Infrastructure system services

Badri Nath
Rutgers University
badri@cs.rutgers.edu

1. MapReduce Programming Model
2. BigTable or Hypertable
3. Chubby Lock Service
Processing lots of data

- O(B) web pages; each O(K) bytes to O(M) bytes gives you O(T) to O(P) bytes of data
- Disk Bandwidth per computer O(50 MB/sec)
- Processing time O(10^6) secs for O(T) data
- Reduce it to O(10^3) with 1 K processors
- Need high parallelism to process web data
- Web Processing: Process, transform, store
Programming model

- Computation takes input of key, value pairs and transforms into output of key value pairs
- Divided into two
  - map function that produces from the input an intermediate set of key value pairs
  - Reduce function that takes the intermediate key value pairs and merges the values for a given key
- Inherent parallelism as map operates on partition of input and no dependency between map processes
List processing

- Model taken from Functional programming such as list
  - Map (square ' (1, 2, 3, 4)) \(\rightarrow\) (1, 4, 9, 16)
  - Reduce (sum ' (1, 4, 9, 16)) \(\rightarrow\) (30)

- Distributed Grep
  - `cat inp.dat | grep | sort | uniq -c | cat > out.dat`
  - Input| map| sort | reduce | output
Map reduce functions

- Map \((k1,v1) \rightarrow list (k2, v2)\)
  - E.g., \(k1\) document names, \(v1\) content of documents
  - E.g., \(k2\) words, \(v2\) count of words in doc
- Reduce \((k2, list(v2)) \rightarrow k2, sum(list(v2))\)
  - E.g., sum is word count for each word
Map reduce

- Parallel programming in the large
- Automatic parallelization and execution
- Low cost, unreliable hardware
- Maps well to GFS
- Simple and restrictive model (2 functions)
- Works on homogenized data sets
Map reduce example

Input
- Angelina jolie
- Jumped over Cat
- Angelina
- Jolie walks Brad Pitt
- Brad Pitt walks over dog

Map
- Angelina 1
- Cat 1
- Jolie 1
- Jumped 1
- Angelina 1
- Brad 1
- Jolie 1
- Pitt 1 walks 1
- Brad 1
- Dog 1
- Pitt 1 walks 1
- Over 1
- Pitt 1 walks 1
- Over 1

Sort & Partition
- A-J
- K-Z

Reduce
- Angelina 2
- Brad 2
- Cat 1
- Dog 1
- Jolie 2
- Jumped 1
- Over 2
- Pitt 2 walks 2

Output
Building inverted Index - Map

Map

People
magazine

Page 1
Angelina
Obama

Page 2
Brad
Pitt

Jessica
Obama
Angelina

Page 23

Map

Emit

(angelina, 1)
(obama, 1)
(Brad, 2)
(Pitt, 2)
(Jessica, 23)
(Obama, 23)
(Angelina, 23)

Sort

(Angelina, 1)
(Angelina 23)
(Brad, 2)
(Jessica, 23)
(Obama, 1)
(Obama, 23)
(Pitt, 2)
Building inverted Index - reduce

(Angelina, 1)
(Angelina, 23)
(Brad, 2)

(Jessica, 23)
(Obama, 1)
(Obama, 23)
(Pitt, 2)

(Angelina, 1, 23)
(Obama, 1, 23)
(Obama, 23)
(Pitt, 2)

Reduce
Natural Join Operation

Natural Join : rows in R union S, where the values of the attributes in R∩S are same

- Notation: \( r \bowtie s \)

Example:

\( R = (A, B, C, D) \)
\( S = (E, B, D) \)

- Result schema = \( (A, B, C, D, E) \)
- \( r \bowtie s \) is defined as:

\[
\Pi_{r.A, r.B, r.C, r.D, s.E} (\sigma_{r.B = s.B \land r.D = s.D} (r \times s))
\]
Natural Join Operation – Example

