High Performance MySQL Practical

Tuesday, April 01, 2014 16:45

1. Optimal Data Types:

a. Choose Data Type Suggestion:

i. Smaller is usually better
ii. Simple is good
iii. Avoid NULL if possible

b. MySQL Data Type usage and Suggestion

Data Type	Type Name
Whole Numbers Types	1. TINYINT (8 bit) 2. SMALLINT (16 bit) 3. MEDIUMINT (24 bit) 4. INT (32 bit) 5. BIGINT (64 bit) 6. <TINYINT/INT/…> UNSIGNED
Real Numbers Types	1. DECIMAL (support approximate calculations; Fast since it is performs by CPU itself; **4 bytes**) 2. FLOAT (support approximate calculations; Fast since it is performs by CPU itself; **8 bytes**) 3. DOUBLE (supports exact math in MySQL 5.0 and newer, and it was only a "storage type" in lower MySQL; slow since CPUs don't support the computations directly) **Notes**: Because of the additional space requirements and computational cost, you should use DECIMAL only when you need exact results for fractional numbers—for example, when storing financial data. But in some high-volume cases it actually makes sense to use a BIGINT instead, and store the data as some multiple of the smallest fraction of currency
String Types	1. CHAR a. Fixed-size b. **MySQL removes any trailing spaces**. 2. VARCHAR a. VARCHAR uses 1 or 2 extra bytes to record the value's length: 1 byte if the column's maximum length is 255 bytes or less, and 2 bytes if it's more. b. rows are variable-length, they can grow when you update them, which can cause extra work. If a row grows and no longer fits in its original location, the behavior is storage engine–dependent. c. **In version 5.0 and newer, MySQL preserves trailing spaces when you store and retrieve values**. In versions 4.1 and older, MySQL strips trailing spaces. 3. BINARY a. store binary b. MySQL pads BINARY values with \0 (the zero byte) instead of spaces and doesn't strip the pad value on retrieval. 4. VARBINARY a. store binary b. MySQL pads BINARY values with \0 (the zero byte) instead of spaces and doesn't strip the pad value on retrieval. 5. BLOB (TINYBLOB, SMALLBOB, MEDIUMBLOB, LONGBLOB) **a. BLOB is a synonym for SMALLBLOB** **b. BLOB types store binary data with no collation or character set** c. MySQL handles each BLOB and TEXT value as an object with its own identity d. InnoDB may use a separate "external" storage area for them when they're large e. Each value requires from one to four bytes of storage space in the row and enough space in external storage to actually hold the value. 6. TEXT (TINYTEXT, SMALLTEXT, MEDIUMTEXT, LONGTEXT) **a. TEXT is a synonym for SMALLTEXT** **b. TEXT types have a character set and collati**on c. MySQL handles each BLOB and TEXT value as an object with its own identity d. InnoDB may use a separate "external" storage area for them when they're large e. Each value requires from one to four bytes of storage space in the row and enough space in external storage to actually hold the value. **Note**: MySQL sorts BLOB and TEXT columns differently from other types: instead of sorting the full length of the string, it sorts only the first max_sort_length bytes of such columns. If you need to sort by only the first few characters, you can either decrease the max_sort_length server variable or use ORDER BY SUBSTRING(column, length). **They can't be indexed by MySQL.**

ENUM Types	1. ENUM
	a. MySQL stores them very compactly, packed into one or two bytes depending on the number of values in the list
	b. It stores each value internally as an integer representing its position in the field definition list, and it keeps the "lookup table" that defines the number-to-string correspondence in the table's .frm file.
	c. Usage example:
	`mysql> CREATE TABLE enum_test(`
	`-> e ENUM('fish', 'apple', 'dog') NOT NULL`
	`-> );`
	`mysql> INSERT INTO enum_test(e) VALUES('fish'), ('dog'), ('apple');`
	d. Sort Usage:
	`mysql> SELECT e FROM enum_test ORDER BY FIELD(e, 'apple', 'dog', 'fish');`
	e. **The biggest downside of ENUM is that the list of strings is fixed, and adding or removing strings requires the use of ALTER TABLE**.
	f. **it can be slower to join a CHAR or VARCHAR column to an ENUM column than to another CHAR or VARCHAR column.**

Date and Time Types	1. DATETIME a. from the year 1001 to the year 9999 b. with a precision of one second c. It stores the date and time packed into an integer in **YYYYMMDDHHMMSS** format **d. independent of time zone** e. uses eight bytes of storage space. 2. TIMESTAMP a. the TIMESTAMP type stores the number of seconds elapsed since midnight, January 1, 1970, Greenwich Mean Time (GMT) b. from the year 1970 to partway through the year 2038 c. uses only four bytes of storage **d. depends on the time zone** **e. TIMESTAMP columns are NOT NULL by default. MySQL will set the first TIMESTAMP column to the current time when you**

	insert a row without specifying a value for the column.4 MySQL also updates the first TIMESTAMP column's value by default when you update the row, unless you assign a value explicitly in the UPDATE statement. **Note:** if you store or access data from multiple time zones, the behavior of TIMESTAMP and DATETIME will be very different. The former preserves values relative to the time zone in use, while the latter preserves the textual representation of the date and time. **Note**: MySQL 5.6.4 and newer support sub-second resolution
Bit-Packed Data Types	1. BIT a. Before MySQL 5.0, BIT is just a synonym for TINYINT. But in MySQL 5.0 and newer, it's a completely different data type with special characteristics. b. maximum length of a BIT column is 64 bits 2. SET

c. Choosing Identifiers

i. **Make sure you use the same type of identifier in all related tables.** Mixing different data types can cause performance problems, and even if it doesn't, implicit type conversions during comparisons can create hard-to-find errors.

ii. Choose the smallest size that can hold your required range of values, and leave room for future growth if necessary.

iii. **Few Tips when choosing identifier type**:

1) Integers are usually the best choice for identifiers, because they're fast and they work with AUTO_INCREMENT.
2) The ENUM and SET types are generally a poor choice for identifiers.
3) Avoid string types for identifiers if possible, because they take up a lot of space and are generally slower than integer types.
4) be very careful with completely "random" strings, such as those produced by MD5(), SHA1(), or UUID(). Followings are the reasons:
   a) They slow INSERT queries because the inserted value has to go in a random location in index.
   b) They slow SELECT queries because logically adjacent rows will be widely dispersed on disk and in memory.
   c) Random values cause caches to perform poorly for all types of queries because they defeat locality of reference, which is how caching works.
5) **If you do store UUID values, you should remove the dashes or, even better, convert the UUID values to 16-byte numbers with UNHEX() and store them in a BINARY(16) column.** You can retrieve the values in hexadecimal format with the HEX() function.

2. Schema Design

a. Too many columns is not efficient

MySQL's storage engine API works by copying rows between the server and the storage engine in a row buffer format; the server then decodes the buffer into columns. But it can be costly to turn the row buffer into the row data structure with the decoded columns. The cost of this conversion depends on the number of columns.

b. Too many joins is not efficient

c. Beware of overusing ENUM

d. NULL not invented here. Suggest considering alternatives when possible

e. Pros & Cons of a Normalized Schema

- The drawbacks of a normalized schema usually have to do with retrieval.
- Normalized updates are usually faster than denormalized updates.
- When the data is well normalized, there's little or no duplicated data, so there's less data to change.
- Normalized tables are usually smaller, so they fit better in memory and perform better.
- The lack of redundant data means there's less need for DISTINCT or GROUP BY queries when retrieving lists of values. Consider the preceding example: it's impossible to get a distinct list of departments from the denormalized schema without DIS TINCT or GROUP BY, but if DEPARTMENT is a separate table, it's a trivial query.

f. Pros & Cons of a Denormalized Schema

- A denormalized schema works well because everything is in the same table, which avoids joins.

g. Using Cache & Summary Table

i. Cache Table: refer to tables that contain data that can be easily, if more slowly, retrieved from the schema (i.e., data that is logically redundant)

ii. Summary Table: tables that hold aggregated data from GROUP BY queries (i.e., data that is not logically redundant)

h. Suggestion for Counter Table

i. Common way: The problem is that this single row is effectively a global "mutex" for any transaction that updates the counter. It will serialize those transactions.

```
mysql> CREATE TABLE hit_counter (
    -> cnt int unsigned not null
    -> ) ENGINE=InnoDB;
mysql> UPDATE hit_counter SET cnt = cnt + 1;
```

ii. Better Way:

```
mysql> CREATE TABLE hit_counter (
    -> slot tinyint unsigned not null primary key,
    -> cnt int unsigned not null
    -> ) ENGINE=InnoDB;
-- query can just choose a random slot and update it
mysql> UPDATE hit_counter SET cnt = cnt + 1 WHERE slot = RAND() * 100;
-- To retrieve statistics, just use aggregate queries:
mysql> SELECT SUM(cnt) FROM hit_counter;
```

i. Speeding Up ALTER TABLE

a. MySQL performs most alterations by making an empty table with the desired new structure, inserting all the data from the old table into the new one, and deleting the old table. This can take a very long time, especially if you¡¯re short on memory and the table is large and has lots of indexes.

b. In general, most ALTER TABLE operations will cause interruption of service in MySQL. We'll show some techniques to work around this in a bit, but those are for special cases.

1) swapping servers around and performing the ALTER on servers that are not in production service
2) "shadow copy" approach. The technique for a shadow copy is to build a new table with the desired structure beside the existing one, and then perform a rename and drop to swap the two.

c. The default value for the column is actually stored in the table's .frm file. any MODIFY COLUMN will cause a table rebuild. However change a column's default with ALTER COLUMN don't, it just change the value in .frm file.

3. Indexing For High Performance

a. B-Tree Index

- **Types of queries that can use a B-Tree index.**
  - □ Match the full value: A match on the full key value specifies values for all columns in the index.
  - □ Match a leftmost prefix
  - □ Match a column prefix: You can match on the first part of a column's value.
  - □ Match a range of values
  - □ Match one part exactly and match a range on another part
  - □ Index-only queries: B-Tree indexes can normally support index-only queries, which are queries that access only the index, not the row storage.
- **Here are some limitations of B-Tree indexes:**
  - □ They are not useful if the lookup does not start from the leftmost side of the indexed columns.
  - □ You can't skip columns in the index.
  - □ The storage engine can't optimize accesses with any columns to the right of the first range condition.

b. Hash Index

- A hash index is built on a hash table and is useful only for exact lookups that use every column in the index.4 For each row, the storage engine computes a hash code of the

indexed columns. It stores the hash codes in the index and stores a pointer to each row in a hash table.

- **In MySQL, only the Memory storage engine supports explicit hash indexes.**
- If multiple values have the same hash code, the index will store their row pointers in the same hash table entry, using a linked list.
- hash indexes havesome limitations:
  - Because the index contains only hash codes and row pointers rather than the values themselves, MySQL can't use the values in the index to avoid reading the rows.
  - MySQL can't use hash indexes for sorting because they don't store rows in sorted order.
  - Hash indexes don't support partial key matching, because they compute the hash from the entire indexed value. That is, if you have an index on (A,B) and your query's WHERE clause refers only to A, the index won't help.
  - Hash indexes support only equality comparisons that use the =, IN(), and <=> operators (note that <> and <=> are not the same operator). They can't speed up range queries, such as WHERE price > 100.
  - Accessing data in a hash index is very quick, unless there are many collisions (multiple values with the same hash). When there are collisions, the storage engine must follow each row pointer in the linked list and compare their values to the lookup value to find the right row(s).
  - Some index maintenance operations can be slow if there are many hash collisions.
- **Building your own hash indexes.**

```
-- Common Way:
mysql> SELECT id FROM url WHERE url="http://www.mysql.com";
-- Better way
mysql> SELECT id FROM url WHERE url="http://www.mysql.com"
    -> AND url_crc=CRC32("http://www.mysql.com");
```

This works well because the MySQL query optimizer notices there's a small, highly selective index on the url_crc column and does an index lookup for entries with that value (1560514994, in this case). Even if several rows have the same url_crc value, it's very easy to find these rows with a fast integer comparison and then examine them to find the one that matches the full URL exactly.

c. **Indexing Strategies for High Performance**

a. **Don't isolating the column:** "Isolating" the column means it should not be part of an expression or be inside a function in the query.
For example, here's a query that can't use the index on actor_id: mysql> SELECT actor_id FROM sakila.actor WHERE actor_id + 1 = 5;

b. **Prefix Indexes and Index Selectivity:** You can often save space and get good performance by indexing the first few characters instead of the whole value. This makes your indexes use less space, but it also makes them less selective.

c. **Individual indexes on lots of columns won't help MySQL improve performance for most queries.** MySQL 5.0 and newer can cope a little with such poorly indexed tables by using a strategy known as **index merge, which permits a query to make limited use of multiple indexes from a single table to locate desired rows.** Earlier versions of MySQL could use only a single index, so when no single index was good enough to help, MySQL often chose a table scan.

```
mysql> SELECT film_id, actor_id FROM sakila.film_actor
    -> WHERE actor_id = 1 OR film_id = 1;

-- In older MySQL versions, that query would produce a table scan unless you wrote it as the UNION of two queries:
mysql> SELECT film_id, actor_id FROM sakila.film_actor WHERE actor_id = 1
    -> UNION ALL
    -> SELECT film_id, actor_id FROM sakila.film_actor WHERE film_id = 1
    -> AND actor_id <> 1;

-- In MySQL 5.0 and newer, however, the query can use both indexes, scanning them simultaneously and merging the results.
```

d. **Choosing a Good Column Order**

d. **Clustered Indexes**

Clustered indexes aren't a separate type of index. Rather, they're an approach to data storage. The term "clustered" refers to the fact that rows with adjacent key values are stored close to each other. **You can have only one clustered index per table, because you can't store the rows in two places at once.**

InnoDB clusters the data by the primary key. If you don't define a primary key, InnoDB will try to use a unique nonnullable index instead. If there's no such index, InnoDB will define a hidden primary key for you and then cluster on that. InnoDB clusters records together only within a page. Pages with adjacent key values might be distant from each other.

