Architecture of Enterprise Applications 11
MySQL Optimization I

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
  - Optimization Overview
  - Optimization and Indexes
  - Optimizing Database Structure
  - 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，包括数据库结构和索引

Optimization Overview

- Database performance depends on several factors at the database level,
  - such as tables, queries, and configuration settings.

- Advanced users look for opportunities to improve the MySQL software itself,
  - or develop their own storage engines and hardware appliances to expand the MySQL ecosystem.

Optimization Overview

- Optimizing at the Database Level
  - The most important factor in making a database application fast is its basic design:
    - Are the tables structured properly? In particular, do the columns have the right data types, and does each table have the appropriate columns for the type of work?
    - Are the right indexes in place to make queries efficient?
    - Are you using the appropriate storage engine for each table, and taking advantage of the strengths and features of each storage engine you use?.
    - Does each table use an appropriate row format?
    - Does the application use an appropriate locking strategy?
    - Are all memory areas used for caching sized correctly?

Optimization Overview

- Optimizing at the Hardware Level
  - Any database application eventually hits hardware limits as the database becomes more and more busy.
  - A DBA must evaluate
    - whether it is possible to tune the application
    - or reconfigure the server to avoid these bottlenecks,
    - or whether more hardware resources are required.
  - System bottlenecks typically arise from these sources:
    - Disk seeks.
    - Disk reading and writing.
    - CPU cycles.
    - Memory bandwidth.

Optimization Overview

- Balancing Portability and Performance
  - To use performance-oriented SQL extensions in a portable MySQL program, you can wrap MySQL-specific keywords in a statement within /*! */ comment delimiters.
  - Other SQL servers ignore the commented keywords.

Optimization and Indexes

- The best way to improve the performance of SELECT operations is
  - to create indexes on one or more of the columns that are tested in the query.
  - The index entries act like pointers to the table rows, allowing the query to quickly determine which rows match a condition in the WHERE clause, and retrieve the other column values for those rows.
  - All MySQL data types can be indexed.
- Although it can be tempting to create an indexes for every possible column used in a query,
  - unnecessary indexes waste space and waste time for MySQL to determine which indexes to use.
  - Indexes also add to the cost of inserts, updates, and deletes because each index must be updated.
  - You must find the right balance to achieve fast queries using the optimal set of indexes.

How MySQL Uses Indexes

- Indexes are used to find rows with specific column values quickly.
  - Without an index, MySQL must begin with the first row and then read through the entire table to find the relevant rows.
  - The larger the table, the more this costs.
  - If the table has an index for the columns in question, MySQL can quickly determine the position to seek to in the middle of the data file without having to look at all the data.
  - This is much faster than reading every row sequentially.

- Most MySQL indexes (PRIMARY KEY, UNIQUE, INDEX, and FULLTEXT) are stored in B-trees. Exceptions:
  - Indexes on spatial data types use R-trees;
  - MEMORY tables also support hash indexes;
  - InnoDB uses inverted lists for FULLTEXT indexes.

How MySQL Uses Indexes

- MySQL uses indexes for these operations:
  - To find the rows matching a WHERE clause quickly.
  - To eliminate rows from consideration.
  - If the table has a multiple-column index, any leftmost prefix of the index can be used by the optimizer to look up rows.
  - To retrieve rows from other tables when performing joins.
  - To find the MIN() or MAX() value for a specific indexed column *key_col*.
  - To sort or group a table if the sorting or grouping is done on a leftmost prefix of a usable index (for example, ORDER BY *key_part1*, *key_part2*).
  - In some cases, a query can be optimized to retrieve values without consulting the data rows.
- Indexes are less important for queries on small tables,
  - or big tables where report queries process most or all of the rows.
  - When a query needs to access most of the rows, reading sequentially is faster than working through an index. Sequential reads minimize disk seeks, even if not all the rows are needed for the query.

Primary Key Optimization

- The primary key for a table represents
  - the column or set of columns that you use in your most vital queries.
  - It has an associated index, for fast query performance.
  - Query performance benefits from the NOT NULL optimization, because it cannot include any NULL values.
  - With the InnoDB storage engine, the table data is physically organized to do ultra-fast lookups and sorts based on the primary key column or columns.

