RocksDB

rocksdb.org

An open-source, embeddable, persistent key-value store for fast storage environments

353 Contributors

10,064 Stars

2,200 Forks

High Performance

Optimized for Flash Storage

Extremely Adaptable

github.com/facebook/rocksdb

Proud sponsors of Percona Live 2018!

Maysam Yabandeh

WritePrepared Transactions

Posted December 19, 2017

RocksDB supports both optimistic and pessimistic concurrency controls. The pessimistic transactions make use of locks to provide isolation between the transactions. The default write policy in pessimistic transactions is *WriteCommitted*, which means that the data is written to the DB, i.e., the memtable, only after the transaction is committed. This policy simplified the implementation but came with some limitations in throughput, transaction size, and variety in supported isolation levels. In the below, we explain these in detail and present the other write policies, *WritePrepared* and *WriteUnprepared*. We then dive into the design of *WritePrepared* transactions.

WritePrepared are to be announced as production-ready soon.

WriteCommitted, Pros and Cons

With *WriteCommitted* write policy, the data is written to the memtable only after the transaction commits. This greatly simplifies the read path as any data that is read by other transactions can be assumed to be committed. This write policy, however, implies that the writes are buffered in memory in the meanwhile. This makes memory a bottleneck for large transactions. The delay of the commit phase in 2PC (two-phase commit) also becomes noticeable since most of the work, i.e., writing to memtable, is done at the commit phase. When the commit of multiple transactions are done in a serial fashion, such as in 2PC implementation of MySQL, the lengthy commit latency becomes a major contributor to lower throughput. Moreover this write policy cannot provide weaker isolation levels, such as READ UNCOMMITTED, that could potentially provide higher throughput for some applications.

Alternatives: *WritePrepared* and *WriteUnprepared*

To tackle the lengthy commit issue, we should do memtable writes at earlier phases of 2PC so that the commit phase become lightweight and fast. 2PC is composed of Write stage, where the

transaction ::Put is invoked, the prepare phase, where ::Prepare is invoked (upon which the DB promises to commit the transaction if later is requested), and commit phase, where ::Commit is invoked and the transaction writes become visible to all readers. To make the commit phase lightweight, the memtable write could be done at either ::Prepare or ::Put stages, resulting into *WritePrepared* and *WriteUnprepared* write policies respectively. The downside is that when another transaction is reading data, it would need a way to tell apart which data is committed, and if they are, whether they are committed before the transaction's start, i.e., in the read snapshot of the transaction. *WritePrepared* would still have the issue of buffering the data, which makes the memory the bottleneck for large transactions. It however provides a good milestone for transitioning from *WriteCommitted* to *WriteUnprepared* write policy. Here we explain the design of *WritePrepared* policy. We will cover the changes that make the design to also supported *WriteUnprepared* in an upcoming post.

WritePrepared in a nutshell

These are the primary design questions that needs to be addressed: 1) How do we identify the key/values in the DB with transactions that wrote them? 2) How do we figure if a key/value written by transaction Txn_w is in the read snapshot of the reading transaction Txn_r? 3) How do we rollback the data written by aborted transactions?

With *WritePrepared*, a transaction still buffers the writes in a write batch object in memory. When 2PC ::Prepare is called, it writes the in-memory write batch to the WAL (write-ahead log) as well as to the memtable(s) (one memtable per column family); We reuse the existing notion of sequence numbers in RocksDB to tag all the key/values in the same write batch with the same sequence number, prepare_seq, which is also used as the identifier for the transaction. At commit time, it writes a commit marker to the WAL, whose sequence number, commit_seq, will be used as the commit timestamp of the transaction. Before releasing the commit sequence number to the readers, it stores a mapping from prepare_seq to commit_seq in an in-memory data structure that we call *CommitCache*. When a transaction reading values from the DB (tagged with prepare_seq) it makes use of the *CommitCache* to figure if commit_seq of the value is in its read snapshot. To

rollback an aborted transaction, we apply the status before the transaction by making another write that cancels out the writes of the aborted transaction.

The *CommitCache* is a lock-free data structure that caches the recent commit entries. Looking up the entries in the cache must be enough for almost all th transactions that commit in a timely manner. When evicting the older entries from the cache, it still maintains some other data structures to cover the corner cases for transactions that takes abnormally too long to finish. We will cover them in the design details below.

Preliminary Results

The full experimental results are to be reported soon. Here we present the improvement in tps observed in some preliminary experiments with MyRocks:

- sysbench update-noindex: 25%
- sysbench read-write: 7.6%
- linkbench: 3.7%

Andrew Kryczka

Auto-tuned Rate Limiter

Posted December 18, 2017

Introduction

Our rate limiter has been hard to configure since users need to pick a value that is low enough to prevent background I/O spikes, which can impact user-visible read/write latencies. Meanwhile, picking too low a value can cause memtables and L0 files to pile up, eventually leading to writes stalling. Tuning the rate limiter has been especially difficult for users whose DB instances have different workloads, or have workloads that vary over time, or commonly both.

To address this, in RocksDB 5.9 we released a dynamic rate limiter that adjusts itself over time according to demand for background I/O. It can be enabled simply by passing auto_tuned=true in the NewGenericRateLimiter() call. In this case rate_bytes_per_sec will indicate the upper-bound of the window within which a rate limit will be picked dynamically. The chosen rate limit will be much lower unless absolutely necessary, so setting this to the device's maximum throughput is a reasonable choice on dedicated hosts.

