NoSQL

Zhi Wang
wangzhi@sz.tsinghua.edu.cn

Outline

- NoSQL
- Bigtable/HBase

History of Relational DB

- Relational Databases – mainstay of business
- Web-based applications caused spikes
  - Especially true for public-facing e-Commerce sites
- Developers begin to front RDBMS with memcache or integrate other caching mechanisms within the application

Scale

- Issues with scaling up when the dataset is just too big
- RDBMS were not designed to be distributed
- Began to look at multi-node database solutions
- Known as 'scaling out' or 'horizontal scaling'
- Different approaches include:
  - Master-slave
  - Sharding

Scaling RDBMS – Master/Slave

- Master-Slave
  - All writes are written to the master
  - All reads performed against the replicated slave databases
  - Critical reads may be incorrect as writes may not have been propagated down
  - Large data sets can pose problems as master needs to duplicate data to slaves

Scaling RDBMS - Sharding

- Partition or sharding
  - Scales well for both reads and writes
  - Not transparent, application needs to be partition-aware
  - Can no longer have relationships/joins across partitions
  - Loss of referential integrity across shards

Other ways to scale RDBMS

- Multi-Master replication
- INSERT only, not UPDATES/DELETES
- No JOINs, thereby reducing query time
  - This involves de-normalizing data
- In-memory databases

What is NoSQL?

- Stands for Not Only SQL
- Class of non-relational data storage systems
- Usually do not require a fixed table schema nor do they use the concept of joins
- All NoSQL offerings relax one or more of the ACID properties

How did we get here?

- Explosion of social media sites (Facebook, Twitter) with large data needs
- Rise of cloud-based solutions such as Amazon S3 (simple storage solution)
- Just as moving to dynamically-typed languages (Ruby/Groovy), a shift to dynamically-typed data with frequent schema changes
- Open-source community

The Perfect Storm

- Large datasets, acceptance of alternatives, and dynamically-typed data has come together in a perfect storm
- Not a backlash/rebellion against RDBMS
- SQL is a rich query language that cannot be rivaled by the current list of NoSQL offerings

CAP Theorem

- Three properties of a system: consistency, availability and partitions
- You can have at most two of these three properties for any shared-data system
- To scale out, you have to partition: that leaves either consistency or availability to choose from
  - In almost all cases, you would choose availability over consistency

Availability

- Traditionally, thought of as the server/process available five 9's (99.999 %).
- However, for large node system, at almost any point in time there's a good chance that a node is either down or there is a network disruption among the nodes.

Consistency Model

- A consistency model determines rules for visibility and apparent order of updates.
- For example:
  - Row X is replicated on nodes M and N
  - Client A writes row X to node N
  - Some period of time t elapses.
  - Client B reads row X from node M
  - Does client B see the write from client A?
  - Consistency is a continuum with tradeoffs
  - For NoSQL, the answer would be: maybe
  - CAP Theorem states: Strict Consistency can't be achieved at the same time as availability and partition-tolerance.

Eventual Consistency

- When no updates occur for a long period of time, eventually all updates will propagate through the system and all the nodes will be consistent
- For a given accepted update and a given node, eventually either the update reaches the node or the node is removed from service
- Known as BASE (Basically Available, Soft state, Eventual consistency), as opposed to ACID

Common Advantages

- Cheap, easy to implement (open source)
- Data is replicated to multiple nodes (therefore identical and fault-tolerant) and can be partitioned
  - Down nodes easily replaced
  - No single point of failure
- Easy to distribute
- Don't require a schema
- Can scale up and down
- Relax the data consistency requirement (CAP)

What am I giving up?

- joins
- group by
- order by
- ACID transactions
- SQL as a sometimes frustrating but still powerful query language
- easy integration with other applications that support SQL

Typical NoSQL API

- Basic API access:
  - get(key) -- Extract the value given a key
  - put(key, value) -- Create or update the value given its key
  - delete(key) -- Remove the key and its associated value
  - execute(key, operation, parameters) -- Invoke an operation to the value (given its key) which is a special data structure (e.g. List, Set, Map .... etc).