- If your table is big and important,
  - but does not have an obvious column or set of columns to use as a primary key,
  - you might create a separate column with auto-increment values to use as the primary key.
  - These unique IDs can serve as pointers to corresponding rows in other tables when you join tables using foreign keys.

SPATIAL Index Optimization

- MySQL permits creation of SPATIAL indexes on NOT NULL geometry-valued columns.
  - For comparisons to work properly, each column in a SPATIAL index must be SRID-restricted.
  - That is, the column definition must include an explicit SRID attribute, and all column values must have the same SRID.

- The optimizer considers SPATIAL indexes only for SRID-restricted columns:
  - Indexes on columns restricted to a Cartesian SRID enable Cartesian bounding box computations.
  - Indexes on columns restricted to a geographic SRID enable geographic bounding box computations.

Foreign Key Optimization

- If a table has many columns, and you query many different combinations of columns,
  - it might be efficient to split the less-frequently used data into separate tables with a few columns each, and relate them back to the main table by duplicating the numeric ID column from the main table.
  - That way, each small table can have a primary key for fast lookups of its data, and you can query just the set of columns that you need using a join operation.
  - Depending on how the data is distributed, the queries might perform less I/O and take up less cache memory because the relevant columns are packed together on disk.
  - (To maximize performance, queries try to read as few data blocks as possible from disk; tables with only a few columns can fit more rows in each data block.)

Column Indexes

- **Index Prefixes**
  - With *col_name*(*N*) syntax in an index specification for a string column, you can create an index that uses only the first *N* characters of the column.
    - Indexing only a prefix of column values in this way can make the index file much smaller. When you index a BLOB or TEXT column, you *must* specify a prefix length for the index. For example:

      CREATE TABLE test (blob_col BLOB, INDEX(blob_col(10)));
  - Prefixes can be up to 767 bytes long for InnoDB tables that use the REDUNDANT or COMPACT row format.
  - The prefix length limit is 3072 bytes for InnoDB tables that use the DYNAMIC or COMPRESSED row format.
  - For MyISAM tables, the prefix length limit is 1000 bytes.
  - If a search term exceeds the index prefix length, the index is used to exclude non-matching rows, and the remaining rows are examined for possible matches.

- **FULLTEXT Indexes**
  - FULLTEXT indexes are used for full-text searches.
    - Only the InnoDB and MyISAM storage engines support FULLTEXT indexes and only for CHAR, VARCHAR, and TEXTcolumns.
    - Indexing always takes place over the entire column and column prefix indexing is not supported.
  - Optimizations are applied to certain kinds of FULLTEXT queries against single InnoDB tables. Queries with these characteristics are particularly efficient:
    - FULLTEXT queries that only return the document ID, or the document ID and the search rank.
    - FULLTEXT queries that sort the matching rows in descending order of score and apply a LIMIT clause to take the top N matching rows. For this optimization to apply, there must be no WHERE clauses and only a single ORDER BY clause in descending order.
    - FULLTEXT queries that retrieve only the COUNT(*) value of rows matching a search term, with no additional WHERE clauses. Code the WHERE clause as WHERE MATCH(*text*) AGAINST ('*other_text*'), without any > 0 comparison operator.

Column Indexes

- **Spatial Indexes**
  - You can create indexes on spatial data types.
  - MyISAM and InnoDB support R-tree indexes on spatial types.
  - Other storage engines use B-trees for indexing spatial types (except for ARCHIVE, which does not support spatial type indexing).

- **Indexes in the MEMORY Storage Engine**
  - The MEMORY storage engine uses HASH indexes by default, but also supports BTREE indexes.

Multiple-Column Indexes

- MySQL can create composite indexes (that is, indexes on multiple columns).
  - An index may consist of up to 16 columns.
  - For certain data types, you can index a prefix of the column.

- MySQL can use multiple-column indexes for
  - queries that test all the columns in the index,
  - or queries that test just the first column, the first two columns, the first three columns, and so on.
  - If you specify the columns in the right order in the index definition, a single composite index can speed up several kinds of queries on the same table.

Multiple-Column Indexes

- A multiple-column index can be considered a sorted array, the rows of which contain values that are created by concatenating the values of the indexed columns.

