Cloud & Cluster Data Management

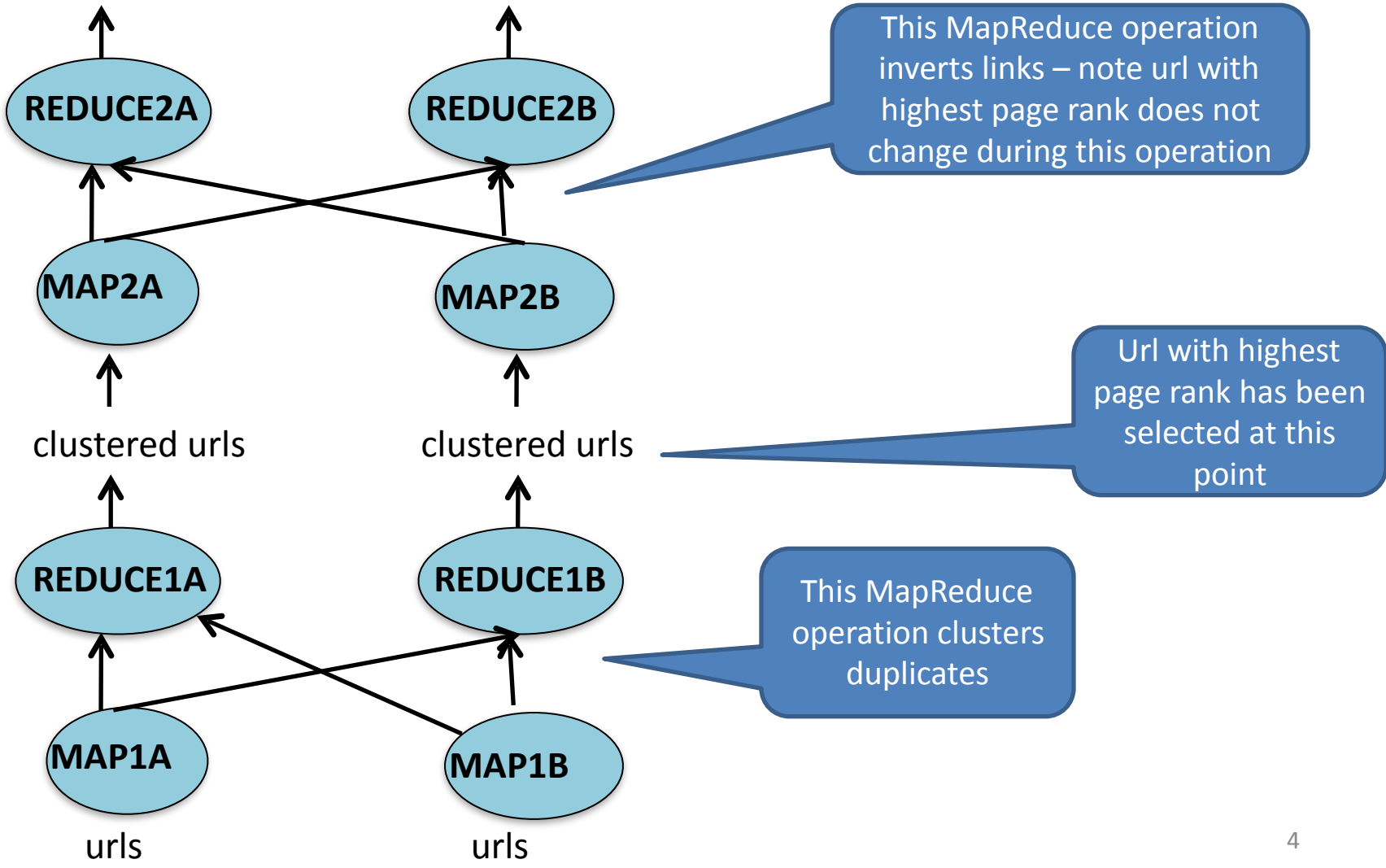
PERCOLATOR

Why Percolator?

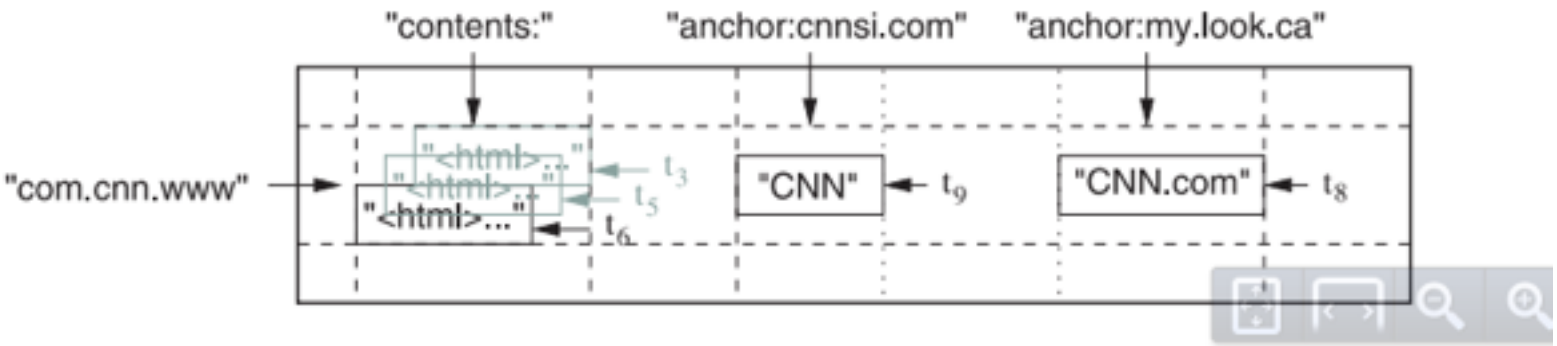
- Transform **large repository** of data with **small independent mutations**
  - Petabytes of data, billions of updates per day on thousands of machiens
- Option 1 : Databases
  - Do not meet storage or throughput requirements
- Option 2: Map Reduce
  - Rely on large batches for efficiency
  - Cannot process small updates individually
  - "**MapReduce and other batch-processing systems cannot process small updates individually as they rely on creating large batches for efficiency**"

Let's Build a Web Index!

