RocksDB StateStore
Changelog Based Checkpointing

Part I: Design sketch

Motivation

We have identified state checkpointing latency as one of the major performance bottlenecks in previous benchmarks.
In general checkpoint latency contributes to a large percent of task duration and it is also a major source of variation in latency, this document proposes changelog checkpointing to reduce this latency.

Limitations of Current RocksDB State Store Checkpointing Implementation

1. **Synchronous operation in batch barrier:** Before uploading the state of a RocksDB instance, there are several synchronous operations performed as part of the task execution
- write writeBatch to db
- flush memtables
- explicit compaction if configured
- pause background flush/compaction
- create checkpoint
- upload to DFS

All these operations make the commit latency unpredictable and brings variance to data processing latency.
2. **Write amplification**: The size of the uploaded artifacts is not proportional to the size of the data change because SST files are merged during LSM compaction. The total bytes written is unpredictable and larger than the actual data change.(33x in level style compaction according to RocksDB doc). Write amplification is a serious problem for RocksDB StateStore because files are synced to DFS instead of local disk.
3. **Write stall**: write stall occurs because background flush and compaction gets interrupted frequently.

This causes undesirable properties of current checkpointing mechanism
1. The checkpoint latency is not predictable: even if the state size and input rate doesn't change, file sizes uploaded and time waiting for blocking operation is undeterministic.
2. The checkpointing is not incremental: as state size grows, checkpoint latency increases.

Requirements

Functional requirements

MUST:

- State commit must be incremental, only persist the state change in micro-batch commit.
- Be backward compatible with existing RocksDB StateStore checkpointing format so that customers can migrate to the new checkpointing mechanism while preserving the existing checkpoint state.
- Snapshotting in the background to avoid blocking task execution. Allow users to configure the snapshot interval to tradeoff between failure recovery and resource usage.
- Atomic and idempotent commits are still required to guarantee correctness under at-least-once task execution.
- State store metrics that are still relevant should be maintained.
- Consistent performance despite the changing state size.

Proposal sketch

1. The key idea is to make the state of a microbatch durable by syncing the change log instead of the state snapshot to the checkpoint directory.
2. Any version can be reconstructed from replaying change logs on a snapshot of an older version.
3. Do version snapshotting in the background to enable changelog purging and faster error recovery.

The write amplification problem is solved by only uploading the change log to DFS. Because snapshots will be captured less frequently, many small intermediate SST files won't be uploaded.

Expensive operations like flushing memtables to disk and pausing background compaction won't be synchronous in state checkpointing because RocksDB snapshots don't need to be captured.

The new checkpointing mechanism is backward compatible with existing RocksDB state store checkpoint directory because every existing version has a corresponding snapshot.

Part II: Detailed design

Mechanism

Changelog checkpoint: Upon each put() delete() call to local rocksdb instance, log the operation to a changelog file. During the state change commit, sync the compressed change log of the current batch to DFS as checkpointDir/{version}.delta.

Version reconstruction: For version j, find latest snapshot i.zip such that i <= j, load

snapshot i, and replay i+1.delta ~ j.delta. This is used in loading the initial state as well as creating the latest version snapshot. Note: If a query is shutdown without exception, there won't be changelog replay during query restart because a maintenance task is executed before the state store instance is unloaded.
Background snapshot: A maintenance thread in executors will launch maintenance tasks periodically. Inside the maintenance task, sync the latest RocksDB local snapshot to DFS as checkpointDir/{version}.zip. Snapshot enables faster failure recovery and allows old versions to be purged.
Garbage collection: Inside the maintenance task, delete snapshot and delta files from DFS for versions that is out of retention range(default retained version number is 100)

Changelog Format

There are 2 types of records, put and delete.
A put record is written as: | key length | key content | value length | value content |
A delete record is written as: | key length | key content | -1 |
Write an Int -1 to signal the end of file.
The overall changelog format is: | put record | delete record | ... | put record | eof |