NoSQL & BigTable

Rong Chen

NoSQL related slides adapted from Adam Marcus marcua@csail.mit.edu

Bigtable slides taken from Richard Venutolo

Outline

The Rise of NoSQL

BigTable

Data-driven Age

Humanity and Social Science

Information Science

Entertainment

Business

Medicine

- CT、MRI、

NoSQL: Not only SQL

SQL

- Internet-Scale data ✗
- High Concurrency ✗
- Highly Scalable ✗

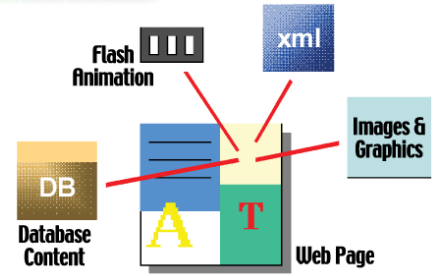
Optimization to RDBMS

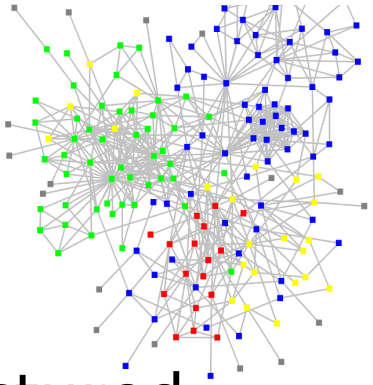
- Master-Slave
- Partition/Sharding
- Multi-Master Replication
- INSERT only, not UPDATE/DELETE
- No JOINs
- In-Memory Database

Not only SQL

Not only SQL

- High Volume
- Semi/UN-Structured
- High Concurrency
- High Scalability





Over 90% data in the world in Unstructured
Such : Pictures、Video、Audio ……

(From IDC)

NoSQL Pricinple

CAP Theory (Eric Brewer)

CAP：Consistency, Availability, Partitions

NoSQL DB can only choose two

High-Scalable NoSQL DB

Must Support Partition

Choose 1 from Consistency and Availability

Mostly choose Availability (for fault tolerance)