- Google Web Index structure:
  - Initially built by crawling every page on the web
  - Only one URL if multiple pages have same content (highest page rank)
  - Links are inverted
  - Links to duplicates -> page with highest page rank
- Initial index creation - series of Map Reduce operations
  - Clustering duplicates, inverting links (note one step finishes before the next)


REDUCE2A
REDUCE2B
This MapReduce operation inverts links – note url with highest page rank does not change during this operation
MAP2A
MAP2B
clustered urls
clustered urls
Url with highest page rank has been selected at this point
REDUCE1A
REDUCE1B
This MapReduce operation clusters duplicates
MAP1A
MAP1B
urls
urls

Web Table


"contents:"
"anchor:cnnsi.com"
"anchor:my.look.ca"
"com.cnn.www"
"<html>..."
"<html>..."
"<html>..."
t3
t5
t6
"CNN"
t9
"CNN.com"
t8

Discussion Question

- How would you update the web index with MapReduce?

- Do you agree with the statement: "**MapReduce and other batch-processing systems cannot process small updates individually as they rely on creating large batches for efficiency**"?

Now we need to Update the Index

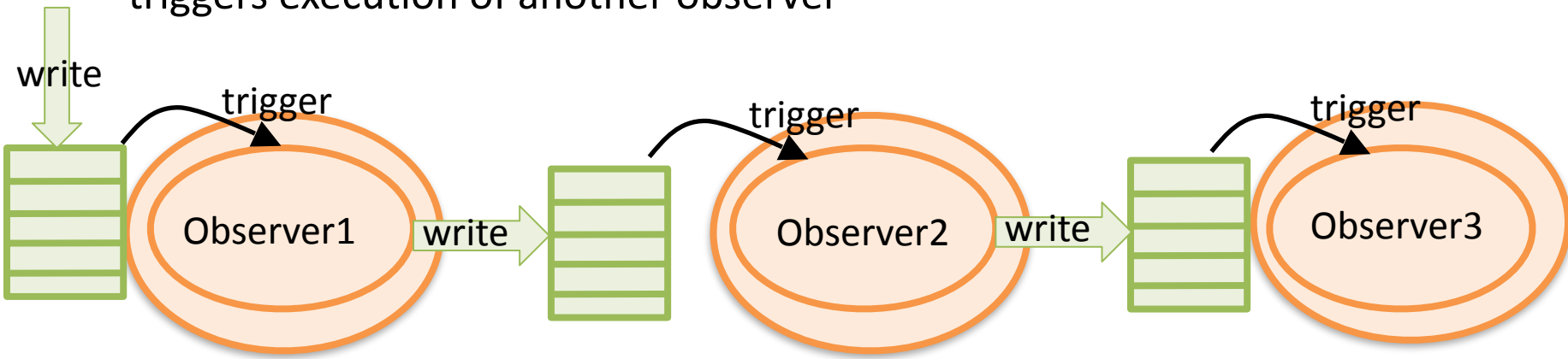
- Option 1:
  - Run MR over the new pages (but there are links between new and old pages)
- Option 2: Rerun MR over entire repository of pages
  - Expensive! Latency proportional to size of repository, not size of update
  - But…it used to be done this way…
- Option 3: Database (updates + xacts)
  - We know the story here…DB can’t handle the volume
- Option 4: Big Table
  - Scales…but…no transactions across rows…

So…Percolator

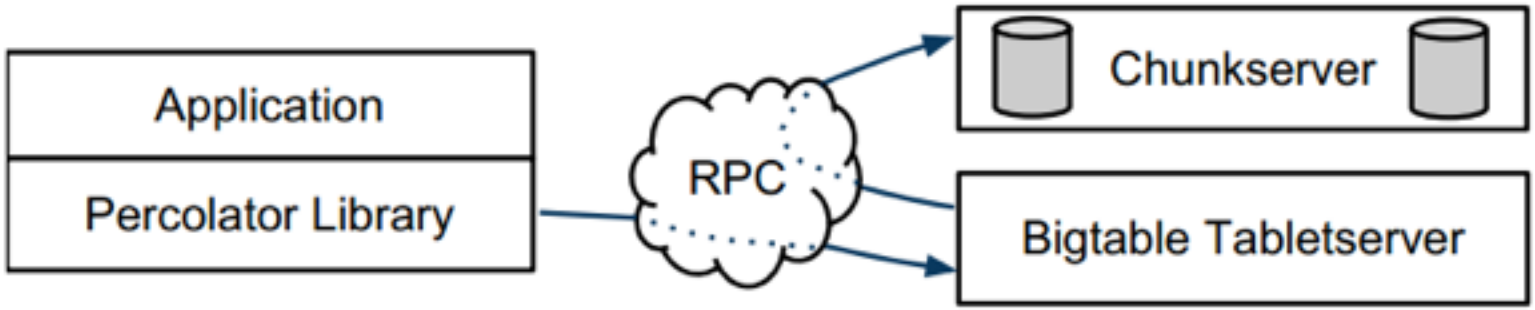
- Features
  - Incrementally process index updates
  - Work proportional to size of updates, not size of repository
- Use cases
  - Strong consistency requirements (else Bigtable)
  - Large computation requirement of some sort (data, CPU, etc.) (else DBMS)
- Use in Google web indexing
  - Process documents as they are crawled
  - Reduce average document processing latency by a factor of 100
  - Reduce average age of document appearing in search result by 50%

Percolator - Features

- ACID-compliant transactions – snapshot isolation semantics
  - **Random access to a multi-PB repository**
  - Required because need many threads/many machines for high throughput
- Observers : organize incremental computation
  - Observer invoked when user-specified column changes
  - Observers complete tasks, create more work by writing to a table – triggers execution of another observer

