Distributed computing: index building and use

Distributed computing Goals

Distributing computation across several machines to

- Do one computation faster - latency
- Do more computations in given time - throughput
- Tolerate failure of 1+ machines

Distributing computations

Ideas?

⇒ Finding results for a query?

- Building index?

- Goals
  - Keep all machines busy
  - Be able to replace badly-behaved machines seamlessly!

Distributed Query Evaluation: Strategies

- Assign different queries to different machines
- Break up multi-term query: assign different query terms to different machines
  - good/bad consequences?
- Break up lexicon: assign different index terms to different machines?
  - good/bad consequences?
- Break up postings lists: Assign different documents to different machines?
  - good/bad consequences?

Keep all machines busy?
Seamlessly replace badly-behaved machines?

Example:
Google query evaluation circa 2002

- Parallelize computation
  - distribute documents randomly to pieces of index
    - Pool of machines for each piece- choose one
    - Why random?
- Load balancing and reliability
  - Scheduler machines
    - assign tasks to pools of machines
    - monitor performance

5

Google Query Evaluation: Details
circa 2002

- Enter query -> DNS-based directed to one of geographically distributed clusters
- w/in cluster, query directed to 1 Google Web Server (GWS)
- GWS distributes query to pools of machines
- Query directed to 1 machine w/in each pool

6

Google Query Evaluation: Details
circa 2002

- Enter query -> DNS-based directed to one of geographically distributed clusters
  - Load balance & fault tolerance
  - Round-trip time
- w/in cluster, query directed to 1 Google Web Server (GWS)
  - Load balance & fault tolerance
- GWS distributes query to pools of machines
  - Load sharing
- Query directed to 1 machine w/in each pool
  - Load balance & fault tolerance

7

Issues for distributed documents

- How many take from each pool to get m results?
- Throughput limits?
  - each machine does full query evaluation
  - disk access limiting constraint?
  - distributing index by term instead may help

8

Distributing computations

✓ Finding results for a query?

⇒ Building index?

9

Distributed Index Building

- Can easily assign different documents to different machines
- Efficient?
- Goals
  - Keep all machines busy
  - Be able to replace badly-behaved machines seamlessly!

10

Google Index Building circa 2003: MapReduce framework

- programming model
- implementation for large clusters

- Google introduced for index building and PageRank
  "for processing and generating large data sets"
- The Apache Hadoop project developed open-source software
- Other applications:
  - database queries
    - join like multi-term query eval.
  - statistics on queries in given time period

11

MapReduce Programming Model

- input set: {(input key_i, $value_i$)| $0 \le i \le$ input size}
  - user chooses type value – e.g. whole document
- output set: {(output key_i, $value_i$)| $0 \le i \le$ output size}

- Map (written by user):

(input key, value) →
{(intermed. key_j, $value_j$)| $0 \le j \le$ Map result size}

- system groups all Map output pairs by intermediate key (shuffle phase)
  - gathers by intermediate key value
  - supply to Reduce by iterator

- Reduce (written by user) process intermediate values:

(intermed. key, list of values) → (output key, value)

12

MapReduce for building inverted index

- Input pair: (docID, contents of doc)
- Map: produce {(term, docID)} for each term appearing in docID
- Input to Reduce: (term, docIDs) pairs for each term
- Output of Reduce: (term, sorted list of docIDs containing that term)
  - postings list!

keys

Matrix – Vector multiplication

i, j range over elements of matrix A and vector ***v***
q ranges over chunks of ***v*** and strips of A
p ranges over chunks of strips of A

Input: tuples (q, (p, chunk A_{pq}, chunk $\boldsymbol{v}_q$))
Map input tuple to tuples for i in range of p:
(i, Σ ($A_{i,j}$ v_j) = x_{iq}) with sum over all j in chunk q: $\boldsymbol{v}_q$ and $A_{p,q}$

Shuffle gives (i, list of $x_{i,q}$ all q)

Reduce to: (i, $\Sigma_q x_{iq} = \Sigma_q \Sigma_{j \text{ in } q} A_{i,j} v_j = (A\boldsymbol{v})_i$

Matrix-vector multiplication diagram

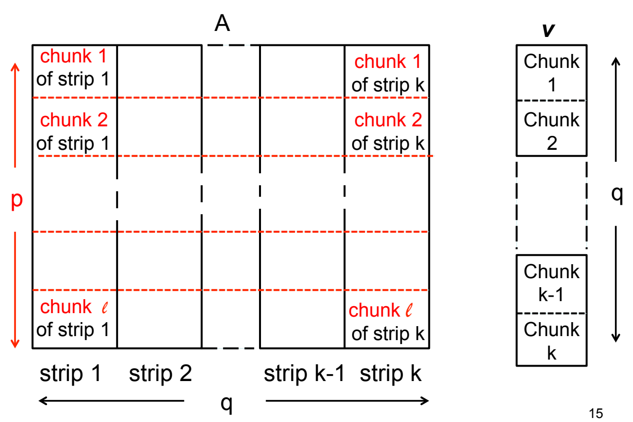

Diagram of computation distribution

See Figure 2.3 (pg 27) in
Mining of Massive Data Sets by Rajaraman, Leskovec and Ullman

Originally appeared as Figure 1 in
MapReduce: Simplified Data Processing on Large Clusters by J. Dean and S. Ghemawat,
Comm. of the ACM,vol. 51, no. 1 (2008), pp. 107-113.

