Large-scale Incremental Processing Using Distributed Transactions and Notifications

Daniel Peng and Frank Dabek
dpeng@google.com, fdabek@google.com
Google, Inc.

Abstract

Updating an index of the web as documents are crawled requires continuously transforming a large repository of existing documents as new documents arrive. This task is one example of a class of data processing tasks that transform a large repository of data via small, independent mutations. These tasks lie in a gap between the capabilities of existing infrastructure. Databases do not meet the storage or throughput requirements of these tasks: Google's indexing system stores tens of petabytes of data and processes billions of updates per day on thousands of machines. MapReduce and other batch-processing systems cannot process small updates individually as they rely on creating large batches for efficiency.

We have built Percolator, a system for incrementally processing updates to a large data set, and deployed it to create the Google web search index. By replacing a batch-based indexing system with an indexing system based on incremental processing using Percolator, we process the same number of documents per day, while reducing the average age of documents in Google search results by 50%.

1 Introduction

Consider the task of building an index of the web that can be used to answer search queries. The indexing system starts by crawling every page on the web and processing them while maintaining a set of invariants on the index. For example, if the same content is crawled under multiple URLs, only the URL with the highest PageRank [28] appears in the index. Each link is also inverted so that the anchor text from each outgoing link is attached to the page the link points to. Link inversion must work across duplicates: links to a duplicate of a page should be forwarded to the highest PageRank duplicate if necessary.

This is a bulk-processing task that can be expressed as a series of MapReduce [13] operations: one for clustering duplicates, one for link inversion, etc. It's easy to maintain invariants since MapReduce limits the parallelism of the computation; all documents finish one processing step before starting the next. For example, when the indexing system is writing inverted links to the current highest-PageRank URL, we need not worry about its PageRank concurrently changing; a previous MapReduce step has already determined its PageRank.

Now, consider how to update that index after recrawling some small portion of the web. It's not sufficient to run the MapReduces over just the new pages since, for example, there are links between the new pages and the rest of the web. The MapReduces must be run again over the entire repository, that is, over both the new pages and the old pages. Given enough computing resources, MapReduce's scalability makes this approach feasible, and, in fact, Google's web search index was produced in this way prior to the work described here. However, reprocessing the entire web discards the work done in earlier runs and makes latency proportional to the size of the repository, rather than the size of an update.

The indexing system could store the repository in a DBMS and update individual documents while using transactions to maintain invariants. However, existing DBMSs can't handle the sheer volume of data: Google's indexing system stores tens of petabytes across thousands of machines [30]. Distributed storage systems like Bigtable [9] can scale to the size of our repository but don't provide tools to help programmers maintain data invariants in the face of concurrent updates.

An ideal data processing system for the task of maintaining the web search index would be optimized for *incremental processing*; that is, it would allow us to maintain a very large repository of documents and update it efficiently as each new document was crawled. Given that the system will be processing many small updates concurrently, an ideal system would also provide mechanisms for maintaining invariants despite concurrent updates and for keeping track of which updates have been processed.

The remainder of this paper describes a particular incremental processing system: Percolator. Percolator provides the user with random access to a multi-PB repository. Random access allows us to process documents in-

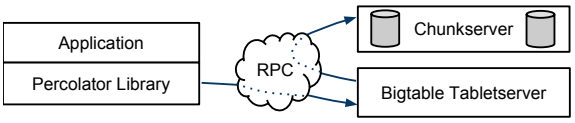


Figure 1: Percolator and its dependencies

dividually, avoiding the global scans of the repository that MapReduce requires. To achieve high throughput, many threads on many machines need to transform the repository concurrently, so Percolator provides ACID-compliant transactions to make it easier for programmers to reason about the state of the repository; we currently implement snapshot isolation semantics [5].

In addition to reasoning about concurrency, programmers of an incremental system need to keep track of the state of the incremental computation. To assist them in this task, Percolator provides observers: pieces of code that are invoked by the system whenever a user-specified column changes. Percolator applications are structured as a series of observers; each observer completes a task and creates more work for "downstream" observers by writing to the table. An external process triggers the first observer in the chain by writing initial data into the table.

Percolator was built specifically for incremental processing and is not intended to supplant existing solutions for most data processing tasks. Computations where the result can't be broken down into small updates (sorting a file, for example) are better handled by MapReduce. Also, the computation should have strong consistency requirements; otherwise, Bigtable is sufficient. Finally, the computation should be very large in some dimension (total data size, CPU required for transformation, etc.); smaller computations not suited to MapReduce or Bigtable can be handled by traditional DBMSs.

Within Google, the primary application of Percolator is preparing web pages for inclusion in the live web search index. By converting the indexing system to an incremental system, we are able to process individual documents as they are crawled. This reduced the average document processing latency by a factor of 100, and the average age of a document appearing in a search result dropped by nearly 50 percent (the age of a search result includes delays other than indexing such as the time between a document being changed and being crawled). The system has also been used to render pages into images; Percolator tracks the relationship between web pages and the resources they depend on, so pages can be reprocessed when any depended-upon resources change.

2 Design

Percolator provides two main abstractions for performing incremental processing at large scale: ACID transactions over a random-access repository and observers, a way to organize an incremental computation.

A Percolator system consists of three binaries that run on every machine in the cluster: a Percolator worker, a Bigtable [9] tablet server, and a GFS [20] chunkserver. All observers are linked into the Percolator worker, which scans the Bigtable for changed columns ("notifications") and invokes the corresponding observers as a function call in the worker process. The observers perform transactions by sending read/write RPCs to Bigtable tablet servers, which in turn send read/write RPCs to GFS chunkservers. The system also depends on two small services: the timestamp oracle and the lightweight lock service. The timestamp oracle provides strictly increasing timestamps: a property required for correct operation of the snapshot isolation protocol. Workers use the lightweight lock service to make the search for dirty notifications more efficient.

From the programmer's perspective, a Percolator repository consists of a small number of tables. Each table is a collection of "cells" indexed by row and column. Each cell contains a value: an uninterpreted array of bytes. (Internally, to support snapshot isolation, we represent each cell as a series of values indexed by timestamp.)

The design of Percolator was influenced by the requirement to run at massive scales and the lack of a requirement for extremely low latency. Relaxed latency requirements let us take, for example, a lazy approach to cleaning up locks left behind by transactions running on failed machines. This lazy, simple-to-implement approach potentially delays transaction commit by tens of seconds. This delay would not be acceptable in a DBMS running OLTP tasks, but it is tolerable in an incremental processing system building an index of the web. Percolator has no central location for transaction management; in particular, it lacks a global deadlock detector. This increases the latency of conflicting transactions but allows the system to scale to thousands of machines.

2.1 Bigtable overview

Percolator is built on top of the Bigtable distributed storage system. Bigtable presents a multi-dimensional sorted map to users: keys are (row, column, timestamp) tuples. Bigtable provides lookup and update operations on each row, and Bigtable row transactions enable atomic read-modify-write operations on individual rows. Bigtable handles petabytes of data and runs reliably on large numbers of (unreliable) machines.

A running Bigtable consists of a collection of tablet servers, each of which is responsible for serving several tablets (contiguous regions of the key space). A master coordinates the operation of tablet servers by, for example, directing them to load or unload tablets. A tablet is stored as a collection of read-only files in the Google

SSTable format. SSTables are stored in GFS; Bigtable relies on GFS to preserve data in the event of disk loss. Bigtable allows users to control the performance characteristics of the table by grouping a set of columns into a locality group. The columns in each locality group are stored in their own set of SSTables, which makes scanning them less expensive since the data in other columns need not be scanned.

