

MySQL性能优化篇

一、性能优化的思路

1. 首先需要使用【慢查询日志】功能，去获取所有查询时间比较长的SQL语句
2. 查看执行计划，查看有问题的SQL的执行计划
3. 针对查询慢的SQL语句进行优化
4. 使用【show profile[s]】查看有问题的SQL的性能使用情况
5. 调整操作系统参数优化
6. 升级服务器硬件

二、慢查询日志

1、慢查询日志介绍

数据库查询快慢是影响项目性能的一大因素，对于数据库，我们除了要优化 SQL，更重要的是得先找到需要优化的SQL。MySQL数据库有一个“慢查询日志”功能，用来记录查询时间超过某个设定值的SQL语句，这将极大程度帮助我们快速定位到症结所在，以便对症下药。

至于查询时间的多少才算慢，每个项目、业务都有不同的要求。

MySQL的慢查询日志功能默认是关闭的，需要手动开启。

2、开启慢查询功能

- 查看是否开启慢查询功能

```
mysql> show variables like '%slow_query%';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| slow_query_log | OFF |
| slow_query_log_file | /var/lib/mysql/localhost-slow.log |
+-----+-----+
2 rows in set (0.00 sec)

mysql> SHOW VARIABLES LIKE 'long_query_time%';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| long_query_time | 10.000000 |
+-----+-----+
1 row in set (0.00 sec)
```

参数说明：

- 1 - 【slow_query_log】 : 是否开启慢查询日志, 1为开启, 0为关闭。
- 2 - 【log-slow-queries】 : 旧版(5.6以下版本)MySQL数据库慢查询日志存储路径。可以不设置该参数, 系统则会默认给一个缺省的文件host_name-slow.log
- 3 - 【slow-query-log-file】: 新版(5.6及以上版本)MySQL数据库慢查询日志存储路径。可以不设置该参数, 系统则会默认给一个缺省的文件host_name-slow.log
- 4 - 【long_query_time】 : 慢查询阈值, 当查询时间多于设定的阈值时, 记录日志, 【单位为秒】。

• 临时开启慢查询功能

在MySQL执行SQL语句设置, 但是如果重启MySQL的话将失效

```
1 set global slow_query_log = ON;  
2 set global long_query_time = 1;
```

```
mysql> set global slow_query_log = ON;  
Query OK, 0 rows affected (1.71 sec)  
  
mysql> set long_query_time = 1;  
Query OK, 0 rows affected (0.00 sec)
```

• 永久开启慢查询功能

修改/etc/my.cnf配置文件, 重启MySQL, 这种永久生效。

```
1 [mysqld]  
2 slow_query_log=ON  
3 long_query_time=1
```

3、慢查询日志格式

```
mysql> select sleep(3);  
+-----+  
| sleep(3) |  
+-----+  
|          0 |  
+-----+  
1 row in set (3.00 sec)
```

```
[root@localhost mysql]# tail -100f localhost-slow.log  
/usr/sbin/mysqld, Version: 5.6.41 (MySQL Community Server (GPL)). started with:  
Tcp port: 3306 Unix socket: /var/lib/mysql/mysql.sock  
Time Id Command Argument  
# Time: 180830 5:51:30  
# User@Host: root[root] @ localhost [] Id: 7  
# Query_time: 3.000600 Lock_time: 0.000000 Rows_sent: 1 Rows_examined: 0  
SET timestamp=1535579490;  
select sleep(3);
```

格式说明:

- 第一行,SQL查询执行的具体时间
- 第二行,执行SQL查询的连接信息, 用户和连接IP
- 第三行,记录了一些我们比较有用的信息, 如下解析

- 1 Query_time, 这条SQL执行的时间, 越长则越慢
- 2 Lock_time, 在MySQL服务器阶段(不是在存储引擎阶段)等待表锁时间
- 3 Rows_sent, 查询返回的行数
- 4 Rows_examined, 查询检查的行数, 越长就当然越费时间

- 第四行, 设置时间戳, 没有实际意义, 只是和第一行对应执行时间。
- 第五行及后面所有行 (第二个# Time:之前), 执行的sql语句记录信息, 因为sql可能会很长。

4、分析慢查询日志的工具

使用mysqldumpslow工具, mysqldumpslow是MySQL自带的慢查询日志工具。可以使用mysqldumpslow工具搜索慢查询日志中的SQL语句。

得到按照时间排序的前10条里面含有左连接的查询语句:

```
1 [root@localhost mysql]# mysqldumpslow -s t -t 10 -g "left join"
  /var/lib/mysql/slow.log
```

常用参数说明:

-s: 是表示按照何种方式排序

al 平均锁定时间
ar 平均返回记录时间
at 平均查询时间 (默认)
c 计数
l 锁定时间
r 返回记录
t 查询时间

-t: 是top n的意思, 即为返回前面多少条的数据

-g: 后边可以写一个正则匹配模式, 大小写不敏感的

```
1 [root@mysql132 mysql]# mysqldumpslow -s t /var/lib/mysql/mysql132-slow.log
2
3 Reading mysql slow query log from /var/lib/mysql/mysql132-slow.log
4 Count: 1 Time=143.16s (143s) Lock=0.00s (0s) Rows=27907961.0 (27907961),
  root[root]@localhost
5   select * from t_slow a left join t_slow b on a.name=b.name
6
7 Count: 5 Time=5.80s (28s) Lock=0.00s (0s) Rows=0.0 (0),
  root[root]@localhost
8   insert into t_slow(name,address) select name,address from t_slow
9
10 Count: 1 Time=3.01s (3s) Lock=0.00s (0s) Rows=1.0 (1),
  root[root]@localhost
11   select sleep(N)
```

三、查看执行计划

1、建表语句

```
1 create table tuser(  
2     id int primary key auto_increment,  
3     name varchar(100),  
4     age int,  
5     sex char(1),  
6     address varchar(100)  
7 );  
8 alter table tuser add index idx_name_age(name(100),age);  
9 alter table tuser add index idx_sex(sex(1));  
10  
11 insert into tuser(id,name,age,sex,address) values (1,'zhangsan',20,'1','北京');  
12 insert into tuser(id,name,age,sex,address) values (2,'lisi',16,'1','上海');  
13 insert into tuser(id,name,age,sex,address) values (3,'wangwu',34,'1','杭州');  
14 insert into tuser(id,name,age,sex,address) values (4,'wangxin',26,'2','广州');  
15 insert into tuser(id,name,age,sex,address) values (5,'wudi',18,'2','上海');
```

