Information Retrieval System: Semantic Content Based Big Data Processing Approaches by Google

By: Sankalp Godugu
Date: January 31, 2018
Background (Big Data)

• What is it?
  • voluminous
  • complex

• Why is it important?
  • to uncover hidden patterns, correlations, and other insights

• Where is it used?
  • various companies
    • social media data
    • browser logs
    • text analytics
Why not just use commercial DB?

- scale is too large for most databases
- even if it weren’t, cost would be very high
  - building internally means system can be applied across many projects for low cost
Goals

● Want asynchronous processes to be continuously updating different pieces of data
  ○ want to access most current data at any time
● need to support:
  ○ very high read/write rates (millions of ops/second)
  ○ efficient scans over all or interesting subsets of data
  ○ efficient joins of large one-to-one or one-to-many datasets
● often want to examine data changes over time
  ○ e.g., contents of a web page over multiple crawls
Background (Bigtable)

- distributed multi-level map
  - with interesting data model
- fault-tolerant, persistent
- scalable
  - thousands of servers
  - terabytes of in-memory data
  - petabyte of disk-based data
  - millions of reads/writes per second, efficient scans
- self-managing
  - servers can be added/removed dynamically
  - servers adjust to load imbalance
Status

- design/initial implementation started in 2004
  - currently ~100 Bigtable clusters
- production use or active development for many projects
  - Crawling/indexing
  - Google Maps/Google Earth
- Largest Bigtable cluster manages ~200TB data spread over several thousand machines
Overview

1. building blocks
2. data model
3. client API
4. underlying Google infrastructure
5. fundamentals of Bigtable implementation
6. several examples
7. lessons learned
Building Blocks

● Building blocks:
  ○ Google File System (GFS): raw storage
  ○ Scheduler: schedules jobs onto machines
  ○ Lock service: distributed lock manager
    ■ also can reliably hold tiny files (100s of bytes) w/ high availability

● Bigtable uses of building blocks:
  ○ GFS: stores persistent state
  ○ Scheduler: schedules jobs involving Bigtable
  ○ Lock Service: master election
  ○ MapReduce: often used to read/write BigTable data
Google File System

- master manages metadata
- chunk servers store chunks of files
- data transfers happen directly between clients and chunk servers
- files broken into chunks (64MB)
- chunks triplicated across 3 machines for safety
MapReduce

- Many Google problems: process lots of data to produce other data
- Many kinds of inputs
  - document records, log files, sorted on-disk data structures, etc.
- want to use easily hundreds or thousands of CPUs
- MapReduce: framework that provides (for certain classes of problems):
  - automatic & efficient parallelization/distribution
  - fault-tolerance, I/O scheduling, status/monitoring
  - user writes Map and Reduce functions
- heavily used: ~3000 jobs, 1000s of machine days each day
Figure 1: A slice of an example table that stores Web pages. The row name is a reversed URL. The contents column family contains the page contents, and the anchor column family contains the text of any anchors that reference the page. CNN’s home page is referenced by both the Sports Illustrated and the MY-look home pages, so the row contains columns named anchor:cnnsi.com and anchor:my.look.ca. Each anchor cell has one version; the contents column has three versions, at timestamps $t_3$, $t_5$, and $t_6$.

Data Model (Figure 1)
- Distributed multi-dimensional sparse map
- Good match for most applications
Data Model (Rows)

- name is an arbitrary string
  - access to data in a row is atomic
  - row creation is implicit upon storing data
- rows ordered lexicographically
  - rows close together lexicographically usually on one or a small number of machines
Data Model (Columns)

- Columns have two-level name structure:
  - family:optional_qualifier
- Column family
  - units of access control
  - has associated type info
- Qualifier gives unbounded columns
  - additional level of indexing, if desired
Timestamps

- Used to store different versions of data in a cell
  - new writes default to current time, but timestamps for writes can also be set explicitly by clients
- Lookup options:
  - “Return most recent K values”
  - “Return all values in timestamp range (or all values)”
- Column families can be marked w/ attributes
  - “Only retain most recent K values in a cell
  - “Keep values until they are older than K seconds”
Data Model (Tablets)

- large tables broken into tablets at row boundaries
  - tablet holds contiguous range of rows
    - clients can often choose row keys to achieve locality
  - aim for ~100MB to ~200MB of data per table
- serving machine responsible for ~100 tablets
  - fast recovery:
    - 100 machines each pick up 1 tablet from failed machine
    - master makes load-balancing decision
Tablet Splitting

