Large-scale Incremental Processing Using Distributed Transactions and Notifications

Daniel Peng and Frank Dabek

Google, Inc

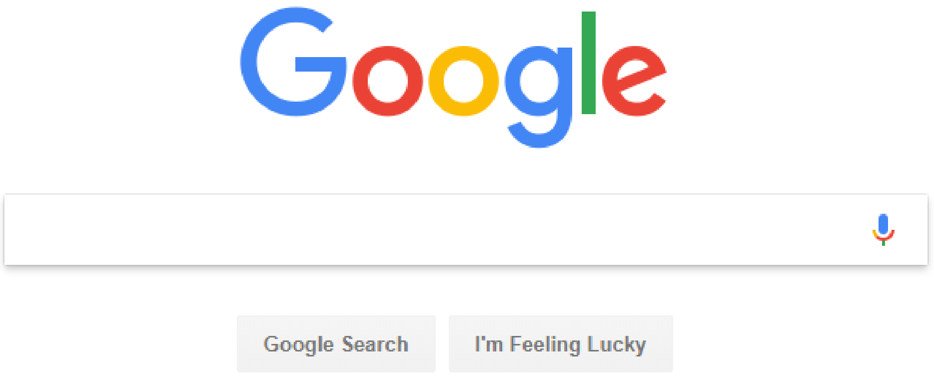
OSDI 2010

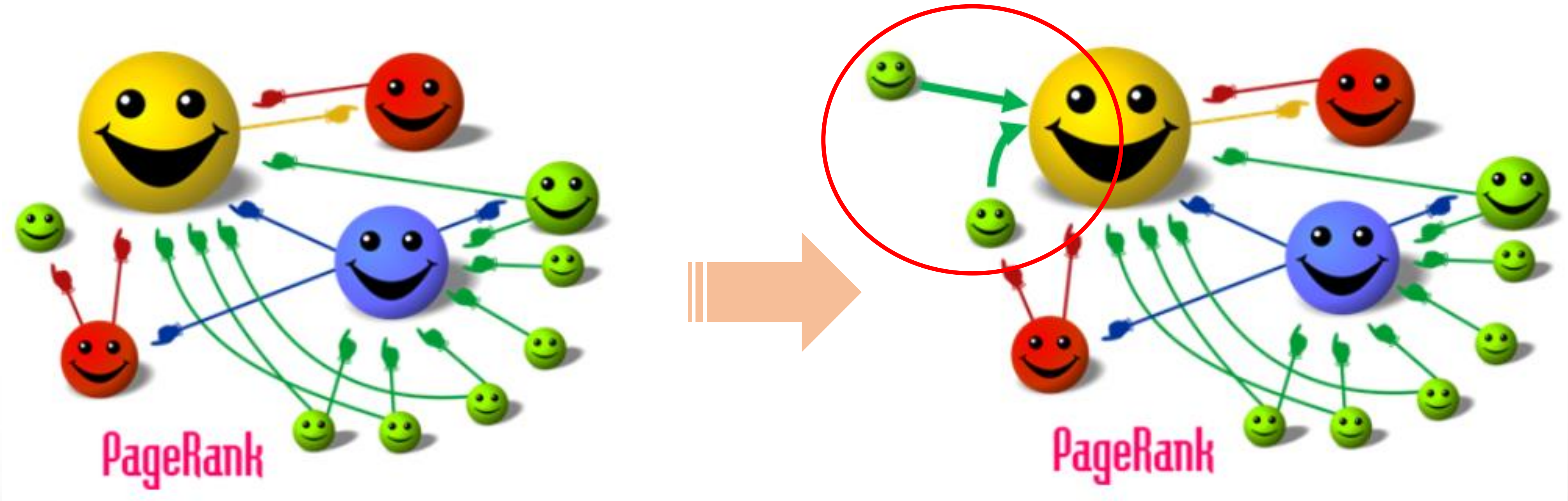
Presentation: 陈渤、黄思、秦培杰
Demo: 丁峰、徐华韬

Contents

Contents

Motivation





Motivation

How to update the index after recrawling some small portion of the web?

