# CS-E4640 - Big Data Platforms
# Lecture 8

Keijo Heljanko

Department of Computer Science
University of Helsinki & Aalto University
keijo.heljanko@helsinki.fi

14.11-2018

# BigTable

- Described in the paper: “Fay Chang, Jeffrey Dean, Sanjay Ghemawat, Wilson C. Hsieh, Deborah A. Wallach, Michael Burrows, Tushar Chandra, Andrew Fikes, Robert E. Gruber: Bigtable: A Distributed Storage System for Structured Data. ACM Trans. Comput. Syst. 26(2): (2008)”
  - A highly scalable consistent and partition tolerant datastore
  - Implemented on top of the Google Filesystem (GFS)
  - GFS provides data persistence by replication but only supports sequential writes
  - BigTable design does no random writes, only sequential writes are used
- Open source clone: Apache HBase

# BigTable

- Recall that sequential writes are much faster than random writes
- BigTable is a write optimized design: Writes are optimized over reads
- The random reads that miss the DRAM cache used are slower than in traditional RDBMS
- The design also minimizes the number of bytes written by using efficient logging and compression of data. Also a good design principle for Flash SSD use

# BigTable Data Model

- Bigtable paper describes itself as: ”. . . a sparse, distributed, persistent multi-dimensional sorted map”
- Namely, BigTable stores data as strings, which can be accessed by three coordinates:
  - `row` - an arbitrary string row key (10-100 bytes typical, max 64 KB)
  - `column` - a column key, consisting of a column family and column qualifier (name) in syntax `family:qualifier`
  - `timestamp` - a 64 bit integer that can be used to store a timestamp
  - Thus we have a map with three coordinates: `(row:string, column:string, time:int64) -> string`

# BigTable Data Layout

- In BigTable data is stored sorted by the row key, and the data is automatically sharded (split to different servers) in large blocks sorted by the row key, allowing for efficient scans in increasing alphabetical row key order
- Columns are grouped in “Column families”, and all columns in a single column family are stored together in compressed form on disk
- Some queries might not access columns in all column families, and this allows data in these column families to be easily skipped for such queries
- The timestamp field allows applications to store several copies of the same data together, e.g., Website content changes over time

# Example: Web Table

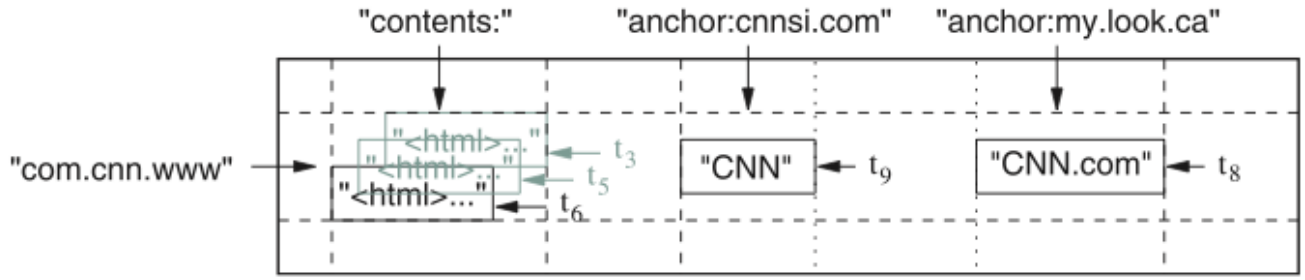



Fig. 1. A slice of an example table that stores Web pages. The row name is a reversed URL. The contents column family contains the page contents, and the anchor column family contains the text of any anchors that reference the page. CNN's home page is referenced by both the Sports Illustrated and the 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 $t_6$.

