2020年7月28日 11:02

一. Sql性能调优

a. 通过explain（优化查询器）来查询SQL语句执行结果，可以帮助选择更好的索引和优化查询语句

i. 字段：

项	说明
id	MySQL Query Optimizer 选定的执行计划中查询的序列号。表示查询中执行 select 子句或操作表的顺序,id 值越大优先级越高,越先被执行。id 相同,执行顺序由上至下。

select_type 查询类型	说明
SIMPLE	简单的 select 查询,不使用 union 及子查询
PRIMARY	最外层的 select 查询
UNION	UNION 中的第二个或随后的 select 查询,不 依赖于外部查询的结果集
DEPENDENT UNION	UNION 中的第二个或随后的 select 查询,依 赖于外部查询的结果集
SUBQUERY	子查询中的第一个 select 查询,不依赖于外 部查询的结果集
DEPENDENT SUBQUERY	子查询中的第一个 select 查询,依赖于外部 查询的结果集
DERIVED	用于 from 子句里有子查询的情况。 MySQL 会 递归执行这些子查询, 把结果放在临时表里。
UNCACHEABLE SUBQUERY	结果集不能被缓存的子查询,必须重新为外 层查询的每一行进行评估。
UNCACHEABLE UNION	UNION 中的第二个或随后的 select 查询,属 于不可缓存的子查询

项	说明
table	输出行所引用的表

type 重要的项,显示连接使用的类型,按最 优到最差的类型排序	说明
system	表仅有一行(=系统表)。这是 const 连接类型的一个特例。
const	const 用于用常数值比较 PRIMARY KEY 时。当 查询的表仅有一行时,使用 System。
eq_ref	const 用于用常数值比较 PRIMARY KEY 时。当 查询的表仅有一行时,使用 System。
ref	连接不能基于关键字选择单个行,可能查找 到多个符合条件的行。 叫做 ref 是因为索引要跟某个参考值相比较。这个参考值或者是一 个常数,或者是来自一个表里的多表查询的 结果值。
ref_or_null	如同 ref, 但是 MySQL 必须在初次查找的结果 里找出 null 条目,然后进行二次查找。
index_merge	说明索引合并优化被使用了。
unique_subquery	在某些 IN 查询中使用此种类型,而不是常规的 ref:value IN (SELECT primary_key FROM single_table WHERE some_expr)
index_subquery	在某些 IN 查询中使用此种类型，与 unique_subquery 类似,但是查询的是非唯一 性索引: value IN (SELECT key_column FROM single_table WHERE some_expr)
range	只检索给定范围的行,使用一个索引来选择 行。key 列显示使用了哪个索引。当使用=、<>、>、>=、<、<=、IS NULL、<=>、BETWEEN 或者 IN 操作符,用常量比较关键字列时,可以使用 range。
index	全表扫描,只是扫描表的时候按照索引次序进行而不是行。主要优点就是避免了排序, 但是开销仍然非常大。
all	最坏的情况,从头到尾全表扫描。

项	说明
possible_keys	指出 MySQL 能在该表中使用哪些索引有助于查询。如果为空,说明没有可用的索引。

项	说明
key	MySQL 实际从 possible_key 选择使用的索引。 如果为 NULL,则没有使用索引。很少的情况 下,MYSQL 会选择优化不足的索引。这种情 况下,可以在 SELECT 语句中使用 USE INDEX (indexname)来强制使用一个索引或者用 IGNORE INDEX(indexname)来强制 MYSQL 忽略索引

项	说明
key_len	使用的索引的长度。在不损失精确性的情况 下,长度越短越好。

项	说明
ref	显示索引的哪一列被使用了

项	说明
rows	MYSQL 认为必须检查的用来返回请求数据的行数

extra 中出现以下 2 项意味着 MYSQL 根本不能使用索引,效率会受到重大影响。应尽可能对此进行优化。

extra 项	说明
Using filesort	表示 MySQL 会对结果使用一个外部索引排序,而不是从表里按索引次序读到相关内容。可能在内存或者磁盘上进行排序。MySQL 中无法利用索引完成的排序操作称为“文件排序”
Using temporary	表示 MySQL 在对查询结果排序时使用临时表。常见于排序 order by 和分组查询 group by。

ii. 注意：

i. 任何地方尽量不要使用*，可以用具体字段代替 * ，也不要返回无用的字段；

ii. 不要写一些没有意义的查询，如需要生成一个空表结构：

```
select col1,col2 into #t from t where 1=0
```

这类代码不会返回任何结果集，但是会消耗系统资源的，应改成这样：

```
create table #t(…)
select count(*) from table;
```