2、介绍

MySQL 提供了一个 **EXPLAIN** 命令, 它可以对 **SELECT** 语句的执行计划进行分析, 并输出 SELECT 执行的详细信息, 以供开发人员针对性优化。

使用explain这个命令来查看一个这些SQL语句的执行计划, 查看该SQL语句有没有使用上了索引, 有没有做全表扫描, 这都可以通过explain命令来查看。

可以通过explain命令深入了解MySQL的基于开销的优化器, 还可以获得很多可能被优化器考虑到的访问策略的细节, 以及当运行SQL语句时哪种策略预计会被优化器采用。

EXPLAIN 命令用法十分简单, 在 **SELECT** 语句前加上 **explain** 就可以了, 例如:

```
mysql> explain select * from user;  
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+  
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |  
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+  
| 1 | SIMPLE | user | ALL | NULL | NULL | NULL | NULL | 7 | NULL |  
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+  
1 row in set (0.00 sec)
```

3、参数说明

EXPLAIN 命令的输出内容大致如下:

```
1 mysql> explain select * from tuser where id = 2 \G  
2 ***** 1. row *****  
3       id: 1  
4       select_type: SIMPLE  
5       table: tuser  
6       partitions: NULL  
7       type: const  
8       possible_keys: PRIMARY  
9       key: PRIMARY  
10      key_len: 4  
11      ref: const  
12      rows: 1  
13      filtered: 100.00  
14      Extra: NULL
```

各列的含义如下:

- id: SELECT 查询的标识符. 每个 SELECT 都会自动分配一个唯一的标识符.
- **select_type**: SELECT 查询的类型.
- table: 查询的是哪个表
- partitions: 匹配的分区
- **type**: join 类型
- possible_keys: 此次查询中可能选用的索引
- key: 此次查询中确切使用到的索引.
- ref: 哪个字段或常数与 key 一起被使用
- rows: 显示此查询一共扫描了多少行. 这个是一个估计值.
- filtered: 表示此查询条件所过滤的数据的百分比
- **extra**: 额外的信息

1) id

每个单位查询的SELECT语句都会自动分配的一个唯一标识符, 表示查询中操作表的顺序, 有四种情况:

- id相同: 执行顺序由上到下

```
mysql> explain
-> select * from t1 left join t2 on t1.aa=t2.aa left JOIN t3 on t2.aa=t3.aa
-> ;
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	t1	NULL	ALL	NULL	NULL	NULL	NULL	12	100.00	NULL
1	SIMPLE	t2	NULL	eq_ref	PRIMARY	PRIMARY	4	hello.t1.aa	1	100.00	NULL
1	SIMPLE	t3	NULL	eq_ref	PRIMARY	PRIMARY	4	hello.t2.aa	1	100.00	NULL

3 rows in set, 1 warning (0.00 sec)

- id不同: 如果是子查询, id号会自增, **id越大, 优先级越高**.

```
mysql> EXPLAIN
-> SELECT * FROM t1 WHERE aa =(SELECT aa FROM t2 WHERE aa = (SELECT aa FROM t3 WHERE aa = 1));
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	PRIMARY	t1	NULL	const	PRIMARY	PRIMARY	4	const	1	100.00	NULL
2	SUBQUERY	t2	NULL	const	PRIMARY	PRIMARY	4	const	1	100.00	Using index
3	SUBQUERY	t3	NULL	const	PRIMARY	PRIMARY	4	const	1	100.00	Using index

3 rows in set, 1 warning (0.00 sec)

- id相同的不同的同时存在

```
mysql> EXPLAIN
-> SELECT t1.* FROM (select t2.bb from t2 where cc=(select t3.cc from t3 where t3.cc=1)) b, t1 where t1.bb=b.bb;
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	PRIMARY	t2	NULL	ALL	NULL	NULL	NULL	NULL	12	10.00	Using where
1	PRIMARY	t1	NULL	ALL	NULL	NULL	NULL	NULL	12	10.00	Using where; Using join buffer (Block Nested Loop)
3	SUBQUERY	t3	NULL	ALL	NULL	NULL	NULL	NULL	12	10.00	Using where

3 rows in set, 1 warning (0.00 sec)

2) select_type (重要)

单位查询的查询类型, 比如: 普通查询、联合查询(union、union all)、子查询等复杂查询。

simple

表示不需要union操作或者不包含子查询的简单select查询。有连接查询时, 外层的查询为simple。

```
mysql> explain select * from user ;
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	user	ALL	NULL	NULL	NULL	NULL	1	NULL

1 row in set (0.00 sec)

primary

一个需要union操作或者含有子查询的select，位于最外层的单位查询的select_type即为primary。

```
mysql> explain select (select name from user) from user ;
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1  | PRIMARY    | user  | index | NULL          | idx_name_name | 308     | NULL | 1    | Using index |
| 2  | SUBQUERY   | user  | index | NULL          | idx_name_name | 308     | NULL | 1    | Using index |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
2 rows in set (0.00 sec)
```

union

union连接的两个select查询，第一个查询是derived派生表，除了第一个表外，第二个以后的表select_type都是union

```
mysql> explain select * from user a union select * from user b;
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1  | PRIMARY    | a     | ALL  | NULL          | NULL | NULL    | NULL | 1    | NULL |
| 2  | UNION      | b     | ALL  | NULL          | NULL | NULL    | NULL | 1    | NULL |
| NULL | UNION RESULT | <union1,2> | ALL | NULL          | NULL | NULL    | NULL | NULL | Using temporary |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
3 rows in set (0.00 sec)
```

```
mysql> explain select * from (select * from user a union select * from user b) c;
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1  | PRIMARY    | <derived2> | ALL  | NULL          | NULL | NULL    | NULL | 4    | NULL |
| 2  | DERIVED    | a     | ALL  | NULL          | NULL | NULL    | NULL | 1    | NULL |
| 3  | UNION      | b     | ALL  | NULL          | NULL | NULL    | NULL | 1    | NULL |
| NULL | UNION RESULT | <union2,3> | ALL | NULL          | NULL | NULL    | NULL | NULL | Using temporary |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
4 rows in set (0.00 sec)
```