# Example: Web Table

- Notice how row key is the reverse of the Web page URL to group pages from the same site together
- The “contents” column family stores the Web site contents at the time pointed by the timestamp
- The “anchor” column family stores the links pointing to the Web page in question
- Now it is easy to skip for example reading the Web page contents if we are only interested in links between Web pages

# BigTable Atomicity Guarantees

- Each row in BigTable can have a different number of columns
- Some rows might have thousands or even more columns (See column family “anchor” in the Web Table example, storing a column for each incoming link to a Web page)
- BigTable atomicity guarantee: Updates to a single row are atomic, irregardless of the number of column families or columns being updated
- No transactions for updating multiple rows atomically are available

# Tablets, SSTables, Chubby

- Rows with consecutive row keys are grouped together to “Tablets”, which are the unit of distribution and load balancing. Tablets are by default split when they grow beyond 1 GB in size
- A immutable datastructure called SSTable is used to store BigTable data files
- The Google paper states: “An SSTable provides a persistent, ordered immutable map from keys to values, where both keys and values are arbitrary byte strings.”
- Inside SSTable are compressed 64KB data blocks combined with an index to find a row key inside these data blocks.
- Compression is heavily used by BigTable
- Also an optional Bloom filter can be added to speed access to non-existent row/column pairs

# Chubby

- BigTable relies on a fault tolerant distributed coordination system (called lock service by Google) called Chubby for coordination between servers
- Basically Chubby is the piece of infrastructure other Google services use to implement common tasks such as leader (master) election and for storing the set of servers a BigTable instance consists of for clients
- The list of BigTable uses for Chubby is extensive: "...to ensure that there is at most one active master at any time; to store the bootstrap location of Bigtable data; to discover tablet servers and finalize tablet server deaths; and to store Bigtable schemas"
- Basically, if Chubby service is down, all other Google services will eventually go down as well

# BigTable Tablet Lookup

- The tablet Lookup in BigTable starts by asking Chubby for the location of (server responsible for) the root Tablet
- An additional layer of indexes pointed by the root Tablet will point to the (servers holding the) user data Tablets

