# 第09章_性能分析工具的使用

讲师：尚硅谷-宋红康（江湖人称：康师傅）

官网： http://www.atguigu.com

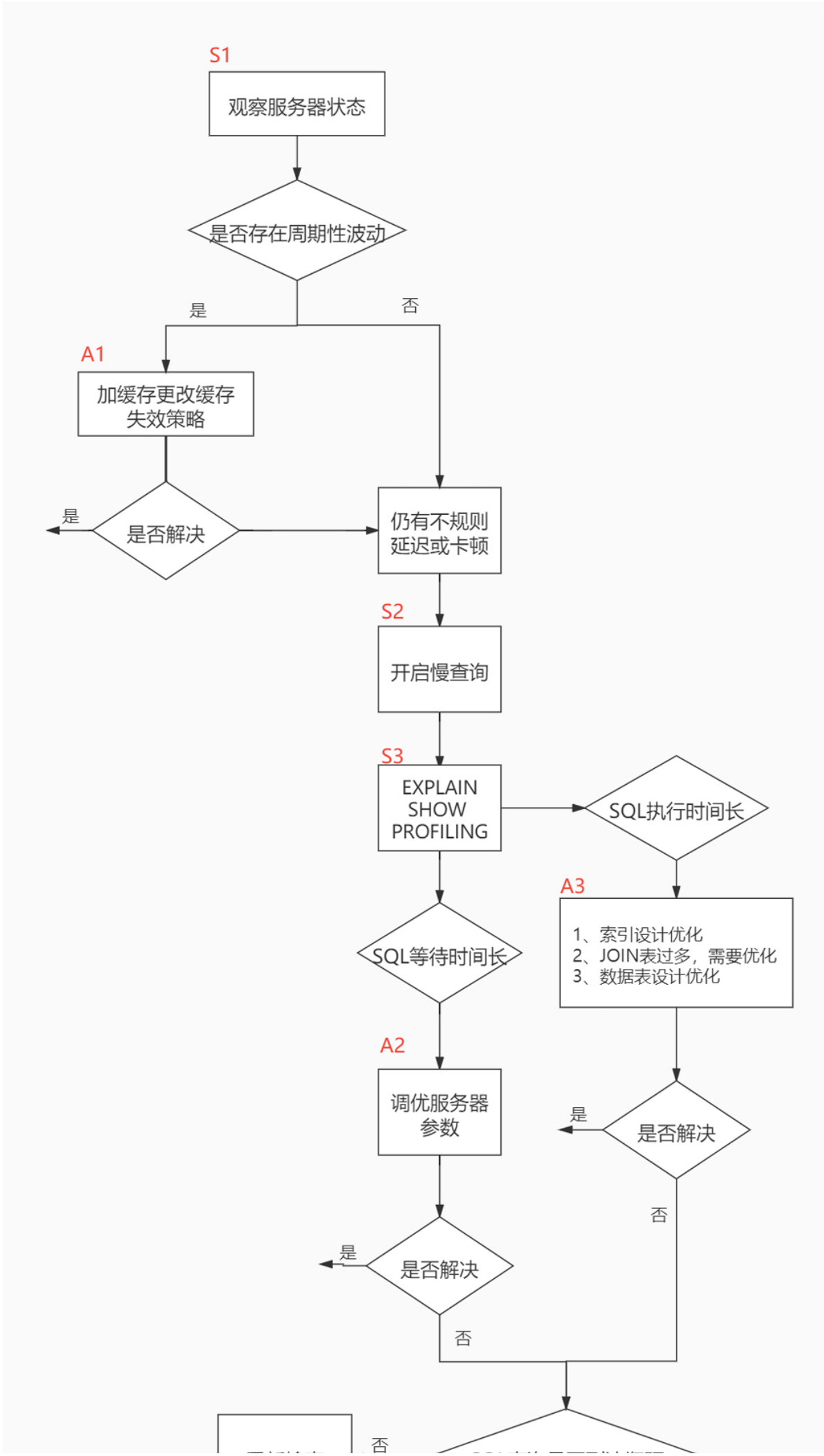
---

在数据库调优中，我们的目标就是`响应时间更快，吞吐量更大`。利用宏观的监控工具和微观的日志分析可以帮我们快速找到调优的思路和方式

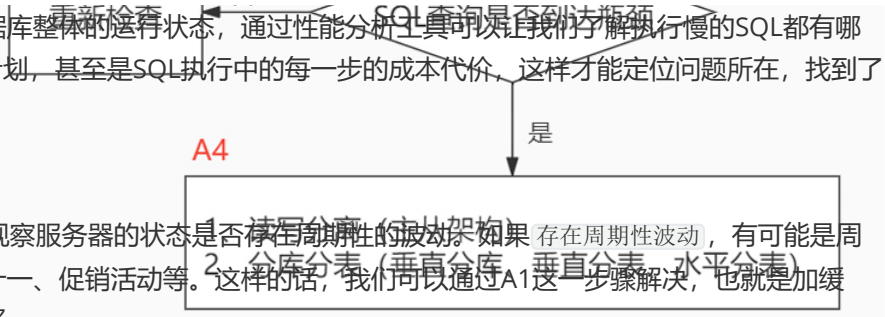
## 1. 数据库服务器的优化步骤

当我们遇到数据库调优问题的时候，该如何思考呢？这里把思考的流程整理成下面这张图。

整个流程划分成了观察`(Show status)`和`行动(Action)`两个部分。字母 S 的部分代表观察（会使用相应的分析工具），字母 A 代表的部分是行动（对应分析可以采取的行动）。


S1
观察服务器状态
是否存在周期性波动
是
否
A1
加缓存更改缓存
失效策略
是
是否解决
仍有不规则
延迟或卡顿
S2
开启慢查询
S3
EXPLAIN
SHOW
PROFILING
SQL执行时间长
A3
1、索引设计优化
2、JOIN表过多，需要优化
3、数据表设计优化
SQL等待时间长
A2
调优服务器
参数
是
是否解决
否
是
是否解决
否
否

我们可以通过观察了解数据库整体的运行状态，通过性能分析工具可以让我们了解执行慢的SQL都有哪些，查看具体的SQL执行计划，甚至是SQL执行中的每一步的成本代价，这样才能定位问题所在，找到了问题，再采取相应的行动。




**详细解释一下这张图：**

首先在S1部分，我们需要观察服务器的状态是否存在周期性的波动。如果`存在周期性波动`，有可能是周期性节点的原因，比如双十一、促销活动等。这样的话，我们可以通过A1这一步骤解决，也就是加缓存，或者更改缓存失效策略。

如果缓存策略没有解决，或者不是周期性波动的原因，我们就需要进一步`分析查询延迟和卡顿的原因`。接下来进入S2这一步，我们需要开启慢查询。慢查询可以帮我们定位执行慢的SQL语句。我们可以通过设置`long-query_time`参数定义“慢"的阈值，如果sQL执行时间超过了long_query_time，则会认为是慢查询。当收集上来这些慢查询之后，我们就可以通过分析工具对慢查询日志进行分析。

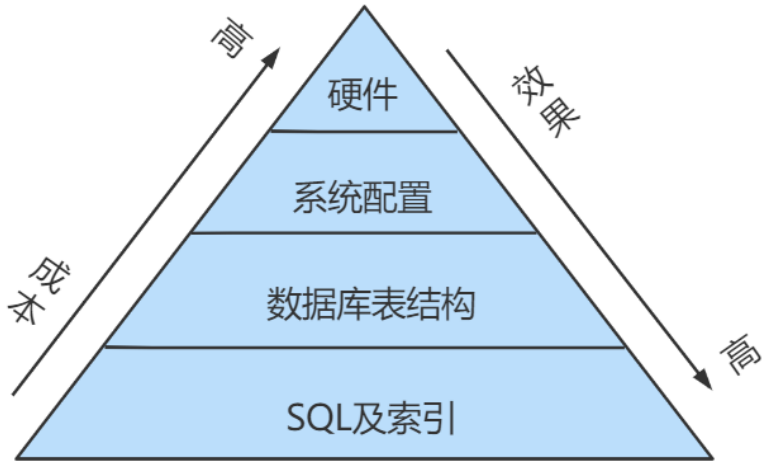
在S3这一步骤中，我们就知道了执行慢的SQL，这样就可以针对性地用`EXPLAIN`查看对应SQL语句的执行计划，或者使用`show profile`查看SQL中每一个步骤的时间成本。这样我们就可以了解SQL查询慢是因为执行时间长，还是等待时间长。

如果是SQL等待时间长，我们进入A2步骤。在这一步骤中，我们可以`调优服务器的参数`，比如适当增加数据库缓冲池等。如果是SQL执行时间长，就进入A3步骤，这一步中我们需要考虑是索引设计的问题？还是查询关联的数据表过多？还是因为数据表的字段设计问题导致了这一现象。然后在这些维度上进行对应的调整。

如果A2和A3都不能解决问题，我们需要考虑数据库自身的SQL查询性能是否已经达到了瓶颈，如果确认没有达到性能瓶颈，就需要重新检查，重复以上的步骤。如果已经达到了`性能瓶颈`，进入A4阶段，需要考虑`增加服务器`，采用`读写分离`的架构，或者考虑对数据库进行`分库分表`，比如垂直分库、垂直分表和水平分表等。

以上就是数据库调优的流程思路。如果我们发现执行SQL时存在不规则延迟或卡顿的时候，就可以采用分析工具帮我们定位有问题的SQL，这三种分析工具你可以理解是SQL调优的三个步骤：`慢查询`、`EXPLAIN`和`SHOW PROFILING`。

**小结：**




可以看到数据库调优的步骤中越往金字塔尖走，其成本越高，效果越差，因此我们在数据库调优的过程中，要重点把握金字塔底部的 sql 及索引调优，数据库表结构调优，系统配置参数调优等软件层面的调优

## 2. 查看系统性能参数

在MySQL中，可以使用 `SHOW STATUS` 语句查询一些MySQL数据库服务器的 `性能参数` 、 `执行频率` 。

SHOW STATUS语句语法如下：

```
SHOW [GLOBAL|SESSION] STATUS LIKE '参数';
```

一些常用的性能参数如下：

- Connections：连接MySQL服务器的次数。
- Uptime：MySQL服务器的上线时间。
- Slow_queries：慢查询的次数。
- Innodb_rows_read：Select查询返回的行数
- Innodb_rows_inserted：执行INSERT操作插入的行数
- Innodb_rows_updated：执行UPDATE操作更新的行数
- Innodb_rows_deleted：执行DELETE操作删除的行数
- Com_select：查询操作的次数。
- Com_insert：插入操作的次数。对于批量插入的 INSERT 操作，只累加一次。
- Com_update：更新操作的次数。
- Com_delete：删除操作的次数。

举例：

- 若查询MySQL服务器的连接次数，则可以执行如下语句：

```
mysql> SHOW STATUS LIKE 'Connections';
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| Connections   | 34    |
+---------------+-------+
1 row in set (0.00 sec)
```

- 若查询服务器工作时间，则可以执行如下语句：

```
mysql> SHOW STATUS LIKE 'Uptime';
+---------------+--------+
| Variable_name | Value  |
+---------------+--------+
| Uptime        | 332933 |
+---------------+--------+
1 row in set (0.00 sec)
```

- 若查询MySQL服务器的慢查询次数，则可以执行如下语句：

```
mysql> SHOW STATUS LIKE 'Slow_queries';
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| Slow_queries  | 0     |
+---------------+-------+
1 row in set (0.00 sec)
```

注：慢查询次数参数可以结合慢查询日志找出慢查询语句，然后针对慢查询语句进行`表结构优化`或者`查询语句优化`

- 查看存储引擎增删改查的行数，则可以执行如下语句：

```
mysql> show status like 'innodb_rows_%';
+----------------------+----------+
| Variable_name        | Value    |
+----------------------+----------+
| Innodb_rows_deleted  | 0        |
| Innodb_rows_inserted | 1000902  |
| Innodb_rows_read     | 37011100 |
| Innodb_rows_updated  | 0        |
+----------------------+----------+
4 rows in set (0.00 sec)
```

# 3. 统计SQL的查询成本：last_query_cost

一条SQL查询语句在执行前需要确定查询执行计划，如果存在多种执行计划的话，MySQL会计算每个执行计划所需要的成本，从中选择`成本最小`的一个作为最终执行的执行计划。

如果我们想要查看某条SQL语句的查询成本，可以在执行完这条SQL语句之后，通过查看当前会话中的`last_query_cost`变量值来得到当前查询的成本。它通常也是我们`评价一个查询的执行效率`的一个常用指标。这个查询成本对应的是`SQL语句所需要读取的页的数量`。

我们依然使用第8章的 student_info 表为例：

```
CREATE TABLE `student_info` (
    `id` INT(11) NOT NULL AUTO_INCREMENT,
    `student_id` INT NOT NULL ,
    `name` VARCHAR(20) DEFAULT NULL,
    `course_id` INT NOT NULL ,
    `class_id` INT(11) DEFAULT NULL,
    `create_time` DATETIME DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
    PRIMARY KEY (`id`)
) ENGINE=INNODB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;
```

如果我们想要查询 id=900001 的记录，然后看下查询成本，我们可以直接在聚簇索引上进行查找：

```
SELECT student_id, class_id, NAME, create_time FROM student_info
WHERE id = 900001;
```

```
mysql> SELECT * FROM student_info WHERE id = 900001;
+--------+------------+--------+-----------+----------+---------------------+
| id     | student_id | name   | course_id | class_id | create_time         |
+--------+------------+--------+-----------+----------+---------------------+
| 900001 |     154633 | SYnwsA |     10019 |    10134 | 2022-08-08 22:33:02 |
+--------+------------+--------+-----------+----------+---------------------+
1 row in set (0.00 sec)
```

运行结果（1 条记录，运行时间为`0.042s`）

然后再看下查询优化器的成本，实际上我们只需要检索一个页即可，`Value`表示 I/O 加载的数据页的页数：

```
mysql> SHOW STATUS LIKE 'last_query_cost';
+-----------------+----------+
| Variable_name   | Value    |
+-----------------+----------+
| Last_query_cost | 1.000000 |
+-----------------+----------+
```

如果我们想要查询 id 在 900001 到 9000100 之间的学生记录呢?

```
SELECT student_id, class_id, NAME, create_time FROM student_info
WHERE id BETWEEN 900001 AND 900100;
```

运行结果（100 条记录，运行时间为`0.046s`）：

然后再看下查询优化器的成本，这时我们大概需要进行 20 个页的查询。

```
mysql> SHOW STATUS LIKE 'last_query_cost';
+-----------------+-----------+
| Variable_name   | Value     |
+-----------------+-----------+
| Last_query_cost | 21.134453 |
+-----------------+-----------+
```

你能看到页的数量是刚才的 20 倍，但是查询的效率并没有明显的变化，实际上这两个 SQL 查询的时间基本上一样，就是因为采用了顺序读取的方式将页面一次性加载到缓冲池中，然后再进行查找。虽然`页数量（last_query_cost）增加了不少`，但是通过缓冲池的机制，并`没有增加多少查询时间`。

②我们扩大下查询范围，student_id> 199900的学生记录呢？运行时间 0.01s，这时我们大概需要进行232个页的查询

```
mysql> SELECT * FROM student_info WHERE student_id > 199900;
+--------+------------+--------+-----------+----------+---------------------
+
| id     | student_id | name   | course_id | class_id | create_time
|
+--------+------------+--------+-----------+----------+---------------------
+
//...
| 523982 |     200000 | vcaUvw |     10010 |    10173 | 2022-08-08 22:32:31
|
```

```
+--------+------------+--------+-----------+----------+--------------------
+
516 rows in set (0.01 sec)

mysql> SHOW STATUS LIKE 'last_query_cost';
+-----------------+------------+
| Variable_name   | Value      |
+-----------------+------------+
| Last_query_cost | 232.459000 |
+-----------------+------------+
1 row in set (0.00 sec)
```

我们再次扩大范围，假若我们想要查询 student_id > 199000的学生记录呢？运行时间 0.02s，这时我们大概需要进行 2279个页的查询。

```
mysql> SELECT * FROM student_info WHERE student_id > 199000;
//...
5065 rows in set (0.02 sec)

mysql> SHOW STATUS LIKE 'last_query_cost';
+-----------------+-------------+
| Variable_name   | Value       |
+-----------------+-------------+
| Last_query_cost | 2279.509000 |
+-----------------+-------------+
1 row in set (0.00 sec)
```

不知道大家有没有发现，上面的查询页的数量是刚才的 10倍，但是查询的效率并没有明显的变化，就是因为采用了顺序读取的方式将页面一次性加载到缓冲池中，然后再进行查找。虽然页数量（`last_query_cost`）增加了不少，但是通过缓冲池的机制，并没有增加多少查询时间。

**使用场景：**查询 `last_query_cost` 对于比较开销是非常有用的，特别是我们有好几种查询方式可选的时候

> SQL查询是一个动态的过程，从页加载的角度，我们可以得到以下两点结论：
>
> 位置决定效率：如果页就在数据库缓冲池中，那么效率是最高的，否则还需要从内存或者磁盘中进行读取，当然针对单个页的读取来说，如果页存在于内存中，会比在磁盘中读取效率高很多。即数据库缓冲池>内存>磁盘
> 批量决定效率：如果我们从磁盘中单一页进行随机读，那么效率是很低的（差不多10ms），而采用顺序读取的方式，批量对页进行读取，平均一页的读取效率就会提升很多，甚至要快于单个页面在内存中的随机读取。即顺序读取>大于随机读取
> 所以说，遇到 I/O 并不用担心，方法找对了，效率还是很高的。我们首先要考虑数据存放的位置，如果是经常使用的数据就要尽量放到缓冲池中，其次我们可以充分利用磁盘的吞吐能力，一次性批量读取数据，这样单个页的读取效率也就得到了提升。
>
> 注：缓冲池和查询缓存并不是一个东西

# 4. 定位执行慢的 SQL：慢查询日志

MySQL的慢查询日志，用来记录在MySQL中 响应时间超过阈值 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10秒以上（不含10秒）的语句，认为是超出了我们的最大忍耐时间值。

它的主要作用是，帮助我们发现那些执行时间特别长的SQL查询，并且有针对性地进行优化，从而提高系统的整体效率。当我们的数据库服务器发生阻塞、运行变慢的时候，检查一下慢查询日志，找到那些慢查询，对解决问题很有帮助。比如一条sq执行超过5秒钟，我们就算慢SQL，希望能收集超过5秒的sql，结合explain进行全面分析。

默认情况下，MySQL数据库`没有开启慢查询日志`，需要我们手动来设置这个参数。**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会或多或少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。