- Relations $r$, $s$:

$$
\begin{array}{|c|c|c|c|}
\hline
A & B & C & D \\
\hline
A & 1 & A & a \\
B & 2 & A & a \\
C & 4 & B & b \\
A & 1 & C & a \\
E & 2 & B & b \\
\hline
\end{array}
\quad
\begin{array}{|c|c|c|}
\hline
B & D & E \\
\hline
1 & a & A \\
3 & a & B \\
1 & a & C \\
2 & b & D \\
3 & b & E \\
\hline
\end{array}
$$

$r \bowtie s$

$$
\begin{array}{|c|c|c|c|c|}
\hline
A & B & C & D & E \\
\hline
A & 1 & A & a & A \\
A & 1 & A & a & C \\
A & 1 & C & a & A \\
A & 1 & C & a & C \\
E & 2 & B & b & D \\
\hline
\end{array}
$$
Converting into map reduce

- Convert common attributes into a key, rest of relation is value
- Do that for R and S
- Send tuples in R, and S with the same key value to reducer
- Partition key values to distribute load among reducers
- Partition R and S among mappers
Map reduce example

<table>
<thead>
<tr>
<th>Input</th>
<th>Map</th>
<th>Sort &amp; Partition</th>
<th>Reduce</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>R (A,B,C,D)</td>
<td>1,a, A A</td>
<td>1,a, A A</td>
<td>1-2</td>
<td>A 1 A a A</td>
</tr>
<tr>
<td></td>
<td>2 a B A</td>
<td>2 a B A</td>
<td></td>
<td>A 1 A a C</td>
</tr>
<tr>
<td></td>
<td>4 b C B</td>
<td>1 a A C</td>
<td></td>
<td>A 1 C a A</td>
</tr>
<tr>
<td></td>
<td>1 a A C</td>
<td>2 b E B</td>
<td>1 a A</td>
<td>A 1 C a C</td>
</tr>
<tr>
<td></td>
<td>2 b E B</td>
<td></td>
<td>2 b D</td>
<td>E 2 B b D</td>
</tr>
<tr>
<td>S(B,D,E)</td>
<td>1 a A</td>
<td></td>
<td>4 b C B</td>
<td></td>
</tr>
<tr>
<td></td>
<td>3 a B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3 a B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>1 a C</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>2 b D</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3 b E</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>3 b E</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>4 b C B</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Join as Map-Reduce

- Each reducer matches all pairs with same key values
- Reducer outputs (A, B, C, D, E) once all mappers are done and no more tuples
- Parallelism can be controlled by assigning horizontal fragments to mappers
- Parallelism can be controlled by adjusting range value of keys in reducers
Map-reduce-merge

map: \((k_1, v_1)_\alpha \rightarrow [(k_2, v_2)]_\alpha\)
reduce: \((k_2, [v_2])_\alpha \rightarrow (k_2, [v_3])_\alpha\)
merge: \(((k_2, [v_3])_\alpha, (k_3, [v_4])_\beta) \rightarrow [(k_4, v_5)]_\gamma\)

Map reduce merge: simplified data processing on large clusters, Yang et.al., SIGMOD 2007
LHS mapper computes emp bonuses

LHS reducer sorts on (dept-id, emp-id) pair and sums up emp bonuses

RHS mapper retrieves bonus adjustment

RHS reducer modified bonus adjustment and sorts on dept-id

A sort-merge merger joins LHS and RHS reduced outputs, then computes final emp bonuses.
Figure 1: Execution overview
**MR execution steps**

- Partition the input file \(1\ldots N\)
- Master thread creates as many worker threads as needed
- Workers is assigned a map task that takes data from partition \((i)\)
- Map outputs to buffer (intermediate values)
- Master notifies the reduce worker, it reads the output produced by the mapper
- Reduce worker iterates over sorted data and applies the reduce function
- The output of the reduce function is the final result
Implementation Issues

- Failures
- Locality
- Partition and Combine function
- Backup tasks
- Ordering of results
Failures

- A parallel, distributed computation
- Running on thousands of machines
- Need to deal with failures
- Master detects worker failures, and has work re-done by another worker.
- Master failure: restart the computation (hopefully not make a habit of it)
Failure Semantics

- Distributed computation should produce the same result as non-faulting sequential execution of single program
- Atomic commit of map and reduce tasks
- Written to files and communicated to master
- If multiple copies of the same reduce task are executed, atomic rename will be used so that only 1 out file is created despite redundancy
Locality