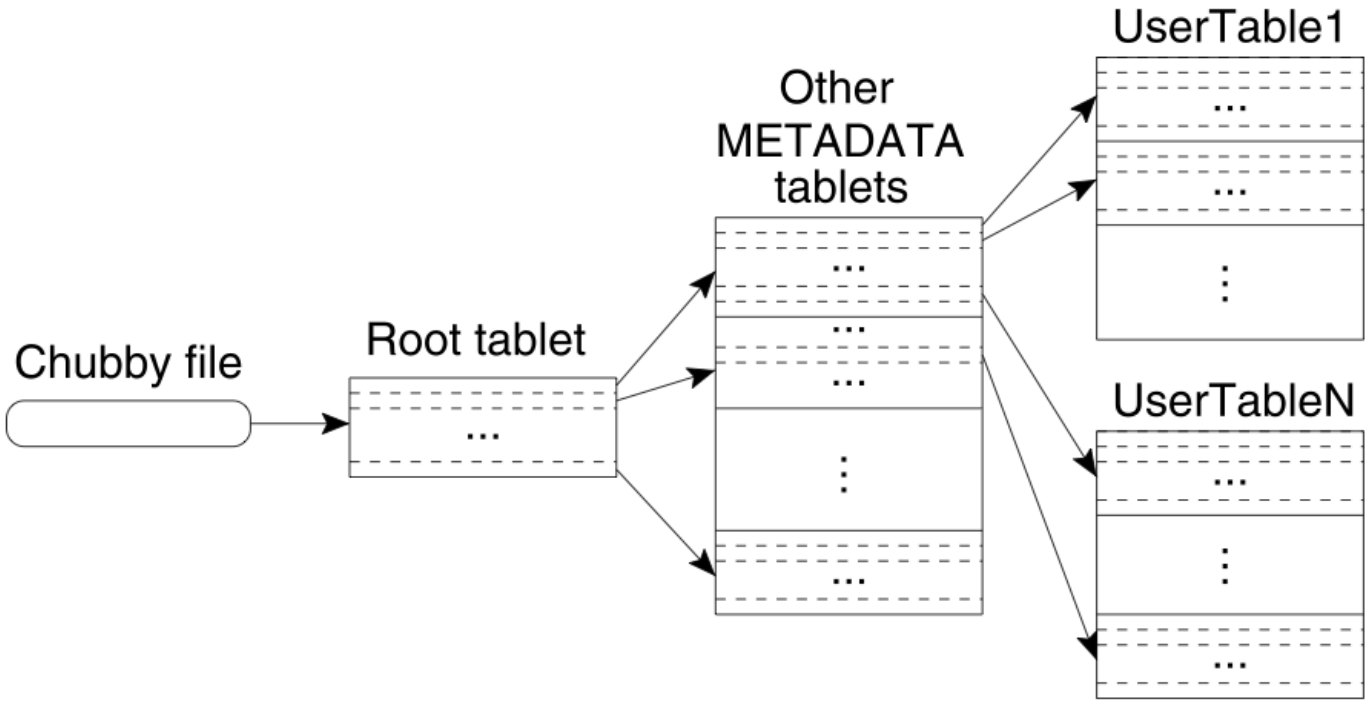



Fig. 5. Tablet location hierarchy.

# BigTable Write Path

- When BigTable Tablet server receives a write, it first commits it to a commit log on GFS
- After the commit, write is added to memtable
- Once memtable fills up, all written data is sorted in memory, and stored into an SSTable on GFS, and the commit log file can be removed

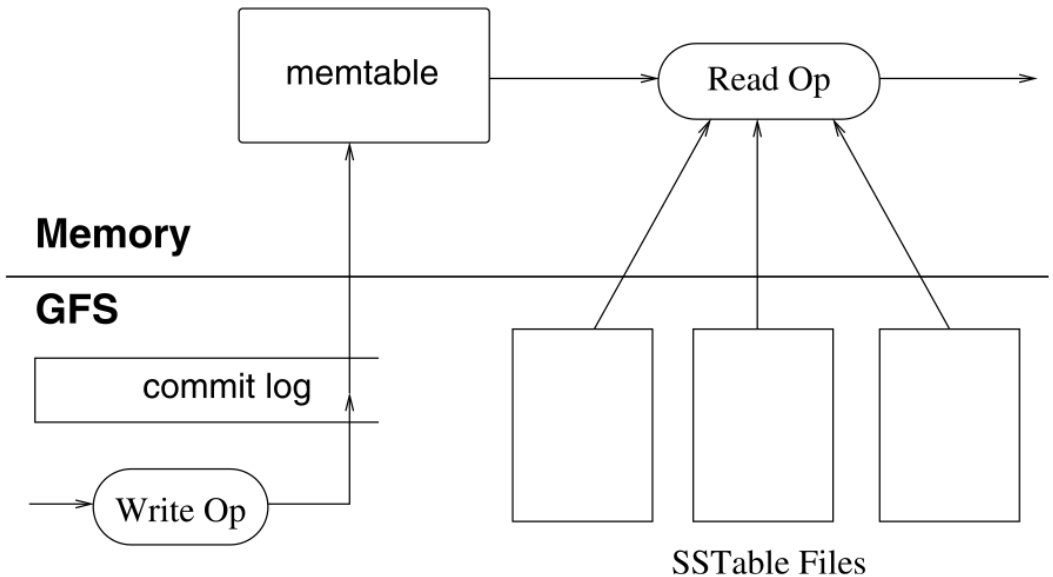



Fig. 6. Tablet representation.

# BigTable Read Path

- When BigTable Tablet server receives a read, it first checks the memtable if the data exists there
- If not, then the SSTables are scanned one at a time from newest to oldest to find the SSTable holding the latest version of the data item
- Bloom filters can help avoid scanning some SSTables

