Key-value Stores
Bigtable & Dynamo

Iraklis Psaroudakis

Data Management in the Cloud

EPFL, March 5th 2012

Outline

- Motivation & NOSQL
- Bigtable
- Dynamo
- Conclusions
- Discussion

Motivation

- Real-time web requires:
  - Processing of large amounts of data while maintaining efficient performance
  - Distributed systems and replicated data (to globally distributed data centers)
    - Commodity hardware
  - High availability and tolerance to network partitioning
  - Requirements varying between high throughput & low latency
  - Simple storage access API

RDBMS?

More database approaches

RDBMS

Col1	Col2	Col3
5	Val1	0101011
6	Val2	1101001

- SQL
- ACID
- Strict table schemas

Spectrum of requirements

Bigtable

Dynamo

Key-value stores

Key	Value
key1	0101011
key2	val_10

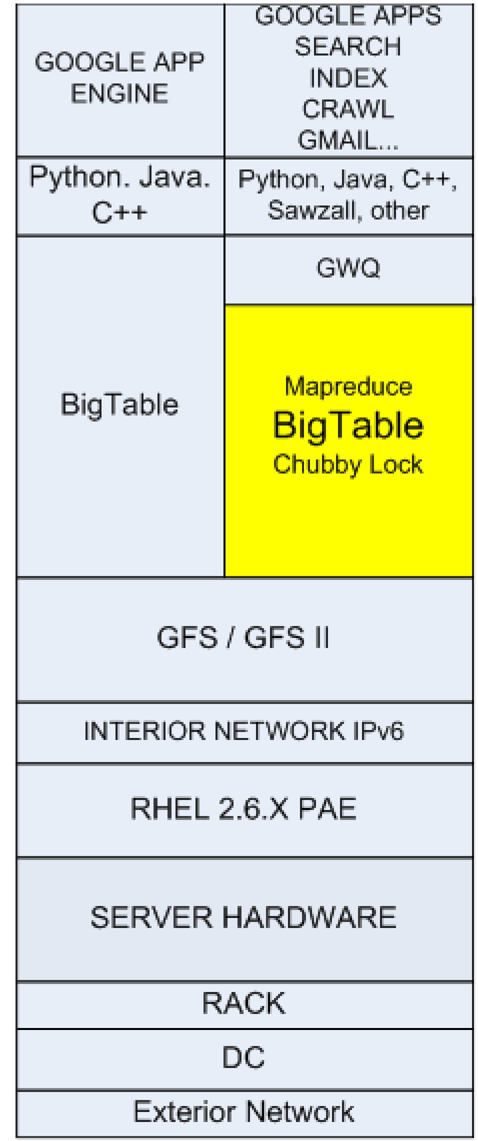
- Less flexible API
- BASE
- Schema-less

NOSQL

- Next Generation Databases mostly addressing some of the points: being **non-relational, distributed, open-source** and **horizontally scalable**. The original intention has been **modern web-scale databases**. The movement began early 2009 and is growing rapidly. Often more characteristics apply such as: **schema-free, easy replication support, simple API, eventually consistent** / **BASE** (not ACID), a **huge amount**, **of data** and more.
  - from http://nosql-database.org/
  - Column stores (Bigtable), document (e.g. XML) stores, key-value stores (Dynamo), Graph & Object DBs etc.

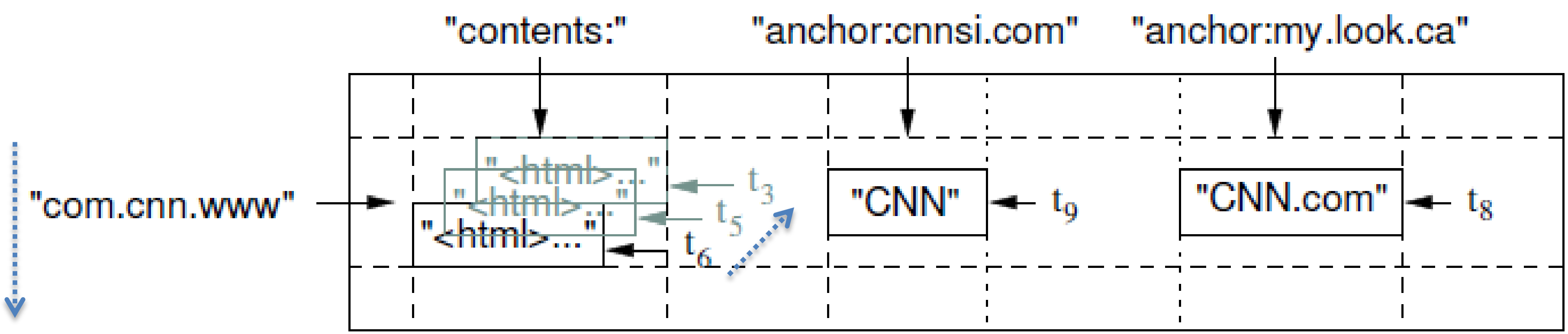
Bigtable

- Distributed storage system for structured data
- High scalability
- Fault tolerant
- Meets throughput-oriented but also latency-sensitive workloads
- Load balancing
- Strong consistency (?)
- Dynamic control over data layout
  - Does not support full relational model
  - Can serve from disk and from memory



Data model

- "Sparse, distributed, persistent, multidim. sorted map"
- (row:string, column:string, time:int64) → string
- Column key = family:qualifier
  - Few families per table, but unlimited qualifiers per family
  - Data in a family is usually of the same type (for compression)
  - Access control lists are defined on a family-level



API

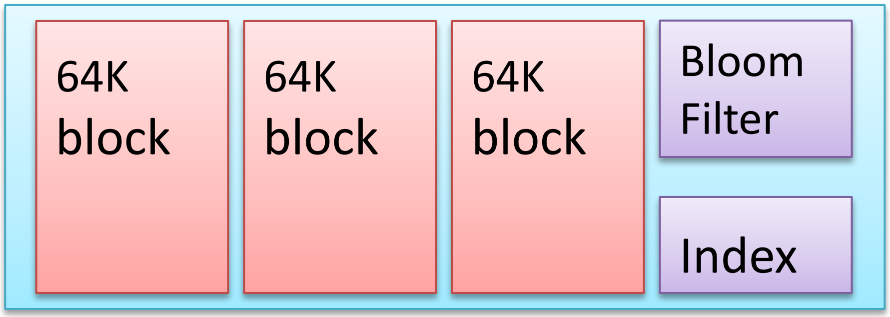
```
// Open the table
Table *T = OpenOrDie("/bigtable/web/webtable");
// Write a new anchor and delete an old anchor
RowMutation r1(T, "com.cnn.www");
r1.Set("anchor:www.c-span.org", "CNN");
r1.Delete("anchor:www.abc.com");
Operation op;
Apply(&op, &r1);
```