这样不带任何条件的count会引起全表扫描，并且没有任何业务意义，是一定要杜绝的。

iii. Update 语句，如果只更改1、2个字段，不要Update全部字段，否则频繁调用会引起明显的性能消耗，同时带来大量日志。

iv. 对于多张大数据量（这里几百条就算大了）的表JOIN，要先分页再JOIN，否则逻辑读会很高，性能很差。

v. 尽可能的使用 varchar/nvarchar 代替 char/nchar ，因为首先变长字段存储空间小，可以节省存储空间，其次对于查询来说，在一个相对较小的字段内搜索效率显然要高些。

vi. 尽量使用表变量来代替临时表。如果表变量包含大量数据，请注意索引非常有限（只有主键索引）。

vii. 避免频繁创建和删除临时表，以减少系统表资源的消耗。临时表并不是不可使用，适当地使用它们可以使某些例程更有效，例如，当需要重复引用大型表或常用表中的某个数据集时。但是，对于一次性事件， 最好使用导出表。

viii. 在新建临时表时，如果一次性插入数据量很大，那么可以使用 select into 代替 create table，避免造成大量 log ，以提高速度；如果数据量不大，为了缓和系统表的资源，应先create table，然后insert。

ix. 如果使用到了临时表，在存储过程的最后务必将所有的临时表显式删除，先 truncate table ，然后 drop table ，这样可以避免系统表的较长时间锁定。

x. 尽量避免使用游标，因为游标的效率较差，如果游标操作的数据超过1万行，那么就应该考虑改写。

xi. 与临时表一样，游标并不是不可使用。对小型数据集使用 FAST_FORWARD 游标通常要优于其他逐行处理方法，尤其是在必须引用几个表才能获得所需的数据时。在结果集中包括“合计”的例程通常要比使用游标执行的速度快。如果开发时 间允许，基于游标的方法和基于集的方法都可以尝试一下，看哪一种方法的效果更好。

xii. 在所有的存储过程和触发器的开始处设置 SET NOCOUNT ON ，在结束时设置 SET NOCOUNT OFF 。无需在执行存储过程和触发器的每个语句后向客户端发送 DONE_IN_PROC 消息。

尽量避免大事务操作，提高系统并发能力。

尽量避免向客户端返回大数据量，若数据量过大，应该考虑相应需求是否合理。

b. 创建并且正确地使用索引

i. 首先以mysql为例：有B-TREE和hash两种，但是hash对于精准的查询可以快速定位，但是模糊查询或者查某一范围内不适用。

1) B-TREE索引包括很多扩展类型，如组合索引、反向索引、函数索引等等，以下是B-TREE索引的简单介绍：

i. B-TREE索引也称为平衡树索引(Balance Tree)，它是一种按字段排好序的树形目录结构，主要用于提升查询性能和唯一约束支持。

2) B-TREE索引的内容包括根节点、分支节点、叶子节点。

i. 叶子节点内容：索引字段内容+表记录ROWID

ii. 根节点，分支节点内容：当一个数据块中不能放下所有索引字段数据时，就会形成树形的根节点或分支节点，根节点与分支节点保存了索引树的顺序及各层级间的引用关系。

ii. 索引字段应该满足的条件

1) 字段出现在查询条件中，并且查询条件可以使用索引

2) 语句执行频率高

3) 需要创建索引的字段：

i. 主键/外键

ii. 有身份意义标识的字段

4) 慎用索引的字段

i. 日期

ii. 状态标识（订单状态等）

iii. 类型（订单类型，图片类型）

iv. 区域（国家，城市）

v. 操作人员（creator，auditor）

vi. 数值（level，score）

vii. 长字符（address，sub）

5) 不适合索引

i. 描述备注（detal，remark）

ii. 大字段

iii. 注意事项：

1) 一般创建索引是在外键或者主键上，但还是要具体问题具体分析。

2) 优先选择整数作为索引尽量不用字符串，原因是生成树时需要进行大小比较，整数的比较比字符串快很多。

3) 不在索引上做运算或者使用函数，这回使数据库放弃索引，全表扫描，对查询进行优化，应尽量避免全表扫描，首先应考

虑在 where 及 order by 涉及的列上建立索引。

4) 应尽量避免在 where 子句中对字段进行 null 值判断，否则将导致引擎放弃使用索引而进行全表扫描，如：

```
select id from t where num is null
```

最好不要给数据库留NULL，尽可能的使用 NOT NULL填充数据库.