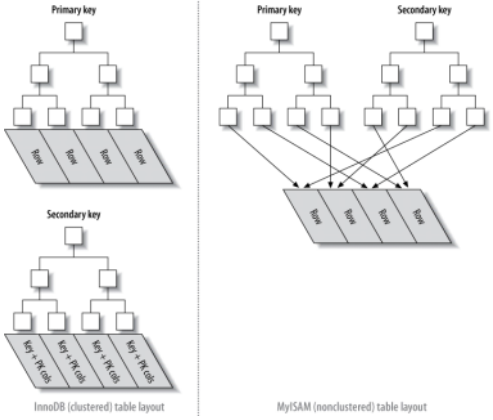
Clustering data has some very important advantages:

- You can keep related data close together.
- Data access is fast. A clustered index holds both the index and the data together in one B-Tree, so retrieving rows from a clustered index is normally faster than a comparable lookup in a nonclustered index.
- Queries that use covering indexes can use the primary key values contained at the leaf node.

However, clustered indexes also have disadvantages:

- Clustering gives the largest improvement for I/O-bound workloads. If the data fits in memory the order in which it's accessed doesn't really matter, so clustering doesn't give much benefit.
- Insert speeds depend heavily on insertion order. Inserting rows in primary key order is the fastest way to load data into an InnoDB table. It might be a good idea to reorganize the table with OPTIMIZE TABLE after loading a lot of data if you didn't load the rows in primary key order.
- Updating the clustered index columns is expensive, because it forces InnoDB to move each updated row to a new location.
- Tables built upon clustered indexes are subject to page splits when new rows are inserted, or when a row's primary key is updated such that the row must be moved. A page split happens when a row's key value dictates that the row must be placed into a page that is full of data. The storage engine must split the page into two to accommodate the row. Page splits can cause a table to use more space on disk.
- Clustered tables can be slower for full table scans, especially if rows are less densely packed or stored nonsequentially because of page splits.
- Secondary (nonclustered) indexes can be larger than you might expect, because their leaf nodes contain the primary key columns of the referenced rows.
- Secondary index accesses require two index lookups instead of one.

Comparison of InnoDB and MyISAM data layout:



e. **Covering Indexes**

An index that contains (or "covers") all the data needed to satisfy a query is called a covering index. **When you issue a query that is covered by an index (an index-covered query), you'll see "Using index" in the Extra column in EXPLAIN**

Example:

a. Table has column a and column b
b. Has query: SELECT a from <table> where b=<value>
c. Has index: index on (b, a)
d. Above index is covering index on above query, since it locate <value> on b in the index and such index contains a's value, then just return a's value from index

f. Using Index Scans for Sorts

MySQL has two ways to produce ordered results:

a. it can use a sort operation, or it can
b. scan an index in order:
   1) Ordering the results by the index works only when the index's order is exactly the same as the ORDER BY clause and all columns are sorted in the same direction
   2) If the query joins multiple tables, it works only when all columns in the ORDER BY clause refer to the first table.
   3) The ORDER BY clause also has the same limitation as lookup queries: it needs to form a leftmost prefix of the index.
   4) In all other cases, MySQL uses a sort.
   5) One case where the ORDER BY clause doesn't have to specify a leftmost prefix of the index is if there are constants for the leading columns.

g. Unused Indexes

you might have some indexes that the server simply doesn't use. These are simply dead weight, and you should consider dropping them.

h. Indexes and Locking

Indexes permit queries to lock fewer rows. If your queries never touch rows they don't need, they'll lock fewer rows, and that's better for performance for two reasons. First, even though InnoDB's row locks are very efficient and use very little memory, there's still some overhead involved in row locking. Secondly, locking more rows than needed increases lock contention and reduces concurrency.

Note:

a. InnoDB locks rows only when it accesses them, and an index can reduce the number of rows InnoDB accesses and therefore locks.
b. However, above reduction works only if InnoDB can filter out the undesired rows at the storage engine level.
c. If the index doesn't permit InnoDB to do that, the MySQL server will have to apply a WHERE clause after InnoDB retrieves the rows and returns them to the server level.17 At this point, it's too late to avoid locking the rows: InnoDB will already have locked them, and they will remain locked for some period of time. In MySQL 5.1 and newer, InnoDB can unlock rows after the server filters them out; in older versions of MySQL, InnoDB doesn't unlock the rows until the transaction commits.

4. Query Performance Optimization

a. Slow Query Basics: Optimize Data Access: We've found it useful to analyze a poorly performing query in two steps:

- Find out whether your application is retrieving more data than you need. That usually means it's accessing too many rows, but it might also be accessing too many columns.
  - Fetching more rows than needed
  - Fetching all columns from a multitable join
  - Fetching all columns
  - Fetching the same data repeatedly
- Find out whether the MySQL server is analyzing more rows than it needs.
  - Response time
  - Number of rows examined
  - Number of rows returned

b. Rows examined and access types

a. The access method(s) appear in the type column in EXPLAIN's output.
b. The access types range from a **full table scan (ALL)** to **index scans**, **range scans**, **unique index lookups**, and **constants**.
c. Each of above type is faster than the one before it, because it requires reading less data.
d. If you aren't getting a good access type, the best way to solve the problem is usually by adding an appropriate index.
e. In general, MySQL can apply a WHERE clause in three ways, from best to worst:
   1) Apply the conditions to the index lookup operation to eliminate nonmatching rows. This happens at the storage engine layer.
   2) Use a covering index ("Using index" in the Extra column) to avoid row accesses, and filter out nonmatching rows after retrieving each result from the index. This happens at the server layer, but it doesn't require reading rows from the table.
   3) Retrieve rows from the table, then filter nonmatching rows ("Using where" in the Extra column). This happens at the server layer and requires the server to read rows from the table before it can filter them.

c. Ways to Restructure Queries

i. Complex Queries vs. Many Queries: The traditional approach to database design emphasizes doing as much work as possible with as few queries as possible. This approach was historically better because of the cost of network communication and the overhead of the query parsing and optimization stages. However, this advice doesn't apply as much to MySQL, because it was designed to handle connecting and disconnecting very efficiently and to respond to small and simple queries very quickly. Modern networks are also significantly faster than they used to be, reducing network latency.

ii. Chopping Up a Query: doing this in one massive query could lock a lot of rows for a long time, fill up transaction logs, hog resources, and block small queries that shouldn't be interrupted.

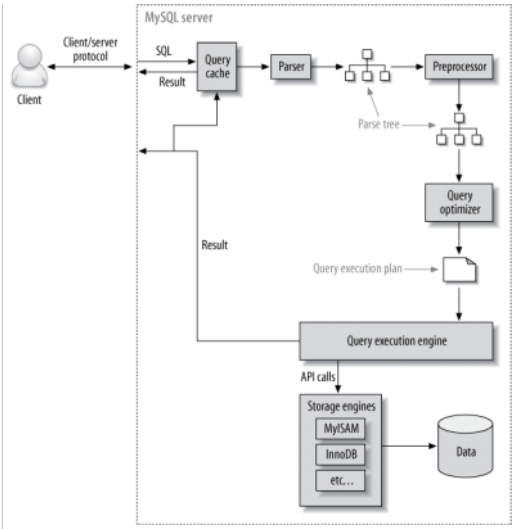
```
-- instead of running this monolithic query:
mysql> DELETE FROM messages WHERE created < DATE_SUB(NOW(),INTERVAL 3 MONTH);

-- you could do something like the following pseudocode:
rows_affected = 0
do {
        rows_affected = do_query(
        "DELETE FROM messages WHERE created < DATE_SUB(NOW(),INTERVAL 3 MONTH)
        LIMIT 10000")
} while rows_affected > 0
```

Deleting 10,000 rows at a time is typically a large enough task to make each query efficient, and a short enough task to minimize the impact on the server4 (transactional storage engines might benefit from smaller transactions). It might also be a good idea to add some sleep time between the DELETE statements to spread the load over time and reduce the amount of time locks are held.

iii. Join Decomposition: Many high-performance applications use join decomposition. You can decompose a join by running multiple single-table queries instead of a multitable join, and then performing the join in the application.

d. Query Execution Basics



what happens when you send MySQL a query:

1. The client sends the SQL statement to the server.
2. The server checks the query cache. If there's a hit, it returns the stored result from the cache; otherwise, it passes the SQL statement to the next step.
3. The server parses, preprocesses, and optimizes the SQL into a query execution plan.
4. The query execution engine executes the plan by making calls to the storage engine API.
5. The server sends the result to the client.

i. The MySQL Client/Server Protocol

1) The protocol is halfduplex, which means that at any given time the MySQL server can be either sending or receiving messages, but not both. It also means there is no way to cut a message short.
2) The client sends a query to the server as a single packet of data. This is why the max_allowed_packet configuration variable is important if you have large queries.5 Once the client sends the query, it doesn't have the ball anymore; it can only wait for results.
3) In contrast, the response from the server usually consists of many packets of data. When the server responds, the client has to receive the entire result set. It cannot simply fetch a few rows and then ask the server not to bother sending the rest, which is why appropriate LIMIT clauses are so important. **Until all the rows have been fetched, the MySQL server will not release the locks and other resources required by the query. The query will be in the "Sending data" state.**
4) Query States:
   a) Sleep: The thread is waiting for a new query from the client.
   b) Query: The thread is either executing the query or sending the result back to the client.
   c) Locked: The thread is waiting for a table lock to be granted at the server level. Locks that are implemented by the storage engine, such as InnoDB's row locks, do not cause the thread to enter the Locked state. This thread state is the classic symptom of MyISAM locking, but it can occur in other storage engines that don't have rowlevel locking, too.
   d) Analyzing and statistics: The thread is checking storage engine statistics and optimizing the query.
   e) Copying to tmp table [on disk]: The thread is processing the query and copying results to a temporary table, probably for a GROUP BY, for a filesort, or to satisfy a UNION. If the state ends with "on disk," MySQL is converting an in-memory table to an on-disk table.
   f) Sorting result: The thread is sorting a result set.
   g) Sending data: This can mean several things: the thread might be sending data between stages of the query, generating the result set, or returning the result set to the client.

ii. The Query Cache

1) operation is a case-sensitive hash lookup. If the query differs from a similar query in the cache by even a single byte, it won't match,7 and the query processing will go to the next stage.
2) If MySQL does find a match in the query cache, it must check privileges before returning the cached query.

iii. The Query Optimization Process

1) Parser:
   a) To begin, MySQL's parser breaks the query into tokens and builds a "parse tree" from them.
   b) The parser uses MySQL's SQL grammar to interpret and validate the query.
   c) it ensures that the tokens in the query are valid and in the proper order, and it checks for mistakes such as quoted strings that aren't terminated.

2) Pre-Processor
   a) The preprocessor then checks the resulting parse tree for additional semantics that the parser can't resolve.
   b) it checks that tables and columns exist, and it resolves names and aliases to ensure that column references aren't ambiguous.
   c) Next, the preprocessor checks privileges.

3) The Query Optimizer
   a) The optimizer's job is to find the best option.
   b) MySQL uses a cost-based optimizer, which means it tries to predict the cost of various execution plans and choose the least expensive.
   c) You can see how expensive the optimizer estimated a query to be by running the query, then inspecting the Last_query_cost session variable:

```
mysql> SELECT SQL_NO_CACHE COUNT(*) FROM sakila.film_actor;
+----------+
| count(*) |
+----------+
| 5462 |
+----------+
mysql> SHOW STATUS LIKE 'Last_query_cost';
+-----------------+------------+
| Variable_name | Value |
+-----------------+------------+
| Last_query_cost | 1040.599000 |
+-----------------+------------+
```

   i) This result means that the optimizer estimated it would need to do about 1,040 random data page reads to execute the query.
   ii) It bases the estimate on statistics:
      (a) the number of pages per table or index,
      (b) the cardinality (number of distinct values) of the indexes,
      (c) The length of the rows and keys
      (d) the key distribution.