Writing data
(atomic mutations)

```
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());
}
```

Reading data

Physical Storage & Refinements



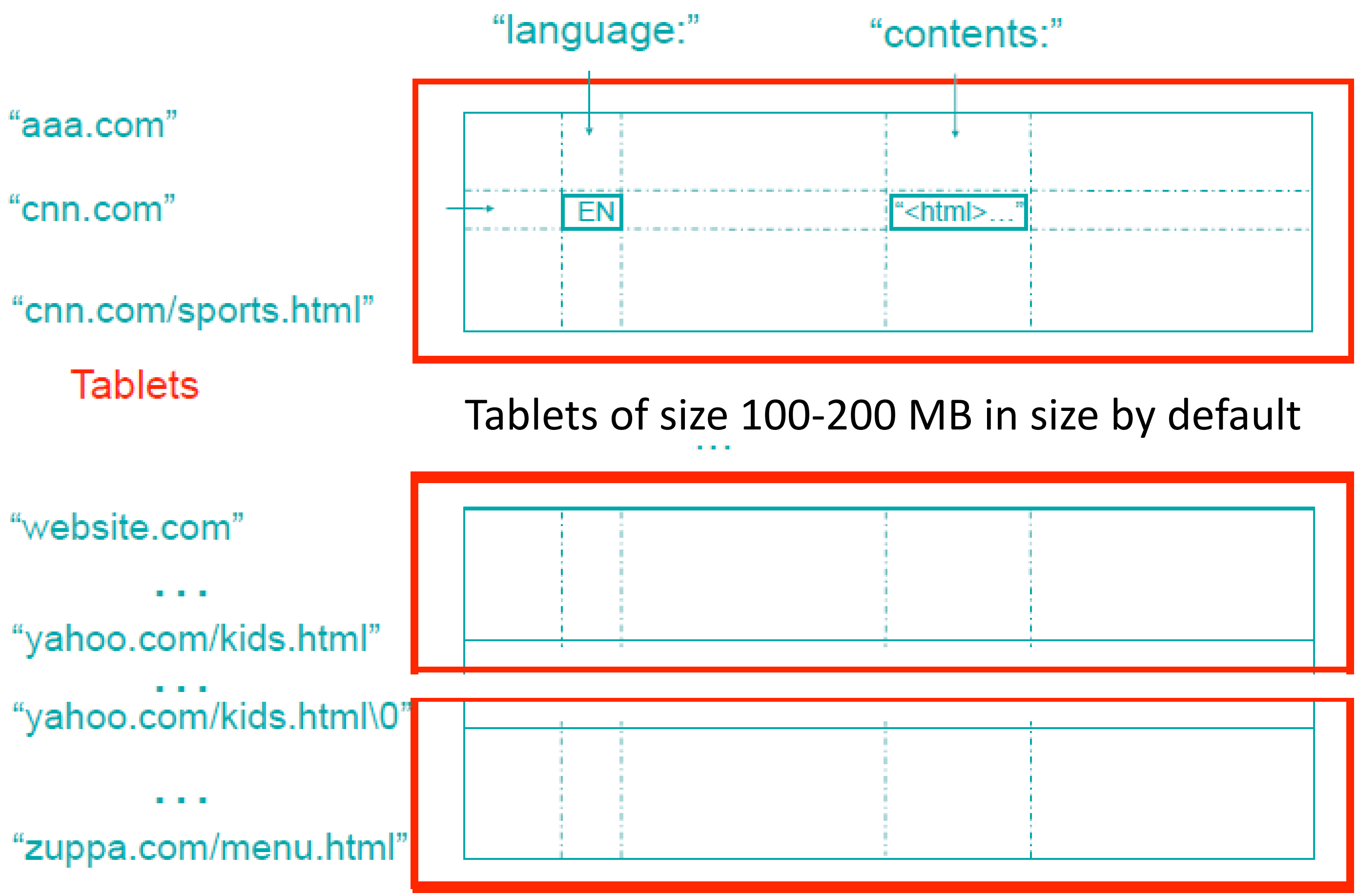
SSTable file format:
Persistent, ordered immutable map from keys to values (in GFS)

rowkey	family	qualifier	ts	value
u1	user	email	2012	rb@example.com
u1	user	email	2007	ra@example.com
u1	user	name	2007	Ricky
u1	social	friend	2007	u2
u2	user	name	2007	Sam
u2	user	telephone	2007	0780000000
u2	social	friend	2007	u1

SSTable for **locality group**:
“user”, “social” families

A locality group supports:
- Being loaded into memory
- User-specified compression algorithm