5) 应尽量避免在 where 子句中使用 != 或 <> 操作符，否则将引擎放弃使用索引而进行全表扫描。

6) 应尽量避免在 where 子句中使用 or 来连接条件，如果一个字段有索引，一个字段没有索引，将导致引擎放弃使用索引而进行全表扫描，如：

```
select id from t where num=10 or Name = 'admin'
```

可以这样查询：

```
select id from t where num = 10
union all
select id from t where Name = 'admin'
```

7) in 和 not in 也要慎用，否则会导致全表扫描，如：

```
select id from t where num in(1,2,3)
```

对于连续的数值，能用 between 就不要用 in 了：

```
select id from t where num between 1 and 3
```

很多时候用 exists 代替 in 是一个好的选择：

```
select num from a where num in(select num from b)
```

用下面的语句替换：

```
select num from a where exists(select 1 from b where num=a.num)
```

8) 下面的查询也将导致全表扫描：

```
select id from t where name like ‘%abc%’
```

若要提高效率，可以考虑全文检索（非结构化数据查询方式；很像字典；虽然建立全文索引很耗时，但是可以多次使用很划算）。

9) 如果在 where 子句中使用参数，也会导致全表扫描。因为SQL只有在运行时才会解析局部变量，但优化程序不能将访问计划的选择推迟到运行时；它必须在编译时进行选择。然 而，如果在编译时建立访问计划，变量的值还是未知的，因而无法作为索引选择的输入项。如下面语句将进行全表扫描：

```
select id from t where num = @num
```

可以改为强制查询使用索引：

```
select id from t with(index(索引名)) where num = @num
```

10) 应尽量避免在 where 子句中对字段进行表达式操作，这将导致引擎放弃使用索引而进行全表扫描。如：

```
select id from t where num/2 = 100
```

应改为：

```
select id from t where num = 100*2
```

11) 应尽量避免在where子句中对字段进行函数操作，这将导致引擎放弃使用索引而进行全表扫描。如：

```
select id from t where substring(name,1,3) = ’ abc’

select id from t where datediff(day,createdate,’ 2005-11-30′ ) = 0
```

应改为：

```
select id from t where name like 'abc%'

select id from t where createdate >= '2005-11-30' and createdate < '2005-12-1'
```

12) 不要在 where 子句中的“=”左边进行函数、算术运算或其他表达式运算，否则系统将可能无法正确使用索引。

13) 在使用索引字段作为条件时，如果该索引是复合索引，那么必须使用到该索引中的第一个字段作为条件时才能保证系统使用该索引，否则该索引将不会被使用，并且应尽可能的让字段顺序与索引顺序相一致。

14) 索引并不是越多越好，索引固然可以提高相应的 select 的效率，但同时也降低了 insert 及 update 的效率，因为 insert 或 update 时有可能会重建索引，所以怎样建索引需要慎重考虑，视具体情况而定。一个表的索引数最好不要超过6个，若太多则应考虑一些不常使用到的列上建的索引是否有必要。