      Note: The server layer, which contains the query optimizer, doesn't store statistics on data and indexes. That's a job for the storage engines, because each storage engine might keep different kinds of statistics (or keep them in a different way). Some engines, such as Archive, don't keep statistics at all!
   iii) The optimizer does not include the effects of any type of caching in its estimates—it assumes every read will result in a disk I/O operation.
   d) The optimizer might not always choose the best plan, for many reasons:
   i) The statistics could be wrong. The server relies on storage engines to provide statistics, and they can range from exactly correct to wildly inaccurate. For example, the InnoDB storage engine doesn't maintain accurate statistics about the number of rows in a table because of its MVCC architecture.
   ii) The cost metric is not exactly equivalent to the true cost of running the query, so even when the statistics are accurate, the query might be more or less expensive than MySQL's approximation. A plan that reads more pages might actually be cheaper in some cases, such as when the reads are sequential so the disk I/O is faster, or when the pages are already cached in memory. MySQL also doesn't understand which pages are in memory and which pages are on disk, so it doesn't really know how much I/O the query will cause.
   iii) MySQL's idea of "optimal" might not match yours. You probably want the fastest execution time, but MySQL doesn't really try to make queries fast; it tries to

minimize their cost, and as we've seen, determining cost is not an exact science.

iv) MySQL doesn't consider other queries that are running concurrently, which can affect how quickly the query runs.

v) MySQL doesn't always do cost-based optimization. Sometimes it just follows the rules, such as "if there's a full-text MATCH() clause, use a FULLTEXT index if one exists." It will do this even when it would be faster to use a different index and a non-FULLTEXT query with a WHERE clause.

vi) The optimizer doesn't take into account the cost of operations not under its control, such as executing stored functions or user-defined functions.

vii) As we'll see later, the optimizer can't always estimate every possible execution plan, so it might miss an optimal plan.

e) Here are some types of optimizations MySQL knows how to do:

i) Reordering joins

ii) **Converting OUT JOINs to INNER JOINS**: Some factors, such as the WHERE clause and table schema, can actually cause an OUTER JOIN to be equivalent to an INNER JOIN. MySQL can recognize this and rewrite the join, which makes it eligible for reordering.

iii) **Applying algebraic equivalence rules**: (5=5 AND a>5) will reduce to just a>5

iv) COUNT(), MIN(), and MAX() optimizations: Indexes and column nullability can often help MySQL optimize away these expressions. To find the minimum value of a column that's leftmost in a B-Tree index. Similarly, to find the maximum value in a B-Tree index, the server reads the last row. If the server uses this optimization, you'll see "Select tables optimized away" in the EXPLAIN plan. This literally means the optimizer has removed the table from the query plan and replaced it with a constant. Likewise, COUNT(*) queries without a WHERE clause can often be optimized away on some storage engines

v) **Evaluating and reducing constant expressions**: When MySQL detects that an expression can be reduced to a constant, it will do so during optimization. User-defined variable can be converted to a constant if it's not changed in the query. Arithmetic expressions are another example. Even something you might consider to be a query can be reduced to a constant during the optimization phase.

vi) Covering indexes

vii) Subquery optimization: MySQL can convert some types of subqueries into more efficient alternative forms, reducing them to index lookups instead of separate queries.

viii) Early termination: MySQL can stop processing a query (or a step in a query) as soon as it fulfills the query or step. The obvious case is a LIMIT clause, but there are several other kinds
of early termination.

ix) Equality propagation: MySQL recognizes when a query holds two columns as equal—for example, in a JOIN condition—and propagates WHERE clauses across equivalent columns.

```
mysql> SELECT film.film_id
    -> FROM sakila.film
    -> INNER JOIN sakila.film_actor USING(film_id)
    -> WHERE film.film_id > 500;

-- no need to do like this:
... WHERE film.film_id > 500 AND film_actor.film_id > 500
```

x) IN() list comparisons: It's not just a synonym for multiple OR clauses. In MySQL, IN sorts the values in the list and uses a fast binary search to see whether a value is in the list. This is O(log n) in the size of the list.

f) MySQL's join execution strategy: it treats every join as a nested-loop join.

i) This means MySQL runs a loop to find a row from a table,

ii) then runs a nested loop to find a matching row in the next table.

iii) It continues until it has found a matching row in each table in the join.

iv) It then builds and returns a row from the columns named in the SELECT list.

v) It tries to build the next row by looking for more matching rows in the last table. If it doesn't find any, it backtracks one table and looks for more rows there.

vi) It keeps backtracking until it finds another row in some table, at which point it looks for a matching row in the next table,

vii) and so on.

Note: **Hash-Join**: The classic hash join algorithm for an inner join of two relations proceeds as follows: First prepare a hash table of the smaller relation. The hash table entries consist of the join attribute and its row. Because the hash table is accessed by applying a hash function to the join attribute, it will be much quicker to find a given join attribute's rows by using this table than by scanning the original relation. Once the hash table is built, scan the larger relation and find the relevant rows from the smaller relation by looking in the hash table. The first phase is usually called the "build" phase, while the second is called the "probe" phase.
Note: a FULL OUTER JOIN can't be executed with nested loops and backtracking as soon as a table with no matching rows is found, because it might begin with a table that has no matching rows. This explains why **MySQL doesn't support FULL OUTER JOIN**.

g) The Execution Plan

i) MySQL doesn't generate byte-code to execute a query, as many other database products do. Instead, the query execution plan is actually a tree of instructions that the query execution engine follows to produce the query results.

ii) If you execute EXPLAIN EXTENDED on a query, followed by SHOW WARNINGS, you'll see the reconstructed query.

h) The Join Optimizer

i) The STRAIGHT_JOIN keyword forces the join to proceed in the order specified in the query.

ii) The join optimizer tries to produce a query execution plan tree with the lowest achievable cost. When possible, it examines all potential combinations of subtrees, beginning with all one-table plans.

iii) a join over n tables will have n-factorial combinations of join orders to examine. This is called the search space of all possible query plans, and it grows very quickly.

iv) When the search space grows too large, it can take far too long to optimize the query, so the server stops doing a full analysis. Instead, it resorts to shortcuts such as "greedy" searches when the number of tables exceeds the limit specified by the **optimizer_search_depth** variable (which you can change if necessary).

i) Sort Optimizations

i) If the values to be sorted will fit into the sort buffer, MySQL can perform the sort entirely in memory with a quicksort.

ii) If MySQL can't do the sort in memory, it performs it on disk by sorting the values in chunks. It uses a quicksort to sort each chunk and then merges the sorted chunks into the results.

iii) There are two filesort algorithms:

(a) Two passes (old): Reads row pointers and ORDER BY columns, sorts them, and then scans the sorted list and rereads the rows for output.

(b) Single pass (new): Reads all the columns needed for the query, sorts them by the ORDER BY columns, and then scans the sorted list and outputs the specified columns.

iv) MySQL might use much more temporary storage space for a filesort than you'd expect, because it allocates a fixed-size record for each tuple it will sort.

v) When sorting a join, MySQL might perform the filesort at two stages during the query execution:

(a) If the ORDER BY clause refers only to columns from the first table in the join order, MySQL can filesort this table and then proceed with the join. **If this happens, EXPLAIN shows "Using filesort" in the Extra column.**

(b) In all other circumstances, MySQL must store the query's results into a temporary table and then filesort the temporary table after the join finishes. In this case, **EXPLAIN shows "Using temporary; Using filesort" in the Extra column**.

(c) If there's a LIMIT, it is applied after the filesort, so the temporary table and the filesort can be very large.

(d) MySQL 5.6 introduces significant changes to how sorts are performed when only a subset of the rows will be needed, such as a LIMIT query. Instead of sorting the entire result set and then returning a portion of it, MySQL 5.6 can sometimes discard unwanted rows before sorting them.

4) The Query Execution Engine

a) MySQL simply follows the instructions given in the query execution plan. Many of the operations in the plan invoke methods implemented by the storage engine interface, also known as the handler API.

b) Each table in the query is represented by an instance of a handler. If a table appears three times in the query, for example, the server creates three handler instances.

c) MySQL actually creates the handler instances early in the optimization stage. The optimizer uses them to get information about the tables, such as their column names and index statistics.

d) To execute the query, the server just repeats the instructions until there are no more rows to examine.

5) Returning Results to the Client

a) Even queries that don't return a result set still reply to the client connection with information about the query, such as how many rows it affected.

b) If the query is cacheable, MySQL will also place the results into the query cache at this stage.

c) The server generates and sends results incrementally

6) **Limitations of the MySQL Query Optimizer**
   a) MySQL's "everything is a nested-loop join" approach to query execution isn't ideal for optimizing every kind of query.
   b) Correlated Subqueries: MySQL sometimes optimizes subqueries very badly. The worst offenders are IN() subqueries in the WHERE clause.
   c) UNION Limitations: MySQL sometimes can't "push down" conditions from the outside of a UNION to the inside, where they could be used to limit results or enable additional optimizations. For example, if you UNION together two tables and LIMIT the result to the first few rows, but MySQL will store all the result rows into a temporary table before retrieve just few rows from it. So we can avoid it by adding LIMIT redundantly to eacy query inside th e UNION.
   d) Parallel Execution: MySQL can't execute a single query in parallel on many CPUs. This is a feature offered by some other database servers.
   e) Equality Propagation: For example, consider a huge IN() list on a column the optimizer knows will be equal to some columns on other tables. When the list is very large, it can result in slower optimization and execution. There's no built-in workaround for this problem at the time of this writing.

iv. **Query Optimizer Hints**

1) **HIGH_PRIORITY and LOW_PRIORITY:** HIGH_PRIORITY tells MySQL to schedule a SELECT statement before other statements that might be waiting for table locks so they can modify data. LOW_PRIORITY is the reverse: it makes the statement wait at the very end of the queue if there are any other statements that want to access the tables—even if the other statements are issued after it.
   Note: These hints are effective on storage engines with table-level locking, but you should never need them on InnoDB or other engines with fine-grained locking and concurrency control. Be careful when using them on MyISAM, because they can disable concurrent inserts and greatly reduce performance. they simply affect how the server queues statements that are waiting for access to a table.
2) **DELAYED: This hint is for use with INSERT and REPLACE**. It lets the statement to which it is applied return immediately and places the inserted rows into a buffer, which will be inserted in bulk when the table is free. For example, logging insertion. There are many limitations; for example, delayed inserts are not implemented in all storage engines, and LAST _INSERT_ID() doesn't work with them.
3) **STRAIGHT_JOIN:** This hint can appear either just after the SELECT keyword in a SELECT statement, or in any statement between two joined tables. The first usage forces all tables in the query to be joined in the order in which they're listed in the statement. The second usage forces a join order on the two tables between which the hint appears.
4) **SQL_SMALL_RESULT and SQL_BIG_RESULT:** These hints are for SELECT statements. They tell the optimizer how and when to use temporary tables and sort in GROUP BY or DISTINCT queries. SQL_SMALL_RESULT tells the optimizer that the result set will be small and can be put into indexed temporary tables to avoid sorting for the grouping, whereas SQL_BIG_RESULT indicates that the result will be large and that it will be better to use temporary tables on disk with sorting.
5) **SQL_BUFFER_RESULT:** This hint tells the optimizer to put the results into a temporary table and release table locks as soon as possible.
6) **SQL_CACHE and SQL_NO_CACHE:** These hints instruct the server that the query either is or is not a candidate for caching in the query cache.
7) **SQL_CALC_FOUND_ROWS:** It tells MySQL to calculate a full result set when there's a LIMIT clause, even though it returns only LIMIT rows. You can retrieve the total number of rows it found via FOUND_ROWS()
8) **FOR UPDATE and LOCK IN SHARE MODE:** they control locking for SELECT statements, but only for storage engines that have row-level locks.
9) **USE INDEX, IGNORE INDEX, and FORCE INDEX:** These hints tell the optimizer which indexes to use or ignore for finding rows in a table FORCE INDEX is the same as USE INDEX, but it tells the optimizer that a table scan is extremely expensive compared to the index, even if the index is not very useful.

v. **Optimizing Specific Types of Queries**

1) **COUNT(*)**
   Approach 1:

```
mysql> SELECT COUNT(*) FROM world.City WHERE ID > 5;

-- if there are many rows with id bigger than 5, try following  SQL:
mysql> SELECT (SELECT COUNT(*) FROM world.City) - COUNT(*)
    -> FROM world.City WHERE ID <= 5;
```

   Approach 2: Using an approximation. Sometimes you don't need an accurate count, so you can just use an approximation. The optimizer's estimated rows in EXPLAIN often serves well for this. Just execute an EXPLAIN query instead of the real query.
   Approach 3: Using Covering index
   Approach 4: Summary tables
2) **JOIN QUERIES**
   a) Make sure there are indexes on the columns in the ON or USING clauses. Consider the join order when adding indexes.
      Note: If you're joining tables A and B on column c and the query optimizer decides to join the tables in the order B, A, you don't need to index the column on table B. Unused
   b) Try to ensure that any GROUP BY or ORDER BY expression refers only to columns from a single table, so MySQL can try to use an index for that operation.
   c) Be careful when upgrading MySQL, because the join syntax, operator precedence, and other behaviors have changed at various times.
3) **Optimizing GROUP BY and DISTINCT**
   a) Using SQL_BIG_RESULT and SQL_SMALL_RESULT optimizer hints to force MySQL to use a temporary table or a filesort to perform the grouping.
   b) If you need to group a join by a value that comes from a lookup table, it's usually more efficient to group by the lookup table's identifier than by the value.
   c) MySQL automatically orders grouped queries by the columns in the GROUP BY clause, unless you specify an ORDER BY clause explicitly. If you don't care about the order and you see this causing a filesort, you can use ORDER BY NULL to skip the automatic sort.
   d) You can also add an optional DESC or ASC keyword right after the GROUP BY clause to order the results in the desired direction by the clause's columns.
   e) You can do this with a WITH ROLLUP clause, but it might not be as well optimized as you need. The best approach might be to move the WITH ROLLUP functionality into your application code.