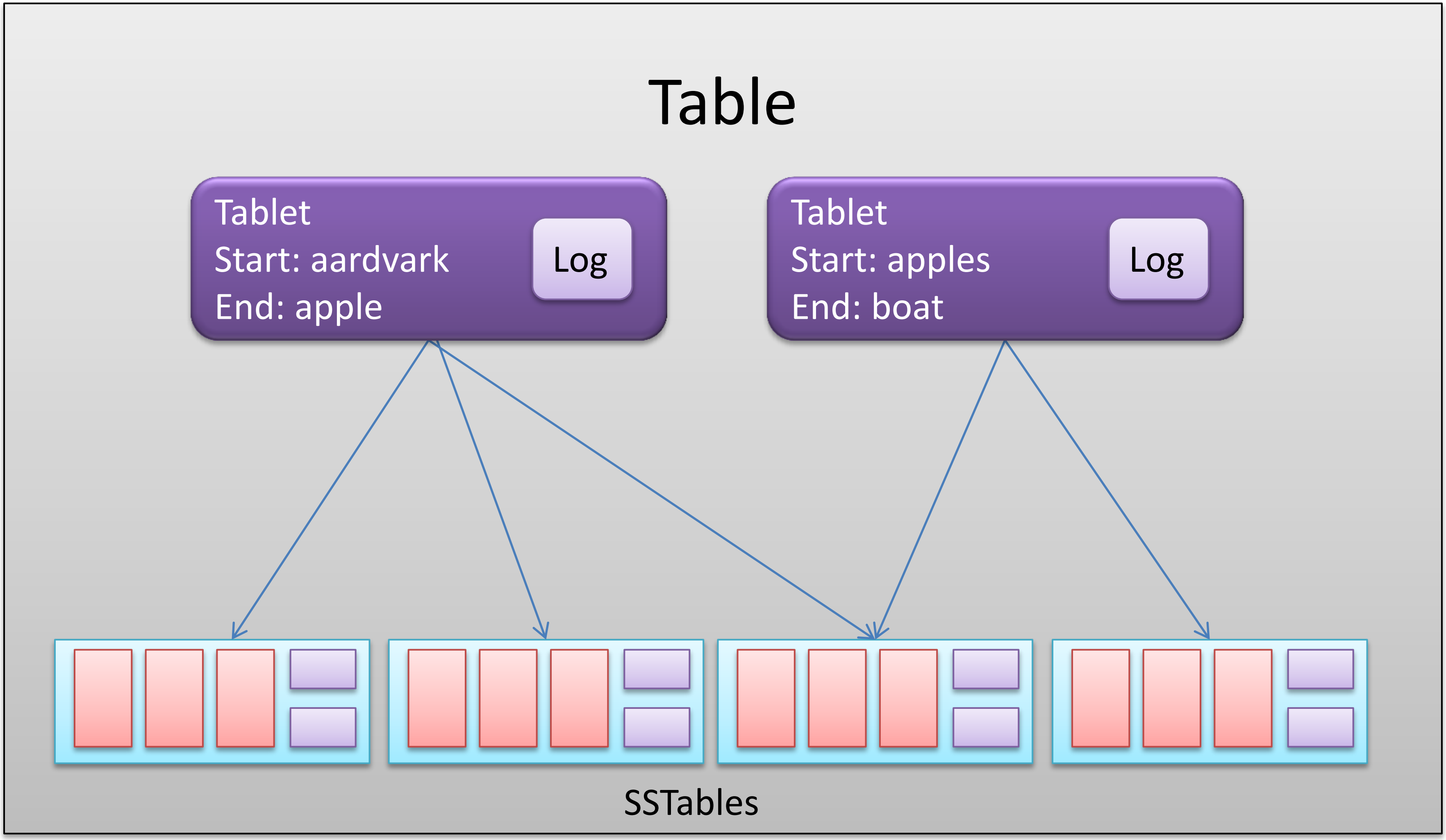
Splitting into tablets



Tablets of size 100-200 MB in size by default

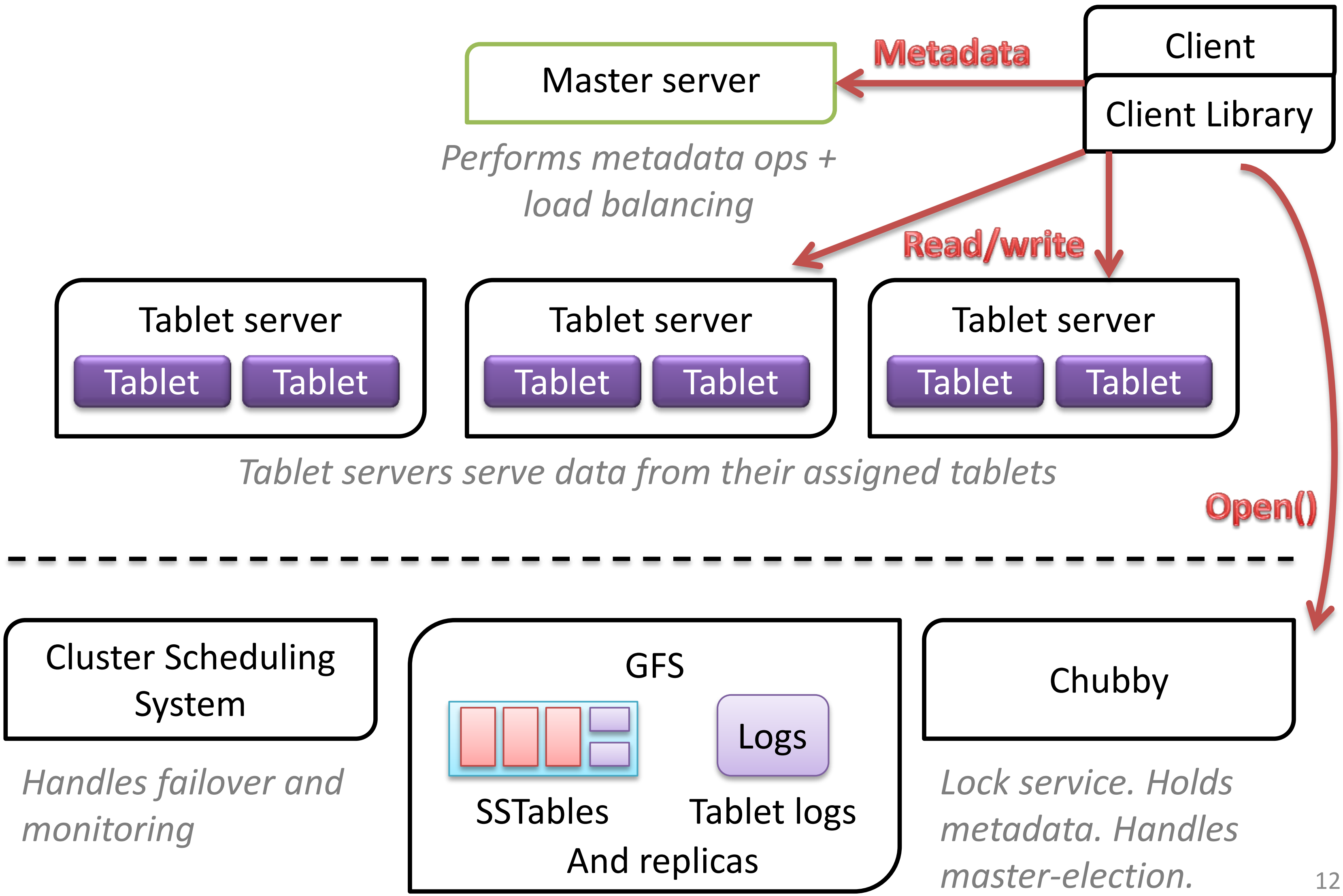
[Discussion] Column indices?

Componentization of Tables



[Discussion] Why are SSTables shared?