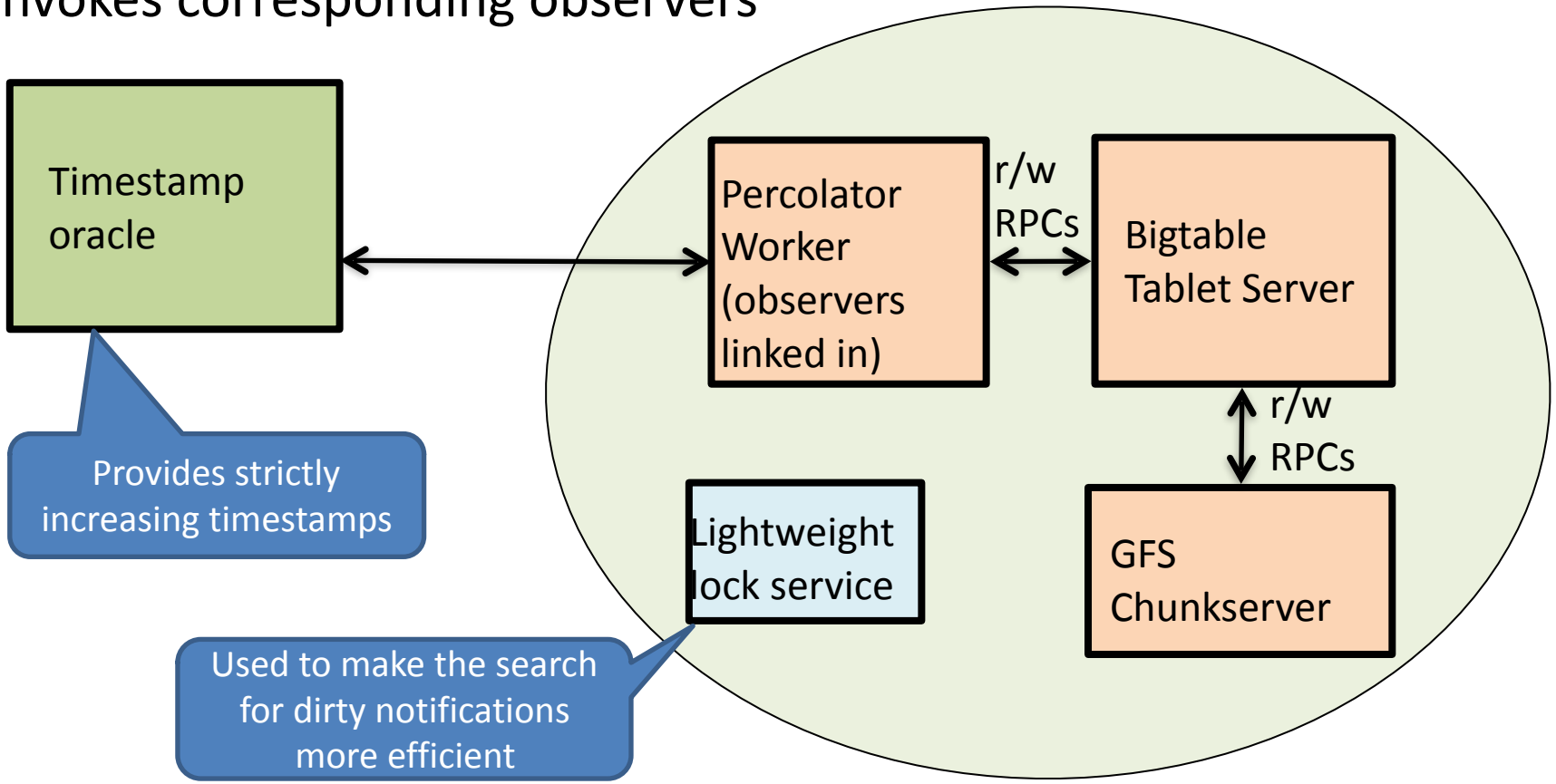


Percolator - Design



Percolator - Design

- Two main abstractions: ACID transactions, Observers
- Observers linked into Percolator worker
- Percolator worker scans Bigtable for column changes and invokes corresponding observers