GROUP BY WITH ROLLUP	-- show the total profit for each year
	mysql> SELECT year, SUM(profit) FROM sales GROUP BY year;
	+--------+-----------------+
	| year | SUM(profit) |
	+--------+-----------------+
	| 2000 | 4525 |
	| 2001 | 3010 |
	+--------+-----------------+
	
	-- determine the total profit summed over all years
	mysql> SELECT year, SUM(profit) FROM sales GROUP BY year WITH ROLLUP;
	+--------+-----------------+
	| year | SUM(profit) |
	+--------+-----------------+
	| 2000 | 4525 |
	| 2001 | 3010 |
	| NULL | 7535 |
	+--------+-----------------+

4) **Optimizing LIMIT and OFFSET**
   PROBLEM: A frequent problem is having a high value for the offset. If your query looks like LIMIT 10000, 20, it is generating 10,020 rows and throwing away the first 10,000 of them, which is very expensive.
   SOLUTION 1: Do the offset on a covering index, rather than the full rows. You can then join the result to the full row and retrieve the additional columns you need.
   EXAMPLE:

```
mysql> SELECT film_id, description FROM sakila.film ORDER BY title LIMIT 50, 5;

-- if the table is very large, this query is better written as follows:
mysql> SELECT film.film_id, film.description
    -> FROM sakila.film
    -> INNER JOIN (
```

```
    -> SELECT film_id FROM sakila.film
    -> ORDER BY title LIMIT 50, 5
    -> ) AS lim USING(film_id);
```

This "deferred join" works because it lets the server examine as little data as possible in an index without accessing rows, and then, once the desired rows are found, join them against the full table to retrieve the other columns from the row. A similar technique applies to joins with LIMIT clauses.

SOLUTION 2: you can also convert the limit to a positional query, which the server can execute as an index range scan.

EXAMPLE:

```
-- if you precalculate and index a position column, you can rewrite the query as follows:
mysql> SELECT film_id, description FROM sakila.film
    -> WHERE position BETWEEN 50 AND 54 ORDER BY position;
```

5) **Optimizing SQL_CALC_FOUND_ROWS**
   PROBLEM: This option just tells the server to generate and throw away the rest of the result set, instead of stopping when it reaches the desired number of rows. That's very expensive.
   SOLUTION 1: A better design is to convert the pager to a "next" link. Assuming there are 20 results per page, the query should then use a LIMIT of 21 rows and display only 20. If the 21st row exists in the results, there's a next page, and you can render the "next" link.
   SOLUTION 2: Another possibility is to fetch and cache many more rows than you need.
   SOLUTION 3: you can also just estimate the full size of the result set by running an EXPLAIN query and looking at the rows column in the result
6) **Optimizing UNION**
   PROBLEM: MySQL always executes UNION queries by creating a temporary table and filling it with the UNION results. MySQL can't apply as many optimizations to UNION queries as you might be used to.
   SOLUTION 1: You might have to help the optimizer by manually "pushing down" WHERE, LIMIT, ORDER BY, and other conditions.
   SOLUTION 2: It's important to always use UNION ALL, unless you need the server to eliminate duplicate rows. If you omit thAeL L keyword, MySQL adds the distinct option to the temporary table, which uses the full row to determine uniqueness. This is quite expensive. Be aware that the ALL keyword doesn't eliminate the temporary table, though. MySQL always places results into a temporary table and then reads them out again, even when it's not really necessary
7) **User-Defined Variables**
   a) They disable the query cache.
   b) You can't use them where a literal or identifier is needed, such as for a table or column name, or in the LIMIT clause.
   c) They are connection-specific, so you can't use them for interconnection communication.
   d) If you're using connection pooling or persistent connections, they can cause seemingly isolated parts of your code to interact. (If so, it's because of a bug in your code or the connection pool, but it can still happen.)
   e) They are case sensitive in MySQL versions prior to 5.0, so beware of compatibility issues.
   f) You can't explicitly declare these variables' types, and the point at which types are decided for undefined variables differs across MySQL versions. The best thing to do is initially assign a value of 0 for variables you want to use for integers, 0.0 for floating-point numbers, or '' (the empty string) for strings. A variable's type changes when it is assigned to; MySQL's user-defined variable typing is dynamic.
   g) The optimizer might optimize away these variables in some situations, preventing them from doing what you want.
   h) Order of assignment, and indeed even the time of assignment, can be nondeterministic and depend on the query plan the optimizer chose. The results can be very confusing, as you'll see later.
   i) The := assignment operator has lower precedence than any other operator, so you have to be careful to parenthesize explicitly.
   j) Undefined variables do not generate a syntax error, so it's easy to make mistakes without knowing it.
8) **Use of User-Defined Variables:** Avoiding retrieving the row just modified

```
UPDATE t1 SET lastUpdated = NOW() WHERE id = 1;
SELECT lastUpdated FROM t1 WHERE id = 1;

-- We rewrote those queries to use a variable instead, as follows:
UPDATE t1 SET lastUpdated = NOW() WHERE id = 1 AND @now := NOW();
SELECT @now;
```

9) **Use of User-Defined Variables:** Writing a lazy UNION

```
SELECT id FROM users WHERE id = 123
UNION ALL
SELECT id FROM users_archived WHERE id = 123;

-- lazy UNION approach:
SELECT GREATEST(@found := −1, id) AS id, 'users' AS which_tbl
FROM users WHERE id = 1
UNION ALL
SELECT id, 'users_archived'
FROM users_archived WHERE id = 1 AND @found IS NULL
UNION ALL
SELECT 1, 'reset' FROM DUAL WHERE ( @found := NULL ) IS NOT NULL;
```

5. Advanced MySQL Features

a. Partitioned Tables

i. A partitioned table is a single logical table that is composed of multiple physical subtables.
ii. Partitioning is a kind of black box that hides the underlying partitions from you at the SQL layer, although you can see them quite easily by looking at the filesystem, where you'll see the component tables with a hash-delimited naming convention.
iii. MySQL decides which partition holds each row of data based on the PARTITION BY clause that you define for the table. The query optimizer can prune partitions when you execute queries, so the queries don't examine all partitions—just the ones that hold the data you are looking for.
iv. How Partitioning Works
   1) How it works on queries:
      a) SELECT queries: When you query a partitioned table, the partitioning layer opens and locks all of the underlying partitions, the query optimizer determines whether any of the partitions can be ignored (pruned), and then the partitioning layer forwards the handler API calls to the storage engine that manages the partitions.
      b) INSERT queries: When you insert a row, the partitioning layer opens and locks all partitions, determines which partition should receive the row, and forwards the row to that partition.
      c) DELETE queries: When you delete a row, the partitioning layer opens and locks all partitions, determines which partition contains the row, and forwards the deletion request to that partition.
      d) UPDATE queries: When you modify a row, the partitioning layer (you guessed it) opens and locks all partitions, determines which partition contains the row, fetches the row, modifies the row and determines which partition should contain the new row, forwards the row with an insertion request to the destination partition, and forwards the deletion request to the source partition.
   2) Although the partitioning layer opens and locks all partitions, this doesn't mean that the partitions remain locked. A storage engine such as InnoDB, which handles its own locking at the row level, will instruct the partitioning layer to unlock the partitions. This lock-and-unlock cycle is similar to how queries against ordinary InnoDB tables are executed.
   3) Example:

```
CREATE TABLE sales (
order_date DATETIME NOT NULL,
-- Other columns omitted
)
```

```
ENGINE=InnoDB
PARTITION BY RANGE(YEAR(order_date)) (
    PARTITION p_2010 VALUES LESS THAN (2010),
    PARTITION p_2011 VALUES LESS THAN (2011),
    PARTITION p_2012 VALUES LESS THAN (2012),
    PARTITION p_catchall VALUES LESS THAN MAXVALUE );
```

v. Partitioning can be very beneficial, especially in specific scenarios:
   1) When the table is much too big to fit in memory, or when you have "hot" rows at the end of a table that has lots of historical data.
   2) Partitioned data is easier to maintain than nonpartitioned data. For example, it's easier to discard old data by dropping an entire partition, which you can do quickly. You can also optimize, check, and repair individual partitions.
   3) Partitioned data can be distributed physically, enabling the server to use multiple hard drives more efficiently.
   4) You can use partitioning to avoid some bottlenecks in specific workloads, such as per-index mutexes with InnoDB or per-inode locking with the ext3 filesystem.
   5) If you really need to, you can back up and restore individual partitions, which is very helpful with extremely large datasets.

vi. A few limitations apply to partitioned tables. Here are the most important ones:
   1) There's a limit of 1,024 partitions per table.
   2) In MySQL 5.1, the partitioning expression must be an integer or an expression that returns an integer. In MySQL 5.5, you can partition by columns in certain cases.
   3) Any primary key or unique index must include all columns in the partitioning expression.
   4) You can't use foreign key constraints.
   5) All partitions have to use the same storage engine.
   6) There are some restrictions on the functions and expressions you can use in a partitioning function.
   7) Some storage engines don't work with partitioning.
   8) For MyISAM tables, you can't use LOAD INDEX INTO CACHE.
   9) For MyISAM tables, a partitioned table requires more open file descriptors than a normal table containing the same data. Even though it looks like a single table, as you know, it's really many tables. As a result, a single table cache entry can create many file descriptors. Therefore, even if you have configured the table cache to protect your server against exceeding the operating system's per-process filedescriptor limits, partitioned tables can cause you to exceed that limit anyway.

vii. Other partitioning techniques we've seen include:
   1) You can partition by key to help reduce contention on InnoDB mutexes.
   2) You can partition by range using a modulo function to create a round-robin table that retains only a desired amount of data. For example, you can partition datebased data by day modulo 7, or simply by day of week, if you want to retain only the most recent days of data.
   3) Suppose you have a table with an autoincrementing idprimary key, but you want to partition the data temporally so the "hot" recent data is clustered together. You can't partition by a timestamp column unless you include it in the primary key, but that defeats the purpose of a primary key. You can partition by an expression such as HASH(id DIV 1000000), which creates a new partition for each million rows inserted. This achieves the goal without requiring you to change the primary key.
   4) It has the added benefit that you don't need to constantly create partitions to hold new ranges of dates, as you'd need to do with range-based partitioning.

viii. What Can Go Wrong
   1) NULLs can defeat pruning: Partitioning works in a funny way when the result of the partitioning function can be NULL: it treats the first partition as special.
   2) Mismatched PARTITION BY and index: If you define an index that doesn't match the partitioning clause, queries might not be prunable.
   3) Selecting partitions can be costly.
   4) Opening and locking partitions can be costly: Opening and locking occur before pruning, so this isn't a prunable overhead.
   5) Maintenance operations can be costly: Some partition maintenance operations are very quick, such as creating or dropping partitions. (Dropping the underlying table might be slow, but that's another matter.) Other operations, such as REORGANIZE PARTITION, operate similarly to the way ALTER works: by copying rows around. For example, REORGANIZE PARTITION works by creating a new temporary partition, moving rows into it, and deleting the old partition when it's done.

ix. Optimizing Queries
   1) it's very important to specify the partitioned key in the WHERE clause, even if it's otherwise redundant, so the optimizer can prune unneeded partitions.
   2) **You can use EXPLAIN PARTITIONS to see whether the optimizer is pruning partitions.**
      For example:

```
mysql> EXPLAIN PARTITIONS SELECT * FROM sales \G
*************************** 1. row ***************************
id: 1
select_type: SIMPLE
table: sales_by_day
partitions: p_2010,p_2011,p_2012
...
```

   3) **MySQL can prune only on comparisons to the partitioning function's columns. It cannot prune on the result of an expression, even if the expression is the same as the partitioning function. For example: SELECT * FROM sales_by_day WHERE YEAR(day) = 2010**

b. Merge Table

i. Merge tables are sort of an earlier, simpler kind of partitioning with different restrictions and fewer optimizations.

ii. The merge table is really just a container that holds the real tables. You specify which tables to include with a special UNION syntax to CREATE TABLE. Here's an example that demonstrates many aspects of merge tables:

```
mysql> CREATE TABLE t1(a INT NOT NULL PRIMARY KEY)ENGINE=MyISAM;
mysql> CREATE TABLE t2(a INT NOT NULL PRIMARY KEY)ENGINE=MyISAM;
mysql> INSERT INTO t1(a) VALUES(1),(2);
mysql> INSERT INTO t2(a) VALUES(1),(2);
mysql> CREATE TABLE mrg(a INT NOT NULL PRIMARY KEY)
-> ENGINE=MERGE UNION=(t1, t2) INSERT_METHOD=LAST;
mysql> SELECT a FROM mrg;
+------+
| a |
+------+
| 1 |
| 1 |
| 2 |
| 2 |
+------+
```

iii. The INSERT_METHOD=LAST instruction to the table tells MySQL to send all INSERT statements to the last table in the merge. Specifying FIRST or LAST is the only control you have over where rows inserted into the merge table are placed

```
mysql> INSERT INTO mrg(a) VALUES(3);
mysql> SELECT a FROM t2;
+---+
| a |
+---+
| 1 |
| 2 |
| 3 |
+---+
```

iv. Dropping a merge table leaves its "child" tables untouched, but dropping one of the child tables has a different effect, which is operating system–specific. On GNU/Linux, for

example, the underlying table's file descriptor stays open and the table continues to exist, but only via the merge table.

v. A variety of other limitations and special behaviors exist. Here are some aspects of merge tables you should keep in mind:
   1) The CREATE statement that creates a merge table doesn't check that the underlying tables are compatible. If the underlying tables are defined slightly differently, MySQL might create a merge table that it can't use later. Also, if you alter one of the underlying tables after creating a valid merge table, it will stop working and you'll see this error: "ERROR 1168 (HY000): Unable to open underlying table which is differently defined or of non-MyISAM type or doesn't exist."
   2) REPLACE doesn't work at all on a merge table, and AUTO_INCREMENT won't work as you might expect.
   3) Queries that access a merge table access every underlying table. This can make single-row key lookups relatively slow, compared to a lookup in a single table. Therefore, it's a good idea to limit the number of underlying tables in a merge table, especially if it is the second or later table in a join. The less data you access with each operation, the more important the cost of accessing each table becomes, relative to the entire operation.