- DERIVED 在FROM列表中包含的子查询被标记为DERIVED（衍生），MySQL会递归执行这些子查询，把结果放在临时表中
- UNION 若第二个SELECT出现在UNION之后，则被标记为UNION：若UNION包含在FROM子句的子查询中，外层SELECT将被标记为：DERIVED
- UNION RESULT 从UNION表获取结果的SELECT

dependent union

与union一样，出现在union 或union all语句中，但是这个查询要受到外部查询的影响

```
mysql> explain select * from user u where u.id in (select id from user a union select id from user b);
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1  | PRIMARY    | u     | ALL  | NULL          | NULL | NULL    | NULL | 1    | Using where |
| 2  | DEPENDENT SUBQUERY | a     | eq_ref | PRIMARY      | PRIMARY | 4       | func | 1    | Using index |
| 3  | DEPENDENT UNION | b     | eq_ref | PRIMARY      | PRIMARY | 4       | func | 1    | Using index |
| NULL | UNION RESULT | <union2,3> | ALL | NULL          | NULL | NULL    | NULL | NULL | Using temporary |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
4 rows in set (0.00 sec)
```

union result

包含union的结果集，在union和union all语句中,因为它不需要参与查询，所以id字段为null

```
mysql> explain select * from user a union select * from user b;
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1  | PRIMARY    | a     | ALL  | NULL          | NULL | NULL    | NULL | 1    | NULL |
| 2  | UNION      | b     | ALL  | NULL          | NULL | NULL    | NULL | 1    | NULL |
| NULL | UNION RESULT | <union1,2> | ALL | NULL          | NULL | NULL    | NULL | NULL | Using temporary |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
3 rows in set (0.00 sec)
```

subquery

除了from字句中包含的子查询外，其他地方出现的子查询都可能是subquery

```
mysql> explain select (select name from user) from user ;
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1  | PRIMARY    | user  | index | NULL          | idx_name_name | 308     | NULL | 1    | Using index |
| 2  | SUBQUERY   | user  | index | NULL          | idx_name_name | 308     | NULL | 1    | Using index |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
2 rows in set (0.00 sec)
```

dependent subquery

与dependent union类似，表示这个subquery的查询要受到外部表查询的影响

```
mysql> explain select (select name from user a where a.id = b.id) from user b;
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	PRIMARY	b	index	NULL	idx_name_name	308	NULL	1	Using index
2	DEPENDENT SUBQUERY	a	eq_ref	PRIMARY	PRIMARY	4	kbb.b.id	1	NULL

2 rows in set (0.00 sec)

derived

from字句中出现的子查询，也叫做派生表，其他数据库中可能叫做内联视图或嵌套select

```
mysql> explain select * from (select * from user) t;
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	PRIMARY	<derived2>	ALL	NULL	NULL	NULL	NULL	2	NULL
2	DERIVED	user	ALL	NULL	NULL	NULL	NULL	1	NULL

2 rows in set (0.00 sec)

3) table

显示的单位查询的表名，有如下几种情况：

- 如果查询使用了别名，那么这里显示的是别名
- 如果不涉及对数据表的操作，那么这显示为null
- 如果显示为尖括号括起来的就表示这个是临时表，后边的N就是执行计划中的id，表示结果来自于这个查询产生。
- 如果是尖括号括起来的<union M,N>，与类似，也是一个临时表，表示这个结果来自于union查询的id为M,N的结果集。

```
1 EXPLAIN
2 select * from (select * from tuser union select * from tuser) t
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	PRIMARY	<derived2>	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	10	100.00	(Null)
2	DERIVED	tuser	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	5	100.00	(Null)
3	UNION	tuser	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	5	100.00	(Null)
Full UNION RESULT		<union2,3>	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	(Null)	(Null)	Using temporary

4) partitions

使用的哪些分区（对于非分区表值为null）。

5.7之后的版本默认会有 partitions 和 filtered两列，但是5.6版本中是没有的，需要

使用explain partitions select来显示带有partitions 的列，

使用explain extended select来显示带有filtered的列。

什么是分区表？

mysql内部实现的表的水平拆分，所有数据还在一个表中，但物理存储根据一定的规则放在不同的文件中。这个是mysql支持的功能，业务代码无需改动。

技术现状：

业内进行一些技术交流的时候也更多的是自己分库分表，而不是使用分区表。

- 1) 分区表，分区键设计不太灵活，如果不走分区键，很容易出现全表锁
- 2) 一旦数据量并发量上来，如果在分区表实施关联，就是一个灾难
- 3) 自己分库分表，自己掌控业务场景与访问模式，可控。分区表，研发写了一个sql，都不确定mysql是怎么玩的，不太可控

5) type (重要)

显示的是单位查询的**连接类型**或者理解为**访问类型**，访问性能依次从好到差：

```

1  system
2  const
3  eq_ref
4  ref
5  fulltext
6  ref_or_null
7  unique_subquery
8  index_subquery
9  range
10 index_merge
11 index
12 ALL

```

注意事项：

- 1 - 除了all之外，其他的type都可以使用到索引
- 2 - 除了index_merge之外，其他的type只可以用到一个索引
- 3 - 最少要使用到range级别

system

表中**只有一行数据或者是空表**。等于系统表，这是const类型的特例，平时不会出现，这个也可以忽略不计

```
mysql> explain select * from (select * from user where id = 1) t;
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	PRIMARY	<derived2>	system	NULL	NULL	NULL	NULL	1	NULL
2	DERIVED	user	const	PRIMARY	PRIMARY	4	const	1	NULL

2 rows in set (0.00 sec)

注：上图为mysql5.6效果，mysql5.7只有一行计划而且select_type为simple

const (重要)