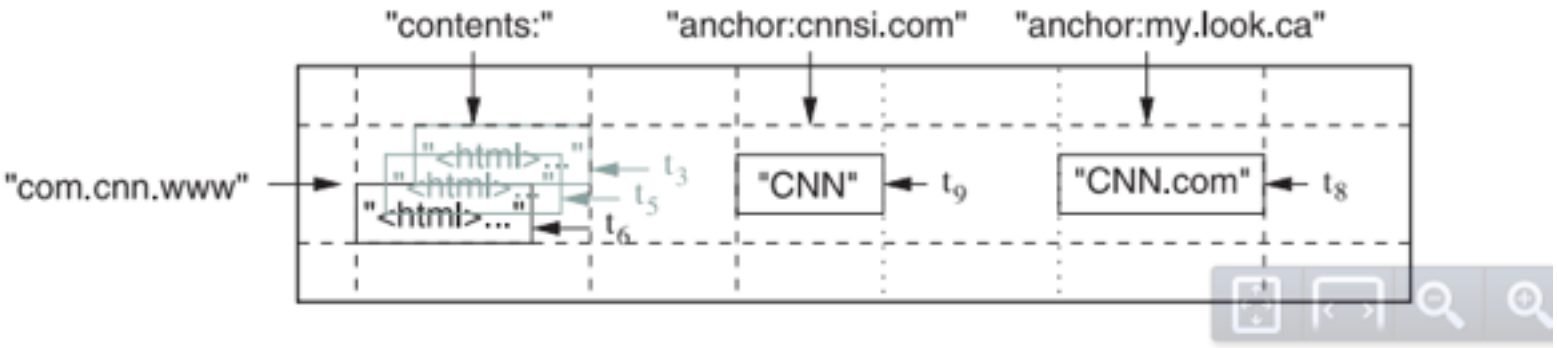


Design Considerations

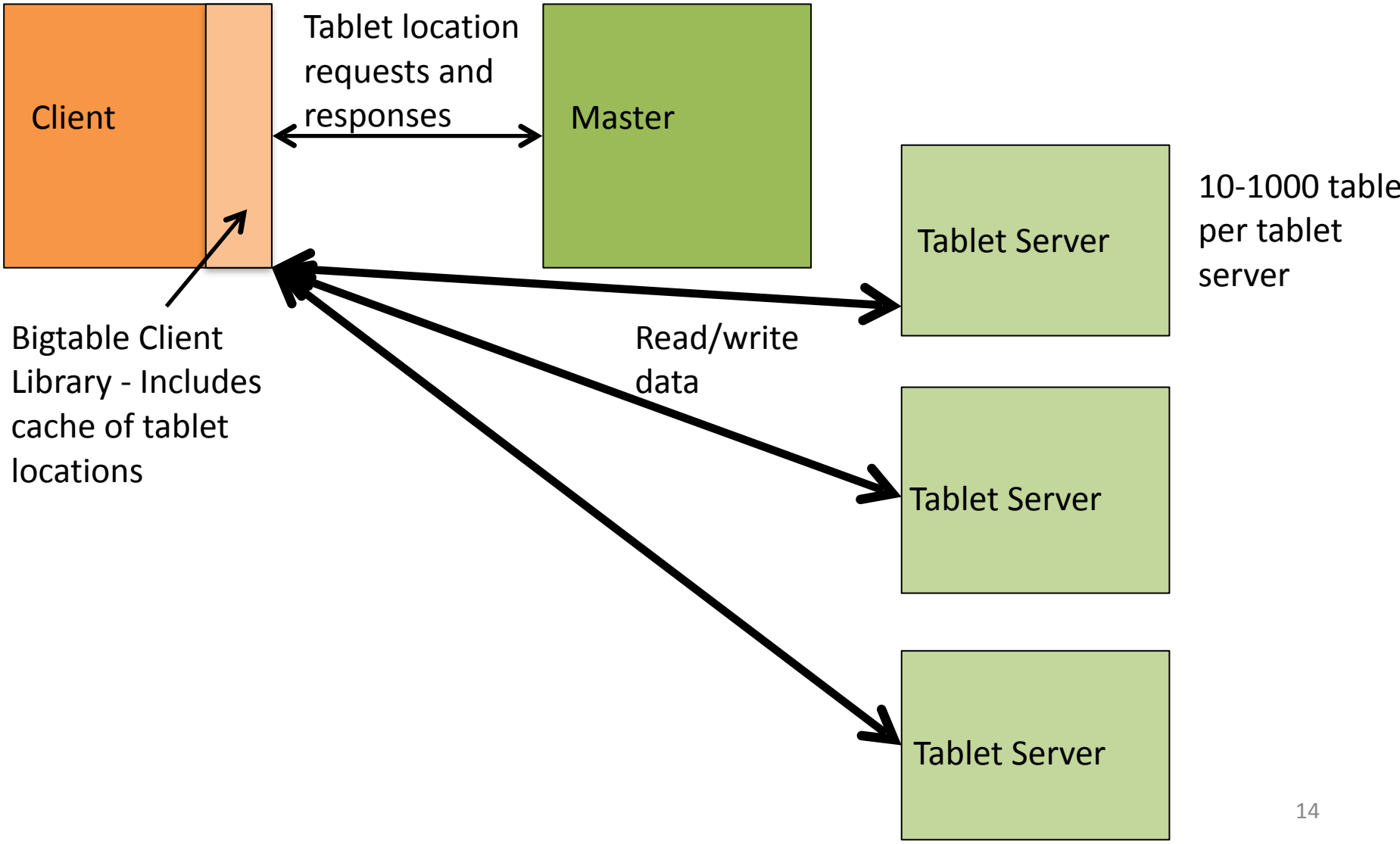
- Design Influences
  - Requirement to run at massive scales
  - Lack of requirement for low-latency
- Lazy approach to cleaning up locks left behind by failed transactions
  - Simple-to-implement
  - Delays transaction commits by tens of seconds
  - Tolerable for a system incrementally updating a web index (not tolerable for OLTP systems)
- No central location for transaction management
  - No global deadlock detector
  - Increases latency of conflicting transactions
  - Allows scaling

Bigtable Review

- Column family data model
- Atomic read-modify-write operations on individual rows
- Architecture: single master, many tablet servers
- Locality groups for column families
  - Storage is per locality group (vertical partitioning)
  - Can be declared to be in-memory



Bigtable - Implementation



Transactions

- Cross-row, cross-table transactions
- ACID snapshot isolation semantics
- API
  - C++ code with calls to Percolator API in the code
  - Calls to Get() and Commit() are blocking
- Usefulness
  - Can assume hash of contents of document is consistent with table that indexes duplicates, entries in both tables match
  - Without xacts, could have entry in doc table that corresponds to no URL in duplicates table (Invariant: Only one URL if multiple pages have same content (highest page rank)

Transaction API Example

```
bool UpdateDocument(Document doc) {
  Transaction t(&cluster);
  t.Set(doc.url(), "contents", "document", doc.contents());
  int hash = Hash(doc.contents());

  // dups table maps hash → canonical URL
  string canonical;
  if (!t.Get(hash, "canonical-url", "dups", &canonical)) {
    // No canonical yet; write myself in
    t.Set(hash, "canonical-url", "dups", doc.url());
  } // else this document already exists, ignore new copy
  return t.Commit();
}
```

Gets and Commits are blocking

Commit fails if two URLs with same content hash were processed at the same time

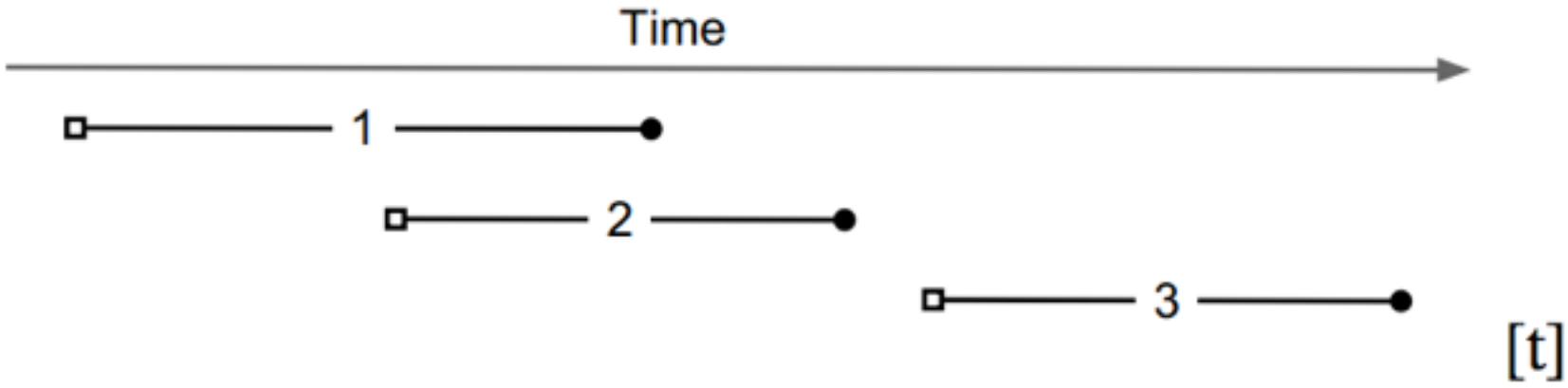
Snapshot Isolation

- Presents each transaction with appearance of reading from stable snapshot at some timestamp
- Protects against write-write conflicts
  - Two concurrent transactions (A & B) that write to same cell – either A or B will commit – one will abort
- Does not provide serializability
- Subject to write skew
  - A and B both read values v1 and v2, A updates v1, B updates v2
  - A and B both commit
- Reads are much more efficient
  - Data read: Bigtable look up at a given timestamp
  - Reads do not acquire locks
- Writes appear at a later timestamp than the read timestamp

This is the big advantage

But didn't you just say Gets are blocking?