## 4.1 开启慢查询日志参数

### 1. 开启slow_query_log

在使用前，我们需要先看下慢查询是否已经开启，使用下面这条命令即可：

查看慢查询日志是否开启，以及日志的位置

```
mysql> show variables like '%slow_query_log%';
+---------------------+----------------------------------+
| Variable_name       | Value                            |
+---------------------+----------------------------------+
| slow_query_log      | OFF                              |
| slow_query_log_file | /var/lib/mysql/centos01-slow.log |
+---------------------+----------------------------------+
2 rows in set (0.00 sec)
```

修改慢查询日志状态为开启，注意这里要加`global`，因为它是全局系统变量，否则会报错。

```
mysql> set global slow_query_log='ON';
Query OK, 0 rows affected (0.02 sec)
```

然后我们再来查看下慢查询日志是否开启，以及慢查询日志文件的位置：

```
mysql> show variables like '%slow_query_log%';
+---------------------+----------------------------------+
| Variable_name       | Value                            |
+---------------------+----------------------------------+
| slow_query_log      | ON                               |
| slow_query_log_file | /var/lib/mysql/centos01-slow.log |
+---------------------+----------------------------------+
2 rows in set (0.00 sec)
```

你能看到这时慢查询分析已经开启，同时文件保存在`/var/lib/mysql/atguigu02-slow.log`文件中。

### 2. 修改long_query_time阈值

接下来我们来看下慢查询的时间阈值设置，使用如下命令：

```
mysql > show variables like '%long_query_time%';
```

```
mysql> show variables like '%long_query_time%';
+-----------------+-----------+
| Variable_name   | Value     |
+-----------------+-----------+
| long_query_time | 10.000000 |
+-----------------+-----------+
1 row in set (0.00 sec)
```

这里如果我们想把时间缩短，比如设置为 1 秒，可以这样设置：

```
#测试发现：设置global的方式对当前session的long_query_time失效。对新连接的客户端有效。所以可以一并执行下述语句
mysql > set global long_query_time = 1;
mysql> show global variables like '%long_query_time%';
mysql> set long_query_time=1;
mysql> show variables like '%long_query_time%';
```

```
mysql> show global variables like '%long_query_time%';
+-----------------+----------+
| Variable_name   | Value    |
+-----------------+----------+
| long_query_time | 1.000000 |
+-----------------+----------+
1 row in set, 1 warning (0.00 sec)
```

**补充：配置文件中一并设置参数**

如下的方式相较于前面的命令行方式，可以看作是永久设置的方式。

修改`my.cnf`文件，[mysqld]下增加或修改参数`long_query_time`、`slow_query_log`和`slow_query_log_file`后，然后重启MySQL服务器。

```
[mysqld]
slow_query_log=ON#开启慢查询日志的开关
slow_query_log_file=/var/lib/mysql/atguigu-slow.log#慢查询日志的目录和文件名信息
long-query_time=3 #设置慢查询的阈值为3秒，超出此设定值的SQL即被记录到慢查询日志
log_output=FILE
```

如果不指定存储路径，慢查询日志将默认存储到MySQL数据库的数据文件夹下。如果不指定文件名，默认文件名为hostname-slow.log。

## 4.2 查看慢查询数目

查询当前系统中有多少条慢查询记录

```
SHOW GLOBAL STATUS LIKE '%Slow_queries%';
```

## 4.3 案例演示

**步骤1.建表**

```
CREATE TABLE `student` (
    `id` INT(11) NOT NULL AUTO_INCREMENT,
    `stuno` INT NOT NULL ,
    `name` VARCHAR(20) DEFAULT NULL,
    `age` INT(3) DEFAULT NULL,
    `classId` INT(11) DEFAULT NULL,
    PRIMARY KEY (`id`)
) ENGINE=INNODB AUTO_INCREMENT=1 DEFAULT CHARSET=utf8;
```

**步骤2：设置参数log_bin_trust_function_creators**

创建函数，假如报错：

```
This function has none of DETERMINISTIC......
```

- 命令开启：允许创建函数设置：

```
set global log_bin_trust_function_creators=1; # 不加global只是当前窗口有效。
```

**步骤3：创建函数**

随机产生字符串：（同上一章）

```
DELIMITER //
CREATE FUNCTION rand_string(n INT)
    RETURNS VARCHAR(255) #该函数会返回一个字符串
BEGIN
    DECLARE chars_str VARCHAR(100)
DEFAULT'abcdefghijklmnopqrstuvwxyzABCDEFJHIJKLMNOPQRSTUVWXYZ';
    DECLARE return_str VARCHAR(255) DEFAULT '';
    DECLARE i INT DEFAULT 0;
    WHILE i < n DO
        SET return_str
=CONCAT(return_str,SUBSTRING(chars_str,FLOOR(1+RAND()*52),1));
        SET i = i + 1;
    END WHILE;
    RETURN return_str;
END //
DELIMITER ;

#测试
SELECT rand_string(10);
```

产生随机数值：（同上一章）

```
DELIMITER //
CREATE FUNCTION rand_num (from_num INT ,to_num INT) RETURNS INT(11)
BEGIN
    DECLARE i INT DEFAULT 0;
    SET i = FLOOR(from_num +RAND()*(to_num - from_num+1)) ;
    RETURN i;
END //
DELIMITER ;

#测试:
SELECT rand_num(10,100);
```

**步骤4：创建存储过程**

```
DELIMITER //
CREATE PROCEDURE insert_stu1( START INT , max_num INT )
BEGIN
    DECLARE i INT DEFAULT 0;
    SET autocommit = 0; #设置手动提交事务
    REPEAT #循环
    SET i = i + 1; #赋值
    INSERT INTO student (stuno, NAME ,age ,classId ) VALUES
 ((START+i),rand_string(6),rand_num(10,100),rand_num(10,1000));
    UNTIL i = max_num
    END REPEAT;
    COMMIT; #提交事务
END //
DELIMITER ;
```

**步骤5：调用存储过程**

```
#调用刚刚写好的函数, 4000000条记录,从100001号开始
CALL insert_stu1(100001,4000000);
```

## 4.4 测试及分析

**1.执行一下下面的查询操作，进行慢查询语句的测试**

```
# 注意：此时long_query_time已经设置为1了哦~
mysql> SELECT * FROM student WHERE stuno = 3455655;
+---------+---------+--------+------+---------+
| id      | stuno   | name   | age  | classId |
+---------+---------+--------+------+---------+
| 3523633 | 3455655 | oQmLUr | 19   | 39      |
+---------+---------+--------+------+---------+
1 row in set (2.09 sec)

mysql> SELECT * FROM student WHERE name = 'oQmLUr';
+---------+---------+--------+------+---------+
| id      | stuno   | name   | age  | classId |
+---------+---------+--------+------+---------+
| 1154002 | 1243200 | OQMlUR | 266  | 28      |
| 1405708 | 1437740 | OQMlUR | 245  | 439     |
```

```
| 1748070 | 1680092 | OQMlUR | 240   | 414      |
| 2119892 | 2051914 | oQmLUr | 17    | 32       |
| 2893154 | 2825176 | OQMlUR | 245   | 435      |
| 3523633 | 3455655 | oQmLUr | 19    | 39       |
+---------+---------+--------+------+---------+
6 rows in set (2.39 sec)
```

从上面的结果可以看出来，查询学生编号为“3455655”的学生信息花费时间为2.09秒。查询学生姓名为“oQmLUr”的学生信息花费时间为2.39秒。已经达到了秒的数量级，说明目前查询效率是比较低的，下面的小节我们分析一下原因。

**2. 先查看下慢查询的记录**

```
mysql> show status like 'slow_queries';
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| Slow_queries  | 2     |
+---------------+-------+
1 row in set (0.01 sec)
```

> 🎯补充说明：
>
> 在Mysql中，除了上述变量，控制慢查询日志的还有另外一个变量 min_examined_row_limit 。这个变量的意思是，查询`扫描过的最少记录数`。这个变量和查询执行时间，共同组成了判别一个查询是否慢查询的条件。如果查询扫描过的记录数大于等于这个变量的值，并且查询执行时间超过 long_query_time 的值，那么这个查询就被记录到慢查询日志中。反之，则不被记录到慢查询日志中。另外，min_examined_row_limit 默认是 0，我们也一般不会去修改它。
>
> ```
> mysql> SHOW VARIABLES like 'min%';
> +------------------------+-------+
> | Variable_name          | Value |
> +------------------------+-------+
> | min_examined_row_limit | 0     |
> +------------------------+-------+
> 1 row in set (0.02 sec)
> ```
>
> 当这个值为默认值0时，与 long_query_time=10合在一起，表示只要查询的执行时间超过10秒钟，哪怕一个记录也没有扫描过，都要被记录到慢查询日志中。你也可以根据需要，通过修改"my.ini"文件，来修改查询时长，或者通过SET指令，用SQL语句修改`min_examined_row_limit`的值。

## 4.5 慢查询日志分析工具：mysqldumpslow

在生产环境中，如果要手工分析日志，查找、分析SQL，显然是个体力活，MySQL提供了日志分析工具`mysqldumpslow`。

> 📄注意:
> 1.该工具并不是 MySQL 内置的，不要在 MySQL 下执行，可以直接在根目录或者其他位置执行
> 2.该工具只有 Linux 下才是开箱可用的，实际上生产中mysql数据库一般也是部署在linux环境中的。如果您是windows环境下，可以参考博客https://www.cnblogs.com/-mrl/p/15770811.html。

查看mysqldumpslow的帮助信息

```
mysqldumpslow --help
```

```
[root@centos01 ~]# mysqldumpslow --help
Usage: mysqldumpslow [ OPTS... ] [ LOGS... ]

Parse and summarize the MySQL slow query log. Options are

  --verbose    verbose
  --debug      debug
  --help       write this text to standard output

  -v           verbose
  -d           debug
  -s ORDER     what to sort by (al, at, ar, c, l, r, t), 'at' is default
                al: average lock time
                ar: average rows sent
                at: average query time
                 c: count
                 l: lock time
                 r: rows sent
                 t: query time
  -r           reverse the sort order (largest last instead of first)
  -t NUM       just show the top n queries
  -a           don't abstract all numbers to N and strings to 'S'
  -n NUM       abstract numbers with at least n digits within names
  -g PATTERN   grep: only consider stmts that include this string
  -h HOSTNAME  hostname of db server for *-slow.log filename (can be wildcard),
               default is '*', i.e. match all
  -i NAME      name of server instance (if using mysql.server startup script)
  -l           don't subtract lock time from total time
```

mysqldumpslow 命令的具体参数如下：

- -a: 不将数字抽象成N，字符串抽象成S
- -s: 是表示按照何种方式排序：
  - c: 访问次数
  - l: 锁定时间
  - r: 返回记录
  - **t:查询时间**
  - al:平均锁定时间
  - ar:平均返回记录数
  - at:平均查询时间 （默认方式）
  - ac:平均查询次数
- -t: 即为返回前面多少条的数据；
- -g: 后边搭配一个正则匹配模式，大小写不敏感的；

接下来我们可以找到慢查询日志的位置

```
[root@hadoop102 ~]# cd /var/lib/mysql
[root@hadoop102 mysql]# ll
总用量 902948
drwxr-x---. 2 mysql mysql      4096 7月  12 23:04 atguigudb
drwxr-x---. 2 mysql mysql      4096 8月  11 17:08 atguigudb1
-rw-r-----. 1 mysql mysql        56 5月   9 21:36 auto.cnf
-rw-r-----. 1 mysql mysql       156 7月   9 13:05 binlog.000006
-rw-r-----. 1 mysql mysql       179 7月   9 23:45 binlog.000007
-rw-r-----. 1 mysql mysql     24481 7月  13 01:29 binlog.000008
-rw-r-----. 1 mysql mysql       179 7月  13 23:33 binlog.000009
-rw-r-----. 1 mysql mysql       179 7月  14 14:25 binlog.000010
-rw-r-----. 1 mysql mysql 647260544 8月  11 17:16 binlog.000011
-rw-r-----. 1 mysql mysql        96 8月   7 15:45 binlog.index
-rw-------. 1 mysql mysql      1676 5月   9 21:36 ca-key.pem
-rw-r--r--. 1 mysql mysql      1112 5月   9 21:36 ca.pem
-rw-r--r--. 1 mysql mysql      1112 5月   9 21:36 client-cert.pem
-rw-------. 1 mysql mysql      1680 5月   9 21:36 client-key.pem
drwxr-x---. 2 mysql mysql      4096 8月   8 18:33 dbtest2
-rw-r-----. 1 mysql mysql       909 8月  11 17:23 hadoop102-slow.log
-rw-r-----. 1 mysql mysql    196608 8月  11 17:16 #ib_16384_0.dblwr
```

举例：我们想要按照查询时间排序，查看前五条 SQL 语句，这样写即可：

```
mysqldumpslow -s t -t 5 /var/lib/mysql/atguigu01-slow.log
```

```
[root@bogon ~]# mysqldumpslow -s t -t 5 /var/lib/mysql/atguigu01-slow.log
Reading mysql slow query log from /var/lib/mysql/atguigu01-slow.log
Count: 1 Time=2.39s (2s) Lock=0.00s (0s) Rows=13.0 (13), root[root]@localhost
SELECT * FROM student WHERE name = 'S'
Count: 1 Time=2.09s (2s) Lock=0.00s (0s) Rows=2.0 (2), root[root]@localhost
SELECT * FROM student WHERE stuno = N
Died at /usr/bin/mysqldumpslow line 162, <> chunk 2.
```

举例：我们想要按照查询时间排序，查看前五条 SQL 语句，这样写即可：

```
[root@hadoop102 mysql]# mysqldumpslow -s t -t 5 /var/lib/mysql/hadoop102-slow.log

Reading mysql slow query log from /var/lib/mysql/hadoop102-slow.log
Count: 1  Time=283.29s (283s)  Lock=0.00s (0s)  Rows=0.0 (0), root[root]@hadoop102
  CALL insert_stu1(N,N)

Count: 1  Time=1.09s (1s)  Lock=0.00s (0s)  Rows=5.0 (5), root[root]@localhost
  SELECT * FROM student WHERE name = 'S'

Count: 1  Time=1.03s (1s)  Lock=0.00s (0s)  Rows=1.0 (1), root[root]@localhost
  SELECT * FROM student WHERE stuno = N

Died at /usr/bin/mysqldumpslow line 162, <> chunk 3.
```

可以看到上面 sql 中具体的数值类都被N代替，字符串都被使用 S 代替，如果想要显示真实的数据，可以加上参数 `-a`

```
[root@hadoop102 mysql]# mysqldumpslow -a -s t -t 5 /var/lib/mysql/hadoop102-slow.log

Reading mysql slow query log from /var/lib/mysql/hadoop102-slow.log
Count: 1  Time=283.29s (283s)  Lock=0.00s (0s)  Rows=0.0 (0), root[root]@hadoop102
  CALL insert_stu1(100001,4000000)

Count: 1  Time=1.09s (1s)  Lock=0.00s (0s)  Rows=5.0 (5), root[root]@localhost
  SELECT * FROM student WHERE name = 'ZfCwDz'

Count: 1  Time=1.03s (1s)  Lock=0.00s (0s)  Rows=1.0 (1), root[root]@localhost
  SELECT * FROM student WHERE stuno = 3455655

Died at /usr/bin/mysqldumpslow line 162, <> chunk 3.
```

**工作常用参考：**

```
#得到返回记录集最多的10个SQL
mysqldumpslow -s r -t 10 /var/lib/mysql/atguigu-slow.log
#得到访问次数最多的10个SQL
mysqldumpslow -s c -t 10 /var/lib/mysql/atguigu-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" /var/lib/mysql/atguigu-slow.log
#另外建议在使用这些命令时结合 | 和more 使用 ，否则有可能出现爆屏情况
mysqldumpslow -s r -t 10 /var/lib/mysql/atguigu-slow.log | more
```

## 4.6 关闭慢查询日志

MySQL服务器停止慢查询日志功能有两种方法：

**方式1：永久性方式**

修改my.cnf或my.ini文件，把【mysqld】组下的slow_query_log值设置为OFF，修改保存后，再重启MySQL服务，即可生效。

```
[mysqld]
slow_query_log=OFF
```

或者，把slow_query_log一项注释掉 或 删除

```
[mysqld]
#slow_query_log =OFF
```

重启MySQL服务，执行如下语句查询慢日志功能。

```
SHOW VARIABLES LIKE '%slow%'; #查询慢查询日志所在目录
SHOW VARIABLES LIKE '%long_query_time%'; #查询超时时长
```

可以看到，MySQL系统中的慢查询日志是关闭的。

**方式2：临时性方式**

使用SET语句来设置。

1. 停止MySQL慢查询日志功能，具体SQL语句如下。

```
SET GLOBAL slow_query_log=off;
```

2. **重启MySQL服务**，使用SHOW语句查询慢查询日志功能信息，具体SQL语句如下

```
SHOW VARIABLES LIKE '%slow%';
#以及
SHOW VARIABLES LIKE '%long_query_time%';
```

```
[root@hadoop102 mysql]# systemctl restart mysqld;
[root@hadoop102 mysql]# mysql -hlocalhost -P3306 -uroot -p
Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 10
Server version: 8.0.25 MySQL Community Server - GPL

Copyright (c) 2000, 2021, Oracle and/or its affiliates.
mysql> SHOW VARIABLES LIKE '%slow%';
+---------------------------+-----------------------------------+
| Variable_name             | Value                             |
+---------------------------+-----------------------------------+
| log_slow_admin_statements | OFF                               |
| log_slow_extra            | OFF                               |
| log_slow_slave_statements | OFF                               |
| slow_launch_time          | 2                                 |
| slow_query_log            | OFF  #慢查询日志已关闭               |
| slow_query_log_file       | /var/lib/mysql/hadoop102-slow.log |
+---------------------------+-----------------------------------+
6 rows in set (0.00 sec)

mysql> SHOW VARIABLES LIKE '%long_query_time%';
+-----------------+-----------+
| Variable_name   | Value     |
+-----------------+-----------+
| long_query_time | 10.000000 | #已恢复至默认的 10s
+-----------------+-----------+
1 row in set (0.01 sec)
```

## 4.7删除慢查询日志

使用SHOW语句显示慢查询日志信息，具体SQL语句如下：

```
mysql> SHOW VARIABLES LIKE '%slow_query_log%';
+---------------------+-----------------------------------+
| Variable_name       | Value                             |
+---------------------+-----------------------------------+
| slow_query_log      | ON                                |
| slow_query_log_file | /var/lib/mysql/hadoop102-slow.log |
+---------------------+-----------------------------------+
2 rows in set (0.00 sec)
```

```
mysql> SHOW VARIABLES LIKE 'slow_query_log%';
+---------------------+-----------------------------------+
| Variable_name       | Value                             |
+---------------------+-----------------------------------+
| slow_query_log      | OFF                               |
| slow_query_log_file | /var/lib/mysql/atguigu01-slow.log |
+---------------------+-----------------------------------+
2 rows in set (0.00 sec)
```