- Master program: assigns task threads based on location of data: tries to have map() tasks on same machine as physical file data, or at least same rack/switch
- map() task inputs are divided into 64 MB blocks: same size as Google File System chunks
- Working set mapped to underlying GFS
Task Granularity

- $M + R >>$ Number of machines (for load balancing)
- Not too many Rs, final result need to be combined
- Master needs to keep a mapping of $O(M*R)$
- $M = K$ (Number of machines) $K$ is 100
- $R = F$ (number of machines) $F = 2.5$
Backup Tasks

- Reduce task cannot start until map is complete
- Straggler is a machine that takes unusually long (e.g., bad disk) to finish its work.
- A straggler can delay final completion.
- When task is close to finishing, master schedules backup executions for remaining *in-progress* tasks.
- Must be able to eliminate duplicate results
Partitioning function $\text{hash (key)} \mod R$

<table>
<thead>
<tr>
<th>Input</th>
<th>Map</th>
<th>Sort &amp; Partition</th>
<th>Reduce</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Filenames from A - H</td>
<td>M1</td>
<td>Angelina 1 Cat1 Jolie 1 Jumped 1</td>
<td>A-J</td>
<td>Angelina 2 Brad 2 Cat1 Dog 1 Jolie 2 Jumped 1</td>
</tr>
<tr>
<td>Filenames from I - P</td>
<td>M2</td>
<td>Angelina 1 Brad 1 Jolie 1</td>
<td>K-Z</td>
<td>Over 2 Pitt 2 walks 2</td>
</tr>
<tr>
<td>Filenames from Q - Z</td>
<td>M3</td>
<td>Brad 1 Dog 1 Over 1 Pitt 1 walks 1</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Partitioning function: $\text{hash (key)} \mod R$
Combine or aggregator Function

Assuming commutative, associative operations

Combine is like a local reduce applied before distribution:

Select count(*) from intermediate-file Group By word
Input/output

- Each line as key, value (grep)
- Key is the line # and value is content of line
- Sequence of key value pairs, ordered by keys
- Output file of reducer can also be stored in key order
- Any new file type should be transformed so that it is suitable for range partitioning to map tasks
Performance

- Cluster Configuration
- 1800 machines
- Each machine
  - 2GHz Xeons,
  - 4GB RAM, 2 160GB disk, Gb Ethernet.
  - Two level tree switched network with 100-200 Gbps aggregate at root
Performance

- Grep experiment: look for a pattern
- $M=15000$, $R = 1$
- 10 G, 100 byte records
- 1 minute startup, 150 seconds total!
- Startup cost includes code propagation, opening files in GFS, getting GFS metadata for locality optimization.
- Completes 80 seconds after startup
Performance

- Sort
- Input is 10 G, 100 byte records
- M = 15000, R = 4000
- Completes in 891 secs
- Terasort benchmark 1057!!
MapReduce Conclusions

- MapReduce has proven to be a useful abstraction for large scale data processing using clusters
- Greatly simplifies large-scale computations at Google
- Shown that functional programming paradigm can be applied to large-scale applications
- Lots of problems in processing of web data can be cast as map-reduce
- Now database people have joined the party
  - Map-reduce-merge (sigmod 2007)
New MapReduce Programs Per Month

Summer intern effect
Bigtable “quotes”

- MapReduce on Bigtable: A major step backwards - The Database Column
- *BigTable* and Why it Changes Everything
- Why Have many normalized tables when you can have one Big table
Motivation

- Web 1.0, 2.0, ...
- Characterized by Huge (Terabyte, Petabyte, Exabyte) of data
  - $10^{12}$, $10^{15}$, $10^{18}$ bytes of data
- Google, Facebook, Yahoo, YouTube, Flickr, …
- Different kinds of sources
  - Crawl data – whole web, replicated
  - Portal data – pictures of everyone and everyone they know, inbox for every user
  - Geographic data – satellite images at various resolutions of the entire earth
Relational model

