Motivation

- Lots of (semi-)structured data at Google
  - URLs:
    • Contents, crawl metadata, links, anchors, pagerank, ...
  - Per-user data:
    • User preference settings, recent queries/search results, ...
  - Geographic locations:
    • Physical entities (shops, restaurants, etc.), roads, satellite image data, user annotations, ...

- Scale is large
  - billions of URLs, many versions/page (~20K/version)
  - Hundreds of millions of users, thousands of q/sec
  - 100TB+ of satellite image data
Why not just use commercial DB?

• Scale is too large for most commercial databases

• Even if it weren’t, cost would be very high
  – Building internally means system can be applied across many projects for low incremental cost

• Low-level storage optimizations help performance significantly
  – Much harder to do when running on top of a database layer

Also fun and challenging to build large-scale systems :)
Goals

• Want asynchronous processes to be continuously updating different pieces of data
  – Want access to most current data at any time

• Need to support:
  – Very high read/write rates (millions of ops per second)
  – Efficient scans over all or interesting subsets of data
  – Efficient joins of large one-to-one and one-to-many datasets

• Often want to examine data changes over time
  – E.g. Contents of a web page over multiple crawls
BigTable

- Distributed multi-level map
  - With an 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 beginning of 2004
• Currently ~100 BigTable cells
• Production use or active development for many projects:
  – Google Print
  – My Search History
  – Orkut
  – Crawling/indexing pipeline
  – Google Maps/Google Earth
  – Blogger
  – ...
• Largest bigtable cell manages ~200TB of data spread over several thousand machines (larger cells planned)
Background: 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
- **MapReduce:** simplified large-scale data processing

BigTable uses of building blocks:
- **GFS:** stores persistent state
- **Scheduler:** schedules jobs involved in BigTable serving
- **Lock service:** master election, location bootstrapping
- **MapReduce:** often used to read/write BigTable data
Google File System (GFS)

- Master manages metadata
- Data transfers happen directly between clients/chunkservers
- Files broken into chunks (typically 64 MB)
- Chunks triplicated across three machines for safety
- See SOSP'03 paper at http://labs.google.com/papers/gfs.html
MapReduce: Easy-to-use Cycles

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

See: "MapReduce: Simplified Data Processing on Large Clusters", OSDI'04

BigTable can be input and/or output for MapReduce computations
BigTable Overview

• Data Model
• Implementation Structure
  – Tablets, compactions, locality groups, …
• API
• Details
  – Shared logs, compression, replication, …
• Current/Future Work
Basic Data Model

- Distributed multi-dimensional sparse map
  
  \[(row, column, timestamp) \rightarrow cell\ contents\]

- Good match for most of our applications
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
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 \(~100\)MB to \(200\)MB of data per tablet
- Serving machine responsible for \(~100\) tablets
  - Fast recovery:
    - 100 machines each pick up 1 tablet from failed machine
  - Fine-grained load balancing:
    - Migrate tablets away from overloaded machine
    - Master makes load-balancing decisions
Tablets & Splitting

```
"aaa.com"
"cnn.com"
"cnn.com/sports.html"

---

"website.com"

---

"zuppa.com/menu.html"
```
System Structure

Bigtable Cell

- **Bigtable master**
  performs metadata ops + load balancing

- **Bigtable tablet server**
- **Bigtable tablet server**
- **Bigtable tablet server**
System Structure

Bigtable Cell

- **Bigtable master**: performs metadata ops + load balancing
- **Bigtable tablet server**: serves data
- **Cluster scheduling system**: handles failover, monitoring
- **GFS**: holds tablet data, logs
- **Lock service**: holds metadata, handles master-election
Locating Tablets

- Since tablets move around from server to server, given a row, how do clients find the right machine?
  - Need to find tablet whose row range covers the target row

- One approach: could use the BigTable master
  - Central server almost certainly would be bottleneck in large system

- Instead: store special tables containing tablet location info in BigTable cell itself
Locating Tablets (cont.)

- Our approach: 3-level hierarchical lookup scheme for tablets
  - Location is ip:port of relevant server
  - 1st level: bootstrapped from lock service, points to owner of META0
  - 2nd level: Uses META0 data to find owner of appropriate META1 tablet
  - 3rd level: META1 table holds locations of tablets of all other tables
    - META1 table itself can be split into multiple tablets
Tablet Representation

- Write buffer in memory (random-access)
- Append-only log on GFS

SSTable: Immutable on-disk ordered map from string->string
String keys: <row, column, timestamp> triples
Compactions

- Tablet state represented as set of immutable compacted SSTable files, plus tail of log (buffered in memory)

- Minor compaction:
  - When in-memory state fills up, pick tablet with most data and write contents to SSTables stored in GFS
    - Separate file for each locality group for each tablet

- Major compaction:
  - Periodically compact all SSTables for tablet into new base SSTable on GFS
    - Storage reclaimed from deletions at this point
Columns

- Columns have two-level name structure:
  - family:optional_qualifier

- Column family
  - Unit of access control
  - Has associated type information

- 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”
Locality Groups

- Column families can be assigned to a **locality group**
  - Used to organize underlying storage representation for performance
    - scans over one locality group are $O(\text{bytes\_in\_locality\_group})$, not $O(\text{bytes\_in\_table})$
  - Data in a locality group can be explicitly memory-mapped
Locality Groups

- contents:
  - www.cnn.com
  - ...