Before **MapReduce**

- Process the entire repository, not just the new documents
- Discard the work done in earlier runs
- Make latency proportional to the size of the repository, rather than the size of an update

Batch-based indexing system

After **Percolator**

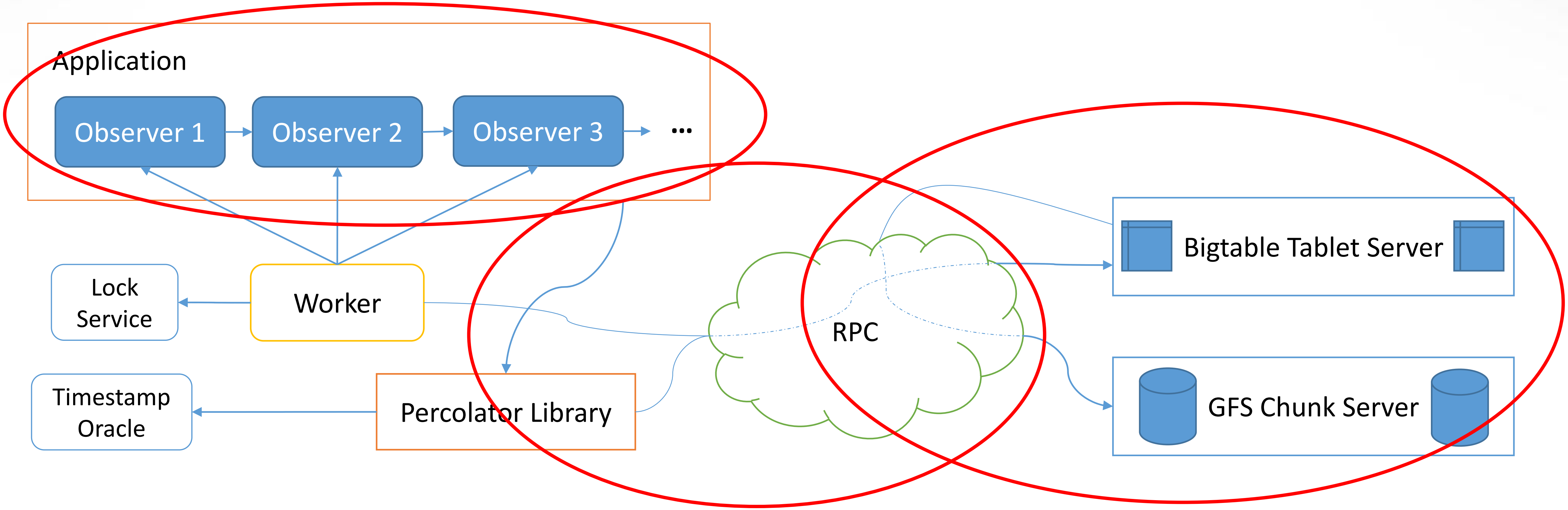
- An incremental processing system
- Avoid redoing work that has already been done

Incremental-based indexing system

✓ **Reduce the average document processing latency by a factor of 100.**
✓ **Reduce the average age of documents in Google search results by 50%.**