- Normalized relations
- Canonical example
- Part          Supplier          Order
  - Part #, Partname, cost  s#, sname, saddress, part#  S#, P#, Qty
  - Each of the fields need to be atomic – 1NF
  - Other normalization forms to avoid
  - Delete anomaly, update anomaly, etc
A random entry in Facebook
### Friends-- facebook

Users 100 M

<table>
<thead>
<tr>
<th>userid</th>
<th>picture</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><img src="image1" alt="User 1 Picture" /></td>
</tr>
<tr>
<td>2</td>
<td><img src="image2" alt="User 2 Picture" /></td>
</tr>
<tr>
<td>3</td>
<td><img src="image3" alt="User 3 Picture" /></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Friends</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>User id</th>
<th>User id</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>

Every time retrieve user picture and pictures of friends
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
  - Modified at will to suit requirements
  - Nothing like owning source code

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

- A new data model that satisfies the requirements of web data processing
BigTable

- Forget all (most of it) that was taught in DB course
- Store everything in one big table
- Not even 1NF
## Friends- big table

<table>
<thead>
<tr>
<th>User id</th>
<th>picture</th>
<th>friends</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><img src="image1.png" alt="User 1 Picture" /></td>
<td><img src="image2.png" alt="Friend 1" /></td>
</tr>
<tr>
<td>2</td>
<td><img src="image4.png" alt="User 2 Picture" /></td>
<td><img src="image5.png" alt="Friend 3" /></td>
</tr>
<tr>
<td>3</td>
<td><img src="image7.png" alt="User 3 Picture" /></td>
<td><img src="image8.png" alt="Friend 5" /></td>
</tr>
</tbody>
</table>

Each column has GB of data
Big table - Data model

- Sparse, distributed, persistent, multidimensional map
- 4 dimensional
- Map has a row indexed by a key .. 1 dimension
- Column family ... 2 dimension
- Each column family has several columns ... 3 dimension
- Each Column has version based on a timestamp ... 4 dimension
<table>
<thead>
<tr>
<th>User id</th>
<th>picture</th>
<th>Column Fmly Name:friends Sort: username</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><img src="image1.png" alt="Image" /></td>
<td><img src="image2.png" alt="Image" /></td>
</tr>
<tr>
<td>2</td>
<td><img src="image3.png" alt="Image" /></td>
<td><img src="image4.png" alt="Image" /></td>
</tr>
<tr>
<td>3</td>
<td><img src="image5.png" alt="Image" /></td>
<td><img src="image6.png" alt="Image" /></td>
</tr>
</tbody>
</table>

Each column has GB of data
4D map

| Row 1 | | Row 4 |
|-------|---------|
| key   | val     |     |

| Row 2 | |       |
|-------|---------|

| Row 3 | |       |
|-------|---------|

<table>
<thead>
<tr>
<th>ColumnFMLY 1</th>
<th>ColumnFMLY 2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Basic Data Model

- Distributed multi-dimensional sparse map
  \((\text{row, column, timestamp}) \rightarrow \text{cell contents}\)

- Column family is web page that refers to row URL
- No joins, parallel processing on column family
Data model - Row

- Keys are 64KB
- Big table horizontally partitioned into tablet
- Rows are ordered
- A tablet covers a range of keys
- Good locality for data access
Data model - column

- Column families -- static
- Contains columns -- variable
- Has type, name and sort attribute
- Each column has a qualifier or label and value
- Examples: anchor, language, etc

<table>
<thead>
<tr>
<th>ColFmly1 name:anchor</th>
<th>type:simple</th>
<th>sort:tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tag</td>
<td>value</td>
<td>timestamp</td>
</tr>
</tbody>
</table>
Data model - timestamp

- Each cell can have multiple versions
- Indexed by timestamps 64-bit integers
- Garbage collect older versions if needed
Bigtable API

- Implements calls with signatures to
  - create and delete tables and column families,
  - modify cluster, table, and column family metadata such as access control rights,
  - Write or delete values in Bigtable
  - Iterate over all column families
  - Look up values from individual rows,
  - Atomic operation on data stored in a single row key
  - No support for transactions across multiple rows
Building Blocks

- Google File System
  - Cluster file system, High availability.
- SSTable
  - A key/value database mapped to the underlying GFS
- Chubby
  - Distributed lock service
**SSTable**

- A SSTable file format stores bigtable data
  - Stores and retrieves key/data pairs.
    - Key and data are arbitrary byte strings.
  - Select or range query to iterate key/value pairs given a selection predicate (exact and range).
  - Configurable to use either persistent store (disk) or memory-mapped.