从执行结果可以看出，慢查询日志的目录默认为MySQL的数据目录，在该目录下`手动删除慢查询日志文件`即可。

使用命令`mysqladmin flush-logs`来重新生成查询日志文件，具体命令如下，执行完毕会在数据目录下重新成慢查询日志文件。

调优结束可以及时删除慢查询日志节省磁盘空间哟，当然手工删除也是可以的

```
[root@hadoop102 mysql]# rm hadoop102-slow.log
rm: 是否删除普通文件 "hadoop102-slow.log"? y
[root@hadoop102 mysql]# ll
总用量 902932
drwxr-x---. 2 mysql mysql      4096 7月  12 23:04 atguigudb
drwxr-x---. 2 mysql mysql      4096 8月  11 17:08 atguigudb1
-rw-r-----. 1 mysql mysql        56 5月   9 21:36 auto.cnf
-rw-r-----. 1 mysql mysql       179 7月  13 23:33 binlog.000009
-rw-r-----. 1 mysql mysql       179 7月  14 14:25 binlog.000010
```

如果误删了，而且还没有了备份，可以使用下面的命令来重新恢复生成哟，执行完毕后会在数据目录下重新生成查询日志文件

```
#先要打开慢查询日志
SET GLOBAL slow_query_log=ON;
#恢复慢查询日志
mysqladmin -uroot -p flush-logs slow
```

> 提示
>
> 慢查询日志都是使用`mysqladmin -uroot -p flush-logs slow`命令来删除重建的。使用时一定要注意，一旦执行了这个命令，慢查询日志都只存在于新的日志文件中，如果需要旧的查询日志，就必须事先备份

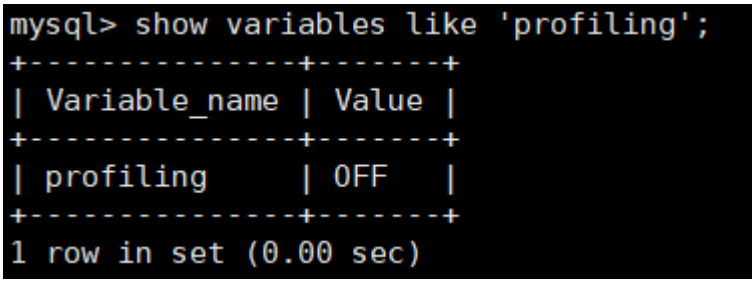
# 5. 查看 SQL 执行成本：SHOW PROFILE

show profile在《逻辑架构》章节中讲过，这里作为复习。

Show Profile是MySQL提供的可以用来分析当前会话中SQL都做了什么、执行的资源消耗情况的工具，可用于sql调优的测量。`默认情况下处于关闭状态`，并保存最近15次的运行结果。

我们可以在会话级别开启这个功能

```
mysql > show variables like 'profiling';
```




然后执行相关的查询语句。接着看下当前会话下有哪些profiles

```
mysql> SELECT * FROM student WHERE stuno = 3455655;
//...
mysql> SELECT * FROM student WHERE name = 'ZfCwDz';
//...
mysql> show profiles;
+----------+------------+-------------------------------------------+
| Query_ID | Duration   | Query                                     |
+----------+------------+-------------------------------------------+
|        1 | 0.00133475 | show variables like 'profiling'           |
|        2 | 0.00021050 | SELECT * FROM student WHERE stuno = 3455655 |
|        3 | 0.00053600 | SELECT DATABASE()                         |
|        4 | 0.01693325 | show databases                            |
|        5 | 0.00375125 | show tables                               |
|        6 | 1.75597875 | SELECT * FROM student WHERE stuno = 3455655 |
|        7 | 1.11115150 | SELECT * FROM student WHERE name = 'ZfCwDz' |
+----------+------------+-------------------------------------------+
7 rows in set, 1 warning (0.00 sec)
```

你能看到当前会话一共有7个查询，如果我们想要查看最近一次查询的开销，可以使用

```
show profile;
```

```
mysql> show profile;
+--------------------------------+----------+
| Status                         | Duration |
+--------------------------------+----------+
| starting                       | 0.000072 |
| Executing hook on transaction  | 0.000003 |
| starting                       | 0.000007 |
| checking permissions           | 0.000005 |
| Opening tables                 | 0.000036 |
| init                           | 0.000004 |
| System lock                    | 0.000007 |
| optimizing                     | 0.000044 |
| statistics                     | 0.000020 |
| preparing                      | 0.000018 |
| executing                      | 1.110858 |
| end                            | 0.000017 |
| query end                      | 0.000005 |
| waiting for handler commit     | 0.000011 |
| closing tables                 | 0.000011 |
| freeing items                  | 0.000023 |
| cleaning up                    | 0.000011 |
+--------------------------------+----------+
17 rows in set, 1 warning (0.00 sec)
```

我们也可以查看指定的Query ID的开销，只需要后面跟上 `for num`。也可以查看不同部分的开销，比如CPU、block.io等

```
1  show profile cpu,block io for query 7;
```

```
mysql> show profile cpu,block io for query 7;
+--------------------------------+----------+----------+------------+--------------+---------------+
| Status                         | Duration | CPU_user | CPU_system | Block_ops_in | Block_ops_out |
+--------------------------------+----------+----------+------------+--------------+---------------+
| starting                       | 0.000072 | 0.000066 |   0.000000 |            0 |             0 |
| Executing hook on transaction  | 0.000003 | 0.000003 |   0.000000 |            0 |             0 |
| starting                       | 0.000007 | 0.000007 |   0.000000 |            0 |             0 |
| checking permissions           | 0.000005 | 0.000005 |   0.000000 |            0 |             0 |
| Opening tables                 | 0.000036 | 0.000037 |   0.000000 |            0 |             0 |
| init                           | 0.000004 | 0.000003 |   0.000000 |            0 |             0 |
| System lock                    | 0.000007 | 0.000007 |   0.000000 |            0 |             0 |
| optimizing                     | 0.000044 | 0.000045 |   0.000000 |            0 |             0 |
| statistics                     | 0.000020 | 0.000019 |   0.000000 |            0 |             0 |
| preparing                      | 0.000018 | 0.000018 |   0.000000 |            0 |             0 |
| executing                      | 1.110858 | 1.147034 |   0.000000 |            0 |             0 |
| end                            | 0.000017 | 0.000010 |   0.000000 |            0 |             0 |
| query end                      | 0.000005 | 0.000004 |   0.000000 |            0 |             0 |
| waiting for handler commit     | 0.000011 | 0.000011 |   0.000000 |            0 |             0 |
| closing tables                 | 0.000011 | 0.000011 |   0.000000 |            0 |             0 |
| freeing items                  | 0.000023 | 0.000024 |   0.000000 |            0 |             0 |
| cleaning up                    | 0.000011 | 0.000011 |   0.000000 |            0 |             0 |
+--------------------------------+----------+----------+------------+--------------+---------------+
17 rows in set, 1 warning (0.00 sec)
```

通过如果发现上一条 sql 慢的原因在于执行慢（`executing`字段耗时多），就可以接着用 `Explain` 进行分析具体的 sql 语句。等后面我们为其建立索引，就可以大大提高效率了

通过设置`profiling='ON'`来开启 show profile：

```
1  mysql > set profiling = 'ON';
```

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

mysql> show variables like 'profiling';
+---------------+-------+
| Variable_name | Value |
+---------------+-------+
| profiling     | ON    |
+---------------+-------+
1 row in set (0.00 sec)
```

然后执行相关的查询语句。接着看下当前会话都有哪些 profiles，使用下面这条命令：

```
mysql > show profiles;
```

```
mysql> show profiles;
+----------+------------+---------------------------------+
| Query_ID | Duration   | Query                           |
+----------+------------+---------------------------------+
|        1 | 0.00107850 | show variables like 'profiling' |
|        2 | 0.00292675 | select * from employees         |
+----------+------------+---------------------------------+
2 rows in set, 1 warning (0.00 sec)
```

你能看到当前会话一共有 2 个查询。如果我们想要查看最近一次查询的开销，可以使用：

```
mysql > show profile;
```

```
mysql> show profile;
+----------------------------------+----------+
| Status                           | Duration |
+----------------------------------+----------+
| starting                         | 0.000102 |
| Executing hook on transaction    | 0.000003 |
| starting                         | 0.000007 |
| checking permissions             | 0.000006 |
| Opening tables                   | 0.000029 |
| init                             | 0.000004 |
| System lock                      | 0.000007 |
| optimizing                       | 0.000003 |
| statistics                       | 0.000013 |
| preparing                        | 0.000011 |
| executing                        | 0.002616 |
| end                              | 0.000008 |
| query end                        | 0.000004 |
| waiting for handler commit       | 0.000006 |
| closing tables                   | 0.000007 |
| freeing items                    | 0.000090 |
| cleaning up                      | 0.000013 |
+----------------------------------+----------+
17 rows in set, 1 warning (0.00 sec)
```

我们也可以查看指定的Query ID的开销，比如`show profile for query`2查询结果是一样的。在SHOW PROFILE中我们可以查看不同部分的开销，比如cpu、block.io等：

```
1 mysql> show profile cpu,block io for query 2;
```

```
mysql> show profile cpu,block io for query 2;
+------------------------------+----------+----------+------------+-------------+--------------+
| Status                       | Duration | CPU_user | CPU_system | Block_ops_in | Block_ops_out |
+------------------------------+----------+----------+------------+-------------+--------------+
| starting                     | 0.000102 | 0.000018 |   0.000057 |            0 |            0 |
| Executing hook on transaction | 0.000003 | 0.000000 |   0.000002 |            0 |            0 |
| starting                     | 0.000007 | 0.000002 |   0.000006 |            0 |            0 |
| checking permissions         | 0.000006 | 0.000001 |   0.000004 |            0 |            0 |
| Opening tables               | 0.000029 | 0.000007 |   0.000023 |            0 |            0 |
| init                         | 0.000004 | 0.000001 |   0.000003 |            0 |            0 |
| System lock                  | 0.000007 | 0.000002 |   0.000005 |            0 |            0 |
| optimizing                   | 0.000003 | 0.000001 |   0.000002 |            0 |            0 |
| statistics                   | 0.000013 | 0.000003 |   0.000010 |            0 |            0 |
| preparing                    | 0.000011 | 0.000002 |   0.000008 |            0 |            0 |
| executing                    | 0.002616 | 0.000341 |   0.001084 |          288 |            0 |
| end                          | 0.000008 | 0.000001 |   0.000004 |            0 |            0 |
| query end                    | 0.000004 | 0.000001 |   0.000003 |            0 |            0 |
| waiting for handler commit   | 0.000006 | 0.000001 |   0.000005 |            0 |            0 |
| closing tables               | 0.000007 | 0.000002 |   0.000004 |            0 |            0 |
| freeing items                | 0.000090 | 0.000000 |   0.000091 |            0 |            0 |
| cleaning up                  | 0.000013 | 0.000000 |   0.000012 |            0 |            0 |
+------------------------------+----------+----------+------------+-------------+--------------+
17 rows in set, 1 warning (0.00 sec)
```

通过上面的结果，我们可以弄清楚每一步骤的耗时，以及在不同部分，比如CPU、block.io的执行时间，这样我们就可以判断出来SQL到底慢在哪里。

**show profile的常用查询参数：**

1. ALL：显示所有的开销信息。
2. BLOCK IO：显示块IO开销。

3. CONTEXT SWITCHES：上下文切换开销。
4. CPU：显示CPU开销信息。
5. IPC：显示发送和接收开销信息。
6. MEMORY：显示内存开销信息。
7. PAGE FAULTS：显示页面错误开销信息。
8. SOURCE：显示和Source_function，Source_file， Source_line相关的开销信息。
9. SWAPS：显示交换次数开销信息。

日常开发需注意的结论：

1. `converting HEAP to MyISAM`：查询结果太大，内存不够，数据往磁盘上搬了。
2. `Creating tmp table`：创建临时表。先拷贝数据到临时表，用完后再删除临时表。
3. `Copying to tmp table on disk`：把内存中临时表复制到磁盘上，警惕！
4. `locked`。

如果在show profile诊断结果中出现了以上4条结果中的任何一条，则sql语句需要优化。

注意：

不过SHOW PROFILE命令将被弃用，我们可以从information_schema中的profiling数据表进行查看。

# 6. 分析查询语句：EXPLAIN

## 6.1 概述

**定位了查询慢的SQL之后，我们就可以使用EXPLAIN或DESCRIBE工具做针对性的分析查询语句。**
DESCRIBE语句的使用方法与EXPLAIN语句是一样的，并且分析结果也是一样的。

MySQL中有专门负责优化SELECT语句的优化器模块，主要功能：通过计算分析系统中收集到的统计信息，为客户端请求的Query提供它认为最优的`执行计划`（他认为最优的数据检索方式，但不见得是DBA认为是最优的，这部分最耗费时间）。

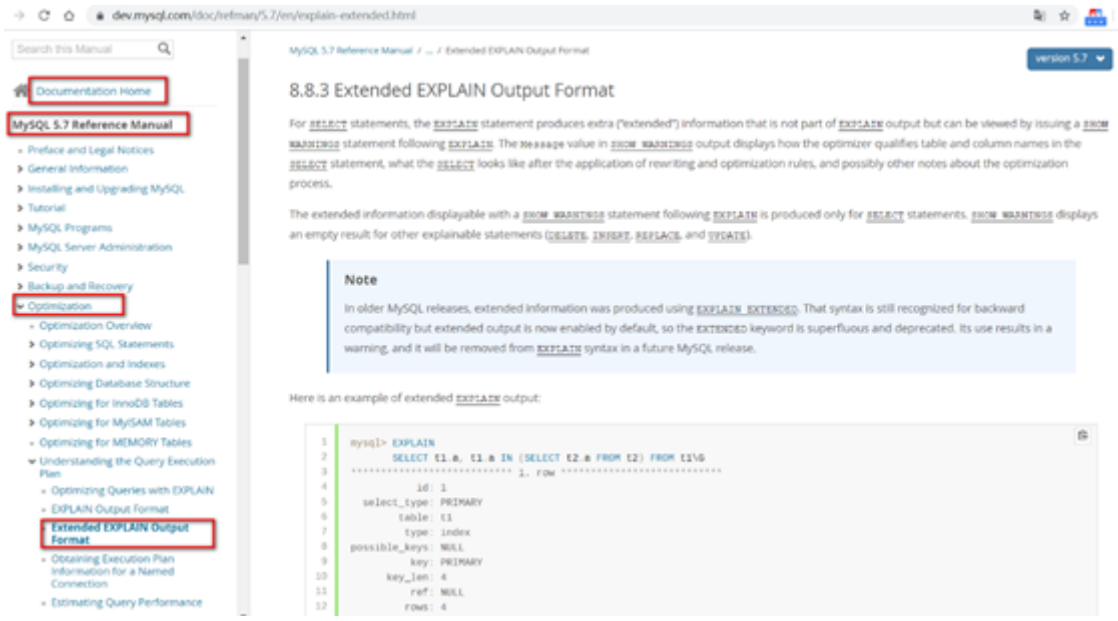
这个执行计划展示了接下来具体执行查询的方式，比如多表连接的顺序是什么，对于每个表采用什么访问方法来具体执行查询等等。MySQL为我们提供了`EXPLAIN`语句来帮助我们查看某个查询语句的具体执行计划，大家看懂`EXPLAIN`语句的各个输出项，可以有针对性的提升我们查询语句的性能。

1. **能做什么?**

- 表的读取顺序
- 数据读取操作的操作类型
- 哪些索引可以使用
- **哪些索引被实际使用**
- 表之间的引用
- **每张表有多少行被优化器查询**

2. **官网介绍**

https://dev.mysql.com/doc/refman/5.7/en/explain-output.html
https://dev.mysql.com/doc/refman/8.0/en/explain-output.html




4. **版本情况**

- MySQL 5.6.3以前只能 `EXPLAIN SELECT` ； MYSQL 5.6.3以后就可以 `EXPLAIN SELECT，UPDATE，DELETE`
- 在5.7以前的版本中，想要显示 partitions 需要使用`partitions`命令；想要显示`filtered`需要使用`explain extended`命令。在5.7版本后，默认explain直接显示partitions和filtered中的信息。

```
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------+
```

**注意：**EXPLAIN 仅仅是查看执行计划，不会真实的执行 sql

## 6.2 基本语法

EXPLAIN 或 DESCRIBE语句的语法形式如下：

```
EXPLAIN SELECT select_options
#或者
DESCRIBE SELECT select_options
```

如果我们想看看某个查询的执行计划的话，可以在具体的查询语句前边加一个 `EXPLAIN`，就像这样：

```
mysql> EXPLAIN SELECT 1;
```

```
mysql> EXPLAIN SELECT 1;
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra          |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------+
|  1 | SIMPLE      | NULL  | NULL       | NULL | NULL          | NULL | NULL    | NULL | NULL |     NULL | No tables used |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------+
1 row in set, 1 warning (0.00 sec)
```

输出的上述信息就是所谓的`执行计划`。在这个执行计划的辅助下，我们需要知道应该怎样改进自己的查询语句以使查询执行起来更高效。其实除了以`SELECT`开头的查询语句，其余的`DELETE`、`INSERT`、`REPLACE`以及`UPDATE`语句等都可以加上`EXPLAIN`，用来查看这些语句的执行计划，只是平时我们对`SELECT`语句更感兴趣。

注意：执行EXPLAIN时并没有真正的执行该后面的语句，因此可以安全的查看执行计划。

`EXPLAIN`语句输出的各个列的作用如下：

| 列名 | 描述 |
| --- | --- |
| id | 在一个大的查询语句中每个SELECT关键字都对应一个 唯一的id |
| select_type | SELECT关键字对应的那个查询的类型 |
| table | 表名 |
| partitions | 匹配的分区信息 |
| type | 针对单表的访问方法 |
| possible_keys | 可能用到的索引 |
| key | 实际上使用的索引 |
| key_len | 实际使用到的索引长度 |
| ref | 当使用索引列等值查询时，与索引列进行等值匹配的对象信息 |
| rows | 预估的需要读取的记录条数 |
| filtered | 某个表经过搜索条件过滤后剩余记录条数的百分比 |
| Extra | 一些额外的信息 |

在这里把它们都列出来只是为了描述一个轮廓，让大家有一个大致的印象。

## 6.3 数据准备

**1. 建表**

