Architecture of Enterprise Applications 12
MySQL Optimization II

Haopeng Chen

***RE**liable, **IN**telligent and **S**calable Systems Group (**REINS**)*
Shanghai Jiao Tong University
Shanghai, China
http://reins.se.sjtu.edu.cn/~chenhp
e-mail: chen-hp@sjtu.edu.cn

Contents and Objectives

- Contents
  - Optimizing for InnoDB Tables
  - Optimizing for MyISAM Tables
  - Optimizing for MEMORY Tables
  - From: https://dev.mysql.com/doc/refman/8.0/en/optimization.html

- Objectives
  - 能够根据数据访问的具体场景，设计数据库性能调优方案，包括索引优化、缓存设置和参数调优等

Optimizing for InnoDB Tables

- Optimizing Storage Layout for InnoDB Tables
  - Once your data reaches a stable size, or a growing table has increased by tens or some hundreds of megabytes, consider using the OPTIMIZE TABLE statement to reorganize the table and compact any wasted space.
    - OPTIMIZE TABLE copies the data part of the table and rebuilds the indexes. The benefits come from improved packing of data within indexes, and reduced fragmentation within the tablespaces and on disk.
  - In InnoDB, having a long PRIMARY KEY (either a single column with a lengthy value, or several columns that form a long composite value) wastes a lot of disk space.
    - Create an AUTO_INCREMENT column as the primary key if your primary key is long, or index a prefix of a long VARCHAR column instead of the entire column.
  - Use the VARCHAR data type instead of CHAR to store variable-length strings or for columns with many NULL values.
  - For tables that are big, or contain lots of repetitive text or numeric data, consider using COMPRESSED row format.

Optimizing for InnoDB Tables

- Optimizing InnoDB Transaction Management
  - The default MySQL setting AUTOCOMMIT=1 can impose performance limitations on a busy database server.
    - Where practical, wrap several related data change operations into a single transaction, by issuing SET AUTOCOMMIT=0 or a START TRANSACTION statement, followed by a COMMIT statement after making all the changes.
  - Alternatively, for transactions that consist only of a single SELECT statement, turning on AUTOCOMMIT helps InnoDB to recognize read-only transactions and optimize them.
  - Avoid performing rollbacks after inserting, updating, or deleting huge numbers of rows.
    - If a big transaction is slowing down server performance, rolling it back can make the problem worse, potentially taking several times as long to perform as the original data change operations.
    - Killing the database process does not help, because the rollback starts again on server startup.

Optimizing for InnoDB Tables

- Optimizing InnoDB Transaction Management
  - To minimize the chance of this issue occurring:
    - Increase the size of the buffer pool so that all the data change changes can be cached rather than immediately written to disk.
    - Set innodb_change_buffering=all so that update and delete operations are buffered in addition to inserts.
    - Consider issuing COMMIT statements periodically during the big data change operation, possibly breaking a single delete or update into multiple statements that operate on smaller numbers of rows.
  - A long-running transaction can prevent InnoDB from purging data that was changed by a different transaction.
    - When rows are modified or deleted within a long-running transaction, other transactions using the READ COMMITTED and REPEATABLE READ isolation levels have to do more work to reconstruct the older data if they read those same rows.
    - When a long-running transaction modifies a table, queries against that table from other transactions do not make use of the covering index technique.

Optimizing for InnoDB Tables

- Optimizing InnoDB Read-Only Transactions
  - InnoDB can avoid the overhead associated with setting up the transaction ID (TRX_ID field) for transactions that are known to be read-only.
  - InnoDB detects read-only transactions when:
    - The transaction is started with the START TRANSACTION READ ONLY statement.
    - The autocommit setting is turned on, so that the transaction is guaranteed to be a single statement, and the single statement making up the transaction is a "non-locking" SELECT statement.
    - The transaction is started without the READ ONLY option, but no updates or statements that explicitly lock rows have been executed yet.
  - Thus, for a read-intensive application such as a report generator, you can tune a sequence of InnoDB queries
    - by grouping them inside START TRANSACTION READ ONLY and COMMIT, or
    - by turning on the autocommit setting before running the SELECT statements, or
    - simply by avoiding any data change statements interspersed with the queries.