What kinds of NoSQL

- NoSQL solutions fall into two major areas:
  - Key/Value or 'the big hash table'.
    - Bigtable
    - Dynamo
  - Schema-less which comes in multiple flavors, column-based, document-based or graph-based.
    - Cassandra (column-based)

Key/Value

Pros:

- very fast
- very scalable
- simple model
- able to distribute horizontally

Cons:

- many data structures (objects) can't be easily modeled as key value pairs

Schema-less

Pros:

- Schema-less data model is richer than key/value pairs
- eventual consistency
- many are distributed
- still provide excellent performance and scalability

Cons:

- typically no ACID transactions or joins

Google Bigtable

Bigtable: A Distributed Storage System for Structured Data

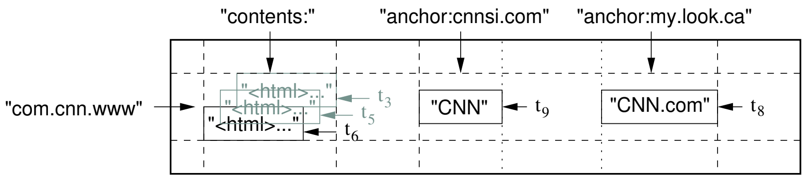
Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach
Mike Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber
{fay,jeff,sanjay,wilsonh,kerr,m3b,tushar,fikes,gruber}@google.com
Google, Inc.

BigTable Features

- Fault-tolerant, persistent
- Scalable
  - Thousands of servers
  - Terabytes of in-memory data
  - Petabytes 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

Data model

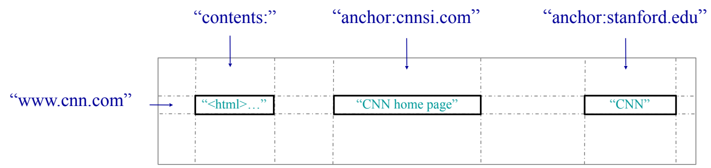
- Distributed multi-dimensional sparse map
  (row, column, timestamp) → cell contents



Rows

- Name is an arbitrary string (64KB)
  - Access to data in a row is **atomic**
  - Row creation is implicit upon storing data
- Rows ordered **lexicographically**
  - Rows close together lexicographically usually reside on one or a small number of machines
- Each row range is called a **tablet**

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

Column Families

- Must be created before data can be stored
- Small number of column families
- Unbounded number of columns

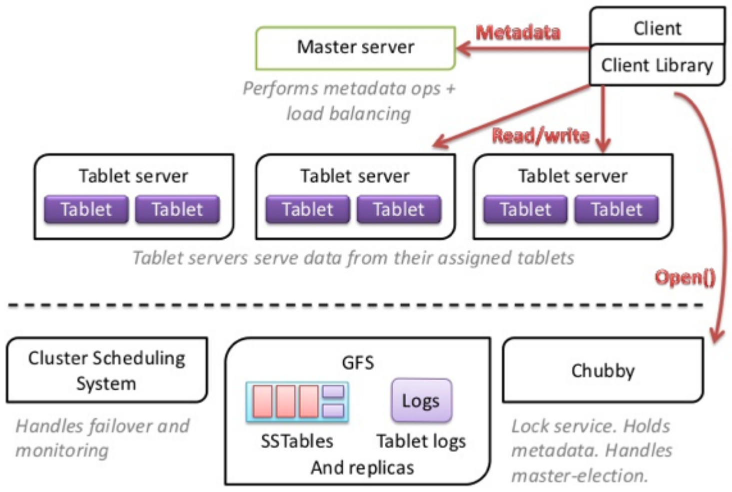
Timestamps

- Used to store different **versions** of data in a cell (64 bit)
  - New writes default to current time, but timestamps for writes can also be set explicitly by clients

Timestamps

- **Garbage Collection**
  - Per-column-family settings to tell Bigtable to GC
  - *“Only retain most recent K values in a cell”*
  - *“Keep values until they are older than K seconds”*

Architecture



Chubby

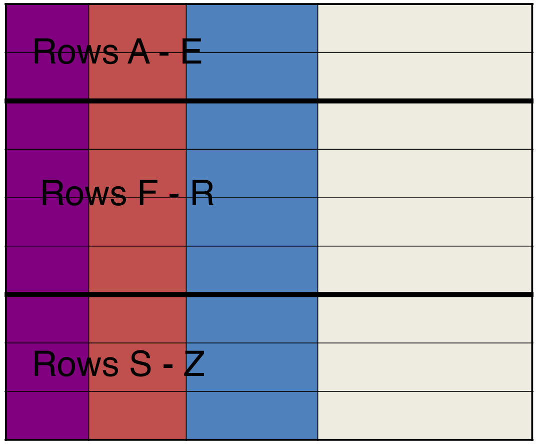
- Namespace that consists of directories and small files
  - Each directory or file can be used as **lock**
- Chubby client maintains session with Chubby service
  - Expires if unable to renew its session lease within expiration time
  - If expired, client loses any locks and open handles
- Atomic Reads / Writes