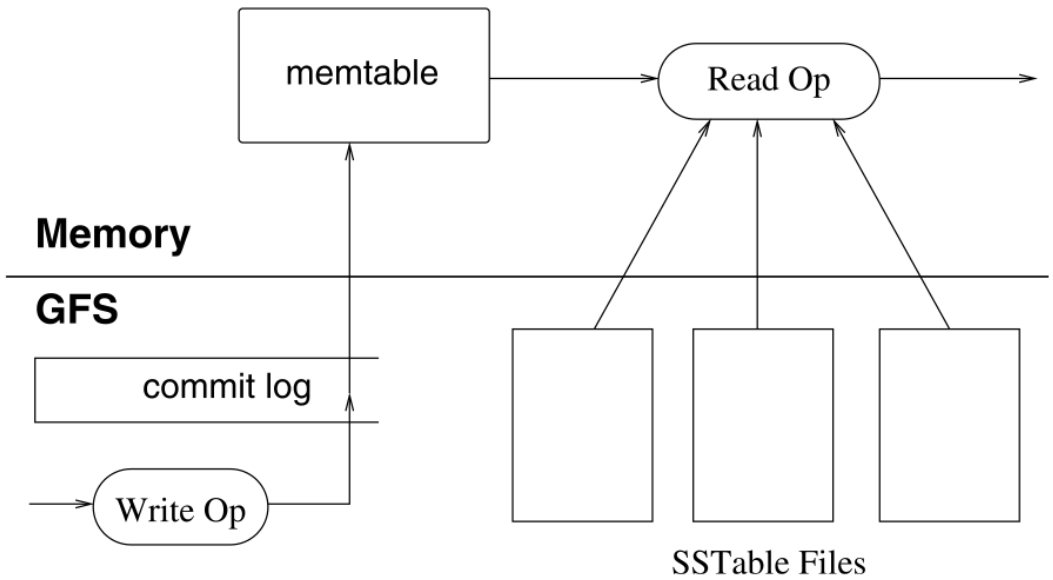



Fig. 6. Tablet representation.

# Compactions

- If data is continuously written, the number of SSTables grows, and read performance will degrade over time
- To keep the number of SSTables under control, a “compaction” is run on the background, which merges several SSTables into a single SSTable using a merge sort like algorithm
- An item is marked as deleted by having special “tombstone” records, that mark that the item has been deleted
- In a “major compaction” all SSTables are merged into a one file, only at which point tombstone records can be removed by removing the relevant data items

# Compactions

- The compaction idea is motivated by the paper: “Patrick E. O’Neil, Edward Cheng, Dieter Gawlick, Elizabeth J. O’Neil: The Log-Structured Merge-Tree (LSM-Tree). Acta Inf. 33(4): 351-385 (1996)”
- Compactions only use sequential reads and writes, they are also the only option on a WORM filesystem such as GFS
- They can be scheduled at periods of low activity
- A large number of deletes can be bad for BigTable performance

# BigTable Performance Numbers

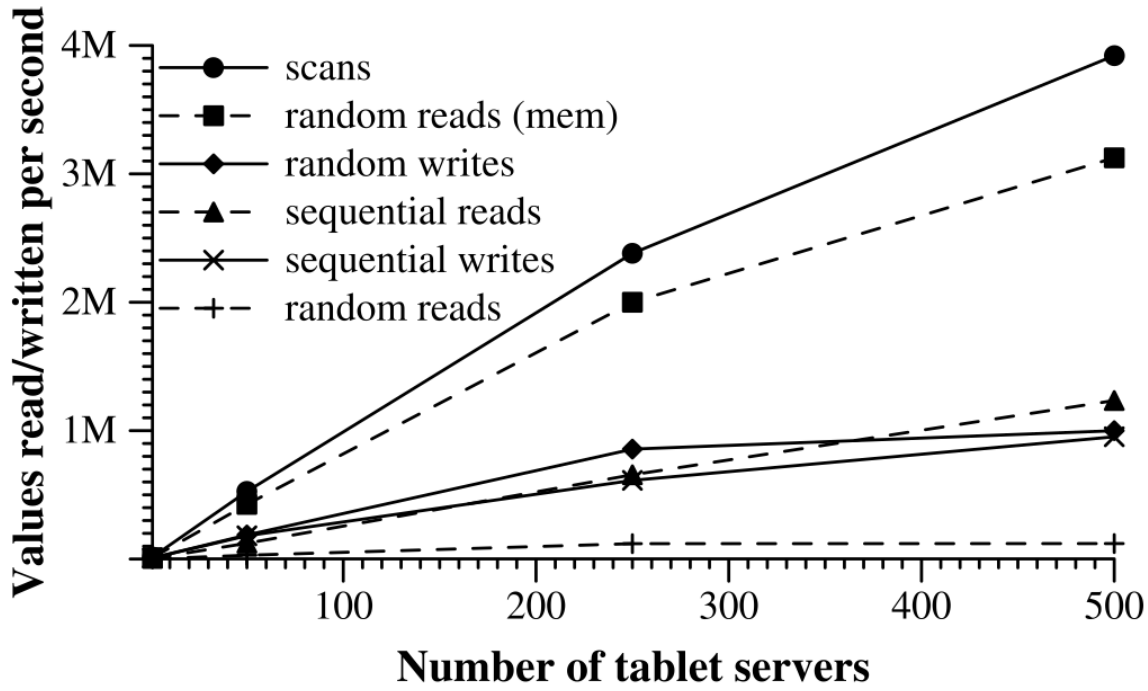
- Notice how BigTable random read performance from disk is an order of magnitude smaller at large scale than random write performance