The decision to build on Bigtable defined the overall shape of Percolator. Percolator maintains the gist of Bigtable's interface: data is organized into Bigtable rows and columns, with Percolator metadata stored alongside in special columns (see Figure 5). Percolator's API closely resembles Bigtable's API: the Percolator library largely consists of Bigtable operations wrapped in Percolator-specific computation. The challenge, then, in implementing Percolator is providing the features that Bigtable does not: multirow transactions and the observer framework.

2.2 Transactions

Percolator provides cross-row, cross-table transactions with ACID snapshot-isolation semantics. Percolator users write their transaction code in an imperative language (currently C++) and mix calls to the Percolator API with their code. Figure 2 shows a simplified version of clustering documents by a hash of their contents. In this example, if Commit() returns false, the transaction has conflicted (in this case, because two URLs with the same content hash were processed simultaneously) and should be retried after a backoff. Calls to Get() and Commit() are blocking; parallelism is achieved by running many transactions simultaneously in a thread pool.

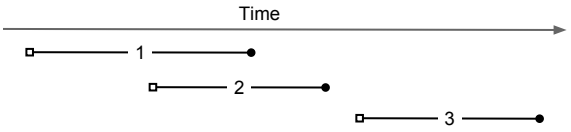
While it is possible to incrementally process data without the benefit of strong transactions, transactions make it more tractable for the user to reason about the state of the system and to avoid the introduction of errors into a long-lived repository. For example, in a transactional web-indexing system the programmer can make assumptions like: the hash of the contents of a document is always consistent with the table that indexes duplicates. Without transactions, an ill-timed crash could result in a permanent error: an entry in the document table that corresponds to no URL in the duplicates table. Transactions also make it easy to build index tables that are always up to date and consistent. Note that both of these examples require transactions that span rows, rather than the single-row transactions that Bigtable already provides.

Percolator stores multiple versions of each data item using Bigtable's timestamp dimension. Multiple versions are required to provide snapshot isolation [5], which presents each transaction with the appearance of reading from a stable snapshot at some timestamp. Writes appear in a different, later, timestamp. Snapshot isolation pro-

```
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();
}
```

Figure 2: Example usage of the Percolator API to perform basic checksum clustering and eliminate documents with the same content.



[t]

Figure 3: Transactions under snapshot isolation perform reads at a start timestamp (represented here by an open square) and writes at a commit timestamp (closed circle). In this example, transaction 2 would not see writes from transaction 1 since transaction 2's start timestamp is before transaction 1's commit timestamp. Transaction 3, however, will see writes from both 1 and 2. Transaction 1 and 2 are running concurrently: if they both write the same cell, at least one will abort.

tects against write-write conflicts: if transactions A and B, running concurrently, write to the same cell, at most one will commit. Snapshot isolation does not provide serializability; in particular, transactions running under snapshot isolation are subject to write skew [5]. The main advantage of snapshot isolation over a serializable protocol is more efficient reads. Because any timestamp represents a consistent snapshot, reading a cell requires only performing a Bigtable lookup at the given timestamp; acquiring locks is not necessary. Figure 3 illustrates the relationship between transactions under snapshot isolation.

Because it is built as a client library accessing Bigtable, rather than controlling access to storage itself, Percolator faces a different set of challenges implementing distributed transactions than traditional PDBMSs. Other parallel databases integrate locking into the system component that manages access to the disk: since each node already mediates access to data on the disk it can grant locks on requests and deny accesses that violate locking requirements.

By contrast, any node in Percolator can (and does) issue requests to directly modify state in Bigtable: there is no convenient place to intercept traffic and assign locks. As a result, Percolator must explicitly maintain locks. Locks must persist in the face of machine failure; if a lock could disappear between the two phases of com-

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:*

1. Initial state: Joe's account contains $2 dollars, Bob's $10.

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:*

2. The transfer transaction begins by locking Bob's account balance by writing the lock column. This lock is the primary for the transaction. The transaction also writes data at its start timestamp, 7.

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:*

3. The transaction now locks Joe's account and writes Joe's new balance (again, at the start timestamp). The lock is a secondary for the transaction and contains a reference to the primary lock (stored in row "Bob," column "bal"); in case this lock is stranded due to a crash, a transaction that wishes to clean up the lock needs the location of the primary to synchronize the cleanup.

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:*

4. The transaction has now reached the commit point: it erases the primary lock and replaces it with a write record at a new timestamp (called the commit timestamp): 8. The write record contains a pointer to the timestamp where the data is stored. Future readers of the column "bal" in row "Bob" will now see the value $3.

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:*

5. The transaction completes by adding write records and deleting locks at the secondary cells. In this case, there is only one secondary: Joe.

Figure 4: This figure shows the Bigtable writes performed by a Percolator transaction that mutates two rows. The transaction transfers 7 dollars from Bob to Joe. Each Percolator column is stored as 3 Bigtable columns: data, write metadata, and lock metadata. Bigtable's timestamp dimension is shown within each cell; 12: "data" indicates that "data" has been written at Bigtable timestamp 12. Newly written data is shown in boldface.

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

Figure 5: The columns in the Bigtable representation of a Percolator column named "c."

mit, the system could mistakenly commit two transactions that should have conflicted. The lock service must provide high throughput; thousands of machines will be requesting locks simultaneously. The lock service should also be low-latency; each Get() operation requires reading locks in addition to data, and we prefer to minimize this latency. Given these requirements, the lock server will need to be replicated (to survive failure), distributed and balanced (to handle load), and write to a persistent data store. Bigtable itself satisfies all of our requirements, and so Percolator stores its locks in special in-memory columns in the same Bigtable that stores data and reads or modifies the locks in a Bigtable row transaction when accessing data in that row.

We'll now consider the transaction protocol in more detail. Figure 6 shows the pseudocode for Percolator transactions, and Figure 4 shows the layout of Percolator data and metadata during the execution of a transaction. These various metadata columns used by the system are described in Figure 5. The transaction's constructor asks the timestamp oracle for a start timestamp (line 6), which determines the consistent snapshot seen by Get(). Calls to Set() are buffered (line 7) until commit time. The basic approach for committing buffered writes is two-phase commit, which is coordinated by the client. Transactions on different machines interact through row transactions on Bigtable tablet servers.