Transaction Example



- Transactions perform reads at start timestamp (open square)
- Perform writes at commit timestamp (closed circle)
- T2 does not see writes from T1
- T3 sees writes from T1 and T2
- If T1 and T2 write the same cell, one or both will abort

Impacts of Percolator as Client Library

- Where to integrate locking?
- Parallel databases
  - Integrate into component that manages disk access
  - Each node grants locks and deny accesses to data it owns
  - Distributed deadlock detection required
- Percolator
  - No convenient place to intercept traffic and assign locks
  - Must explicitly maintain locks
  - Locks must persist through machine failure
  - Lock service must provide high throughput and low latency

Percolator Lock Server

- Lock server requirements
  - Replicated (survive machine failure)
  - Distributed and balanced (to handle load)
  - Write to a persistent data store
- Percolator uses Bigtable to store locks
- Locks
  - Stored in in-memory columns in the same Bigtable database that stores data
  - Lock columns are added to Bigtable rows (c:lock, c:write, c:data, c:notify, c:ack)
  - Lock columns stored in same row as data
  - Uses Bigtable row transactions to read and modify locks while reading data in a row

Transaction Protocol - Set

- Transaction constructor asks timestamp oracle for start timestamp
- Calls to Set (updates) are buffered until commit time
- Commit Protocol: two-phase commit

Transaction Protocol – Two Phase Commit

- Phase I “prewrite”
  - Attempt to lock all the cells being written (designate one as primary)
  - Aborts if:
    - Sees write record after it’s start timestamp (avoids write-write conflicts)
    - Sees another lock at any timestamp (possibly abort unnecessarily if xact is slow releasing locks, but that is considered unlikely)
  - If no conflict, write lock and data to cell at start timestamp
- If no cells conflict -> Phase II
  - Client obtains commit timestamp from timestamp oracle
  - For each cell (starting with primary) – replace lock with write record - makes write visible to readers
  - Once primary write is visible transaction is committed

Transaction Protocol - Get

- Locks are read by Get requests, but Get Requests do not acquire locks
- Get()
  - Checks for lock with timestamp before start timestamp
  - If lock is present, must wait
  - If no conflicting lock, read latest write record and return data
  - Note that no read locks are required
- Comment: Get must return all committed writes before the xact's start timestamp
- Comment: Transactions on different machines interact through row transactions on Bigtable tablet servers

Transaction Protocol - Example

key	*bal:data*	*bal:lock*	*bal:write*
Bob	*6:* *5:* $10	*6:* *5:*	*6:* data @ 5 *5:*
Joe	*6:* *5:* $2	*6:* *5:*	*6:* data @ 5 *5:*

Initial state

Bob	*7:***$3** *6:* *5:* $10	*7:* **I am primary** *6:* *5:*	*7:* *6:* data @ 5 *5:*
Joe	*6:* *5:* $2	*6:* *5:*	*6:* data @ 5 *5:*

Locks Bob's acct balance

Bob	*7:* $3 *6:* *5:* $10	*7:* I am primary *6:* *5:*	*7:* *6:* data @ 5 *5:*
Joe	***7:* $9** *6:* *5:* $2	***7:* primary @ Bob.bal** *6:* *5:*	*7:* *6:* data @ 5 *5:*

Locks Joe's acct balance (secondary lock)

Transaction Protocol - Example

Bob	*8:* *7:* $3 *6:* *5:* $10	*8:* ***7:*** *6:* *5:*	*8:* **data @ 7** *7:* *6:* data @ 5 *5:*
Joe	*7:* $9 *6:* *5:* $2	*7:* primary @ Bob.bal *6:* *5:*	*7:* *6:*data @ 5 *5:*

Commit point, erase primary lock and creates a write record, balance $3 now visible to readers

Bob	*8:* *7:* $3 *6:* *5:* $10	*8:* *7:* *6:* *5:*	*8:* data @ 7 *7:* *6:* data @ 5 *5:*
Joe	*8:* *7:* $9 *6:* *5:*$2	*8:* ***7:*** *6:* *5:*	***8:* data @ 7** *7:* *6:* data @ 5 *5:*

Delete lock on Joe's balance and writes Joe's balance

Percolator: Columns

Column	*Use*
c:lock	An uncommitted transaction is writing this cell; contains the location of primary lock
c:write	Committed data present; stores the Bigtable timestamp of the data
c:data	Stores the data itself
c:notify	Hint: observers may need to run
c:ack_O	Observer "O" has run ; stores start timestamp of successful last run

Client Failure

- Percolator client fails during commit -> locks will be left behind
- Locks must be cleaned up so future xacts do not hang indefinitely
- Lazy Cleanup of Locks
  - Wait until transaction A encounters a conflicting lock to clean up
  - If A encounters locks left behind by B, may erase B's locks
  - Designation of primary is used to avoid A cleaning up B's transactions if B is just slow (but not failed)
  - Performing cleanup or commit must modify the primary lock
    - B's primary lock in this example
    - B must check primary & replace with write to commit
    - A must check for primary & make sure it exists to erase any of B's locks (what if primary is missing??)

Client Failure II

- Percolator client fails during Phase II
  - Transaction has committed (primary data is visible to other readers)
  - Must perform roll-forward in such cases
  - Transaction (A) that encounters locks can distinguish by seeing if the primary lock (from xact B) exists or has been replaced by a write record
  - Stranded lock is replaced with a write record