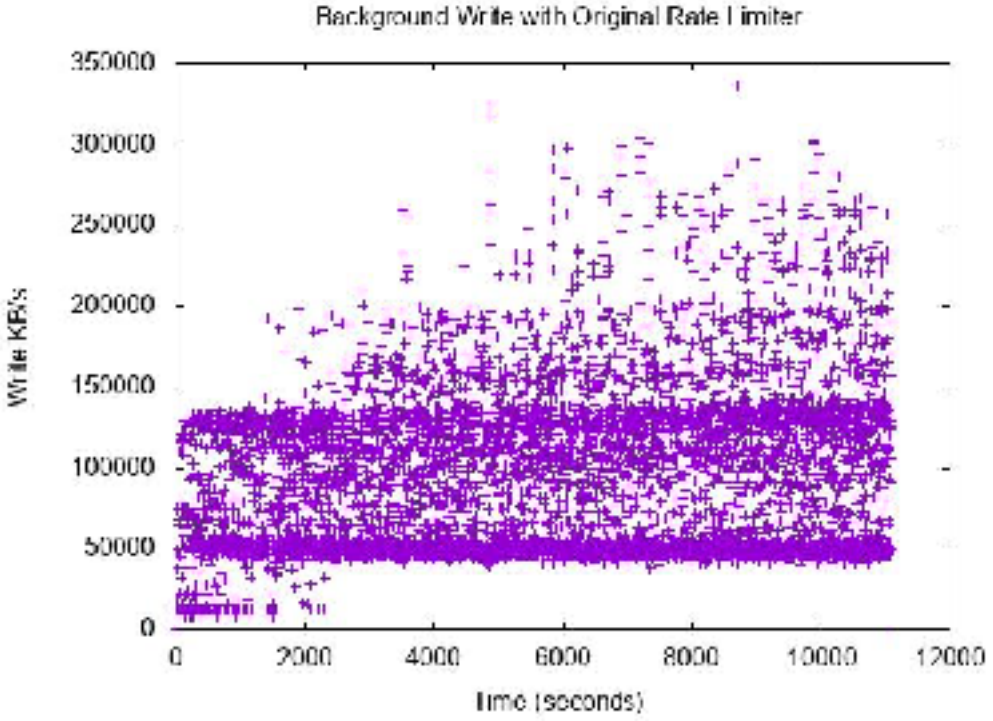
Algorithm

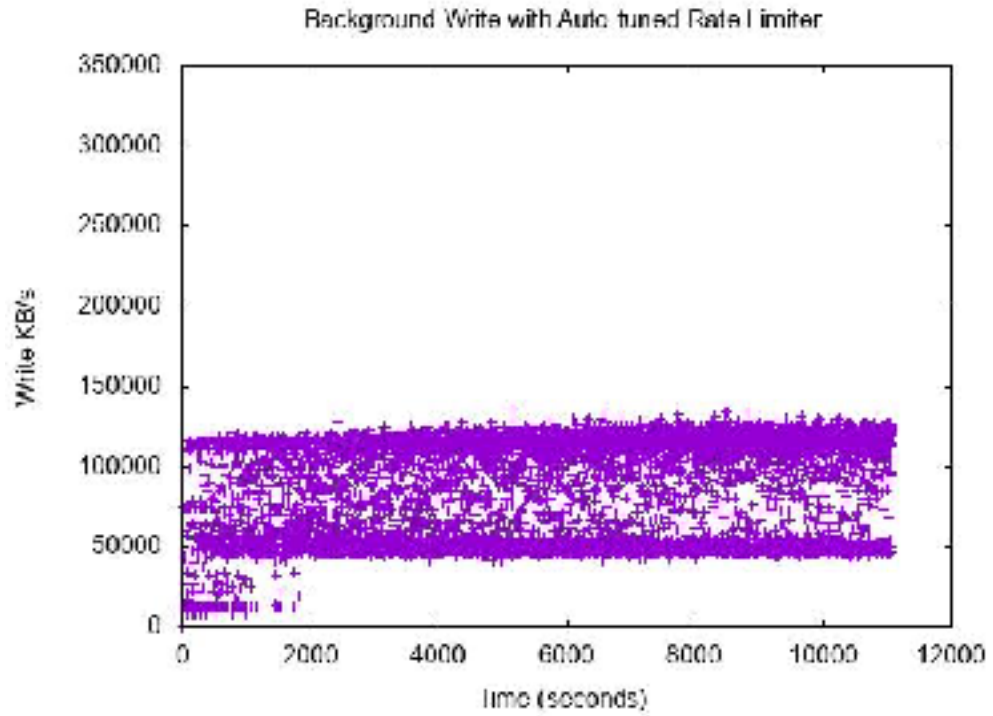
We use a simple multiplicative-increase, multiplicative-decrease algorithm. We measure demand for background I/O as the ratio of intervals where the rate limiter is drained. There are low and high watermarks for this ratio, which will trigger a change in rate limit when breached. The rate limit can move within a window bounded by the user-specified upper-bound, and a lower-bound that we derive internally. Users can expect this lower bound to be 1-2 orders of magnitude less than the provided upper-bound (so don't provide INT64_MAX as your upper-bound), although it's subject to change.

Benchmark Results

Data is ingested at 10MB/s and the rate limiter was created with 1000MB/s as its upper bound. The dynamically chosen rate limit hovers around 125MB/s. The other clustering of points at

50MB/s is due to number of compaction threads being reduced to one when there's no compaction pressure.





The following graph summarizes the above two time series graphs in CDF form. In particular, notice the p90 - p100 for background write rate are significantly lower with auto-tuned rate limiter enabled.