- Suppose that a table has the following specification:

```
CREATE TABLE test (
  id INT NOT NULL,
  last_name CHAR(30) NOT NULL,
  first_name CHAR(30) NOT NULL,
  PRIMARY KEY (id),
  INDEX name (last_name,first_name) );
```

- Therefore, the name index is used for lookups in the following queries:

```
SELECT * FROM test
    WHERE last_name='Jones';
SELECT * FROM test
    WHERE last_name='Jones' AND first_name='John';
SELECT * FROM test
    WHERE last_name='Jones' AND (first_name='John' OR first_name='Jon');
SELECT * FROM test
    WHERE last_name='Jones' AND first_name >='M' AND first_name < 'N';
```

Multiple-Column Indexes

- A multiple-column index can be considered a sorted array, the rows of which contain values that are created by concatenating the values of the indexed columns.

- Suppose that a table has the following specification:

```
CREATE TABLE test (
  id INT NOT NULL,
  last_name CHAR(30) NOT NULL,
  first_name CHAR(30) NOT NULL,
  PRIMARY KEY (id),
  INDEX name (last_name,first_name) );
```

- However, the name index is not used for lookups in the following queries:

```
SELECT * FROM test
   WHERE first_name='John';
SELECT * FROM test
   WHERE last_name='Jones' OR first_name='John';
```

Multiple-Column Indexes

- MySQL cannot use the index to perform lookups if the columns do not form a leftmost prefix of the index.

- Suppose that you have the SELECT statements shown here:

```
SELECT * FROM tbl_name WHERE col1=val1;
SELECT * FROM tbl_name WHERE col1=val1 AND col2=val2;

SELECT * FROM tbl_name WHERE col2=val2;
SELECT * FROM tbl_name WHERE col2=val2 AND col3=val3;
```

- If an index exists on (col1, col2, col3),
  - only the first two queries use the index.
  - The third and fourth queries do involve indexed columns, but do not use an index to perform lookups because (col2) and (col2, col3) are not leftmost prefixes of (col1, col2, col3).

Comparison of B-Tree and Hash Indexes

- **B-Tree Index Characteristics**
  - A B-tree index can be used for column comparisons in expressions that use the =, >, >=, <, <=, or BETWEEN operators.
  - The index also can be used for LIKE comparisons if the argument to LIKE is a constant string that does not start with a wildcard character.

- For example, the following SELECT statements use indexes:

```
SELECT * FROM tbl_name WHERE key_col LIKE 'Patrick%';
SELECT * FROM tbl_name WHERE key_col LIKE 'Pat%_ck%';
```

- The following SELECT statements do not use indexes:

```
SELECT * FROM tbl_name WHERE key_col LIKE '%Patrick%';
SELECT * FROM tbl_name WHERE key_col LIKE other_col;
```

Comparison of B-Tree and Hash Indexes

- **B-Tree Index Characteristics**
  - The following WHERE clauses use indexes:

```
... WHERE index_part1=1 AND index_part2=2 AND other_column=3
/* index = 1 OR index = 2 */
... WHERE index=1 OR A=10 AND index=2
/* optimized like "index_part1='hello'" */
... WHERE index_part1='hello' AND index_part3=5
/* Can use index on index1 but not on index2 or index3 */
... WHERE index1=1 AND index2=2 OR index1=3 AND index3=3;
```

  - These WHERE clauses do not use indexes:

```
/* index_part1 is not used */
... WHERE index_part2=1 AND index_part3=2
/* Index is not used in both parts of the WHERE clause */
... WHERE index=1 OR A=10
/* No index spans all rows */
... WHERE index_part1=1 OR index_part2=10
```

Comparison of B-Tree and Hash Indexes

- **Hash Index Characteristics**
  - They are used only for equality comparisons that use the = or <=> operators (but are *very* fast). They are not used for comparison operators such as < that find a range of values.
  - Systems that rely on this type of single-value lookup are known as "key-value stores"; to use MySQL for such applications, use hash indexes wherever possible.
  - The optimizer cannot use a hash index to speed up ORDER BY operations. (This type of index cannot be used to search for the next entry in order.)
  - MySQL cannot determine approximately how many rows there are between two values (this is used by the range optimizer to decide which index to use).
  - Only whole keys can be used to search for a row.