Optimizing for InnoDB Tables

- Optimizing InnoDB Redo Logging
  - Make your redo log files big, even as big as the buffer pool.
    - The size and number of redo log files are configured using the innodb_log_file_size and innodb_log_files_in_group configuration options.
  - Consider increasing the size of the log buffer.
    - Log buffer size is configured using the innodb_log_buffer_size configuration option, which can be configured dynamically in MySQL 8.0.
  - Configure the innodb_log_write_ahead_size configuration option to avoid “read-on-write”.
    - Set innodb_log_write_ahead_size to match the operating system or file system cache block size.
    - Valid values for innodb_log_write_ahead_size are multiples of the InnoDB log file block size (2^n).
  - MySQL 8.0.11 introduced dedicated log writer threads for writing redo log records from the log buffer to the system buffers and flushing the system buffers to the redo log files
    - As of MySQL 8.0.22, you can enable or disable log writer threads using theinnodb_log_writer_threads variable.
  - Optimize the use of spin delay by user threads waiting for flushed redo.
    - innodb_log_wait_for_flush_spin_hwm: Defines the maximum average log flush time beyond which user threads no longer spin while waiting for flushed redo. The default value is 400 microseconds.
    - innodb_log_spin_cpu_abs_lwm: Defines the minimum amount of CPU usage below which user threads no longer spin while waiting for flushed redo. The value is expressed as a sum of CPU core usage.
    - innodb_log_spin_cpu_pct_hwm: Defines the maximum amount of CPU usage above which user threads no longer spin while waiting for flushed redo.
    - The innodb_log_spin_cpu_pct_hwm configuration option respects processor affinity. For example, if a server has 48 cores but the **mysqld** process is pinned to only four CPU cores, the other 44 CPU cores are ignored.

Optimizing for InnoDB Tables

- Bulk Data Loading for InnoDB Tables
  - When importing data into InnoDB, turn off autocommit mode, because it performs a log flush to disk for every insert.

```
SET autocommit=0;
    ... SQL import statements ...
COMMIT;
```

  - If you have UNIQUE constraints on secondary keys, you can speed up table imports by temporarily turning off the uniqueness checks during the import session:

```
SET unique_checks=0;
    ... SQL import statements ...
SET unique_checks=1;
```

    For big tables, this saves a lot of disk I/O because InnoDB can use its change buffer to write secondary index records in a batch. Be certain that the data contains no duplicate keys.

Optimizing for InnoDB Tables

- Bulk Data Loading for InnoDB Tables
  - If you have FOREIGN KEY constraints in your tables, you can speed up table imports by turning off the foreign key checks for the duration of the import session:

    ```
    SET foreign_key_checks=0;
       ... SQL import statements ...
    SET foreign_key_checks=1;
    ```

    For big tables, this can save a lot of disk I/O.
  - Use the multiple-row INSERT syntax to reduce communication overhead between the client and the server if you need to insert many rows:

    ```
    INSERT INTO yourtable VALUES (1,2), (5,5), ...;
    ```

    This tip is valid for inserts into any table, not just InnoDB tables.
  - When doing bulk inserts into tables with auto-increment columns, set innodb_autoinc_lock_mode to 2 (interleaved) instead of 1 (consecutive).

Optimizing for InnoDB Tables

- Optimizing InnoDB Queries
  - Because each InnoDB table has a primary key (whether you request one or not), specify a set of primary key columns for each table, columns that are used in the most important and time-critical queries.
  - Do not specify too many or too long columns in the primary key, because these column values are duplicated in each secondary index.
  - Do not create a separate secondary index for each column, because each query can only make use of one index.
    - If you have many queries for the same table, testing different combinations of columns, try to create a small number of concatenated indexes rather than a large number of single-column indexes.
  - If an indexed column cannot contain any NULL values, declare it as NOT NULL when you create the table.
  - You can optimize single-query transactions for InnoDB tables, Optimizing InnoDB Read-Only Transactions

Optimizing for InnoDB Tables

- Optimizing InnoDB DDL Operations
  - Use TRUNCATE TABLE to empty a table, not DELETE FROM *tbl_name*.
    - Foreign key constraints can make a TRUNCATE statement work like a regular DELETE statement, in which case a sequence of commands like DROP TABLE and CREATE TABLE might be fastest.
  - Because the primary key is integral to the storage layout of each InnoDB table,
    - and changing the definition of the primary key involves reorganizing the whole table,
    - always set up the primary key as part of the CREATE TABLE statement,
    - and plan ahead so that you do not need to ALTER or DROP the primary key afterward.