c. 返回较少的数据

i. 分页技术

1) 避免select 字段名 from 表名，一次将表中的数据全部查出，浪费时间，资源。

ii. 返回只需要的字段

1) 优点：

1、减少数据在网络上传输开销

2、减少服务器数据处理开销

3、减少客户端内存占用

4、字段变更时提前发现问题，减少程序BUG

5、如果访问的所有字段刚好在一个索引里面，则可以使用纯索引访问提高性能。

2) 缺点：增加编码工作量

由于会增加一些编码工作量，所以一般需求通过开发规范来要求程序员这么做，否则等项目上线后再整改工作量更大。

3) 如果你的查询表中有大字段或内容较多的字段，如备注信息、文件内容等等，那在查询表时一定要注意这方面的问题，否则可能会带来严重的性能问题。

4) 如果表经常要查询并且请求大内容字段的概率很低，我们可以采用分表处理，将一个大表分拆成两个一对一的关系表，将不常用的大内容字段放在一张单独的表中。

d. 优化数据库对象

i. 优化数据库对象的类型，可以使用procedure analyse（）函数，对表进行分析，该函数对表中的列中的数据进行优化建议；

1) 类型原则：可以正确表示并且要使用存储数据最短类型。这样可以减少内存，磁盘，CPU缓存使用。

例：select 字段名 from 表名procedure analyse（）；

二. 其他方面优化

a. 硬件优化：

i. CPU优化，选择多核或者主频高的CPU

ii. 内存优化，使用更大内存，将尽可能多的内存分配给MySql做缓存；

iii. 磁盘I/O优化：使用RAID-0 1，（是 0 和 1 的结合，RAID-0是没有数据冗余，没有数据校验的磁盘阵列；RAID-1是两款磁盘所

构成的磁盘阵列，容积仅相当于一块，另一块作为镜像）有了数据安全保障和存储性能保障；

iv. 调整寻道算法：减少寻道时间

b. MySql自身优化：

i. 主要是对my.cof中各项参数进行优化调整；如可以指定MySql查询缓冲区的大小，可以调整MySql最大连接进程数；

c. 应用优化：

i. 使用数据库连接池

1) 磁盘 I/O 需要消耗的时间很多，而在内存中进行操作，效率则会高很多，为了能让数据表或者索引中的数据随时被我们所用，DBMS 会申请占用内存来作为数据缓冲池，这样做的好处是可以让磁盘活动最小化，从而减少与磁盘直接进行 I/O 的时间。要知道，这种策略对提升 SQL 语句的查询性能来说至关重要。如果索引的数据在缓冲池里，那么访问的成本就会降低很多。

i.

```
mysql> show variables like 'innodb_buffer_pool_size';
+-------------------------+---------+
| Variable_name           | Value   |
+-------------------------+---------+
| innodb_buffer_pool_size | 8388608 |
+-------------------------+---------+
1 row in set, 1 warning (0.01 sec)
```

i) 8388608/1024/1024=8MB

ii.

```
mysql> show variables like 'innodb_buffer_pool_instances';
+------------------------------+-------+
| Variable_name                | Value |
+------------------------------+-------+
| innodb_buffer_pool_instances | 1     |
+------------------------------+-------+
1 row in set, 1 warning (0.01 sec)
```

i) 只有一个缓冲池。实际上innodb_buffer_pool_instances默认情况下为 8，为什么只显示只有一个呢？这里需要说明的是，如果想要开启多个缓冲池，你首先需要将innodb_buffer_pool_size参数设置为大于等于 1GB，这时innodb_buffer_pool_instances才会大于 1。你可以在 MySQL 的配置文件中对innodb_buffer_pool_size进行设置，大于等于 1GB，然后再针对innodb_buffer_pool_instances参数进行修改。

ii. 使用查询缓存（适合不频繁更新的表）

三. 数据库性能监控

a. 学习了mysql中自带的performance_schema表

i. 配置使用

1) show variables like 'performance_schema';

2) instruments通俗讲就是监控项可以通过setup_instruments表设置需要开启对哪些项监控进行统计
consumers就是控制是否将监控到的结果进行记录

i. UPDATE setup_instruments SET ENABLED = 'YES', TIMED = 'YES';
UPDATE setup_consumers SET ENABLED = 'YES';

3) 如果不开启instruments和consumers则相应的事件检测则不能使用，也就不能收集相应的等待事件和性能的统计。
当然我们也可以单独设置某一个instruments的开启和关闭

i. UPDATE setup_instruments SET ENABLED = 'NO' WHERE NAME = 'wait/io/file/sql/binlog';

ii.

```
1  select * from setup_consumers;
```

信息 | 结果1 | 概况 | 状态

NAME	ENABLED
events_stages_current	YES
events_stages_history	YES
events_stages_history_lon	YES
events_statements_curren	YES
events_statements_histor	YES
events_statements_histor	YES
events_transactions_curre	YES
events_transactions_histo	YES
events_transactions_histo	YES
events_waits_current	YES
events_waits_history	YES
events_waits_history_long	YES
global_instrumentation	YES
thread_instrumentation	YES
statements_digest	YES

4) setup_consumer层级关系

global_instrumentation
thread_instrumentation
events_waits_current
events_waits_history
events_waits_history_long
events_stages_current
events_stages_history
events_stages_history_long
events_statements_current
events_statements_history
events_statements_history_long
statements_digest

```
1  select name,count(*) from setup_instruments group by left(name,5);
```