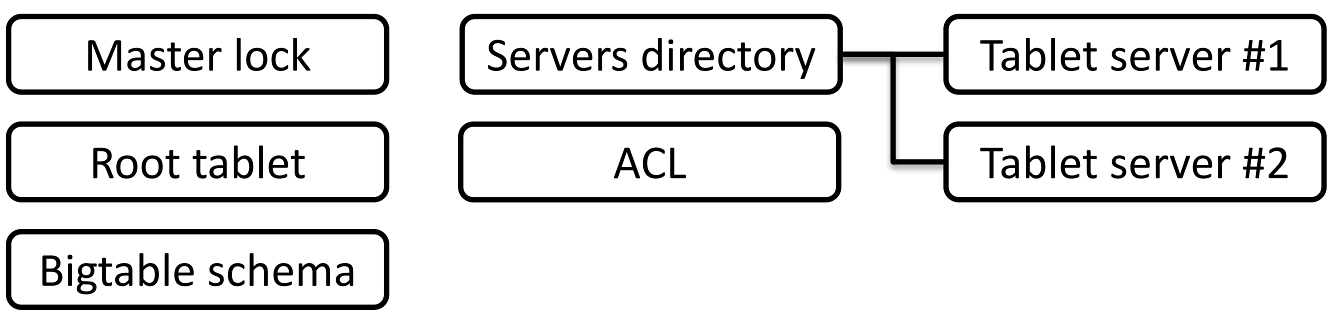
Bigtable System Architecture



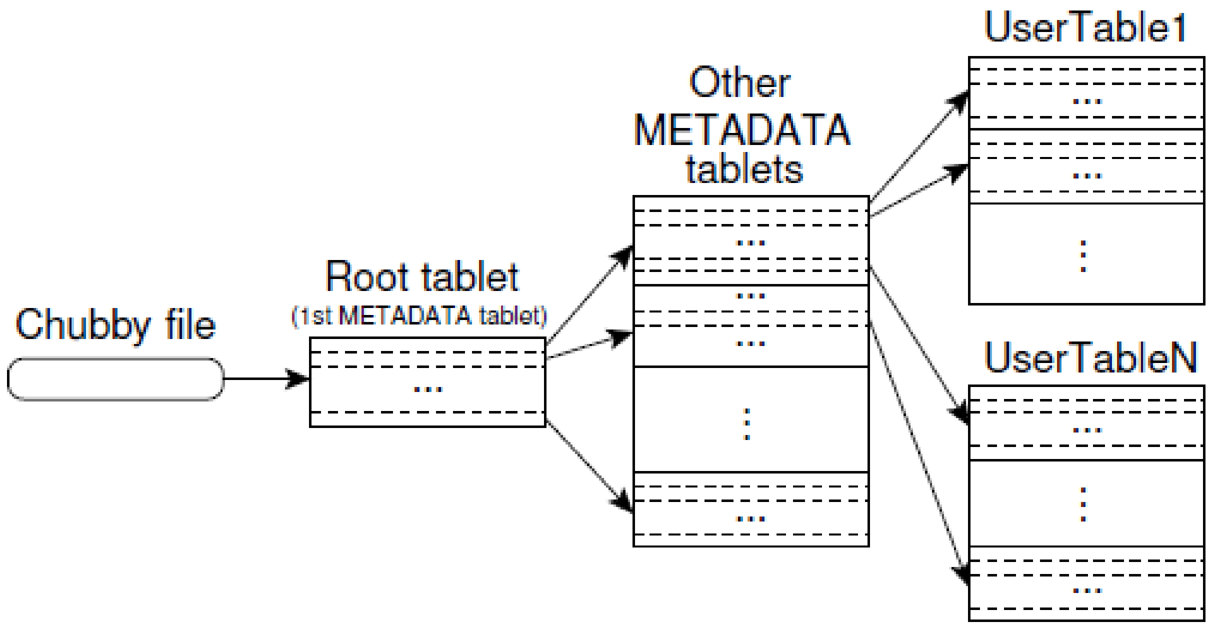
Chubby

- Distributed lock service
- File System {directory/file}, for locking
- High availability
  - 5 replicas, one elected as master
  - Service live when majority is live
  - Uses Paxos algorithm to solve consensus
- A client leases a session with the service

Bigtable and Chubby



Finding a tablet



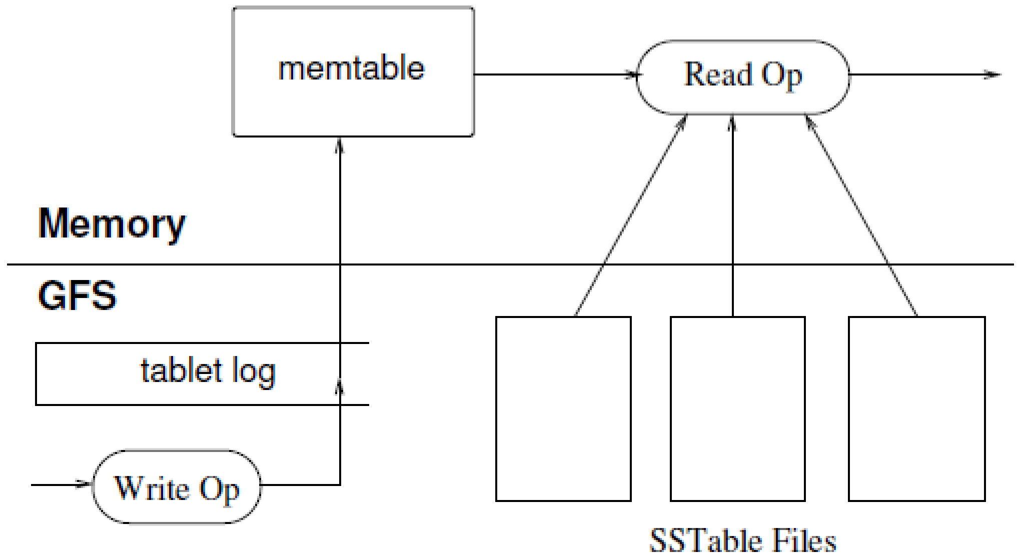
METADATA table

table_key	end_row_key	tablet_server
UserTable1	aardvark	server1.google.com
UserTable1	boat	server1.google.com
UserTable1	ear	server9.google.com
UserTable2	acre	server3.google.com

Master operations

- Uses Chubby to monitor the health of tablet servers
  - Each tablet server holds a lock in the servers directory of Chubby
  - The server needs to renew its lease periodically with Chuby
  - If the server has lost the lock, it re-assigns its tablets to another server. Data and log are still in GFS.
- When (new) master starts
  - Grabs master lock in Chubby
  - Finds live tablet servers by scanning Chubby
  - Communicates with them to find assigned tablets
  - Scans the METADATA table in order to find unassigned tablets
- Handles table creation and merging of tablets
  - Tablet servers directly update METADATA on tablet split and then notify the master. Any failure is detected lazily by the master.

Tablet Serving



- Commit log stores the writes
  - Recent writes are stores in the memtable
  - Older writes are stores in SSTables
- A read operation sees a merged view of the memtable and the SSTables
- Checks authorization from ACL stored in Chubby

[Discussion] What operations are in the log?

