TiKV Version 6.5

Percolator

TiKV supports distributed transactions, which is inspired by Google's Percolator. In this section, we will briefly introduce Percolator and how we make use of it in TiKV.

What is Percolator?

Percolator is a system built by Google for incremental processing on a very large data set. Since this is just a brief introduction, you can view the full paper here for more details. If you are already very familiar with it, you can skip this section and go directly to Percolator in TiKV

Percolator is built based on Google's BigTable, a distributed storage system that supports single-row transactions. Percolator implements distributed transactions in ACID snapshot-isolation semantics, which is not supported by BigTable. A column `c` of Percolator is actually divided into the following internal columns of BigTable:

- `c:lock`
- `c:write`
- `c:data`
- `c:notify`

Percolator also relies on a service named *timestamp oracle*. The timestamp oracle can produce timestamps in a strictly increasing order. All read and write operations need to apply for timestamps from the timestamp oracle, and a timestamp coming from the timestamp oracle will be used as the time when the read/write operation happens.

Percolator is a multi-version storage, and a data item's version is represented by the timestamp when the transaction was committed.

For example,

key	v:data	v:lock	v:write
k1	14:“value2" 12: 10:“value1”	14:primary 12: 10:	14: 12:data@10 10:

The table shows different versions of data for a single cell. The state shown in the table means that for key `k1`, value `"value1"` was committed at timestamp `12`. Then there is an uncommitted version whose value is `"value2"`, and it's uncommitted because there's a lock. You will understand why it is like this after understanding how transactions work.

The remaining columns, `c:notify` and `c:ack_O`, are used for Percolator's incremental processing. After a modification, `c:notify` column is used to mark the modified cell to be dirty. Users can add some *observers* to Percolator which can do user-specified operations when they find data of their observed columns has changed. To find whether data is “acknowledgment” column of observer `O`, which is used to prevent a row from being incorrectly notified twice. It saves the timestamp of the observer’s last execution.

Writing

Percolator’s transactions are committed by a 2-phase commit (2PC) algorithm. Its two phases are `Prewrite` and `Commit`.

In `Prewrite` phase:

1. Get a timestamp from the timestamp oracle, and we call the timestamp `start_ts` of the transaction.
2. For each row involved in the transaction, put a lock in the `lock` column and write the value to the `data` column with the timestamp `start_ts`. One of these locks will be chosen as the *primary* lock, while others are *secondary* locks. Each lock contains the transaction’s `start_ts`. Each secondary lock, in addition, contains the location of the primary lock.
    - If there’s already a lock or newer version than `start_ts`, the current transaction will be rolled back because of write conflict.

And then, in the `Commit` phase:

1. Get another timestamp, namely `commit_ts`.
2. Remove the primary lock, and at the same time write a record to the `write` column with timestamp

fails.

3. Repeat the process above for all secondary locks.

Once step 2 (committing the primary) is done, the whole transaction is done. It doesn't matter if the process of committing the secondary locks failed.

Let's see the example from the paper of Percolator. Assume we are writing two rows in a single transaction. At first, the data looks like this:

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:

This table shows Bob and Joe's balance. Now Bob wants to transfer his $7 to Joe's account. The first step is `Prewrite` :

1. Get the `start_ts` of the transaction. In our example, it's `7` .
2. For each row involved in this transaction, put a lock in the `lock` column, and write the data to the `data` column. One of the locks will be chosen as the primary lock.

After `Prewrite` , our data looks like this:

Bob	7:$3 6: 5:$10	7: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:

Then `Commit` :

1. Get the `commit_ts` , in our case, `8` .
2. Commit the primary: Remove the primary lock and write the commit record to the `write` column.

key	bal:data	bal:lock	bal:write
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:

3. Commit all secondary locks to complete the writing process.

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:

Reading

Reading from Percolator also requires a timestamp. The procedure to perform a read operation is as follows:

1. Get a timestamp `ts` .
2. Check if the row we are going to read is locked with a timestamp in the range `[0, ts]` .
    - If there is a lock with the timestamp in range `[0, ts]` , it means the row was locked by an earlier-started transaction. Then we are not sure whether that transaction will be committed before or after `ts` . In this case the reading will backoff and try again then.
    - If there is no lock or the lock's timestamp is greater than `ts` , the read can continue.
3. Get the latest record in the row's `write` column whose `commit_ts` is in range `[0, ts]` . The record contains

4. Get the row's value in the `data` column whose timestamp is exactly `start_ts`. Then the value is what we want.

For example, consider this table again:

key	bal:data	bal:lock	bal:write
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:

Let's read Bob's balance.

1. Get a timestamp. Assume it's `9`.
2. Check the lock of the row. The row of Bob is not locked, so we continue.
3. Get the latest record in the `write` column committed before `9`. We get a record with `commit_ts` equals to `8`, and `start_ts` `7`, which means, its corresponding data is at timestamp `7` in the `data` column.
4. Get the value in the `data` column with timestamp `7`. `$3` is the result to the read.

This algorithm provides us with the abilities of both lock-free read and historical read. In the above example, if we specify timestamp `5` .