信息 | 结果1 | 概况 | 状态

5)

name	count(*)
idle	1
memory/performance_schema/mutex_instances	381
stage/sql/After create	129
statement/sql/select	193
transaction	1
wait/synch/mutex/sql/TC_LOG_MMAP::LOCK_tc	318

ii. 简单使用

1) 查询没有使用到索引或者索引效率低下的语句：

```
SELECT OBJECT_SCHEMA,  THREAD_ID TID, SUBSTR(SQL_TEXT, 1, 50) SQL_TEXT, ROWS_SENT RS, ROWS_EXAMINED
RE,CREATED_TMP_TABLES,NO_INDEX_USED,NO_GOOD_INDEX_USED FROM
performance_schema.events_statements_history WHERE (NO_INDEX_USED=1 OR NO_GOOD_INDEX_USED=1) and
sql_text  NOT LIKE '%performance_schema%'
```

2) 查看哪些索引没有被使用过

```
SELECT OBJECT_SCHEMA,OBJECT_NAME,INDEX_NAME
FROM table_io_waits_summary_by_index_usage
WHERE INDEX_NAME IS NOT NULL AND COUNT_STAR = 0 AND OBJECT_SCHEMA <> 'mysql'
ORDER BY OBJECT_SCHEMA,OBJECT_NAME;
```

3) 查看SQL语句在哪个阶段消耗最大

```
SELECT eshl.event_name, sql_text, eshl.timer_wait/1000000000000 w_s
FROM performance_schema.events_stages_history_long eshl
JOIN performance_schema.events_statements_history_long esthl
ON (eshl.nesting_event_id = esthl.event_id)
WHERE eshl.timer_wait > 1*10000000000
```

4) summary 表提供了整个时间段的一些统计信息。他们统计事件的处理方式和之前都不一样。
如果想知道某个instrument 被执行的最频繁，或者发生的频率非常高，

i.

```
mysql> SELECT EVENT_NAME, COUNT_STAR FROM events_waits_summary_global_by_event_name ORDER BY COUNT_STAR DESC LIMIT 10;
+---------------------------------------------+------------+
| EVENT_NAME                                  | COUNT_STAR |
+---------------------------------------------+------------+
| wait/io/file/sql/FRM                        |       1404 |
| wait/io/file/innodb/innodb_data_file        |        781 |
| wait/synch/mutex/innodb/buf_pool_mutex      |         81 |
| wait/synch/mutex/innodb/log_sys_mutex       |         79 |
| wait/synch/mutex/innodb/sync_array_mutex    |         54 |
| wait/synch/mutex/innodb/flush_list_mutex    |         53 |
| wait/io/file/myisam/kfile                   |         33 |
| wait/synch/mutex/innodb/lock_wait_mutex     |         27 |
| wait/synch/mutex/innodb/buf_dblwr_mutex     |         27 |
| wait/synch/mutex/innodb/dict_sys_mutex      |         26 |
+---------------------------------------------+------------+
```

b. 慢日志

i. MySQL的慢查询日志是MySQL提供的一种日志记录，它用来记录在MySQL中响应时间超过阀值的语句，具体指运行时间超过long_query_time值的SQL，则会被记录到慢查询日志中。long_query_time的默认值为10，意思是运行10s以上的语句。

ii. 默认情况下，MySQL数据库并不启动慢查询日志，需要我们手动来设置这个参数，当然，如果不是调优需要的话，一般不建议启动该参数，因为开启慢查询日志或多或少会带来一定的性能影响。慢查询日志支持将日志记录写入文件，也支持将日志记录写入数据库表。

iii. 产生慢日志的原因：

1) 执行时间过长（大于设置的long_query_time阀值）；

2) 第二未使用索引，或者未使用最优的索引。

iv. 设置日志

```
mysql> show variables like "%slow%";
+---------------------+-----------------------------------------------------+
| Variable_name       | Value                                               |
+---------------------+-----------------------------------------------------+
| log_slow_queries    | OFF                                                 |
| slow_launch_time    | 2                                                   |
| slow_query_log      | OFF                                                 |
| slow_query_log_file | D:\install\Mysql\ProgramData\Data\bells-PC-slow.log |
+---------------------+-----------------------------------------------------+
4 rows in set (0.06 sec)

mysql> show variables like "%long%";
+---------------------------------------------+-----------+
| Variable_name                               | Value     |
+---------------------------------------------+-----------+
| long_query_time                             | 10.000000 |
| max_long_data_size                          | 1048576   |
| performance_schema_events_waits_history_long_size | 10000     |
+---------------------------------------------+-----------+
3 rows in set (0.01 sec)
```

