Block Cache

Edit New Page

Jump to bottom

Pavel Paulau edited this page on May 19, 2017 · 23 revisions

Block cache is where RocksDB caches data in memory for reads. User can pass in a `Cache` object to a RocksDB instance with a desired capacity (size). A `Cache` object can be shared by multiple RocksDB instances in the same process, allowing users to control the overall cache capacity. The block cache stores uncompressed blocks. Optionally user can set a second block cache storing compressed blocks. Reads will fetch data blocks first from uncompressed block cache, then compressed block cache. The compressed block cache can be a replacement of OS page cache, if Direct-IO is used.

There are two cache implementations in RocksDB, namely `LRUCache` and `ClockCache`. Both types of the cache are sharded to mitigate lock contention. Capacity is divided evenly to each shard and shards don't share capacity. By default each cache will be sharded into at most 64 shards, with each shard has no less than 512k bytes of capacity.

Usage

Out of box, RocksDB will use LRU-based block cache implementation with 8MB capacity. To set a customized block cache, call `NewLRUCache()` or `NewClockCache()` to create a cache object, and set it to block based table options. Users can also have their own cache implementation by implementing the `Cache` interface.

```
std::shared_ptr<Cache> cache = NewLRUCache(capacity);
BlockBasedTableOptions table_options;
table_options.block_cache = cache;
Options options;
options.table_factory.reset(new BlockBasedTableFactory(table_options));
```

To set compressed block cache:

```
table_options.block_cache_compressed = another_cache;
```

RocksDB will create the default block cache if `block_cache` is set to `nullptr`. To disable block cache completely:

```
table_options.no_block_cache = true;
```

LRU Cache

Out of box, RocksDB will use LRU-based block cache implementation with 8MB capacity. Each shard of the cache maintains its own LRU list and its own hash table for lookup. Synchronization is done via a per-shard mutex. Both lookup and insert to the cache would require a locking mutex of the shard. User can create a LRU cache by calling `NewLRUCache()`. The function provides several useful options to set to the cache:

- `capacity`: Total size of the cache.
- `num_shard_bits`: The number of bits from cache keys to be use as shard id. The cache will be sharded into `2^num_shard_bits` shards.
- `strict_capacity_limit`: In rare case, block cache size can go larger than its capacity. This is when ongoing reads or iterations over DB pin blocks in block cache, and the total size of pinned blocks exceeds the capacity. If there are further reads which try to insert blocks into block cache, if `strict_capacity_limit=false` (default), the cache will fail to respect its capacity limit and allow the insertion. This can create undesired OOM error that crashes the DB if the host don't have enough memory. Setting the option to `true` will reject further insertion to the cache and fail the read or iteration. The option works on per-shard basis, means it is possible one shard is rejecting insert when it is full, while another shard still have extra unpinned space.
- `high_pri_pool_ratio`: The ratio of capacity reserved for high priority blocks. See Caching index and filter blocks section below for more information.

Clock Cache

`ClockCache` implements the CLOCK algorithm. Each shard of clock cache maintains a circular list of cache entries. A clock handle runs over the circular list looking for unpinned entries to evict, but also giving each entry a second chance to stay in cache if it has been used since last scan. A `tbb::concurrent_hash_map` is used for lookup.

The benefit over `LRUCache` is it has finer-granularity locking. In case of LRU cache, the per-shard mutex has to be locked even on lookup, since it needs to update its LRU-list. Looking up from a clock cache won't require locking per-shard mutex, but only looking up the concurrent hash map, which has fine-granularity locking. Only inserts needs to lock the per-shard mutex. With clock cache we see boost of read throughput over LRU cache in contented environment (see inline comments in `cache/clock_cache.cc` for benchmark setup):

```
  Threads Cache     Cache              ClockCache               LRUCache
          Size  Index/Filter Throughput(MB/s)   Hit Throughput(MB/s)    Hit
```