Rules of Thumb

Almost always, use PostgreSQL or MySQL

Most problems can be solved by a single (large) machine

Consider paying a DB vendor to solve problem

But Sometimes, Need >1 Machines

Analytics on MapReduce or Column Stores

Facebook famously stores 1B+ users on ~10K machines

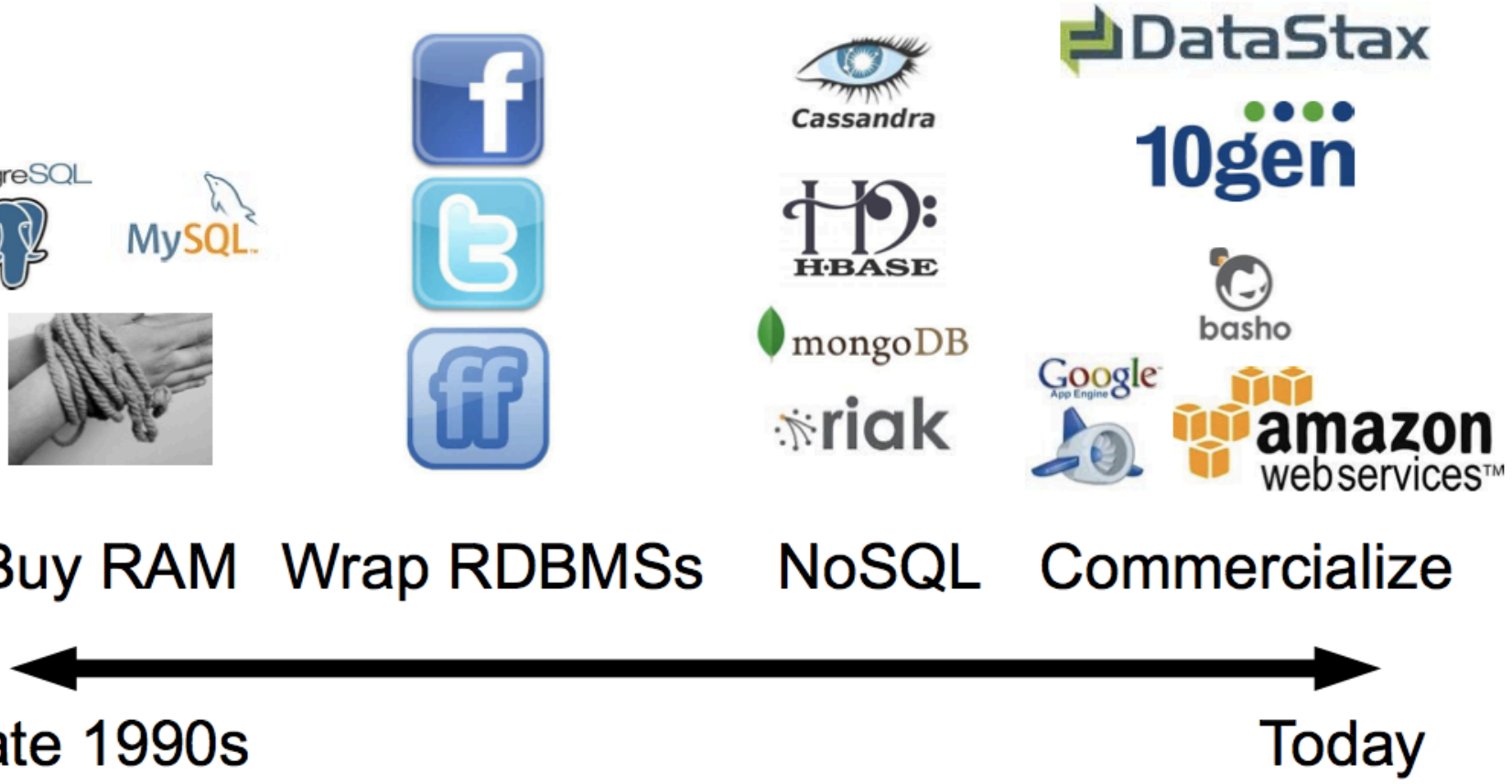
Scaling Transactional Workloads

Google spends thousands of person-hours on Megastore/Spanner

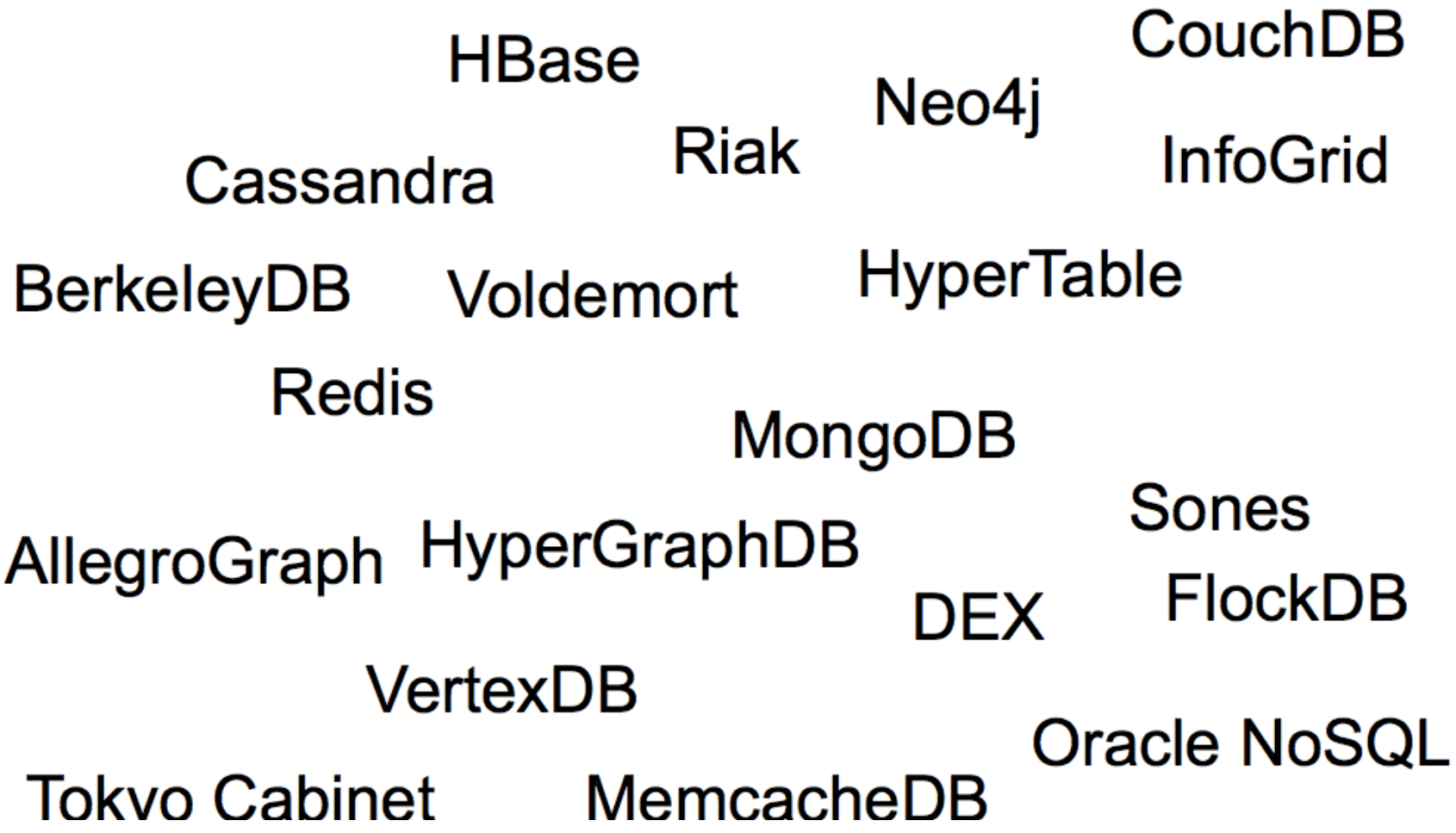
Everyone else jumps through hoops to stick to "relational model" and "SQL"