c. Views

i. MySQL can use both methods. It calls the two algorithms **MERGE** and **TEMPTABLE**, and it tries to use the MERGE algorithm when possible. **You can see the results of the query rewrite with EXPLAIN EXTENDED, followed by SHOW WARNINGS. If a view uses the TEMPTABLE algorithm, EXPLAIN will usually show it as a DERIVED table**
Details: Assume has view:

```
mysql> CREATE VIEW Oceania AS
    -> SELECT * FROM Country WHERE Continent = 'Oceania'
    -> WITH CHECK OPTION;
```

MERGE	TEMPTABLE
SELECT Code, Name FROM Oceania WHERE Name = 'Australia'; **Translate to:** mysql> SELECT Code, Name FROM Country -> WHERE Continent = 'Oceania' AND Name = 'Australia';	SELECT Code, Name FROM Oceania WHERE Name = 'Australia'; **Translate to** mysql> CREATE TEMPORARY TABLE TMP_Oceania_123 AS -> SELECT * FROM Country WHERE Continent = 'Oceania'; mysql> SELECT Code, Name FROM TMP_Oceania_123 WHERE Name = 'Australia';

i. MySQL uses TEMPTABLE when the view definition contains GROUP BY, DISTINCT, aggregate functions, UNION, subqueries, or any other construct that doesn't preserve a one-to-one relationship between the rows in the underlying base tables and the rows returned from the view.
ii. A view is not updatable if it contains GROUP BY, UNION, an aggregate function, or any of a few other exceptions. A query that changes data might contain a join, but the columns to be changed must all be in a single table. Any view that uses the TEMPTABLE algorithm is not updatable.
iii. The CHECK OPTION clause, which we included when we created the view in the previous section, ensures that any rows changed through the view continue to match the view's WHERE clause after the change.
iv. Limitations of Views
   1) MySQL does not support the materialized views. And also doesn't support indexed views. You can emulate materialized and/or indexed views by building cache and summary tables
   2) MySQL doesn't preserve your original view SQL, so if you ever try to edit a view by executing SHOW CREATE VIEW and changing the resulting SQL, you're in for a nasty surprise. The query will be expanded to the fully canonicalized and quoted internal format, without the benefit of formatting, comments, and indenting.

d. Query Cache

i. MySQL stores complete result sets for SELECT statements. The MySQL query cache holds the exact bits that a completed query returned to the client.
ii. The best approach is actually to disable it by default, and configure a small query cache (no more than a few dozen megabytes) only if it's very beneficial.
iii. The way MySQL checks for a cache hit is simple: the cache is a lookup table. The lookup key is a hash of the query text itself, the current database, the client protocol version, and a handful of other things that might affect the actual bytes in the query's result.
iv. Any difference in character case, spacing, or comments—any difference at all—will prevent a query from matching a previously cached version.
v. Another caching consideration is that the query cache will not store a result unless the query that generated it was deterministic. Thus, any query that contains a nondeterministic function, such as NOW() or CURRENT_DATE(), will not be cached. Similarly, functions such as CURRENT_USER() or CONNECTION_ID() might vary when executed by different users, thereby preventing a cache hit.
vi. Enabling the query cache adds some overhead for both reads and writes:
   1) Read queries must check the cache before beginning.
   2) If the query is cacheable and isn't in the cache yet, there's some overhead due to storing the result after generating it.
   3) There's overhead for write queries, which must invalidate the cache entries for queries that use tables they change. Invalidation can be very costly if the cache is fragmented and/or large (has many cached queries, or is configured to use a large amount of memory).
vii. For InnoDB users, when a statement inside a transaction modifies a table, the server invalidates any cached queries that refer to the table. Invalidation can become a very serious problem with a large query cache. If there are many queries in the cache, the invalidation can take a long time and cause the entire system to stall while it works. This is because there's a single global lock on the query cache, which will block all queries that need to access it. Accessing happens both when checking for a hit and when checking whether there are any queries to invalidate.
viii. Cache related configuration
   1) query_cache_type:
   Whether the query cache is enabled. Possible values are OFF, ON, or DEMAND, where the latter means that only queries containing the SQL_CACHE modifier are eligible for caching. This is both a session-level and a global variable.
   2) query_cache_size:
   The total memory to allocate to the query cache, in bytes. This must be a multiple of 1,024 bytes, so MySQL might use a slightly different value than the one you specify.
   3) query_cache_min_res_unit:
   The minimum size when allocating a block. We explained this setting previously; it's discussed further in the next section.
   4) query_cache_limit:
   The largest result set that MySQL will cache. Queries whose results are larger than this setting will not be cached. Remember that the server caches results as it generates them, so it doesn't know in advance when a result will be too large to cache. If the result exceeds the specified limit, MySQL will increment the Qcache_ not_cached status variable and discard the results cached so far. If you know this happens a lot, you can add the SQL_NO_CACHE hint to queries you don't want to incur this overhead.
   5) query_cache_wlock_invalidate:
   Whether to serve cached results that refer to tables other connections have locked. The default value is OFF, which makes the query cache change the server's semantics because it lets you read cached data from a table another connection has locked, which you wouldn't normally be able to do. Changing it to ON will keep you from reading this data, but it might increase lock waits. This really doesn't matter for most applications, so the default is generally fine.
ix. General Query Cache Optimizations
   1) Having multiple smaller tables instead of one huge one can help the query cache. This design effectively makes the invalidation strategy work at a finer level of granularity. Don't let this unduly influence your schema design, though, as other factors can easily outweigh the benefit.
   2) It's more efficient to batch writes than to do them singly, because this method invalidates cached cache entries only once. (Be careful not to delay and batch so much that the invalidations caused by the writes will stall the server for too long, however.)
   3) We've noticed that the server can stall for a long time while invalidating entries in or pruning a very large query cache. A possible solution is to not make query_cache_size very large, but in some cases you simply have to disable it altogether, because nothing is small enough.
   4) You cannot control the query cache on a per-database or per-table basis, but you can include or exclude individual queries with the SQL_CACHE and SQL_NO_CACHE modifiers in the SELECT statement. You can also enable or disable the query cache on a per-connection basis by setting the session-level query_cache_type server ariable to the appropriate value.
   5) For a write-heavy application, disabling the query cache completely might improve performance. Doing so eliminates the overhead of caching queries that would be invalidated soon anyway. Remember to set query_cache_size to 0 when you disable it, so it doesn't consume any memory.
   6) Disabling the query cache might be beneficial for a read-heavy application, too, because of contention on the single query cache mutex. If you need good performance at high concurrency, be sure to validate it with high-concurrency tests, because enabling the query cache and testing at low concurrency can be very misleading.

e. Foreign Key Constraints

i. InnoDB is currently the only bundled storage engine that supports foreign keys in MySQL, limiting your choice of storage engines if you require them.
ii. You can sometimes use triggers instead of foreign keys.

iii. Instead of using foreign keys as constraints, it's often a good idea to constrain the values in the application.

f. Storing Code

i. Four types: triggers, stored procedures, stored functions and events

ii. The advantages:

1) It runs where the data is, so you can save bandwidth and reduce latency by running tasks on the database server.
2) It's a form of code reuse.
3) It can ease release policies and maintenance.
4) It can provide some security advantages and a way to control privileges more finely. A common example is a stored procedure for funds transfer at a bank: the procedure transfers the money within a transaction and logs the entire operation for auditing. You can let applications call the stored procedure without granting access to the underlying tables.
5) The server caches stored procedure execution plans, which lowers the overhead of repeated calls.
6) Because it's stored in the server and can be deployed, backed up, and maintained with the server, stored code is well suited for maintenance jobs. It doesn't have any external dependencies.
7) It enables division of labor between application programmers and database programmers. It can be preferable for a database expert to write the stored procedures, as not every application programmer is good at writing efficient SQL queries.

iii. Disadvantages include the following:

1) MySQL doesn't provide good developing and debugging tools.
2) The language is slow and primitive compared to application languages. The number of functions you can use is limited, and it's hard to do complex string manipulations and write intricate logic.
3) Stored code can actually add complexity to deploying your application.
4) Because stored routines are stored with the database, they can create a security vulnerability. Having nonstandard cryptographic functions inside a stored routine, for example, will not protect your data if the database is compromised. If the cryptographic function were in the code, the attacker would have to compromise both the code and the database.
5) Storing routines moves the load to the database server, which is typically harder to scale and more expensive than application or web servers.
6) MySQL doesn't give you much control over the resources stored code can allocate, so a mistake can bring down the server.
7) MySQL's implementation of stored code is pretty limited
8) It is hard to profile code with stored procedures in MySQL.
9) It doesn't play well with statement-based binary logging or replication.

iv. Stored Procedures and Functions

1) Disadvantages:
   a) The optimizer doesn't use the DETERMINISTIC modifier in stored functions to optimize away multiple calls within a single query.
   b) The optimizer cannot estimate how much it will cost to execute a stored function.
   c) Each connection has its own stored procedure execution plan cache. If many connections call the same procedure, they'll waste resources caching the same execution plan over and over. (If you use connection pooling or persistent connections, the execution plan cache can have a longer useful life.)
   d) Stored routines and replication are a tricky combination
2) We usually prefer to keep stored routines small and simple.
3) stored procedures can be much faster for certain types of operations—especially when a single stored procedure call with a loop inside it can replace many small queries.

v. Triggers

1) Limitation in MySQL
   a) You can have only one trigger per table for each event
   b) MySQL supports only row-level triggers—that is, triggers always operate FOR EACH ROW rather than for the statement as a whole.
   c) They can obscure what your server is really doing, because a simple statement can make the server perform a lot of "invisible" work.
   d) Triggers can be hard to debug, and it's often difficult to analyze performance bottlenecks when triggers are involved.
   e) Triggers can cause nonobvious deadlocks and lock waits. If a trigger fails the original query will fail, and if you're not aware the trigger exists, it can be hard to decipher the error code.

vi. Events

1) Events are a new form of stored code in MySQL 5.1.
2) The usual practice is to wrap the complex SQL in a stored procedure, so the event merely needs to perform a CALL.
3) Example:

```
CREATE EVENT optimize_somedb ON SCHEDULE EVERY 1 WEEK
DO
CALL optimize_tables('somedb');
```

g. Cursors

i. MySQL provides read-only, forward-only server-side cursors that you can use only from within a MySQL stored procedure or the low-level client API.

ii. MySQL's cursors are read-only because they iterate over temporary tables rather than the tables where the data originated.

iii. The most important thing to know is that a cursor executes the entire query when you open it

```
1 CREATE PROCEDURE bad_cursor()
2 BEGIN
3 DECLARE film_id INT;
4 DECLARE f CURSOR FOR SELECT film_id FROM sakila.film;
5 OPEN f;
6 FETCH f INTO film_id;
7 CLOSE f;
8 END

-- This example shows that you can close a cursor before iterating through all of its results.
-- But in MySQL it causes a lot of unnecessary work.
-- Profiling this procedure with SHOW STATUS shows that it does 1,000 index reads in step 5.
-- It means step loads all the 1000 rows into cursor even if you only want single row.
```

h. Prepared Statements

i. The SQL Interface to Prepared Statements

```
mysql> SET @sql := 'SELECT actor_id, first_name, last_name
    -> FROM sakila.actor WHERE first_name = ?';
mysql> PREPARE stmt_fetch_actor FROM @sql;
mysql> SET @actor_name := 'Penelope';
mysql> EXECUTE stmt_fetch_actor USING @actor_name;
+----------+------------+-----------+
| actor_id | first_name | last_name |
+----------+------------+-----------+
| 1 | PENELOPE | GUINESS |
| 54 | PENELOPE | PINKETT |
| 104 | PENELOPE | CRONYN |
| 120 | PENELOPE | MONROE |
+----------+------------+-----------+
mysql> DEALLOCATE PREPARE stmt_fetch_actor;
```

ii. Using prepared statements can be more efficient than executing a query repeatedly, for several reasons:

1) The server has to parse the query only once.
2) The server has to perform some query optimization steps only once, as it caches a partial query execution plan.
3) Sending parameters via the binary protocol is more efficient than sending them as ASCII text.
4) Only the parameters—not the entire query text—need to be sent for each execution, which reduces network traffic.
5) MySQL stores the parameters directly into buffers on the server, which eliminates the need for the server to copy values around in memory.

iii. Limitations of Prepared Statements
1) Prepared statements are local to a connection, so another connection cannot use the same handle. For the same reason, a client that disconnects and reconnects loses the statements.
2) Prepared statements cannot use the query cache in MySQL versions prior to 5.1.
3) It's not always more efficient to use prepared statements. If you use a prepared statement only once, you might spend more time preparing it than you would just executing it as normal SQL.
4) You cannot currently use a prepared statement inside a stored function (but you can use prepared statements inside stored procedures).
5) You can accidentally "leak" a prepared statement by forgetting to deallocate it. This can consume a lot of resources on the server. Also, because there is a single global limit on the number of prepared statements, a mistake such as this can interfere with other connections' use of prepared statements.
6) Some operations, such as BEGIN, cannot be performed in prepared statements.