```
CREATE TABLE s1 (
    id INT AUTO_INCREMENT,
    key1 VARCHAR(100),
    key2 INT,
    key3 VARCHAR(100),
    key_part1 VARCHAR(100),
    key_part2 VARCHAR(100),
    key_part3 VARCHAR(100),
    common_field VARCHAR(100),
    PRIMARY KEY (id),
    INDEX idx_key1 (key1),
    UNIQUE INDEX idx_key2 (key2),
    INDEX idx_key3 (key3),
    INDEX idx_key_part(key_part1, key_part2, key_part3)
) ENGINE=INNODB CHARSET=utf8;
```

```
CREATE TABLE s2 (
    id INT AUTO_INCREMENT,
    key1 VARCHAR(100),
    key2 INT,
    key3 VARCHAR(100),
    key_part1 VARCHAR(100),
    key_part2 VARCHAR(100),
    key_part3 VARCHAR(100),
```

```
    common_field VARCHAR(100),
    PRIMARY KEY (id),
    INDEX idx_key1 (key1),
    UNIQUE INDEX idx_key2 (key2),
    INDEX idx_key3 (key3),
    INDEX idx_key_part(key_part1, key_part2, key_part3)
) ENGINE=INNODB CHARSET=utf8;
```

**2.设置参数log_bin_trust_function_creators**

创建函数，假如报错，需开启如下命令：允许创建函数设置：

```
set global log_bin_trust_function_creators=1; # 不加global只是当前窗口有效。
```

**3.创建函数**

```
DELIMITER //
CREATE FUNCTION rand_string1(n INT)
    RETURNS VARCHAR(255) #该函数会返回一个字符串
BEGIN
    DECLARE chars_str VARCHAR(100) DEFAULT
'abcdefghijklmnopqrstuvwxyzABCDEFJHIJKLMNOPQRSTUVWXYZ';
    DECLARE return_str VARCHAR(255) DEFAULT '';
    DECLARE i INT DEFAULT 0;
    WHILE i < n DO
        SET return_str
=CONCAT(return_str,SUBSTRING(chars_str,FLOOR(1+RAND()*52),1));
        SET i = i + 1;
    END WHILE;
    RETURN return_str;
END //
DELIMITER ;
```

**4.创建存储过程**

创建往s1表中插入数据的存储过程：

```
DELIMITER //
CREATE PROCEDURE insert_s1 (IN min_num INT (10),IN max_num INT (10))
BEGIN
    DECLARE i INT DEFAULT 0;
    SET autocommit = 0;
    REPEAT
    SET i = i + 1;
    INSERT INTO s1 VALUES(
    (min_num + i),
    rand_string1(6),
    (min_num + 30 * i + 5),
    rand_string1(6),
    rand_string1(10),
    rand_string1(5),
    rand_string1(10),
    rand_string1(10));
    UNTIL i = max_num
    END REPEAT;
```

```
    COMMIT;
END //
DELIMITER ;
```

创建往s2表中插入数据的存储过程：

```
DELIMITER //
CREATE PROCEDURE insert_s2 (IN min_num INT (10),IN max_num INT (10))
BEGIN
    DECLARE i INT DEFAULT 0;
    SET autocommit = 0;
    REPEAT
    SET i = i + 1;
    INSERT INTO s2 VALUES(
        (min_num + i),
        rand_string1(6),
        (min_num + 30 * i + 5),
        rand_string1(6),
        rand_string1(10),
        rand_string1(5),
        rand_string1(10),
        rand_string1(10));
    UNTIL i = max_num
    END REPEAT;
    COMMIT;
END //
DELIMITER ;
```

**5.调用存储过程**

s1表数据的添加：加入1万条记录：

```
CALL insert_s1(10001,10000);
```

s2表数据的添加：加入1万条记录：

```
CALL insert_s2(10001,10000);
```

## 6.4 EXPLAIN各列作用

为了让大家有比较好的体验，我们调整了下`EXPLAIN`输出列的顺序。

### 1. table

查询的每一行记录都对应着一个单表

不论我们的查询语句有多复杂，里边儿`包含了多少个表`，到最后也是需要对每个表进行`单表访问`的，所以MySQL规定**EXPLAIN语句输出的每条记录都对应着某个单表的访问方法**，该条记录的table列代表着该表的表名（有时不是真实的表名字，可能是简称）。

```
EXPLAIN SELECT * FROM s1 INNER JOIN s2;
```

如下图，一张表对应一个记录。

```
mysql> EXPLAIN SELECT * FROM s1 INNER JOIN s2;
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra                      |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------------------+
|  1 | SIMPLE      | s1    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 9895 |   100.00 | NULL                       |
|  1 | SIMPLE      | s2    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 9895 |   100.00 | Using join buffer (hash join) |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------------------+
2 rows in set, 1 warning (0.00 sec)
```

注：临时表也会有对应的记录，比如我们使用UNION时就会出现临时表

查询的每一行记录都对应着一个单表

```
mysql> EXPLAIN SELECT * FROM s1;
```

```
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------+
|  1 | SIMPLE      | s1    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 9688 |   100.00 | NULL  |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------+
1 row in set, 1 warning (0.00 sec)
```

当然，连接查询也算是`SIMPLE`类型，比如：

```
mysql> EXPLAIN SELECT * FROM s1 INNER JOIN s2;#第一个是驱动表，第二个是被驱动表
```

```
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------------------------------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra                                        |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------------------------------------+
|  1 | SIMPLE      | s1    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 9688 |   100.00 | NULL                                         |
|  1 | SIMPLE      | s2    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 9954 |   100.00 | Using join buffer (Block Nested Loop)        |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------------------------------------+
2 rows in set, 1 warning (0.01 sec)
```

## 2. id

在一个大的查询语句中每个SELECT关键字都对应一个唯一的id

我们写的查询语句一般都以`SELECT`关键字开头，比较简单的查询语句里只有一个`SELECT`关键字，比如下边这个查询语句：

```
SELECT * FROM s1 WHERE key1 = 'a';
```

稍微复杂一点的连接查询中也只有一个`SELECT`关键字，比如：

```
SELECT * FROM s1 INNER JOIN s2
ON s1.key1 = s2.key1
WHERE s1.common_field = 'a';
```

但是下边两种情况下在一条查询语句中会出现多个`SELECT`关键字：

- 查询中包含子查询的情况

  比如下边这个查询语句中就包含2个`SELECT`关键字：

  ```
  SELECT * FROM s1 WHERE key1 IN (SELECT key3 FROM s2);
  ```

- 查询中包含UNION语句的情况
  比如下边这个查询语句中也包含2个`SELECT`关键字：

  ```
  SELECT * FROM s1 UNION SELECT * FROM s2;
  ```

查询语句中每出现一个SELECT关键字，MySQL就会为它分配一个唯一的id值。这个id值就是EXPLAIN语句的第一个列，比如下边这个查询中只有一个 SELECT关键字，所以 EXPLAIN的结果中也就只有一条id列为1的记录：

```
EXPLAIN SELECT * FROM s1 WHERE key1 = 'a';
```

```
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------+
| id | select_type | table | partitions | type | possible_keys | key      | key_len | ref   | rows | filtered | Extra |
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------+
|  1 | SIMPLE      | s1    | NULL       | ref  | idx_key1      | idx_key1 | 303     | const |    8 |   100.00 | NULL  |
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------+
1 row in set, 1 warning (0.03 sec)
```

对于连接查询来说，一个`SELECT`关键字后边的`FROM`子句中可以跟随多个表，所以在连接查询的执行计划中，每个表都会对应一条记录，但是这些记录的id值都是相同的，比如：

```
mysql> EXPLAIN SELECT * FROM s1 INNER JOIN s2;
```

```
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------------------------------------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra                                              |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------------------------------------------+
|  1 | SIMPLE      | s1    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 9688 |   100.00 | NULL                                               |
|  1 | SIMPLE      | s2    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 9954 |   100.00 | Using join buffer (Block Nested Loop) |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------------------------------------------+
2 rows in set, 1 warning (0.01 sec)
```

可以看到，上述连接查询中参与连接的s1和s2表分别对应一条记录，但是这两条记录对应的`id`值都是`1`。这里需要大家记住的是，**在连接查询的执行计划中，每个表都会对应一条记录，这些记录的id列的值是相同的**，出现在前边的表表示`驱动表`，出现在后边的表表示`被驱动表`。所以从上边的EXPLAIN输出中我们可以看出，查询优化器准备让s1表作为驱动表，让s2表作为被驱动表来执行查询。

> 在join连接查询中，驱动表在SQL语句执行的过程中总是先被读取。而被驱动表在SQL语句执行的过程中总是后被读取。
>
> 在读取驱动表数据后，放入到join_buffer后，再去读取被驱动表中的数据来和驱动表中的数据进行匹配。如果匹配成功,就返回结果,否则该丢弃, 继续匹配下一条

对于包含子查询的查询语句来说，就可能涉及多个`SELECT`关键字，所以在**包含子查询的查询语句的执行计划中，每个SELECT关键字都会对应一个唯一的id值**，比如这样：

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 IN (SELECT key1 FROM s2) OR
key3 = 'a';
```

```
+----+-------------+-------+------------+-------+---------------+----------+---------+------+------+----------+-------------+
| id | select_type | table | partitions | type  | possible_keys | key      | key_len | ref  | rows | filtered | Extra       |
+----+-------------+-------+------------+-------+---------------+----------+---------+------+------+----------+-------------+
|  1 | PRIMARY     | s1    | NULL       | ALL   | idx_key3      | NULL     | NULL    | NULL | 9688 |   100.00 | Using where |
|  2 | SUBQUERY    | s2    | NULL       | index | idx_key1      | idx_key1 | 303     | NULL | 9954 |   100.00 | Using index |
+----+-------------+-------+------------+-------+---------------+----------+---------+------+------+----------+-------------+
2 rows in set, 1 warning (0.02 sec)
```

从输出结果中我们可以看到，s1表在外层查询中，外层查询有一个独立的`SELECT`关键字，所以第一条记录的`id`值就是`1`，s2表在子查询中，子查询有一个`独立的SELECT`关键字，所以第二条记录的`id`值就是`2`。

但是这里大家需要特别注意，**查询优化器可能对涉及子查询的查询语句进行重写，从而转换为连接查询。**所以如果我们想知道查询优化器对某个包含子查询的语句是否进行了重写，直接查看执行计划就好了，比如说：

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 IN (SELECT key2 FROM s2 WHERE common_field ='a');
```

```
+----+-------------+-------+------------+------+---------------+----------+---------+-------------------+------+----------+--------------------------+
| id | select_type | table | partitions | type | possible_keys | key      | key_len | ref               | rows | filtered | Extra                    |
+----+-------------+-------+------------+------+---------------+----------+---------+-------------------+------+----------+--------------------------+
|  1 | SIMPLE      | s2    | NULL       | ALL  | idx_key3      | NULL     | NULL    | NULL              | 9954 |    10.00 | Using where; Start temporary |
|  1 | SIMPLE      | s1    | NULL       | ref  | idx_key1      | idx_key1 | 303     | xiaohaizi.s2.key3 |    1 |   100.00 | End temporary            |
+----+-------------+-------+------------+------+---------------+----------+---------+-------------------+------+----------+--------------------------+
2 rows in set, 1 warning (0.00 sec)
```

可以看到，虽然我们的查询语句是一个子查询，但是执行计划中s1和s2表对应的记录的`id`值全部是`1`，这就表明了`查询优化器将子查询转换为了连接查询`。

对于包含`UNION`子句的查询语句来说，每个`SELECT`关键字对应一个`id`值也是没错的，不过还是有点儿特别的东西，比方说下边这个查询：

```
mysql> EXPLAIN SELECT * FROM s1 UNION SELECT * FROM s2;
```

```
+----+--------------+------------+------------+------+---------------+------+---------+------+------+----------+-----------------+
| id | select_type  | table      | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra           |
+----+--------------+------------+------------+------+---------------+------+---------+------+------+----------+-----------------+
|  1 | PRIMARY      | s1         | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 9688 |   100.00 | NULL            |
|  2 | UNION        | s2         | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 9954 |   100.00 | NULL            |
| NULL | UNION RESULT | <union1,2> | NULL       | ALL  | NULL          | NULL | NULL    | NULL | NULL |     NULL | Using temporary |
+----+--------------+------------+------------+------+---------------+------+---------+------+------+----------+-----------------+
3 rows in set, 1 warning (0.00 sec)
```

例1：下面的查询结果，两个记录似乎id都是1.这是为什么呢?

```
mysql> EXPLAIN SELECT * FROM s1 INNER JOIN s2;
```

```
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+--------------------------------------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra                                      |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+--------------------------------------------+
|  1 | SIMPLE      | s1    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 9688 |   100.00 | NULL                                       |
|  1 | SIMPLE      | s2    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 9954 |   100.00 | Using join buffer (Block Nested Loop)      |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+--------------------------------------------+
2 rows in set, 1 warning (0.01 sec)
```

实际上，在查询语句中每出现一个SELECT关键字，MySQL就会为它分配一个唯一的id，代表着一次查询。这个id 就是`EXPLAIN`语句的第一列。

例2：下面的查询中只有一个SELECT，所以`EXPLAIN`的结果中也就只有一条id为 1 的记录喽~

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 = 'a';
```

```
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------+
| id | select_type | table | partitions | type | possible_keys | key      | key_len | ref   | rows | filtered | Extra |
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------+
|  1 | SIMPLE      | s1    | NULL       | ref  | idx_key1      | idx_key1 | 303     | const |    8 |   100.00 | NULL  |
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------+
1 row in set, 1 warning (0.03 sec)
```

例3：下面的查询有两个SELECT，所以`EXPLAIN`的结果中 会有两条记录，且id分别就是1和2喽~ 。其中s1被称为驱动表，s2被称为 被驱动表

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 IN (SELECT key1 FROM s2) OR key3 = 'a';
```

```
+----+-------------+-------+------------+-------+---------------+----------+---------+------+------+----------+-------------+
| id | select_type | table | partitions | type  | possible_keys | key      | key_len | ref  | rows | filtered | Extra       |
+----+-------------+-------+------------+-------+---------------+----------+---------+------+------+----------+-------------+
|  1 | PRIMARY     | s1    | NULL       | ALL   | idx_key3      | NULL     | NULL    | NULL | 9688 |   100.00 | Using where |
|  2 | SUBQUERY    | s2    | NULL       | index | idx_key1      | idx_key1 | 303     | NULL | 9954 |   100.00 | Using index |
+----+-------------+-------+------------+-------+---------------+----------+---------+------+------+----------+-------------+
2 rows in set, 1 warning (0.02 sec)
```

例4：下面这条SQL有一个坑，请注意！！！

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 IN (SELECT key2 FROM s2 WHERE common_field= 'a');#查询优化器可能对设计子查询的查询语句进行重写，转变为多表查询操作
```

```
+----+-------------+-------+------------+------+---------------+----------+---------+-----------------+------+----------+--------------------------+
| id | select_type | table | partitions | type | possible_keys | key      | key_len | ref             | rows | filtered | Extra                    |
+----+-------------+-------+------------+------+---------------+----------+---------+-----------------+------+----------+--------------------------+
|  1 | SIMPLE      | s2    | NULL       | ALL  | idx_key3      | NULL     | NULL    | NULL            | 9954 |    10.00 | Using where; Start temporary |
|  1 | SIMPLE      | s1    | NULL       | ref  | idx_key1      | idx_key1 | 303     | xiaohaizi.s2.key3 |    1 |   100.00 | End temporary            |
+----+-------------+-------+------------+------+---------------+----------+---------+-----------------+------+----------+--------------------------+
2 rows in set, 1 warning (0.00 sec)
```

两个记录的 id 都是 1，小小的眼睛是否充满了大大的疑惑?

这是因为优化器会对上面的 sql 语句进行优化，将其转换为多表连接，而不是子查询。因为子查询其实是一种嵌套查询的情况，其时间复杂度是 O(n^m)，其中 m 是嵌套的层数，而多表查询的时间复杂度是 O(n*m)

例5：再看看 Union 联合查询的情况。

```
mysql> EXPLAIN SELECT * FROM s1 UNION SELECT * FROM s2;
```

结果是这样，竟然会出现三张表~ Amazing!

```
+----+--------------+------------+------------+------+---------------+------+---------+------+------+----------+-----------------+
| id | select_type  | table      | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra           |
+----+--------------+------------+------------+------+---------------+------+---------+------+------+----------+-----------------+
|  1 | PRIMARY      | s1         | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 9688 |   100.00 | NULL            |
|  2 | UNION        | s2         | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 9954 |   100.00 | NULL            |
| NULL | UNION RESULT | <union1,2> | NULL       | ALL  | NULL          | NULL | NULL    | NULL | NULL |     NULL | Using temporary |
+----+--------------+------------+------------+------+---------------+------+---------+------+------+----------+-----------------+
3 rows in set, 1 warning (0.00 sec)
```

这是因为 Union 是取表的并集，需要建临时表进行去重，因此会有三条记录。可以看到第三条记录的 `Extra` 就标识了它是一张临时表哦。`临时表 id 是 Null`。

例6：再看看 Union ALL：

```
mysql> EXPLAIN SELECT * FROM s1 UNION ALL SELECT * FROM s2;
```

产生两条记录，因为它不会去重~

```
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------+
|  1 | PRIMARY     | s1    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 9688 |   100.00 | NULL  |
|  2 | UNION       | s2    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 9954 |   100.00 | NULL  |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------+
2 rows in set, 1 warning (0.01 sec)
```

**小结**

- id如果相同，可以认为是一组，从上往下顺序执行
- 在所有组中，id值越大，优先级越高，越先执行

- 关注点：id号每个号码，表示一趟独立的查询，一个sql的查询趟数越少越好

## 3. select_type

SELECT关键字对应的那个查询的类型,确定小查询在整个大查询中扮演了一个什么角色

一条大的查询语句里边可以包含若干个SELECT关键字，`每个SELECT关键字代表着一个小的查询语句`，而每个SELECT关键字的FROM子句中都可以包含若干张表（这些表用来做连接查询），`每一张表都对应着执行计划输出中的一条记录`，对于在同一个SELECT关键字中的表来说，它们的id值是相同的。

MySQL为每一个SELECT关键字代表的小查询都定义了一个称之为`select_type`的属性，意思是我们只要知道了某个小查询的`select_type属性`，就知道了这个`小查询在整个大查询中扮演了一个什么角色`，我们看一下select_type都能取哪些值，请看官方文档：

| 名称 | 描述 |
| --- | --- |
| SIMPLE | Simple SELECT (not using UNION or subqueries) |
| PRIMARY | Outermost SELECT |
| UNION | Second or later SELECT statement in a UNION |
| UNION RESULT | Result of a UNION |
| SUBQUERY | First SELECT in subquery |
| DEPENDENT SUBQUERY | First SELECT in subquery, dependent on outer query |
| DEPENDENT UNION | Second or later SELECT statement in a UNION, dependent on outer query |
| DERIVED | Derived table |
| MATERIALIZED | Materialized subquery |
| UNCACHEABLE SUBQUERY | A subquery for which the result cannot be cached and must be re-evaluated for each row of the outer query |
| UNCACHEABLE UNION | The second or later select in a UNION that belongs to an uncacheable subquery (see UNCACHEABLE SUBQUERY) |