● at some point, a tablet may get to large b/c it is larger than a certain size target or load on that tablet is unusually high
● row is chosen as the point of separation for the tablet
API (Figure 2)

- Bigtable API provides various functions
- Client apps can write/delete, search, iterate
  - Figure 2 shows C++ code that uses RowMutation abstraction
  - Call to Apply performs atomic mutation to Webtable

```cpp
// Open the table
Table *T = OpenOrDie("/bigtable/web/webtable");

// Write a new anchor and delete an old anchor
RowMutation r1(T, "com.cnn.www");
r1.Set("anchor:www.c-span.org", "CNN");
r1.Delete("anchor:www.abc.com");
Operation op;
Apply(&op, &r1);
```

Figure 2: Writing to Bigtable.
API (Figure 3)

• Figure 3 shows C++ code that uses Scanner abstraction
  • clients can iterate over multiple families
  • several mechanisms for limiting rows, columns and timestamps
  • for example
    • could restrict the scan above to only produce anchors whose columns match the regular expression “anchor:*\.com”
    • or to only produce anchors whose timestamps fall within 10 days of current time

```c++
Scanner scanner(T);
ScanStream *stream;
stream = scanner.FetchColumnFamily("anchor");
stream->SetReturnAllVersions();
scanner.Lookup("com.cnn.www");
for (; !stream->Done(); stream->Next()) {
    printf("%s %s %lld %s\n",
        scanner.RowName(),
        stream->ColumnName(),
        stream->MicroTimestamp(),
        stream->Value());
}
```

**Figure 3:** Reading from Bigtable.
Implementation

1. Bigtable implementation has 3 major components
   1. library linked to EVERY client (expanded upon in next slide)
   2. ONE master server
      1. assigning tablets to tablet servers
      2. detecting addition/expiration of tablet servers
      3. balancing load of files in GFS
      4. garbage collection
      5. handles schema changes (e.g., table and column family creations)
   3. MANY tablet servers
      1. each tablet server manages a set of tablets (10-1000 tablets/tablet server)
      2. handles read/write requests to tablets that it has loaded
      3. splits tablets that have grown too large
   4. clients communicate directly with tablet servers for reads/writes
      1. Bigtable clients do not rely on master for tablet location info so most clients NEVER communicate with master
      2. master is lightly loaded in practice
Implementation (Tablet Location)

- 3-level hierarchy used (Figure 4)
  - Chubby (lock service) file containing location of root tablet
  - root tablet contains location of ALL tablets in METADATA table
  - each METADATA tablet contains location of set of tablets
- METADATA table stores location of a tablet
- client library caches tablet locations
  - client recursively moves up tablet location hierarchy
  - tablet locations stored in memory so no GFS accesses are required

Figure 4: Tablet location hierarchy.
Implementation (Tablet Assignment)

- each tablet assigned to 1 tablet server at a time
  - master keeps track of the set of live tablet servers and current assignment of tablets
  - master assigns the tablet by sending tablet load request to tablet server when tablet server with enough room is available

- Bigtable uses lock service (Chubby) to keep track of tablet servers
  - when a tablet server starts, Bigtable creates/acquires lock on unique file in Chubby directory
  - master monitors that servers directory to discover tablet servers
  - tablet server stops serving its tablets if it loses its exclusive lock (e.g., network partition)
  - tablet server will attempt to re-acquire exclusive lock on its file (if file exists)
  - if file no longer exists, tablet server kills itself
  - whenever a tablet server terminates, Bigtable attempts to release its lock so master can reassign tablets
Implementation (Tablet serving)

- persistent state of a tablet stored in GFS
  - updates are committed to commit log
  - of these updates, recently committed ones are stored in memory buffer called **memtable**
  - older updates stored in SSTables (file format)
  - to recover a tablet, tablet server reads METADATA table
  - metadata contains list of SSTables that comprise a tablet (pointers to commit logs containing data for tablet)
  - server reads indices of SSTables into memory and reconstructs memtable by applying updates from commit log
- when a write operation arrives at a tablet server, the server checks that it is well-formed and if sender is authorized
  - authorization is performed by reading permitted writer list
- when a read operation arrives at a tablet server, similar checks
  - valid read operation is executed on memtable
Performance Evaluation