6. Optimizing Server Settings

a. How MySQL's Configuration Works

i. where MySQL gets configuration information:
1) From command-line arguments
2) Settings in its configuration file. On Unix-like systems, the configuration file is typically located at **/etc/my.cnf or /etc/mysql/my.cn**f.

ii. you can ask MySQL to find where are the configuration files:

```
$ which mysqld
/usr/sbin/mysqld
$ /usr/sbin/mysqld --verbose --help | grep -A 1 'Default options'
Default options are read from the following files in the given order:
/etc/mysql/my.cnf ~/.my.cnf /usr/etc/my.cnf
```

iii. The configuration file is divided into sections, each of which begins with a line that contains the section name in square brackets.
iv. Many client programs also read the *client* section, which gives you a place to put common settings.
v. The server usually reads the *mysqld* section.
vi. Configuration settings are written in all lowercase, with words separated by underscores or dashes. Followings are equivalent.
`/usr/sbin/mysqld --auto-increment-offset=5`
`/usr/sbin/mysqld --auto_increment_offset=5`
vii. Configuration settings can have several scopes. Some settings are server-wide (global scope); others are different for each connection (session scope); and others are per-object.
viii. Many session-scoped variables have global equivalents, which you can think of as defaults.
ix. In addition to setting variables in the configuration files, you can also change many (but not all) of them while the server is running. MySQL refers to these as dynamic configuration variables.

```
SET             sort_buffer_size = <value>;
SET GLOBAL   sort_buffer_size = <value>;
SET       @@sort_buffer_size := <value>;
SET       @@session.sort_buffer_size := <value>;
SET       @@global.sort_buffer_size := <value>;
```

x. If you set variables dynamically, those settings will be lost when MySQL shuts down. If you want to keep the settings, you'll have to update your configuration file as well.
xi. If you set a variable's global value while the server is running, the values for the current session and any other existing sessions are not affected.
xii. You should inspect the output of SHOW GLOBAL VARIABLES after each change to make sure it's had the desired effect.
xiii. Many variables can be specified with a suffix, such as 1M for one megabyte. However, this works only in the configuration file or as a command-line argument. When you use the SQL SET command, you must use the literal value 1048576, or an expression such as 1024 * 1024. You can't use expressions in configuration files.
xiv. There is also a special value you can assign to variables with the SET command: the keyword DEFAULT.
`SET @@session.sort_buffer_size := DEFAULT;`

b. MySQL Configuration File

i. It's actually a weakness that MySQL is so configurable, because it makes it seem as though you should spend a lot of time on configuration, when in fact most things are fine at their defaults, and you are often better off setting and forgetting.
ii. Our base file looks like this:

```
[mysqld]
# GENERAL
datadir = /var/lib/mysql
socket = /var/lib/mysql/mysql.sock
pid_file = /var/lib/mysql/mysql.pid
user = mysql
port = 3306
storage_engine = InnoDB
# INNODB
innodb_buffer_pool_size = <value>
innodb_log_file_size = <value>
innodb_file_per_table = 1
innodb_flush_method = O_DIRECT
# MyISAM
key_buffer_size = <value>
# LOGGING
log_error = /var/lib/mysql/mysql-error.log
log_slow_queries = /var/lib/mysql/mysql-slow.log
# OTHER
tmp_table_size = 32M
max_heap_table_size = 32M
query_cache_type = 0
query_cache_size = 0
max_connections = <value>
thread_cache_size = <value>
table_cache_size = <value>
open_files_limit = 65535
[client]
socket = /var/lib/mysql/mysql.sock
port = 3306
```

c. Configuration Memory Usage

i. Configuring MySQL to use memory correctly is vital to good performance.
ii. You can approach memory configuration in steps:

1) Determine the absolute upper limit of memory MySQL can possibly use.
    a) MySQL runs in a single process with multiple threads
2) Determine how much memory MySQL will use for per-connection needs, such as sort buffers and temporary tables.
    a) MySQL needs a small amount of memory just to hold a connection (thread) open.
    b) It also requires a base amount of memory to execute any given query.
    c) You'll need to set aside enough memory for MySQL to execute queries during peak load times.
    d) It's useful to know how much memory MySQL will consume during peak usage, but some usage patterns can unexpectedly consume a lot of memory, which makes this hard to predict. Prepared statements are one example, because you can have many of them open at once. Another example is the InnoDB data dictionary (more about this later).
    e) Rather than calculating worst cases, a better approach is to watch your server under a real workload and see how much memory it uses, which you can see by watching the process's virtual memory size. In many Unix-like systems, this is reported in the VIRT column in top, or VSZ in ps
3) Determine how much memory the operating system needs to run well. Include memory for other programs that run on the same machine, such as periodic jobs.
    a) The best indication that the operating system has enough memory is that it's not actively swapping (paging) virtual memory to disk.
4) Assuming that it makes sense to do so, use the rest of the memory for MySQL's caches, such as the InnoDB buffer pool.
    a) If the server is dedicated to MySQL, any memory you don't reserve for the operating system or for query processing is available for caches.
    b) MySQL needs more memory for caches than anything else. It uses caches to avoid disk access, which is orders of magnitude slower than accessing data in memory.
    c) The following are the most important caches to consider for most installations:
        i) The InnoDB buffer pool
        ii) The operating system caches for InnoDB log files and MyISAM data
        iii) MyISAM key caches
        iv) The query cache
        v) Caches you can't really configure, such as the operating system's caches of binary logs and table definition files
    d) Using InnoDB Buffer Pool
        i) If you use mostly InnoDB tables, the InnoDB buffer pool probably needs more memory than anything else.
        ii) The InnoDB buffer pool doesn't just cache indexes: it also holds row data, the adaptive hash index, the insert buffer, locks, and other internal structures. InnoDB also uses the buffer pool to help it delay writes, so it can merge many writes together and perform them sequentially. In short, InnoDB relies heavily on the buffer pool, and you should be sure to allocate enough memory to it
        iii) You can use variables from SHOW commands or tools such as innotop to monitor your InnoDB buffer pool's memory usage.
        iv) If you don't have much data, and you know that your data won't grow quickly, you don't need to overallocate memory to the buffer pool. It's not really beneficial to make it much larger than the size of the tables and indexes that it will hold.
        v) Large buffer pools come with some challenges, such as long shutdown and warmup times. If there are a lot of dirty (modified) pages in the buffer pool InnoDB can take a long time to shut down, because it writes the dirty pages to the data files upon shutdown. You can force it to shut down quickly, but then it just has to do more recovery when it restarts.
        vi) If you know in advance when you need to shut down, you can change the varia innodb_max_dirty_pages_pctble at runtime to a lower value, wait for the flush thread to clean up the buffer pool, and then shut down once the number of dirty pages becomes small.
        vii) You can monitor the number of dirty pages by watching the Innodb_buffer_pool_pages_dirty server status variable or using innotop to monitor SHOW INNODB STATUS. (SHOW STATUS LIKE 'Innodb_buffer_pool_pages_dirty')
        viii) Lowering the value of the innodb_max_dirty_pages_pct variable doesn't actually guarantee that InnoDB will keep fewer dirty pages in the buffer pool. Instead, it controls the threshold at which InnoDB stops being "lazy."
        ix) InnoDB's default behavior is to flush dirty pages with a background thread, merging writes together and performing them sequentially for efficiency. This behavior is called "lazy" because it lets InnoDB delay flushing dirty pages in the buffer pool, unless it needs to use the space for some other data. When the percentage of dirty pages exceeds the threshold, InnoDB will flush pages as quickly as it can to try to keep the dirty page count lower. InnoDB will also go into "furious flushing" mode when there isn't enough space left in the transaction logs, which is one reason that large logs can improve performance.
        x) When you have a large buffer pool, especially in combination with slow disks, the server might take a long time (many hours or even days) to warm up after a restart.
            (a) you might benefit from using Percona Server's feature to reload the pages after restart. This can reduce warmup times to a few minutes. This is especially beneficial on replicas, which pay an extra warmup penalty due to the single-threaded nature of replication.
            (b) If you can't use Percona Server's fast warmup feature, some people issue full-table scans or index scans immediately after a restart to load indexes into the buffer pool. This is crude, but can sometimes be better than nothing. You can use the init_file setting to accomplish this. You can place SQL into a file that's executed when MySQL starts up. The filename must be specified in the init_file option, and the file can include multiple SQL commands, each on a single line (no comments are allowed).
    e) Thread Cache
        i) The thread cache holds threads that aren't currently associated with a connection but are ready to serve new connections.
        ii) When there's a thread in the cache and a new connection is created, MySQL removes the thread from the cache and gives it to the new connection. When the connection is closed, MySQL places the thread back into the cache, if there's room. If there isn't room, MySQL destroys the thread.
        iii) As long as MySQL has a free thread in the cache it can respond rapidly to connection requests, because it doesn't have to create a new thread for each connection.
        iv) The thread_cache_size variable specifies the number of threads MySQL can keep in the cache. You probably won't need to configure this value unless your server gets many connection requests.
        v) To check whether the thread cache is large enough, watch the *Threads_created* status variable. We generally try to keep the thread cache large enough that we see fewer than 10 new threads created each second, but it's often pretty easy to get this number lower than 1 per second.
        vi) A good approach is to watch the Threads_connected variable and try to set thread_cache_size large enough to handle the typical fluctuation in your workload.
        vii) Making the thread cache very large is probably not necessary for most uses, but keeping it small doesn't save much memory, so there's little benefit in doing so. Each thread that's in the thread cache or sleeping typically uses around 256 KB of memory. This is not very much compared to the amount of memory a thread can use when a connection is actively processing a query.
        viii) In general, you should keep your thread cache large enough that Threads_created doesn't increase very often.
        ix) If this is a very large number, however (e.g., many thousand threads), you might want to set it lower because some operating systems don't handle very large numbers of threads well, even when most of them are sleeping.
    f) Table Cache
        i) The table cache is similar in concept to the thread cache, but it stores objects that represent tables. Each object in the cache contains the associated table's parsed .frm file, plus other data. Exactly what else is in the object depends on the table's storage engine.
        ii) In MySQL 5.1, the table cache is separated into two parts: a cache of open tables and a table definition cache (configured via the table_open_cache and table_definition_cache variables).
        iii) If the Opened_tables status variable is large or increasing, the table cache might not be large enough, and you can consider increasing the table_cache system variable (or table_open_cache, in MySQL 5.1). However, note that this counter increases when you create and drop temporary tables, so if you do that a lot, you'll never get the counter to stop increasing.
        iv) One downside to making the table cache very large is that it might cause longer shutdown times when your server has a lot of MyISAM tables, because the key blocks have to be flushed and the tables have to be marked as no longer open. It can also make FLUSH TABLES WITH READ LOCK take a long time to complete, for the same reason.
    g) The InnoDB Data Dictionary
        i) InnoDB has its own per-table cache, variously called a table definition cache or data dictionary, which you cannot configure in current versions of MySQL.
        ii) When InnoDB opens a table, it adds a corresponding object to the data dictionary. Each table can take up 4 KB or more of memory (although much less space is required in MySQL 5.1). Tables are not removed from the data dictionary when they are closed.
        iii) As a result, the server can appear to leak memory over time, due to an ever-increasing number of entries in the dictionary cache. It isn't truly leaking memory; it just isn't implementing any kind of cache expiration. If this is a problem for you, you can use Percona Server, which has an option to limit the data dictionary's size by removing tables that are unused. There is a similar feature in the yet-to-be-released MySQL 5.6.

d. Configuring MySQL's I/O Behavior
    i. InnoDB I/O Configuration
        1) InnoDB uses its log to reduce the cost of committing transactions. Instead of flushing the buffer pool to disk when each transaction commits, it logs the transactions.
        2) **InnoDB uses its log to convert this random disk I/O into sequential I/O**. Once the log is safely on disk, the transactions are permanent, even though the changes haven't

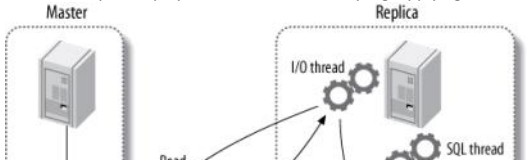
been written to the data files yet. If something bad happens (such as a power failure), InnoDB can replay the log and recover the committed transactions. InnoDB uses a background thread to flush the changes to the data files intelligently. This thread can group writes together and make the data writes sequential,

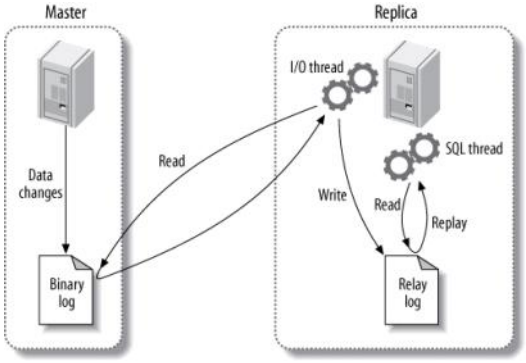
3) InnoDB It writes to the log in a circular fashion: when it reaches the end of the log, it wraps around to the beginning. It can't overwrite a log record if the changes contained there haven't been applied to the data files
4) Configuration Details
    a) If the log is too small, InnoDB will have to do more checkpoints, causing more log writes.
    b) On the other hand, if the log is too large, InnoDB might have to do a lot of work when it recovers. This can greatly increase recovery time, although this process is much more efficient in newer MySQL versions.
5) When InnoDB changes any data, it writes a record of the change into its log buffer, which it keeps in memory. InnoDB flushes the buffer to the log files on disk when the buffer gets full, when a transaction commits, or once per second—whichever comes first.
6) Increasing the buffer size, which is 1 MB by default, can help reduce I/O if you have large transactions. The variable that controls the buffer size is called innodb_log_buffer_size. You usually don't need to make the buffer very large. The recommended range is 1 to 8 MB, and this usually will be enough unless you write a lot of huge BLOB records.
7) When InnoDB flushes the log buffer to the log files on disk, it locks the buffer with a mutex, flushes it up to the desired point, and then moves any remaining entries to the front of the buffer.