Table I. Number of 1000-byte values read/written per second. The values are the rate per tablet server.

| Experiment | # of Tablet Servers | | | |
|---|---|---|---|---|
| | 1 | 50 | 250 | 500 |
| random reads | 1212 | 593 | 479 | 241 |
| random reads (mem) | 10811 | 8511 | 8000 | 6250 |
| random writes | 8850 | 3745 | 3425 | 2000 |
| sequential reads | 4425 | 2463 | 2625 | 2469 |
| sequential writes | 8547 | 3623 | 2451 | 1905 |
| scans | 15385 | 10526 | 9524 | 7843 |

# BigTable Performance Plots

- Notice how BigTable table scans are much faster compared to single row accesses
- Also notice how much faster random reads from DRAM are compared to random reads from disk



# BigTable Conclusions

- BigTable is a scalable write optimized database design
- It uses only sequential writes for improved write performance
- For read intensive workloads BigTable can be a good fit if most of the working set fits into DRAM
- For read intensive working sets much larger than DRAM, traditional RDBMS systems are still a good match
- BigTable also has an impressive data scan speed, so scan based workloads (getting MapReduce input data from BigTable) are a good match for its performance
- Some optimizations such as agressive compression and use of Bloom filters are only viable because SSTables are immutable data objects

# Apache HBase

- Apache HBase is an open source Google BigTable clone
- It very closely follows the BigTable design but has the following differences
  - Instead of GFS, HBase runs on top of HDFS
  - Instead of Chubby, HBase uses Apache Zookeeper
  - SSTable of BigTable are called in HBase HFile (and HFile V2)
  - HBase only partially supports fully memory mapped data