Why did they choose not to write immediately to SSTables?

Compactions

- Minor compaction – convert the memtable into an SSTable
  - Reduce memory usage
  - Reduce log traffic on restart
- Merging compaction
  - Reduce number of SSTables
- Major compaction
  - Merging compaction that results in only one SSTable
  - No deletion records, only live data (garbage collection)
  - Good place to apply policy “keep only N versions”

Refinements

- Locality groups – Groups families
- Compression – Per SSTable block
- Caches
  - Scan cache: key-value pairs
  - Block cache: SSTable blocks
- Bloom Filters – Minimize I/O accesses
- Exploiting SSTable immutability
  - No synchronization for reading the file system
  - Efficient concurrency control per row
  - Comfortable garbage collection of obsolete SSTables
  - Enables quick tablet splits (SSTables can be shared)

Log handling

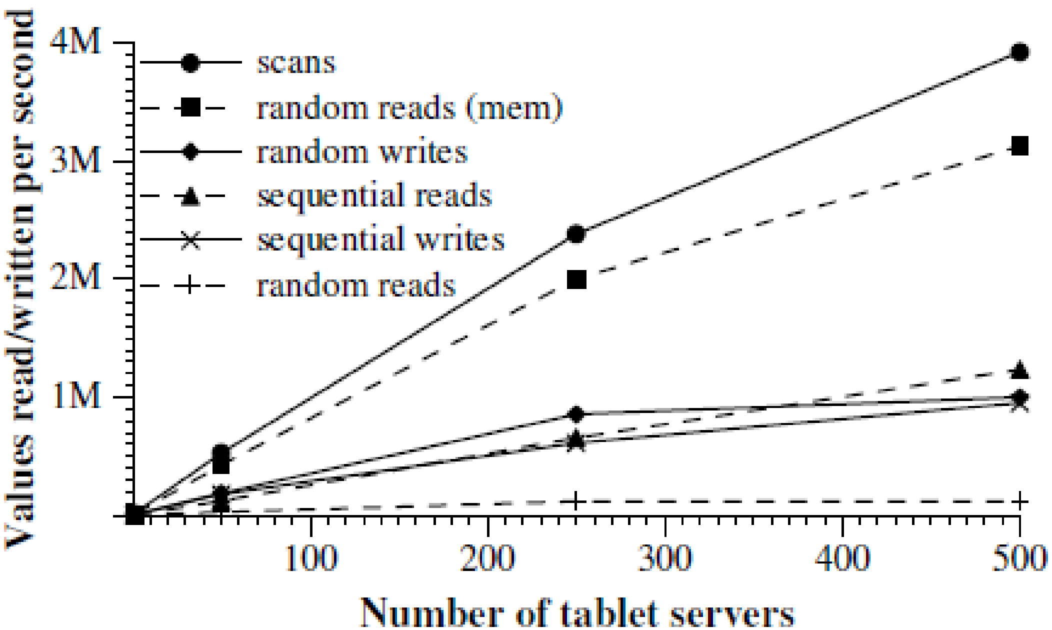
- Commit-log per tablet server
  - Co-mingling a server's tablets' logs into one commit log of the server.
    - Minimizes arbitrary I/O accesses
    - Optimizes group commits
  - Complicates tablet movement
- GFS delay spikes can mess up log write
  - Solution: two separate logs, one active at a time
  - Can have duplicates between the two that are resolved with sequence numbers

Benchmarks

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

Number of 1000-byte values read/written per second

Per tablet server



Number of 1000-byte values read/written per second

Aggregate values

Dynamo

- Highly-available key-value storage system
  - Targeted for primary key accesses and small values (< 1MB).
- Scalable and decentralized
- Gives tight control over tradeoffs between:
  - Availability, consistency, performance.
- Data partitioned using consistent hashing
- Consistency facilitated by object versioning
  - Quorum-like technique for replicas consistency
  - Decentralized replica synchronization protocol
  - Eventual consistency

Dynamo (2)

- Gossip protocol for:
  - Failure detection
  - Membership protocol
- Service Level Agreements (SLAs)
  - Include client's expected request rate distribution and expected service latency.
  - e.g.: Response time < 300ms, for 99.9% of requests, for a peak load of 500 requests/sec.
- Trusted network, no authentication
- Incremental scalability
- Symmetry
- Heterogeneity, Load distribution

Related Work

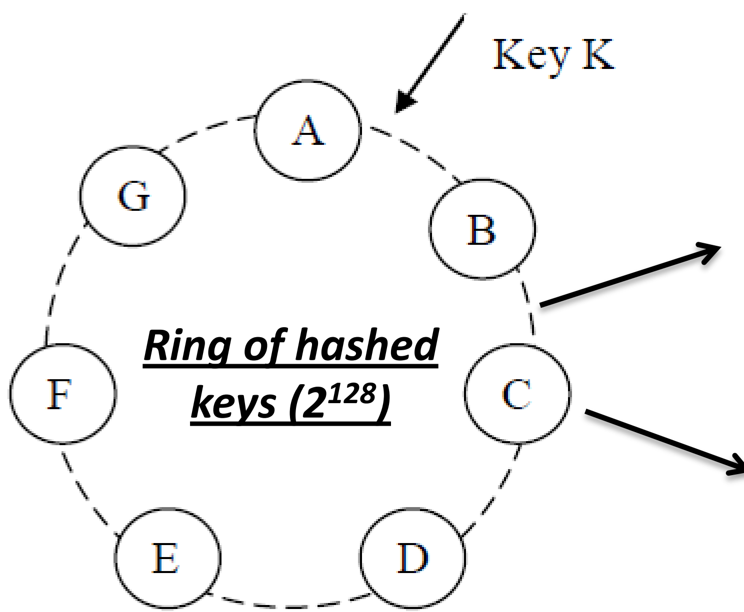
- P2P Systems (Unstructured and Structured)
- Distributed File Systems
- Databases
- However, Dynamo aims to:
  - be always writeable
  - support only key-value API (no hierarchical namespaces or relational schema)
  - efficient latencies (no multi-hops)

Data partitioning

- get(key) : (context, value)
- put(key, context, value)
- MD5 hash on key (yields 128-bit identifier)

New nodes are randomly assigned “tokens”.

All nodes have full membership info.



Range of hashed keys (C is responsible)

Node (physical) / token

Virtual nodes

- Random assignment leads to:
  - Non-uniform data
  - Uneven load distribution
- Solution: “virtual nodes”
  - A single node (physical machine) is assigned multiple random positions (“tokens”) on the ring.
  - On failure of a node, the load is evenly dispersed
  - On joining, a node accepts an equivalent load
  - Number of virtual nodes assigned to a physical node can be decided based on its capacity.