Client Failure - Notes

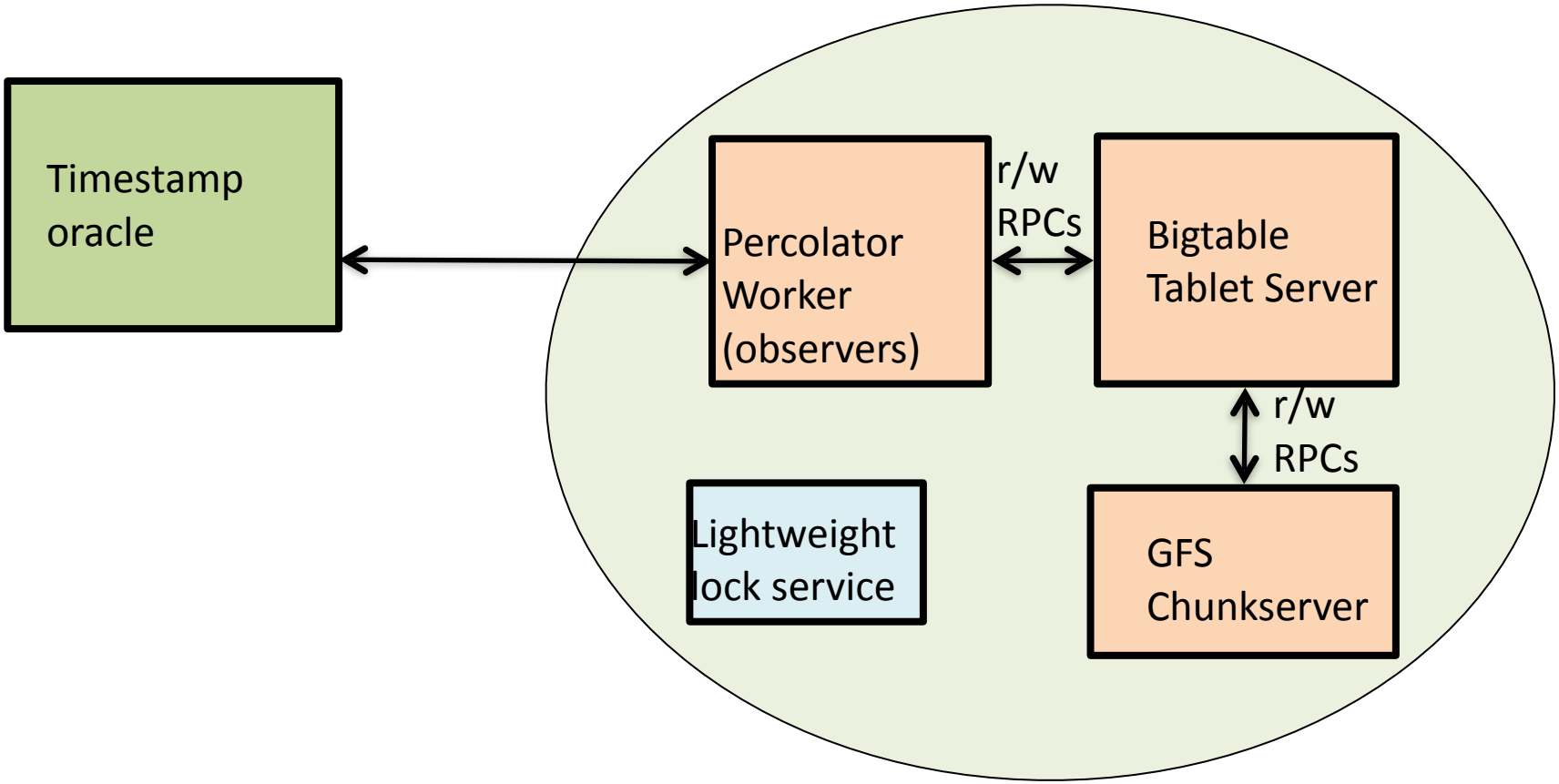
- Safe, but undesirable, to clean up locks held by live clients (performance penalty)
- Locks cleaned up when belong to dead, stuck worker
- Running workers write tokens into chubby lock service, tokens serve as liveness indicator
- Additionally write wall time into lock – if lock contains too-old wall time, will be cleaned up (even if liveness token is valid)
- Wall time updated during committing

Timestamps

- Timestamp oracle hands out timestamps in strictly increasing order
- Every xact contacts timestamp oracle twice, so must scale well
  - Allocates range of timestamps, writing highest allocated timestamp to stable storage
  - Satisfies requests from memory
  - Restart -> may skip timestamp, but won't to backwards
- Percolator worker batches timestamp requests across transactions (saves RPCs)
- 2 million timestamps per second served from a single machine

Notifications

- Observers
  - Written by users
  - Observers are triggered to run by changes to the table
  - All observers linked into the Percolator worker binary
  - Observers register a function and set of columns with Percolator – percolator invokes function after data is written to a column
- Observers complete a task and create more work by writing to a table
  - MapReduce runs loader transactions to trigger Percolator
  - Triggers Document processor (parse, extract) -> Document processor triggers Clustering -> Clustering triggers Export


Timestamp oracle
Percolator Worker (observers)
r/w RPCs
Bigtable Tablet Server
r/w RPCs
Lightweight lock service
GFS Chunkserver

Notifications – Notes

- Triggered observer runs in a separate transaction from triggering write
- Focus is incremental computation (not data integrity)
- Avoid multiple observers on one column
- At most one observer's transaction will commit for each change of an observed column
- But: multiple writes to an observed column may cause observer to be invoked only once (message collapsing)

Notifications - Implementation

- Each observed column has related “ack” column
  - Contains the latest start timestamp at which the observer ran
  - If observed column written after last ack, run observer, else do not run
- Efficiently find dirty cells with observers that need to be run
  - Notifications are rare
- Solution: “notify” Bigtable column
  - One entry for each dirty cell, write a notify cell when an observed cell is written – workers – distributed scan over notify column
  - Notify column is a hint
  - Notify stored as separate locality group (vertical partitioning for improved read performance)
- Issue: two observer – one row (solution: lightweight locks)
- Issue: bus bunching (solution: teleporting)

Discussion

- Percolator – ~50 Bigtable ops / document vs. MR – large read to GFS to obtain data for 100s of web pages
- Percolator – large # RPCs

- Added read-modify-write in a single RPC to Bigtable API
- Collect lock operations into batches – delays lock for several seconds
  - Adds a few seconds to latency
  - Increases time window for conflicts, but environment is low-contention
- Batch read operations
- Prefetching (reduces Bigtable reads by factor of 10)

Thread-per-Request

- Decision: API calls blocking – run thousands of threads / machine to provide parallelism & CPU utilization
- Decision: use thread-per-request model
- Thread-per-request - positives
  - Simplified application code
  - Bundling state for each data fetch from the table -> complicate development
  - Crash debugging simplified – meaningful stack traces
- Thread-per-request - negatives
  - Potential race conditions (less than expected)
  - Linux kernel high thread count scalability issues (hacked around it)