1. **查询语句中不包含`UNION`或者子查询的查询都算作是`SIMPLE`类型**

   直询语句中不包含`UNION`或者子查询的查询都算作是`SIMPLE`类型，比方说下边这个单表查询的`select_type`的值就是`SIMPLE`；

   ```
   mysql> EXPLAIN SELECT * FROM s1;
   ```

   ```
   +----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------+
   | id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra |
   +----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------+
   |  1 | SIMPLE      | s1    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 9688 |   100.00 | NULL  |
   +----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------+
   1 row in set, 1 warning (0.00 sec)
   ```

   当然，连接查询也算是SIMPLE 类型，比如：

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 = 'a';
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------+
| id | select_type | table | partitions | type | possible_keys | key      | key_len | ref   | rows | filtered | Extra |
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------+
|  1 | SIMPLE      | s1    | NULL       | ref  | idx_key1      | idx_key1 | 303     | const |    1 |   100.00 | NULL  |
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------+
1 row in set, 1 warning (0.00 sec)
```

```
1 mysql> EXPLAIN SELECT * FROM s1 INNER JOIN s2;
```

```
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------------------------------------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra                                              |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------------------------------------------+
|  1 | SIMPLE      | s1    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 9688 |   100.00 | NULL                                               |
|  1 | SIMPLE      | s2    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 9954 |   100.00 | Using join buffer (Block Nested Loop)              |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------------------------------------------+
2 rows in set, 1 warning (0.01 sec)
```

2. Union 联合查询。其左边的查询是 `Primary`，右边的查询类型是 `Union`，去重的临时表查询类型是： Union Result

   - 对于包含 `UNION` 或者 `UNION ALL` 的大查询来说，它是由几个小查询组成的，其中除了最左边的那个查询的 `select_type` 值就是 `PRIMARY`，其余的小查询的 `select_type` 值就是 `UNION`
   - `MySQL` 选择使用临时表来完成 `UNION` 查询的去重工作，针对该临时表的查询的 `select_type` 就是 `UNION RESULT`
   - 对应子查询的大查询来说，子查询是外边的那个是 `PRIMARY`

     ```
     1 mysql> EXPLAIN SELECT * FROM s1 UNION SELECT * FROM s2;
     ```

     ```
     +----+--------------+------------+------------+------+---------------+------+---------+------+------+----------+-----------------+
     | id | select_type  | table      | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra           |
     +----+--------------+------------+------------+------+---------------+------+---------+------+------+----------+-----------------+
     |  1 | PRIMARY      | s1         | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 9688 |   100.00 | NULL            |
     |  2 | UNION        | s2         | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 9954 |   100.00 | NULL            |
     | NULL | UNION RESULT | <union1,2> | NULL       | ALL  | NULL          | NULL | NULL    | NULL | NULL |     NULL | Using temporary |
     +----+--------------+------------+------------+------+---------------+------+---------+------+------+----------+-----------------+
     3 rows in set, 1 warning (0.00 sec)
     ```

     从结果中可以看到，最左边的小查询 `SELECT * FROM s1` 对应的是执行计划中的第一条记录，它的 `select_type` 值就是 `PRIMARY`。

```
mysql> EXPLAIN SELECT * FROM s1 UNION SELECT * FROM s2;
+----+--------------+------------+------------+------+---------------+------+---------+------+------+----------+-----------------+
| id | select_type  | table      | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra           |
+----+--------------+------------+------------+------+---------------+------+---------+------+------+----------+-----------------+
|  1 | PRIMARY      | s1         | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 9895 |   100.00 | NULL            |
|  2 | UNION        | s2         | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 9895 |   100.00 | NULL            |
| NULL | UNION RESULT | <union1,2> | NULL       | ALL  | NULL          | NULL | NULL    | NULL | NULL |     NULL | Using temporary |
+----+--------------+------------+------------+------+---------------+------+---------+------+------+----------+-----------------+
3 rows in set, 1 warning (0.01 sec)
```

```
mysql> EXPLAIN SELECT * FROM s1 UNION ALL SELECT * FROM s2;
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------+
|  1 | PRIMARY     | s1    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 9895 |   100.00 | NULL  |
|  2 | UNION       | s2    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 9895 |   100.00 | NULL  |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------+
2 rows in set, 1 warning (0.00 sec) 因为不需要排序，索引不会产生一个临时表
```

3. 不会被优化成多表连接的子查询

如果包含子查询的查询语句不能够转为对应的多表连接的形式（`semi-join` 的形式）(也就是不会被优化器进行自动的优化)，并且该子查询是不相关的子查询,该子查询的第一个 `SELECT` 关键字代表的那个查询的 `select_type` 就是 `SUBQUERY`

该子查询的第一个 `SELECT` 关键字代表的那个查询的 `select_type` 就是 `SUBQUERY`。也就是外层查询是 `Primary`，内层查询是 `SUBQUERY`

```
#如果包含子查询的查询语句不能够转为对应的`semi-join`的形式，并且该子查询是相关子查询，
#则该子查询的第一个`SELECT`关键字代表的那个查询的`select_type`就是`DEPENDENT SUBQUERY`
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 IN (SELECT key1 FROM s2) OR key3 = 'a';
```

```
+----+-------------+-------+------------+-------+---------------+----------+---------+------+------+----------+-------------+
| id | select_type | table | partitions | type  | possible_keys | key      | key_len | ref  | rows | filtered | Extra       |
+----+-------------+-------+------------+-------+---------------+----------+---------+------+------+----------+-------------+
|  1 | PRIMARY     | s1    | NULL       | ALL   | idx_key3      | NULL     | NULL    | NULL | 9688 |   100.00 | Using where |
|  2 | SUBQUERY    | s2    | NULL       | index | idx_key1      | idx_key1 | 303     | NULL | 9954 |   100.00 | Using index |
+----+-------------+-------+------------+-------+---------------+----------+---------+------+------+----------+-------------+
2 rows in set, 1 warning (0.00 sec)
```

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 IN (SELECT key1 FROM s2) OR key3 = 'a';
+----+-------------+-------+------------+-------+---------------+----------+---------+------+------+----------+-------------+
| id | select_type | table | partitions | type  | possible_keys | key      | key_len | ref  | rows | filtered | Extra       |
+----+-------------+-------+------------+-------+---------------+----------+---------+------+------+----------+-------------+
|  1 | PRIMARY     | s1    | NULL       | ALL   | idx_key3      | NULL     | NULL    | NULL | 9895 |   100.00 | Using where |
|  2 | SUBQUERY    | s2    | NULL       | index | idx_key1      | idx_key1 | 303     | NULL | 9895 |   100.00 | Using index |
+----+-------------+-------+------------+-------+---------------+----------+---------+------+------+----------+-------------+
2 rows in set, 1 warning (0.00 sec)
```

如果子查询不能被转换为多表连接的形式，并且该子查询是相关子查询。

比如下面的查询在内部子查询使用了外部的表。则该子查询的第一个`SELECT`关键字代表的那个查询的`select_type`就是`DEPENDENT SUBQUERY`。 外层查询是`Primary`，内层查询是`DEPENDENT SUBQUERY`

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 IN (SELECT key1 FROM s2 WHERE s1.key2 = s2.key2) OR key3 = 'a';
 #注意的是，select_type为`DEPENDENT SUBQUERY`的查询可能会被执行多次。
```

```
+----+--------------------+-------+------------+------+-----------------+----------+---------+-------------------+------+----------+-------------+
| id | select_type        | table | partitions | type | possible_keys   | key      | key_len | ref               | rows | filtered | Extra       |
+----+--------------------+-------+------------+------+-----------------+----------+---------+-------------------+------+----------+-------------+
|  1 | PRIMARY            | s1    | NULL       | ALL  | idx_key3        | NULL     | NULL    | NULL              | 9688 |   100.00 | Using where |
|  2 | DEPENDENT SUBQUERY | s2    | NULL       | ref  | idx_key2,idx_key1 | idx_key2 | 5       | xiaohaizi.s1.key2 |    1 |    10.00 | Using where |
+----+--------------------+-------+------------+------+-----------------+----------+---------+-------------------+------+----------+-------------+
2 rows in set, 2 warnings (0.00 sec)
```

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 IN (SELECT key1 FROM s2 WHERE s1.key2 = s2.key2) OR key3 = 'a';
+----+--------------------+-------+------------+--------+-----------------+----------+---------+-------------------+------+----------+-------------+
| id | select_type        | table | partitions | type   | possible_keys   | key      | key_len | ref               | rows | filtered | Extra       |
+----+--------------------+-------+------------+--------+-----------------+----------+---------+-------------------+------+----------+-------------+
|  1 | PRIMARY            | s1    | NULL       | ALL    | idx_key3        | NULL     | NULL    | NULL              | 9895 |   100.00 | Using where |
|  2 | DEPENDENT SUBQUERY | s2    | NULL       | eq_ref | idx_key2,idx_key1 | idx_key2 | 5       | atguigudb1.s1.key2 |    1 |    10.00 | Using where |
+----+--------------------+-------+------------+--------+-----------------+----------+---------+-------------------+------+----------+-------------+
2 rows in set, 2 warnings (0.01 sec)
```

需要注意的是`DEPENDENT SUBQUERY`的查询语句可能会被执行多次，因为内层查询依赖于外层的查询，因此可能会是外层传一个值，内层就执行一次的模式。

> 子查询需要执行多次，即采用循环的方式，先从外部查询开始，每次都传入子查询进行查询，然后再将结果反馈给外部，这种嵌套的执行方式就称为相关子查询。
>
> 子查询从数据表中查询了数据结果，如果这个数据结果只执行一次，然后这个数据结果作为主查询的条 件进行执行，那么这样的子查询叫做不相关子查询。

4. **包含`UNION`或者`UNION ALL`的子查询**

   对于包含`UNION`或者`UNION ALL`的大查询来说，它是由几个小查询组成的，其中除了最左边的那个小查询以外，其余的小查询的`select_type`值就是`UNION`，可以对比上一个例子的效果。

   在包含`Union`或者`Union All`的子查询 sql 中，如果各个小查询都依赖于外查询，那么除了最左边的小查询外，各个小查询的类型都是`DEPENDENT UNION`

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 IN (SELECT key1 FROM s2 WHERE key1 = 'a' UNION SELECT key1 FROM s1 WHERE key1 = 'b');
```

```
+----+--------------------+------------+------------+-------+---------------+---------+---------+-------+------+----------+-------------------------+
| id | select_type        | table      | partitions | type  | possible_keys | key     | key_len | ref   | rows | filtered | Extra                   |
+----+--------------------+------------+------------+-------+---------------+---------+---------+-------+------+----------+-------------------------+
|  1 | PRIMARY            | s1         | NULL       | ALL   | NULL          | NULL    | NULL    | NULL  | 9688 |   100.00 | Using where             |
|  2 | DEPENDENT SUBQUERY | s2         | NULL       | ref   | idx_key1      | idx_key1| 303     | const |   12 |   100.00 | Using where; Using index |
|  3 | DEPENDENT UNION    | s1         | NULL       | ref   | idx_key1      | idx_key1| 303     | const |    8 |   100.00 | Using where; Using index |
| NULL | UNION RESULT     | <union2,3> | NULL       | ALL   | NULL          | NULL    | NULL    | NULL  | NULL |     NULL | Using temporary         |
+----+--------------------+------------+------------+-------+---------------+---------+---------+-------+------+----------+-------------------------+
4 rows in set, 1 warning (0.03 sec)
```

MySQL选择使用临时表来完成`UNION`查询的去重工作，针对该临时表的查询的`se1ect_type`就是`UNION RESULT`，例子上边有

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 IN (SELECT key1 FROM s2 WHERE key1 = 'a' UNION SELECT key1 FROM s1 WHERE key1 = 'b');
+----+--------------------+------------+------------+-------+---------------+---------+---------+-------+------+----------+-------------------------+
| id | select_type        | table      | partitions | type  | possible_keys | key     | key_len | ref   | rows | filtered | Extra                   |
+----+--------------------+------------+------------+-------+---------------+---------+---------+-------+------+----------+-------------------------+
|  1 | PRIMARY            | s1         | NULL       | ALL   | NULL          | NULL    | NULL    | NULL  | 9895 |   100.00 | Using where             |
|  2 | DEPENDENT SUBQUERY | s2         | NULL       | ref   | idx_key1      | idx_key1| 303     | const |    1 |   100.00 | Using where; Using index |
|  3 | DEPENDENT UNION    | s1         | NULL       | ref   | idx_key1      | idx_key1| 303     | const |    1 |   100.00 | Using where; Using index |
| NULL | UNION RESULT     | <union2,3> | NULL       | ALL   | NULL          | NULL    | NULL    | NULL  | NULL |     NULL | Using temporary         |
+----+--------------------+------------+------------+-------+---------------+---------+---------+-------+------+----------+-------------------------+
4 rows in set, 1 warning (0.01 sec)
```

外查询是`Primary`，最左边的子查询是`DEPENDENT SUBQUERY`，后面的子查询是`DEPENDENT UNION`，临时去重表的类型是`Union Result`。这里大家可能要困惑，第一个子查询中也没有看到依赖 s1 啊。这其实也是优化器会在执行时进行优化，将`IN`改成`Exist`，并且把外部的表移到内部去。这里我们了解就行，以后会有文章给大家介绍优化器的。

5. **关于派生表的子查询**

   对于包含`派生表`的查询，该派生表对应的子查询的`select_type`就是`DERIVED`

```
mysql> EXPLAIN SELECT * FROM (SELECT key1, count(*) as c FROM s1 GROUP BY key1) AS derived_s1 where c > 1;
```

```
+----+-------------+------------+------------+-------+---------------+---------+---------+------+------+----------+-------------+
| id | select_type | table      | partitions | type  | possible_keys | key     | key_len | ref  | rows | filtered | Extra       |
+----+-------------+------------+------------+-------+---------------+---------+---------+------+------+----------+-------------+
|  1 | PRIMARY     | <derived2> | NULL       | ALL   | NULL          | NULL    | NULL    | NULL | 9688 |    33.33 | Using where |
|  2 | DERIVED     | s1         | NULL       | index | idx_key1      | idx_key1| 303     | NULL | 9688 |   100.00 | Using index |
+----+-------------+------------+------------+-------+---------------+---------+---------+------+------+----------+-------------+
2 rows in set, 1 warning (0.00 sec)
```

```
mysql> EXPLAIN SELECT * FROM (SELECT key1, count(*) as c FROM s1 GROUP BY key1) AS derived_s1 where c > 1;
+----+-------------+------------+------------+-------+---------------+---------+---------+------+------+----------+-------------+
| id | select_type | table      | partitions | type  | possible_keys | key     | key_len | ref  | rows | filtered | Extra       |
+----+-------------+------------+------------+-------+---------------+---------+---------+------+------+----------+-------------+
|  1 | PRIMARY     | <derived2> | NULL       | ALL   | NULL          | NULL    | NULL    | NULL | 9895 |   100.00 | NULL        |
|  2 | DERIVED     | s1         | NULL       | index | idx_key1      | idx_key1| 303     | NULL | 9895 |   100.00 | Using index |
+----+-------------+------------+------------+-------+---------------+---------+---------+------+------+----------+-------------+
2 rows in set, 1 warning (0.01 sec)
由 id = 2 这条查询 所派生出来的表
```

6. 子查询的物化后与外层连接查询

当优化器在执行子查询时选择把子查询优化成为一张物化表，与外层查询进行连接查询时。

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 IN (SELECT key1 FROM s2);
```

```
+----+-------------+-------------+------------+--------+---------------+-----------+---------+-------------------+------+----------+-------------+
| id | select_type | table       | partitions | type   | possible_keys | key       | key_len | ref               | rows | filtered | Extra       |
+----+-------------+-------------+------------+--------+---------------+-----------+---------+-------------------+------+----------+-------------+
|  1 | SIMPLE      | s1          | NULL       | ALL    | idx_key1      | NULL      | NULL    | NULL              | 9688 |   100.00 | Using where |
|  1 | SIMPLE      | <subquery2> | NULL       | eq_ref | <auto_key>    | <auto_key>| 303     | xiaohaizi.s1.key1 |    1 |   100.00 | NULL        |
|  2 | MATERIALIZED| s2          | NULL       | index  | idx_key1      | idx_key1  | 303     | NULL              | 9954 |   100.00 | Using index |
+----+-------------+-------------+------------+--------+---------------+-----------+---------+-------------------+------+----------+-------------+
3 rows in set, 1 warning (0.01 sec)
```

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 IN (SELECT key1 FROM s2);
+----+-------------+-------------+------------+--------+---------------+-----------------+---------+-------------------+------+----------+-------------+
| id | select_type | table       | partitions | type   | possible_keys | key             | key_len | ref               | rows | filtered | Extra       |
+----+-------------+-------------+------------+--------+---------------+-----------------+---------+-------------------+------+----------+-------------+
|  1 | SIMPLE      | s1          | NULL       | ALL    | idx_key1      | NULL            | NULL    | NULL              | 9895 |   100.00 | Using where |
|  1 | SIMPLE      | <subquery2> | NULL       | eq_ref | <auto_key>    | <auto_key>      | 303     | atguigudb1.s1.key1|    1 |   100.00 | NULL        |
|  2 | MATERIALIZED| s2          | NULL       | index  | idx_key1      | idx_key1        | 303     | NULL              | 9895 |   100.00 | Using index |
+----+-------------+-------------+------------+--------+---------------+-----------------+---------+-------------------+------+----------+-------------+
3 rows in set, 1 warning (0.01 sec)
将id为2的查询结果物化，形成一张物化表subquery2，然后再与外层的表作连接查询
```

从下往上看，子查询的查询类型是`MATERIALIZED`；物化过程是基于 id 为 2 的查询结果表进行的，其 table 是`subquery 2`，查询类型是`SIMPLE`，而外层也相当于是与固定的直接值进行查询，其类型也是`SIMPLE`

上面的介绍都是一些基本的情况，还没有真正的介绍与索引相关的情况哦。觉得是不是晕晕的了，我们用一个表格进行下总结吧

6.4.4 partitions (可略)
代表分区表中的命中情况，非分区表，该项为 NULL。一般情况下我们的查询语句的执行计划的 partitions 列的值都是NULL