使用**唯一索引或者主键**，返回记录一定是1行记录的等值where条件时，通常type是const。其他数据库也叫做唯一索引扫描。

```
mysql> explain select * from user where id = 1;
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	user	const	PRIMARY	PRIMARY	4	const	1	NULL

1 row in set (0.00 sec)

eq_ref (重要)

唯一性索引扫描，对于每个索引键，表中只有一条记录与之匹配。常见于主键或唯一索引扫描

```
mysql> explain select * from user a left join user b on a.id = b.id;
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	a	ALL	NULL	NULL	NULL	NULL	1	NULL
1	SIMPLE	b	eq_ref	PRIMARY	PRIMARY	4	kbb.a.id	1	NULL

2 rows in set (0.00 sec)

ref (重要)

非唯一性索引扫描，返回匹配某个单独值的所有行，本质上也是一种索引访问，它返回所有匹配某个单独值的行，然而，它可能会找到多个符合条件的行，所以他应该属于查找和扫描的混合体。

- 组合索引

```
mysql> explain select * from user a left join user b on a.name = b.name;
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | a | ALL | NULL | NULL | NULL | NULL | 1 | NULL |
| 1 | SIMPLE | b | ref | idx_name_name | idx_name_name | 303 | kkb.a.name | 1 | NULL |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
2 rows in set (0.00 sec)

mysql> explain select * from user where name = 'zhangsan';
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | user | ref | idx_name_name | idx_name_name | 303 | const | 1 | Using index condition |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.01 sec)
```

- 非唯一索引

```
mysql> explain select * from user where sex = '1' ;
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | user | ref | idx_sex | idx_sex | 4 | const | 1 | Using index condition |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

fulltext

全文索引检索，要注意，全文索引的优先级很高，若全文索引和普通索引同时存在时，mysql不管代价，优先选择使用全文索引

ref_or_null

与ref方法类似，只是增加了null值的比较。实际用的不多。

unique_subquery

用于where中的in形式子查询，子查询返回不重复值唯一值

index_subquery

用于in形式子查询使用到了辅助索引或者in常数列表，子查询可能返回重复值，可以使用索引将子查询去重。

range (重要)

索引范围扫描，常见于使用>, <, is null, between, in, like等运算符的查询中。

```
mysql> explain select * from user where name like 'a%';
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | user | range | idx_name_name | idx_name_name | 303 | NULL | 1 | Using index condition |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql> explain select * from user where id > 1;
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | user | range | PRIMARY | PRIMARY | 4 | NULL | 1 | Using where |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

index_merge

表示查询使用了两个以上的索引，最后取交集或者并集，常见and, or的条件使用了不同的索引，官方排序这个在ref_or_null之后，但是实际上由于要读取所有索引，性能可能大部分时间都不如range

index (重要)

select结果列中使用到了索引，type会显示为index。

全部索引扫描，把索引从头到尾扫一遍，常见于使用索引列就可以处理不需要读取数据文件的查询、可以使用索引排序或者分组的查询。

```
mysql> explain select name from user ;
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key          | key_len | ref | rows | Extra      |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1  | SIMPLE      | user  | index | NULL          | idx_name_age | 308     | NULL | 1    | Using index |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql> explain select age from user ;
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key          | key_len | ref | rows | Extra      |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1  | SIMPLE      | user  | index | NULL          | idx_name_age | 308     | NULL | 1    | Using index |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

all (重要)

这个就是**全表扫描数据文件**，然后再在**server层进行过滤**返回符合要求的记录。

```
mysql> explain select * from user ;
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1  | SIMPLE      | user  | ALL  | NULL          | NULL | NULL    | NULL | 1    | NULL  |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql> explain select * from user where address='致真大厦';
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra      |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1  | SIMPLE      | user  | ALL  | NULL          | NULL | NULL    | NULL | 1    | Using where |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

6) possible_keys

此次查询中可能选用的索引，一个或多个

7) key

查询真正使用到的索引，select_type为index_merge时，这里可能出现两个以上的索引，其他的select_type这里只会出现一个。

8) key_len

key_len越小 索引效果越好。

- 1 key_len的长度计算公式:
- 2 varchar(10)变长字段且允许NULL = 10 * (character set:
utf8=3,gbk=2,latin1=1)+1(NULL)+2(变长字段)
- 3 varchar(10)变长字段且不允许NULL = 10 * (character set:
utf8=3,gbk=2,latin1=1)+2(变长字段)
- 4 char(10)固定字段且允许NULL = 10 * (character set:
utf8=3,gbk=2,latin1=1)+1(NULL)
- 5 char(10)固定字段且不允许NULL = 10 * (character set:
utf8=3,gbk=2,latin1=1)
- 6 bigint的长度是8bytes
- 7 int key_len长度是4 , tinyint的长度是1
- 8 smallint 长度是2 middleint长度是3

- 用于处理查询的索引长度，如果是单列索引，那就整个索引长度算进去，如果是多列索引，那么查询不一定都能使用到所有的列，具体使用到了多少个列的索引，这里就会计算进去，没有使用到的列，这里不会计算进去。
- 留意下这个列的值，算一下你的多列索引总长度就知道有没有使用到所有的列了。

- 另外, `key_len`只计算where条件用到的索引长度, 而排序和分组就算用到了索引, 也不会计算到`key_len`中。

9) ref

- 如果是使用的常数等值查询, 这里会显示const
- 如果是连接查询, 被驱动表的执行计划这里会显示驱动表的关联字段
- 如果是条件使用了表达式或者函数, 或者条件列发生了内部隐式转换, 这里可能显示为func

10) rows

这里是执行计划中估算的扫描行数, 不是精确值 (InnoDB不是精确的值, MyISAM是精确的值, 主要原因是InnoDB里面使用了MVCC并发机制)

11) filtered

filtered列指示将由mysql server层需要对存储引擎层返回的记录进行筛选的估计百分比, 也就是说存储引擎层返回的结果中包含有效记录数的百分比。最大值为100, 这意味着没有对行进行筛选。值从100减小表示过滤量增加。rows显示检查的估计行数, `rows*filtered`显示将与下表联接的行数。例如, 如果rows为1000, filtered为50.00 (50%), 则要与下表联接的行数为 $1000 \times 50\% = 500$ 。

12) extra (重要)