- A SSTable is stored in GFS.
Implementation details

1. A Bigtable library linked to every client.
2. Many tablet servers.
   - Tablet servers are added and removed dynamically.
   - Ten to a thousand tablets assigned to a tablet server.
   - Each tablet is typically 100-200 MB in size.
3. One master server responsible for:
   - Assigning tablets to tablet servers,
   - Detecting the addition and deletion of tablet servers,
   - Balancing tablet-server load,
   - Garbage collection of files in GFS.
   - Client communicates directly with tablet server for reads/writes.
Implementation

Bigtable cell
- Bigtable master server: performs metadata ops, load balancing
- Bigtable tablet server: serves data
  - SSStable
- Cluster Scheduling Master: handles failover, monitoring

Bigtable client
- Bigtable client library: Open()
- Bigtable tablet server: serves data
  - SSStable
- GFS: holds tablet data, logs
- Chubby: holds metadata, handles master-election
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 tablet
- Serving machine responsible for 10~1000 tablets per server
  - 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

- abc.com
- cnn.com
- Espn.com
- fox.com
- Rutgers.edu/cs
- Rutgers.edu/cs/graduate
- Rutgers.edu/cs/research
- Wsj.com
- Yahoo.com
Locating Tablets

- Tablets are dynamically mapped to servers
- Need a lookup service lookup(row) → machine that holds the data covering the row
- One approach: could use a BigTable master
  - Not ideal for the workload; will become a bottleneck
- Instead: store special tables containing tablet location info in BigTable cell itself
Locating Tablets: double index

- Root tablet contains pointer to second level metadata tablets
- Metadata tablets point to user tablets
- Metadata is 1 K; metadata tablet size is 128 M
- # of entries 128 K ($2^{17}$)
- Two levels=$2^{34}$

Figure 4: Tablet location hierarchy.
Tablet lookup

Root tablet

Read(crawltable, key 150)

Bigtable Tablet

<table>
<thead>
<tr>
<th>Key</th>
<th>URL</th>
<th>HOST</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td><a href="http://www.xyz">www.xyz</a>.</td>
<td>128.6</td>
</tr>
<tr>
<td>2</td>
<td><a href="http://www.abc">www.abc</a>.</td>
<td>134.8</td>
</tr>
<tr>
<td></td>
<td>..</td>
<td>..</td>
</tr>
<tr>
<td>100</td>
<td>..</td>
<td>..</td>
</tr>
</tbody>
</table>

UserTable tablet

<table>
<thead>
<tr>
<th>Tabletid</th>
<th>Last key</th>
<th>servid</th>
</tr>
</thead>
<tbody>
<tr>
<td>MT1</td>
<td>1000</td>
<td>6</td>
</tr>
<tr>
<td>MT12</td>
<td>2000</td>
<td>4</td>
</tr>
</tbody>
</table>

Metadata tablet

<table>
<thead>
<tr>
<th>Tabletid</th>
<th>Last key</th>
<th>servid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crawl1</td>
<td>100</td>
<td>3</td>
</tr>
<tr>
<td>Crawl2</td>
<td>200</td>
<td>3</td>
</tr>
</tbody>
</table>

/crawltable
Placement of Tablets

- A tablet is assigned to one tablet server at a time.
- Master keeps track of:
  - The set of live tablet servers,
  - Current assignment of tablets to tablet servers (including the unassigned ones)
- Each tablet server needs to have an exclusive lock on the tablet
Tablet server Starts

- Master and set of servers maintained in chubby

CMS

Tablet server

/chubby

/servers/tab_srv9

BigTable master

Start server
Tab-srv9 IP 128.6.4.3

2. Create a chubby file =servid

3. Acquire lock

Assign tablets

Still have lock?