# Facebook Messaging on HBase

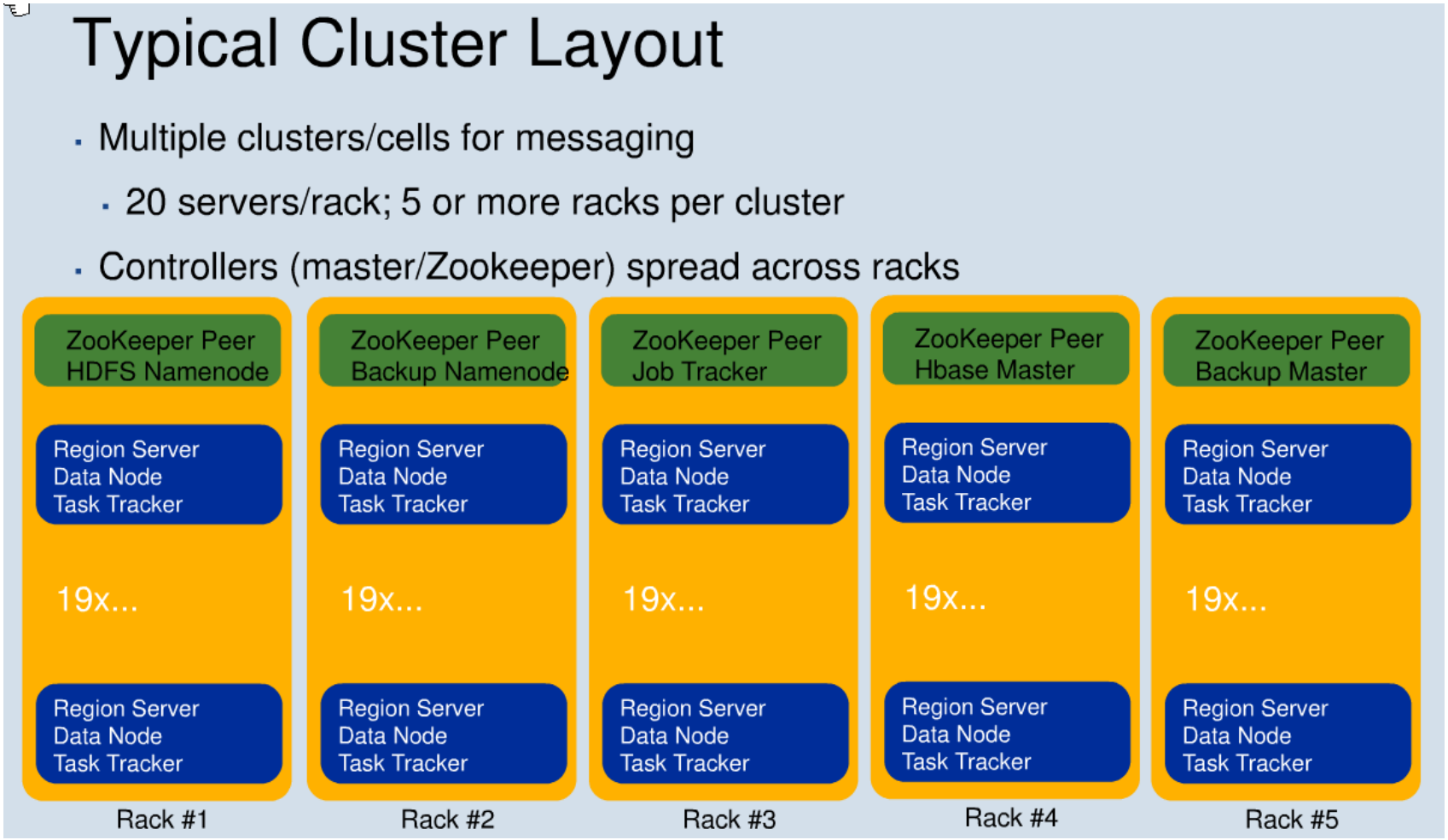
- In the paper “Dhruba Borthakur, Jonathan Gray, Joydeep Sen Sarma, Kannan Muthukkaruppan, Nicolas Spiegelberg, Hairong Kuang, Karthik Ranganathan, Dmytro Molkov, Aravind Menon, Samuel Rash, Rodrigo Schmidt, Amitanand S. Aiyer: Apache Hadoop goes realtime at Facebook. SIGMOD Conference 2011: 1071-1080” authors from Facebook describe the infrastructure behind Facebook messaging
  - 500 million users, several Petabytes of data
  - 135 000 000 000 messages per month

# Reasons for selecting HBase

- Elasticity: Ability to add new nodes as storage as needed
- High write throughput
- Consistency guarantees given within a single datacenter
- Comparable (but not better) read performance compared to RDBMS
- High Availability and disaster recovery features
- Fault isolation of e.g., hard disk failures (a hard disk fails every 30 mins)
- Atomic read-modify-write primitives (e.g., for atomic counters)
- Efficient range scans: e.g., give last 100 messages of the user

# Typical 100 node HBase Deployment

From presentation: “Nicolas Spiegelber: Facebook messages & HBase”:



# Challenges for HBase Solution

- Each user is served by a single cluster but Facebook has quite a few of 100 node HBase clusters
- HDFS was not designed for realtime but batch processing - large timeouts etc.
- HDFS NameNode single point of failure was addressed by “AvatarNode” - a new hot backup NameNode solution

# Data Storage Engines with BigTable Techniques

- LevelDB (`http://leveldb.org/`) is a light weight DB engine by the BigTable authors with similar tradeoffs, used e.g., inside Google Chrome
- RocksDB (`http://rocksdb.org/`) a key-value store further developed from LevelDB
- CockroachDB (`http://cockroachdb.org/`) a (still in early development) distributed database on top of RocksDB
- Also Cassandra (`http://cassandra.apache.org/`) uses log structured merge trees to implement its storage layer