Tablets

- Large tables are broken into ***tablets*** at row boundaries
  - Tablet holds **contiguous** range of rows
  - Aim for ~100MB to 200MB of data per tablet
- **Tablet server** responsible for tablets
  - Fine-grained load balancing:
    - Migrate tablets away from overloaded machine
    - Master makes **load-balancing** decisions

Tablets

As table grows, split tables into tablets (100-200MB)

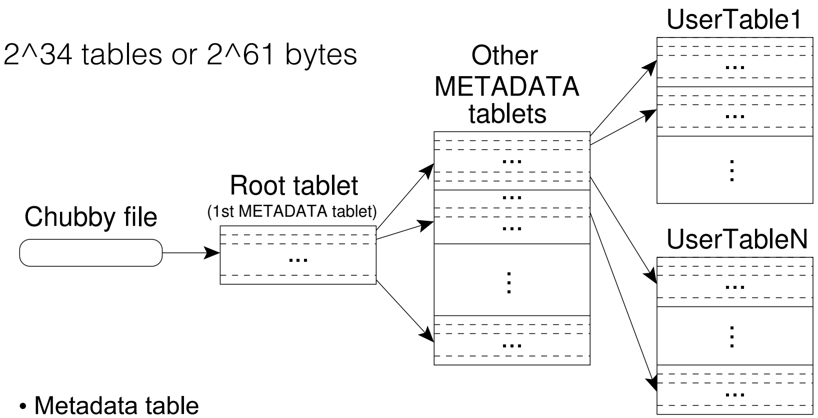


Tablet Server

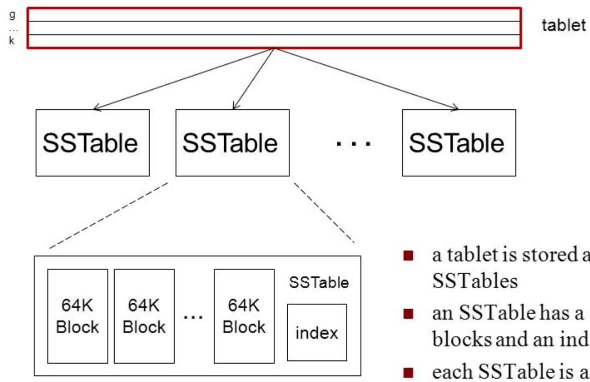
- Master assigns tablets to table servers
- **Tablet server**
  - Handles reads / writes requests to tablets
  - Splits large tablets
- Client does not move data through master

Finding a tablet

2^34 tables or 2^61 bytes



- Metadata table
  - includes log of all events pertaining to each tablet
  - **never splits**
- Client library caches tablet locations



- a tablet is stored as a set of SSTables
- an SSTable has a set of 64K blocks and an index
- each SSTable is a GFS file

SSTable

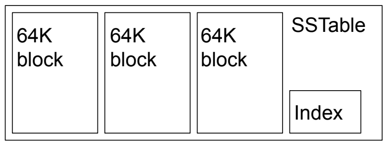
- File-format for storing files
- Key-Value Map
  - Persistent
  - Ordered
  - Immutable
  - Keys and values are strings

SSTable

- Operations
  - Look up value for key
  - Iterate over all key/value pairs in specified range
- Sequence of blocks (64 KB)
  - Block index used to locate blocks
- Block index
  - Binary search on in-memory index
  - Or, map complete SSTable into memory

SSTable

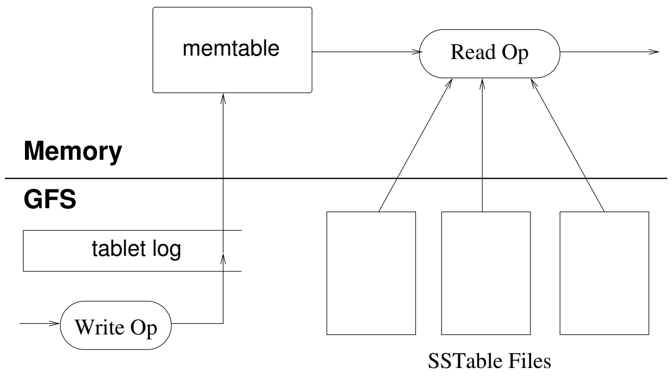
- Immutable, sorted file of key-value pairs
- Chunks of data plus an index
  - Index is of block ranges, not values
  - Index loaded into memory when SSTable is opened
  - Lookup is a single disk seek
- Alternatively, client can load SSTable into memory