In the first phase of commit ("prewrite"), we try to lock all the cells being written. (To handle client failure, we designate one lock arbitrarily as the primary; we'll discuss this mechanism below.) The transaction reads metadata to check for conflicts in each cell being written. There are two kinds of conflicting metadata: if the transaction sees another write record after its start timestamp, it aborts (line 32); this is the write-write conflict that snapshot isolation guards against. If the transaction sees another lock at any timestamp, it also aborts (line 34). It's possible that the other transaction is just being slow to release its lock after having already committed below our start timestamp, but we consider this unlikely, so we abort. If there is no conflict, we write the lock and

```
1  class Transaction {
2   struct Write { Row row; Column col; string value; };
3   vector<Write> writes_;
4   int start_ts_;
5
6   Transaction() : start_ts_(oracle.GetTimestamp()) {}
7   void Set(Write w) { writes_.push_back(w); }
8   bool Get(Row row, Column c, string* value) {
9    while (true) {
10    bigtable::Txn T = bigtable::StartRowTransaction(row);
11    // Check for locks that signal concurrent writes.
12    if (T.Read(row, c+"lock", [0, start_ts_])) {
13     // There is a pending lock; try to clean it and wait
14     BackoffAndMaybeCleanupLock(row, c);
15     continue;
16    }
17
18    // Find the latest write below our start_timestamp.
19    latest_write = T.Read(row, c+"write", [0, start_ts_]);
20    if (!latest_write.found()) return false; // no data
21    int data_ts = latest_write.start_timestamp();
22    *value = T.Read(row, c+"data", [data_ts, data_ts]);
23    return true;
24   }
25  }
26 // Prewrite tries to lock cell w, returning false in case of conflict.
27  bool Prewrite(Write w, Write primary) {
28   Column c = w.col;
29   bigtable::Txn T = bigtable::StartRowTransaction(w.row);
30
31   // Abort on writes after our start timestamp ...
32   if (T.Read(w.row, c+"write", [start_ts_, ∞])) return false;
33   // ... or locks at any timestamp.
34   if (T.Read(w.row, c+"lock", [0, ∞])) return false;
35
36   T.Write(w.row, c+"data", start_ts_, w.value);
37   T.Write(w.row, c+"lock", start_ts_,
38     {primary.row, primary.col});      // The primary's location.
39   return T.Commit();
40  }
41  bool Commit() {
42   Write primary = writes_[0];
43   vector<Write> secondaries(writes_.begin()+1, writes_.end());
44   if (!Prewrite(primary, primary)) return false;
45   for (Write w : secondaries)
46    if (!Prewrite(w, primary)) return false;
47
48   int commit_ts = oracle_.GetTimestamp();
49
50   // Commit primary first.
51   Write p = primary;
52   bigtable::Txn T = bigtable::StartRowTransaction(p.row);
53   if (!T.Read(p.row, p.col+"lock", [start_ts_, start_ts_]))
54    return false;    // aborted while working
55   T.Write(p.row, p.col+"write", commit_ts,
56        start_ts_); // Pointer to data written at start_ts_
57   T.Erase(p.row, p.col+"lock", commit_ts);
58   if (!T.Commit()) return false;      // commit point
59
60   // Second phase: write out write records for secondary cells.
61   for (Write w : secondaries) {
62    bigtable::Write(w.row, w.col+"write", commit_ts, start_ts_);
63    bigtable::Erase(w.row, w.col+"lock", commit_ts);
64   }
65   return true;
66  }
67 } // class Transaction
```

Figure 6: Pseudocode for Percolator transaction protocol.

the data to each cell at the start timestamp (lines 36-38).

If no cells conflict, the transaction may commit and proceeds to the second phase. At the beginning of the second phase, the client obtains the commit timestamp from the timestamp oracle (line 48). Then, at each cell (starting with the primary), the client releases its lock and make its write visible to readers by replacing the lock with a write record. The write record indicates to readers that committed data exists in this cell; it contains a pointer to the start timestamp where readers can find the actual data. Once the primary's write is visible (line 58), the transaction must commit since it has made a write visible to readers.

A Get() operation first checks for a lock in the timestamp range [0, start_timestamp], which is the range of timestamps visible in the transaction's snapshot (line 12). If a lock is present, another transaction is concurrently writing this cell, so the reading transaction must wait until the lock is released. If no conflicting lock is found, Get() reads the latest write record in that timestamp range (line 19) and returns the data item corresponding to that write record (line 22).

Transaction processing is complicated by the possibility of client failure (tablet server failure does not affect the system since Bigtable guarantees that written locks persist across tablet server failures). If a client fails while a transaction is being committed, locks will be left behind. Percolator must clean up those locks or they will cause future transactions to hang indefinitely. 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.

It is very difficult for A to be perfectly confident in its judgment that B is failed; as a result we must avoid a race between A cleaning up B's transaction and a not-actually-failed B committing the same transaction. Percolator handles this by designating one cell in every transaction as a synchronizing point for any commit or cleanup operations. This cell's lock is called the primary lock. Both A and B agree on which lock is primary (the location of the primary is written into the locks at all other cells). Performing either a cleanup or commit operation requires modifying the primary lock; since this modification is performed under a Bigtable row transaction, only one of the cleanup or commit operations will succeed. Specifically: before B commits, it must check that it still holds the primary lock and replace it with a write record. Before A erases B's lock, A must check the primary to ensure that B has not committed; if the primary lock is still present, then it can safely erase the lock.

When a client crashes during the second phase of commit, a transaction will be past the commit point (it has written at least one write record) but will still

have locks outstanding. We must perform roll-forward on these transactions. A transaction that encounters a lock can distinguish between the two cases by inspecting the primary lock: if the primary lock has been replaced by a write record, the transaction which wrote the lock must have committed and the lock must be rolled forward, otherwise it should be rolled back (since we always commit the primary first, we can be sure that it is safe to roll back if the primary is not committed). To roll forward, the transaction performing the cleanup replaces the stranded lock with a write record as the original transaction would have done.

Since cleanup is synchronized on the primary lock, it is safe to clean up locks held by live clients; however, this incurs a performance penalty since rollback forces the transaction to abort. So, a transaction will not clean up a lock unless it suspects that a lock belongs to a dead or stuck worker. Percolator uses simple mechanisms to determine the liveness of another transaction. Running workers write a token into the Chubby lockservice [8] to indicate they belong to the system; other workers can use the existence of this token as a sign that the worker is alive (the token is automatically deleted when the process exits). To handle a worker that is live, but not working, we additionally write the wall time into the lock; a lock that contains a too-old wall time will be cleaned up even if the worker's liveness token is valid. To handle long-running commit operations, workers periodically update this wall time while committing.

2.3 Timestamps

The timestamp oracle is a server that hands out timestamps in strictly increasing order. Since every transaction requires contacting the timestamp oracle twice, this service must scale well. The oracle periodically allocates a range of timestamps by writing the highest allocated timestamp to stable storage; given an allocated range of timestamps, the oracle can satisfy future requests strictly from memory. If the oracle restarts, the timestamps will jump forward to the maximum allocated timestamp (but will never go backwards). To save RPC overhead (at the cost of increasing transaction latency) each Percolator worker batches timestamp requests across transactions by maintaining only one pending RPC to the oracle. As the oracle becomes more loaded, the batching naturally increases to compensate. Batching increases the scalability of the oracle but does not affect the timestamp guarantees. Our oracle serves around 2 million timestamps per second from a single machine.

The transaction protocol uses strictly increasing timestamps to guarantee that Get() returns all committed writes before the transaction's start timestamp. To see how it provides this guarantee, consider a transaction R reading at timestamp T_R and a transaction W that committed at timestamp $T_W < T_R$; we will show that R sees W's writes. Since $T_W < T_R$, we know that the timestamp oracle gave out T_W before or in the same batch as T_R; hence, W requested T_W before R received T_R. We know that R can't do reads before receiving its start timestamp T_R and that W wrote locks before requesting its commit timestamp T_W. Therefore, the above property guarantees that W must have at least written all its locks before R did any reads; R's Get() will see either the fully-committed write record or the lock, in which case W will block until the lock is released. Either way, W's write is visible to R's Get().

2.4 Notifications

Transactions let the user mutate the table while maintaining invariants, but users also need a way to trigger and run the transactions. In Percolator, the user writes code ("observers") to be triggered by changes to the table, and we link all the observers into a binary running alongside every tablet server in the system. Each observer registers a function and a set of columns with Percolator, and Percolator invokes the function after data is written to one of those columns in any row.

Percolator applications are structured as a series of observers; each observer completes a task and creates more work for "downstream" observers by writing to the table. In our indexing system, a MapReduce loads crawled documents into Percolator by running loader transactions, which trigger the document processor transaction to index the document (parse, extract links, etc.). The document processor transaction triggers further transactions like clustering. The clustering transaction, in turn, triggers transactions to export changed document clusters to the serving system.

Notifications are similar to database triggers or events in active databases [29], but unlike database triggers, they cannot be used to maintain database invariants. In particular, the triggered observer runs in a separate transaction from the triggering write, so the triggering write and the triggered observer's writes are not atomic. Notifications are intended to help structure an incremental computation rather than to help maintain data integrity.