1)

v. 修改慢日志

1)

```
mysql> show variables like "%slow%";
+---------------------+-----------------------------+
| Variable_name       | Value                       |
+---------------------+-----------------------------+
| log_slow_queries    | ON                          |
| slow_launch_time    | 2                           |
| slow_query_log      | ON                          |
| slow_query_log_file | D:/temp/mysqlslowquery.log  |
+---------------------+-----------------------------+
4 rows in set (0.00 sec)

mysql> show variables like "%long%";
+-----------------------------------------------------+-----------+
| Variable_name                                       | Value     |
+-----------------------------------------------------+-----------+
| long_query_time                                     | 1.000000  |
| max_long_data_size                                  | 1048576   |
| performance_schema_events_waits_history_long_size   | 10000     |
+-----------------------------------------------------+-----------+
3 rows in set (0.00 sec)
```

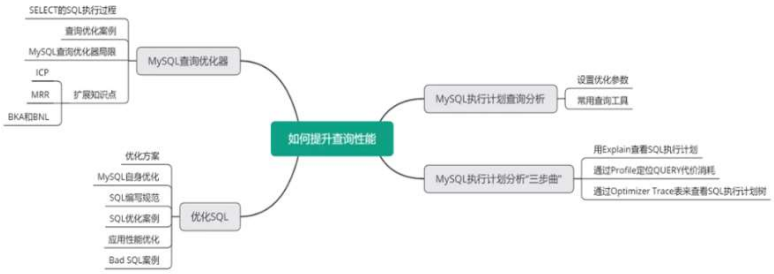
2) 在global级别动态修改
   mysql> set global log_slow_queries=ON;
   mysql慢日志使用最多的地方就是定位sql的性能问题
3) 参数：
   i. slow_query_log 的值为ON为开启慢查询日志，OFF则为关闭慢查询日志。
   ii. slow_query_log_file 的值是记录的慢查询日志到文件中（注意：默认名为主机名.log，慢查询日志是否写入指定文件中，需要指定慢查询的输出日志格式为文件，相关命令为：show variables like ‘%log_output%’；去查看输出的格式）。
   iii. long_query_time 指定了慢查询的阈值，即如果执行语句的时间超过该阈值则为慢查询语句，默认值为10秒。
   iv. log_queries_not_using_indexes 如果值设置为ON，则会记录所有没有利用索引的查询（注意：如果只是将log_queries_not_using_indexes设置为ON，而将slow_query_log设置为OFF，此时该设置也不会生效，即该设置生效的前提是slow_query_log的值设置为ON），一般在性能调优的时候会暂时开启。
4) mysql慢日志查询分析工具：mysqldumpslow
   i. 命令行选项：
      i) -s：是表示按照何种方式排序。c、t、l、r分别是按照记录次数、时间、查询时间、返回的记录数来排序，前面加a(ac、at、al、ar)表示相应的倒叙
      ii) -t： 后面跟一个数字，表示返回前面多少条的数据
      iii) -g：后边跟一个正则匹配模式（注意：大小写是不敏感的）
   ii. 属性：
      i) 出现次数(Count),
      ii) 执行最长时间(Time),
      iii) 累计总耗费时间(Time),
      iv) 等待锁的时间(Lock),
      v) 发送给客户端的行总数(Rows),
      vi) 扫描的行总数(Rows),
      vii) 用户以及sql语句本身(抽象了一下格式，比如 limit 1, 20 用 limit N,N 表示).
   iii. 得到返回记录集最多的10个SQL。
      mysqldumpslow -s r -t 10 /database/mysql/mysql06_slow.log
      得到访问次数最多的10个SQL
      mysqldumpslow -s c -t 10 /database/mysql/mysql06_slow.log
      得到按照时间排序的前10条里面含有左连接的查询语句。
      mysqldumpslow -s t -t 10 -g “left join” /database/mysql/mysql06_slow.log
      另外建议在使用这些命令时结合 | 和more 使用 ，否则有可能出现刷屏的情况。
      mysqldumpslow -s r -t 20 /mysqldata/mysql/mysql06-slow.log | more
5) 慢日志格式

```
# Time: 2016-12-22T09:41:17.777439Z
# User@Host: homestead[homestead] @  [192.168.10.1]  Id:  1543
# Query_time: 2.001423  Lock_time: 0.000000 Rows_sent: 1  Rows_examined: 0
use homestead;
SET timestamp=1482399677;
select sleep(2);
```