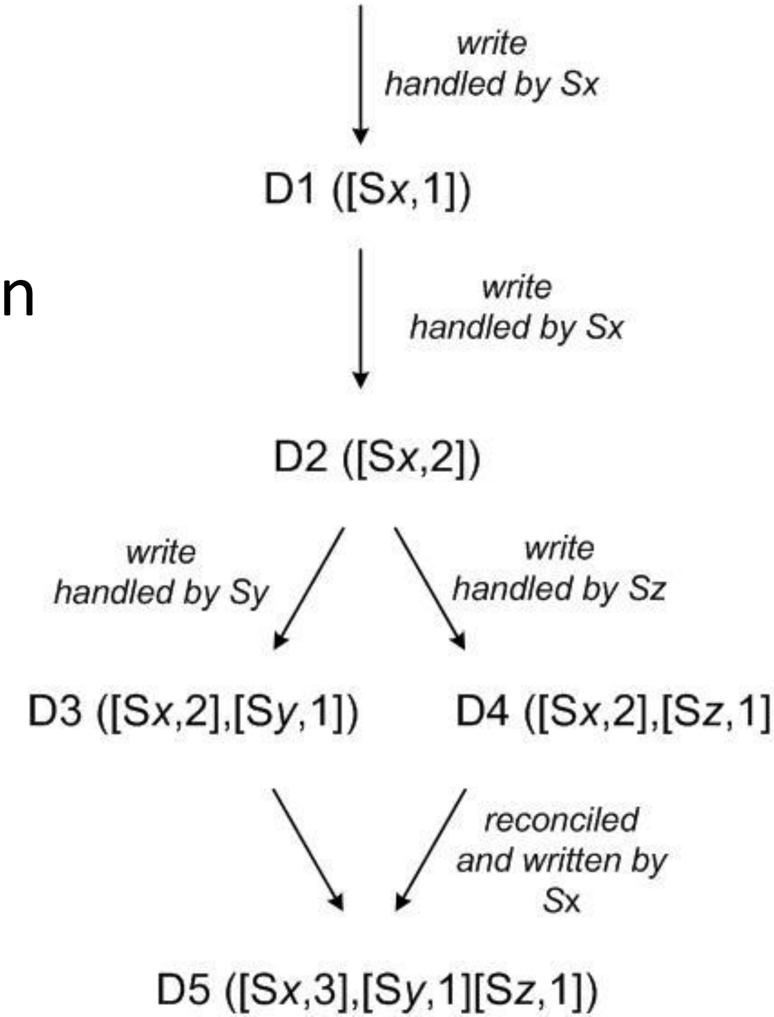
Replication

- N = number of replicated machines
  - Coordinator node replicates the data item to the next N nodes (machines) in the ring.
  - Preference list: List of nodes responsible for storing a key, with coordinator on top.
- Data versioning
  - Eventual consistency: Updates are propagated to all replicas asynchronously
  - Multiple versions of an object can be present
  - Timestamp-based or business-logic reconciliation

[Discussion] How is Bigtable's versioning different than Dynamo's?

Reconciliation of conflicting versions

- Vector clock: (node, counter) pair.
  - Captures causality between different versions
  - One vector clock per version per value
- Syntactic reconciliation: performed by system
- Semantic reconciliation: performed by client



Execution

- Any node can receive a get() or put()
- Client can select a node for the request using:
  - A generic load balancer (max one hop)
  - A partition-aware client library (zero hop)
- Coordinator for a request:
  - One of the top N healthy nodes in the preference list of a key, typically the first.

Consistency

- Quorum-like consistency among replicas. Three configurable parameters:
  - N: number of replicas
  - R: min. no. of nodes that must read a key
  - W: min. no. of nodes that must write an update
- Full-quorum systems: R+W > N
- Dynamo sacrifices consisten
  - Usually R < N and W < N
  - Typical values: N=3, R=2, W=2

Knobs for:

- Availability
- Performance
- Consistency

Handling failures

- Temporary failures:
  - "Sloppy quorum" sends read/write operations to N ***healthy*** machines on the ring.
    - If a node is down, its replica is sent to an extra machine ("hinted handoff"). It later sends the updates back to the failed machine.
- Permanent failures:
  - Synchronizing replicas periodically (anti-entropy)
    - Each node maintains a separate Merkle tree for each key range it is assigned.

Membership and Failure detection

- Explicit addition/removal of a node
  - Gossip protocol propagates membership changes
  - Also propagates partitioning information
- Local notion of failure detection
  - If node A does not receive reply from node B, it considers B failed.
  - Periodically retries communication.
  - No need for a globally consistent view of failures.

Implementation

Storage Node

Java

Request Coordination

- Built on top of event-driven messaging substrate
- Uses Java NIO
- Coordinator executes client read & write requests
- State machines created on nodes serving requests

Membership & Failure Detection

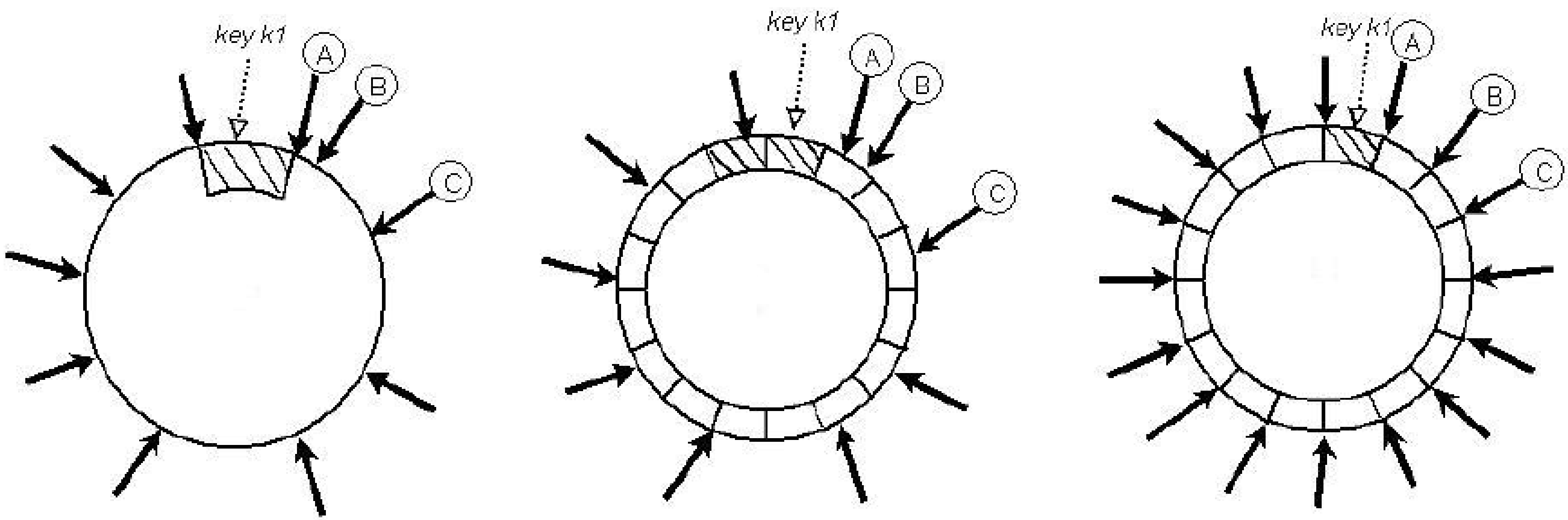
- Each state machine instance handles exactly one client request
- State machine contains entire process and failure handling logic