Descending Indexes

- MySQL supports descending indexes:
  - DESC in an index definition is no longer ignored but causes storage of key values in descending order.
  - Previously, indexes could be scanned in reverse order but at a performance penalty.
  - A descending index can be scanned in forward order, which is more efficient.
  - Descending indexes also make it possible for the optimizer to use multiple-column indexes when the most efficient scan order mixes ascending order for some columns and descending order for others.

Descending Indexes

- Consider the following table definition

```
CREATE TABLE t (
    c1 INT, c2 INT,
    INDEX idx1 (c1 ASC, c2 ASC),
    INDEX idx2 (c1 ASC, c2 DESC),
    INDEX idx3 (c1 DESC, c2 ASC),
    INDEX idx4 (c1 DESC, c2 DESC)
);

ORDER BY c1 ASC, c2 ASC -- optimizer can use idx1
ORDER BY c1 DESC, c2 DESC -- optimizer can use idx4
ORDER BY c1 ASC, c2 DESC -- optimizer can use idx2
ORDER BY c1 DESC, c2 ASC -- optimizer can use idx3
```

Optimizing Database Structure

- In your role as a database designer, look for the most efficient way to organize your schemas, tables, and columns.

- As when tuning application code,
  - you minimize I/O,
  - keep related items together,
  - and plan ahead so that performance stays high as the data volume increases.
- Starting with an efficient database design makes
  - it easier for team members to write high-performing application code,
  - and makes the database likely to endure as applications evolve and are rewritten.

Optimizing Data Size

- Design your tables to minimize their space on the disk.
  - This can result in huge improvements by reducing the amount of data written to and read from disk.
  - Smaller tables normally require less main memory while their contents are being actively processed during query execution.
  - Any space reduction for table data also results in smaller indexes that can be processed faster.

- You can get better performance for a table and minimize storage space by using the techniques listed here:
  - Table Columns
  - Row Format
  - Indexes
  - Joins
  - Normalization

Optimizing Data Size

- **Table Columns**
  - Use the most efficient (smallest) data types possible. MySQL has many specialized types that save disk space and memory.
    - For example, use the smaller integer types if possible to get smaller tables. MEDIUMINT is often a better choice than INT because a MEDIUMINT column uses 25% less space.
  - Declare columns to be NOT NULL if possible.
    - It makes SQL operations faster, by enabling better use of indexes and eliminating overhead for testing whether each value is NULL.
    - You also save some storage space, one bit per column.
    - If you really need NULL values in your tables, use them. Just avoid the default setting that allows NULL values in every column.

Optimizing Data Size

- **Row Format**
  - InnoDB tables are created using the DYNAMIC row format by default. To use a row format other than DYNAMIC, configure innodb_default_row_format, or specify the ROW_FORMAT option explicitly in a CREATE TABLE or ALTER TABLE statement.

    The compact family of row formats, which includes COMPACT, DYNAMIC, and COMPRESSED, decreases row storage space at the cost of increasing CPU use for some operations.

    The compact family of row formats also optimizes CHAR column storage when using a variable-length character set such as utf8mb3 or utf8mb4.
  - With ROW_FORMAT=REDUNDANT, CHAR(*N*) occupies *N* × the maximum byte length of the character set. Many languages can be written primarily using single-byte utf8 characters, so a fixed storage length often wastes space.
  - With the compact family of rows formats, InnoDB allocates a variable amount of storage in the range of *N* to *N* × the maximum byte length of the character set for these columns by stripping trailing spaces. The minimum storage length is *N* bytes to facilitate in-place updates in typical cases.

Optimizing Data Size

- **Row Format**
  - To minimize space even further by storing table data in compressed form,
    - specify ROW_FORMAT=COMPRESSED when creating InnoDB tables,
    - or run the**myisampack** command on an existing MyISAM table.
    - (InnoDB compressed tables are readable and writable, while MyISAM compressed tables are read-only.)
  - For MyISAM tables,
    - if you do not have any variable-length columns (VARCHAR, TEXT, or BLOB columns), a fixed-size row format is used.
    - This is faster but may waste some space.
    - You can hint that you want to have fixed length rows even if you have VARCHAR columns with the CREATE TABLE option ROW_FORMAT=FIXED.