32	2GB	yes	466.7	85.9%	433.7	86.5%
32	2GB	no	529.9	72.7%	532.7	73.9%
32	64GB	yes	649.9	99.9%	507.9	99.9%
32	64GB	no	740.4	99.9%	662.8	99.9%
16	2GB	yes	278.4	85.9%	283.4	86.5%
16	2GB	no	318.6	72.7%	335.8	73.9%
16	64GB	yes	391.9	99.9%	353.3	99.9%
16	64GB	no	433.8	99.8%	419.4	99.8%

To create a clock cache, call `NewClockCache()`. To make clock cache available, RocksDB needs to be linked with Intel TBB library. Again there are several options users can set when creating a clock cache:

- `capacity` : Same as LRUCache.
- `num_shard_bits` : Same as LRUCache.
- `strict_capacity_limit` : Same as LRUCache.

Caching Index and Filter Blocks

By default index and filter blocks are cached outside of block cache, and users won't be able to control how much memory should be use to cache these blocks, other than setting `max_open_files`. Users can opt to cache index and filter blocks in block cache, which allows for better control of memory used by RocksDB. To cache index and filter blocks in block cache:

```
BlockBasedTableOptions table_options;
table_options.cache_index_and_filter_blocks = true;
```

By putting index and filter blocks in block cache, these blocks have to compete against data blocks for staying in cache. Although index and filter blocks are being accessed more frequently than data block, there are scenarios where these blocks can be thrashing. This is undesired because index and filter blocks tend to be much larger than data blocks, and they are usually of higher value to stay in cache. There are two options to tune to mitigate the problem:

- `cache_index_and_filter_blocks_with_high_priority` : Set priority to high for index and filter blocks in block cache. It only affect `LRUCache` so far, and need to use together with `high_pri_pool_ratio` when calling `NewLRUCache()` . If the feature is enabled, LRU-list in LRU cache will be split into two parts, one for high-pri blocks and one for low-pri blocks. Data blocks will be inserted to the head of low-pri pool. Index and filter blocks will be inserted to the head of high-pri pool. If the total usage in the high-pri pool exceed `capacity * high_pri_pool_ratio` , the block at the tail of high-pri pool will overflow to the head of low-pri pool, after which it will compete against data blocks to stay in cache. Eviction will start from the tail of low-pri pool.

- `pin_l0_filter_and_index_blocks_in_cache` : Pin level-0 file's index and filter blocks in block cache, to avoid them from being evicted. Level-0 index and filters are typically accessed more frequently. Also they tend to be smaller in size so hopefully pinning them in cache won't consume too much capacity.

Simulated Cache

`SimCache` is an utility to predict cache hit rate if cache capacity or number of shards is changed. It wraps around the real `Cache` object that the DB is using, and runs a shadow LRU cache simulating the given capacity and number of shards, and measure cache hits and misses of the shadow cache. The utility is useful when user wants to open a DB with, say, 4GB cache size, but would like to know what the cache hit rate will become if cache size enlarge to, say, 64GB. To create a simulated cache:

```
// This cache is the actual cache use by the DB.
std::shared_ptr<Cache> cache = NewLRUCache(capacity);
// This is the simulated cache.
std::shared_ptr<Cache> sim_cache = NewSimCache(cache, sim_capacity,
sim_num_shard_bits);
BlockBasedTableOptions table_options;
table_options.block_cache = sim_cache;
```

The extra memory overhead of the simulated cache is less than 2% of `sim_capacity` .