This difference in semantics and intent makes observer behavior much easier to understand than the complex semantics of overlapping triggers. Percolator applications consist of very few observers — the Google indexing system has roughly 10 observers. Each observer is explicitly constructed in the `main()` of the worker binary, so it is clear what observers are active. It is possible for several observers to observe the same column, but we avoid this feature so it is clear what observer will run when a particular column is written. Users do need to be wary about infinite cycles of notifications, but Percolator does nothing to prevent this; the user typically constructs

a series of observers to avoid infinite cycles.

We do provide one guarantee: at most one observer's transaction will commit for each change of an observed column. The converse is not true, however: multiple writes to an observed column may cause the corresponding observer to be invoked only once. We call this feature message collapsing, since it helps avoid computation by amortizing the cost of responding to many notifications. For example, it is sufficient for `http://google.com` to be reprocessed periodically rather than every time we discover a new link pointing to it.

To provide these semantics for notifications, each observed column has an accompanying "acknowledgment" column for each observer, containing the latest start timestamp at which the observer ran. When the observed column is written, Percolator starts a transaction to process the notification. The transaction reads the observed column and its corresponding acknowledgment column. If the observed column was written after its last acknowledgment, then we run the observer and set the acknowledgment column to our start timestamp. Otherwise, the observer has already been run, so we do not run it again. Note that if Percolator accidentally starts two transactions concurrently for a particular notification, they will both see the dirty notification and run the observer, but one will abort because they will conflict on the acknowledgment column. We promise that at most one observer will *commit* for each notification.

To implement notifications, Percolator needs to efficiently find dirty cells with observers that need to be run. This search is complicated by the fact that notifications are rare: our table has trillions of cells, but, if the system is keeping up with applied load, there will only be millions of notifications. Additionally, observer code is run on a large number of client processes distributed across a collection of machines, meaning that this search for dirty cells must be distributed.

To identify dirty cells, Percolator maintains a special "notify" Bigtable column, containing an entry for each dirty cell. When a transaction writes an observed cell, it also sets the corresponding notify cell. The workers perform a distributed scan over the notify column to find dirty cells. After the observer is triggered and the transaction commits, we remove the notify cell. Since the notify column is just a Bigtable column, not a Percolator column, it has no transactional properties and serves only as a hint to the scanner to check the acknowledgment column to determine if the observer should be run.

To make this scan efficient, Percolator stores the notify column in a separate Bigtable locality group so that scanning over the column requires reading only the millions of dirty cells rather than the trillions of total data cells. Each Percolator worker dedicates several threads to the scan. For each thread, the worker chooses a portion of the table to scan by first picking a random Bigtable tablet, then picking a random key in the tablet, and finally scanning the table from that position. Since each worker is scanning a random region of the table, we worry about two workers running observers on the same row concurrently. While this behavior will not cause correctness problems due to the transactional nature of notifications, it is inefficient. To avoid this, each worker acquires a lock from a lightweight lock service before scanning the row. This lock server need not persist state since it is advisory and thus is very scalable.

The random-scanning approach requires one additional tweak: when it was first deployed we noticed that scanning threads would tend to clump together in a few regions of the table, effectively reducing the parallelism of the scan. This phenomenon is commonly seen in public transportation systems where it is known as "platooning" or "bus clumping" and occurs when a bus is slowed down (perhaps by traffic or slow loading). Since the number of passengers at each stop grows with time, loading delays become even worse, further slowing the bus. Simultaneously, any bus behind the slow bus speeds up as it needs to load fewer passengers at each stop. The result is a clump of buses arriving simultaneously at a stop [19]. Our scanning threads behaved analogously: a thread that was running observers slowed down while threads "behind" it quickly skipped past the now-clean rows to clump with the lead thread and failed to pass the lead thread because the clump of threads overloaded tablet servers. To solve this problem, we modified our system in a way that public transportation systems cannot: when a scanning thread discovers that it is scanning the same row as another thread, it chooses a new random location in the table to scan. To further the transportation analogy, the buses (scanner threads) in our city avoid clumping by teleporting themselves to a random stop (location in the table) if they get too close to the bus in front of them.

Finally, experience with notifications led us to introduce a lighter-weight but semantically weaker notification mechanism. We found that when many duplicates of the same page were processed concurrently, each transaction would conflict trying to trigger reprocessing of the same duplicate cluster. This led us to devise a way to notify a cell without the possibility of transactional conflict. We implement this weak notification by writing *only* to the Bigtable "notify" column. To preserve the transactional semantics of the rest of Percolator, we restrict these weak notifications to a special type of column that cannot be written, only notified. The weaker semantics also mean that multiple observers may run and commit as a result of a single weak notification (though the system tries to minimize this occurrence). This has become an important feature for managing conflicts; if an observer

frequently conflicts on a hotspot, it often helps to break it into two observers connected by a non-transactional notification on the hotspot.

2.5 Discussion

One of the inefficiencies of Percolator relative to a MapReduce-based system is the number of RPCs sent per work-unit. While MapReduce does a single large read to GFS and obtains all of the data for 10s or 100s of web pages, Percolator performs around 50 individual Bigtable operations to process a single document.

One source of additional RPCs occurs during commit. When writing a lock, we must do a read-modify-write operation requiring two Bigtable RPCs: one to read for conflicting locks or writes and another to write the new lock. To reduce this overhead, we modified the Bigtable API by adding conditional mutations which implements the read-modify-write step in a single RPC. Many conditional mutations destined for the same tablet server can also be batched together into a single RPC to further reduce the total number of RPCs we send. We create batches by delaying lock operations for several seconds to collect them into batches. Because locks are acquired in parallel, this adds only a few seconds to the latency of each transaction; we compensate for the additional latency with greater parallelism. Batching also increases the time window in which conflicts may occur, but in our low-contention environment this has not proved to be a problem.

We also perform the same batching when reading from the table: every read operation is delayed to give it a chance to form a batch with other reads to the same tablet server. This delays each read, potentially greatly increasing transaction latency. A final optimization mitigates this effect, however: prefetching. Prefetching takes advantage of the fact that reading two or more values in the same row is essentially the same cost as reading one value. In either case, Bigtable must read the entire SSTable block from the file system and decompress it. Percolator attempts to predict, each time a column is read, what other columns in a row will be read later in the transaction. This prediction is made based on past behavior. Prefetching, combined with a cache of items that have already been read, reduces the number of Bigtable reads the system would otherwise do by a factor of 10.

Early in the implementation of Percolator, we decided to make all API calls blocking and rely on running thousands of threads per machine to provide enough parallelism to maintain good CPU utilization. We chose this thread-per-request model mainly to make application code easier to write, compared to the event-driven model. Forcing users to bundle up their state each of the (many) times they fetched a data item from the table would have made application development much more difficult. Our experience with thread-per-request was, on the whole, positive: application code is simple, we achieve good utilization on many-core machines, and crash debugging is simplified by meaningful and complete stack traces. We encountered fewer race conditions in application code than we feared. The biggest drawbacks of the approach were scalability issues in the Linux kernel and Google infrastructure related to high thread counts. Our in-house kernel development team was able to deploy fixes to address the kernel issues.

3 Evaluation

Percolator lies somewhere in the performance space between MapReduce and DBMSs. For example, because Percolator is a distributed system, it uses far more resources to process a fixed amount of data than a traditional DBMS would; this is the cost of its scalability. Compared to MapReduce, Percolator can process data with far lower latency, but again, at the cost of additional resources required to support random lookups. These are engineering tradeoffs which are difficult to quantify: how much of an efficiency loss is too much to pay for the ability to add capacity endlessly simply by purchasing more machines? Or: how does one trade off the reduction in development time provided by a layered system against the corresponding decrease in efficiency?