Optimizing Data Size

- **Indexes**
  - The primary index of a table should be as short as possible.
    - This makes identification of each row easy and efficient.
    - For InnoDB tables, the primary key columns are duplicated in each secondary index entry, so a short primary key saves considerable space if you have many secondary indexes.
  - Create only the indexes that you need to improve query performance.
    - Indexes are good for retrieval, but slow down insert and update operations.
    - If you access a table mostly by searching on a combination of columns, create a single composite index on them rather than a separate index for each column. The first part of the index should be the column most used.
    - If you *always* use many columns when selecting from the table, the first column in the index should be the one with the most duplicates, to obtain better compression of the index.

Optimizing Data Size

- **Indexes**
  - If it is very likely that a long string column has a unique prefix on the first number of characters,
    - it is better to index only this prefix, using MySQL's support for creating an index on the leftmost part of the column.
    - Shorter indexes are faster, not only because they require less disk space, but because they also give you more hits in the index cache, and thus fewer disk seeks.

Optimizing Data Size

- **Joins**
  - In some circumstances, it can be beneficial to split into two a table that is scanned very often.
    - This is especially true if it is a dynamic-format table
    - and it is possible to use a smaller static format table that can be used to find the relevant rows when scanning the table.
  - Declare columns with identical information in different tables with identical data types,
    - to speed up joins based on the corresponding columns.
  - Keep column names simple, so that you can use the same name across different tables and simplify join queries.
    - For example, in a table named customer, use a column name of name instead of customer_name.
    - To make your names portable to other SQL servers, consider keeping them shorter than 18 characters.

Optimizing Data Size

- **Normalization**
  - Normally, try to keep all data nonredundant (observing what is referred to in database theory as third normal form).
    - Instead of repeating lengthy values such as names and addresses,
    - assign them unique IDs, repeat these IDs as needed across multiple smaller tables, and join the tables in queries by referencing the IDs in the join clause.
  - If speed is more important than disk space and the maintenance costs of keeping multiple copies of data,
    - for example in a business intelligence scenario where you analyze all the data from large tables,
    - you can relax the normalization rules, duplicating information or creating summary tables to gain more speed.

- **Optimizing for Numeric Data**
  - For unique IDs or other values that can be represented as either strings or numbers,
    - prefer numeric columns to string columns.
    - Since large numeric values can be stored in fewer bytes than the corresponding strings, it is faster and takes less memory to transfer and compare them.
  - If you are using numeric data,
    - it is faster in many cases to access information from a database (using a live connection) than to access a text file.
    - Information in the database is likely to be stored in a more compact format than in the text file, so accessing it involves fewer disk accesses.
    - You also save code in your application because you can avoid parsing the text file to find line and column boundaries.

Optimizing MySQL Data Types

- **Optimizing for Character and String Types**
  - Use binary collation order for fast comparison and sort operations,
    - when you do not need language-specific collation features. You can use the BINARY operator to use binary collation within a particular query.
  - When comparing values from different columns,
    - declare those columns with the same character set and collation wherever possible, to avoid string conversions while running the query.
  - For column values less than 8KB in size, use binary VARCHAR instead of BLOB.
    - The GROUP BY and ORDER BY clauses can generate temporary tables, and these temporary tables can use the MEMORY storage engine if the original table does not contain any BLOB columns.

- **Optimizing for Character and String Types**
  - If a table contains string columns such as name and address, but many queries do not retrieve those columns,
    - consider splitting the string columns into a separate table and using join queries with a foreign key when necessary.
    - When MySQL retrieves any value from a row, it reads a data block containing all the columns of that row (and possibly other adjacent rows). Keeping each row small, with only the most frequently used columns, allows more rows to fit in each data block.
    - Such compact tables reduce disk I/O and memory usage for common queries.
  - When you use a randomly generated value as a primary key in an InnoDB table,
    - prefix it with an ascending value such as the current date and time if possible. When consecutive primary values are physically stored near each other, InnoDB can insert and retrieve them faster.

