CS122D: Beyond SQL Data Management
—Lecture #15—

Mike Carey
UC Irvine
mjcarey@ics.uci.edu
Announcements

• We are no longer in NoSQL territory...!

<table>
<thead>
<tr>
<th>Date</th>
<th>Topic</th>
<th>Material</th>
</tr>
</thead>
<tbody>
<tr>
<td>W 4/21</td>
<td>Document stores: JSON and MongoDB</td>
<td>Ch. 9 NoSQL Distilled (old!)</td>
</tr>
<tr>
<td>M 4/26</td>
<td>Document stores: MongoDB (cont.)</td>
<td>MongoDB materials (as needed)</td>
</tr>
<tr>
<td>W 4/28</td>
<td>NoSQL DB design principles</td>
<td>Ch. 3 NoSQL Distilled</td>
</tr>
<tr>
<td>M 5/03</td>
<td><strong>Midterm Exam (Checkpoint)</strong></td>
<td><strong>3:30-4:50 PM -- be there!!!</strong></td>
</tr>
<tr>
<td>W 5/05</td>
<td>Document DBs: Couchbase Server &amp; N1QL</td>
<td>CB Analytics paper</td>
</tr>
<tr>
<td>M 5/10</td>
<td>Document DBs: Couchbase Server (cont.)</td>
<td>CB materials (as needed)</td>
</tr>
<tr>
<td>W 5/12</td>
<td>Graph DBs: Graph modeling &amp; Neo4J</td>
<td>Ch. 11 NoSQL Distilled (old!)</td>
</tr>
<tr>
<td>M 5/17</td>
<td>Graph DBs: Neo4J (cont.)</td>
<td>Neo4J materials (as needed)</td>
</tr>
<tr>
<td>W 5/19</td>
<td>Big Data Analytics: Google, MapReduce, HDFS</td>
<td>Big Data Platforms paper (skim)</td>
</tr>
</tbody>
</table>

• HW #5 is in flight – and due one week from today

<table>
<thead>
<tr>
<th>HW4</th>
<th>Th 5/06</th>
<th>Mo 5/17 (11:59 PM)</th>
<th>CB Server</th>
<th>HW4 Setup</th>
<th>HW4 Details</th>
<th>HW4 Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>HW5</td>
<td>Mo 5/17</td>
<td>Th 5/27 (11:59 PM)</td>
<td>Neo4J</td>
<td>HW5 Setup</td>
<td>HW5 Details</td>
<td>HW5 Solution</td>
</tr>
</tbody>
</table>

• Today: **First-Generation Big Data Platforms**
Everyone’s Talking About Big Data

• Driven by unprecedented growth in data being generated and its potential uses and value
  – Tweets, social networks (statuses, check-ins, shared content), blogs, click streams, various logs, ...
  – Facebook: > 2.8B monthly active users, > 1.8B daily!
  – Twitter: > 192M daily active users, > 500M tweets/day
Today’s Device-ive World
Big Data: The “Four V’s”

• Volume
  • These days, data can be in the petabyte (PB) range

• Variety
  • Structured (and variant) data, text, spatial data, images, audio, video, ...

• Velocity
  • Arriving at high speed (and also continuously)

• Veracity
  • From many sources, of varying quality, so “dirty data” and possibly noisy or biased or ...
Big Data in the *Database* World

• Enterprises needed to store and query historical business data (data warehouses)
  • 1980’s: Parallel database systems based on “shared-nothing” architectures (Gamma/GRACE, Teradata)
  • 2000’s: Netezza, Aster Data, DAT Allegro, Greenplum, Vertica, ParAccel (“Big $” acquisitions!)

• OLTP is another category (a source of Big Data)
  • 1980’s: Tandem’s NonStop SQL system
Parallel Database Software Stack

Notes:
• One storage manager per machine in a parallel cluster
• Upper layers orchestrate their shared-nothing cooperation
• *One way in/out*: Through the SQL door at the top
Big Data in the Systems World

• Google and others needed a way to process web-scale data (late 1990’s) – SQL not the answer
  • Data much larger than what fits on one machine
  • Need parallel processing to get results in reasonable time
  • Use cheap commodity machines to do the job!

• A useable solution must
  • Scale to 1000s of compute nodes
  • Must automatically handle faults
  • Provide monitoring of jobs
  • Be “easy” for programmers to use

MapReduce Programming Model

- Input and Output are sets of key/value pairs
- Programmer simply provides two functions
  - **map**\((K_1, V_1) \rightarrow \text{list}(K_2, V_2)\)
    - Produces a list of intermediate key/value pairs for each input key/value pair
  - **reduce**\((K_2, \text{list}(V_2)) \rightarrow \text{list}(K_3, V_3)\)
    - Produces a list of result values for all intermediate values that are associated with the same intermediate key

- MapReduce platform applies these UDFs to the data
  - In parallel,
  - At scale, and
  - Even if *(when)* things fail!