In this section we attempt to answer some of these questions by first comparing Percolator to batch processing systems via our experiences with converting a MapReduce-based indexing pipeline to use Percolator. We'll also evaluate Percolator with microbenchmarks and a synthetic workload based on the well-known TPC-E benchmark [1]; this test will give us a chance to evaluate the scalability and efficiency of Percolator relative to Bigtable and DBMSs.

All of the experiments in this section are run on a subset of the servers in a Google data center. The servers run the Linux operating system on x86 processors; each machine is connected to several commodity SATA drives.

3.1 Converting from MapReduce

We built Percolator to create Google's large "base" index, a task previously performed by MapReduce. In our previous system, each day we crawled several billion documents and fed them along with a repository of existing documents through a series of 100 MapReduces. The result was an index which answered user queries. Though not all 100 MapReduces were on the critical path for every document, the organization of the system as a series of MapReduces meant that each document spent 2-3 days being indexed before it could be returned as a search result.

The Percolator-based indexing system (known as Caffeine [25]), crawls the same number of documents,

but we feed each document through Percolator as it is crawled. The immediate advantage, and main design goal, of Caffeine is a reduction in latency: the median document moves through Caffeine over 100x faster than the previous system. This latency improvement grows as the system becomes more complex: adding a new clustering phase to the Percolator-based system requires an extra lookup for each document rather an extra scan over the repository. Additional clustering phases can also be implemented in the same transaction rather than in another MapReduce; this simplification is one reason the number of observers in Caffeine (10) is far smaller than the number of MapReduces in the previous system (100). This organization also allows for the possibility of performing additional processing on only a subset of the repository without rescanning the entire repository.

Adding additional clustering phases isn't free in an incremental system: more resources are required to make sure the system keeps up with the input, but this is still an improvement over batch processing systems where no amount of resources can overcome delays introduced by stragglers in an additional pass over the repository. Caffeine is essentially immune to stragglers that were a serious problem in our batch-based indexing system because the bulk of the processing does not get held up by a few very slow operations. The radically-lower latency of the new system also enables us to remove the rigid distinctions between large, slow-to-update indexes and smaller, more rapidly updated indexes. Because Percolator frees us from needing to process the repository each time we index documents, we can also make it larger: Caffeine's document collection is currently 3x larger than the previous system's and is limited only by available disk space.

Compared to the system it replaced, Caffeine uses roughly twice as many resources to process the same crawl rate. However, Caffeine makes good use of the extra resources. If we were to run the old indexing system with twice as many resources, we could either increase the index size or reduce latency by at most a factor of two (but not do both). On the other hand, if Caffeine were run with half the resources, it would not be able to process as many documents per day as the old system (but the documents it did produce would have much lower latency).

The new system is also easier to operate. Caffeine has far fewer moving parts: we run tablet servers, Percolator workers, and chunkservers. In the old system, each of a hundred different MapReduces needed to be individually configured and could independently fail. Also, the "peaky" nature of the MapReduce workload made it hard to fully utilize the resources of a datacenter compared to Percolator's much smoother resource usage.

The simplicity of writing straight-line code and the ability to do random lookups into the repository makes developing new features for Percolator easy. Under

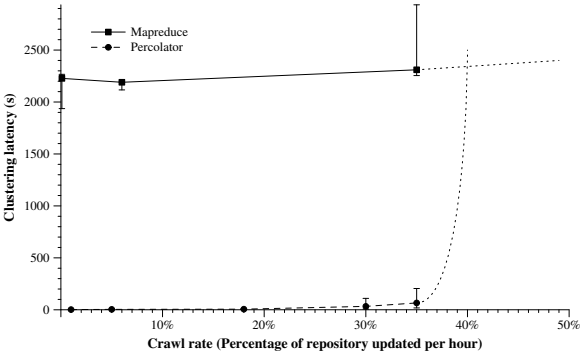


Figure 7: Median document clustering delay for Percolator (dashed line) and MapReduce (solid line). For MapReduce, all documents finish processing at the same time and error bars represent the min, median, and max of three runs of the clustering MapReduce. For Percolator, we are able to measure the delay of individual documents, so the error bars represent the 5th- and 95th-percentile delay on a per-document level.

MapReduce, random lookups are awkward and costly. On the other hand, Caffeine developers need to reason about concurrency where it did not exist in the MapReduce paradigm. Transactions help deal with this concurrency, but can't fully eliminate the added complexity.

To quantify the benefits of moving from MapReduce to Percolator, we created a synthetic benchmark that clusters newly crawled documents against a billion-document repository to remove duplicates in much the same way Google's indexing pipeline operates. Documents are clustered by three clustering keys. In a real system, the clustering keys would be properties of the document like redirect target or content hash, but in this experiment we selected them uniformly at random from a collection of 750M possible keys. The average cluster in our synthetic repository contains 3.3 documents, and 93% of the documents are in a non-singleton cluster. This distribution of keys exercises the clustering logic, but does not expose it to the few extremely large clusters we have seen in practice. These clusters only affect the latency tail and not the results we present here. In the Percolator clustering implementation, each crawled document is immediately written to the repository to be clustered by an observer. The observer maintains an index table for each clustering key and compares the document against each index to determine if it is a duplicate (an elaboration of Figure 2). MapReduce implements clustering of continually arriving documents by repeatedly running a sequence of three clustering MapReduces (one for each clustering key). The sequence of three MapReduces processes the entire repository and any crawled documents that accumulated while the previous three were running.

This experiment simulates clustering documents crawled at a uniform rate. Whether MapReduce or Percolator performs better under this metric is a function of the how frequently documents are crawled (the crawl rate)

and the repository size. We explore this space by fixing the size of the repository and varying the rate at which new documents arrive, expressed as a percentage of the repository crawled per hour. In a practical system, a very small percentage of the repository would be crawled per hour: there are over 1 trillion web pages on the web (and ideally in an indexing system's repository), far too many to crawl a reasonable fraction of in a single day. When the new input is a small fraction of the repository (low crawl rate), we expect Percolator to outperform MapReduce since MapReduce must map over the (large) repository to cluster the (small) batch of new documents while Percolator does work proportional only to the small batch of newly arrived documents (a lookup in up to three index tables per document). At very large crawl rates where the number of newly crawled documents approaches the size of the repository, MapReduce will perform better than Percolator. This cross-over occurs because streaming data from disk is much cheaper, per byte, than performing random lookups. At the cross-over the total cost of the lookups required to cluster the new documents under Percolator equals the cost to stream the documents and the repository through MapReduce. At crawl rates higher than that, one is better off using MapReduce.

We ran this benchmark on 240 machines and measured the median delay between when a document is crawled and when it is clustered. Figure 7 plots the median latency of document processing for both implementations as a function of crawl rate. When the crawl rate is low, Percolator clusters documents faster than MapReduce as expected; this scenario is illustrated by the leftmost pair of points which correspond to crawling 1 percent of documents per hour. MapReduce requires approximately 20 minutes to cluster the documents because it takes 20 minutes just to process the repository through the three MapReduces (the effect of the few newly crawled documents on the runtime is negligible). This results in an average delay between crawling a document and clustering of around 30 minutes: a random document waits 10 minutes after being crawled for the previous sequence of MapReduces to finish and then spends 20 minutes being processed by the three MapReduces. Percolator, on the other hand, finds a newly loaded document and processes it in two seconds on average, or about 1000x faster than MapReduce. The two seconds includes the time to find the dirty notification and run the transaction that performs the clustering. Note that this 1000x latency improvement could be made arbitrarily large by increasing the size of the repository.

