Large-scale Incremental Processing Using Distributed Transactions and Notifications

wu

May 13, 2024

1 Abstract

Target: a class of data processing tasks that transform a large repository of data via small, independent mutations.

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.

2 Introduction

The indexing system could store the repository in a DBMS and update individual documents while using transactions to maintain invariants.

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. 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;

- 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.

- Requirements:

  1. Computations where the result can't be broken down into small updates (sorting a file, for example) are better handled by MapReduce.
  2. the computation should have strong consistency requirements; otherwise, Bigtable is sufficient.
  3. the computation should be very large in some dimension (total data size, CPU required for transformation, etc.);

3 Design

Percolator provides two main abstractions for performing incremental processing at large scale:

1. ACID transactions over a random-access repository

2. 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 tablet server, and a GFS 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.

The design of Percolator was influenced by the requirement to run at massive scales and the lack of a requirement for extremely low latency. Percolator has no central location for transaction management; in particular, it lacks a global deadlock detector.

3.1 Bigtable overview

3.2 Transactions

Percolator provides cross-row, cross-table transactions with ACID snapshot-isolation semantics.

```
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.

If `Commit()` return 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.

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. 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.

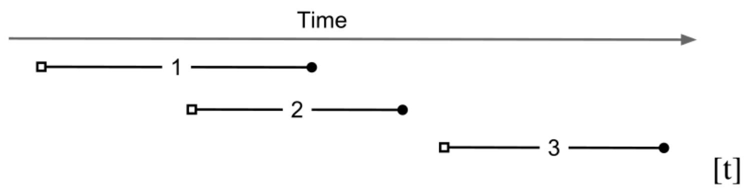

Figure 3: Transactions under snapshot isolation perform reads at a start timestamp (represented here by an open square) and writes at a commit timestamp (closed circle). In this example, transaction 2 would not see writes from transaction 1 since transaction 2's start timestamp is before transaction 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.

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 should:

1. Locks must persist in the face of machine failure; if a lock could disappear between the two phases of commit, the system could mistakenly commit two transactions that should have conflicted.

2. provide high throughput; thousands of machines will be requesting locks simultaneously.

3. 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.

```
class Transaction {
    struct Write { Row row; Column col; string value; };
    vector<Write> writes_;
    int start_ts_;
```

```
Transaction() : start_ts_(oracle.GetTimestamp()) {}
void Set(Write w) { writes_.push back(w); }
bool Get(Row row, Column c, string* value) {
    while (true) {
        bigtable::Txn T = bigtable::StartRowTransaction(row);
        // Check for locks that signal concurrent writes.
        if (T.Read(row, c+"lock", [0, start_ts_])) {
            // There is a pending lock; try to clean it and wait
            BackoffAndMaybeCleanupLock(row, c);
            continue;
        }
        // Find the latest write below our start_timestamp.
        latest write = T.Read(row, c+"write", [0, start_ts_]);
        if (!latest write.found()) return false; // no data
        int data_ts = latest write.start timestamp();
        *value = T.Read(row, c+"data", [data_ts, data_ts]);
        return true;
    }
}
// Prewrite tries to lock cell w, returning false in case of conflict.
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();
}
bool Commit() {
    Write primary = writes_[0];
    vector<Write> secondaries(writes_.begin()+1, writes_.end());
    if (!Prewrite(primary, primary)) return false;
    for (Write w : secondaries)
        if (!Prewrite(w, primary)) return false;
    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_); // Pointer to data written at 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);
        }
        return true;
    }
} // class Transaction
```

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

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

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

Table 1: The columns in the Bigtable representation of a Percolator column named "c"

Column	Use
`c:lock`	An uncommitted transaction is writing this cell; the location of primary lock
`c:write`	committed data present; stores the Bigtable timestamp of the data
`c:data`	stores the data itself
`c:tabnotify`	Hint: observers may need to run
`c:ack_()`	Observer "O" has run; stores start timestamp of successful last run

The transaction's constructor asks the timestamp oracle for a start timestamp, which determines the consistent snapshot seen by `Get()`. Calls to `Set()` are buffered 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. The transaction reads metadata to check for conflicts in each cell being written. There are two kinds of conflicting metadata:

1. if the transaction sees another write record after its start timestamp, it aborts; this is the write-write conflict that snapshot isolation guards against.

2. If the transaction sees another lock at any timestamp, it also aborts.

   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 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. 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 actually data. Once the primary's write is visible (commit point), the transaction must commit since it has made a write visible to readers.

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).

4 Problems

1. what if two concurrent writes in `prewrite`, e.g. two `T.Write(w.row, c+"data", start_ts_, w.value);`, the lock check does not do anything actually.

Problems	Desc
??	what's duplicates
??	whats primary for
what if two writes	

5 References

References