A Comparison of Approaches to Large-Scale Data Analysis

Sam Madden
MIT CSAIL

with

Andrew Pavlo, Erik Paulson, Alexander Rasin, Daniel Abadi, David DeWitt, and Michael Stonebraker

In SIGMOD 2009
MapReduce (MR) vs. Databases

• Our goal: understand performance and architectural differences

• Both are suitable for large-scale data processing
  – I.e. analytical processing workloads
  – Bulk loads
  – Queries over large amounts of data
  – *Not transactional*
Why Compare Them?

• *Reason 1:* CACM ’09: MapReduce is a new way of thinking about programming large distributed systems
  – To DBMS researchers, programming model doesn’t feel new
  – Other communities don’t know this; important to evangelize

• *Reason 2:* Facebook is using Hive (a SQL layer) on Hadoop (MapReduce) to manage 4 TB data warehouse*

• Database community risk losing hearts and minds
  – Important to show where database systems are the right choice

• Important to educate other communities
  – E.g., inform Hive developers about how to architect a parallel DBMS

This Talk

• Comparison of the Two Architectures

• Benchmark
  – Tasks that either system should execute well

• Results on two shared nothing DBMSs & Hadoop

• Navel Gazing
Architectural Differences

• MapReduce operates on *in-situ* data, without requiring transformation or loading

• Schemas:
  – MapReduce doesn’t require them, DBMSs do
  – Easy to write simple MR problems
  – No logical data independence

• Indexes
  – MR provides no built in support
Architectural Differences: Programming Model

• Common to write multiple MapReduce jobs to perform a complex task
  – Google search index construction > 10 MR jobs

• Analogous to multiple joins/subqueries in DBMS
  – E.g., select from subquery

• No built in optimizer in MR to order/unnest these
  – Semantic analysis of arbitrary imperative programs is hard

• MR intermediate results go to disk, pulled by next task
  – Multiple rounds of redistribution likely slow
Architectural Differences: Expressiveness

• SQL+UDFs almost as expressive as MR
  – “Impedance mismatch” is no fun
    • Inelegant programming model
  – DBMS make this much harder than they should
  – UDFs complicate optimization
Architectural Differences: Fault Tolerance

• MR supports mid-query fault tolerance

• DBMSs typically don’t
  – Only important as the number of nodes gets large
  – 1 failure/mo, 1 hour/query →
    \[
    \Pr(\text{mid query failure} | 10 \text{ nodes}) = 1\%
    \]
    \[
    \Pr(\text{mid query failure} | 100 \text{ nodes}) = 13\%
    \]
    \[
    \Pr(\text{mid query failure} | 1000 \text{ nodes}) = 75\%
    \]

• MR doesn’t provide transactions, disaster recovery, etc…
  – Not fundamental
The Benchmark

• Goals
  – Understand differences in load and query time for some common data processing tasks
  – Choose representative set of tasks that:
    • Both should excel at
    • MapReduce should excel at
    • Databases should excel at

• Ran on 100 node Linux cluster at U. Wisconsin
Software

• Hadoop
  – 0.19.0
  – Java 1.6

• DBMS-X
  – Parallel shared nothing row-store from a major vendor
  – Hash partitioned, sorted and indexed beneficially
  – Compression enabled
  – Great deal of manual tuning required

• Vertica
  – Parallel shared nothing column-oriented database (version 2.6)
  – Compression enabled
  – Default parameters, except hints that only one query runs at a time
  – No secondary indices, tables sorted beneficially
• Used in original MapReduce paper

• Look for a 3 character pattern in 90 byte field of 100 byte records with schema:

key VARCHAR(10) PRIMARY KEY
field VARCHAR(90)
– Pattern occurs in .01% of records

SELECT * FROM T WHERE field LIKE ‘%XYZ%’

• 1 TB of data spread across 25, 50, or 100 nodes
– ~10 billion records, 10–40 GB / node

• Expected Hadoop to perform well
Databases don’t scale linearly*; Hadoop does

* Vertica 3.0 adds improved parallel loading; early experiments show linear scalability
Vertica’s compression works better than DBMS-X here

Near linear speedup

<table>
<thead>
<tr>
<th>Dataset</th>
<th>Time (seconds)</th>
</tr>
</thead>
<tbody>
<tr>
<td>25x40GB</td>
<td>1400</td>
</tr>
<tr>
<td>50x20GB</td>
<td>800</td>
</tr>
<tr>
<td>100x10GB</td>
<td>400</td>
</tr>
</tbody>
</table>
Analytical Tasks

- Simple web processing schema
- 600,000 randomly generated documents / node
  - Contents embeds URLs
  - ~8 GB / node
- 155 million user visits / node
  - ~20 GB / node
- See paper for complete schema

```
CREATE TABLE Documents (  
    url VARCHAR(100) PRIMARY KEY,  
    contents TEXT  
);
```

```
CREATE TABLE UserVisits (  
    sourceIP VARCHAR(16),  
    adRevenue FLOAT,  
    ... //fields omitted  
);
```