SSTable

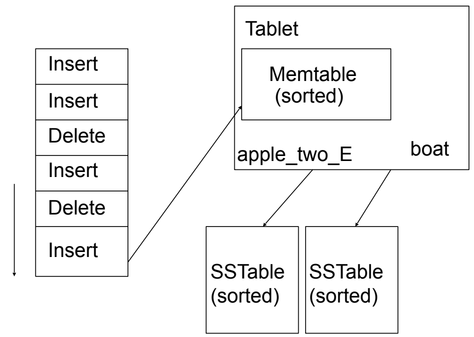
- Relies on lock service called Chubby
  - Ensure there is at most one active master
  - Store bootstrap location of Bigtable data
  - Finalize table server death
  - Store column family information
  - Store access control lists

Tablet representation



Editing/Reading a table

- Mutations are committed to a commit log (in GFS); then applied to an in-memory version (memtable)
  - For concurrency, each memtable row is copy-on-write
- Reads applied to merged view of SSTables & memtable
  - Reads & writes continue during tablet split or merge



Master Startup

- Grab unique **master** lock in Chubby
- Scan servers directory in Chubby to find live **servers**
- Communicate with every live **tablet** to discover which tablets are assigned
- Scan **METADATA** table to learn set of tablets
  - Track unassigned tablet

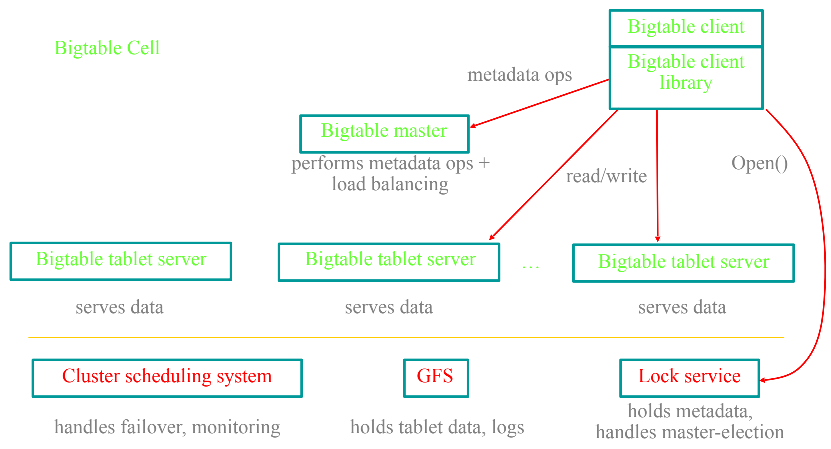
Tablet Assignment

- Master has list of unassigned tablets
- When a tablet is unassigned, and a tablet server has room for it, master sends tablet load request to tablet server

Tablet Serving

- Persistent state of tablet is stored in GFS
- Updates committed to log that stores redo records
  - *Memtable*: sorted buffer in memory of recent commits
  - Older updates stored in SSTable

System Structure



API

```
// 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);
```

API

```
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());
}
```

Google: The Big Picture

- *Custom solutions for unique problems!*
- GFS: Stores data reliably
  - But just raw files
- BigTable: gives us key/value map
  - Database like, but doesn't provide everything we need
  - Chubby: locking mechanism
  - SSTable file format
- MapReduce: lets us process data from BigTable (and other sources)

Common Principles

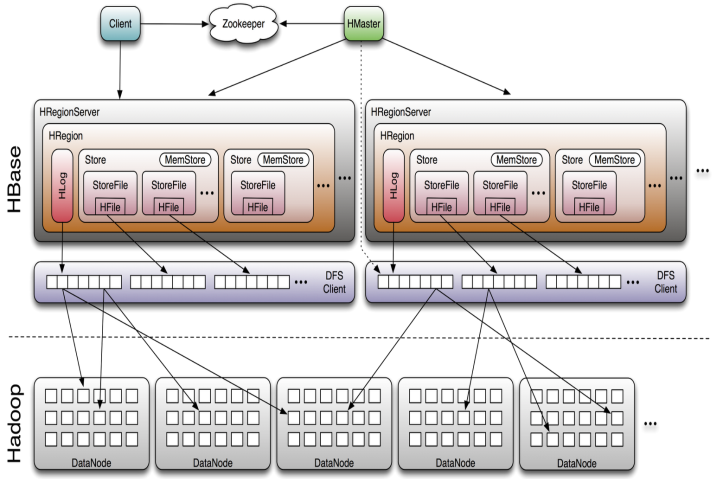
- One master, multiple helpers
  - MapReduce: master coordinates work amongst map / reduce workers
  - Bigtable: master knows about location of tablet servers
  - GFS: master coordinates data across chunkservers
- Issues with a single master
  - What about master failure?
  - How do you avoid bottlenecks?