Local Persistence Engine

Pluggable Storage Engines

- *Berkeley Database (BDB) Transactional Data Store*
- *BDB Java Edition*
- *MySQL*
- *In-memory buffer with persistent backing store*

Partitioning schemes



Strategy 1

T random tokens per node and partition by token value

Strategy 2

T random tokens per node and Q equal-sized partitions (Q is large)

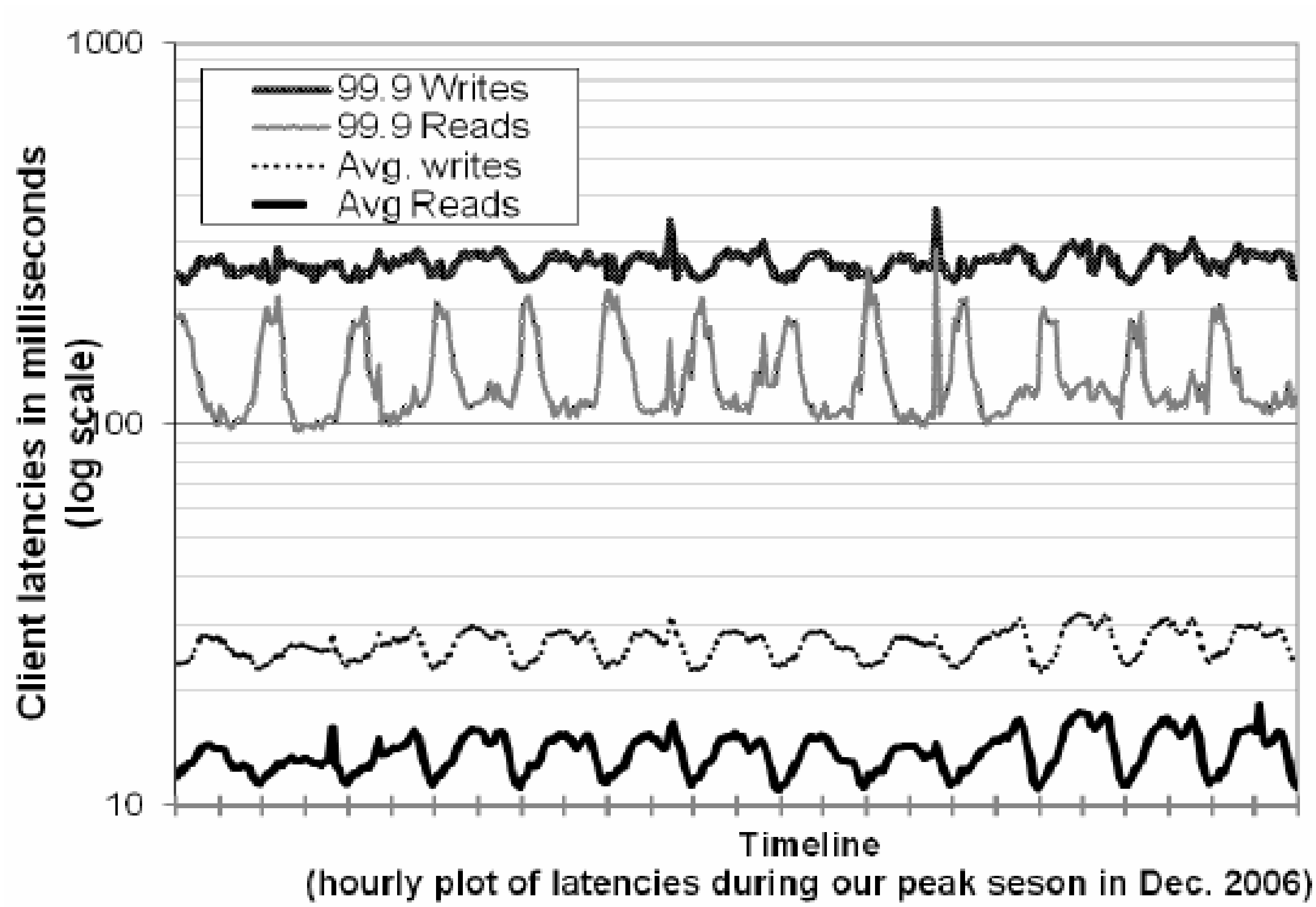
Strategy 3

Q/S tokens per node, equal-sized partitions (S machines)