As the crawl rate increases, MapReduce's processing time grows correspondingly. Ideally, it would be proportional to the combined size of the repository and the input which grows with the crawl rate. In practice, the running time of a small MapReduce like this is limited by stragglers, so the growth in processing time (and thus clustering latency) is only weakly correlated to crawl rate at low crawl rates. The 6 percent crawl rate, for example, only adds 150GB to a 1TB data set; the extra time to process 150GB is in the noise. The latency of Percolator is relatively unchanged as the crawl rate grows until it suddenly increases to effectively infinity at a crawl rate of 40% per hour. At this point, Percolator saturates the resources of the test cluster, is no longer able to keep up with the crawl rate, and begins building an unbounded queue of unprocessed documents. The dotted asymptote at 40% is an extrapolation of Percolator's performance beyond this breaking point. MapReduce is subject to the same effect: eventually crawled documents accumulate faster than MapReduce is able to cluster them, and the batch size will grow without bound in subsequent runs. In this particular configuration, however, MapReduce can sustain crawl rates in excess of 100% (the dotted line, again, extrapolates performance).

These results show that Percolator can process documents at orders of magnitude better latency than MapReduce in the regime where we expect real systems to operate (single-digit crawl rates).

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

Figure 8: The overhead of Percolator operations relative to Bigtable. Write overhead is due to additional operations Percolator needs to check for conflicts.

3.2 Microbenchmarks

In this section, we determine the cost of the transactional semantics provided by Percolator. In these experiments, we compare Percolator to a "raw" Bigtable. We are only interested in the relative performance of Bigtable and Percolator since any improvement in Bigtable performance will translate directly into an improvement in Percolator performance. Figure 8 shows the performance of Percolator and raw Bigtable running against a single tablet server. All data was in the tablet server's cache during the experiments and Percolator's batching optimizations were disabled.

As expected, Percolator introduces overhead relative to Bigtable. We first measure the number of random writes that the two systems can perform. In the case of Percolator, we execute transactions that write a single cell and then commit; this represents the worst case for Percolator overhead. When doing a write, Percolator incurs roughly a factor of four overhead on this benchmark. This is the result of the extra operations Percolator requires for commit beyond the single write that Bigtable issues: a read to check for locks, a write to add the lock, and a second write to remove the lock record. The read, in particular, is more expensive than a write and accounts

for most of the overhead. In this test, the limiting factor was the performance of the tablet server, so the additional overhead of fetching timestamps is not measured. We also tested random reads: Percolator performs a single Bigtable operation per read, but that read operation is somewhat more complex than the raw Bigtable operation (the Percolator read looks at metadata columns in addition to data columns).

3.3 Synthetic Workload

To evaluate Percolator on a more realistic workload, we implemented a synthetic benchmark based on TPC-E [1]. This isn't the ideal benchmark for Percolator since TPC-E is designed for OLTP systems, and a number of Percolator's tradeoffs impact desirable properties of OLTP systems (the latency of conflicting transactions, for example). TPC-E is a widely recognized and understood benchmark, however, and it allows us to understand the cost of our system against more traditional databases.

TPC-E simulates a brokerage firm with customers who perform trades, market search, and account inquiries. The brokerage submits trade orders to a market exchange, which executes the trade and updates broker and customer state. The benchmark measures the number of trades executed. On average, each customer performs a trade once every 500 seconds, so the benchmark scales by adding customers and associated data.

TPC-E traditionally has three components – a customer emulator, a market emulator, and a DBMS running stored SQL procedures. Since Percolator is a client library running against Bigtable, our implementation is a combined customer/market emulator that calls into the Percolator library to perform operations against Bigtable. Percolator provides a low-level Get/Set/iterator API rather than a high-level SQL interface, so we created indexes and did all the 'query planning' by hand.

Since Percolator is an incremental processing system rather than an OLTP system, we don't attempt to meet the TPC-E latency targets. Our average transaction latency is 2 to 5 seconds, but outliers can take several minutes. Outliers are caused by, for example, exponential backoff on conflicts and Bigtable tablet unavailability. Finally, we made a small modification to the TPC-E transactions. In TPC-E, each trade result increases the broker's commission and increments his trade count. Each broker services a hundred customers, so the average broker must be updated once every 5 seconds, which causes repeated write conflicts in Percolator. In Percolator, we would implement this feature by writing the increment to a side table and periodically aggregating each broker's increments; for the benchmark, we choose to simply omit this write.

Figure 9 shows how the resource usage of Percolator scales as demand increases. We will measure resource

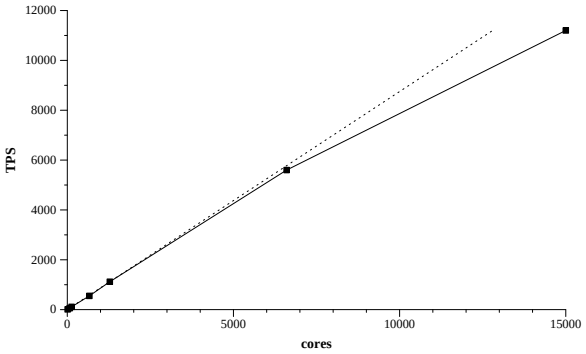


Figure 9: Transaction rate on a TPC-E-like benchmark as a function of cores used. The dotted line shows linear scaling.

usage in CPU cores since that is the limiting resource in our experimental environment. We were able to procure a small number of machines for testing, but our test Bigtable cell shares the disk resources of a much larger production cluster. As a result, disk bandwidth is not a factor in the system's performance. In this experiment, we configured the benchmark with increasing numbers of customers and measured both the achieved performance and the number of cores used by all parts of the system including cores used for background maintenance such as Bigtable compactions. The relationship between performance and resource usage is essentially linear across several orders of magnitude, from 11 cores to 15,000 cores.

This experiment also provides an opportunity to measure the overheads in Percolator relative to a DBMS. The fastest commercial TPC-E system today performs 3,183 tpsE using a single large shared-memory machine with 64 Intel Nehalem cores with 2 hyperthreads per core [33]. Our synthetic benchmark based on TPC-E performs 11,200 tps using 15,000 cores. This comparison is very rough: the Nehalem cores in the comparison machine are significantly faster than the cores in our test cell (small-scale testing on Nehalem processors shows that they are 20-30% faster per-thread compared to the cores in the test cluster). However, we estimate that Percolator uses roughly 30 times more CPU per transaction than the benchmark system. On a cost-per-transaction basis, the gap is likely much less than 30 since our test cluster uses cheaper, commodity hardware compared to the enterprise-class hardware in the reference machine.

The conventional wisdom on implementing databases is to "get close to the iron" and use hardware as directly as possible since even operating system structures like disk caches and schedulers make it hard to implement an efficient database [32]. In Percolator we not only interposed an operating system between our database and the hardware, but also several layers of software and network links. The conventional wisdom is correct: this arrangement has a cost. There are substantial overheads in

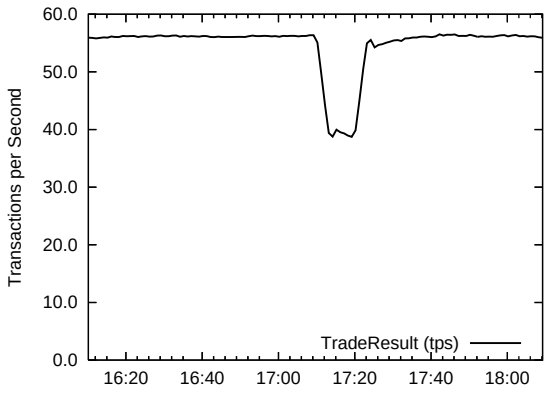


Figure 10: Recovery of tps after 33% tablet server mortality

preparing requests to go on the wire, sending them, and processing them on a remote machine. To illustrate these overheads in Percolator, consider the act of mutating the database. In a DBMS, this incurs a function call to store the data in memory and a system call to force the log to hardware controlled RAID array. In Percolator, a client performing a transaction commit sends multiple RPCs to Bigtable, which commits the mutation by logging it to 3 chunkservers, which make system calls to actually flush the data to disk. Later, that same data will be compacted into minor and major sstables, each of which will be again replicated to multiple chunkservers.