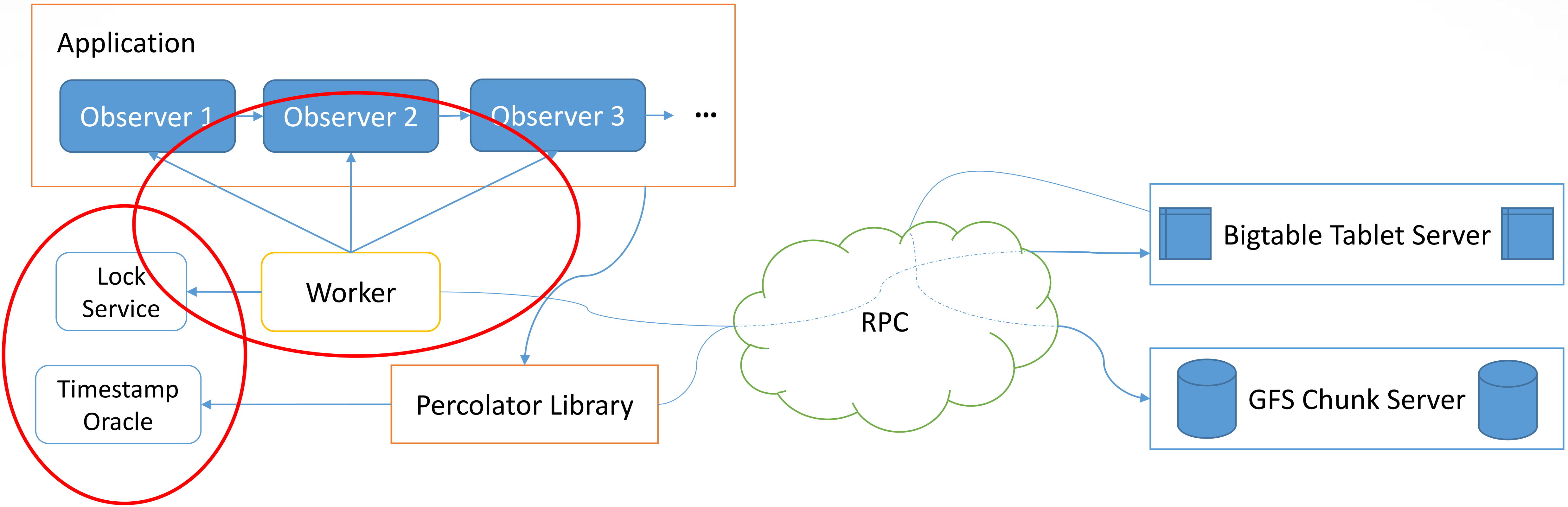
Contents

System design - Overview



- Percolator is built on top of the Bigtable distributed storage system.
- The observer applications … write RPCs to Bigtable tablet servers.
- Each observer completes a task and creates more work for “downstream” observers by writing to the table.

System design - Overview



- All observers are linked into the Percolator worker
- The system also depends on two small services: the timestamp oracle and the lightweight lock service.
- Worker scans the Bigtable for changed columns (“notifications”) and invokes the corresponding observers

System design - Bigtable

Feature:

- Bigtable presents a multi-dimensional sorted map to users:

  keys are (row, column, timestamp) tuples

- Bigtable provides lookup, update operations and transactions on each row.

Weakness:

- Bigtable does not provide multi-row transactions.

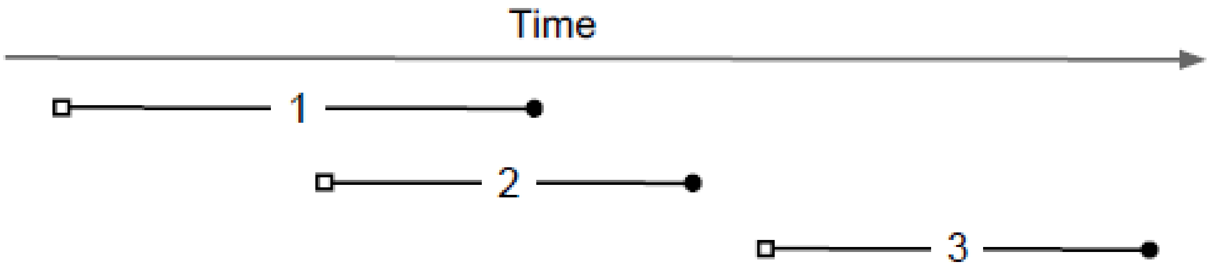
Solution:

- Percolator provides APIs similar to Bigtable's API.
- Percolator supports multirow transactions and designs the observer framework.