- **Optimizing for BLOB Types**
  - When storing a large blob containing textual data, consider compressing it first.
  - For a table with several columns, to reduce memory requirements for queries that do not use the BLOB column,
    - consider splitting the BLOB column into a separate table and referencing it with a join query when needed.
  - You could put the BLOB-specific table on a different storage device or even a separate database instance.
  - Rather than testing for equality against a very long text string, you can store a hash of the column value in a separate column, index that column, and test the hashed value in queries.
    - Since hash functions can produce duplicate results for different inputs, you still include a clause AND *blob_column = long_string_value* in the query to guard against false matches; the performance benefit comes from the smaller, easily scanned index for the hashed values.

Optimizing for Many Tables

- How MySQL Opens and Closes Tables
- Disadvantages of Creating Many Tables in the Same Database

- Some techniques for keeping individual queries fast involve splitting data across many tables.
  - When the number of tables runs into the thousands or even millions, the overhead of dealing with all these tables becomes a new performance consideration.

Optimizing for Many Tables

- **How MySQL Opens and Closes Tables**
  - When you execute a **mysqladmin status** command, you should see something like this:

    Uptime: 426 Running threads: 1 Questions: 11082

    Reloads: 1 Open tables: 12
  - The Open tables value of 12 can be somewhat puzzling if you have fewer than 12 tables.
  - MySQL is multithreaded, so there may be many clients issuing queries for a given table simultaneously.
    - To minimize the problem with multiple client sessions having different states on the same table, the table is opened independently by each concurrent session.
    - This uses additional memory but normally increases performance.
    - With MyISAM tables, one extra file descriptor is required for the data file for each client that has the table open.

Optimizing for Many Tables

- **How MySQL Opens and Closes Tables**
  - The table_open_cache and max_connections system variables affect the maximum number of files the server keeps open.
    - If you increase one or both of these values, you may run up against a limit imposed by your operating system on the per-process number of open file descriptors.
  - table_open_cache is related to max_connections.
    - For example, for 200 concurrent running connections, specify a table cache size of at least 200 * *N*, where *N* is the maximum number of tables per join in any of the queries which you execute. You must also reserve some extra file descriptors for temporary tables and files.
  - Make sure that your operating system can handle the number of open file descriptors implied by the table_open_cache setting.
    - If table_open_cache is set too high, MySQL may run out of file descriptors and exhibit symptoms such as refusing connections or failing to perform queries.

Optimizing for Many Tables

- **How MySQL Opens and Closes Tables**
  - MySQL closes an unused table and removes it from the table cache under the following circumstances:
    - When the cache is full and a thread tries to open a table that is not in the cache.
    - When the cache contains more than table_open_cache entries and a table in the cache is no longer being used by any threads.
    - When a table-flushing operation occurs. This happens when someone issues a FLUSH TABLES statement or executes a **mysqladmin flush-tables** or **mysqladmin refresh** command.

Optimizing for Many Tables

- **How MySQL Opens and Closes Tables**
  - When the table cache fills up, the server uses the following procedure to locate a cache entry to use:
    - Tables not currently in use are released, beginning with the table least recently used.
    - If a new table must be opened, but the cache is full and no tables can be released, the cache is temporarily extended as necessary.
    - When the cache is in a temporarily extended state and a table goes from a used to unused state, the table is closed and released from the cache.

Optimizing for Many Tables

- **How MySQL Opens and Closes Tables**
  - To determine whether your table cache is too small, check the Opened_tables status variable, which indicates the number of table-opening operations since the server started:

```
mysql> SHOW GLOBAL STATUS LIKE 'Opened_tables';
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| Opened_tables | 2741  |
+---------------+-------+
```

    - If the value is very large or increases rapidly, even when you have not issued many FLUSH TABLES statements, increase the table_open_cache value at server startup.

Optimizing for Many Tables

- **Disadvantages of Creating Many Tables in the Same Database**

- If you have many MyISAM tables in the same database directory,
  - open, close, and create operations are slow.
- If you execute SELECT statements on many different tables,
  - there is a little overhead when the table cache is full,
  - because for every table that has to be opened, another must be closed.
  - You can reduce this overhead by increasing the number of entries permitted in the table cache.