e. Configuring MySQL Concurrency (Details to see the book)
f. Complete the Basic Configuration
    i. **tmp_table_size** and **max_heap_table_size**:
        1) These settings control how large an in-memory temporary table using the Memory storage engine can grow.
        2) **If an implicit temporary table's size exceeds either of these settings, it will be converted to an on-disk MyISAM table so it can keep growing**.
        3) You should simply set both of these variables to the same value.
        4) Beware of setting this variable too large. It's good for temporary tables to live in memory, but if they're simply going to be huge, it's actually best for them to just use on-disk tables, or you could run the server out of memory.
        5) You can look at how the server's SHOW STATUS counters change over time to understand how often you create temporary tables and whether they go to disk.
        6) You can't tell whether a table was created in memory and then converted to on-disk or just created on-disk to begin with (perhaps because of a BLOB column), but you can at least see how often the tables go to disk. Examine the Created_tmp_disk_tables and Created_tmp_tables variables.
    ii. **max_connections**
        1) This setting acts like an emergency brake to keep your server from being overwhelmed by a surge of connections from the application.
        2) Set max_connections high enough to accommodate the usual load that you think you'll experience, as well as a safety margin to permit logging in and administering the server.
        3) Beware also of surprises that might make you hit the limit of connections. For example, if you restart an application server, it might not close its connections cleanly, and MySQL might not realize they've been closed.
        4) Watch the Max_used_connections status variable over time. It is a high-water mark that shows you if the server has had a spike in connections at some point.
    iii. **thread_cache_size**
        1) Watch the Threads_connected status variable and find its typical maximum and minimum.
        2) A related status variable is Slow_launch_threads. A large value for this status variable means that something is delaying new threads upon connection.
    iv. **table_cache_size**
        1) This cache (or the two caches into which it was split in MySQL 5.1) should be set large enough to keep from reopening and reparsing table definitions all the time.
        2) You can check this by inspecting the value of **open_tables**. If you see many Opened_tables per second, your table_cache value might not be large enough.
        3) Even if the table cache is useful, you should not set this variable too large. It turns out that the table cache can be counterproductive in two circumstances.
            a) First, MySQL doesn't use a very efficient algorithm to check the cache, so if it's really big, it can get really slow. You probably shouldn't set it higher than 10,000 in most cases, or 10,240.
            b) The second reason to avoid setting this very large is that some workloads simply aren't cacheable.

7. Replication

a. Replication lets you configure one or more servers as replicas1 of another server, keeping their data synchronized with the master copy. This is not just useful for high-performance applications—it is also the cornerstone of many strategies for high availability, scalability, disaster recovery, backups, analysis, data warehousing, and many other tasks. In fact, calability and high availability are related topics.
b. The basic problem replication solves is keeping one server's data synchronized with another's.
c. MySQL supports two kinds of replication: statement-based replication and row-based replication.
    i. Statement-based Replication
        1) MySQL 5.0 and earlier support only statement-based replication (also called logical replication)
        2) The most obvious benefit is that it's fairly simple to implement.
        3) Another benefit of statement-based replication is that the binary log events tend to be reasonably compact. So, relatively speaking, statement-based replication doesn't use a lot of bandwidth.
        4) MySQL's binary log format includes more than just the query text; it also transmits several bits of metadata, such as the current timestamp.
        5) Even so, there are some statements that MySQL can¡¯t replicate correctly, such as queries that use the CURRENT_USER() function.
        6) Stored routines and triggers are also problematic with statement-based replication.
        7) Another issue with statement-based replication is that the modifications must be serializable. This requires more locking—sometimes significantly more.
        8) Not all storage engines work with statement-based replication, although those provided with the official MySQL server distribution up to and including MySQL 5.5 do.
    ii. Row-based Replication
        1) MySQL 5.1 added support for row-based replication, which records the actual data changes in the binary log and is more similar to how most other database products implement replication.
        2) The biggest advantages are that MySQL can replicate every statement correctly.
        3) Some statements can be replicated much more efficiently. Because the replica doesn't have to replay the queries that changed the rows on the master.
        4) On the other hand, the following event is much cheaper to replicate with statementbased replication:
        ```
        mysql> UPDATE enormous_table SET col1 = 0;
        ```
    iii. Because neither format is perfect for every situation, MySQL can switch between statement-based and row-based replication dynamically. By default, it uses statementbased replication, but when it detects an event that cannot be replicated correctly with a statement, it switches to row-based replication. You can also control the format as needed by setting the binlog_format session variable.
d. MySQL's replication is mostly backward-compatible.
e. Replication is relatively good for scaling reads, which you can direct to a replica, but it's not a good way to scale writes unless you design it right.
f. Replication is also wasteful with more than a few replicas, because it essentially duplicates a lot of data needlessly.
g. Problems Solved by Replication
    i. Data distribution
    ii. Load balancing
    iii. Backups
    iv. High availability and failover
    v. Testing MySQL upgrades: It's common practice to set up a replica with an upgraded MySQL version and use it to ensure that your queries work as expected, before upgrading every instance.
h. How Replication Works
    i. The master records changes to its data in its binary log. (These records are called binary log events.)
    ii. The replica copies the master's binary log events to its relay log.
    iii. The replica replays the events in the relay log, applying the changes to its own data.





i. Setting Up Replication

i. At a high level, the process is as follows:

1) Set up replication accounts on each server.

```
mysql> GRANT REPLICATION SLAVE, REPLICATION CLIENT ON *.*
    -> TO repl@'192.168.0.%' IDENTIFIED BY 'p4ssword',;
```

2) Configure the master and replica.

a) You need to enable binary logging and specify a server ID. Enter (or verify the presence of) the following lines in the master's my.cnf file:

```
log_bin = mysql-bin
server_id = 10
```

b) The replica requires a configuration in its my.cnf file similar to the master, and you'll also need to restart MySQL on the replica:

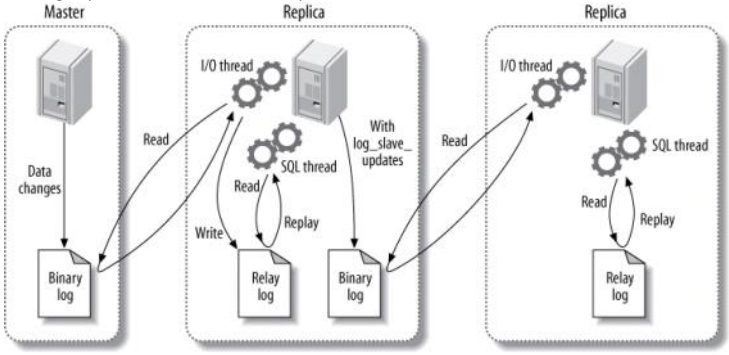
```
log_bin = mysql-bin
server_id = 2
relay_log = /var/lib/mysql/mysql-relay-bin
log_slave_updates = 1
read_only = 1
```

3) Instruct the replica to connect to and replicate from the master.

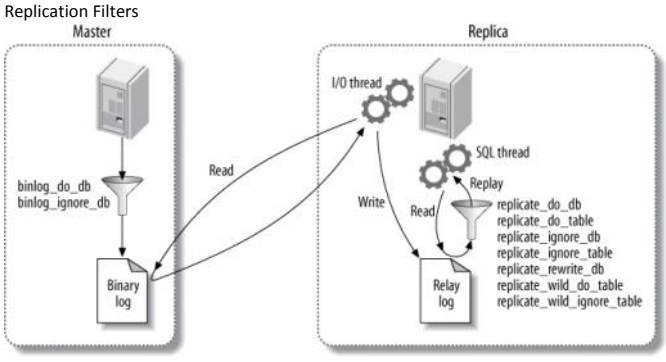
a) Use the CHANGE MASTER TO statement to start replica

```
mysql> CHANGE MASTER TO MASTER_HOST='server1',
    -> MASTER_USER='repl',
    -> MASTER_PASSWORD='p4ssword',
    -> MASTER_LOG_FILE='mysql-bin.000001',
    -> MASTER_LOG_POS=0;
```

j. Sending Replication Events to Other Replicas



k. Replication Filters



8. Backup and Recovery (Details to see the e-book)

9. MySQL Server Status

a. SHOW VARIABLES

i. **MySQL exposes many system variables through the SHOW VARIABLES SQL command**, as variables you can use in expressions

ii. **Or with *mysqladmin* variables at the command line.**

iii. From MySQL 5.1, you can also access them through tables in the INFORMATION_SCHEMA database. Refer to https://dev.mysql.com/doc/refman/5.0/en/information-schema.html

iv. These variables represent a variety of configuration information, such as the server's default storage engine (storage_engine), the available time zones, the connection's collation, and startup parameters.

b. SHOW STATUS

i. The SHOW STATUS command shows server status variables in a two-column name-value table. Unlike the server variables we mentioned in the previous section, these are readonly.

ii. **You can view the variables by either executing SHOW STATUS as a SQL command**.

iii. Or executing *mysqladmin extended-status* as a shell command.

iv. If you use the SQL command, you can use LIKE and WHERE to limit the results; the LIKE does a standard pattern match on the variable name.

v. The commands return a table of results, but you can't sort it, join it to other tables, or do other standard things you can do with MySQL tables.
vi. In MySQL 5.1 and newer, you can select values directly from the INFORMATION_SCHEMA.GLOBAL_STATUS and INFORMATION_SCHEMA.SESSION_STATUS tables.
vii. SHOW STATUS contains a mixture of global and session variables. Many of them have dual scope: there's both a global and a session variable, and they have the same name.
viii. **SHOW STATUS also now shows session variables by default**, so if you were accustomed to running SHOW STATUS and seeing global variables, you won't see them anymore; **now you have to run SHOW GLOBAL STATUS instead**
ix. There are hundreds of status variables. Most either are counters or contain the current value of some status metric.
x. Sometimes there are several variables that seem to refer to the same thing, such as Connections (the number of connection attempts to the server) and Threads_connected; in this case, the variables are related, but similar names don't always imply a relationship.
xi. Some Important Status (Details to see the e-book)
1) Thread and Connection Statistics
2) Binary Logging Status
3) Command Counters: The Com_* variables count the number of times each type of SQL or C API command has been issued.
4) Temporary Files and Tables
5) Handler Operations: The handler API is the interface between MySQL and its storage engines.
6) MyISAM Key Buffer: The Key_* variables contain metrics and counters about the MyISAM key buffer.
7) File Descriptors
8) Query Cache: You can inspect the query cache by looking at the Qcache_* status variables
9) SELECT Types: The Select_* variables are counters for certain types of SELECT queries. They can help you see the ratio of SELECT queries that use various query plans.
10) Sorts
11) Table Locking: The Table_locks_immediate and Table_locks_waited variables tell you how many locks were granted immediately and how many had to be waited for.
12) InnoDB-Specific: The Innodb_* variables show some of the data included in SHOW ENGINE INNODB STATUS
13) Plugin-Specific

c. SHOW PROCESSLIST

i. The process list is the list of connections, or threads, that are currently connected to MySQL. SHOW PROCESSLIST lists the threads, with information about each thread's status.

10. Using EXPLAIN

a. When it executes the query, the flag causes it to return information about each step in the execution plan, instead of executing it. It returns one or more rows, which show each part of the execution plan and the order of execution.
b. There's one row in the output per table in the query. If the query joins two tables, there will be two rows of output. An aliased table counts as a separate table, so if you join a table to itself, there will be two rows in the output.
c. MySQL 5.6 added support for EXPLAIN INSERT/UPDATE/DELETE.
d. There are two important variations on EXPLAIN:
i. EXPLAIN EXTENDED appears to behave just like a normal EXPLAIN, but it tells the server to "reverse compile" the execution plan into a SELECT statement. You can see this generated statement by running SHOW WARNINGS immediately afterward. The statement comes directly from the execution plan, not from the original SQL statement, which by this point has been reduced to a data structure. It will not be the same as the original statement in most cases. You can examine it to see exactly how the query optimizer has transformed the statement. EXPLAIN EXTENDED is available in MySQL 5.0 and newer, and it adds an extra filtered column in MySQL 5.1.
ii. EXPLAIN PARTITIONS shows the partitions the query will access, if applicable. It is available only in MySQL 5.1 and newer.
e. It's a common mistake to think that MySQL doesn't execute a query when you add EXPLAIN to it. In fact, if the query contains a subquery in the FROM clause, MySQL actually executes the subquery, places its results into a temporary table, and then finishes optimizing the outer query. It has to process all such subqueries before it can optimize the outer query fully, which it must do for EXPLAIN.1 This means EXPLAIN can actually cause a great deal of work for the server if the statement contains expensive subqueries or views that use the TEMPTABLE algorithm.
f. EXPLAIN is an approximation, nothing more.
g. Here are some of its limitations:
i. EXPLAIN doesn't tell you anything about how triggers, stored functions, or UDFs will affect your query.
ii. It doesn't work for stored procedures, although you can extract the queries manually and EXPLAIN them individually.
iii. It doesn't tell you about ad hoc optimizations MySQL does during query execution.
iv. Some of the statistics it shows are estimates and can be very inaccurate.
v. It doesn't show you everything there is to know about a query's execution plan.
vi. It doesn't distinguish between some things with the same name. For example, it uses "filesort" for in-memory sorts and for temporary files, and it displays "Using temporary" for temporary tables on disk and in memory.

h. The Columns in EXPLAIN

i. The id Column
1) This column always contains a number, which identifies the SELECT to which the row belongs.
2) If there are no subqueries or unions in the statement, there is only one SELECT, so every row will show a 1 in this column. Otherwise, the inner SELECT statements generally will be numbered sequentially, according to their positions in the original statement.
ii. The select_type Column
1) MySQL divides SELECT queries into simple and complex types, and the complex types can be grouped into three broad classes: simple subqueries, so-called derived tables (subqueries in the FROM clause), and UNIONs.
2) This column shows whether the row is a simple or complex SELECT (and if it's the latter, which of the three complex types it is).
3) The value SIMPLE means the query contains no subqueries or UNIONs.
4) If the query has any such complex subparts, the outermost part is labeled PRIMARY, and other parts are labeled as follows:
a) SUBQUERY: A SELECT that is contained in a subquery in the SELECT list (in other words, not in the FROM clause) is labeled as SUBQUERY.
b) DERIVED: The value DERIVED is used for a SELECT that is contained in a subquery in the FROM clause, which MySQL executes recursively and places into a temporary table. The server refers to this as a "derived table" internally, because the temporary table is derived from the subquery.
c) UNION: The second and subsequent SELECTs in a UNION are labeled as UNION. The first SELECT is labeled as though it is executed as part of the outer query. This is why the previous example showed the first SELECT in the UNION as PRIMARY. If the UNION were contained in a subquery in the FROM clause, its first SELECT would be labeled as DERIVED.
d) UNION RESULT: The SELECT used to retrieve results from the UNION's anonymous temporary table is labeled as UNION RESULT.