- Bigtable cluster with $N$ tablet servers to measure performance and scalability of Bigtable as $N$ is varied
- Tablet servers and master, test clients, GFS servers all ran on same set of machines
- $R$ is number of distinct number of Bigtable row keys involved in the test
- **sequential write benchmark** used row keys with names 0 to $R - 1$
  - row keys partitioned into $10N$ equal-sized ranges (tablets)
  - these ranges were assigned to the $N$ clients by central scheduler that assigned the next available range to a client as soon as client finished processing the previous range assigned to it (sequential)
  - single string written under each row key
  - each string generated randomly
  - strings under different row keys were distinct
- **random write benchmark**
  - similar to sequential write except that row key was hashed modulo $R$ immediately before writing so that write load is spread uniformly across entire row space
Performance Evaluation (continued)

- **sequential read benchmark**
  - generated row keys in exactly the same way as sequential write benchmark
  - read the string stored under the row key (instead of writing)
- **random read benchmark**
  - shadowed operation of random write benchmark
- **scan benchmark**
  - similar to sequential read benchmark
  - uses support provided by Bigtable API for scanning over all values in a row range
- **random reads (mem) benchmark**
  - similar to random read benchmark
  - locality group that contains benchmark data marked as in-memory (reads satisfied from tablet server’s memory instead of requiring GFS)
Performance Evaluation (Figure 6)

- shows two views on performance of benchmarks
- Table shows # operations/second per tablet server
- Graph shows aggregate # operations/second

<table>
<thead>
<tr>
<th>Experiment</th>
<th>1</th>
<th>50</th>
<th>250</th>
<th>500</th>
</tr>
</thead>
<tbody>
<tr>
<td>random reads</td>
<td>1212</td>
<td>593</td>
<td>479</td>
<td>241</td>
</tr>
<tr>
<td>random reads (mem)</td>
<td>10811</td>
<td>8511</td>
<td>8000</td>
<td>6250</td>
</tr>
<tr>
<td>random writes</td>
<td>8850</td>
<td>3745</td>
<td>3425</td>
<td>2000</td>
</tr>
<tr>
<td>sequential reads</td>
<td>4425</td>
<td>2463</td>
<td>2625</td>
<td>2469</td>
</tr>
<tr>
<td>sequential writes</td>
<td>8547</td>
<td>3623</td>
<td>2451</td>
<td>1905</td>
</tr>
<tr>
<td>scans</td>
<td>15385</td>
<td>10526</td>
<td>9524</td>
<td>7843</td>
</tr>
</tbody>
</table>

Figure 6: Number of 1000-byte values read/written per second. The table shows the rate per tablet server; the graph shows the aggregate rate.
Performance Evaluation (Single tablet-server performance)

1. When # of Tablet Servers = 1
   1. random reads → slower than ALL other operations
      1. each random read involves transfer of a 64KB SSTable block
      2. tablet server executes approx. 1200 reads/second
   2. random reads (mem) → much faster
      1. each 1KB read is satisfied from tablet server’s local memory
   3. random and sequential writes → perform better than random reads
      1. each tablet server appends all incoming writes
      2. each tablet server uses group commit
      3. no significant difference
      4. in both cases, all writes to tablet server are recorded
   4. sequential reads → perform better than random reads
      1. every 64KB SSTable block that is fetched from GFS is stored into block cache
   5. scans → even faster
      1. tablet server can return large number of values
      2. therefore, RPC overhead is amortized
Performance Evaluation (Scaling)

1. aggregate throughput increases dramatically
   1. example → performance of random reads from memory increases by factor ~300
   2. this behavior occurs b/c the bottleneck
2. performance does NOT increase linearly
   1. for most benchmarks, there is significant drop in per-server throughput
   2. drop caused by imbalance in load in multiple server configs
   3. load balancing algorithm attempts to deal with this imbalance but cannot do perfect job for 2 reasons:
      1. rebalancing is throttled to reduce # tablet movements
      2. load generated by benchmark shifts around as benchmark progresses
3. random read benchmark shows worst scaling
   1. increase in aggregate throughput by only a factor of ~100
   2. this behavior occurs b/c one large 64KB block is transferred
   3. this transfer saturates various shared 1Gb links in network
Real Applications

- As of August 2006 → 388 Bigtable clusters running in various Google machine clusters, with combined total of ~24,500 tablet servers
- Many of these clusters used for development purposes
  - some tables store data that is served to users
  - others store data for batch processing
  - tables range widely in
    - total size
    - average cell size,
    - % data served from memory
    - complexity of table schema
- 3 products teams
Table 1: Distribution of number of tablet servers in Bigtable clusters.