Optimizing for InnoDB Tables

- Optimizing InnoDB Disk I/O
  - If you follow best practices for database design and tuning techniques for SQL operations, but your database is still slow due to heavy disk I/O activity, consider these disk I/O optimizations.
  - If the Unix top tool or the Windows Task Manager shows that the CPU usage percentage with your workload is less than 70%, your workload is probably disk-bound.
  - Increase buffer pool size
    - When table data is cached in the InnoDB buffer pool, it can be accessed repeatedly by queries without requiring any disk I/O. Specify the size of the buffer pool with the innodb_buffer_pool_size option.
  - Adjust the flush method
    - If database write performance is an issue, conduct benchmarks with the innodb_flush_method parameter set to O_DSYNC.
  - Configure an fsync threshold
    - You can use the innodb_fsync_threshold variable to define a threshold value, in bytes.
    - Specifying a threshold to force smaller, periodic flushes may be beneficial in cases where multiple MySQL instances use the same storage devices.

Optimizing for InnoDB Tables

- Optimizing InnoDB Disk I/O
  - Use a noop or deadline I/O scheduler with native AIO on Linux
    - InnoDB uses the asynchronous I/O subsystem (native AIO) on Linux to perform read-ahead and write requests for data file pages. This behavior is controlled by the innodb_use_native_aio configuration option, which is enabled by default.
  - Use direct I/O on Solaris 10 for x86_64 architecture
    - To apply direct I/O only to InnoDB file operations rather than the whole file system, set innodb_flush_method = O_DIRECT.
  - Use raw storage for data and log files with Solaris 2.6 or later
    - Users of the Veritas file system VxFS should use the convosync=direct mount option.
  - Use additional storage devices
    - Additional storage devices could be used to set up a RAID configuration.
    - Alternatively, InnoDB tablespace data files and log files can be placed on different physical disks.

Optimizing for InnoDB Tables

- Optimizing InnoDB Disk I/O
  - Consider non-rotational storage
    - Non-rotational storage generally provides better performance for random I/O operations; and rotational storage for sequential I/O operations. When distributing data and log files across rotational and non-rotational storage devices, consider the type of I/O operations that are predominantly performed on each file.
    - Random I/O-oriented files typically include file-per-table and general tablespace data files, undo tablespace files, and temporary tablespace files.
    - Sequential I/O-oriented files include InnoDB system tablespace files (due to doublewrite buffering prior to MySQL 8.0.20 and change buffering), doublewrite files introduced in MySQL 8.0.20, and log files such as binary log files and redo log files.
  - Increase I/O capacity to avoid backlogs
    - If throughput drops periodically because of InnoDB checkpoint operations, consider increasing the value of the innodb_io_capacity configuration option.
    - Higher values cause more frequent flushing, avoiding the backlog of work that can cause dips in throughput.

Optimizing for InnoDB Tables

- Optimizing InnoDB Disk I/O
  - Lower I/O capacity if flushing does not fall behind
    - In a typical scenario where you could lower the option value, you might see a combination like this in the output from SHOW ENGINE INNODB STATUS:
      - History list length low, below a few thousand.
      - Insert buffer merges close to rows inserted.
      - Modified pages in buffer pool consistently well below innodb_max_dirty_pages_pct of the buffer pool.
      - Log sequence number - Last checkpoint is at less than 7/8 or ideally less than 6/8 of the total size of the InnoDB log files.
  - Store system tablespace files on Fusion-io devices
    - You can take advantage of a doublewrite buffer-related I/O optimization by storing the files that contain the doublewrite storage area on Fusion-io devices that support atomic writes.
  - Disable logging of compressed pages
    - If you are certain that the zlib version is not subject to change, disableinnodb_log_compressed_pages to reduce redo log generation for workloads that modify compressed data.