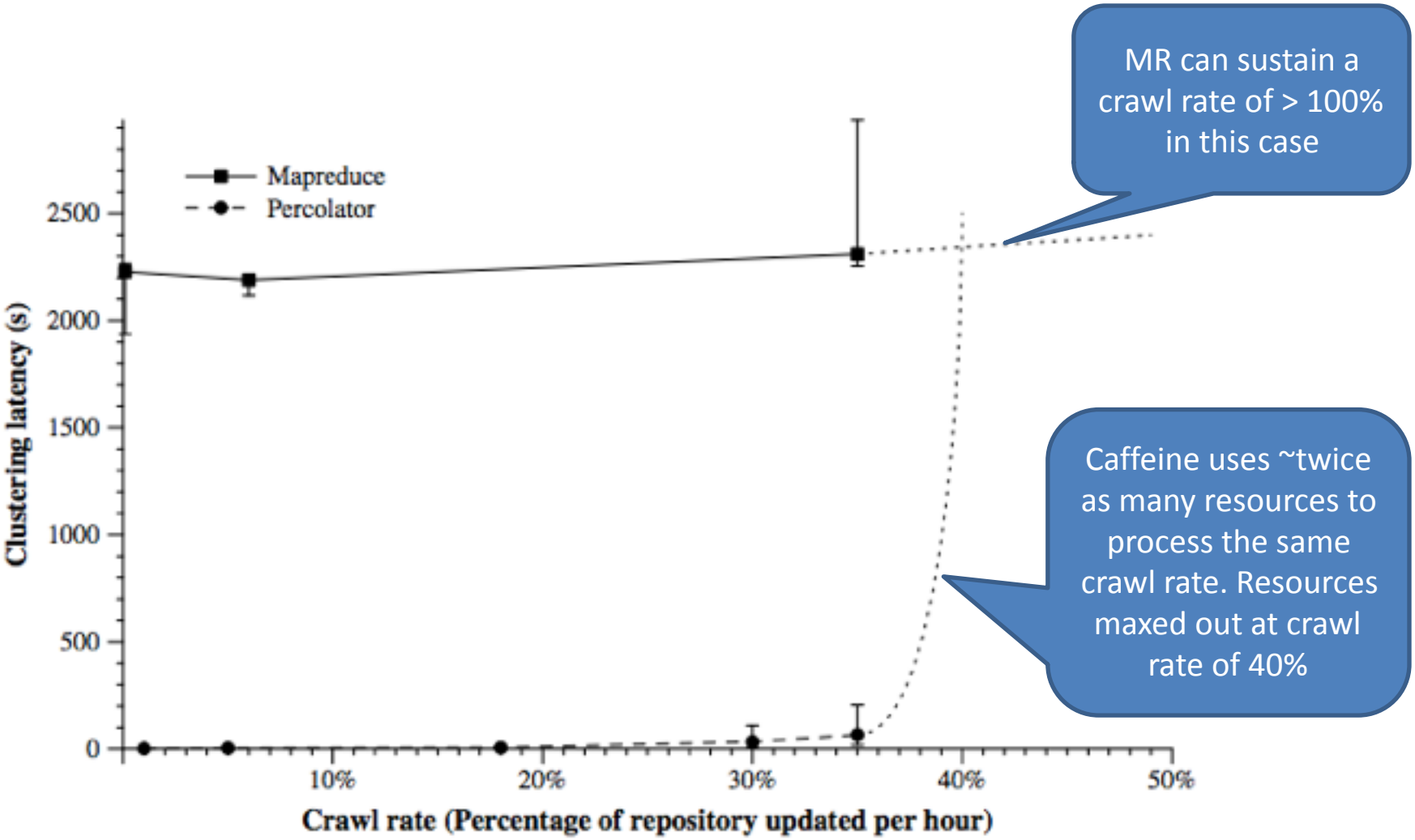
Engineering Issues

- Percolator uses more resources to process a fixed amount of data than a traditional DBMS – cost of scalability
- Percolator has less latency, but more resources than MapReduce
- Questions:
  - "How much of an efficiency loss is too much to pay for the ability to add capacity needlessly simply by purchasing more machines?"
  - "How does one trade off the reduction in development time provided by a layered system against the corresponding decrease in efficiency?"

Evaluation – Converting From Map Reduce

- Converted Google "base" index updates to Percolator
- MR – crawled documents – fed those + existing documents thorough 100 MapReduces
  - 2-3 days to index each document before could be returned as search result
- Caffeine (based on Percolator)
  - Same # documents, median document processed 100x faster
  - Adding a new clustering phase -> additional lookup vs. extra repository scan
  - Approximately 10 observers (multiple clustering phases in one xact vs. 100 MRs)
  - Essentially immune to stragglers (stragglers big issue in MR system)
- MR – each of the 100 MRs needed to be configured and could fail individually
  - Newer system easier to operate

Clustering over Synthetic Benchmark



Clusters new docs against billion-document repository – three clustering keys (avg 3.3 docs/cluster)

Cost of Transactional Semantics

	Bigtable	Percolator	Relative
Read/s	15513	14590	0.94
Write/s	31003	7232	0.23

Comparison of Percolator to “raw “ Bigtable using microbenchmarks on a single tablet server.

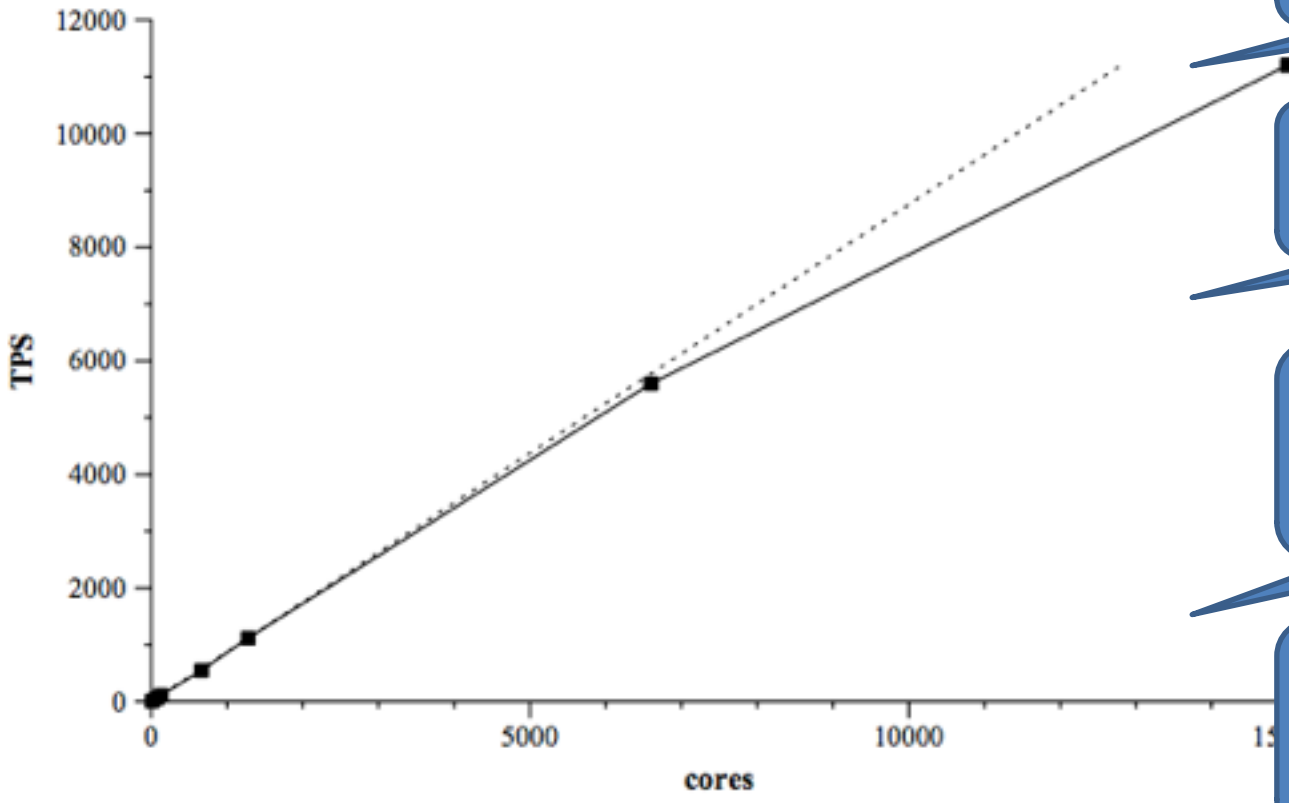
Data in tablet server’s cache and batching optimizations disabled.