System design - Transactions

Feature:

- Provides cross-row, cross-table transactions with ACID snapshot-isolation semantics (protects against write-write conflicts).



System design - Transactions

Feature:

- Provides cross-row, cross-table transactions with ACID snapshot-isolation semantics (protects against write-write conflicts).
- Stores multiple versions of each data item using Bigtable's timestamp dimension.
- Two phase commit, strong consistency.
- Stores locks in special in-memory columns in the same Bigtable.

System design - Transactions

Example: Tao transfer $7 to Feng

Bigtable

```
Transaction() : start_ts_(oracle.GetTimestamp()) {}
```

Phase1

```
bool Prewrite(Write w, Write primary) {
  Column c = w.col;
  bigtable::Txn T = bigtable::StartRowTransaction(w.row);

  // Abort on writes after our start timestamp . . .
  if (T.Read(w.row, c+"write", [start_ts_, ∞])) return false;
  // . . . or locks at any timestamp.
  if (T.Read(w.row, c+"lock", [0, ∞])) return false;

  T.Write(w.row, c+"data", start_ts_, w.value);
  T.Write(w.row, c+"lock", start_ts_,
          {primary.row, primary.col}); // The primary's location.
  return T.Commit();
}
```

Key	Bal: data	Bal: lock	Bal: write
Tao			
Feng			

System design - Transactions

Example: Tao transfer $7 to Feng

First get timestamp

```
Transaction() : start_ts_(oracle.GetTimestamp()) {}
```

Phase1

```
bool Prewrite(Write w, Write primary) {
  Column c = w.col;
  bigtable::Txn T = bigtable::StartRowTransaction(w.row);

  // Abort on writes after our start timestamp . . .
  if (T.Read(w.row, c+"write", [start_ts_, ∞])) return false;
  // . . . or locks at any timestamp.
  if (T.Read(w.row, c+"lock", [0, ∞])) return false;

  T.Write(w.row, c+"data", start_ts_, w.value);
  T.Write(w.row, c+"lock", start_ts_,
          {primary.row, primary.col}); // The primary's location.
  return T.Commit();
}
```

Bigtable

Key	Bal: data	Bal: lock	Bal: write
Tao	6: 5:$10	6: 5:	6:data@5 5:
Feng	6: 5:$2	6: 5:	6:data@5 5:

System design - Transactions

Example: Tao transfer $7 to Feng

Bigtable

```
Transaction() : start_ts_(oracle.GetTimestamp()) {}
```

Phase1

```
bool Prewrite(Write w, Write primary) {
  Column c = w.col;
  bigtable::Txn T = bigtable::StartRowTransaction(w.row);

  // Abort on writes after our start timestamp . . .
  if (T.Read(w.row, c+"write", [start_ts_, ∞])) return false;
  // . . . or locks at any timestamp.
  if (T.Read(w.row, c+"lock", [0, ∞])) return false;

  T.Write(w.row, c+"data", start_ts_, w.value);
  T.Write(w.row, c+"lock", start_ts_,
          {primary.row, primary.col}); // The primary's location.
  return T.Commit();
}
```

Key	Bal: data	Bal: lock	Bal: write
Tao	6: 5:$10	6: 5:	6:data@5 5:
Feng	6: 5:$2	6: 5:	6:data@5 5:

System design - Transactions

Example: Tao transfer $7 to Feng

Bigtable

```
Transaction() : start_ts_(oracle.GetTimestamp()) {}
```

Phase1

```
bool Prewrite(Write w, Write primary) {
  Column c = w.col;
  bigtable::Txn T = bigtable::StartRowTransaction(w.row);

  // Abort on writes after our start timestamp . . .
  if (T.Read(w.row, c+"write", [start_ts_, ∞])) return false;
  // . . . or locks at any timestamp.
  if (T.Read(w.row, c+"lock", [0, ∞])) return false;

  T.Write(w.row, c+"data", start_ts_, w.value);
  T.Write(w.row, c+"lock", start_ts_,
          {primary.row, primary.col}); // The primary's location.
  return T.Commit();
}
```

detect conflict

Key	Bal: data	Bal: lock	Bal: write
Tao	6: 5:$10	6: 5:	6:data@5 5:
Feng	6: 5:$2	6: 5:	6:data@5 5:

System design - Transactions

Example: Tao transfer $7 to Feng

Bigtable

```
Transaction() : start_ts_(oracle.GetTimestamp()) {}
```

Phase1

```
bool Prewrite(Write w, Write primary) {
  Column c = w.col;
  bigtable::Txn T = bigtable::StartRowTransaction(w.row);

  // Abort on writes after our start timestamp . . .
  if (T.Read(w.row, c+"write", [start_ts_, ∞])) return false;
  // . . . or locks at any timestamp.
  if (T.Read(w.row, c+"lock", [0, ∞])) return false;

  T.Write(w.row, c+"data", start_ts_, w.value);
  T.Write(w.row, c+"lock", start_ts_,
          {primary.row, primary.col}); // The primary's location.
  return T.Commit();
}
```

Key	Bal: data	Bal: lock	Bal: write
Tao	7:$3 6: 5:$10	7:primary 6: 5:	7: 6:data@5 5:
Feng	6: 5:$2	6: 5:	6:data@5 5:

declare primary lock

System design - Transactions

Example: Tao transfer $7 to Feng

Bigtable

```
Transaction() : start_ts_(oracle.GetTimestamp()) {}
```

Phase1

```
bool Prewrite(Write w, Write primary) {
  Column c = w.col;
  bigtable::Txn T = bigtable::StartRowTransaction(w.row);

  // Abort on writes after our start timestamp . . .
  if (T.Read(w.row, c+"write", [start_ts_, ∞])) return false;
  // . . . or locks at any timestamp.
  if (T.Read(w.row, c+"lock", [0, ∞])) return false;

  T.Write(w.row, c+"data", start_ts_, w.value);
  T.Write(w.row, c+"lock", start_ts_,
          {primary.row, primary.col}); // The primary's location.
  return T.Commit();
}
```

Key	Bal: data	Bal: lock	Bal: write
Tao	7:$3 6: 5:$10	7:primary 6: 5:	7: 6:data@5 5:
Feng	7:$9 6: 5:$2	7:primary@Tao.bal 6: 5:	7: 6:data@5 5:

secondary lock

System design - Transactions

Example: Tao transfer $7 to Feng

Second get timestamp

Bigtable

Phase2

```
int commit_ts = oracle_.GetTimestamp();

// Commit primary first.
Write p = primary;
bigtable::Txn T = bigtable::StartRowTransaction(p.row);
if (!T.Read(p.row, p.col+"lock", [start_ts_, start_ts_]))
    return false; // aborted while working
T.Write(p.row, p.col+"write", commit_ts, start_ts_);
T.Erase(p.row, p.col+"lock", commit_ts);
if (!T.Commit()) return false; // commit point

// Second phase: write out write records for secondary cells.
for (Write w : secondaries) {
    bigtable::Write(w.row, w.col+"write", commit_ts, start_ts_);
    bigtable::Erase(w.row, w.col+"lock", commit_ts);
}
```

Key	Bal: data	Bal: lock	Bal: write
Tao	8: 7:$3 6: 5:$10	8: 7: primary 6: 5:	8:data@7 7: 6:data@5 5:
Feng	7:$9 6: 5:$2	7:primary@Tao.bal 6: 5:	7: 6:data@5 5:

System design - Transactions

Example: Tao transfer $7 to Feng

Phase2

```
int commit_ts = oracle_.GetTimestamp();

// Commit primary first.
Write p = primary;
bigtable::Txn T = bigtable::StartRowTransaction(p.row);
if (!T.Read(p.row, p.col+"lock", [start_ts_, start_ts_]))
   return false; // aborted while working
T.Write(p.row, p.col+"write", commit_ts, start_ts_);
T.Erase(p.row, p.col+"lock", commit_ts);
if (!T.Commit()) return false; // commit point

// Second phase: write out write records for secondary cells.
for (Write w : secondaries) {
   bigtable::Write(w.row, w.col+"write", commit_ts, start_ts_);
   bigtable::Erase(w.row, w.col+"lock", commit_ts);
}
```

Bigtable

Key	Bal: data	Bal: lock	Bal: write
Tao	8: 7:$3 6: 5:$10	8: 7: 6: 5:	8:data@7 7: 6:data@5 5:
Feng	8: 7:$9 6: 5:$2	8: 7: primary@Tao.bal 6: 5:	8:data@7 7: 6:data@5 5:

Release all lock

Release all lock

?

System design - Transactions

Solution

- Percolator takes a lazy approach to cleanup:

  when a transaction A encounters a conflicting lock left behind by transaction B, A may determine that B has failed and erase its locks.
- This modification is performed under a Bigtable row transaction, so only one of the cleanup or commit operations will succeed.
- Check the primary lock to decide to cleanup or commit.
  - Write record: commit, roll forward transaction B.
  - Lock: cleanup, safely erase the lock and roll back transaction B.

System design - Timestamps

Feature:

- The timestamp oracle is a server that hands out timestamps in strictly increasing order.
- Timestamp requests are batched to decrease RPC’s.
- For failure recovery, the timestamp oracle needs to write the highest allocated timestamp to disk before responding to a request.

System design - Notifications

Percolator applications are structured as a series of observers:

Each observer registers a function on a set of columns:

- Executed when any row in that column is written.

Use notifications to track and trigger observers:

- **Dirty cell**: cell with observer that need to be run
- **Message collapsing**: only run once if multiple writes to an observed column