Optimizing for InnoDB Tables

- Optimizing InnoDB Configuration Variables
  - Controlling the types of data change operations for which InnoDB buffers the changed data, to avoid frequent small disk writes.
  - Turning the adaptive hash indexing feature on and off using the innodb_adaptive_hash_index option.
  - Setting a limit on the number of concurrent threads that InnoDB processes, if context switching is a bottleneck.
  - Controlling the amount of prefetching that InnoDB does with its read-ahead operations.
  - Increasing the number of background threads for read or write operations, if you have a high-end I/O subsystem that is not fully utilized by the default values
  - Controlling how much I/O InnoDB performs in the background.
  - Controlling the algorithm that determines when InnoDB performs certain types of background writes.

Optimizing for InnoDB Tables

- Optimizing InnoDB Configuration Variables
  - Taking advantage of multicore processors and their cache memory configuration, to minimize delays in context switching.
  - Preventing one-time operations such as table scans from interfering with the frequently accessed data stored in the InnoDB buffer cache.
  - Adjusting log files to a size that makes sense for reliability and crash recovery.
  - Configuring the size and number of instances for the InnoDB buffer pool, especially important for systems with multi-gigabyte buffer pools.
  - Increasing the maximum number of concurrent transactions, which dramatically improves scalability for the busiest databases.
  - Moving purge operations (a type of garbage collection) into a background thread.
  - Reducing the amount of switching that InnoDB does between concurrent threads, so that SQL operations on a busy server do not queue up and form a “traffic jam”.

Optimizing for InnoDB Tables

- Optimizing InnoDB for Systems with Many Tables
  - If you have configured non-persistent optimizer statistics (a non-default configuration),
    - InnoDB computes index cardinality values for a table the first time that table is accessed after startup, instead of storing such values in the table.
    - This step can take significant time on systems that partition the data into many tables.
    - Since this overhead only applies to the initial table open operation, to "warm up" a table for later use, access it immediately after startup by issuing a statement such as SELECT 1 FROM *tbl_name* LIMIT 1.
  - Optimizer statistics are persisted to disk by default, enabled by the innodb_stats_persistent configuration option.

Optimizing for MyISAM Tables

- Optimizing MyISAM Queries
  - To help MySQL better optimize queries, use ANALYZE TABLE
    - or run **myisamchk --analyze** on a table after it has been loaded with data.
  - To sort an index and data according to an index, use **myisamchk --sort-index --sort-records=1** .
  - Try to avoid complex SELECT queries on MyISAM tables that are updated frequently,
    - to avoid problems with table locking that occur due to contention between readers and writers.
  - MyISAM supports concurrent inserts:
    - If a table has no free blocks in the middle of the data file, you can INSERT new rows into it at the same time that other threads are reading from the table. If it is important to be able to do this, consider using the table in ways that avoid deleting rows.
  - For MyISAM tables that change frequently,
    - try to avoid all variable-length columns (VARCHAR, BLOB, and TEXT). The table uses dynamic row format if it includes even a single variable-length column.

Optimizing for MyISAM Tables

- Optimizing MyISAM Queries
  - It is normally not useful to split a table into different tables just because the rows become large.
    - In accessing a row, the biggest performance hit is the disk seek needed to find the first byte of the row. After finding the data, most modern disks can read the entire row fast enough for most applications.
  - Use ALTER TABLE ... ORDER BY *expr1*, *expr2*, ... if you usually retrieve rows in *expr1*, *expr2*, ... order.
    - By using this option after extensive changes to the table, you may be able to get higher performance.
  - If you often need to calculate results such as counts based on information from a lot of rows, it may be preferable to introduce a new table and update the counter in real time. An update of the following form is very fast:

    UPDATE *tbl_name* SET *count_col*=*count_col*+1 WHERE *key_col*=*constant*;
  - Use OPTIMIZE TABLE periodically to avoid fragmentation with dynamic-format MyISAM tables.
  - Declaring a MyISAM table with the DELAY_KEY_WRITE=1 table option makes index updates faster because they are not flushed to disk until the table is closed.

Optimizing for MyISAM Tables

- Bulk Data Loading for MyISAM Tables
  - For a MyISAM table, you can use concurrent inserts to add rows at the same time that SELECT statements are running, if there are no deleted rows in middle of the data file.
  - With some extra work, it is possible to make LOAD DATA run even faster for a MyISAM table when the table has many indexes. Use the following procedure:
    - Execute a FLUSH TABLES statement or a **mysqladmin flush-tables** command.
    - Use **myisamchk --keys-used=0 -rq *path/to/db/tbl_name*** to remove all use of indexes for the table.
    - Insert data into the table with LOAD DATA. This does not update any indexes and therefore is very fast.
    - If you intend only to read from the table in the future, use **myisampack** to compress it.
    - Re-create the indexes with **myisamchk -rq *path/to/db/tbl_name***. This creates the index tree in memory before writing it to disk, which is much faster than updating the index during LOAD DATA because it avoids lots of disk seeks. The resulting index tree is also perfectly balanced.
    - Execute a FLUSH TABLES statement or a **mysqladmin flush-tables** command.