- language: "pagerank:
  - EN
  - 0.88

"Google"
API

- **Metadata operations**
  - Create/delete tables, column families, change metadata

- **Writes (atomic)**
  - `set()`: write cells in a row
  - `delete_cells()`: delete cells in a row
  - `delete_row()`: delete all cells in a row

- **Reads**
  - `Scanner`: read arbitrary cells in a bigtable
    - Each row read is atomic
    - Can restrict returned rows to a particular range
    - Can ask for just data from 1 row, all rows, etc.
    - Can ask for all columns, just certain column families, or specific columns
Shared Logs

- Designed for 1M tablets, 1000s of tablet servers
  - 1M logs being simultaneously written performs badly
- Solution: shared logs
  - Write log file per tablet server instead of per tablet
    - Updates for many tablets co-mingled in same file
  - Start new log chunks every so often (64 MB)
- Problem: during recovery, server needs to read log data to apply mutations for a tablet
  - Lots of wasted I/O if lots of machines need to read data for many tablets from same log chunk
Shared Log Recovery

Recovery:
- Servers inform master of log chunks they need to read
- Master aggregates and orchestrates sorting of needed chunks
  - Assigns log chunks to be sorted to different tablet servers
  - Servers sort chunks by tablet, writes sorted data to local disk
- Other tablet servers ask master which servers have sorted chunks they need
- Tablet servers issue direct RPCs to peer tablet servers to read sorted data for its tablets
Compression

• Many opportunities for compression
  – Similar values in the same row/column at different timestamps
  – Similar values in different columns
  – Similar values across adjacent rows

• Within each SSTable for a locality group, encode compressed blocks
  – Keep blocks small for random access (~64KB compressed data)
  – Exploit fact that many values very similar
  – Needs to be low CPU cost for encoding/decoding

• Two building blocks: BMDiff, Zippy
BMDiff

- Bentley, McIlroy DCC'99: "Data Compression Using Long Common Strings"
- Input: dictionary = source
- Output: sequence of
  - COPY: <x> bytes from offset <y>
  - LITERAL: <literal text>

- Store hash at every 32-byte aligned boundary in
  - Dictionary
  - Source processed so far
- For every new source byte
  - Compute incremental hash of last 32 bytes
  - Lookup in hash table
  - On hit, expand match forwards & backwards and emit COPY
- Encode: ~100 MB/s, Decode: ~1000 MB/s
Zippy

- LZW-like: Store hash of last four bytes in 16K entry table
- For every input byte:
  - Compute hash of last four bytes
  - Lookup in table
  - Emit COPY or LITERAL

- Differences from BMDiff:
  - Much smaller compression window (local repetitions)
  - Hash table is not associative
  - Careful encoding of COPY/LITERAL tags and lengths

- Sloppy but fast:

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>% remaining</th>
<th>Encoding</th>
<th>Decoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gzip</td>
<td>13.4%</td>
<td>21 MB/s</td>
<td>118 MB/s</td>
</tr>
<tr>
<td>LZO</td>
<td>20.5%</td>
<td>135 MB/s</td>
<td>410 MB/s</td>
</tr>
<tr>
<td>Zippy</td>
<td>22.2%</td>
<td>172 MB/s</td>
<td>409 MB/s</td>
</tr>
</tbody>
</table>
BigTable Compression

- **Keys:**
  - Sorted strings of (Row, Column, Timestamp): prefix compression

- **Values:**
  - Group together values by “type” (e.g. column family name)
  - BMDiff across all values in one family
    - BMDiff output for values 1...N is dictionary for value N+1

- **Zippy as final pass over whole block**
  - Catches more localized repetitions
  - Also catches cross-column-family repetition, compresses keys
Compression Effectiveness

- Experiment: store contents for 2.1B page crawl in BigTable instance
  - Key: URL of pages, with host-name portion reversed
  - Groups pages from same site together
    - Good for compression (neighboring rows tend to have similar contents)
    - Good for clients: efficient to scan over all pages on a web site

- One compression strategy: gzip each page: ~28% bytes remaining
- BigTable: BMDiff + Zippy:

<table>
<thead>
<tr>
<th>Type</th>
<th>Count (B)</th>
<th>Space (TB)</th>
<th>Compressed</th>
<th>% remaining</th>
</tr>
</thead>
<tbody>
<tr>
<td>Web page contents</td>
<td>2.1</td>
<td>45.1 TB</td>
<td>4.2 TB</td>
<td>9.2%</td>
</tr>
<tr>
<td>Links</td>
<td>1.8</td>
<td>11.2 TB</td>
<td>1.6 TB</td>
<td>13.9%</td>
</tr>
<tr>
<td>Anchors</td>
<td>126.3</td>
<td>22.8 TB</td>
<td>2.9 TB</td>
<td>12.7%</td>
</tr>
</tbody>
</table>
In Development/Future Plans

- More expressive data manipulation/access
  - Allow sending small scripts to perform read/modify/write transactions so that they execute on server?
- Multi-row (i.e. distributed) transaction support
- General performance work for very large cells
- BigTable as a service?
  - Interesting issues of resource fairness, performance isolation, prioritization, etc. across different clients