System design - Notifications

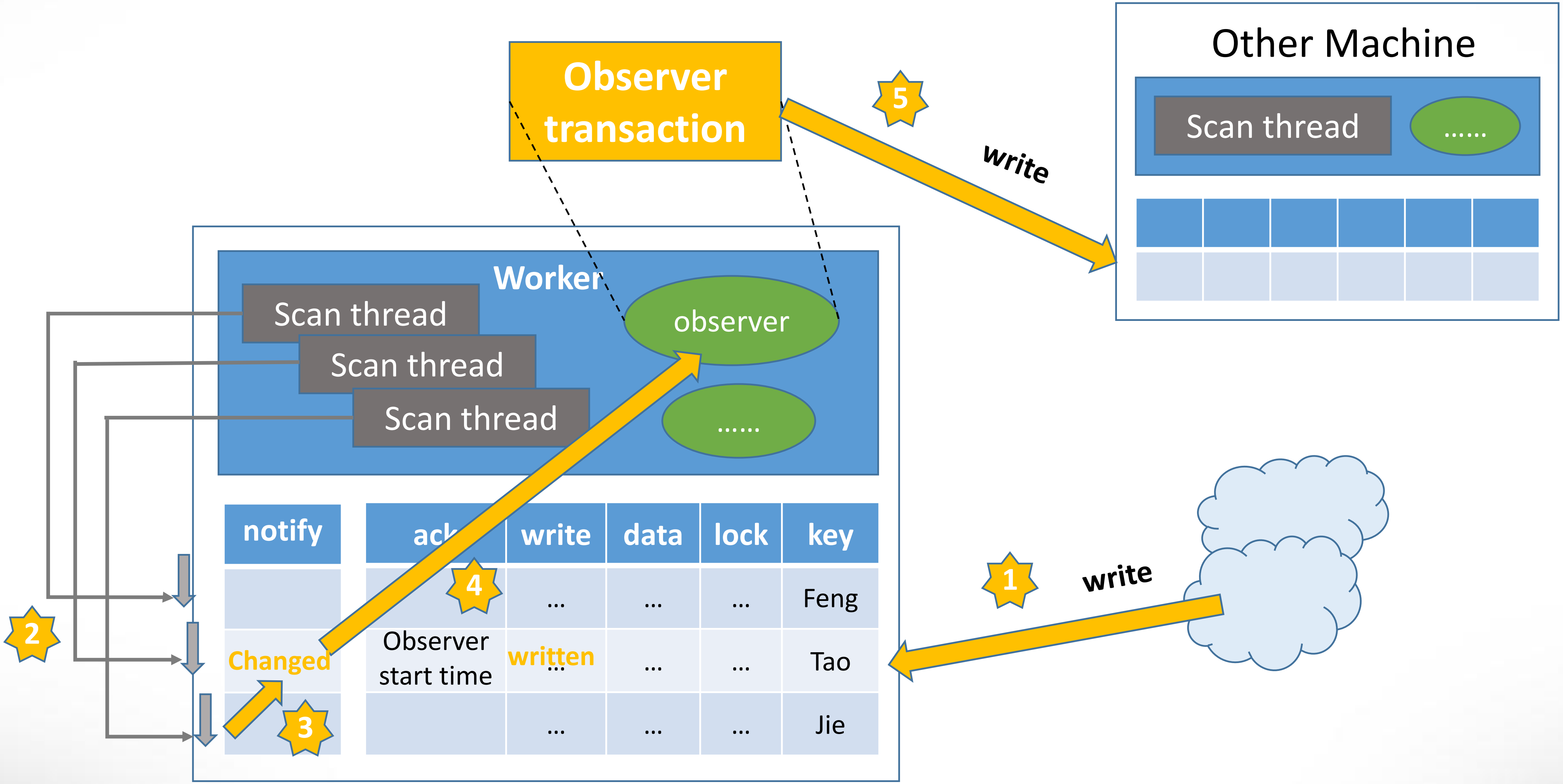
"notify" Bigtable column: an entry for each dirty cell

- Set notify cell after write and remove it after triggering the observer.
- Each worker with several threads scans the notify Bigtable column efficiently.
- Random portion – bus clumping. A lock or a new random location.

"acknowledgment" column: the latest start TS of last observer

- Read the observed column and its ack column before triggering.
- If $TS_{obs} > TS_{ack}$, run the observer and set the start TS to the ack column. If not, do not run.
- At most one observer will commit for each notification.