Optimizing for MyISAM Tables

- Bulk Data Loading for MyISAM Tables
  - LOAD DATA performs the preceding optimization automatically if the MyISAM table into which you insert data is empty.
    - The main difference between automatic optimization and using the procedure explicitly is that you can let **myisamchk** allocate much more temporary memory for the index creation than you might want the server to allocate for index re-creation when it executes the LOAD DATA statement.
    - You can also disable or enable the nonunique indexes for a MyISAM table by using the following statements rather than **myisamchk**. If you use these statements, you can skip the FLUSH TABLES operations:

      ALTER TABLE *tbl_name* DISABLE KEYS;

      ALTER TABLE *tbl_name* ENABLE KEYS;

Optimizing for MyISAM Tables

- Bulk Data Loading for MyISAM Tables
  - To speed up INSERT operations that are performed with multiple statements for nontransactional tables, lock your tables:

    ```
    LOCK TABLES a WRITE;
    INSERT INTO a VALUES (1,23),(2,34),(4,33);
    INSERT INTO a VALUES (8,26),(6,29);
    ...
    UNLOCK TABLES;
    ```
  - This benefits performance because the index buffer is flushed to disk only once, after all INSERT statements have completed.
  - Locking also lowers the total time for multiple-connection tests, although the maximum wait time for individual connections might go up because they wait for locks.
  - Suppose that five clients attempt to perform inserts simultaneously as follows:

    Connection 1 does 1000 inserts
    Connections 2, 3, and 4 do 1 insert
    Connection 5 does 1000 inserts
  - If you do not use locking, connections 2, 3, and 4 finish before 1 and 5.
  - If you use locking, connections 2, 3, and 4 probably do not finish before 1 or 5, but the total time should be about 40% faster.

Optimizing for MyISAM Tables

- Optimizing REPAIR TABLE Statements
  - REPAIR TABLE for MyISAM tables is similar to using **myisamchk** for repair operations
  - Suppose that a **myisamchk** table-repair operation is done using the following options to set its memory-allocation variables:

    --key_buffer_size=128M --myisam_sort_buffer_size=256M

    --read_buffer_size=64M --write_buffer_size=64M
  - Some of those **myisamchk** variables correspond to server system variables:

myisamchk Variable	**System Variable**
key_buffer_size	key_buffer_size
myisam_sort_buffer_size	myisam_sort_buffer_size
read_buffer_size	read_buffer_size
write_buffer_size	none

Optimizing for MEMORY Tables

- Consider using MEMORY tables for
  - noncritical data that is accessed often, and is read-only or rarely updated.
- For best performance with MEMORY tables,
  - examine the kinds of queries against each table, and specify the type to use for each associated index, either a B-tree index or a hash index.
  - On the CREATE INDEX statement, use the clause USING BTREE or USING HASH.
    - B-tree indexes are fast for queries that do greater-than or less-than comparisons through operators such as > or BETWEEN.
    - Hash indexes are only fast for queries that look up single values through the = operator, or a restricted set of values through the IN operator.

Buffering and Caching

- InnoDB Buffer Pool Optimization
  - InnoDB maintains a storage area called the buffer pool for caching data and indexes in memory.
  - Configuring InnoDB Buffer Pool Prefetching (Read-Ahead)
  - Configuring Buffer Pool Flushing
  - Making the Buffer Pool Scan Resistant
  - Configuring Multiple Buffer Pool Instances
  - Saving and Restoring the Buffer Pool State
  - Configuring InnoDB Buffer Pool Size