- Partition data to avoid multi-node transactions (2PC)
- Avoid multi-row transactions to prevent locking

(Compressed) Histories of NoSQL



Incomplete List of NoSQL



BigTable

Google, 2006

Column store with sloppy schemas

Strong consistency

Open source: HBase

Dynamo

Amazon, 2007

Key-value store

Allows eventual consistency

Open source: Voldemort, Riak

Standards-compliant SQL Systems

Relational model

Powerful query language

Transactional semantics

Predefined schemas

Strong consistency between replicas

NoSQL: Maybe you don't need all of these?

NoSQL Systems are a Buffet (of progressively larger grenades)

Data model

Query model

Durability

Transactional consistency

Partitioning

Replica consistency

Data Model

Usually key-based...

- Binary blob: Voldemort (used in LinkedIn)
- Documents: MongoDB, CouchDB, Riak
- Data structures: Redis
- Column-families: Bigtable, HBase, Cassandra

Query Model

Redis: data structure-specific operations

CouchDB, Riak: MapReduce

Cassandra, MongoDB: SQL-like languages, no joins or transactions

Third-party

- High-level: PigLatin, HiveQL
- Library: Cascading, Crunch Streaming: Flume, Kafka, S4, Scribe

Transactions

Full ACID for single key

Redis: multi-key single-node transactions

Single-server durability

Memory only: memcached

Single-server durability: the rest

- fsync every N seconds: most
- Write-ahead logging: Cassandra, HBase, Redis, Riak
- Group commit: Cassandra, HBase/HDFS

When one server is not enough

- Replicate
  - Performance
  - K-safety

} Consistency/Availability

- Partition
  - Vertical
  - Horizontal

} Partitioning Scheme

Eventual vs. Strong Consistency: FIGHT!

N = # replicas
W = # write acknowledgements
R = # read acknowledgements

Strong: Appears that all replicas see all writes
Eventual: If left alone, replicas eventually converge
Weak: Replicas have different, divergent versions

The choice the consistency model depends on applications

Consistency wild west

Available, inconsistent on failure
N = 3, W = 1, R = 1

Consistent, unavailable writes on failure
N = 3, W = 3, R = 1

Available, consistent with 1 failure
N = 3, W = 2, R = 2

Consistency math

Strong: $N < R + W$

Easy: $W = N$

Tricky: $W < N$

Eventual/Weak: $N >= R + W$

Consistency Options

BigTable (HBase): Strong

Dynamo (Voldemort/Riak/Cassandra): tunable strong or eventual

...and many others (Yahoo! PNUTs has timeline consistency)

Partitioning

Consistent hashing: Voldemort, Riak

Range partitioning: HBase, MongoDB

Both: Cassandra

Your partitioning scheme matters

- Ideally, all joins happen on one machine
- Example: map all GMail for a user to one machine