vi. 慢日志的清理与备份删除：
   1) 直接删除慢日志文件，执行flush logs（必须的）。
   2) 备份：先用mv重命名文件（不要跨分区），然后执行flush logs（必须的）。
   3) 另外修改系统变量slow_query_log_file也可以立即生效；
   4) 执行flush logs，系统会先close当前的句柄，然后重新open；mv ，rm日志文件系统并不会报错，具体的原因可以Google下linux i_count i_nlink ；
   5) 看完及时关闭，打开慢查询日志可能会对系统性能有一点点影响，如果你的MySQL是主-从结构，可以考虑打开其中一台从服务器的慢查询日志，这样既可以监控慢查询，对系统性能影响又小。

c. 四个指标：
   i. 查询吞吐量
      1) MySQL 有一个名为 Questions 的内部计数器（根据 MySQL 用语，这是一个服务器状态变量），客户端每发送一个查询语句，其值就会加一。由 Questions 指标带来的以客户端为中心的视角常常比相关的Queries 计数器更容易解释。作为存储程序的一部分，后者也会计算已执行语句的数量，以及诸如PREPARE 和 DEALLOCATE PREPARE 指令运行的次数，作为服务器端预处理语句的一部分。
   ii. 查询执行性能

1)



2) 注意：（上面有具体例子）

i. 全表扫描还是索引扫描。

对于小表来说，二者 IO 调用次数和返回时间相差不大；但对于大表，如果全表扫描，那么查询返回的时间就会很长，就需要使用索引扫描加快查询速度。但并不是要求 DBA 根据每一种查询条件组合都要创建索引，索引过多也会降低写入和修改的速度，而且如果导致表数据和索引数据比例失调，也不利于后期的正常维护。

ii. 如何创建索引，在哪些列上建立索引适合业务需求？

一般情况下，你可以在选择度高的列上创建索引，也可以在 status 列上创建索引。创建索引时，要注意避免冗余索引，除非一些特殊情况外。如 index(a,b,c) 和 index(a)，其中 a 的单列索引就是冗余索引。

iii. 创建索引以后，尽量不要过频修改。

业务可以根据现有的索引情况合理使用索引，而不是每次都去修改索引。能在索引中完成的查找，就不要回表查询。比如 SELECT 某个具体字段，就有助于实现覆盖索引从而降低 IO 次数，达到优化 SQL 的目的。

iv. 多表关联的 SQL，在关联列上要有索引且字段类型一致，

这样 MySQL 在进行嵌套循环连接查找时可以使用索引，且不会因为字段类型不匹配而发生隐式转换进而导致无法使用索引的情况发生。在现实情况中，开发经常会出现 SQL 中关联列字段类型不一致或者传入的参数类型与字段类型不匹配的情况，这样就会导致无法使用索引，在优化 SQL 时需要重点排查这种情况。另外索引列上使用函数也不会涉及索引。

多表关联时，尽量让结果集小的表作为驱动表，注意是结果集小的表，不是小表。

v. 在日常中你会发现全模糊匹配的查询会很慢

由于 MySQL 的索引是 B+ 树结构，所以当查询条件为全模糊时，例如‘%**%’，索引无法使用，这时需要通过添加其他选择度高的列或者条件作为一种补充，从而加快查询速度。

当然也可以通过强制 SQL 进行全索引扫描，但这种方式不好，尽量不要在 SQL 中添加hints。对于这种全模糊匹配的场景，可以放到 ES 或者 solr 中解决。尽量不要使用子查询，对子查询产生的临时表再扫描时将无索引可查询，只能进行全表扫描，并且 MySQL 对于出现在 from 中的表无所谓顺序，对于 where 中也无所谓顺序，这也是可以优化 SQL 的地方。