Google brought us “Parallel programming for dummies”
MapReduce in Action

**Map** $(K_1, V_1) \rightarrow list(K_2, V_2)$
- Processes one input key/value pair
- Produces a set of intermediate key/value pairs

**Reduce** $(K_2, list(V_2)) \rightarrow list(K_3, V_3)$
- Combines intermediate values for one particular key
- Produces a set of merged output values (usually one)

Resulting puzzle: How can I cast my problem into a sequence of map/reduce steps...? 😊
Basic MapReduce Pipeline

- **Map**
- **Shuffle (= parallel sort)**
- **Reduce**

Read from DFS

Write to DFS
MapReduce Execution Architecture

MapReduce Job Tracker

Network

MapReduce
MapReduce
MapReduce
MapReduce

Distributed File System (DFS)
Software Components

• Job Tracker (Master)
  • Maintains Cluster membership of workers
  • Accepts MR jobs from clients and dispatches tasks to workers
  • Monitors workers’ progress
  • Restarts tasks in the event of failure

• Task Tracker (Worker)
  • Provides an environment to run a task
  • Maintains and serves intermediate files between Map and Reduce phases
MapReduce Partitioned Parallelism

Hash-based Partitioning
Ex. 1: Count Word Occurrences

• Input: Set of (Document name, Document Contents)
• Output: Set of (Word, Count(Word))
• map(K1, V1):
  for each word w in V1
    emit(w, 1)
• reduce(K2, V2_list):
  int result = 0;
  for each v in V2_list
    result += v;
  emit(K2, result)
Ex. 1: Count Word Occurrences

Map

this, 1
is, 1
a, 1
line, 1

Map

this, 1
is, 1
another, 1
line, 1

Map

another, 1
line, 1

Map

yet, 1
another, 1
line, 1

Reduce

a, 1
another, 1
is, 1
is, 1

Reduce

another, 1
another, 1

Reduce

line, 1
line, 1
this, 1
this, 1

Reduce

line, 1
line, 1
yet, 1

line, 4
this, 2
yet, 1
Ex. 2: Joins in MapReduce

• Input: Rows of Relation R, Rows of Relation S
• Output: R join S on R.x = S.y
• map(K1, V1)
  
  if (input == R)
    emit(V1.x, ["R", v1])
  else
    emit(V1.y, ["S", v1])

  (Emit join key and an R/S tagged record)

• reduce(K2, V2_list)

  for r in V2_list where r[1] == “R”
    for s in V2_list where s[1] == “S”
      emit(1, result(r[2], s[2]))

  (Emit all R/S pairs for this join key)
Word Count Steps (More Detail)

Other Examples

• Distributed grep
• Inverted index construction
• Distributed sort
• Similarity join
• Machine learning
• ....
Fault-Tolerant Execution

• Task fault-tolerance achieved through re-execution
• All consumers consume data only after it has been completely generated by the producer
  • This is an important property to isolate faults to one task
• Task completion committed through Master
• Cannot handle master failure
Optimization: Use of *Combiners*

- Sometimes partial aggregation is possible on the Map side
- May cut down amount of data to be transferred to the reducer (sometimes significantly)
- Signature: `combine(K2, list(V2)) -> K2, list(V2)`
- For Word Count example, `combine == reduce`
Ex. 1: Count Word Occurrences (With Combiners)

Map
- this, 1
- is, 1
- a, 1
- line, 1

Reduce
- a, 1
- another, 1
- is, 2

Map
- another, 1
- line, 1
- yet, 1
- another, 1
- line, 1

Reduce
- line, 2
- this, 2
- line, 2
- yet, 1
- line, 4
- this, 2
- yet, 1
An Aside: Does This Look Familiar?

```sql
SELECT br.brewery_id, COUNT(*) AS num_beers
FROM beers br
GROUP BY br.brewery_id;
```

(1980's/1990's parallel relational DB aggregate query processing...!)
MapReduce Under the Hood

https://data-flair.training/blogs/how-hadoop-mapreduce-works/
Optimization: Task Redundancy

• Slow workers lengthen completion time
• Slowness happens because of
  • Other jobs consuming resources
  • Bad disks/network, etc.
• Solution: Near the end of the job, spawn extra copies of long running tasks
  • Whichever copy finishes first is the winner
  • Kill the rest of the copies
• In Hadoop this is called “speculative execution”
Optimization: Locality

• Task scheduling policy
  • Ask GFS for locations of replicas of input file blocks
  • Map tasks are then scheduled (placed) so that their input blocks will be machine-local or rack-local
• Effect: Tasks can read data at local disk speeds
• Without this, rack switches can limit the data rate
Animated Word Count
Google File System (GFS)

- Google’s answer to a distributed file system
- Used as the persistent “store” of data
- MapReduce jobs read input from GFS and write output to GFS
- Provides a “shared disk” view to applications using local storage on shared-nothing hardware
- Provides redundancy via replication to protect against node/disk failures
GFS Architecture