这个列包含不适合在其他列中显示单十分重要的额外的信息, 这个列可以显示的信息非常多, 有几十种

Using filesort

说明mysql会对数据使用一个外部的索引排序, 而不是按照表内的索引顺序进行读取。MySQL中无法利用索引完成的排序操作称为“文件排序”。需要优化sql。

```
mysql> explain select col1 from t1 where col1 = 'ac' order by col3\G
```

```
***** 1. row *****
      id: 1
    select_type: SIMPLE
        table: t1
         type: ref
possible_keys: idx_col1_col2_col3
          key: idx_col1_col2_col3
       key_len: 13
         ref: const
        rows: 142
     Extra: Using where; Using index; Using filesort
1 row in set (0.00 sec)
```

按照字段col1、col2、col3顺序建立的索引

```
mysql> explain select col1 from t1 where col1 = 'ac' order by col2, col3\G
```

```
***** 1. row *****
      id: 1
    select_type: SIMPLE
        table: t1
         type: ref
possible_keys: idx_col1_col2_col3
          key: idx_col1_col2_col3
       key_len: 13
         ref: const
        rows: 142
     Extra: Using where; Using index
1 row in set (0.00 sec)
```

Using temporary

使用了临时表保存中间结果，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。需要优化SQL

```
mysql> explain select col1 from t1 where col1 in ('ac','ab','aa') group by col2\G
***** 1. row *****
      id: 1
    select_type: SIMPLE
      table: t1
      type: range
possible_keys: idx_col1_col2
      key: idx_col1_col2
     key_len: 13
        ref: NULL
       rows: 569
  Extra: Using where; Using index; Using temporary; Using filesort
1 row in set (0.00 sec)
```

索引字段为col1和col2，而这条语句group by没有按索引顺序使用，只使用了col2

```
mysql> explain select col1 from t1 where col1 in ('ac', 'ab') group by col1, col2\G
***** 1. row *****
      id: 1
    select_type: SIMPLE
      table: t1
      type: range
possible_keys: idx_col1_col2_col3
      key: idx_col1_col2_col3
     key_len: 26
        ref: NULL
       rows: 4
  Extra: Using where; Using index for group-by
```

<https://blog.csdn.net/why15732625998>

using index (重要)

查询时不需要回表查询，直接通过索引就可以获取查询的结果数据。

- 表示相应的SELECT查询中使用到了覆盖索引 (Covering Index)，避免访问表的数据行，效率不错！
- 如果同时出现Using Where，说明索引被用来执行查找索引键值
- 如果没有同时出现Using Where，表明索引用来读取数据而非执行查找动作。

```
mysql> explain select id,name,age from mylock;
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | mylock | index | NULL | idx_name_age | 68 | NULL | 6 | Using index |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql> explain select name,age from mylock;
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | mylock | index | NULL | idx_name_age | 68 | NULL | 6 | Using index |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql> explain select age from mylock;
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | mylock | index | NULL | idx_name_age | 68 | NULL | 6 | Using index |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

using where (重要)

表示Mysql将对storage engine提取的结果进行过滤，过滤条件字段无索引；

```
mysql> explain select * from mylock where address = '致真大厦';
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | mylock | ALL | NULL | NULL | NULL | NULL | 6 | Using where |
+----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

Using join buffer

表明使用了连接缓存,比如说在查询的时候，多表join的次数非常多，那么将配置文件中的缓冲区的join buffer调大一些。

impossible where

where子句的值总是false，不能用来获取任何元组

```
1 | SELECT * FROM t_user WHERE id = '1' and id = '2'
```

四、SQL语句优化（开发人员）

1、索引优化

- 为搜索字段（**where中的条件**）、排序字段、select查询列，创建合适的索引，不过要考虑数据的业务场景：查询多还是增删多？
- 尽量建立**组合索引**并注意组合索引的创建顺序，按照顺序组织查询条件、尽量将筛选粒度大的查询条件放到最左边。
- **尽量使用覆盖索引**，SELECT语句中尽量不要使用*。
- order by、group by语句要尽量使用到索引
- 索引长度尽量短，短索引可以节省索引空间，使查找的速度得到提升，同时内存中也可以装载更多的索引键值。太长的列，可以选择建立前缀索引
- 索引更新不能频繁，更新非常频繁的数据不适宜建索引，因为维护索引的成本。
- order by的索引生效，order by排序应该遵循最佳左前缀查询，如果是使用多个索引字段进行排序，那么排序的规则必须相同（同是升序或者降序），否则索引同样会失效。

2、LIMIT优化

- 如果预计SELECT语句的查询结果是一条，最好使用 **LIMIT 1**，可以停止全表扫描。

```
1 | SELECT * FROM user WHERE username='全力詹'; -- username没有建立唯一索引
2 |
3 | SELECT * FROM user WHERE username='全力詹' LIMIT 1;
```

- 处理分页会使用到 **LIMIT**，当翻页到非常靠后的页面的时候，偏移量会非常大，这时LIMIT的效率会非常差。 **LIMIT OFFSET , SIZE;**

LIMIT的优化问题，其实是 **OFFSET** 的问题，它会导致MySQL扫描大量不需要的行然后再抛弃掉。

解决方案：单表分页时，使用自增主键排序之后，先使用where条件 `id > offset值`，limit后面只写 rows

```
1 | select * from (select * from tuser2 where id > 1000000 and id < 1000500
  | ORDER BY id) t limit 0, 20
```

3、其他查询优化

- 小表驱动大表，建议使用left join时，以小表关联大表，因为使用join的话，第一张表是必须全扫描的，以少关联多就可以减少这个扫描次数。
- 避免全表扫描，mysql在使用不等于(!=或者<>)的时候无法使用索引导致全表扫描。在查询的时候，如果对索引使用不等于的操作将会导致索引失效，进行全表扫描
- 避免mysql放弃索引查询，如果mysql估计使用全表扫描要比使用索引快，则不使用索引。（最典型的场景就是数据量少的时候）
- JOIN两张表的关联字段最好都建立索引，而且最好字段类型是一样的。

```
1 SELECT * FROM orders o LEFT JOIN user u on o.user_id = u.id
2
3 orders表中的user_id和用户表中的id，类型要一致
```