Loading performance similar to grep
Aggregation Task

• Simple aggregation query to find adRevenue by IP prefix

```
SELECT SUBSTR(sourceIP, 1, 7), sum(adRevenue)
FROM userVisits GROUP BY SUBSTR(sourceIP, 1, 7)
```

• Parallel analytics query for DBMSs
  – Compute partial aggregate on each node, merge answers to produce result
  – Yields 2,000 records (24 KB)
public void map(Text key, Text value, OutputCollector<Text, DoubleWritable> output, Reporter reporter) throws IOException {
    //
    // Split the value using VALUE_DELIMITER into separate fields
    // The *third* field should be our revenue
    //
    String fields[] = value.toString().split("\" + BenchmarkBase.VALUE_DELIMITER);
    if (fields.length > 0) {
        try {
            Double revenue = Double.parseDouble(fields[2]);
            key = new Text(key.toString().substring(0, 7));
            output.collect(key, new DoubleWritable(revenue));
        } catch (ArrayIndexOutOfBoundsException ex) {
            System.err.println("ERROR: Invalid record for key " + key + "");
        }
    }
    return;
}
Performance Results

All systems show linear scaleup
DBMS perform better
No parsing overheads
Compression
Vertica wins because of column orientation
Additional DBMS-y Tasks

• Paper also reports selection and join tasks
• Hadoop also performs poorly on these
  – Hadoop code for join gets hairy
• Vertica does very well on some tasks due to column-orientation
Discussion

• Hadoop much easier to set up
• Hadoop load times are faster
  – (Loading is basically non-existential)
• Hadoop query times are a lot slower (1–2 orders of magnitude)
  – Parsing and indexing
  – Compression
  – Execution model
When to Choose MapReduce

• MapReduce is designed for one-off processing tasks
  – Where fast load times are important
  – No repeated access

• As such, lack of a schema, lack of indexing, etc is not so alarming

• However, no compelling reason to choose MR over a database for traditional database workloads
  – E.g., scalability appears to be comparable despite claims to the contrary
Conclusion

• MapReduce and Database Systems fill different niches
  – One-off processing vs repeated re-access

• MapReduce goodness
  – Ease of use, “out of box” experience
  – Single programming language
  – Attractive fault tolerance properties
  – Fast load times

• Database goodness
  – Fast query times
    • Due to indexing and compression
  – Supporting tools

• Many recently proposed hybrids
  – Hive, Pig, Scope, DryadLinq, HadoopDB, etc.
  – Interesting to see how their performance and usability compares
  – How much can be layered on top of MapReduce?

Call to arms
Because Hadoop is open source and massively parallel, people are using it
Open source shared nothing parallel DBMS is needed
Osprey – MapReduce Style Fault Tolerance for DBMSs

Christopher Yang, Christine Yen, Ceryen Tan

To Appear in ICDE 2010
Motivation

• Long running database queries on distributed databases may crash mid-flight
  – Especially as more nodes are added
• Workload skew, machine load, and other factors may lead to imbalanced distribution of work

Would be good to allow queries to tolerate faults and slow workers *a la* MapReduce
Approach

- OspreyDB is a middleware layer built on top of N Postgres installations
- Tables horizontally partitioned across machines
  - Partitions replicated $k$ times using chained declustering
  - Each node is backup for previous $k$ nodes
- Queries decomposed into fragments, each of which runs on one partition
  - Coordinator assembles answers to fragments to answer query
- Job scheduler tracks liveness of replicas, dispatches fragments
  - Fragments of slow and failed nodes reissued
  - Several different scheduling policies
Architecture

User

Query

Answer

Coordinator

Query Transformer

Result Merger

Query Manager and Scheduler

Worker 1

Worker 2

... Worker N

SQL Fragment

Result

Postgres 1 Part 1, 2

Postgres 2 Part 2, 3

... Postgres 3 Part 3, 1
Experiments

Questions:

– How well does it scale?
– How high are the overheads?
– How well does it balance load?
– How well does it tolerate faults?

Setup:

8 Machines, Running Postgres 8.3; k=1
SSB scale 1 (5.5 GB) (1.1 GB/Node)
512 MB buffer pool per node
Scaleup

Linear Speedup Test

Test completion time (minutes)

Number of workers (n)

- Osprey
- Ideal Speedup
Stressing 1–3 Nodes (N=4)

Replication factor k

Load Balancing Test

Test completion time (minutes)

Stress (s)
Failed Node

Load Balance Over Time

Relative Fraction of Total Computation Time

- Red: Worker 1
- Blue: Worker 2
- Green: Worker 3

Time (s)
Conclusion

• Osprey: middleware-based solution for mid-query fault tolerance

• Decomposes queries into sub-jobs, schedules them with different policies

• Achieves linear speedup (for small numbers of nodes) and good fault tolerance properties
  – Results are not on MR cluster because we don’t use Hadoop

• Next talk – similar idea, slightly different architecture, scales to many more nodes and with a much more robust query processor