Internal Temporary Table Use in MySQL

- In some cases, the server creates internal temporary tables while processing statements. Users have no direct control over when this occurs. The server creates temporary tables under conditions such as these:
  - Evaluation of UNION statements, with some exceptions.
  - Evaluation of some views, such those that use the TEMPTABLE algorithm, UNION, or aggregation.
  - Evaluation of derived tables.
  - Evaluation of common table expressions.
  - Tables created for subquery or semijoin materialization.
  - Evaluation of statements that contain an ORDER BY clause and a different GROUP BY clause, or for which the ORDER BY or GROUP BY contains columns from tables other than the first table in the join queue.
  - Evaluation of DISTINCT combined with ORDER BY may require a temporary table.
  - For queries that use the SQL_SMALL_RESULT modifier, MySQL uses an in-memory temporary table, unless the query also contains elements (described later) that require on-disk storage.
  - To evaluate INSERT ... SELECT statements that select from and insert into the same table, MySQL creates an internal temporary table to hold the rows from the SELECT, then inserts those rows into the target table..
  - Evaluation of multiple-table UPDATE statements.
  - Evaluation of GROUP_CONCAT() or COUNT(DISTINCT) expressions.
  - Evaluation of window.

Limits on Number of Databases and Tables

- MySQL has no limit on the number of databases.
  - The underlying file system may have a limit on the number of directories.
- MySQL has no limit on the number of tables.
  - The underlying file system may have a limit on the number of files that represent tables.
  - Individual storage engines may impose engine-specific constraints. InnoDB permits up to 4 billion tables.

Limits on Table Size

- The effective maximum table size for MySQL databases is
  - usually determined by operating system constraints on file sizes, not by MySQL internal limits.
  - For up-to-date information operating system file size limits, refer to the documentation specific to your operating system.
  - Windows users, please note that FAT and VFAT (FAT32) are *not* considered suitable for production use with MySQL. Use NTFS instead.

Limits on Table Size

- If you encounter a full-table error, there are several reasons why it might have occurred:
  - The disk might be full.
  - You are using InnoDB tables and have run out of room in an InnoDB tablespace file.
    - The maximum tablespace size is also the maximum size for a table.
    - Generally, partitioning of tables into multiple tablespace files is recommended for tables larger than 1TB in size.
  - You have hit an operating system file size limit.
    - For example, you are using MyISAM tables on an operating system that supports files only up to 2GB in size and you have hit this limit for the data file or index file.
  - You are using a MyISAM table and the space required for the table exceeds what is permitted by the internal pointer size.
    - MyISAM permits data and index files to grow up to 256TB by default, but this limit can be changed up to the maximum permissible size of 65,536TB ($256^7 - 1$ bytes).

Limits on Table Column Count and Row Size

- **Column Count Limits**
  - MySQL has hard limit of 4096 columns per table, but the effective maximum may be less for a given table. The exact column limit depends on several factors:
    - The maximum row size for a table constrains the number (and possibly size) of columns because the total length of all columns cannot exceed this size.
    - The storage requirements of individual columns constrain the number of columns that fit within a given maximum row size.
    - Storage engines may impose additional restrictions that limit table column count. For example, InnoDB has a limit of 1017 columns per table.
    - Functional key parts are implemented as hidden virtual generated stored columns, so each functional key part in a table index counts against the table total column limit.

Limits on Table Column Count and Row Size

- **Row Size Limits**
  - The maximum row size for a given table is determined by several factors:
    - The internal representation of a MySQL table has a maximum row size limit of 65,535 bytes, even if the storage engine is capable of supporting larger rows. BLOB and TEXT columns only contribute 9 to 12 bytes toward the row size limit because their contents are stored separately from the rest of the row.
    - The maximum row size for an InnoDB table, which applies to data stored locally within a database page, is slightly less than half a page for 4KB, 8KB, 16KB, and 32KB innodb_page_size settings.
    - Different storage formats use different amounts of page header and trailer data, which affects the amount of storage available for rows.