- WHERE条件中尽量不要使用not in语句（建议使用not exists）
- 合理利用慢查询日志、explain执行计划查询、show profile查看SQL执行时的资源使用情况。

五、profile分析语句

1、介绍

Query Profiler是MySQL自带的一种**query诊断分析工具**，通过它可以分析出一条SQL语句的**硬件性能瓶颈**在什么地方。

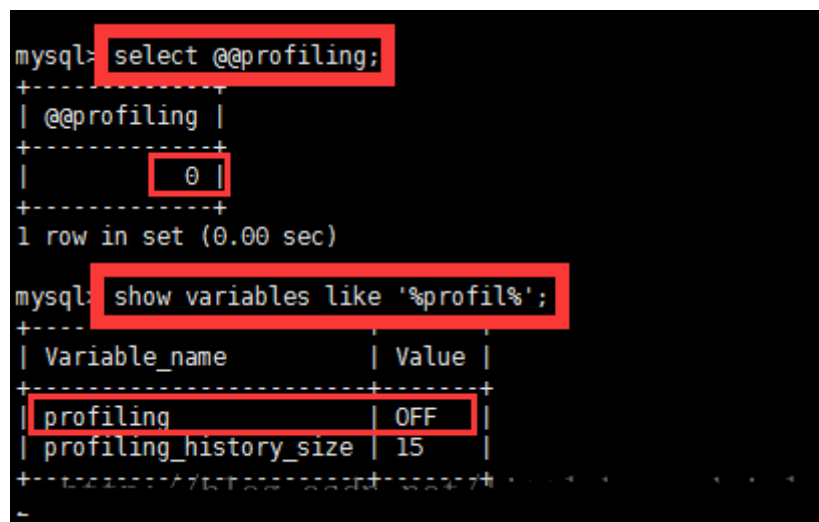
通常我们是使用的explain,以及slow query log都无法做到精确分析，但是Query Profiler却可以定位出一条SQL语句执行的各种资源消耗情况，比如CPU，IO等，以及该SQL执行所耗费的时间等。不过该工具只有在MySQL 5.0.37以及以上版本中才有实现。

默认的情况下，MYSQL的该功能没有打开，需要自己手动启动。

2、开启Profile功能

- Profile 功能由MySQL会话变量：**profiling**控制,默认是**OFF**关闭状态。
- 查看是否开启了Profile功能:

```
1 select @@profiling;
2 -- 或者
3 show variables like '%profil%';
```



```
mysql> select @@profiling;
+-----+
| @@profiling |
+-----+
| 0 |
+-----+
1 row in set (0.00 sec)

mysql> show variables like '%profil%';
+-----+-----+
| Variable_name | Value |
+-----+-----+
| profiling     | OFF   |
| profiling_history_size | 15    |
+-----+-----+
```

- 开启profile功能

```
1 | set profiling=1; --1是开启、0是关闭
```

2、语句使用

- **show profile** 和 **show profiles** 语句可以展示**当前会话**(退出session后,profiling重置为0) 中执行语句的资源使用情况.
- **show profiles** :以列表形式显示最近发送到服务器上执行的语句的**资源使用情况**.显示的记录数由变量:*profiling_history_size* 控制,默认15条

```
mysql> show profiles;
+-----+-----+-----+
| Query_ID | Duration | Query |
+-----+-----+-----+
| 1 | 0.00023200 | select count(*) from sys_user |
+-----+-----+-----+
```

- **show profile**: 展示最近一条语句执行的详细资源占用信息,默认显示 **Status**和**Duration**两列

```
mysql> show profile;
+-----+-----+
| Status | Duration |
+-----+-----+
| starting | 0.000068 |
| Opening tables | 0.000031 |
| System lock | 0.000004 |
| Table lock | 0.000007 |
| init | 0.000014 |
| optimizing | 0.000006 |
| statistics | 0.000010 |
| preparing | 0.000008 |
| executing | 0.000005 |
| Sending data | 0.000058 |
| end | 0.000004 |
| query end | 0.000002 |
| freeing items | 0.000012 |
| logging slow query | 0.000001 |
| cleaning up | 0.000002 |
+-----+-----+
15 rows in set (0.00 sec)
```

- **show profile** 还可根据 **show profiles** 列表中的 **Query_ID** ,选择显示某条记录的性能分析信息


```

1  语法结构:
2
3  SHOW PROFILE [type [, type] ... ]
4      [FOR QUERY n]
5      [LIMIT row_count [OFFSET offset]]
6
7  type:
8      ALL
9      | BLOCK IO
10     | CONTEXT SWITCHES
11     | CPU
12     | IPC
13     | MEMORY
14     | PAGE FAULTS
15     | SOURCE
16     | SWAPS

```

type是可选的，取值范围可以如下：

- ALL 显示所有性能信息
- BLOCK IO 显示块IO操作的次数
- CONTEXT SWITCHES 显示上下文切换次数，不管是主动还是被动
- CPU 显示用户CPU时间、系统CPU时间
- IPC 显示发送和接收的消息数量
- MEMORY [暂未实现]
- PAGE FAULTS 显示页错误数量
- SOURCE 显示源码中的函数名称与位置
- SWAPS 显示SWAP的次数

4、示例

查看是否打开了性能分析功能

```
1 | select @@profiling;
```

```

mysql> select @@profiling;
+-----+
| @@profiling |
+-----+
| 0 |
+-----+
1 row in set (0.00 sec)

```

打开 profiling 功能

```
1 | set profiling=1;
```

```

mysql> set profiling=1;
Query OK, 0 rows affected (0.00 sec)

mysql>

```

执行sql语句

```
mysql> select count(*) from sys_user;
+-----+
| count(*) |
+-----+
|      30 |
+-----+
1 row in set (0.00 sec)

mysql> select * from sys_user where user_name like '于%';
```

执行 show profiles 查看分析列表

```
mysql> show profiles;
+-----+-----+-----+
| Query_ID | Duration | Query |
+-----+-----+-----+
|      1 | 0.00023400 | select count(*) from sys_user |
|      2 | 0.00037800 | select * from sys_user where user_name like '于%' |
+-----+-----+-----+
2 rows in set (0.00 sec)

mysql>
```