NoSQL Use-cases

Cassandra at Netflix

HBase at Facebook

MongoDB at Craigslist

Cassandra

BigTable data model: key → column family

Dynamo sharding model: consistent hashing

Eventual or strong consistency

At netflix

- Transitioned from Oracle
- Store customer profiles, customer:movie watch log, and detailed usage logging
- In-datacenter: 3 replicas, per-app consistency

Cassandra at Netflix

Benefit: async inter-datacenter replication

Benefit: no downtime for schema changes

Benefit: hooks for live backups

HBase

Data model: key → column family

Sharding model: range partitioning

Strong consistency

Applications

- Logging events/crawls, storing analytics

- Twitter: replicate data from MySQL
- Hadoop analytics
- Facebook Messages

HBase for Facebook Messages

Cassandra/Dynamo eventual consistency was difficult to program against

Benefit: simple consistency model

Benefit: flexible data model

Benefit: simple partitioning, load balancing, replication

When you see “NoSQL,” think “tradeoffs”

Data model

Query Model

Transactions

Consistency vs. Availability Partitioning schemes

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 @ Google

Overview

BigTable is a distributed storage system for managing structured data.

Designed to scale to a very large size

- Petabytes of data across thousands of servers

Used for many Google projects

- Web indexing, Personalized Search, Google Earth, Google Analytics, Google Finance, …

Flexible, high-performance solution for all of Google's products

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 or 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

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
- 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

Building Blocks

Building blocks:

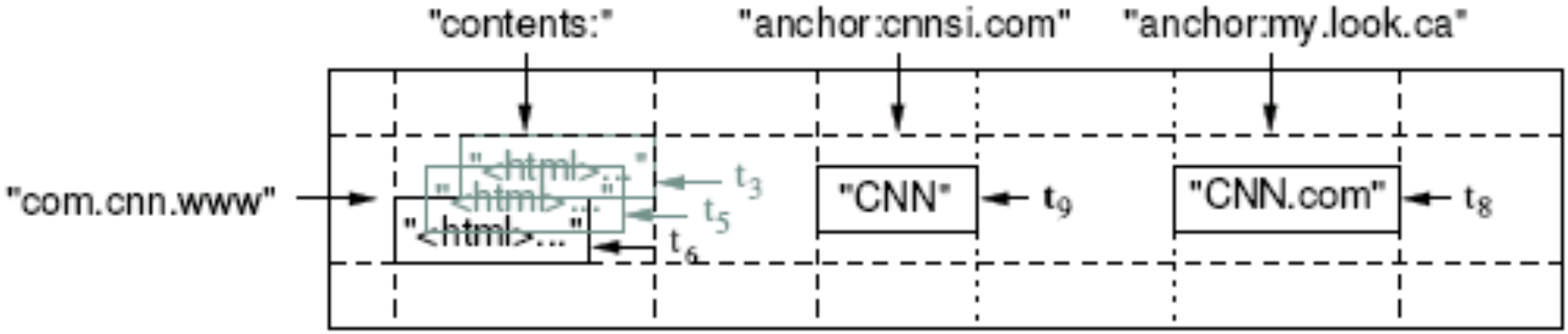
- Google File System (GFS): Raw storage
- Scheduler: schedules jobs onto machines
- Lock service: distributed lock manager
- MapReduce: simplified large-scale data processing

BigTable uses of building blocks:

- GFS: stores persistent data (SSTable file format for storage of data)
- Scheduler: schedules jobs involved in BigTable serving
- Lock service: master election, location bootstrapping
- Map Reduce: often used to read/write BigTable data

Data model: a big map

- <Row, Column, Timestamp> triple for key
  - Each value is an uninterpreted array of bytes
- Arbitrary “columns” on a row-by-row basis
  - Column family:qualifier. Family is heavyweight, qualifier lightweight
  - Column-oriented physical store- rows are sparse!
- Lookup, insert, delete API
  - Each read or write of data under a single row key is atomic