In terms of load distribution efficiency,

Stategy 3 > Strategy 1 > Strategy 2

Experiments: Latencies

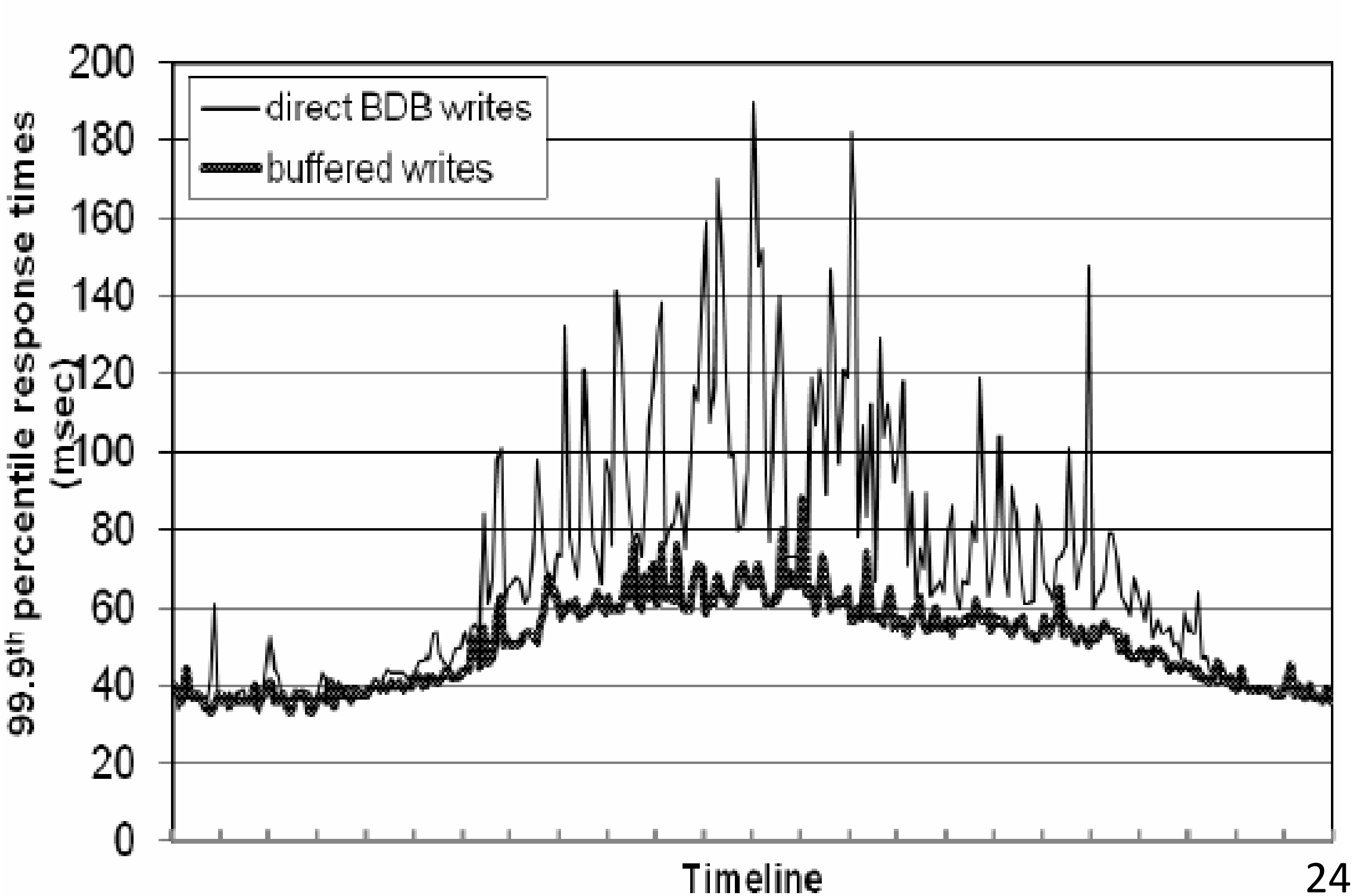


SLA of 300ms

One order of magnitude

1 interval = 12h

Experiments: Buffered writes



24h

1 interval = 1h

Experiments: Divergent Versions

- Divergent versions arise when:
  - The system is facing failure scenarios
  - Large number of concurrent write requests for a value. Multiple nodes end up coordinating.
- Experiment during a 24h period showed:
  - 99.94% requests saw one version
  - 0.00057% saw 2
  - 0.00047% saw 3
  - 0.00009% saw 4

Comparison

	Dynamo	**Bigtable**
Data Model	key value, row store	column store
API	single tuple	single tuple and range
Data Partition	random	ordered
Optimized for	writes	writes
Consistency	eventual	atomic
Multiple Versions	version	timestamp
Replication	quorum	file system
Data Center Aware	yes	yes *
Persistency	local and plug-gable	replicated and distributed file system
Architecture	decentralized	hierarchical
Client Library	yes	yes

Cassandra website:
Bigtable: “How can we build a distributed DB on top of GFS?”
Dynamo: “How can we build a distributed hash table appropriate for the data center?”

Conclusions

- Bigtable and Dynamo offer two very different approaches for distributed data stores.
  - Bigtable is more tuned for scalability, consistency and Google's specific needs.
  - Dynamo is more tuned for various SLAs, sacrificing consistency and achieving high availability. Scales only to a couple hundred machines.
- Dynamo proves to be more flexible due to N, R, W properties. Also correlates to academic research (P2P, Quorum, Merkle trees etc.).

Open-source implementations

- Dynamo – Kai
- Bigtable – HBase (based on Hadoop)
- Apache Cassandra
  - Brings together Dynamo’s fully distributed design and Bigtable’s data model

Thank you!

iraklis.psaroudakis@epfl.ch

Parts taken from:

- Designs, Lessons and Advice from Building Large Distributed Systems – Jeff Dean (Google Fellow)
- The Anatomy of the Google Architecture (Ed Austin)
- Dynamo: Amazon's Highly Available Key-value Store (Jagrut Sharma)
- Ricardo Vilaca, Francisco Cruz, and Rui Oliveira. 2010. On the expressiveness and trade-offs of large scale tuple stores. In Proceedings of the 2010 international conference on On the move to meaningful internet systems: Part II (OTM'10)

Papers on Bigtable and Dynamo:

- Fay Chang, Jeffrey Dean, et al. 2008. Bigtable: A Distributed Storage System for Structured Data.
- Giuseppe DeCandia, et al. 2007. Dynamo: amazon's highly available key-value store. In Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles (SOSP '07).

CAP theorem

- Eric Brewer: You can have at most two of (**C**onsistency, **A**vailability, tolerance to network **P**artitions) for any shared-data system
  - CA: ACID DBs & Bigtable?, AP: Dynamo
  - Is Bigtable really strongly consistent, is GFS?
- Daniel Abadi: PACELC
  - "If there is a partition (**P**) how does the system tradeoff between **A**vailability and **C**onsistency; else (**E**) when the system is running as normal in the absence of partitions, how does the system tradeoff between Latency (**L**) and Consistency (**C**)."
  - PC/EC: ACID DBs & Bigtable?, PA/EL: Dynamo
- Interesting article in IEEE: "Overcoming CAP with Consistent Soft-State Replication"

Discussion topics

- What kind of services need RDBMS?
  - In specific, what would you not implement with NOSQL?
  - Which services need transactions and strong consistency?
- Centralized / hierarchical approaches or decentralized?
  - Bigtable's membership information is kept in METADATA table
  - Dynamo's membership information is gossiped
- How would you scale Dynamo to large numbers of machines?
- Who answers read/write requests faster, Dynamo or Bigtable?
- Has anyone used another NOSQL system, e.g. Cassandra?