另外 order by/group by 的 SQL 涉及排序，尽量在索引中包含排序字段，并让排序字段的排序顺序与索引列中的顺序相同，这样可以避免排序或减少排序次数。

vi. MySQL 自身优化 SQL

另外 MySQL 自身也对 SQL 自动进行了优化处理。MySQL 能够处理的优化类型有下面这些。

i) 重新定义表的关联顺序。多表关联查询时，MySQL 日益强大的优化器会自动选择驱动表，以及表的连接顺序，基于 cost 规则极大减少 SQL 执行的时间。

ii) 使用等价变化规则。MySQL 可以合并或减少一些比较，还可以移除一些恒成立或恒不成立的判断。

iii) 优化 count()、min() 和 max()。索引和列是否可为空通常可以帮助 MySQL 优化这类表达式，如查找最小值只需找到索引树最左边的第一条记录即可。

iii. 连接情况

1) 连接数

i. 经常会遇见”MySQL: ERROR 1040: Too many connections”的情况，一种是访问量确实很高，MySQL服务器抗不住，这个时候就要考虑增加从服务器分散读压力，另外一种情况是MySQL配置文件中max_connections值过小：

ii.

```
mysql> show variables like 'max_connections';
+-----------------+-------+
| Variable_name   | Value |
+-----------------+-------+
| max_connections | 151   |
+-----------------+-------+
1 row in set, 1 warning (0.00 sec)
```

这台MySQL服务器最大连接数是256，然后查询一下服务器响应的最大连接数：

mysql> show global status like ‘Max_used_connections’;

MySQL服务器过去的最大连接数是245，没有达到服务器连接数上限256，应该没有出现1040错误，比较理想的设置是：

Max_used_connections / max_connections * 100% ≈ 85%

最大连接数占上限连接数的85%左右，如果发现比例在10%以下，MySQL服务器连接数上限设置的过高了。

iv. 缓冲池使用情况（上面有提到）

四. 总结

※a. 就是sql语句要更加简化，高效，严格按照sql编写规范，防止sql注入问题。

b. Sql语句将时间大部分花在了执行时间，也就是查数据和取数据。减少数据访问，可以使用redis缓存一些热门数据，这样可以避免总去数据库中查询这些经常使用但是不常更新的数据。

c. 返回较少的数据，比如使用分页技术

d. 减少交互次数，批处理

e. 查询尽量走索引

f. 减少服务器cpu开销或者增加资源（仅仅适用于紧急情况）

g. 使用慢查询日志去发现慢查询，使用执行计划去判断查询是否正常运行，总是去测试你的查询看看是否他们运行在最佳状态下。久而久之性能总会变化，避免在整个表上使用count(*),它可能锁住整张表，使查询保持一致以便后续相似的查询可以使用查询缓存，在适当的情形下使用GROUP BY而不是DISTINCT，在WHERE，GROUP BY和ORDER BY子句中使用有索引的列，保持索引简单,不在多个索引中包含同一个列，有时候MySQL会使用错误的索引,对于这种情况使用USE INDEX，检查使用SQL_MODE=STRICT的问题，对于记录数小于5的索引字段，在UNION的时候使用LIMIT不是是用OR。

h. 为了避免在更新前SELECT，使用INSERT ON DUPLICATE KEY或者INSERT IGNORE ,不要用UPDATE去实现，不要使用 MAX,使用索引字段和ORDER BY子句，LIMIT M，N实际上可以减缓查询在某些情况下，有节制地使用，在WHERE子句中使用UNION代替子查询，在重新启动的MySQL，以确保您的数据在内存和查询速度快，考虑持久连接，而不是多个连接，以减少开销，基准查询，包括使用服务器上的负载。

i. 分析效率比较低的sql语句：

   i. 通过EXPLAIN 分析低效 SQL的执行计划：

      通过以上步骤查询到效率低的 SQL 后，我们可以通过 explain 或者 desc 获取MySQL 如何执行 SELECT 语句的信息，包括 select 语句执行过程表如何连接和连接 的次序。

   ii. 优化数据库对象的类型，可以使用procedure analyse（）函数，对表进行分析，该函数对表中的列中的数据进行优化建议；

      i. 类型原则：可以正确表示并且要使用存储数据最短类型。这样可以减少内存，磁盘，CPU缓存使用。

         例：select * from 表名procedure analyse（）；

   iii. 可以通过慢查询日志定位那些执行效率较低的 sql 语句，用 --log-slow-queries[=file_name] 选项启动时， mysqld 写一个包含所有执行时间超过long_query_time 秒的 SQL 语句的日志文件。可以链接到管理维护中的相关章节。

   iv. 使用show processlist查看当前MYSQL的线程， 命令慢查询日志在查询结束以后才纪录，所以在应用反映执行效率出现问题的时候查 询慢查询日志并不能定位问题，可以使用 show processlist 命令查看当前 MySQL 在进行的线程，包括线程的状态，是否锁表等等，可以实时的查看 SQL 执行情况， 同时对一些锁表操作进行优化。