The CPU inflation factor is the cost of our layering. In exchange, we get scalability (our fastest result, though not directly comparable to TPC-E, is more than 3x the current official record [33]), and we inherit the useful features of the systems we build upon, like resilience to failures. To demonstrate the latter, we ran the benchmark with 15 tablet servers and allowed the performance to stabilize. Figure 10 shows the performance of the system over time. The dip in performance at 17:09 corresponds to a failure event: we killed a third of the tablet servers. Performance drops immediately after the failure event but recovers as the tablets are reloaded by other tablet servers. We allowed the killed tablet servers to restart so performance eventually returns to the original level.

4 Related Work

Batch processing systems like MapReduce [13, 22, 24] are well suited for efficiently transforming or analyzing an entire corpus: these systems can simultaneously use a large number of machines to process huge amounts of data quickly. Despite this scalability, re-running a MapReduce pipeline on each small batch of updates results in unacceptable latency and wasted work. Overlapping or pipelining the adjacent stages can reduce latency [10], but straggler shards still set the minimum time to complete the pipeline. Percolator avoids the expense of repeated scans by, essentially, creating indexes on the keys used to cluster documents; one of criticisms leveled by Stonebraker and DeWitt in their initial critique of MapReduce [16] was that MapReduce did not support such indexes.

Several proposed modifications to MapReduce [18, 26, 35] reduce the cost of processing changes to a repository by allowing workers to randomly read a base repository while mapping over only newly arrived work. To implement clustering in these systems, we would likely maintain a repository per clustering phase. Avoiding the need to re-map the entire repository would allow us to make batches smaller, reducing latency. DryadInc [31] attacks the same problem by reusing identical portions of the computation from previous runs and allowing the user to specify a merge function that combines new input with previous iterations' outputs. These systems represent a middle-ground between mapping over the entire repository using MapReduce and processing a single document at a time with Percolator.

Databases satisfy many of the requirements of an incremental system: a RDBMS can make many independent and concurrent changes to a large corpus and provides a flexible language for expressing computation (SQL). In fact, Percolator presents the user with a database-like interface: it supports transactions, iterators, and secondary indexes. While Percolator provides distributed transactions, it is by no means a full-fledged DBMS: it lacks a query language, for example, as well as full relational operations such as join. Percolator is also designed to operate at much larger scales than existing parallel databases and to deal better with failed machines. Unlike Percolator, database systems tend to emphasize latency over throughput since a human is often waiting for the results of a database query.

The organization of data in Percolator mirrors that of shared-nothing parallel databases [7, 15, 4]. Data is distributed across a number of commodity machines in shared-nothing fashion: the machines communicate only via explicit RPCs; no shared memory or shared disks are used. Data stored by Percolator is partitioned by Bigtable into tablets of contiguous rows which are distributed among machines; this mirrors the declustering performed by parallel databases.

The transaction management of Percolator builds on a long line of work on distributed transactions for database systems. Percolator implements snapshot isolation [5] by extending multi-version timestamp ordering [6] across a distributed system using two-phase commit.

An analogy can be drawn between the role of observers in Percolator to incrementally move the system towards a "clean" state and the incremental maintenance of materialized views in traditional databases (see Gupta and Mumick [21] for a survey of the field). In practice, while some indexing tasks like clustering documents by

contents could be expressed in a form appropriate for incremental view maintenance it would likely be hard to express the transformation of a raw document into an indexed document in such a form.

The utility of parallel databases and, by extension, a system like Percolator, has been questioned several times [17] over their history. Hardware trends have, in the past, worked against parallel databases. CPUs have become so much faster than disks that a few CPUs in a shared-memory machine can drive enough disk heads to service required loads without the complexity of distributed transactions: the top TPC-E benchmark results today are achieved on large shared-memory machines connected to a SAN. This trend is beginning to reverse itself, however, as the enormous datasets like those Percolator is intended to process become far too large for a single shared-memory machine to handle. These datasets require a distributed solution that can scale to 1000s of machines, while existing parallel databases can utilize only 100s of machines [30]. Percolator provides a system that is scalable enough for Internet-sized datasets by sacrificing some (but not all) of the flexibility and low-latency of parallel databases.

Distributed storage systems like Bigtable have the scalability and fault-tolerance properties of MapReduce but provide a more natural abstraction for storing a repository. Using a distributed storage system allows for low-latency updates since the system can change state by mutating the repository rather than rewriting it. However, Percolator is a data transformation system, not only a data storage system: it provides a way to structure computation to transform that data. In contrast, systems like Dynamo [14], Bigtable, and PNUTS [11] provide highly available data storage without the attendant mechanisms of transformation. These systems can also be grouped with the NoSQL databases (MongoDB [27], to name one of many): both offer higher performance and scale better than traditional databases, but provide weaker semantics.

Percolator extends Bigtable with multi-row, distributed transactions, and it provides the observer interface to allow applications to be structured around notifications of changed data. We considered building the new indexing system directly on Bigtable, but the complexity of reasoning about concurrent state modification without the aid of strong consistency was daunting. Percolator does not inherit all of Bigtable's features: it has limited support for replication of tables across data centers, for example. Since Bigtable's cross data center replication strategy is consistent only on a per-tablet basis, replication is likely to break invariants between writes in a distributed transaction. Unlike Dynamo and PNUTS which serve responses to users, Percolator is willing to accept the lower availability of a single data center in return for stricter consistency.

Several research systems have, like Percolator, extended distributed storage systems to include strong consistency. Sinfonia [3] provides a transactional interface to a distributed repository. Earlier published versions of Sinfonia [2] also offered a notification mechanism similar to the Percolator's observer model. Sinfonia and Percolator differ in their intended use: Sinfonia is designed to build distributed infrastructure while Percolator is intended to be used directly by applications (this probably explains why Sinfonia's authors dropped its notification mechanism). Additionally, Sinfonia's mini-transactions have limited semantics compared to the transactions provided by RDBMSs or Percolator: the user must specify a list of items to compare, read, and write prior to issuing the transaction. The mini-transactions are sufficient to create a wide variety of infrastructure but could be limiting for application builders.

CloudTPS [34], like Percolator, builds an ACID-compliant datastore on top of a distributed storage system (HBase [23] or Bigtable). Percolator and CloudTPS systems differ in design, however: the transaction management layer of CloudTPS is handled by an intermediate layer of servers called local transaction managers that cache mutations before they are persisted to the underlying distributed storage system. By contrast, Percolator uses clients, directly communicating with Bigtable, to coordinate transaction management. The focus of the systems is also different: CloudTPS is intended to be a backend for a website and, as such, has a stronger focus on latency and partition tolerance than Percolator.

ElasTraS [12], a transactional data store, is architecturally similar to Percolator; the Owning Transaction Managers in ElasTraS are essentially tablet servers. Unlike Percolator, ElasTraS offers limited transactional semantics (Sinfonia-like mini-transactions) when dynamically partitioning the dataset and has no support for structuring computation.

5 Conclusion and Future Work

We have built and deployed Percolator and it has been used to produce Google's websearch index since April, 2010. The system achieved the goals we set for reducing the latency of indexing a single document with an acceptable increase in resource usage compared to the previous indexing system.