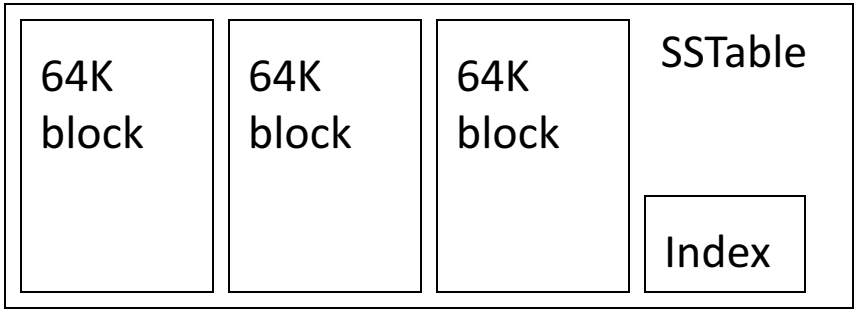
SSTable

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 mem

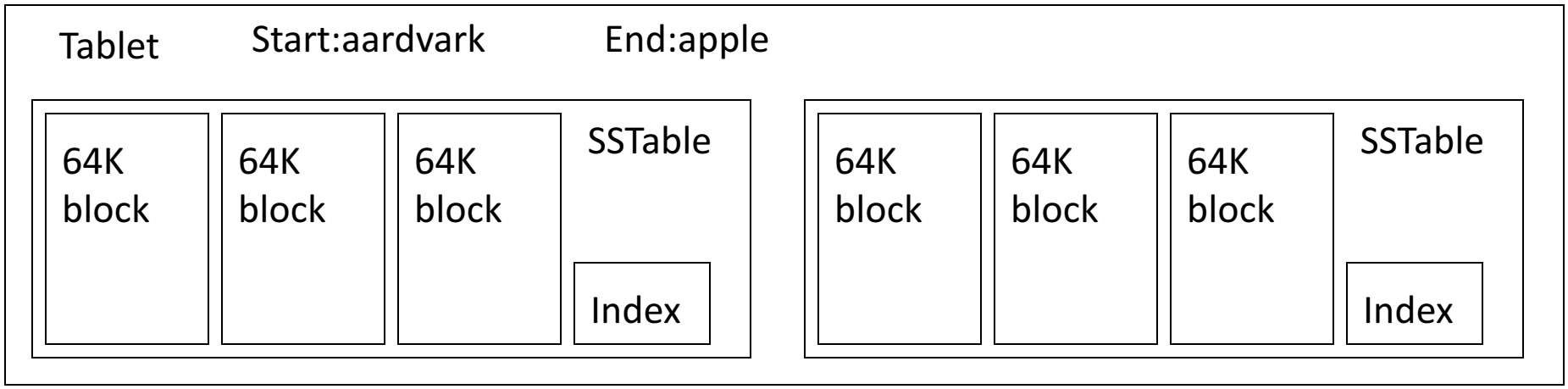


Tablet

Contains some range of rows of the table

Unit of distribution & load balance

Built out of multiple SSTables

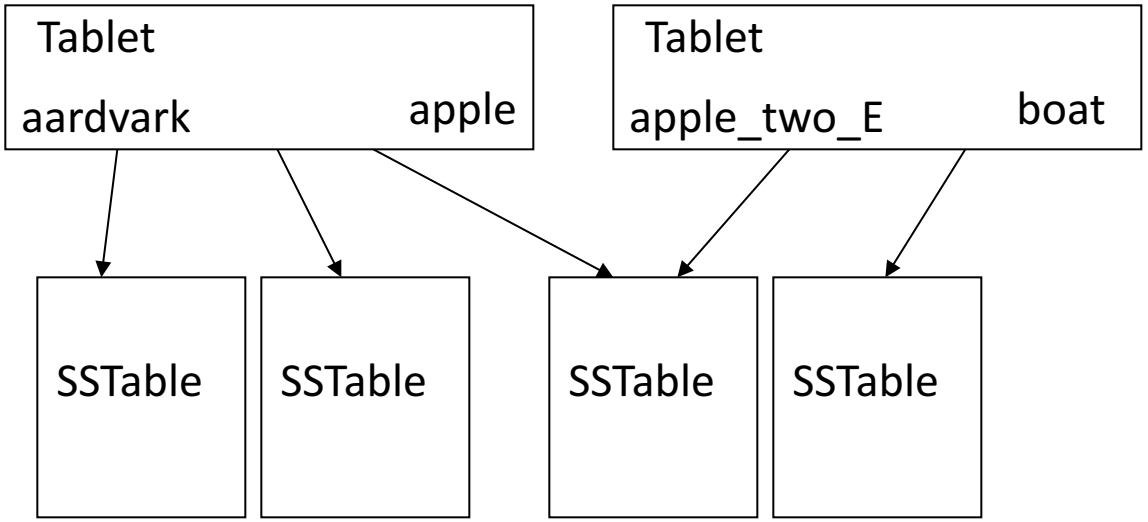


Table

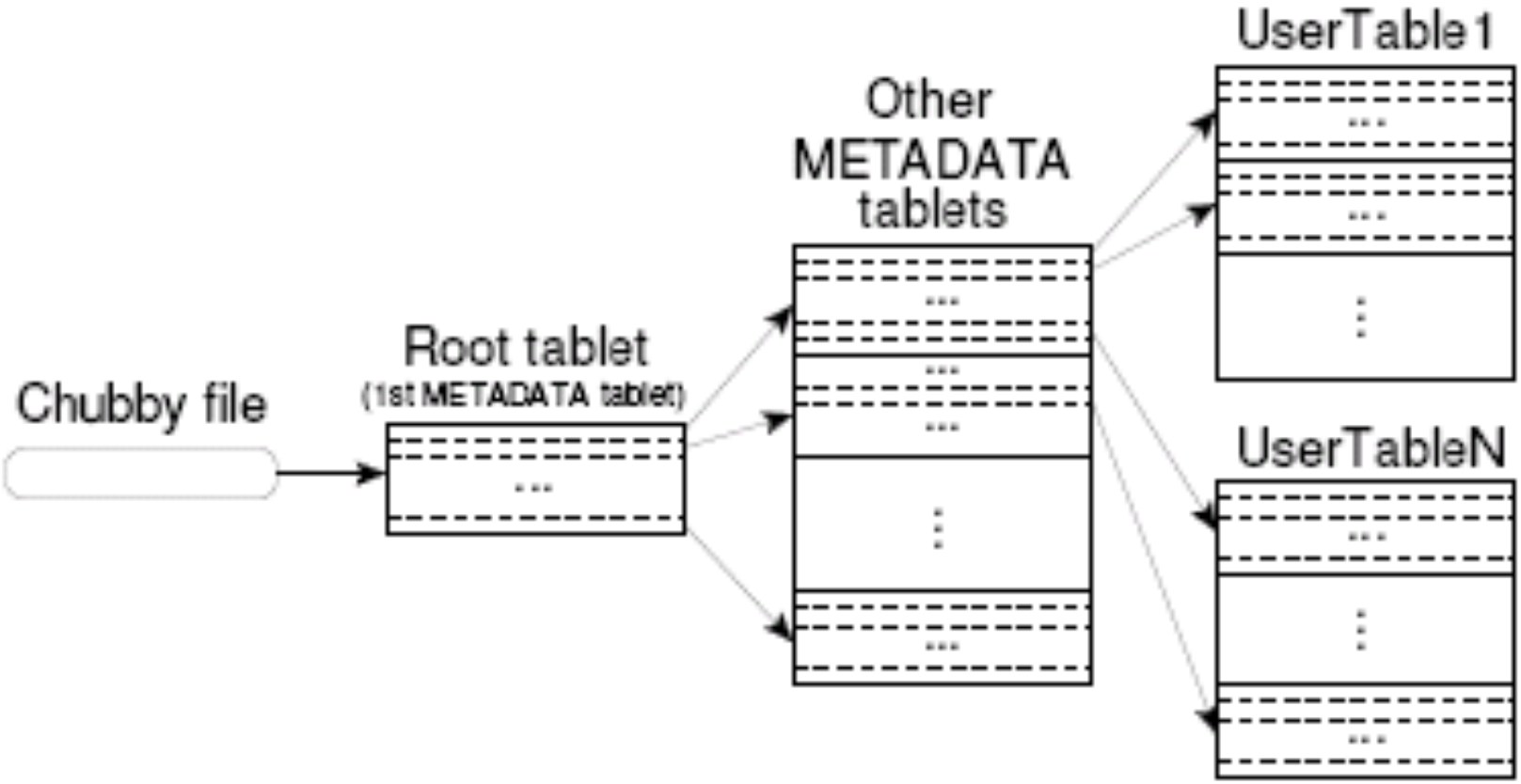
Multiple tablets make up the table

SSTables can be shared

Tablets do not overlap, SSTables can overlap



Finding a tablet



Client library caches tablet locations

Metadata table includes log of all events pertaining to each tablet

Servers

Tablet servers manage tablets, multiple tablets per server
Each tablet is 100-200 MBs