官方文档：https://dev.mysql.com/doc/refman/8.0/en/alter-table-partition-operations.html

如果想详细了解，可以如下方式测试。创建分区表：

```
-- 创建分区表，
-- 按照id分区，id<100 p0分区，其他p1分区
CREATE TABLE user_partitions (id INT auto_increment,
    NAME VARCHAR(12),PRIMARY KEY(id))
    PARTITION BY RANGE(id)(
    PARTITION p0 VALUES less than(100),
    PARTITION p1 VALUES less than MAXVALUE
 );
```

查询 id 大于200（200>100，p1分区）的记录，查看执行计划，partitions 是 p1，符合我们的分区规则

```
mysql> DESC SELECT * FROM user_partitions WHERE id>200;
+----+-------------+----------------+------------+------+---------------+---------+---------+------+------+----------+-------------+
| id | select_type | table          | partitions | type | possible_keys | key     | key_len | ref  | rows | filtered | Extra       |
+----+-------------+----------------+------------+------+---------------+---------+---------+------+------+----------+-------------+
|  1 | SIMPLE      | user_partitions | p1         | range | PRIMARY       | PRIMARY | 4       | NULL |    1 |   100.00 | Using where |
+----+-------------+----------------+------------+------+---------------+---------+---------+------+------+----------+-------------+
1 row in set, 1 warning (0.01 sec)
```

- `UNCACHEABLE SUBQUERY`
  不常用，就不多说了。
- `UNCACHEABLE UNION`
  不常用，就不多说了。

## 4. partitions (可略)

- 代表分区表中的命中情况，非分区表，该项为`NULL`。一般情况下我们的查询语句的执行计划的`partitions`列的值都是`NULL`。
- https://dev.mysql.com/doc/refman/5.7/en/alter-table-partition-operations.html
- 如果想详细了解，可以如下方式测试。创建分区表：

```
-- 创建分区表，
-- 按照id分区，id<100 p0分区，其他p1分区
CREATE TABLE user_partitions (id INT auto_increment,
    NAME VARCHAR(12),PRIMARY KEY(id))
    PARTITION BY RANGE(id)(
        PARTITION p0 VALUES less than(100),
        PARTITION p1 VALUES less than MAXVALUE
);
```

```
mysql> create table user_partitions (id int auto_increment, name varchar(12),primary key(id))
    -> partition by range(id)(partition p0 values less than(100),
    -> partition p1 values less than maxvalue);
Query OK, 0 rows affected (0.11 sec)
```

```
DESC SELECT * FROM user_partitions WHERE id>200;
```

查询id大于200（200>100，p1分区）的记录，查看执行计划，partitions是p1，符合我们的分区规则

```
mysql> desc select * from user_partitions where id>200;
+----+-------------+-----------------+------------+-------+---------------+---------+---------+------+------+----------+-------------+
| id | select_type | table           | partitions | type  | possible_keys | key     | key_len | ref  | rows | filtered | Extra       |
+----+-------------+-----------------+------------+-------+---------------+---------+---------+------+------+----------+-------------+
|  1 | SIMPLE      | user_partitions | p1         | range | PRIMARY       | PRIMARY | 4       | NULL |    1 |   100.00 | Using where |
+----+-------------+-----------------+------------+-------+---------------+---------+---------+------+------+----------+-------------+
1 row in set, 1 warning (0.01 sec)
```

## 5. type ☆

执行计划的一条记录就代表着MySQL对某个表的`执行查询时的访问方法`，又称"访问类型”，其中的`type`列就表明了这个访问方法是啥，是较为重要的一个指标。比如，看到`type`列的值是`ref`，表明`MySQL即将`使用`ref`访问方法来执行对s1表的查询。

完整的访问方法如下：`system`，`const`，`eq_ref`，`ref`，`fulltext`，`ref_or_null`，`index_merge`，`unique_subquery`，`index_subquery`，`range`，`index`，`ALL`。

我们详细解释一下：

- `system`
  当表中只有一条记录，并且该表中存储引擎统计数据是精确的，比如 MYISAM，Memory，那么其访问方法就是`System`。这种方式几乎是性能最高的，当然我们几乎用不上。

  ```
  mysql> CREATE TABLE t(i int) Engine=MyISAM;
  Query OK, 0 rows affected (0.05 sec)
  mysql> INSERT INTO t VALUES(1);
  Query OK, 1 row affected (0.01 sec)
  ```

  然后我们看一下查询这个表的执行计划：

  ```
  mysql> EXPLAIN SELECT * FROM t;
  ```

  ```
  +----+-------------+-------+------------+--------+---------------+------+---------+------+------+----------+-------+
  | id | select_type | table | partitions | type   | possible_keys | key  | key_len | ref  | rows | filtered | Extra |
  +----+-------------+-------+------------+--------+---------------+------+---------+------+------+----------+-------+
  |  1 | SIMPLE      | t     | NULL       | system | NULL          | NULL | NULL    | NULL |    1 |   100.00 | NULL  |
  +----+-------------+-------+------------+--------+---------------+------+---------+------+------+----------+-------+
  1 row in set, 1 warning (0.00 sec)
  ```

  但凡我们再插入一条数据，其访问方式就变成了性能最差的全表扫描`ALL`。

  ```
  mysql> INSERT INTO t VALUES(2);
  Query OK, 1 row affected (0.01 sec)

  mysql> EXPLAIN SELECT * FROM t;
  +----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------+
  | id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra |
  +----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------+
  |  1 | SIMPLE      | t     | NULL       | ALL  | NULL          | NULL | NULL    | NULL |    2 |   100.00 | NULL  |
  +----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------+
  1 row in set, 1 warning (0.00 sec)
  ```

  如果存储引擎是InnoDB，即使只有一条数据，其访问方式也是ALL，这是因为 InnnoDB 访问数据不是精确的

  ```
  mysql> CREATE TABLE tt(i INT) ENGINE=INNODB;
  Query OK, 0 rows affected (0.03 sec)

  mysql> INSERT INTO tt VALUES(1);
  Query OK, 1 row affected (0.00 sec)

  mysql> EXPLAIN SELECT * FROM tt;
  +----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------+
  | id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra |
  +----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------+
  |  1 | SIMPLE      | tt    | NULL       | ALL  | NULL          | NULL | NULL    | NULL |    1 |   100.00 | NULL  |
  +----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------+
  1 row in set, 1 warning (0.00 sec)
  ```

- `const`

当我们根据主键或者唯一的二级索引，与常数进行等值匹配时，对单表的访问方法就是`const`。这个访问方式的效率低于`system`，但也是很高效的。

比如对主键与常数匹配，进行等值查询

```
mysql> EXPLAIN SELECT * FROM s1 WHERE id = 10005;
```

```
+----+-------------+-------+------------+-------+---------------+---------+---------+-------+------+----------+-------+
| id | select_type | table | partitions | type  | possible_keys | key     | key_len | ref   | rows | filtered | Extra |
+----+-------------+-------+------------+-------+---------------+---------+---------+-------+------+----------+-------+
|  1 | SIMPLE      | s1    | NULL       | const | PRIMARY       | PRIMARY | 4       | const |    1 |   100.00 | NULL  |
+----+-------------+-------+------------+-------+---------------+---------+---------+-------+------+----------+-------+
1 row in set, 1 warning (0.01 sec)
```

比如对Unique标识的唯一二级索引key2与常数匹配，进行等值查询。

```
EXPLAIN SELECT * FROM s1 WHERE key2 = 10066;
```

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key2 = 10066;
+----+-------------+-------+------------+-------+---------------+----------+---------+-------+------+----------+-------+
| id | select_type | table | partitions | type  | possible_keys | key      | key_len | ref   | rows | filtered | Extra |
+----+-------------+-------+------------+-------+---------------+----------+---------+-------+------+----------+-------+
|  1 | SIMPLE      | s1    | NULL       | const | idx_key2      | idx_key2 | 5       | const |    1 |   100.00 | NULL  |
+----+-------------+-------+------------+-------+---------------+----------+---------+-------+------+----------+-------+
1 row in set, 1 warning (0.01 sec)
```

当我们把where后面的关键字改成key3 普通索引时，由于key3的字段值是可重复的。type类型就变成了 All，全表查询

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key3 = 10066;
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra       |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------------+
|  1 | SIMPLE      | s1    | NULL       | ALL  | idx_key3      | NULL | NULL    | NULL | 9895 |    10.00 | Using where |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------------+
1 row in set, 3 warnings (0.01 sec)
```

注意：此时可能有细心的小伙伴可能注意到我们的key3不是也有索引`idx_key3`，性能再查也不至于全表查询吧~ 但是忘记了一个细节，key3本身是varchar类型的，`key3 = 10066`会进行一个隐式类型转换，从而会导致 索引失效，我们也可以看到`key`列对应的为空。

当修改成key3 = 10066后，结果如下图：

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key3 = '10066';
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------+
| id | select_type | table | partitions | type | possible_keys | key      | key_len | ref   | rows | filtered | Extra |
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------+
|  1 | SIMPLE      | s1    | NULL       | ref  | idx_key3      | idx_key3 | 303     | const |    1 |   100.00 | NULL  |
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------+
1 row in set, 1 warning (0.00 sec)
```

- `eq_ref`

在进行**连接查询**时，如果**被驱动表**是通过主键或者唯一二级索引等值匹配的方式进行查询的（如果该主键或者唯一耳机索引是联合索引的话，所有的索引列都必须进行等值比较），那么被驱动表的访问方式是`eq_ref`。这也是一种性能很不错的方式。

```
EXPLAIN SELECT * FROM s1 INNER JOIN s2 ON s1.id = s2.id;
```

上面连接查询语句，对于驱动表来说，就是对s1全表进行扫描，找到符合条件的数据，因此其`type`是`All`,对被驱动表来说，相于直接访问驱动表查询到的数据进行等值查询，因此其访问方式是`eq_ref`

```
mysql> EXPLAIN SELECT * FROM s1 INNER JOIN s2 ON s1.id = s2.id;
+----+-------------+-------+------------+--------+---------------+---------+---------+-----------------+------+----------+-------+
| id | select_type | table | partitions | type   | possible_keys | key     | key_len | ref             | rows | filtered | Extra |
+----+-------------+-------+------------+--------+---------------+---------+---------+-----------------+------+----------+-------+
|  1 | SIMPLE      | s1    | NULL       | ALL    | PRIMARY       | NULL    | NULL    | NULL            | 9895 |   100.00 | NULL  |
|  1 | SIMPLE      | s2    | NULL       | eq_ref | PRIMARY       | PRIMARY | 4       | atguigudb1.s1.id |    1 |   100.00 | NULL  |
+----+-------------+-------+------------+--------+---------------+---------+---------+-----------------+------+----------+-------+
2 rows in set, 1 warning (0.01 sec)
```

```
mysql> EXPLAIN SELECT * FROM s1 INNER JOIN s2 ON s1.id = s2.id;
```

```
+----+-------------+-------+------------+--------+---------------+---------+---------+------------+------+----------+-------+
| id | select_type | table | partitions | type   | possible_keys | key     | key_len | ref        | rows | filtered | Extra |
+----+-------------+-------+------------+--------+---------------+---------+---------+------------+------+----------+-------+
|  1 | SIMPLE      | s2    | NULL       | ALL    | PRIMARY       | NULL    | NULL    | NULL       | 9895 |   100.00 | NULL  |
|  1 | SIMPLE      | s1    | NULL       | eq_ref | PRIMARY       | PRIMARY | 4       | temp.s2.id |    1 |   100.00 | NULL  |
+----+-------------+-------+------------+--------+---------------+---------+---------+------------+------+----------+-------+
2 rows in set, 1 warning (0.01 sec)
```

从执行计划的结果中可以看出，MySQL打算将s2作为驱动表，s1作为被驱动表，重点关注s1的访问方法是`eq_ref`，表明在访问s1表的时候可以`通过主键的等值匹配`来进行访问。

- `ref`

  当使用普通的二级索引与常量进行等值匹配时来查询某个表，那么对该表的访问方法就可能是`ref`

```
EXPLAIN SELECT * FROM s1 WHERE key1 = 'a';
```

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 = 'a';
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------+
| id | select_type | table | partitions | type | possible_keys | key      | key_len | ref   | rows | filtered | Extra |
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------+
|  1 | SIMPLE      | s1    | NULL       | ref  | idx_key1      | idx_key1 | 303     | const |    1 |   100.00 | NULL  |
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------+
1 row in set, 1 warning (0.00 sec)
```

以下 sql 的引用类型是什么呢?

```
EXPLAIN SELECT * FROM s1 WHERE key3 = 10066;
```

看看答案。你是不是猜错了。是`All`。这是因为 key3 的字段 varchar 类型，但是我们这里常量值是整形，因此需要使用函数进行隐式的类型转换，一旦使用函数，索引就失效了，因此访问类型变成了全表扫描`All`

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key3 = 10066;
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra       |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------------+
|  1 | SIMPLE      | s1    | NULL       | ALL  | idx_key3      | NULL | NULL    | NULL | 9895 |    10.00 | Using where |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------------+
1 row in set, 3 warnings (0.00 sec)
```

当我们常量使用对应的类型，就是期望的`ref`访问类型了

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key3 = '10066';
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------+
| id | select_type | table | partitions | type | possible_keys | key      | key_len | ref   | rows | filtered | Extra |
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------+
|  1 | SIMPLE      | s1    | NULL       | ref  | idx_key3      | idx_key3 | 303     | const |    1 |   100.00 | NULL  |
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------+
1 row in set, 1 warning (0.00 sec)
```

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 = 'a';
```

```
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------+
| id | select_type | table | partitions | type | possible_keys | key      | key_len | ref   | rows | filtered | Extra |
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------+
|  1 | SIMPLE      | s1    | NULL       | ref  | idx_key1      | idx_key1 | 303     | const |    8 |   100.00 | NULL  |
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------+
1 row in set, 1 warning (0.04 sec)
```

- `fulltext`

  全文索引

- `ref_or_null`

  当使用普通的二级索引进行等值匹配时，该索引列的值也可以是`NULL`值时，那么对该表的访问方法就可能是`ref_or_null`

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 = 'a' OR key1 IS NULL;
```

```
+----+-------------+-------+------------+-------------+---------------+----------+---------+-------+------+----------+-----------------------+
| id | select_type | table | partitions | type        | possible_keys | key      | key_len | ref   | rows | filtered | Extra                 |
+----+-------------+-------+------------+-------------+---------------+----------+---------+-------+------+----------+-----------------------+
|  1 | SIMPLE      | s1    | NULL       | ref_or_null | idx_key1      | idx_key1 | 303     | const |    9 |   100.00 | Using index condition |
+----+-------------+-------+------------+-------------+---------------+----------+---------+-------+------+----------+-----------------------+
1 row in set, 1 warning (0.01 sec)
```

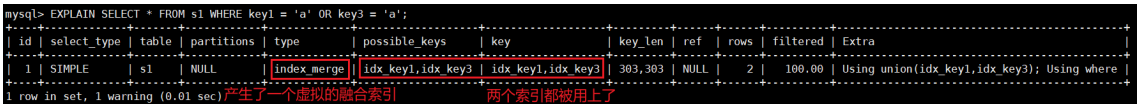
- `index_merge`

当进行单表访问时，如果多个查询字段分别建立了单列索引，使用 OR 连接，其访问类型是 `index_merge`。同时还可以看到 `key` 这一字段，是使用了多个索引

一般情况下对于某个表的查询只能使用到一个索引，但单表访问方法时在某些场景下可以使用 `Intersection`、`Union`、`Sort-Union` 这三种索引合并的方式来执行查询。我们看一下执行计划中是怎么体现MySQL使用索引合并的方式来对某个表执行查询的：

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 = 'a' OR key3 = 'a';#必须是or
```

```
+----+-------------+-------+------------+-------------+-----------------+-----------------+---------+------+------+----------+-----------------------------------------------------+
| id | select_type | table | partitions | type        | possible_keys   | key             | key_len | ref  | rows | filtered | Extra                                               |
+----+-------------+-------+------------+-------------+-----------------+-----------------+---------+------+------+----------+-----------------------------------------------------+
|  1 | SIMPLE      | s1    | NULL       | index_merge | idx_key1,idx_key3 | idx_key1,idx_key3 | 303,303 | NULL |   14 |   100.00 | Using union(idx_key1,idx_key3); Using where |
+----+-------------+-------+------------+-------------+-----------------+-----------------+---------+------+------+----------+-----------------------------------------------------+
1 row in set, 1 warning (0.01 sec)
```

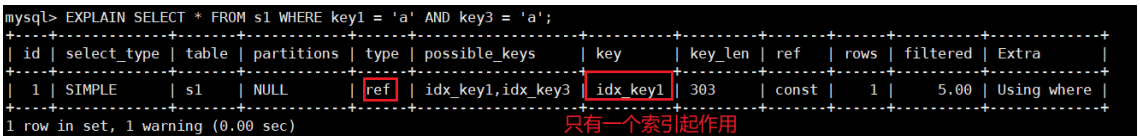



从执行计划的 `type` 列的值是 `index_merge` 就可以看出，MySQL 打算使用索引合并的方式来执行对 `s1` 表的查询。

猜猜下面 sql 的引用类型

```
EXPLAIN SELECT * FROM s1 WHERE key1 = 'a' AND key3 = 'a';
```

猜对了吗？答案是 `ref`，这是因为用 AND 连接两个查询时，实际上只使用了 key1 的索引




- `unique_subquery`

类似于两表连接中被驱动表的 `eq_ref` 访问方法，`unique_subquery` 是针对在一些包含 `IN` 子查询的查询语句中，如果查询优化器决定将 `IN` 子查询转换为 `EXISTS` 子查询，而且子查询可以使用到主键进行等值匹配的话，那么该子查询执行计划的 `type` 列的值就是 `unique_subquery`，比如下边的这个查询语句：

针对一些包含 `IN` 的 子查询的查询语句中，如果优化器决定将 IN 子查询优化为 EXIST 子查询，而且子查询可以使用主键进行等值匹配的话，那么该子查询执行计划的 `type` 就是 `unique_subquery`

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key2 IN (SELECT id FROM s2 where
s1.key1 = s2.key1) OR key3 = 'a';
```

```
+----+--------------------+-------+------------+-----------------+-----------------+---------+---------+------+------+----------+-----------------------+
| id | select_type        | table | partitions | type            | possible_keys   | key     | key_len | ref  | rows | filtered | Extra                 |
+----+--------------------+-------+------------+-----------------+-----------------+---------+---------+------+------+----------+-----------------------+
|  1 | PRIMARY            | s1    | NULL       | ALL             | idx_key3        | NULL    | NULL    | NULL | 9688 |   100.00 | Using where           |
|  2 | DEPENDENT SUBQUERY | s2    | NULL       | unique_subquery | PRIMARY,idx_key1 | PRIMARY | 4       | func |    1 |    10.00 | Using where           |
+----+--------------------+-------+------------+-----------------+-----------------+---------+---------+------+------+----------+-----------------------+
2 rows in set, 2 warnings (0.00 sec)
```