The TPC-E results suggest a promising direction for future investigation. We chose an architecture that scales linearly over many orders of magnitude on commodity machines, but we've seen that this costs a significant 30-fold overhead compared to traditional database architectures. We are very interested in exploring this tradeoff and characterizing the nature of this overhead: how much is fundamental to distributed storage systems, and how much can be optimized away?

Acknowledgments

Percolator could not have been built without the assistance of many individuals and teams. We are especially grateful to the members of the indexing team, our primary users, and the developers of the many pieces of infrastructure who never failed to improve their services to meet our increasingly large demands.

References

[1] TPC benchmark E standard specification version 1.9.0. Tech. rep., Transaction Processing Performance Council, September 2009.

[2] AGUILERA, M. K., KARAMANOLIS, C., MERCHANT, A., SHAH, M., AND VEITCH, A. Building distributed applications using Sinfonia. Tech. rep., Hewlett-Packard Labs, 2006.

[3] AGUILERA, M. K., MERCHANT, A., SHAH, M., VEITCH, A., AND KARAMANOLIS, C. Sinfonia: a new paradigm for building scalable distributed systems. In *SOSP '07* (2007), ACM, pp. 159–174.

[4] BARU, C., FECTEAU, G., GOYAL, A., HSIAO, H.-I., JHINGRAN, A., PADMANABHAN, S., WILSON, W., AND I HSIAO, A. G. H. DB2 parallel edition, 1995.

[5] BERENSON, H., BERNSTEIN, P., GRAY, J., MELTON, J., O'NEIL, E., AND O'NEIL, P. A critique of ANSI SQL isolation levels. In *SIGMOD* (New York, NY, USA, 1995), ACM, pp. 1–10.

[6] BERNSTEIN, P. A., AND GOODMAN, N. Concurrency control in distributed database systems. *ACM Computer Surveys 13*, 2 (1981), 185–221.

[7] BORAL, H., ALEXANDER, W., CLAY, L., COPELAND, G., DANFORTH, S., FRANKLIN, M., HART, B., SMITH, M., AND VALDURIEZ, P. Prototyping Bubba, a highly parallel database system. *IEEE Transactions on Knowledge and Data Engineering 2*, 1 (1990), 4–24.

[8] BURROWS, M. The Chubby lock service for loosely-coupled distributed systems. In *7th OSDI* (Nov. 2006).

[9] CHANG, F., DEAN, J., GHEMAWAT, S., HSIEH, W. C., WALLACH, D. A., BURROWS, M., CHANDRA, T., FIKES, A., AND GRUBER, R. E. Bigtable: A distributed storage system for structured data. In *7th OSDI* (Nov. 2006), pp. 205–218.

[10] CONDIE, T., CONWAY, N., ALVARO, P., AND HELLERSTIEN, J. M. MapReduce online. In *7th NSDI* (2010).

[11] COOPER, B. F., RAMAKRISHNAN, R., SRIVASTAVA, U., SILBERSTEIN, A., BOHANNON, P., JACOBSEN, H.-A., PUZ, N., WEAVER, D., AND YERNENI, R. PNUTS: Yahoo!'s hosted data serving platform. In *Proceedings of VLDB* (2008).

[12] DAS, S., AGRAWAL, D., AND ABBADI, A. E. ElasTraS: An elastic transactional data store in the cloud. In *USENIX HotCloud* (June 2009).

[13] DEAN, J., AND GHEMAWAT, S. MapReduce: Simplified data processing on large clusters. In *6th OSDI* (Dec. 2004), pp. 137–150.

[14] DECANDIA, G., HASTORUN, D., JAMPANI, M., KAKULAPATI, G., LAKSHMAN, A., PILCHIN, A., SIVASUBRAMANIAN, S., VOSSHALL, P., AND VOGELS, W. Dynamo: Amazon's highly available key-value store. In *SOSP '07* (2007), pp. 205–220.

[15] DEWITT, D., GHANDEHARIZADEH, S., SCHNEIDER, D., BRICKER, A., HSIAO, H.-I., AND RASMUSSEN, R. The gamma database machine project. *IEEE Transactions on Knowledge and Data Engineering 2* (1990), 44–62.

[16] DEWITT, D., AND STONEBRAKER, M. MapReduce: A major step backwards. http://databasecolumn.vertica.com/database-innovation/mapreduce-a-major-step-backwards/.

[17] DEWITT, D. J., AND GRAY, J. Parallel database systems: the future of database processing or a passing fad? *SIGMOD Rec. 19*, 4 (1990), 104–112.

[18] EKANAYAKE, J., LI, H., ZHANG, B., GUNARATHNE, T., BAE, S.-H., QIU, J., AND FOX, G. Twister: A runtime for iterative MapReduce. In *The First International Workshop on MapReduce and its Applications* (2010).

[19] GERSHENSON, C., AND PINEDA, L. A. Why does public transport not arrive on time? The pervasiveness of equal headway instability. *PLoS ONE 4*, 10 (10 2009).

[20] GHEMAWAT, S., GOBIOFF, H., AND LEUNG, S.-T. The Google file system. vol. 37, pp. 29–43.

[21] GUPTA, A., AND MUMICK, I. S. Maintenance of materialized views: Problems, techniques, and applications, 1995.

[22] Hadoop. http://hadoop.apache.org/.

[23] HBase. http://hbase.apache.org/.

[24] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: Distributed data-parallel programs from sequential building blocks. In *EuroSys '07* (New York, NY, USA, 2007), ACM, pp. 59–72.

[25] IYER, S., AND CUTTS, M. Help test some next-generation infrstructure. http://googlewebmastercentral.blogspot.com/2009/08/help-test-some-next-generation.html, August 2009.

[26] LOGOTHETIS, D., OLSTON, C., REED, B., WEBB, K. C., AND YOCUM, K. Stateful bulk processing for incremental analytics. In *SoCC '10: Proceedings of the 1st ACM symposium on cloud computing* (2010), pp. 51–62.

[27] MongoDB. http://mongodb.org/.

[28] PAGE, L., BRIN, S., MOTWANI, R., AND WINOGRAD, T. The PageRank citation ranking: Bringing order to the web. Tech. rep., Stanford Digital Library Technologies Project, 1998.

[29] PATON, N. W., AND DÍAZ, O. Active database systems. *ACM Computing Surveys 31*, 1 (1999), 63–103.

[30] PAVLO, A., PAULSON, E., RASIN, A., ABADI, D. J., DEWITT, D. J., MADDEN, S., AND STONEBRAKER, M. A comparison of approaches to large-scale data analysis. In *SIGMOD '09* (June 2009), ACM.

[31] POPA, L., BUDIU, M., YU, Y., AND ISARD, M. DryadInc: Reusing work in large-scale computations. In *USENIX workshop on Hot Topics in Cloud Computing* (2009).

[32] STONEBRAKER, M. Operating system support for database management. *Communications of the ACM 24*, 7 (1981), 412–418.

[33] NEC Express5800/A1080a-E TPC-E results. http://www.tpc.org/tpce/results/tpce_result_detail.asp?id=110033001, Mar. 2010.

[34] WEI, Z., PIERRE, G., AND CHI, C.-H. CloudTPS: Scalable transactions for Web applications in the cloud. Tech. Rep. IR-CS-053, Vrije Universiteit, Amsterdam, The Netherlands, Feb. 2010. http://www.globule.org/publi/CSTWAC_ircs53.html.

[35] ZAHARIA, M., CHOWDHURY, M., FRANKLIN, M., SHENKER, S., AND STOICA, I. Spark: Cluster computing with working sets. In *2nd USENIX workshop on Hot Topics in Cloud Computing* (2010).