Buffering and Caching

- Configuring InnoDB Buffer Pool Size
  - When increasing or decreasing innodb_buffer_pool_size, the operation is performed in chunks.
    - Chunk size is defined by the innodb_buffer_pool_chunk_size configuration option, which has a default of 128M. Buffer pool size must always be equal to or a multiple of innodb_buffer_pool_chunk_size * innodb_buffer_pool_instances.

```
shell> mysqld --innodb-buffer-pool-size=8G --innodb-buffer-pool-instances=16
mysql> SELECT @@innodb_buffer_pool_size/1024/1024/1024;
+----------------------------------------------------------+
|     @@innodb_buffer_pool_size/1024/1024/1024 |
+----------------------------------------------------------+
|                                     8.000000000000 |
+----------------------------------------------------------+

shell> mysqld --innodb-buffer-pool-size=9G --innodb-buffer-pool-instances=16
mysql> SELECT @@innodb_buffer_pool_size/1024/1024/1024;
+----------------------------------------------------------+
|     @@innodb_buffer_pool_size/1024/1024/1024 |
+----------------------------------------------------------+
|                                    10.000000000000 |
+----------------------------------------------------------+
```

Buffering and Caching

- Configuring Multiple Buffer Pool Instances
  - For systems with buffer pools in the multi-gigabyte range, dividing the buffer pool into separate instances can improve concurrency, by reducing contention as different threads read and write to cached pages.
  - Multiple buffer pool instances are configured using the innodb_buffer_pool_instances configuration option, and you might also adjust the innodb_buffer_pool_size value.
  - When the InnoDB buffer pool is large, many data requests can be satisfied by retrieving from memory.
  - To enable multiple buffer pool instances, set the innodb_buffer_pool_instances configuration option to a value greater than 1 (the default) up to 64 (the maximum).
    - This option takes effect only when you set innodb_buffer_pool_size to a size of 1GB or more. The total size you specify is divided among all the buffer pools. For best efficiency, specify a combination of innodb_buffer_pool_instances and innodb_buffer_pool_size so that each buffer pool instance is at least 1GB.

- Making the Buffer Pool Scan Resistant
  - Rather than using a strict LRU algorithm,
    - InnoDB uses a technique to minimize the amount of data that is brought into the buffer pool and never accessed again.
    - The goal is to make sure that frequently accessed ("hot") pages remain in the buffer pool, even as read-ahead and full table scans bring in new blocks that might or might not be accessed afterward.
  - Newly read blocks are inserted into the middle of the LRU list.
    - All newly read pages are inserted at a location that by default is 3/8 from the tail of the LRU list.
    - The pages are moved to the front of the list (the most-recently used end) when they are accessed in the buffer pool for the first time.
    - Thus, pages that are never accessed never make it to the front portion of the LRU list, and "age out" sooner than with a strict LRU approach.
    - This arrangement divides the LRU list into two segments, where the pages downstream of the insertion point are considered "old" and are desirable victims for LRU eviction.

- Configuring InnoDB Buffer Pool Prefetching (Read-Ahead)
  - A read-ahead request is an I/O request to prefetch multiple pages in the buffer pool asynchronously, in anticipation of impending need for these pages.
  - The requests bring in all the pages in one extent. InnoDB uses two read-ahead algorithms to improve I/O performance:
    - **Linear** read-ahead is a technique that predicts what pages might be needed soon based on pages in the buffer pool being accessed sequentially.
    - **Random** read-ahead is a technique that predicts when pages might be needed soon based on pages already in the buffer pool, regardless of the order in which those pages were read.
  - The SHOW ENGINE INNODB STATUS command displays statistics to help you evaluate the effectiveness of the read-ahead algorithm.
    - The information can be useful when fine-tuning the innodb_random_read_ahead setting.

- Configuring Buffer Pool Flushing
  - InnoDB performs certain tasks in the background, including flushing of dirty pages from the buffer pool.
    - Dirty pages are those that have been modified but are not yet written to the data files on disk.
  - In MySQL 8.0, buffer pool flushing is performed by page cleaner threads.
    - The number of page cleaner threads is controlled by the innodb_page_cleaners variable, which has a default value of 4.
    - However, if the number of page cleaner threads exceeds the number of buffer pool instances, innodb_page_cleaners is automatically set to the same value as innodb_buffer_pool_instances.
  - Buffer pool flushing is initiated when the percentage of dirty pages reaches the low water mark value defined by the innodb_max_dirty_pages_pct_lwm variable.
    - The default low water mark is 10% of buffer pool pages. A innodb_max_dirty_pages_pct_lwm value of 0 disables this early flushing behaviour.