Write single cell, then commit. Worst-case for Percolator.

Note reads more expensive than writes – accounts for much of the Write difference (Percolator Write is Bigtable Read, Write, Write)

Timestamp fetching overhead not measured.

TPC-E Benchmark (OLTP)



Percolator: 11,200 TPS on 15,000 cores (cheaper cores)

DBMS: 3,183 TPS on 64 Intel Nehalem cores

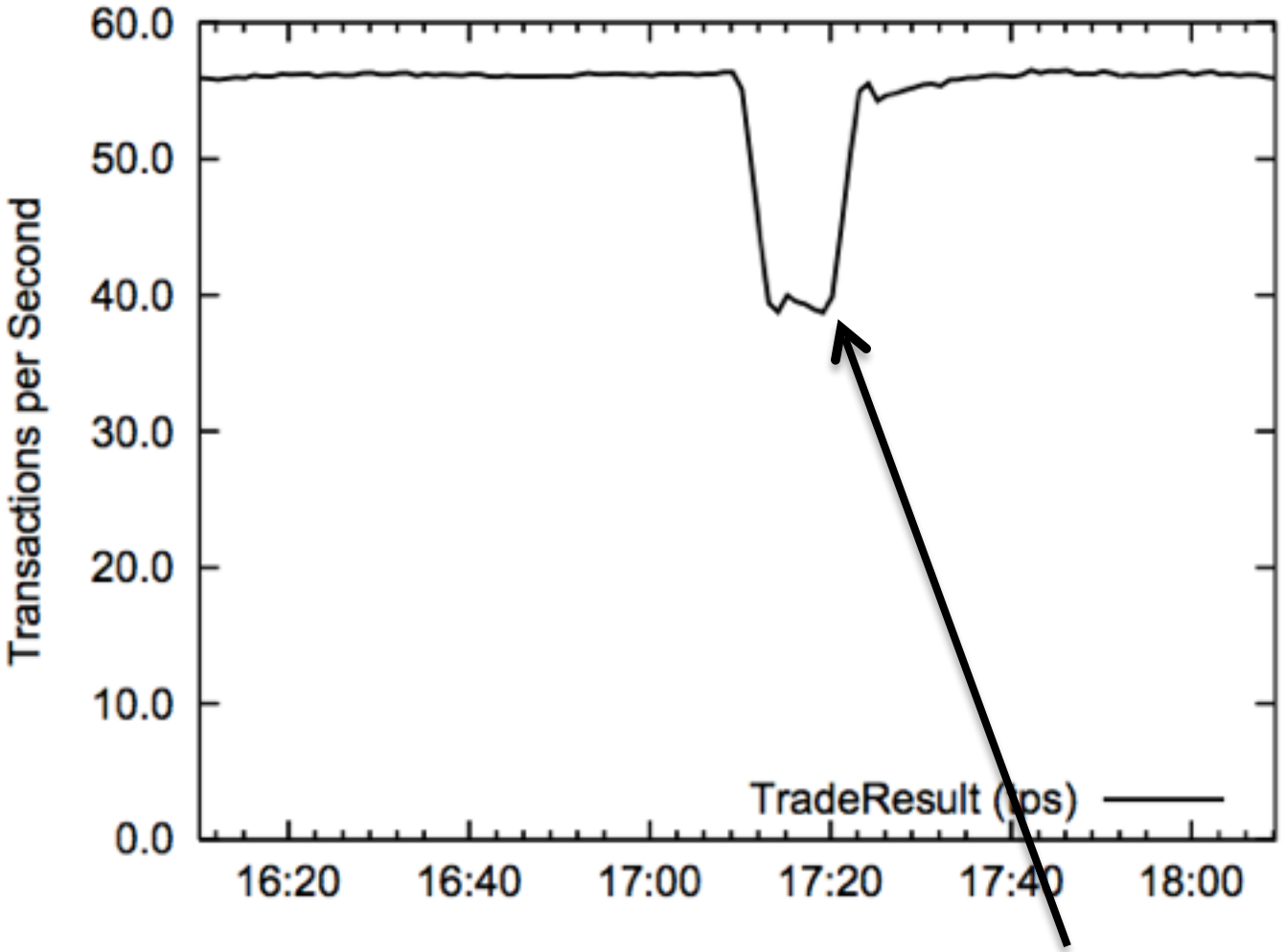
30x more CPU per transaction than the benchmark system, in this test

This increase is cost of layering, but they get scalability…

Did not measure TPC-E latency

Near-linear scaling, CPU cores are limiting resource

Recovery after Failure



Comparison Percolator vs DBMS & Bigtable

- Percolator achieved its goal of reducing the latency of indexing a single document
- Percolator lacks query language
- Percolator lacks full set of relational operators (i.e. join)
- Scales better than existing parallel databases
- Deals better with failed machines than existing database
- Percolator: Commodity machines, shared-nothing hardware, communication is explicit RPCs only