- Single Master (with backups) that track DFS file name to chunk mapping
- Several Chunk servers that store chunks on local disks
- Chunk Size ~ 64MB or larger
- Chunks are replicated
- Master only used for chunk lookups – does not participate in transfers of data

Replication in GFS

• Maintains several replicas of each chunk
  • Usually 3 copies

• Replicas usually spread across racks and data centers to maximize availability

• GFS master tracks location of each replica of a chunk

• When chunk failure is detected, master automatically rebuilds a new replica to maintain the replication level

• Automatically picks chunk servers for new replicas based on utilization
Soon a Star Was Born

• Yahoo!, Facebook, and friends read Google’s papers
  • HDFS and Hadoop MapReduce gained wide usage for indexing, clickstream analysis, log analysis, ...

• Higher-level languages were subsequently developed
  • Pig (Yahoo!): Relational algebra based
  • Hive (Facebook): SQL based
  • Jaql (IBM) and others....
  • HLLs became quickly gained favor over raw MapReduce

• Similar story unfolded at Microsoft
  • Cosmos DFS
  • Dryad vs. SCOPE (SQL-like) for jobs
Open Source Big Data Stack

Architecture:
- Giant byte sequence files at the bottom
- Map, sort, shuffle, reduce layer in middle
- Possible storage layer in middle as well
- **Now at the top**: HLL’s

*Note:* Google’s papers inspired open source cloning!
Open Source Big Data Stack

We've talked a lot about M/R ... so now let’s talk about its HLLs and compilers.

We’ve talked a lot about M/R ...

---

Note: Google’s papers inspired open source cloning!
Apache Pig

• Scripting language inspired by the relational algebra
  • Compiles down to a series of Hadoop MR jobs
  • Relational operators include LOAD, FOREACH, FILTER, GROUP, COGROUP, JOIN, ORDER BY, LIMIT, ...

4. Pig Word Count

We show an example of classic word count application

1. Pig Word Count Download Package

2. Pig Word Count Scripts

```pig
A = load './input.txt';
B = foreach A generate flatten(TOKENIZE((chararray)$0)) as word;
C = group B by word;
D = foreach C generate COUNT(B), group;
store D into './wordcount';
```
Apache Hive (and HiveQL)

- Query language inspired by an old favorite: SQL
  - Compiles down to a series of Hadoop MR jobs
  - Supports various HDFS file formats (text, columnar, ...)
  - Numerous contenders appeared since that take a non-MR-based runtime approach – these include Impala, Stinger, and Spark SQL, ...

```
CREATE TABLE docs (line STRING);
LOAD DATA INPATH 'text' OVERWRITE INTO TABLE docs;
CREATE TABLE word_counts AS
SELECT word, count(1) AS count FROM
(SELECT explode(split(line, '\s')) AS word FROM docs) w
GROUP BY word
ORDER BY word;
```
The Big Data Tangle...
Another Approach... 😊

**Semistructured Data Management**

“One size fits a bunch”

**BDMS Desiderata:**
- Flexible data model
- Efficient runtime
- Full query capability
- Cost proportional to task at hand (!)
- Designed for continuous data ingestion
- Support today’s “Big Data data types”
- : 
- : 

**Parallel Database Systems**

**Data-Intensive Computing**
Another Approach... 😊

**Semistructured Data Management**

**Query**

```
FROM Messages AS msg,
    word_tokens(msg.message) AS word
GROUP BY word
SELECT word, COUNT(1) AS wordcnt
ORDER BY wordcnt DESC;
```

**Output**

```
Results:

{ "word": "line", "wordcnt": 4 }
{ "word": "another", "wordcnt": 3 }
{ "word": "this", "wordcnt": 2 }
{ "word": "is", "wordcnt": 2 }
{ "word": "yet", "wordcnt": 1 }
{ "word": "a", "wordcnt": 1 }

Duration of all jobs: 0.088 sec
```

**SQL++**

**Parallel Database Systems**

**Data-Intensive Computing**

M. Carey, Spring 2021: CS122D
MapReduce/Hadoop Summary

- The innovation here might be summarized as: “Fault-tolerant parallel programming for dummies”
  - We just need to write map(), reduce(), and combine() functions

- The platform handles their partitioned-parallel execution
  - Map tasks read records from (local) storage and produce key value pairs – so they can be run in parallel on the storage nodes
  - Reduce tasks take the pairs for a single grouping key and boil them down (i.e., aggregate them) into a single output for that key – so they can be run in parallel for different grouping keys
  - A shuffle (sort) in the middle transports the groups from the mappers to the appropriate reducers, with local storage (not the DFS) being used in the middle
  - MapReduce jobs read their input from and write their output to files in the DFS
  - Various optimizations (most notably combiners) can also be employed

- Hadoop is the Apache open source ecosystem for all of this
  - Hadoop Map/Reduce
  - HDFS: Hadoop Distributed File System
Questions?