<table>
<thead>
<tr>
<th># of tablet servers</th>
<th># of clusters</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 .. 19</td>
<td>259</td>
</tr>
<tr>
<td>20 .. 49</td>
<td>47</td>
</tr>
<tr>
<td>50 .. 99</td>
<td>20</td>
</tr>
<tr>
<td>100 .. 499</td>
<td>50</td>
</tr>
<tr>
<td>&gt; 500</td>
<td>12</td>
</tr>
<tr>
<td>Project name</td>
<td>Table size (TB)</td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------</td>
</tr>
<tr>
<td>Crawl</td>
<td>800</td>
</tr>
<tr>
<td>Crawl</td>
<td>50</td>
</tr>
<tr>
<td>Google Analytics</td>
<td>20</td>
</tr>
<tr>
<td>Google Analytics</td>
<td>200</td>
</tr>
<tr>
<td>Google Base</td>
<td>2</td>
</tr>
<tr>
<td>Google Earth</td>
<td>0.5</td>
</tr>
<tr>
<td>Google Earth</td>
<td>70</td>
</tr>
<tr>
<td>Orlait</td>
<td>9</td>
</tr>
<tr>
<td>Personalized Search</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 2: Characteristics of a few tables in production use. Table size (measured before compression) and # Cells indicate approximate sizes. Compression ratio is not given for tables that have compression disabled.
Real Applications (Google Analytics)

- provides aggregate stats such as % users that made purchase, given that they earlier viewed a specific page
  - # unique visitors/day
  - page views/URL per day
  - site-tracking reports
- to enable service, webmasters embed small JavaScript program in their web pages
  - program is invoked whenever page is visited
  - records various info about request in GA (e.g., user identifier and info about page)
  - GA summarizes this data and makes it available to webmasters
- Two of the tables used by GA
  - raw click table (~200TB) maintains a row for each user/client
    - row name is a tuple containing websites name and time at which session was created
    - this schema ensures that sessions that visit same web site are contiguous and sorted
  - summary table (~20TB) contains predefined summaries for each website
    - generated from raw click table by periodic MapReduce jobs
    - each MapReduce job extracts recent data from raw click table
Real Applications (Google Earth)

- operates collection of services that provide users with access to high-res satellite imagery
  - products allow users to navigate across world’s surface
    - System uses one table to preprocess data and different set of tables for serving client data
  - preprocessing pipeline uses one table to store raw imagery
    - during preprocessing, imagery is cleaned
    - this table contains approx. 70TBs data (thus served from disk)
- each row in imagery table corresponds to single geographic segment
  - rows named to ensure adjacent geographic segments stored next to each other
  - preprocessing pipelines rely heavily on MapReduce over Bigtable to transform data
- serving system uses one table to index data stored in GFS
  - this table is relatively small (~500GB) but must serve many queries w/ low latency
  - as a result, this table is hosted across hundreds of tablet servers
Real Applications (Personalized Search)

- opt-in service that records user queries and clicks
  - users can browse their search histories (e.g., revisit old queries/clicks)
  - they can ask for personalized search results based on their historical usage
- Personalized Search stores each user’s data in Bigtable
  - each user has unique userID assigned as row key
  - all user actions are stored in a table
  - separate column family is reserved for each type of action (e.g., column family that stores all web queries)
  - each data element uses as its Bigtable timestamp time at which corresponding action occurred
- using MapReduce over Bigtable, PS generates user profiles for personalization
Lessons

• large distributed systems are vulnerable to many types of failures
  • memory and network corruption
  • hung machines
  • extended and asymmetric network partitions
  • bugs in other systems that are used
  • overflow of GFS quotas
  • planned/unplanned hardware maintenance

• importance of proper system-level monitoring
  • for a sample of RPCs, detailed trace of important actions done is kept
    • fix slow writes to GFS
    • stuck accesses to METADATA table
  • every Bigtable cluster is registered in Chubby
    • allows for discovering size of clusters
    • how much traffic
    • whether or not there are any problems
References

- https://storify.com/haileybrooks/the-5-w-s-of-big-data
- https://www.bernardmarr.com/default.asp?contentID=1076
- https://www.cs.rutgers.edu/~pxk/417/notes/content/bigtable.html