查询第二条语句的执行情况

```
1 | show profile for query 2;
```

```
mysql> show profile for query 2;
+-----+-----+
| Status | Duration |
+-----+-----+
| starting | 0.000072 |
| Opening tables | 0.000015 |
| System lock | 0.000004 |
| Table lock | 0.000008 |
| init | 0.000044 |
| optimizing | 0.000008 |
| statistics | 0.000013 |
| preparing | 0.000008 |
| executing | 0.000003 |
| Sending data | 0.000178 |
| end | 0.000004 |
| query end | 0.000002 |
| freeing items | 0.000015 |
| logging slow query | 0.000002 |
| cleaning up | 0.000002 |
+-----+-----+
15 rows in set (0.00 sec)
```

可指定资源类型查询

```
1 | show profile cpu,swaps for query 2;
```

```
mysql> show profile cpu,swaps for query 2;
```

Status	Duration	CPU_user	CPU_system	Swaps
starting	0.000072	0.000000	0.000000	0
Opening tables	0.000015	0.000000	0.000000	0
System lock	0.000004	0.000000	0.000000	0
Table lock	0.000008	0.000000	0.000000	0
init	0.000044	0.000000	0.000000	0
optimizing	0.000008	0.000000	0.000000	0
statistics	0.000013	0.000000	0.000000	0
preparing	0.000008	0.000000	0.000000	0
executing	0.000003	0.000000	0.000000	0
Sending data	0.000178	0.000000	0.000000	0
end	0.000004	0.000000	0.000000	0
query end	0.000002	0.000000	0.000000	0
freeing items	0.000015	0.000000	0.000000	0
logging slow query	0.000002	0.000000	0.000000	0
cleaning up	0.000002	0.000000	0.000000	0

5.应注意的结论

```
mysql> show profile cpu,block io for query 8;
```

Status	Duration	CPU_user	CPU_system	Block_ops_in	Block_ops_out
starting	0.000032	0.000000	0.000000	0	0
Waiting for query cache lock	0.000006	0.000000	0.000000	0	0
checking query cache for query	0.000195	0.000999	0.000000	0	0
checking permissions	0.000014	0.000000	0.000000	0	0
Opening tables	0.185543	0.001000	0.003000	0	0
System lock	0.000018	0.000000	0.000000	0	0
Waiting for query cache lock	0.000023	0.000000	0.000000	0	0
init	0.000114	0.001000	0.000000	0	0
optimizing	0.000006	0.000000	0.000000	0	0
statistics	0.000073	0.000000	0.000000	0	0
preparing	0.000013	0.000000	0.000000	0	0
Creating tmp table	0.000127	0.000000	0.000000	0	0
executing	0.000005	0.000000	0.000000	0	0
Copying to tmp table	2.585168	1.203817	0.591910	0	0
Sorting result	0.000106	0.000000	0.000000	0	0
Sending data	0.000096	0.000000	0.000000	0	0
end	0.000006	0.000000	0.000000	0	0
removing tmp table	0.000026	0.000000	0.000000	0	0
end	0.000005	0.000000	0.000000	0	0
query end	0.000005	0.000000	0.000000	0	0

Creating tmp table 创建临时表	
starting	0.000020
Waiting for query cache lock	0.000006
checking query cache for query	0.000049
checking permissions	0.000007
Opening tables	0.000017
System lock	0.000016
Waiting for query cache lock	0.000023
init	0.000040
optimizing	0.000005
statistics	0.000025
preparing	0.000055
Creating tmp table	0.000071
executing	0.000004
Copying to tmp table	1.386291
Sorting result	0.000058
Sending data	0.000055
end	0.000004
removing tmp table	0.000022
end	0.000007
query end	0.000006
closing tables	0.000008
freeing items	0.000012
Waiting for query cache lock	0.000003
freeing items	0.000050
Waiting for query cache lock	0.000004
freeing items	0.000002
storing result in query cache	0.000004
logging slow query	0.000002
cleaning up	0.000003

converting HEAP to MyISAM 查询结果太大，内存不够用往磁盘上搬；

Creating tmp table 创建临时表：拷贝数据到临时表，用完再删除；

Copying to tmp table on disk 把内存临时表复制到磁盘，危险。

七、服务器层面优化

1、缓冲区优化

将数据保存在内存中，保证从内存读取数据

- 设置足够大的 `innodb_buffer_pool_size`，将数据读取到内存中。

1 | 建议 `innodb_buffer_pool_size` 设置为总内存大小的 3/4 或者 4/5。

- 怎样确定 `innodb_buffer_pool_size` 足够大。数据是从内存读取而不是硬盘？

```
mysql> SHOW GLOBAL STATUS LIKE 'innodb_buffer_pool_pages_%';
+-----+
| Variable_name          | Value |
+-----+
| Innodb_buffer_pool_pages_data | 129037 |
| Innodb_buffer_pool_pages_dirty | 362 |
| Innodb_buffer_pool_pages_flushed | 9998 |
| Innodb_buffer_pool_pages_free | 0 为0则表示buffer pool已经被用光
| Innodb_buffer_pool_pages_misc | 2035 |
| Innodb_buffer_pool_pages_total | 131072 |
+-----+
6 rows in set (0.00 sec)
```

2、降低磁盘写入次数

- 对于生产环境来说，很多日志是不需要开启的，比如：**通用查询日志、慢查询日志、错误日志**
- 使用足够大的写入缓存 `innodb_log_file_size`

```
1 | 推荐 innodb_log_file_size 设置为 0.25 * innodb_buffer_pool_size
```

- 设置合适的`innodb_flush_log_at_trx_commit`，和日志落盘有关系。

3、MySQL数据库配置优化

- 表示缓冲池字节大小。
推荐值为物理内存的50%~80%。

```
1 | innodb_buffer_pool_size
```

- 用来控制redo log刷新到磁盘的策略。

```
1 | innodb_flush_log_at_trx_commit=1
```

- 每提交1次事务同步写到磁盘中，可以设置为n。

```
1 | sync_binlog=1
```

- 脏页占`innodb_buffer_pool_size`的比例时，触发刷脏页到磁盘。 推荐值为25%~50%。

```
1 | innodb_max_dirty_pages_pct=30
```