4. Acquire lock

5. Remove server

6. ?any new

7. 8. Acquire lock

CMS=Cluster Management System
Placement of tablet

- Chubby maintains tablet servers:
  - A tablet server creates and acquires an exclusive lock on a uniquely named file in a specific chubby directory (named server directory),
  - Master monitors server directory to discover tablet server,
  - A tablet server stops processing requests if it loses its exclusive lock (due to network partitioning).
    - Tablet server will try to obtain an exclusive lock on its uniquely named file as long as it exists.
    - If the uniquely named file of a tablet server no longer exists then the tablet server kills itself. Goes back to a free pool to be assigned tablets by the master.
Tablet master Starts

- Master and set of servers maintained in chubby

Start master msrv IP 128.8.4.3

1. Start master
2. Create a chubby file =msrv
3. Acquire lock
4. Read Tablet servers in /servers
5. Check Assigned tablets
6. Scan metatable
7.Assigned reassign tablets

CMS=Cluster Management System
Read/write flow

- SSTable: Immutable on-disk ordered map from key→value
- String keys: <row, column, timestamp> triples
Client Write Operation

- Write operation arrives at a tablet server:
  - Server ensures the client has sufficient privileges for the write operation (Chubby),
  - A log record is generated to the commit log file,
  - Once the write commits, its contents are inserted into the memtable.
Client Read Operation

- When Read operation arrives at a tablet server:
  - Server ensures client has sufficient privileges for the read operation,
  - Read is performed on a merged view of (a) the SSTables that constitute the tablet, and (b) the memtable.
Write Operations

- As writes execute, size of memtable increases.
- Once memtable reaches a threshold:
  - Memtable is frozen,
  - A new memtable is created,
  - Frozen memtable is converted to an SSTable and written to GFS.
- Periodic flushing
  - Reduces memory usage at tablet server
  - Recovery time
Compactions

- As new SSTables are created, merge cost increases for read
- Need to keep as few as possible, better yet just one
- Merging compaction in the background
- Keep a few SSTables
- Major compaction:
  - Periodically compact all SSTables for tablet into new base
    SSTable on GFS
    - Storage reclaimed from deletions at this point
Minor Compaction

Write buffer in memory (random-access)

memtable

Write

K1 2
K2 3
K5 7
K18 6

K1 1
K4 3
K6 7

K5 1
K9 3
K11 7

K8 1
K19 3
K21 7

Append-only log on GFS

Tablet

New SSTable on GFS

SSTable on GFS

SSTable on GFS
Major Compaction

Write buffer in memory (random-access)

Tablet

ST1
K1 2
K2 3
K5 7
K18 6

ST2
K1 1
K4 3
K6 7

ST3
K5 1
K9 3
K11 7

ST4
K8 1
K19 3
K21 7

Write

Append-only log on GFS

New SS table S5
Enhancements

- **Locality**
  - Grouping of column family
  - Column store vs row store
- **Compression**
  - SStables can be compressed
- **Caching**
  - Key-value pair caching as well as block cache
- **Bloom filter**
  - Key hashed into vector of bits using multiple hash functions
  - If $bf_i[k]$ is 0 for some $i$ then key is not present else may be present
  - Bit vector length $M$ and hash functions 1…$k$ are design parameters
System Performance

- Experiments involving random reads (from GFS and main memory) and writes, sequential reads and writes, and scans.
  - Scan: A single RPC fetches a large sequence of values from the tablet server.

<table>
<thead>
<tr>
<th>Experiment</th>
<th># of Tablet Servers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>random reads</td>
<td>1212</td>
</tr>
<tr>
<td>random reads (mem)</td>
<td>10811</td>
</tr>
<tr>
<td>random writes</td>
<td>8850</td>
</tr>
<tr>
<td>sequential reads</td>
<td>4425</td>
</tr>
<tr>
<td>sequential writes</td>
<td>8547</td>
</tr>
<tr>
<td>scans</td>
<td>15385</td>
</tr>
</tbody>
</table>

Figure 6: Number of 1000-byte values read/written per second, aggregate rate.
Random Reads