Handling Conflicts

Conflicts are identified by checking the `lock` column. A row can have many versions of data, but it can have at most one lock at any time.

When we are performing a write operation, we try to lock every affected row in the `Prewrite` phase. If we failed to lock some of these rows, the whole transaction will be rolled back. Using an optimistic lock algorithm, sometimes Percolator's transactional write may encounter performance regressions in scenarios where conflicts occur frequently.

To roll back a row, just simply remove its lock and its corresponding value in `data` column.

Tolerating crashes

Percolator has the ability to survive crashes without breaking data integrity.

First, let's see what will happen after a crash. A crash may happen during `Prewrite` , `Commit` or between these two phases. We can simply divide these conditions into two types: before committing the primary, or after committing the primary.

So, when a transaction `T1` (either reading or writing) finds that a row `R1` has a lock which belongs to an earlier lock.

- If the primary lock has disappeared and there's a record `data @ T0.start_ts` in the `write` column, it means that `T0` has been successfully committed. Then row `R1` 's stale lock can also be committed. Usually we call this `rolling forward` . After this, the new transaction `T1` resumes.
- If the primary lock has disappeared with nothing left, it means the transaction has been rolled back. Then row `R1` 's stale lock should also be rolled back. After this, `T1` resumes.
- If the primary lock exists but it's too old (we can determine this by saving the wall time to locks), it indicates that the transaction has crashed before being committed or rolled back. Roll back `T1` and it will resume.
- Otherwise, we consider transaction `T0` to be still running. `T1` can rollback itself, or try to wait for a while to see whether `T0` will be committed before `T1.start_ts` .

Percolator in TiKV

TiKV is a distributed transactional key-value storage engine. Each key-value pair can be regarded as a row in Percolator.

TiKV internally uses RocksDB, a key-value storage engine library, to persist data to local disk. RocksDB's atomic write batch and TiKV's transaction scheduler make it atomic to

RocksDB provides a feature named *Column Family* (hereafter referred to as *CF*). An instance of RocksDB may have multiple CFs, and each CF is a separated key namespace and has its own LSM-Tree. However different CFs in the same RocksDB instance uses a common WAL, providing the ability to write to different CFs atomically.

We divide a RocksDB instance to three CFs: `CF_DEFAULT`, `CF_LOCK` and `CF_WRITE`, which corresponds to Percolator's `data` column, `lock` column and `write` column respectively. There's an extra CF named `CF_RAFT` which is used to save some metadata of Raft, but that's beside our topic. The `notify` and `ack_O` columns are not present in TiKV, because for now TiKV doesn't need the ability of incremental processing.

Then, we need to represent different versions of a key. We can simply compound a key and a timestamp as an internal key, which can be used in RocksDB. However, since a key can have at most one lock at a time, we don't need to add a timestamp to the key in `CF_LOCK`. Hence the content of each CF:

- `CF_DEFAULT`: `(key, start_ts)` -> `value`
- `CF_LOCK`: `key` -> `lock_info`
- `CF_WRITE`: `(key, commit_ts)` -> `write_info`

Our approach to compound user keys and timestamps together is:

1. Encode the user key to memcomparable

3. Append the encoded timestamp to the encoded key.

For example, key `"key1"` and timestamp `3` will be encoded as `"key1\x00\x00\x00\x00\xfb\xff\xff\xff\xff\xff\xff\xff\xfe"`, where the first 9 bytes is the memcomparable-encoded key and the remaining 8 bytes is the inverted timestamp in big-endian. In this way, different versions of the same key are always adjacent in RocksDB; and for each key, newer versions are always before older ones.

There are some differences between TiKV and the Percolator's paper. In TiKV, records in `CF_WRITE` has four different types: `Put`, `Delete`, `Rollback` and `Lock`. Only `Put` records need a corresponding value in `CF_DEFAULT`. When rolling back transactions, we don't simply remove the lock but writes a `Rollback` record in `CF_WRITE`. Different from Percolator's lock, the `Lock` type of write records in TiKV is produced by queries like `SELECT ... FOR UPDATE` in TiDB. For keys affected by this query, they are not only the objects for read, but the reading is also part of a write operation. To guarantee to be in snapshot-isolation, we make it acts like a write operation (though it doesn't write anything) to ensure the keys are locked and won't change before committing the transaction.

TiKV Version 6.5

Docs

- What's New
- Get Started
- Deploy
- Develop
- Reference

Deep Dive

- Consensus algorithm
- Key-value engine
- Distributed transaction
- Scalability
- Remote Procedure Calls (RPC)
- Resource scheduling
- Distributed SQL over TiKV
- Testing

Community

- Contribute
- Chat
- Branding
- Adopters
- CNCF

Blog

- Why SHAREit Selects TiKV for Data Storage for Its 2.4-Billion-User Business
- NodeJS Client and CNCF LFX Mentorship Experience
- Getting Started with JuiceFS Using TiKV
- Looking Back at the LFX Mentorship Program Spring '21: My Journey to Becoming a TiKV Contributor
- TiKV Rust Client - 0.1 release
- More...

TiKV is an Apache 2.0 licensed open source distributed

TiKV Version 6.5