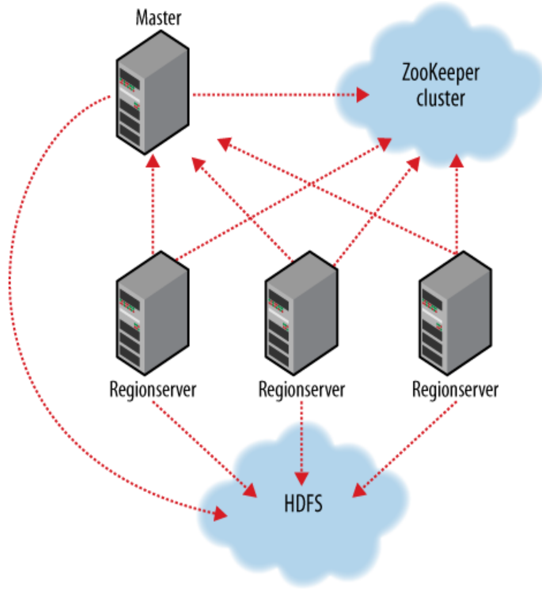
Difference between MapReduce and BigTable?

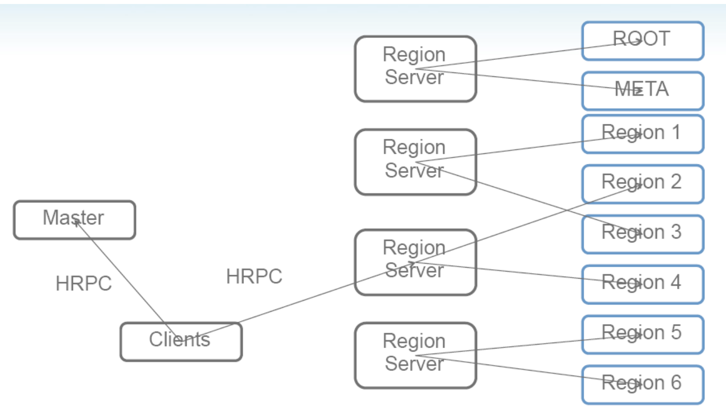
HBase

- HBase is a Bigtable clone.
- It is open source
- It has a good community and promise for the future
- It is developed on top of and has good integration for the Hadoop platform, if you are using Hadoop already.
- It has a Cascading connector.

HBase Architecture







codes

```
$ hbase shell
> create 'test', 'data'
0 row(s) in 4.3066 seconds
> list
test
1 row(s) in 0.1485 seconds
> put 'test', 'row1', 'data:1', 'value1'
0 row(s) in 0.0454 seconds
> put 'test', 'row2', 'data:2', 'value2'
0 row(s) in 0.0035 seconds
> put 'test', 'row3', 'data:3', 'value3'
0 row(s) in 0.0090 seconds
```

```
> scan 'test'
ROW COLUMN+CELL
row1 column=data:1, timestamp=1240148026198,
        value=value1
row2 column=data:2, timestamp=1240148040035,
        value=value2
row3 column=data:3, timestamp=1240148047497,
        value=value3
3 row(s) in 0.0825 seconds
> disable 'test'
09/04/19 06:40:13 INFO client.HBaseAdmin: Disabled
        test
0 row(s) in 6.0426 seconds
> drop 'test'
09/04/19 06:40:17 INFO client.HBaseAdmin: Deleted
        test
0 row(s) in 0.0210 seconds
> list
0 row(s) in 2.0645 seconds
```

Connection to HBase

Java client

get(byte [] row, byte [] column, long timestamp, int versions);

Non-Java clients

Thrift server hosting HBase client instance

Sample ruby, c++, & java (via thrift) clients

REST server hosts HBase client

TableInput/OutputFormat for MapReduce

HBase as MR source or sink

HBase Shell

JRuby IRB with "DSL" to add get, scan, and admin

./bin/hbase shell YOUR_SCRIPT

SQL vs. NoSQL

- https://www.youtube.com/watch?v=rRoy6I4gKWU

Bigtable vs. HBase

- https://www.youtube.com/watch?v=INSsFCh4wmk