MapReduce parallelism

- Map phase and shuffle phase may overlap
- Shuffle phase and reduce phase may overlap
- Map phase must finish before reduce phase starts
  - reduce depends on all values associated with a given key

17

MapReduce Fault Tolerance

- Master fails => restart whole computation
- Worker node fails
  - Master detects failure
  - must redo all Map tasks assigned to worker
    - output of completed Map tasks on failed worker's disk
  - for failed Map worker, Master
    - reschedules each Map task
    - notifies reducer workers of change in input location
  - for failed Reduce worker, Master
    - reschedules each Reduce task
  - rescheduling occurs as live workers become available

18

Hadoop

"The Apache Hadoop project develops open-source software for reliable, scalable, distributed computing. "

Includes MapReduce

http://hadoop.apache.org/index.html

19

Remarks

- Google built on large collections of inexpensive "commodity PCs"
  - always some not functioning
- Solve fault-tolerance problem in software
  - redundancy & flexibility NOT special-purpose hardware
- Keep machines relative generalists
  - machine becomes free ⇒ assign to any one of set of tasks

20

June 2010 New Google index building: Caffeine

- daily crawl "several billion" documents
- Before:
  - Rebuild index: new + existing
  - series of 100 MapReduces to build index
  - "each doc. spent 2-3 days being indexed"
- After:
  - Each document fed through Percolator: incremental update of index
  - Document indexed 100 times faster (median)
  - Avg. age doc. in search result decr. "nearly 50%"

21

Percolator

- Built on top of *Bigtable* distributed storage
  - "tens of petabytes" in indexing system
- Provides random access
  - Requires extra resources over MapReduce
- Provides transaction semantics
  - Repository transformation highly concurrent
  - Requires some consistency guarantees for data
- "Observers" do tasks; write to table
- Writing to table creates work for other observers
- "around 50" Bigtable op.s to process 1 doc.

22

Bigtable Overview

- Distributed database system
  - One big, sparse table
  - Sorted by row key
- Rows partitioned into tablets
  - contiguous key space
- Tablet servers execute operations
  - large number tablet servers: Performance!
- Fault tolerance
  - replication of data
  - transaction log
    - server take over for failed server

23

Percolator observers

- users write observer code
- run distributed across collection of machines
- observer "registers" function and set of columns with Percolator
- Percolator invokes function after data written in one of columns (any row)
  - Percolator must find "dirty" cells
    - search distributed across machines
  - avoid >1 observer for a single column

24

Caffeine versus MapReduce

- Caffeine uses "roughly twice as many resources" to process same crawl rate
- New document collection "currently 3x larger than previous systems"
  - Only limit available disk space
- Document indexed 100 times faster (median)
- If number newly-crawled docs near size index, MapReduce better
  - random lookup v.s. streaming

Earlybird: Real-Time Search at Twitter
by many Twitter researchers (2012)

- Designed for properties of tweets
  - Handle high rate of queries
  - Handle large number updates in real time
    - "Flash crowds"
    - Update info, eg number of retweets
  - Large number concurrent reads and writes
  - Time stamp dominant ranking signal

Elements

- Distributed server architecture
  - Tweets hash partitioned across servers
- New concurrency management
- Customized query processing
- Customized inverted index

Query processing

- Full Boolean query language
- Results returned most recent first
- Personalized signals in relevance algorithm (not described)
  - User's local social graph
  - "actual query algorithm isn't particularly interesting"

    "reuse existing Lucene query eval code"

Inverted Index

- Dictionary
  - Hash table on term ID
  - Term ID points to tail of postings list
- Postings lists
  - organized in segments
  - Each server has small number segments (12)
  - Each segment has small number tweets, ≤ 2^{23}
  - Only one segment active
  - In-active segments read-only

29

Active segment index

- Posting is 32-bit integer
  - 24 bits doc ID; 8 bits term position
  - each occurrence in tweet is new posting
- Postings list: pre-allocated integer array
  - Dynamic allocation
- Traversing newest first = iterate bkwds
- Can traverse bkwds from any point while concurrently adding new postings
- Can binary search for doc ID
  - Eliminate need skip pointers

30

In-active segments

- Replaces an active segment when filled
- One fixed-size integer array
  - Dictionary points to different postings lists
- Arranged reverse chronologically
- Compressed
  - Short postings list: as before
  - Long postings list:
    - uses gaps
    - block-based compression

31

Earlybird performance

- Compare prior MySQL-based
  - 1000 tweets per second indexing
  - 12,000 queries per second
- Earlybird memory
  - Full active index segment (16M tweets) 6.7 GB
  - Full in-active index segment ~ 55% above
- Queries per second
  - 5000 for fully-loaded server (114M tweets)
- Tweets per second
  - 7000 in “stress test”- heavy query load

32