- 后台进程最大IO性能指标。
默认200，如果SSD，调整为5000~20000

```
1 | innodb_io_capacity=200
```

在MySQL5.1.X版本中，由于代码写死，因此最多只会刷新100个脏页到磁盘、合并20个插入缓冲，即使磁盘有能力处理更多的请求，也只会处理这么多，这样在更新量较大（比如大批量INSERT）的时候，脏页刷新可能就会跟不上，导致性能下降。

而在MySQL5.5.X版本里，innodb_io_capacity参数可以动态调整刷新脏页的数量，这在一定程度上解决了这一问题。

innodb_io_capacity参数默认是200，单位是页。该参数设置的大小取决于硬盘的IOPS，即每秒的输入输出量（或读写次数）。

至于什么样的磁盘配置应该设置innodb_io_capacity参数的值是多少，大家可参考下表。

表 1-1 磁盘配置与 innodb_io_capacity 参数值

innodb_io_capacity	磁盘配置
200	单盘 SAS/SATA
2000	SAS*12 RAID10
5000	SSD
50 000	FUSION-IO

- 指定innodb共享表空间文件的大小。

```
1 | innodb_data_file_path
```

- 慢查询日志的阈值设置，单位秒。

```
1 | long_query_time=0.3
```

- mysql复制的形式，row为MySQL8.0的默认形式。

```
1 | binlog_format=row
```

- 调高该参数则应降低interactive_timeout、wait_timeout的值。

```
1 | max_connections=200
```

- 过大，实例恢复时间长；过小，造成日志切换频繁。

```
1 | innodb_log_file_size
```

- 全量日志建议关闭。
默认关闭。

4、操作系统优化

1) 内核参数优化

CentOS系统针对mysql的参数优化

内核相关参数(/etc/sysctl.conf)

以下参数可以直接放到sysctl.conf文件的末尾。

1.增加监听队列上限：

```
net.core.somaxconn = 65535
net.core.netdev_max_backlog = 65535
net.ipv4.tcp_max_syn_backlog = 65535
```

2.加快TCP连接的回收：

```
net.ipv4.tcp_fin_timeout = 10
net.ipv4.tcp_tw_reuse = 1
net.ipv4.tcp_tw_recycle = 1
```

3.TCP连接接收和发送缓冲区大小的默认值和最大值：

```
net.core.wmem_default = 87380
net.core.wmem_max = 16777216
net.core.rmem_default = 87380
net.core.rmem_max = 16777216
```

4.减少失效连接所占用的TCP资源的数量，加快资源回收的效率：

```
net.ipv4.tcp_keepalive_time = 120
net.ipv4.tcp_keepalive_intvl = 30
net.ipv4.tcp_keepalive_probes = 3
```

5.单个共享内存段的最大值：

```
kernel.shmmax = 4294967295
```

1. 这个参数应该设置的足够大，以便能在一个共享内存段下容纳整个的InnoDB缓冲池的大小。
2. 这个值的大小对于64位linux系统，可取的最大值为(物理内存值-1)byte，建议值为大于物理内存的一半，一般取值大于InnoDB缓冲池的大小即可。

6.控制换出运行时内存的相对权重：

```
vm.swappiness = 0
```

这个参数当内存不足时会对性能产生比较明显的影响。（设置为0，表示Linux内核虚拟内存完全被占用，才会使用交换区。）

Linux系统内存交换区：

在Linux系统安装时都会有一个特殊的磁盘分区，称之为系统交换分区。

使用 `free -m` 命令可以看到swap就是内存交换区。

作用：当操作系统没有足够的内存时，就会将部分虚拟内存写到磁盘的交换区中，这样就会发生内存交换。

如果Linux系统上完全禁用交换分区，带来的风险：

1. 降低操作系统的性能
2. 容易造成内存溢出，崩溃，或都被操作系统kill掉

2) 增加资源限制

打开文件数的限制以下参数可以直接放到(/etc/security/limit.conf)文件的末尾：

```
* soft nfile 65535
```

```
* hard nfile 65535
```

*：表示对所有用户有效
soft：表示当前系统生效的设置（soft不能大于hard）
hard：表明系统中所能设定的最大值
nfile：表示所限制的资源是打开文件的最大数目
65535：限制的数量

以上两行配置将可打开的文件数量增加到65535个，以保证可以打开足够多的文件句柄。

注意：这个文件的修改需要重启系统才能生效。

3) 磁盘调度策略

1.cfq (完全公平队列策略，Linux2.6.18之后内核的系统默认策略)

该模式按进程创建多个队列，各个进程发来的IO请求会被cfq以轮循方式处理，对每个IO请求都是公平的。该策略适合离散读的应用。

2.deadline (截止时间调度策略)

deadline，包含读和写两个队列，确保在一个截止时间内服务请求（截止时间是可调整的），而默认读期限短于写期限。这样就防止了写操作因为不能被读取而饿死的现象，deadline对数据库类应用是最好的选择。

3.noop (电梯式调度策略)

noop只实现一个简单的FIFO队列，倾向饿死读而利于写，因此noop对于闪存设备、RAM及嵌入式系统是最好的选择。

4.anticipatory (预料I/O调度策略)

本质上与deadline策略一样，但在最后一次读操作之后，要等待6ms，才能继续进行对其它I/O请求进行调度。它会在每个6ms中插入新的I/O操作，合并写入流，用写入延时换取最大的写入吞吐量。anticipatory适合于写入较多的环境，比如文件服务器。该策略对数据库环境表现很差。

查看调度策略的方法：

```
cat /sys/block/devname/queue/scheduler
```

修改调度策略的方法：

```
echo /sys/block/devname/queue/scheduler
```

5、服务器硬件优化

提升硬件设备，例如选择尽量高频率的内存（频率不能高于主板的支持）、提升网络带宽、使用SSD高速磁盘、提升CPU性能等。

CPU的选择：

- 对于数据库并发比较高的场景，CPU的数量比频率重要。

- 对于CPU密集型场景和频繁执行复杂SQL的场景，CPU的频率越高越好。