Statistics

A list of block cache counters can be accessed through `Options.statistics` if it is non-null.

```
// total block cache misses
// REQUIRES: BLOCK_CACHE_MISS == BLOCK_CACHE_INDEX_MISS +
//                               BLOCK_CACHE_FILTER_MISS +
//                               BLOCK_CACHE_DATA_MISS;
```

```
BLOCK_CACHE_MISS = 0,
// total block cache hit
// REQUIRES: BLOCK_CACHE_HIT == BLOCK_CACHE_INDEX_HIT +
//                              BLOCK_CACHE_FILTER_HIT +
//                              BLOCK_CACHE_DATA_HIT;
BLOCK_CACHE_HIT,
// # of blocks added to block cache.
BLOCK_CACHE_ADD,
// # of failures when adding blocks to block cache.
BLOCK_CACHE_ADD_FAILURES,
// # of times cache miss when accessing index block from block cache.
BLOCK_CACHE_INDEX_MISS,
// # of times cache hit when accessing index block from block cache.
BLOCK_CACHE_INDEX_HIT,
// # of times cache miss when accessing filter block from block cache.
BLOCK_CACHE_FILTER_MISS,
// # of times cache hit when accessing filter block from block cache.
BLOCK_CACHE_FILTER_HIT,
// # of times cache miss when accessing data block from block cache.
BLOCK_CACHE_DATA_MISS,
// # of times cache hit when accessing data block from block cache.
BLOCK_CACHE_DATA_HIT,
// # of bytes read from cache.
BLOCK_CACHE_BYTES_READ,
// # of bytes written into cache.
BLOCK_CACHE_BYTES_WRITE,
```

See also: Memory-usage-in-RocksDB#block-cache

        - Compaction Filter
        - Read-Modify-Write Operator
        - Column Families
        - Creating and Ingesting SST files
        - Single Delete
        - Low Priority Write
        - Time to Live (TTL) Support
        - Transactions
    - Known Issues
    - Compatibility Between Releases
    - Options
        - Setup Options and Basic Tuning
        - Option String and Option Map
        - RocksDB Options File
    - Compression
        - Dictionary Compression
    - IO
        - Rate Limiter
        - Direct I/O
    - Background Error Handling
    - MANIFEST
    - Block Cache
    - MemTable
    - Huge Page TLB Support
    - Write Ahead Log
        - Write Ahead Log File Format
        - WAL Recovery Modes
    - Write Buffer Manager
    - Compaction
        - Leveled Compaction
        - Universal compaction style
        - FIFO compaction style
        - Manual Compaction
        - Sub-Compaction
    - Managing Disk Space Utilization
    - SST File Formats
        - Block-based Table Format
        - PlainTable Format
        - Bloom Filter
        - Data Block Hash Index
    - Logging and Monitoring
        - Logger
        - Statistics
        - Perf Context and IO Stats Context
        - EventListener
- Tools / Utilities

- Tools / Utilities
  - Administration and Data Access Tool
  - Benchmarking Tools
  - How to Backup RocksDB?
  - Checkpoints
  - How to persist in-memory RocksDB database
  - Stress Test
  - Third-party language bindings
  - RocksDB Trace, Replay, and Analyzer
- Implementation Details
  - Delete Stale Files
  - Partitioned Index/Filters
  - WritePrepared-Transactions
  - WriteUnprepared-Transactions
  - How we keep track of live SST files
  - How we index SST
  - Merge Operator Implementation
  - Choose Level Compaction Files
  - RocksDB Repairer
  - Two Phase Commit
  - Iterator's Implementation
  - Simulation Cache
  - Persistent Read Cache
- RocksJava
  - RocksJava Basics
  - RocksJava Performance on Flash Storage
  - JNI Debugging
  - RocksJava API TODO
- Lua
  - Lua CompactionFilter
- Performance
  - Performance on Flash Storage
  - In Memory Workload Performance
  - Read-Modify-Write Performance
  - Delete A Range Of Keys
  - Write Stalls
  - Pipelined Write
  - Tuning Guide
  - Memory usage in RocksDB
  - Speed-Up DB Open
  - Implement Queue Service Using RocksDB
- Misc
  - Building on Windows
  - Open Projects
  - Talks
  - Publication

- Publication
- Features Not in LevelDB
- How to ask a performance-related question?
- Articles about Rocks

Clone this wiki locally

https://github.com/facebook/rocksdb.wiki.git