- Single tablet performance

<table>
<thead>
<tr>
<th>Experiment</th>
<th># of Tablet Servers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>random reads</td>
<td>1212</td>
</tr>
<tr>
<td>random reads (mem)</td>
<td>10811</td>
</tr>
<tr>
<td>random writes</td>
<td>8850</td>
</tr>
<tr>
<td>sequential reads</td>
<td>4425</td>
</tr>
<tr>
<td>sequential writes</td>
<td>8547</td>
</tr>
<tr>
<td>scans</td>
<td>15385</td>
</tr>
</tbody>
</table>

Figure 6: Number of 1000-byte values read/written per second. aggregate rate.
Mmap read and seq Reads

- A read request is for 1000 bytes.
- In memread; no network access
- Seq read better de to 64 KB caching of SStable

![Figure 6: Number of 1000-byte values read/written per second. Aggregate rate.](image)
Seq Writes and random Writes

Table: Number of 1000-byte values read/written per second.

<table>
<thead>
<tr>
<th>Experiment</th>
<th># of Tablet Servers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>random reads</td>
<td>1212</td>
</tr>
<tr>
<td>random reads (mem)</td>
<td>10811</td>
</tr>
<tr>
<td>random writes</td>
<td>8850</td>
</tr>
<tr>
<td>sequential reads</td>
<td>4425</td>
</tr>
<tr>
<td>sequential writes</td>
<td>8547</td>
</tr>
<tr>
<td>scans</td>
<td>15385</td>
</tr>
</tbody>
</table>

Figure 6: Number of 1000-byte values read/written per second. Aggregate rate.

- Tablet server appends all incoming writes to a single commit log and uses group commit to stream these writes to GFS efficiently.
Application 1: Google Analytics

- Helps webmasters to analyze traffic pattern at their web sites.
- Provides aggregate statistics such as:
  - Number of unique visitors per day
  - Page views per URL per day,
  - Percentage of users that made a purchase given that they earlier viewed a specific page.
- How?
  - Each click, data is sent to analytics.google.com by a small JavaScript program Every time the page is visited, the program is executed.
  - Program records the following information about each request:
    - User identifier
    - The page being fetched
Application 1: Google Analytics (Cont...)

- Two Bigtables
  - **Raw click table (~ 200 TB)**
    - A row for each end-user session.
    - Row name include website’s name and the time at which the session was created.
    - Clustering of sessions that visit the same web site. And a sorted chronological order.
    - Compression factor of 6-7.
  - **Summary table (~ 20 TB)**
    - Stores predefined summaries for each web site.
    - Generated from the raw click table by periodically scheduled MapReduce jobs.
    - Each MapReduce job extracts recent session data from the raw click table.
    - Row name includes website’s name and the column family is the aggregate summaries.
    - Compression factor is 2-3.
Application 2: Google Earth

- Access to image data
- Operations to Pan, view, and annotate satellite imagery at different resolution levels.
- One Bigtable stores raw imagery (~ 70 TB):
  - Row name is a geographic segment. Names are chosen to ensure adjacent geographic segments are clustered together.
  - Column family maintains sources of data for each segment.
  - Each column contains raw image data
- Index data of 500 GB to serve queries
Application 3: Personalized Search

- Records user queries and clicks across Google properties.
- Users browse their search histories and request for personalized search results based on their historical usage patterns.
- One Bigtable for personalized search:
  - Row name is userid
  - A column family is reserved for each action type, e.g., web queries, clicks.
  - User profiles are generated from activity using MapReduce.
    - These profiles personalize live search results.
  - Replicated geographically to reduce latency and increase availability.
Conclusion

- Benefits of building a system to suit requirements
- New Data model and parallel processing
- Can handle web size data (tera, peta, exa bytes)
- User level processing on columns
- Applicable in a wide variety of settings
Impact

- Lots of web properties using bigtable
- Yahoo – htable; facebook- cassandra
- New data model
Db vs bigtable

- Not in 1 NF
- No transaction across rows
- Simplified query processing
  - No joins
- Design decisions, architecture tailored to cluster
- Lots of innovative features to get performance while dealing with large sets of data