>cnn.com</td>
<td>News</td>
<td>0.9</td>
</tr>
<tr>
<td>bbc.com</td>
<td>News</td>
<td>0.8</td>
</tr>
<tr>
<td>flickr.com</td>
<td>Photos</td>
<td>0.7</td>
</tr>
<tr>
<td>espn.com</td>
<td>Sports</td>
<td>0.9</td>
</tr>
</tbody>
</table>

Pig Slides adapted from Olston et al. (SIGMOD 2008) via Lin et al. Big Data Infrastructure, UMD Spring 2015.
Pig Script

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';
Pig Query Plan

Load Visits

Group by url

Foreach url generate count

Load Url Info

Join on url

Group by category

Foreach category generate top10(urls)

Pig Slides adapted from Olston et al. (SIGMOD 2008) via Lin et al. Big Data Infrastructure, UMD Spring 2015.
Pig Script in Hadoop

Load Visits

Group by url

Foreach url generate count

Load Url Info

Join on url

Group by category

Foreach category generate top10(urls)

Reduce1

Reduce2

Reduce3

Map1

Map2

Map3

Pig Slides adapted from Olston et al. (SIGMOD 2008) via Lin et al. Big Data Infrastructure, UMD Spring 2015.
What’s next?

The datacenter *is* the computer!

What’s the instruction set?
Shuffle and Sort: aggregate values by keys

Answer?

Load Visits

Group by url

Foreach url generate count

Load Url Info

Join on url

Group by category

Foreach category generate top10(urls)

Map1

Reduce1

Map2

Reduce2

Map3

Reduce3

Pig Slides adapted from Olston et al. (SIGMOD 2008) via Lin et al. Big Data Infrastructure, UMD Spring 2015.
Generically, what is this?

Collections of tuples

Load Visits

Group by url

Foreach url
generate count

Load Url Info

Join on url

Group by category

Foreach category
generate top10(urls)

Transformations on those collections

Pig Slides adapted from Olston et al. (SIGMOD 2008) via Lin et al. Big Data Infrastructure, UMD Spring 2015.
Dataflows

• Comprised of:
  • Collections of records
  • Transformations on those collections

• Two important questions:
  • What are the logical operators?
  • What are the physical operators?

Spark

- One popular answer to “What’s beyond MapReduce?”
- Open-source engine for large-scale batch processing
  - Supports generalized dataflows
  - Written in Scala, with bindings in Java and Python
- Brief history:
  - Developed at UC Berkeley AMPLab in 2009
  - Open-sourced in 2010
  - Became top-level Apache project in February 2014
  - Commercial support provided by DataBricks

Resilient Distributed Datasets

- RDD: Spark “primitive” representing a collection of records
  - Immutable
  - Partitioned (the D in RDD)
- Transformations operate on an RDD to create another RDD
  - Coarse-grained manipulations only
  - RDDs keep track of lineage
- Persistence
  - RDDs can be materialized in memory or on disk
  - OOM or machine failures: What happens?
- Fault tolerance (the R in RDD):
  - RDDs can always be recomputed from stable storage (disk)

Operations on RDDs

- Transformations (lazy):
  - map
  - flatMap
  - filter
  - union/intersection
  - join
  - reduceByKey
  - groupByKey
  - ...

- Actions (actually trigger computations)
  - collect
  - saveAsTextFile/saveAsSequenceFile
  - ...

Spark Architecture

Spark Physical Operators

Narrow Dependencies:
- map, filter
- union
- join with inputs co-partitioned

Wide Dependencies:
- groupByKey
- join with inputs not co-partitioned

Spark Execution Plan

Stage 1

A: 

B: 

groupBy

Stage 2

C: 

D: 

map

E: 

union

Stage 3

F: 

G: 

join

Graph Processing
What makes graph processing hard?

- Lessons learned so far:
  - Partition
  - Replicate
  - Reduce cross-partition communication
- What makes MapReduce “work”? 

Characteristics of Graph Algorithms

- What are some common features of graph algorithms?
  - Graph traversals
  - Computations involving vertices and their neighbors
  - Passing information along graph edges
- What's the obvious idea?
  - Keep “neighborhoods” together!

Simple Partitioning Techniques

• Hash partitioning
• Range partitioning on some underlying linearization
  • Web pages: lexicographic sort of domain-reversed URLs
  • Social networks: sort by demographic characteristics

Country Structure in Facebook

Analysis of 721 million active users (May 2011)

54 countries w/ >1m active users, >50% penetration

Ugander et al. (2011) The Anatomy of the Facebook Social Graph.
What makes graph processing hard?

- It’s tough to apply our “usual tricks”:
  - Partition
  - Replicate
  - Reduce cross-partition communication
Pregel: Computational Model

- Based on Bulk Synchronous Parallel (BSP)
  - Computational units encoded in a directed graph
  - Computation proceeds in a series of supersteps
  - Message passing architecture
- Each vertex, at each superstep:
  - Receives messages directed at it from previous superstep
  - Executes a user-defined function (modifying state)
  - Emits messages to other vertices (for the next superstep)
- Termination:
  - A vertex can choose to deactivate itself
  - Is “woken up” if new messages received
  - Computation halts when all vertices are inactive

Pregel

superstep $t$

superstep $t+1$

superstep $t+2$

Pregel: Implementation

• Master-Slave architecture
  • Vertices are hash partitioned (by default) and assigned to workers
  • Everything happens in memory
• Processing cycle:
  • Master tells all workers to advance a single superstep
  • Worker delivers messages from previous superstep, executing vertex computation
  • Messages sent asynchronously (in batches)
  • Worker notifies master of number of active vertices
• Fault tolerance
  • Checkpointing
  • Heartbeat/revert

class PageRankVertex : public Vertex<double, void, double> {
public:
    virtual void Compute(MessageIterator* msgs) {
        if (superstep() >= 1) {
            double sum = 0;
            for (; !msgs->Done(); msgs->Next())
                sum += msgs->Value();
            *MutableValue() = 0.15 / NumVertices() + 0.85 * sum;
        }

        if (superstep() < 30) {
            const int64 n = GetOutEdgeIterator().size();
            SendMessageToAllNeighbors(GetValue() / n);
        } else {
            VoteToHalt();
        }
    }
};
Questions?