Limits on Table Column Count and Row Size

- **Row Size Limits Examples**

```
mysql> CREATE TABLE t (a VARCHAR(10000), b VARCHAR(10000),
         c VARCHAR(10000), d VARCHAR(10000), e VARCHAR(10000),
        f VARCHAR(10000), g VARCHAR(6000)) ENGINE=InnoDB CHARACTER SET latin1;
ERROR 1118 (42000): Row size too large. The maximum row size for the used
table type, not counting BLOBs, is 65535. This includes storage overhead,
check the manual. You have to change some columns to TEXT or BLOBs

mysql> CREATE TABLE t (a VARCHAR(10000), b VARCHAR(10000),
         c VARCHAR(10000), d VARCHAR(10000), e VARCHAR(10000),
         f VARCHAR(10000), g VARCHAR(6000)) ENGINE=MyISAM CHARACTER SET latin1;
ERROR 1118 (42000): Row size too large. The maximum row size for the used
table type, not counting BLOBs, is 65535. This includes storage overhead,
check the manual. You have to change some columns to TEXT or BLOBs
```

Limits on Table Column Count and Row Size

- **Row Size Limits Examples**

```
mysql> CREATE TABLE t (a VARCHAR(10000), b VARCHAR(10000),
        c VARCHAR(10000), d VARCHAR(10000), e VARCHAR(10000),
        f VARCHAR(10000), g TEXT(6000)) ENGINE=MyISAM CHARACTER SET latin1;
Query OK, 0 rows affected (0.02 sec)

mysql> CREATE TABLE t (a VARCHAR(10000), b VARCHAR(10000),
        c VARCHAR(10000), d VARCHAR(10000), e VARCHAR(10000),
        f VARCHAR(10000), g TEXT(6000)) ENGINE=InnoDB CHARACTER SET latin1;
Query OK, 0 rows affected (0.02 sec)
```

Limits on Table Column Count and Row Size

- **Row Size Limits Examples**

```
mysql> CREATE TABLE t1
        (c1 VARCHAR(32765) NOT NULL, c2 VARCHAR(32766) NOT NULL)
        ENGINE = InnoDB CHARACTER SET latin1;
Query OK, 0 rows affected (0.02 sec)

mysql> CREATE TABLE t2
        (c1 VARCHAR(65535) NOT NULL)
        ENGINE = InnoDB CHARACTER SET latin1;
ERROR 1118 (42000): Row size too large. The maximum row size for the used table type, not
counting BLOBs, is 65535. This includes storage overhead, check the manual. You have to change
some columns to TEXT or BLOBs

mysql> CREATE TABLE t2
        (c1 VARCHAR(65533) NOT NULL)
        ENGINE = InnoDB CHARACTER SET latin1;
Query OK, 0 rows affected (0.01 sec)
```

Limits on Table Column Count and Row Size

- **Row Size Limits Examples**

```
mysql> CREATE TABLE t3
        (c1 VARCHAR(32765) NULL, c2 VARCHAR(32766) NULL)
        ENGINE = MyISAM CHARACTER SET latin1;
```

ERROR 1118 (42000): Row size too large. The maximum row size for the used table type, not counting BLOBs, is 65535. This includes storage overhead, check the manual. You have to change some columns to TEXT or BLOBs

Limits on Table Column Count and Row Size

- **Row Size Limits Examples**

```
mysql> CREATE TABLE t4 (
        c1 CHAR(255),c2 CHAR(255),c3 CHAR(255),
        c4 CHAR(255),c5 CHAR(255),c6 CHAR(255),
        c7 CHAR(255),c8 CHAR(255),c9 CHAR(255),
        c10 CHAR(255),c11 CHAR(255),c12 CHAR(255),
        c13 CHAR(255),c14 CHAR(255),c15 CHAR(255),
        c16 CHAR(255),c17 CHAR(255),c18 CHAR(255),
        c19 CHAR(255),c20 CHAR(255),c21 CHAR(255),
        c22 CHAR(255),c23 CHAR(255),c24 CHAR(255),
        c25 CHAR(255),c26 CHAR(255),c27 CHAR(255),
        c28 CHAR(255),c29 CHAR(255),c30 CHAR(255),
        c31 CHAR(255),c32 CHAR(255),c33 CHAR(255) )
        ENGINE=InnoDB ROW_FORMAT=DYNAMIC DEFAULT CHARSET latin1;
ERROR 1118 (42000): Row size too large (> 8126). Changing some columns to TEXT or BLOB may help. In
current row format, BLOB prefix of 0 bytes is stored inline.
```

作业六(第一部分)

- 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？

  – 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!