System design - Notifications



Contents

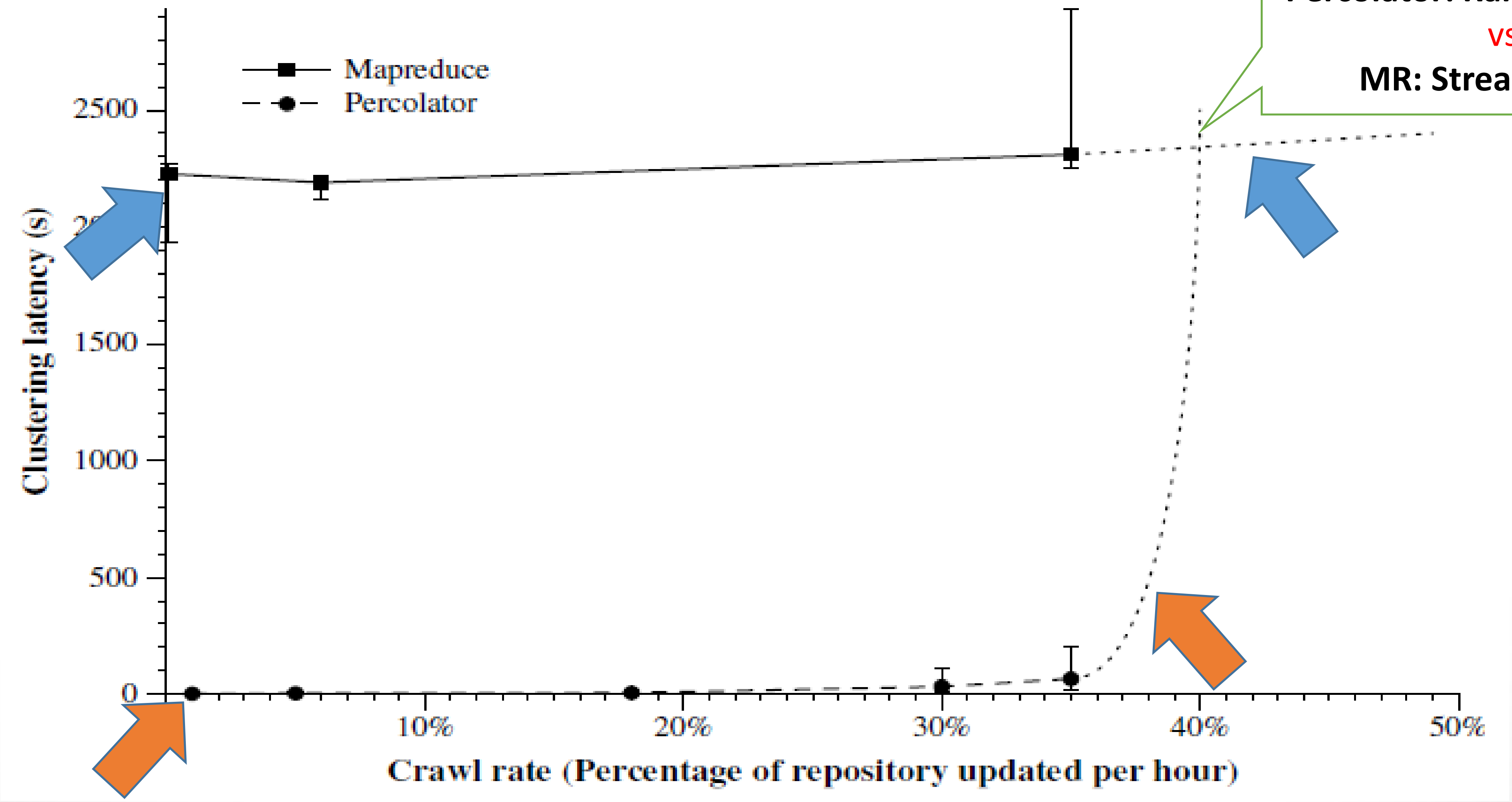
Evaluation

- MR vs. Percolator: performance

Percolator: Random lookups

vs

MR: Streaming I/O



Evaluation

- Latency
  - 100x faster than the MapReduce system
- Simplification
  - The number of observers in Percolator: 10
  - The number of MapReduces in the previous system: 100
- Tradeoff
  - 3x larger repository
  - 2x resources
- Easier to operate
  - Far fewer moving parts: tablet servers, Percolator workers, chunkservers
  - Each MapReduce needs to be individually configured and may fail independently

Evaluation

- Bigtable vs. Percolator: performance

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

Write： 4x overhead, extra operations beyond the single write

- a read to check for locks
- a write to add the lock
- a second write to remove the lock record

Read： looks at metadata columns in addition to data columns

Contents

Conclusion

- Percolator provides two main abstractions

✓Transactions: cross-row, cross-table transactions with ACID snapshot-isolation semantics

✓Observers: triggered by notification

- Percolator now building the "Caffeine" web-search index

✓50% fresher results

✓3x larger repository

Thank you!