Buffering and Caching

- Saving and Restoring the Buffer Pool State
  - To reduce the warmup period after restarting the server, InnoDB saves a percentage of the most recently used pages for each buffer pool at server shutdown and restores these pages at server startup.
  - The percentage of recently used pages that is stored is defined by the innodb_buffer_pool_dump_pct configuration option.
  - Operations related to saving and restoring the buffer pool state are described in the following topics:
    - Configuring the Dump Percentage for Buffer Pool Pages
    - Saving the Buffer Pool State at Shutdown and Restoring it at Startup
    - Saving and Restoring the Buffer Pool State Online
    - Displaying Buffer Pool Dump Progress
    - Displaying Buffer Pool Load Progress
    - Aborting a Buffer Pool Load Operation
    - Monitoring Buffer Pool Load Progress Using Performance Schema

Buffering and Caching

- The MyISAM Key Cache
  - To minimize disk I/O, MyISAM storage engine employs a cache mechanism to keep the most frequently accessed table blocks in memory:
    - For index blocks, a special structure called the key cache (or key buffer) is maintained. The structure contains a number of block buffers where the most-used index blocks are placed.
    - For data blocks, MySQL uses no special cache. Instead it relies on the native operating system file system cache.
  - To control the size of the key cache, use the key_buffer_size system variable.
    - If this variable is set equal to zero, no key cache is used. The key cache also is not used if the key_buffer_size value is too small to allocate the minimal number of block buffers (8).
    - When the key cache is not operational, index files are accessed using only the native file system buffering provided by the operating system. (In other words, table index blocks are accessed using the same strategy as that employed for table data blocks.)

Buffering and Caching

- Shared Key Cache Access
  - Threads can access key cache buffers simultaneously, subject to the following conditions:
    - A buffer that is not being updated can be accessed by multiple sessions.
    - A buffer that is being updated causes sessions that need to use it to wait until the update is complete.
    - Multiple sessions can initiate requests that result in cache block replacements, as long as they do not interfere with each other (that is, as long as they need different index blocks, and thus cause different cache blocks to be replaced).
  - Shared access to the key cache enables the server to improve throughput significantly.

Buffering and Caching

- Multiple Key Caches
  - To reduce key cache access contention further, MySQL also provides multiple key caches.
    - This feature enables you to assign different table indexes to different key caches.
  - Where there are multiple key caches, the server must know which cache to use when processing queries for a given MyISAM table.
    - By default, all MyISAM table indexes are cached in the default key cache.
    - To assign table indexes to a specific key cache, use the CACHE INDEX statement

```
mysql> CACHE INDEX t1, t2, t3 IN hot_cache;
+----------+--------------------------+-------------+------------+
| Table    | Op                       | Msg_type    | Msg_text   |
+----------+--------------------------+-------------+------------+
| test.t1  | assign_to_keycache       |      status |         OK |
| test.t2  | assign_to_keycache       |      status |         OK |
| test.t3  | assign_to_keycache       |      status |         OK |
+----------+--------------------------+-------------+------------+
```

Buffering and Caching

- Multiple Key Caches
  - For a busy server, you can use a strategy that involves three key caches:
    - A "hot" key cache that takes up 20% of the space allocated for all key caches. Use this for tables that are heavily used for searches but that are not updated.
    - A "cold" key cache that takes up 20% of the space allocated for all key caches. Use this cache for medium-sized, intensively modified tables, such as temporary tables.
    - A "warm" key cache that takes up 60% of the key cache space. Employ this as the default key cache, to be used by default for all other tables.
  - The following example assigns several tables each to hot_cache and cold_cache:

```
CACHE INDEX db1.t1, db1.t2, db2.t3 IN hot_cache
CACHE INDEX db1.t4, db2.t5, db2.t6 IN cold_cache
```

- Midpoint Insertion Strategy
  - By default, the key cache management system uses a simple LRU strategy for choosing key cache blocks to be evicted, but it also supports a more sophisticated method called the midpoint insertion strategy.
  - When using the midpoint insertion strategy, the LRU chain is divided into two parts: a hot sublist and a warm sublist.
    - The division point between two parts is not fixed, but the key cache management system takes care that the warm part is not "too short," always containing at least key_cache_division_limit percent of the key cache blocks.
    - key_cache_division_limit is a component of structured key cache variables, so its value is a parameter that can be set per cache.
  - The midpoint insertion strategy helps to improve performance when execution of a query that requires an index scan effectively pushes out of the cache all the index blocks corresponding to valuable high-level B-tree nodes.