可以看到执行计划的第二条记录的 type 值就是 unique_subquery，说明在执行子查询时会使用到id列的索引。

- index_subquery

  index_subquery与unique_subquery类似，只不过访问子查询中的表时使用的是普通的索引，比如这样：

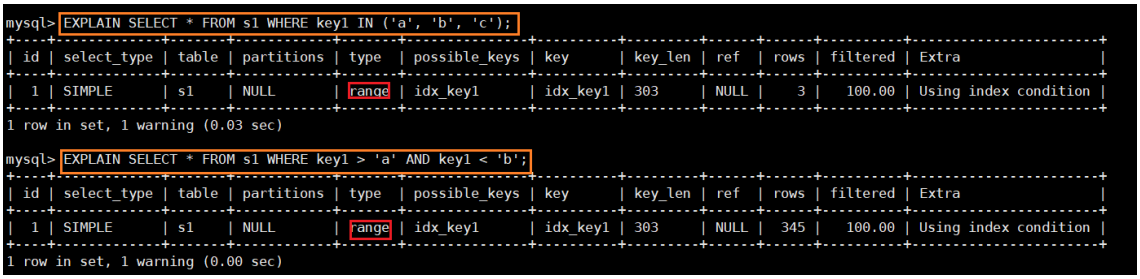
```
mysql> EXPLAIN SELECT * FROM s1 WHERE common_field IN (SELECT key3 FROM s2 where s1.key1 = s2.key1) OR key3 = 'a';
```

```
+----+--------------------+-------+------------+----------------+------------------+---------+---------+------+------+----------+-------------+
| id | select_type        | table | partitions | type           | possible_keys    | key     | key_len | ref  | rows | filtered | Extra       |
+----+--------------------+-------+------------+----------------+------------------+---------+---------+------+------+----------+-------------+
|  1 | PRIMARY            | s1    | NULL       | ALL            | idx_key3         | NULL    | NULL    | NULL | 9688 |   100.00 | Using where |
|  2 | DEPENDENT SUBQUERY | s2    | NULL       | index_subquery | idx_key1,idx_key3 | idx_key3 | 303     | func |    1 |    10.00 | Using where |
+----+--------------------+-------+------------+----------------+------------------+---------+---------+------+------+----------+-------------+
2 rows in set, 2 warnings (0.01 sec)
```

- range

  如果使用索引获取某些范围区间的记录，那么就可能使用到range访问方法

```
EXPLAIN SELECT * FROM s1 WHERE key1 IN ('a', 'b', 'c');
EXPLAIN SELECT * FROM s1 WHERE key1 > 'a' AND key1 < 'b';
```




```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 IN ('a', 'b', 'c');
```

```
+----+-------------+-------+------------+-------+---------------+----------+---------+------+------+----------+-----------------------+
| id | select_type | table | partitions | type  | possible_keys | key      | key_len | ref  | rows | filtered | Extra                 |
+----+-------------+-------+------------+-------+---------------+----------+---------+------+------+----------+-----------------------+
|  1 | SIMPLE      | s1    | NULL       | range | idx_key1      | idx_key1 | 303     | NULL |   27 |   100.00 | Using index condition |
+----+-------------+-------+------------+-------+---------------+----------+---------+------+------+----------+-----------------------+
1 row in set, 1 warning (0.01 sec)
```

或者：

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 > 'a' AND key1 < 'b';
```

```
+----+-------------+-------+------------+-------+---------------+----------+---------+------+------+----------+-----------------------+
| id | select_type | table | partitions | type  | possible_keys | key      | key_len | ref  | rows | filtered | Extra                 |
+----+-------------+-------+------------+-------+---------------+----------+---------+------+------+----------+-----------------------+
|  1 | SIMPLE      | s1    | NULL       | range | idx_key1      | idx_key1 | 303     | NULL |  294 |   100.00 | Using index condition |
+----+-------------+-------+------------+-------+---------------+----------+---------+------+------+----------+-----------------------+
1 row in set, 1 warning (0.00 sec)
```

- index

  当我们可以使用索引覆盖，但是需要扫描的全部的索引记录时，该表的访问方式就是 index。

  索引覆盖后面文章介绍优化器时会详细介绍，为了便于大家理解，先简单介绍如下。比如下面 sql 语句中，key_part2 ， key_part2 都属于联合索引 idx_key_part(key_part1, key_part2, key_part3) 的一部分，在查找数据时可以用上这个联合索引，而不用进行回表操作，这种情况即索引覆盖

```
mysql> EXPLAIN SELECT key_part2 FROM s1 WHERE key_part3 = 'a';
```

```
+----+-------------+-------+------------+-------+---------------+-------------+---------+------+------+----------+--------------------------+
| id | select_type | table | partitions | type  | possible_keys | key         | key_len | ref  | rows | filtered | Extra                    |
+----+-------------+-------+------------+-------+---------------+-------------+---------+------+------+----------+--------------------------+
|  1 | SIMPLE      | s1    | NULL       | index | NULL          | idx_key_part | 909     | NULL | 9688 |    10.00 | Using where; Using index |
+----+-------------+-------+------------+-------+---------------+-------------+---------+------+------+----------+--------------------------+
1 row in set, 1 warning (0.00 sec)
```

上述查询中的搜索列表中只有`key_part2`一个列，而且搜索条件中也只有`key_part3`一个列，这两个列又恰好包含在`idx_key_part`这个索引中，可是搜索条件`key_part3`不能直接使用该索引进行`ref`或者`range`方式的访问，只能扫描整个`idx_key_part`索引的记录，所以查询计划的`type`列的值就是`index`。

> 再一次强调，对于使用InnoDB存储引擎的表来说，二级索引的记录只包含索引列和主键列的值，而聚簇索引中包含用户定义的全部列以及一些隐藏列，所以扫描二级索引的代价比直接全表扫描，也就是扫描聚索引的代价更低一些。

- `ALL`

  最熟悉的全表扫描，就不多说了，直接看例子：

```
mysql> EXPLAIN SELECT * FROM s1;
```

```
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------+
|  1 | SIMPLE      | s1    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 9688 |   100.00 | NULL  |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------+
1 row in set, 1 warning (0.00 sec)
```

一般来说，这些访问方法中除了`All`这个访问方法外，其余的访问方法都能用到索引，除了`index_merge`访问方法外，其余的访问方法都最多只能用到一个索引。

**小结：**

**结果值从最好到最坏依次是： system > const > eq_ref > ref > fulltext > ref_or_null > index_merge > unique_subquery > index_subquery > range > index > ALL 其中比较重要的几个提取出来（见上图中的蓝色）。**

**SQL性能优化的目标：至少要达到range级别，要求是ref级别，最好是consts级别。（阿里巴巴开发手册要求）**

## 6. possible_keys和key

在EXPLAIN语句输出的执行计划中，`possible_keys`列表示在某个查询语句中，对某个表执行`单表查询时可能用到的索引`有哪些。一般查询涉及到的字段上若存在索引，则该索引将被列出，但不一定被查询使用。`key`列表示`实际用到的索引`有哪些，如果为NULL，则没有使用索引。比方说下边这个查询：

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 > 'z' AND key3 = 'a';
```

```
+----+-------------+-------+------------+------+-------------------+----------+---------+-------+------+----------+-------------+
| id | select_type | table | partitions | type | possible_keys     | key      | key_len | ref   | rows | filtered | Extra       |
+----+-------------+-------+------------+------+-------------------+----------+---------+-------+------+----------+-------------+
|  1 | SIMPLE      | s1    | NULL       | ref  | idx_key1,idx_key3 | idx_key3 | 303     | const |    6 |     2.75 | Using where |
+----+-------------+-------+------------+------+-------------------+----------+---------+-------+------+----------+-------------+
1 row in set, 1 warning (0.01 sec)
```

上述执行计划的`possible_keys`列的值是`idx_key1,idx_key3`，表示该查询可能使用到`idx_key1,idx_key3`两个索引，然后key列的值是`idx_key3`，表示经过查询优化器计算使用不同索引的成本后，最后决定使用`idx_key3`来执行查询比较划算。

对应优化器来说，可以选择的 possible_keys 越少越好，因为选项越多，进行过滤花的时间也就对应更多。另外，优化器会对各个索引进行查询的效率进行评估，以此来选择实际使用的 key。而且由于优化器会对 sql 进行优化，完全可能会出现 possible_keys 是 null，但是 key 不为 null 的情况

## 7. key_len ☆

实际使用的索引的长度，单位是字节。可以帮助你检查是否充分利用了索引，主要针对联合索引具有一定的参考意义，对同一索引来说，key_len 值越大越好（与自己比较，后面将解释）。

```
mysql> EXPLAIN SELECT * FROM s1 WHERE id = 10005;
```

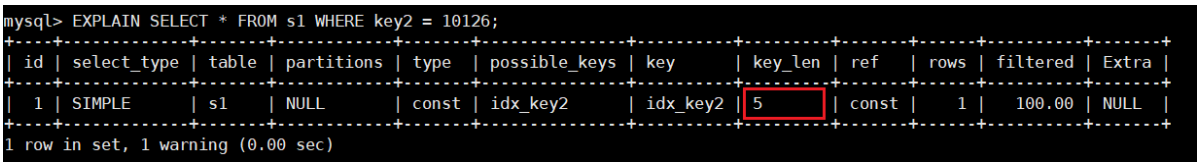
```
+----+-------------+-------+------------+-------+---------------+---------+---------+-------+------+----------+-------+
| id | select_type | table | partitions | type  | possible_keys | key     | key_len | ref   | rows | filtered | Extra |
+----+-------------+-------+------------+-------+---------------+---------+---------+-------+------+----------+-------+
|  1 | SIMPLE      | s1    | NULL       | const | PRIMARY       | PRIMARY | 4       | const |    1 |   100.00 | NULL  |
+----+-------------+-------+------------+-------+---------------+---------+---------+-------+------+----------+-------+
1 row in set, 1 warning (0.01 sec)
```

这是因为使用的是主键 id 作为索引，其类型是 int，占 4 个字节

再来猜猜下面的 key_len 是多少~

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key2 = 10126;
```

```
+----+-------------+-------+------------+-------+---------------+----------+---------+-------+------+----------+-------+
| id | select_type | table | partitions | type  | possible_keys | key      | key_len | ref   | rows | filtered | Extra |
+----+-------------+-------+------------+-------+---------------+----------+---------+-------+------+----------+-------+
|  1 | SIMPLE      | s1    | NULL       | const | idx_key2      | idx_key2 | 5       | const |    1 |   100.00 | NULL  |
+----+-------------+-------+------------+-------+---------------+----------+---------+-------+------+----------+-------+
1 row in set, 1 warning (0.00 sec)
```




这是因为虽然 key2 也是 int 类型，但是它被 unique 修饰，并没有标识非空（而主键都是非空的），因此加上空值标记，一共是5字节

字符类型的索引长度为多少呢

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 = 'a';
```

```
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------+
| id | select_type | table | partitions | type | possible_keys | key      | key_len | ref   | rows | filtered | Extra |
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------+
|  1 | SIMPLE      | s1    | NULL       | ref  | idx_key1      | idx_key1 | 303     | const |    8 |   100.00 | NULL  |
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------+
1 row in set, 1 warning (0.00 sec)
```

答案是 303，因为类型是 varchar(100)，100 个字符，utf-8 每个字符占 3 个字节，共 300 个字节，加上变长列表 2 个字节与一个空值标识占一个字节，共 303 字节。

看看联合索引的情况

- 看下面的联合索引，key_len还是303，不需要解释了吧

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key_part1 = 'a';
```

```
+----+-------------+-------+------------+------+---------------+-------------+---------+-------+------+----------+-------+
| id | select_type | table | partitions | type | possible_keys | key         | key_len | ref   | rows | filtered | Extra |
+----+-------------+-------+------------+------+---------------+-------------+---------+-------+------+----------+-------+
|  1 | SIMPLE      | s1    | NULL       | ref  | idx_key_part  | idx_key_part | 303     | const |   12 |   100.00 | NULL  |
+----+-------------+-------+------------+------+---------------+-------------+---------+-------+------+----------+-------+
1 row in set, 1 warning (0.00 sec)
```

- 再看看下面这个联合索引，其结果是 606

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key_part1 = 'a' AND key_part2 = 'b';
```

```
+----+-------------+-------+------------+------+---------------+-------------+---------+-------------+------+----------+-------+
| id | select_type | table | partitions | type | possible_keys | key         | key_len | ref         | rows | filtered | Extra |
+----+-------------+-------+------------+------+---------------+-------------+---------+-------------+------+----------+-------+
|  1 | SIMPLE      | s1    | NULL       | ref  | idx_key_part  | idx_key_part | 606     | const,const |    1 |   100.00 | NULL  |
+----+-------------+-------+------------+------+---------------+-------------+---------+-------------+------+----------+-------+
1 row in set, 1 warning (0.01 sec)
```

这个查询的 key-len 比上面的查询大，性能就比上面的好，怎么理解呢？其实只要你看过我之前介绍 B+树的文章就很容易理解了。因为在目录页我除了考虑 key_part1 ，还会考虑 key_part2，定位到的数据就更加精准，范围更小，需要加载 I/O 的数据页数量就会更少，这样是不是性能就比较好啊~

- 猜猜下面的 sql 执行后 key_len 是多少

  ```
  EXPLAIN SELECT * FROM s1 WHERE key_part3 = 'a';
  ```

  ```
  mysql> EXPLAIN SELECT * FROM s1 WHERE key_part3 = 'a';
  +----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------------+
  | id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra       |
  +----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------------+
  |  1 | SIMPLE      | s1    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 9895 |    10.00 | Using where |
  +----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------------+
  1 row in set, 1 warning (0.01 sec)
  ```

  是空哦，因为都不会使用到索引，这就是我们一直在提的**最左前缀原则**，后面会详细介绍的

**练习：**

**key_len的长度计算公式：**

```
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)

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)

char(10)固定字段且允许NULL = 10 * ( character set：utf8=3,gbk=2,latin1=1)+1(NULL)

char(10)固定字段且不允许NULL = 10 * ( character set： utf8=3,gbk=2,latin1=1)
```

## 8. ref

当使用索引列等值查询时，与索引列进行等值匹配的对象信息。

显示索引的哪一列被使用了，如果可能的话，是一个常数。哪些列或常量被用于查找索引列上的值。

当使用索引列等值匹配的条件去执行查询时，也就是在访问方法是`const`、`eq_ref`、`ref`、`ref_or_null`、`unique_subquery`、`index_subquery`其中之一时，`ref`列展示的就是与索引列作等值匹配的结构是什么，比如只是一个常数或者是某个列。大家看下边这个查询：

① 比如只是一个常数或者是某个列，其 `ref` 是 `const`

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 = 'a';
```

```
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------+
| id | select_type | table | partitions | type | possible_keys | key      | key_len | ref   | rows | filtered | Extra |
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------+
|  1 | SIMPLE      | s1    | NULL       | ref  | idx_key1      | idx_key1 | 303     | const |    8 |   100.00 | NULL  |
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------+
1 row in set, 1 warning (0.01 sec)
```

可以看到`ref`列的值是`const`，表明在使用`idx_key1`索引执行查询时，与`key1`列作等值匹配的对象是一个常数，当然有时候更复杂一点：

当进行多表连接查询时，对被驱动表s2执行的查询引用了atguigudb1.s1.id字段进行等值查询

```
mysql> EXPLAIN SELECT * FROM s1 INNER JOIN s2 ON s1.id = s2.id;
```

```
+----+-------------+-------+------------+--------+---------------+---------+---------+---------------+------+----------+-------+
| id | select_type | table | partitions | type   | possible_keys | key     | key_len | ref           | rows | filtered | Extra |
+----+-------------+-------+------------+--------+---------------+---------+---------+---------------+------+----------+-------+
|  1 | SIMPLE      | s1    | NULL       | ALL    | PRIMARY       | NULL    | NULL    | NULL          | 9688 |   100.00 | NULL  |
|  1 | SIMPLE      | s2    | NULL       | eq_ref | PRIMARY       | PRIMARY | 4       | atguigu.s1.id |    1 |   100.00 | NULL  |
+----+-------------+-------+------------+--------+---------------+---------+---------+---------------+------+----------+-------+
2 rows in set, 1 warning (0.00 sec)
```

当连接条件使用函数时，其`ref`就是`func`

```
mysql> EXPLAIN SELECT * FROM s1 INNER JOIN s2 ON s2.key1 = UPPER(s1.key1);
```

```
+----+-------------+-------+------------+------+---------------+----------+---------+------+------+----------+-----------------------+
| id | select_type | table | partitions | type | possible_keys | key      | key_len | ref  | rows | filtered | Extra                 |
+----+-------------+-------+------------+------+---------------+----------+---------+------+------+----------+-----------------------+
|  1 | SIMPLE      | s1    | NULL       | ALL  | NULL          | NULL     | NULL    | NULL | 9688 |   100.00 | NULL                  |
|  1 | SIMPLE      | s2    | NULL       | ref  | idx_key1      | idx_key1 | 303     | func |    1 |   100.00 | Using index condition |
+----+-------------+-------+------------+------+---------------+----------+---------+------+------+----------+-----------------------+
2 rows in set, 1 warning (0.00 sec)
```

## 9. rows ☆

预估的需要读取的记录条目数，条目数越小越好。这是因为值越小，加载I/O的页数就越少~

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 > 'z';
```

```
+----+-------------+-------+------------+-------+---------------+----------+---------+------+------+----------+-----------------------+
| id | select_type | table | partitions | type  | possible_keys | key      | key_len | ref  | rows | filtered | Extra                 |
+----+-------------+-------+------------+-------+---------------+----------+---------+------+------+----------+-----------------------+
|  1 | SIMPLE      | s1    | NULL       | range | idx_key1      | idx_key1 | 303     | NULL |  266 |   100.00 | Using index condition |
+----+-------------+-------+------------+-------+---------------+----------+---------+------+------+----------+-----------------------+
1 row in set, 1 warning (0.00 sec)
```

## 10. filtered

经过搜索条件后过滤剩下的记录所占的百分比。百分比越高越好，比如同样 rows 是 40，如果 filter 是 100，则是从 40 条记录里进行查找，如果 filter 是 10，则是从 400 条记录里进行查找，相比较而言当然是前者的效率更高哦。