- Each tablet lives at only one server
- Tablet server splits tablets that get too big

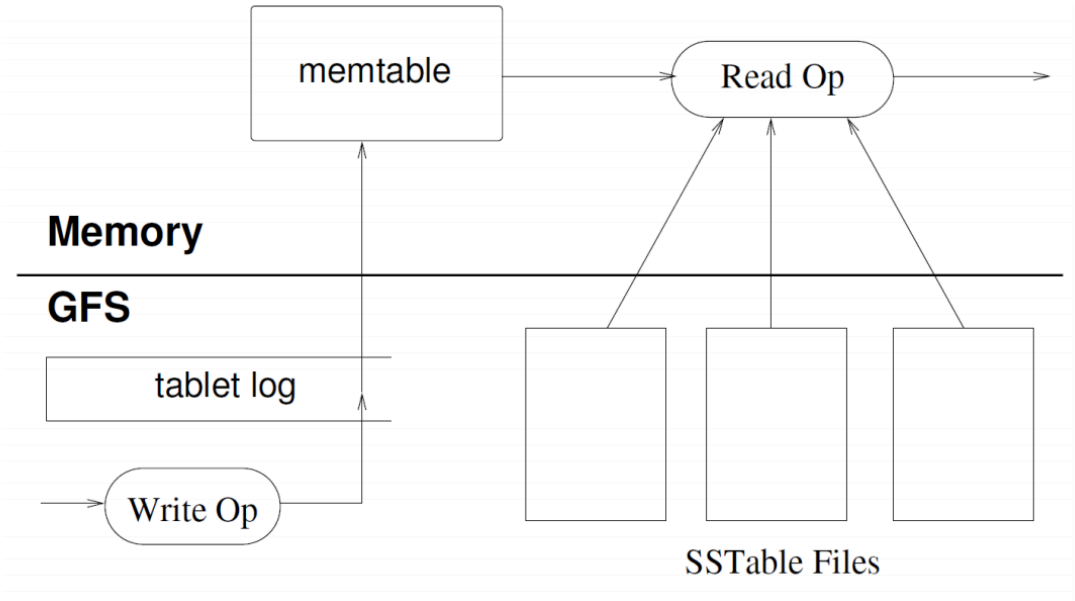
Master responsible for load balancing and fault tolerance

- Use Chubby to monitor health of tablet servers, restart failed servers
- GFS replicates data. Prefer to start tablet server on same machine that the data is already at

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



Compactions

- Minor compaction (convert the memtable into an SSTable)
  - Shrink the memory usage of the tablet server
  - Reduce the commit log
- Merging compaction
  - Merges several SSTables.
  - Reduce number of SSTables
- Major compaction
  - Merging compaction that results in only one SSTable.

Refinements: Locality Groups

Can group multiple column families into a *locality group*

- Separate SSTable is created for each locality group in each tablet.

Segregating columns families that are not typically accessed together enables more efficient reads

- In WebTable, page metadata can be in one group and contents of the page in another group.

Refinements: 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

Two-pass custom compressions scheme

- First pass: compress long common strings across a large window
- Second pass: look for repetitions in small window

Good space reduction (10-to-1)

Refinements: Bloom Filters

Read operation has to read from disk when desired SSTable isn't in memory

Reduce number of accesses by specifying a Bloom filter.

- Allows us ask if an SSTable might contain data for a specified row/column pair.
- Small amount of memory for Bloom filters drastically reduces the number of disk seeks for read operations
- Use implies that most lookups for non-existent rows or columns do not need to touch disk

Microbenchmarks: Throughput

1786 machines, with two dual-core Operton 2 GHz chips, large physical mem, two 400 GB IDE hard drives each, Gb Ethernet LAN

