Google’s MapReduce & Bigtable

Jun Yang
Duke University
CPS 399.28: Research Seminar & Project in Databases
Jan. 22, 2008

With contents from D. Weld’s lecture slides and
E. Paulson’s OS seminar slides at U. Washington

Announcements

- Two volunteers needed for next week
  - Christopher M. Jermaine, Subramanian Arumugam, Abhijit Pol,
    Alin Dobra: “Scalable approximate query processing with the
    DBO engine.” SIGMOD 2007
  - Lyublena Antova, Christoph Koch, Dan Olteanu: “From complete
    to incomplete information and back.” SIGMOD 2007

- Preliminary reading list will be posted by tomorrow
  - We will have one more iteration

- Beginning of this course may feel like “rapid-firing,” as we
  quickly move across topics
  - Intended to broadly sample the literature and give you some
    ideas for projects
  - Will come back for more depth after project proposal
    presentations

Papers today

- Jeffrey Dean, Sanjay Ghemawat: “MapReduce: Simplified Data
  Processing on Large Clusters.” OSDI 2004
- Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah
  A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, Robert
  Gruber: “Bigtable: A Distributed Storage System for Structured
  Data.” OSDI 2006

Let’s look at what powers the all-powerful Google!

MapReduce

What was a single phrase that you remembered after reading this paper?

For me: restricted programming model
  - Yes, there is stuff that you cannot do, but
  - The restrictions allow for better system-level support
    (optimization, fault tolerance, …)
  - A large class of problems still can be tackled

MapReduce motivation

- Large-scale data processing
  - Want to use 1000s of machines, but don’t want hassle
    of managing things
- MapReduce provides
  - Automatic parallelization & distribution
  - Fault tolerance
  - I/O scheduling
  - Monitoring & status updates

map/reduce

- Programming model from Lisp (and other
  functional languages)
  - (map square ‘(1 2 3 4)) ⇒ (1 4 9 16)
  - (reduce + ‘(1 4 9 16)) ⇒ 30
- Many problems can be phrased this way
- Easy to distribute
- Nice failure/retry semantics
MapReduce of Google

- **map(key, val)** ⇒ (new-key, new-val), …
  - Run on each item in an input set
  - Emit a list of key-value pairs
- **reduce(key, vals)** ⇒ output
  - Run for each unique key and all associated values emitted by map()
  - Emit output

Example: counting words in docs

- Input: (url, contents) pairs
- **map(key=url, val=contents):**
  - For each word w in contents, emit (w, 1)
- **reduce(key=word, vals=counts):**
  - Sum up all values in counts
  - Emit result (word, sum)

Example: inverted index

- Input: (docid, contents) pairs
- **map(key=docid, val=contents):**
  - For each word w at offset in contents, emit (w, (docid, offset))
- **reduce(key=word, vals=list of (docid, offset)):**
  - Sort vals
  - Emit (word, sorted list of (docid, offset))

Counting illustrated

Example: inverted index

- Input: (docid, contents) pairs
- **map(key=docid, val=contents):**
  - For each word w at offset in contents, emit (w, (docid, offset))
- **reduce(key=word, vals=list of (docid, offset)):**
  - Sort vals
  - Emit (word, sorted list of (docid, offset))

Model is widely applicable

- Typical cluster
  - 100s/1000s of 2-CPU x86 machines, 2-4GB of memory
  - Limited bisection bandwidth
  - Storage is on local IDE disks
  - GFS: distributed file system manages data (SOSP 2003)
  - Job scheduling system: jobs made up of tasks; scheduler assigns tasks to machines
- MapReduce implemented as a C++ library linked into user programs

Implementation overview

- For those of us who work in academia, how do we convince people that a model is “widely” applicable?
Execution: conceptual view

Split input key/value pairs into $M$ chunks, run a map() task on each chunk in parallel
Output from map() tasks is periodically written to local disk, partitioned into $R$ regions according to intermediate (emitted) keys
After all $M$ map()s complete, each of the $R$ reduce() tasks gathers its input regions from mappers, sorts (and thereby groups) data by intermediate keys, and processes each group
Output from reduce() typically goes into $R$ files in GFS
A single “master” assigns and coordinate tasks

Execution details

Why not pipeline map and reduce phases?
– Once map() produces an intermediate result, let reduce() consume it
  ◆ What would we gain from pipelining?
Googler’s answer (cf. http://www.youtube.com/watch?v=x6PUdf3Js)
– No good reason to start reduce() when it isn’t working full throttle
– Not a good use of bandwidth—most of which is taken by input to map()
– Mapper crashing is more difficult to handle—need to remember how much of its output has been consumed

Execution: detailed view

Where is the “bottleneck”?

Why not pipeline map and reduce phases?
– Once map() produces an intermediate result, let reduce() consume it
  ◆ What would we gain from pipelining?
Googler’s answer (cf. http://www.youtube.com/watch?v=x6PUdf3Js)
– No good reason to start reduce() when it isn’t working full throttle
– Not a good use of bandwidth—most of which is taken by input to map()
– Mapper crashing is more difficult to handle—need to remember how much of its output has been consumed

So why this big sync. barrier?