Buffering and Caching

- Index Preloading
  - If there are enough blocks in a key cache to hold blocks of an entire index,
    - or at least the blocks corresponding to its nonleaf nodes, it makes sense to preload the key cache with index blocks before starting to use it.
    - Preloading enables you to put the table index blocks into a key cache buffer in the most efficient way: by reading the index blocks from disk sequentially.
  - To preload an index into a cache, use the LOAD INDEX INTO CACHE statement.
  - For example, the following statement preloads nodes (index blocks) of indexes of the tables t1 and t2:

```
mysql> LOAD INDEX INTO CACHE t1, t2 IGNORE LEAVES;
+---------+--------------+----------+----------+
| Table   | Op           | Msg_type | Msg_text |
+---------+--------------+----------+----------+
| test.t1 | preload_keys | status   | OK       |
| test.t2 | preload_keys | status   | OK       |
+---------+--------------+----------+----------+
```

Buffering and Caching

- Key Cache Block Size
  - It is possible to specify the size of the block buffers for an individual key cache using the key_cache_block_size variable.
    - This permits tuning of the performance of I/O operations for index files.
  - The best performance for I/O operations is achieved when the size of read buffers is equal to the size of the native operating system I/O buffers.
    - But setting the size of key nodes equal to the size of the I/O buffer does not always ensure the best overall performance.
    - When reading the big leaf nodes, the server pulls in a lot of unnecessary data, effectively preventing reading other leaf nodes.
  - To control the size of blocks in the .MYI index file of MyISAM tables, use the --myisam-block-size option at server startup.

- Restructuring a Key Cache
  - A key cache can be restructured at any time by updating its parameter values. For example:

    mysql> SET GLOBAL cold_cache.key_buffer_size=4*1024*1024;
  - If you assign to either the key_buffer_size or key_cache_block_size key cache component a value that differs from the component's current value,
    - the server destroys the cache's old structure and creates a new one based on the new values.
    - If the cache contains any dirty blocks, the server saves them to disk before destroying and re-creating the cache.
    - Restructuring does not occur if you change other key cache parameters.
  - Restructuring does not block queries that need to use indexes assigned to the cache.
    - Instead, the server directly accesses the table indexes using native file system caching. File system caching is not as efficient as using a key cache, so although queries execute, a slowdown can be anticipated.

Buffering and Caching

- Caching of Prepared Statements and Stored Programs
  - Prepared statements, both those processed at the SQL level (using the PREPARE statement) and those processed using the binary client/server protocol (using the mysql_stmt_prepare() C API function).
    - The max_prepared_stmt_count system variable controls the total number of statements the server caches. (The sum of the number of prepared statements across all sessions.)
  - Stored programs (stored procedures and functions, triggers, and events). In this case, the server converts and caches the entire program body.
    - The stored_program_cache system variable indicates the approximate number of stored programs the server caches per session.
  - The server maintains caches for prepared statements and stored programs on a per-session basis. Statements cached for one session are not accessible to other sessions. When a session ends, the server discards any statements cached for it.

作业六

- 请你根据上课内容，针对你在E-BookStore项目中的数据库设计，详细回答下列问题：
  1. 你认为在你的数据库中应该建立什么样的索引？为什么？
  2. 你的数据库中每个表中的字段类型和长度是如何确定的？为什么？
  3. 你认为在我们大二上课时讲解ORM映射的Person例子时，每个用户的邮箱如果只有一个，是否还有必要像上课那样将邮箱专门存储在一张表中，然后通过外键关联？为什么？
  4. 你认为主键使用自增主键和UUID各自的优缺点是什么？
  5. 请你搜索参考文献，总结InnoDB和MyISAM两种存储引擎的主要差异，并详细说明你的E-BookStore项目应该选择哪种存储引擎。

  – 请提交包含上述问题答案的文档，文档中附上你更新的数据库的设计方案，包括库结构、表结构和表与表之间的关联

- 评分标准：
  – 上述每个问题 1 分，答案不唯一，只要你的说理合理即可视为正确。

Thank You!