In addition to these values, a SUBQUERY and a UNION can be labeled as DEPENDENT and UNCACHEABLE. DEPENDENT means the SELECT depends on data that is found in an outer query; UNCACHEABLE means something in the SELECT prevents the results from being cached with an Item_cache.

iii. The table Column
1) This column shows which table the row is accessing. In most cases, it's straightforward: it's the table, or its alias if the SQL specifies one.
2) You can read this column from top to bottom to see the join order MySQL's join optimizer chose for the query.
3) When there's a subquery in the FROM clause, the table column is of the form <derivedN>, where N is the subquery's id.
4) When there's a UNION, the UNION RESULT table column contains a list of ids that participate in the UNION. This is always a "backward reference," because the UNION RESULT comes after all of the rows that participate in the UNION. If there are more than about 20 ids in the list, the table column might be truncated to keep it from getting too long, and you won't be able to see all the values.
iv. The type Column
1) The MySQL manual says this column shows the "join type," but we think it's more accurate to say the access type—in other words, how MySQL has decided to find rows in the table. Here are the most important access methods, from worst to best:
a) ALL: This is what most people call a table scan.
b) Index: This is the same as a table scan, except MySQL scans the table in index order instead of the rows. The main advantage is that this avoids sorting; the biggest disadvantage is the cost of reading an entire table in index order. If you also see "Using index" in the Extra column, it mea ns MySQL is using a covering index and scanning only the index's data, not reading each row in index order. This is much less expensive than scanning the table in index order.
c) Range: A range scan is a limited index scan. It begins at some point in the index and returns rows that match a range of values. This is better than a full index scan because it doesn't go through the entire index. Obvious range scans are queries with a BETWEEN or > in the WHERE clause. When MySQL uses an index to look up lists of values, such as IN() and OR lists, it also displays it as a range scan. However, these are quite different types of accesses, and they have important performance differences.
d) Ref: It's called ref because the index is compared to some reference value. The reference value is either a constant or a value from a previous table in a multiple-table query. The ref_or_null access type is a variation on ref. It means MySQL must do a second lookup to find NULL entries after doing the initial lookup.

e) Eq_ref: This is an index lookup that MySQL knows will return at most a single value. You'll see this access method when MySQL decides to use a primary key or unique index to satisfy the query by comparing it to some reference value.
f) const, system: MySQL uses these access types when it can optimize away some part of the query and turn it into a constant.
g) NULL: This access method means MySQL can resolve the query during the optimization phase and will not even access the table or index during the execution stage. For example, selecting the minimum value from an indexed column can be done by looking at the index alone and requires no table access during execution.

v. The possible_keys Column
1) This column shows which indexes could be used for the query, based on the columns the query accesses and the comparison operators used. This list is created early in the optimization phase, so some of the indexes listed might be useless for the query after subsequent optimization phases.

vi. The key Column
1) This column shows which index MySQL decided to use to optimize the access to the table.
2) In other words, possible_keys reveals which indexes can help make row lookups efficient, but key shows which index the optimizer decided to use to minimize query cost

vii. The key_len Column
1) This column shows the number of bytes MySQL will use in the index.

viii. The ref Column
1) This column shows which columns or constants from preceding tables are being used to look up values in the index named in the key column.

ix. The rows Column
1) This column shows the number of rows MySQL estimates it will need to read to find the desired rows. This number is per loop in the nested-loop join plan.
2) That is, it's not just the number of rows MySQL thinks it will need to read from the table; it is the number of rows, on average, MySQL thinks it will have to read to find rows that satisfy the criteria in effect at that point in query execution.
3) This estimate can be quite inaccurate, depending on the table statistics and how selective the indexes are. It also doesn't reflect LIMIT clauses in MySQL 5.0 and earlier.
4) Remember, this is the number of rows MySQL thinks it will examine, not the number of rows in the result set.

x. The filtered Column
1) This column is new in MySQL 5.1 and appears when you use EXPLAIN EXTENDED.
2) It shows a pessimistic estimate of the percentage of rows that will satisfy some condition on the table, such as a WHERE clause or a join condition. If you multiply the rows column by this percentage, you will see the number of rows MySQL estimates it will join with the previous tables in the query plan.

xi. The Extra Column
1) This column contains extra information that doesn't fit into other columns.
2) The most important values you might see frequently are as follows:
a) "Using index": This indicates that MySQL will use a covering index to avoid accessing the table. Don't confuse covering indexes with the index access type.
b) "Using where": This means the MySQL server will post-filter rows after the storage engine retrieves them. Many WHERE conditions that involve columns in an index can be checked by he storage engine when (and if) it reads the index, so not all queries with a WHERE clause will show "Using where." Sometimes the presence of "Using where" is a hint that the query can benefit from different indexing.
c) "Using temporary": This means MySQL will use a temporary table while sorting the query's result.
d) "Using filesort": This means MySQL will use an external sort to order the results, instead of reading the rows from the table in index order. MySQL has two filesort algorithms, which you can read about in Chapter 6. Either type can be done in memory or on disk. EXPLAIN doesn't tell you which type of filesort MySQL will use, and it doesn't tell you whether the sort will be done in memory or on disk.
e) "Range checked for each record (index map: N)": This value means there's no good index, and the indexes will be reevaluated for each row in a join. N is a bitmap of the indexes shown in possible_keys and is redundant.

11. MySQL Data Fragmentation - What, When and How

MySQL tables, including MyISAM and InnoDB, two of the most common types, experience fragmentation as data is inserted and deleted randomly. Fragmentation can leave large holes in your table, blocks which must be read when scanning the table. Optimizing your table can therefore make full table scans and range scans more efficient.

a. Fragmentation - an example

MySQL has quite a few different storage engines to store data in tables. Whenever MySQL deletes rows from your table, the space left behind is then empty. Over time with a lot of DELETEs, this space can grow larger than the used space in your table. When MySQL goes to scan that data, it scans to the high water mark of the table, that is the highest point at which data has been added. If new inserts occur, MySQL will try to use that space, but nevertheless gaps will persist. This extra fragmented space can make reads against the table less efficient than they might otherwise be. Let's look at an example.

We'll create a database (sometimes called a schema) and a test table:

```
(root@localhost) [test]> create database frag_test;
Query OK, 1 row affected (0.03 sec)

(root@localhost) [test]> use frag_test;
Database changed

(root@localhost) [frag_test]> create table frag_test (c1 varchar(64));
Query OK, 0 rows affected (0.05 sec)
```

Next let's add some rows to the table:

```
(root@localhost) [frag_test]> insert into frag_test values ('this is row 1');
Query OK, 1 row affected (0.01 sec)

(root@localhost) [frag_test]> insert into frag_test values ('this is row 2');
Query OK, 1 row affected (0.00 sec)

(root@localhost) [frag_test]> insert into frag_test values ('this is row 3');
Query OK, 1 row affected (0.00 sec)
```

Now we'll check for fragmentation:

```
(root@localhost) [frag_test]> show table status from frag_test\G;
*************************** 1. row ***************************
           Name: frag_test
         Engine: MyISAM
        Version: 10
     Row_format: Dynamic
           Rows: 3
 Avg_row_length: 20
    Data_length: 60
Max_data_length: 281474976710655
   Index_length: 1024
      Data_free: 0
 Auto_increment: NULL
    Create_time: 2011-02-23 14:55:27
    Update_time: 2011-02-23 15:06:55
     Check_time: NULL
      Collation: latin1_swedish_ci
```

```
      Checksum: NULL
 Create_options:
       Comment:
1 row in set (0.00 sec)
```

Now let's delete a row and check again:

```
 (root@localhost) [frag_test]> delete from frag_test where c1 = 'this is row 2';
Query OK, 1 row affected (0.00 sec)

(root@localhost) [frag_test]> show table status from frag_test\G;
*************************** 1. row ***************************
         Name: frag_test
       Engine: MyISAM
      Version: 10
   Row_format: Dynamic
         Rows: 2
 Avg_row_length: 20
   Data_length: 60
Max_data_length: 281474976710655
  Index_length: 1024
     Data_free: 20
 Auto_increment: NULL
    Create_time: 2011-02-23 14:55:27
    Update_time: 2011-02-23 15:07:49
     Check_time: NULL
      Collation: latin1_swedish_ci
       Checksum: NULL
 Create_options:
       Comment:
1 row in set (0.00 sec)
```

Notice the "data_free" column shows the space left by the second row that we deleted. Imagine you had 20,000 rows. They wouldtake 400k bytes of space. Now if you deleted 19,999 rows, there are 20bytes of useful space in the table, but MySQL will still read 400k and data_free will show 39980.

b. Eliminating fragmentation
Luckily MySQL comes with a simple way to fix this once you've identified it. It's called optimize table. Take a look:

```
(root@localhost) [frag_test]> optimize table frag_test;
+---------------------------+-------------+--------------+--------------+
| Table                     | Op          | Msg_type | Msg_text |
+---------------------------+-------------+--------------+--------------+
| frag_test.frag_test | optimize | status       | OK           |
+---------------------------+-------------+--------------+--------------+
1 row in set (0.00 sec)

(root@localhost) [frag_test]> show table status from frag_test\G;
*************************** 1. row ***************************
         Name: frag_test
       Engine: MyISAM
      Version: 10
   Row_format: Dynamic
         Rows: 2
 Avg_row_length: 20
   Data_length: 40
Max_data_length: 281474976710655
  Index_length: 1024
     Data_free: 0
 Auto_increment: NULL
    Create_time: 2011-02-23 14:55:27
    Update_time: 2011-02-23 15:11:05
     Check_time: 2011-02-23 15:11:05
      Collation: latin1_swedish_ci
       Checksum: NULL
 Create_options:
       Comment:
1 row in set (0.00 sec)
```

c. Performance considerations
OPTIMIZE TABLE will lock the entire table while doing its work. For small tables this will work fine because the whole tablecan be read and written quickly. For very large tables, this can take a very long time and interrupt or reduce available to your application. What to do?

Luckily MySQL has a great feature called Master-Master replication. In this configuration, your backend database is actually two separate databases, one active and one passive. Both of the databases are identical in every way. To perform the operations online - including the OPTIMIZE TABLE operation, simply run them on your PASSIVE database. This will not interrupt your application one iota. Once the operation has completed, switch roles so that your application is pointing to your secondary database and make it active. Then make or original database the passive one.

Now that the roles are switched and the application is happily pointing at db2, perform the same OPTIMIZE TABLE operation ondb1. It's now the passive database, so it won't interrupt the primary either.

d. Other commands
Show all the fragmented tables in your database:

```
(root@localhost) [(none)]>
select table_schema, table_name, data_free, engine
from information_schema.tables where table_schema
not in ('information_schema', 'mysql') and data_free > 0;
```

You can also cause a table to be defragmented if you change the storage engine. That's because MySQL has to read the entire table, and write it back to disk using the new storage engine, and in the process the rows and data gets compressed down more efficiently.

Here's what that looks like:

```
(root@localhost) [frag_test]> alter table frag_test engine = innodb;
Query OK, 2 rows affected (0.17 sec)
Records: 2  Duplicates: 0  Warnings: 0

(root@localhost) [frag_test]> show table status from frag_test
    -> \G;
*************************** 1. row ***************************
           Name: frag_test
         Engine: InnoDB
        Version: 10
     Row_format: Compact
           Rows: 2
 Avg_row_length: 8192
    Data_length: 16384
Max_data_length: 0
   Index_length: 0
      Data_free: 0
 Auto_increment: NULL
    Create_time: 2011-02-23 15:41:12
    Update_time: NULL
     Check_time: NULL
      Collation: latin1_swedish_ci
       Checksum: NULL
 Create_options:
        Comment: InnoDB free: 7168 kB
1 row in set (0.00 sec)
```