Fault tolerance

– Handle faults using re-execution
  – Detect failure via periodic heartbeats
  – Re-execute in-progress reduce() tasks
  – Re-execute in-progress + completed (why?) map() tasks on the failed machine
– Master failure?
  – Could handle, but don’t yet—unlikely

Refinements

– Redundant execution
  – Slow workers significantly delay completion
  – Near end of phase, spawn backup tasks—whoever finishes first “wins”
    ◆ Alleviates the bottleneck between map/reduce somewhat
– Locality optimization
  – Try to schedule map() on the same machine with the input file block
– Skipping bad records
  – If master sees two failures for the same record, next restart will be told to skip the record
  – Can work around third-party bugs
More refinements

- Ordering guarantees
  - Within a region, key/value pairs are processed in increasing key order
  - Can this guarantee be too strong?
- Combiner
  - In same region: \((key_1, val_1), (key_2, val_2) \rightarrow (key, val_1 + val_2)\)
  - Reduce intermediate result size
  - By the way, what does map/reduce remind you in databases?
- Counters
  - E.g., number of key/value pairs processed
  - Periodically propagated to master; useful in monitoring
  - Can you just use map/reduce to implement a counter?

Lessons learned

- Restricted programming model
  - Restricted programming model
  - Restricted programming model
  - Restricted programming model
  - …
- Locality is important to meet bandwidth constraint
- Simple solutions for nasty problems—especially when you have a massive farm of machines
  - E.g., re-execution

Bigtable motivation

Google scale

- Lots of data
- Many incoming requests
- No commercial system is big enough
  - Couldn’t afford it even if there was one
  - Might not have made appropriate design choices
- 450,000 machines
  - NY Times estimate, June 14, 2006
- Bigtable: scalable, flexible, application-friendly data service

Building blocks

- Google WorkQueue (scheduler)
- GFS: large-scale distributed file system (SOSP 2003)
  - Master: responsible for metadata
  - Chunk servers: responsible for r/w large chunks of data
  - Chunks replicated on 3 machines; master responsible for ensuring replicas exist
- Chubby: lock/file/name service (OSDI 2006)
  - Coarse-grained locks; can store small amount of data in a lock
  - 5 replicas; need a majority vote to be active

Data model: a big map

- Key: \((row, column, timestamp)\)
- Arbitrary “columns” on a row-by-row basis
  - Each column is identified by family:qualifier
  - Cell-oriented physical storage, because rows are sparse (do not have a fixed format)
- No integrity constraints
- No multirow transactions

Implementation: SSTable

- Lives in GFS
- Immutable, sorted file of key-value pairs
- Chunks of data plus an index
  - Index is on block key ranges, not on values
Implementation: tablet

- Tablet Start:aardvark End:apple
- Tablet log

<table>
<thead>
<tr>
<th>Tablet</th>
<th>SSTable</th>
<th>Index</th>
<th>64K block</th>
<th>64K block</th>
<th>64K block</th>
<th>64K block</th>
</tr>
</thead>
</table>
- 100-200 MB; split if it gets too long
- Contains data in some row range of a bigtable
- Implemented on top of multiple SSTables and a tablet log (in GFS)
  - Not a simple union of SSTables!

Implementation: bigtable

- A bigtable is horizontally partitioned into multiple non-overlapping tablets according to row ranges
- Tablets could share underlying SSTables!

Finding a tablet

- Like a B+-tree, but fixed at 3 levels
- How can we avoid creating a bottleneck at the root?
  - Aggressively cache tablet locations
  - Lookup starts from leaf (bet on it being correct); reverse on miss

Serving a tablet

- Updates are logged
- Each SSTable corresponds to a batch of updates or a snapshot of the tablet taken at some earlier time
- Memtable (sorted by key) caches recent updates (those not yet reflected by SSTables)
- Reads consult both memtable and SSTables

Compactions

- Minor compaction
  - Memtable → a new SSTable
  - Reduce memory usage and redoing during recovery
- Merging compaction
  - A few SSTables + memtable → a new SSTable
  - Reduce number of SSTables
  - Can apply policies such as “keep only N older versions”
- Major compaction
  - All SSTables + memtable into one
  - Deleted data is now completely purged

Refinements

- Locality groups
  - Group similar column families
  - A separate SSTable for each local group
  - Avoid mingling data (e.g., page contents vs. metadata)
  - Small groups may even be kept in memory
- Compression
  - More effective due to locality grouping and versioning
- Bloom filters
  - Read may need to consult all SSTables
  - Use an in-memory Bloom filter to avoid GFS accesses
Some performance numbers

<table>
<thead>
<tr>
<th>Experiment</th>
<th># of Tablet Servers</th>
</tr>
</thead>
<tbody>
<tr>
<td>random reads</td>
<td>515</td>
</tr>
<tr>
<td>random writes</td>
<td>581</td>
</tr>
<tr>
<td>sequential reads</td>
<td>620</td>
</tr>
<tr>
<td>sequential writes</td>
<td>619</td>
</tr>
<tr>
<td>scan</td>
<td>535</td>
</tr>
</tbody>
</table>

- In the table on the left, why is throughput going down when # of tablet servers increases?
- Why are random writes almost as efficient as sequential writes?
- What's the price we are paying for that?

Lessons learned

- Don't go overboard with features
- Many types of failures are possible in a real system
- Big systems need proper system-level monitoring
- KISS = Keep It Simple, Stupid!

For more info, see [video](http://videosrv14.cs.washington.edu/info/videos/mp4/colloq/JDean_051018_OnDemand_100_256K_320x240.mp4)

Discussion

So, in a course on databases, why are we starting with two systems papers?
- Analogous to MapReduce, relational algebra/SQL also followed a restricted programming model
- Analogous to Bigtable, relational database systems also aim to provide a scalable, flexible, application-friendly data service
- Many aspects of database research involve designing such abstractions
- Writing papers about designs is very difficult, but you have seen a couple of good examples!