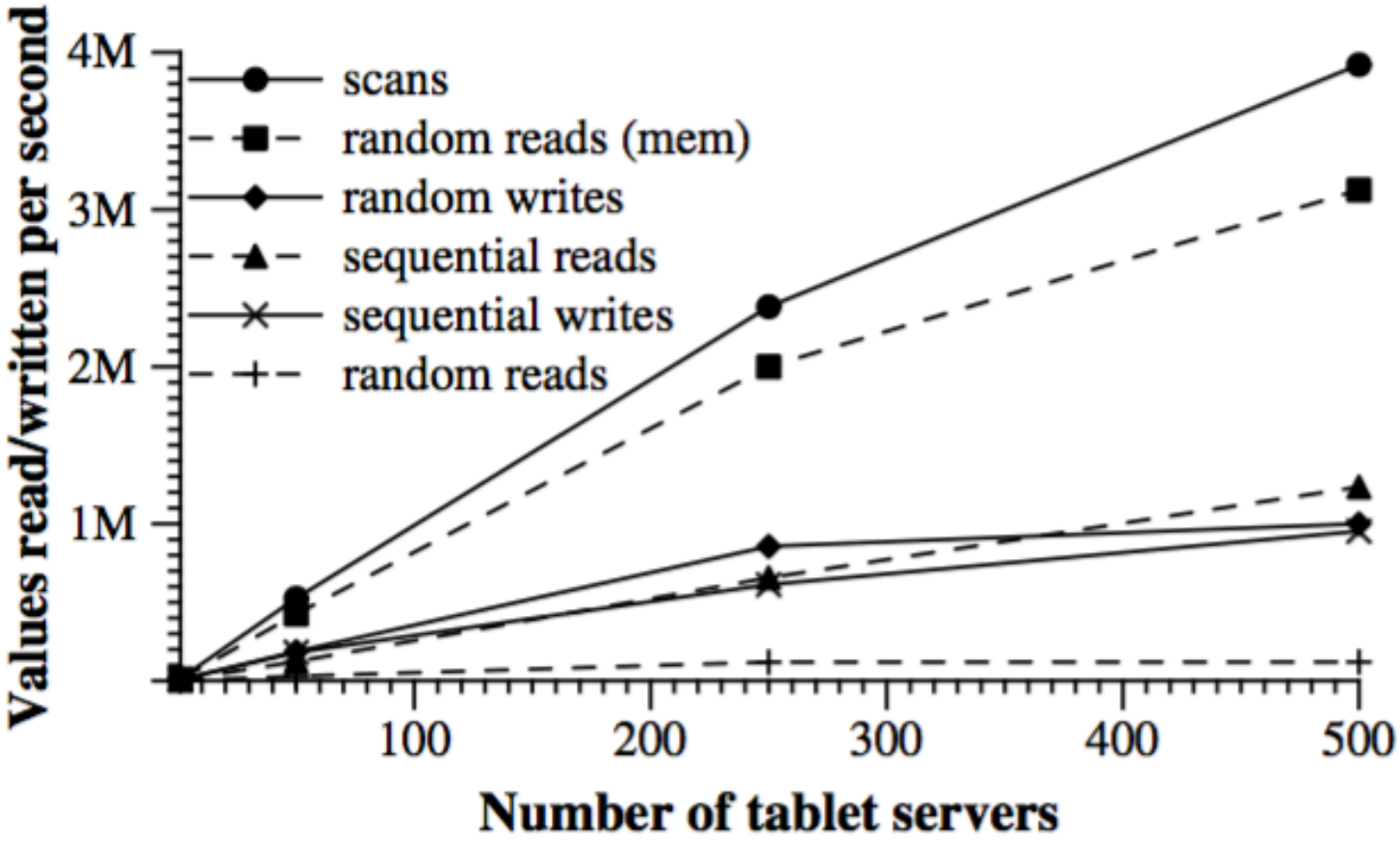
Tablet server: 1 GB mem,

clients = # servers

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

1000-byte values per server per second

Aggregate rate



Application at Google

Project name	Table size (TB)	Compression ratio	# Cells (billions)	# Column Families	# Locality Groups	% in memory	Latency-sensitive?
Crawl	800	11%	1000	16	8	0%	No
Crawl	50	33%	200	2	2	0%	No
Google Analytics	20	29%	10	1	1	0%	Yes
Google Analytics	200	14%	80	1	1	0%	Yes
Google Base	2	31%	10	29	3	15%	Yes
Google Earth	0.5	64%	8	7	2	33%	Yes
Google Earth	70	–	9	8	3	0%	No
Orkut	9	–	0.9	8	5	1%	Yes
Personalized Search	4	47%	6	93	11	5%	Yes