① 如果执行的是单表扫描，那么计算时需要估计除了对应搜索条件外的其他搜索条件满足的记录有多少条 晕了就看看下面的例子

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 > 'z' AND common_field = 'a';
```

结果是 10，表示有 347 条记录满足 key1 > 'z' 的条件，这 347 条记录的 10% 满足 common_field = 'a' 条件。

```
+----+-------------+-------+------------+-------+---------------+----------+---------+-----+------+----------+-----------------------------------+
| id | select_type | table | partitions | type  | possible_keys | key      | key_len | ref | rows | filtered | Extra                             |
+----+-------------+-------+------------+-------+---------------+----------+---------+-----+------+----------+-----------------------------------+
|  1 | SIMPLE      | s1    | NULL       | range | idx_key1      | idx_key1 | 303     | NULL |  266 |    10.00 | Using index condition; Using where |
+----+-------------+-------+------------+-------+---------------+----------+---------+-----+------+----------+-----------------------------------+
1 row in set, 1 warning (0.00 sec)
```

实际上，对于单表查询来说，这个filtered列的值没什么意义，我们更关注在连接查询中驱动表对应的执行计划记录的`filtered`值，它决定了被驱动表要执行的次数(即：rows * filtered)

```
mysql> EXPLAIN SELECT * FROM s1 INNER JOIN s2 ON s1.key1 = s2.key1 WHERE s1.common_field = 'a';
```

结果如下。在标明驱动表 s1 提供给被驱动表的记录数是 9895 条，其中 989.5 条满足过滤条件s1.key1 = s2.key1，那么被驱动表需要执行 990 次查询。

```
+----+-------------+-------+------------+------+---------------+----------+---------+-------------------+------+----------+-------------+
| id | select_type | table | partitions | type | possible_keys | key      | key_len | ref               | rows | filtered | Extra       |
+----+-------------+-------+------------+------+---------------+----------+---------+-------------------+------+----------+-------------+
|  1 | SIMPLE      | s1    | NULL       | ALL  | idx_key1      | NULL     | NULL    | NULL              | 9688 |    10.00 | Using where |
|  1 | SIMPLE      | s2    | NULL       | ref  | idx_key1      | idx_key1 | 303     | xiaohaizi.s1.key1 |    1 |   100.00 | NULL        |
+----+-------------+-------+------------+------+---------------+----------+---------+-------------------+------+----------+-------------+
2 rows in set, 1 warning (0.00 sec)
```

从执行计划中可以看出来，查询优化器打算把 `s1` 当作驱动表，`s2` 当作被驱动表。我们可以看到驱动表 `s1` 表的执行计划的 `rows` 列为 `9688`，`filtered` 列为 `10.00`，这意味着驱动表 `s1` 的扇出值就是 `9688×10.00%=968.8`，这说明还要对被驱动表执行大约 `968` 次查询。

> filtered=(最终查询结果/rows列数据)*100%，越大表示过滤后的数据，越是最终结果。
>
> 相比较filtered越小，减少了数据再次过滤的性能

## 11. Extra ☆

顾名思义，Extra 列是用来说明一些额外信息的，包含不适合在其他列中显示但十分重要的额外信息。我们可以通过这些额外信息来更准确的理解MySQL到底将如何执行给定的查询语句。MySQL提供的额外信息有好几十个，我们就不一一介绍了，所以我们只挑比较重要的额外信息介绍给大家。

- `No tables used`

  当查询语句的没有 `FROM` 子句时将会提示该额外信息

```
mysql> EXPLAIN SELECT 1;
```

```
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra          |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------+
|  1 | SIMPLE      | NULL  | NULL       | NULL | NULL          | NULL | NULL    | NULL | NULL |     NULL | No tables used |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------+
1 row in set, 1 warning (0.00 sec)
```

- `Impossible WHERE`

当查询条件永远不可能满足，查不到数据时会出现该信息。

```
mysql> EXPLAIN SELECT * FROM s1 WHERE 1 != 1;
```

```
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-----------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra           |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-----------------+
|  1 | SIMPLE      | NULL  | NULL       | NULL | NULL          | NULL | NULL    | NULL | NULL |     NULL | Impossible WHERE |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-----------------+
1 row in set, 1 warning (0.01 sec)
```

- `Using where`

  不用读取表中所有信息，仅通过索引就可以获取所需数据，这发生在对表的全部的请求列都是同一个索引的部分的时候，表示mysql服务器将在存储引擎检索行后再进行过滤。表明使用了where过滤。

  当我们使用全表扫描来执行对某个表的查询，并且该语句的`WHERE`子句中有针对该表的搜索条件时，在`Extra`列中会提示上述额外信息。比如下边这个查询：

  - 当没有使用索引，普通的 where 查询时，会出现该信息

```
1 mysql> EXPLAIN SELECT * FROM s1 WHERE common_field = 'a';
```

```
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra       |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------------+
|  1 | SIMPLE      | s1    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 9688 |    10.00 | Using where |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------------+
1 row in set, 1 warning (0.01 sec)
```

  当使用索引访问来执行对某个表的查询，并且该语句的`WHERE`子句中有除了该索引包含的列之外的其他搜索条件时，在`Extra`列中也会提示上述额外信息。比如下边这个查询虽然使用`idx_key1`索引执行查询，但是搜索条件中除了包含`key1`的搜索条件`key1='a'`，还包含`common_field`的搜索条件，所以`Extra`列会显示`Using where`的提示:

  - 使用索引查询，则默默使用索引，什么额外信息也没有。

```
1 EXPLAIN SELECT * FROM s1 WHERE key1 = 'a';
```

```
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------------+
| id | select_type | table | partitions | type | possible_keys | key      | key_len | ref   | rows | filtered | Extra       |
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------------+
|  1 | SIMPLE      | s1    | NULL       | ref  | idx_key1      | idx_key1 | 303     | const |    8 |    10.00 | Using where |
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------------+
1 row in set, 1 warning (0.00 sec)
```

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 = 'a';
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------+
| id | select_type | table | partitions | type | possible_keys | key      | key_len | ref   | rows | filtered | Extra |
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------+
|  1 | SIMPLE      | s1    | NULL       | ref  | idx_key1      | idx_key1 | 303     | const |    1 |   100.00 | NULL  |
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------+
1 row in set, 1 warning (0.00 sec)
```

  - 索引加普通 where，那还是 using where

    当使用索引访问来执行对某个表的查询，并且该语句的`WHERE`子句中有除了该索引包含的列之外的其他搜索条件时，在`Extra`列中也会提示上述额外信息。

```
1 mysql> EXPLAIN SELECT * FROM s1 WHERE key1 = 'a' AND common_field = 'a';
```

```
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------------+
| id | select_type | table | partitions | type | possible_keys | key      | key_len | ref   | rows | filtered | Extra       |
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------------+
|  1 | SIMPLE      | s1    | NULL       | ref  | idx_key1      | idx_key1 | 303     | const |    8 |    10.00 | Using where |
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------------+
1 row in set, 1 warning (0.00 sec)
```

- `No matching min/max row`

  当查询语句中有`MIN`、`MAX`等聚合函数，但是并没有符合`where`条件的搜索记录时，会提供额外信息`No matching min/max row`（表中根本没有满足 where 条件的字句，找 min、max 没有意义）

```
mysql> EXPLAIN SELECT MIN(key1) FROM s1 WHERE key1 = 'abcdefg';
```

```
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------------------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra                   |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------------------------+
|  1 | SIMPLE      | NULL  | NULL       | NULL | NULL          | NULL | NULL    | NULL | NULL |     NULL | No matching min/max row |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------------------------+
1 row in set, 1 warning (0.00 sec)
```

- `Select tables optimized away`

  当查询语句中有 MIN、MAX 等聚合函数，有符合 where 条件的搜索记录时

```
EXPLAIN SELECT MIN(key1) FROM s1 WHERE key1 = 'oCUPss';
```

```
mysql> EXPLAIN SELECT MIN(key1) FROM s1 WHERE key1 = 'oCUPss';
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+--------------------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra                    |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+--------------------------+
|  1 | SIMPLE      | NULL  | NULL       | NULL | NULL          | NULL | NULL    | NULL | NULL |     NULL | Select tables optimized away |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+--------------------------+
1 row in set, 1 warning (0.00 sec)
```

- `Using index`

  在使用覆盖索引的情况提示。所谓覆盖索引，就是索引中覆盖了需要查询的所有字段，不需要再使用聚簇索引进行回表查找。比如下面的例子，使用 key1 作为查找条件，该字段建立了索引，B+ 树可以查找到 key1 字段和主键，因此下面只查找 key1 字段就不用进行回表操作，这是非常棒的情况。

```
mysql> EXPLAIN SELECT key1 FROM s1 WHERE key1 = 'a';
```

```
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------------+
| id | select_type | table | partitions | type | possible_keys | key      | key_len | ref   | rows | filtered | Extra       |
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------------+
|  1 | SIMPLE      | s1    | NULL       | ref  | idx_key1      | idx_key1 | 303     | const |    8 |   100.00 | Using index |
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------------+
1 row in set, 1 warning (0.00 sec)
```

- `Using index condition`

  有些搜索条件中虽然出现了索引列，但却不能使用到索引，比如下边这个查询：

```
SELECT * FROM s1 WHERE key1 > 'z' AND key1 LIKE '%a';
```

  其中的`key1 > 'z'`可以使用到索引，但是`key1 LIKE '%a'`却无法使用到索引，在以前版本的MySQL中，是按照下边步骤来执行这个查询的：

  - 先根据`key1 > 'z'`这个条件，从二级索引`idx_key1`中获取到对应的二级索引记录。
  - 根据上一步骤得到的二级索引记录中的主键值进行`回表`，找到完整的用户记录再检测该记录是否符合`key1 LIKE '%a'`这个条件，将符合条件的记录加入到最后的结果集。

  但是虽然`key1 LIKE '%a'`不能组成范围区间参与`range`访问方法的执行，但这个条件毕竟只涉及到了`key1`列，所以MySQL把上边的步骤改进了一下：

  - 先根据`key1 > 'z'`这个条件，定位到二级索`idx_key1`中对应的二级索引记录。
  - 对于指定的二级索引记录，先不着急回表，而是先检测一下该记录是否满足`key1 LIKE '%a'`这个条件，如果这个条件不满足，则该二级索引记录压根儿就没必要回表。

- 对于满足`key1 LIKE '%a'`这个条件的二级索引记录执行回表操作。

我们说回表操作其实是一个`随机IO`，比较耗时，所以上述修改虽然只改进了一点点，但是可以省去好多回表操作的成本。MySQL把他们的这个改进称之为`索引条件下推`（英文名：`Index Condition Pushdown`）。如果在查询语句的执行过程中将要使用`索引条件下推`这个特性，在Extra列中将会显示`Using index condition`。

搜索列中虽然出现了索引列，但是不能够使用索引，这种情况是比较坑的~

比如下面的查询虽然出现了索引列作为查询条件，但是还是需要进行回表查找，回表操作是一个随机 I/O，比较耗时。

```
SELECT * FROM s1 WHERE key1 > 'z' AND key1 LIKE '%a';
```

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 > 'z' AND key1 LIKE '%b';
```

```
+----+-------------+-------+------------+-------+---------------+----------+---------+------+------+----------+-----------------------+
| id | select_type | table | partitions | type  | possible_keys | key      | key_len | ref  | rows | filtered | Extra                 |
+----+-------------+-------+------------+-------+---------------+----------+---------+------+------+----------+-----------------------+
|  1 | SIMPLE      | s1    | NULL       | range | idx_key1      | idx_key1 | 303     | NULL |  266 |   100.00 | Using index condition |
+----+-------------+-------+------------+-------+---------------+----------+---------+------+------+----------+-----------------------+
1 row in set, 1 warning (0.01 sec)
```

上面这种情况可以使用**索引下推**(可以通过配置项进行配置)，使我们使用 WHERE key1 > 'z' 得到的结果先进行模糊匹配 key1 LIKE '%a'，然后再去回表，就可以减少回表的次数了。

- `Using join buffer (Block Nested Loop)`

在连接查询执行过程中，当被驱动表不能有效的利用索引加快访问速度，MySQL一般会为其分配一块名叫`join buffer`的内存块来加快查询速度，也就是我们所讲的`基于块的嵌套循环算法`

在连接查询中，当被驱动表不能够有效利用索引实现提升速度，数据库就使用缓存来尽可能提升一些性能。

```
mysql> EXPLAIN SELECT * FROM s1 INNER JOIN s2 ON s1.common_field =
s2.common_field;
```

```
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-----------------------------------------------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra                                               |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-----------------------------------------------------+
|  1 | SIMPLE      | s1    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 9688 |   100.00 | NULL                                                |
|  1 | SIMPLE      | s2    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 9954 |    10.00 | Using where; Using join buffer (Block Nested Loop) |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-----------------------------------------------------+
2 rows in set, 1 warning (0.03 sec)
```

可以在对s2表的执行计划的Extra列显示了两个提示：

- `Using join buffer(Block Nested Loop)`：这是因为对表s2的访问不能有效利用索引，只好退而求其次，使用`join buffer`来减少对s2表的访问次数，从而提高性能。
- Using where：可以看到查询语句中有一个`s1.common_field = s2.common_field`条件，因为s1是驱动表，s2是被驱动表，所以在访问s2表时，`s1.common_field`的值已经确定下来了，所以实际上查询s2表的条件就是`s2.common_fied=一个常数`，所以提示了`Using where`额外信息。

- `Not exists`

当我们使用左（外）连接时，如果`WHERE`子句中包含要求被驱动表的某个列等于`NULL`值的搜索条件，而且那个列又是不允许存储`NULL`值的，那么在该表的执行计划的Extra列就会提示`Not exists`额外信息

```
mysql> EXPLAIN SELECT * FROM s1 LEFT JOIN s2 ON s1.key1 = s2.key1 WHERE
s2.id IS NULL;
```

```
+----+-------------+-------+------------+------+---------------+---------+---------+-------------------+------+----------+-------------------------+
| id | select_type | table | partitions | type | possible_keys | key     | key_len | ref               | rows | filtered | Extra                   |
+----+-------------+-------+------------+------+---------------+---------+---------+-------------------+------+----------+-------------------------+
|  1 | SIMPLE      | s1    | NULL       | ALL  | NULL          | NULL    | NULL    | NULL              | 9688 |   100.00 | NULL                    |
|  1 | SIMPLE      | s2    | NULL       | ref  | idx_key1      | idx_key1 | 303    | xiaohaizi.s1.key1 |    1 |    10.00 | Using where; Not exists |
+----+-------------+-------+------------+------+---------------+---------+---------+-------------------+------+----------+-------------------------+
2 rows in set, 1 warning (0.00 sec)
```

上述查询中s1表是驱动表，s2表是被驱动表，`s2.id`列是不允许存储`NULL`值的，而`WHERE`子句中又包含`s2.id IS NULL`的搜索条件，这意味着必定是驱动表的记录在被驱动表中找不到匹配`ON`子句条件的记录才会把该驱动表的记录加入到最终的结果集，所以对于某条驱动表中的记录来说，如果能在被驱动表中找到1条符合`ON`子句条件的记录，那么该驱动表的记录就不会被加入到最终的结果集，也就是说我们没有必要到被驱动表中找到全部符合ON子句条件的记录，这样可以稍微节省一点性能。

> 小贴士：右（外）连接可以被转换为左（外）连接，所以就不提右（外）连接的情况了。

- `Using intersect(...)`、`Using union(...)`和`Using sort_union(...)`
  - 如果执行计划的Extra列出现了Using intersect(...)提示，说明准备使用Intersect索引合并的方式执行查询，括号中的`...`表示需要进行索引合并的索引名称；
  - 如果出现了Using union(...)提示，说明准备使用Union索引合并的方式执行查询；
  - 如果出现了Using sort_union(...)提示，说明准备使用Sort-Union索引合并的方式执行查询

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 = 'a' OR key3 = 'a';
```

```
+----+-------------+-------+------------+-------------+-----------------+-----------------+---------+------+------+----------+-----------------------------------------------------+
| id | select_type | table | partitions | type        | possible_keys   | key             | key_len | ref  | rows | filtered | Extra                                               |
+----+-------------+-------+------------+-------------+-----------------+-----------------+---------+------+------+----------+-----------------------------------------------------+
|  1 | SIMPLE      | s1    | NULL       | index_merge | idx_key1,idx_key3 | idx_key1,idx_key3 | 303,303 | NULL |    2 |   100.00 | Using union(idx_key1,idx_key3); Using where |
+----+-------------+-------+------------+-------------+-----------------+-----------------+---------+------+------+----------+-----------------------------------------------------+
1 row in set, 1 warning (0.00 sec)
```

其中Extra列就显示了`Using union(idx_key3,idx_key1)`，表明`MySQL`即将使用`idx_key3`和`idx_key1`这两个索引进行`Union`索引合并的方式执行查询。

- `Zero limit`

当我们的`LIMIT`子句的参数为`0`时，表示压根儿不打算从表中读出任何记录，将会提示该额外信息

```
mysql> EXPLAIN SELECT * FROM s1 LIMIT 0;
```

```
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra      |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+------------+
|  1 | SIMPLE      | NULL  | NULL       | NULL | NULL          | NULL | NULL    | NULL | NULL |     NULL | Zero limit |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+------------+
1 row in set, 1 warning (0.00 sec)
```

- `Using filesort`

很多情况下排序操作无法使用到索引，只能在内存中（记录较少的时候）或者磁盘中（记录较多的时候）进行排序，MySQL把这种在内存中或者磁盘上进行排序的方式统称为**文件排序**（英文名：`filesort`）。这种情况时比较悲壮的~

```
#有一些情况下对结果集中的记录进行排序是可以使用到索引的。
mysql> EXPLAIN SELECT * FROM s1 ORDER BY key1 LIMIT 10;
```

```
+----+-------------+-------+------------+-------+---------------+----------+---------+------+------+----------+-------+
| id | select_type | table | partitions | type  | possible_keys | key      | key_len | ref  | rows | filtered | Extra |
+----+-------------+-------+------------+-------+---------------+----------+---------+------+------+----------+-------+
|  1 | SIMPLE      | s1    | NULL       | index | NULL          | idx_key1 | 303     | NULL |   10 |   100.00 | NULL  |
+----+-------------+-------+------------+-------+---------------+----------+---------+------+------+----------+-------+
1 row in set, 1 warning (0.03 sec)
```

这个查询语句可以利用`idx_key1`索引直接取出`key1`列的10条记录，然后再进行回表操作就好了。但是很多情况下排序操作无法使用到索引，只能在内存中（记录较少的时候）或者磁盘中（记录较多的时候）进行排序，MySQL把这种在内存中或者磁盘上进行排序的方式统称为文件排序（英文名：`filesort`）。如果某个查询需要使用文件排序的方式执行查询，就会在执行计划的`Extra`列中显示`Using filesort`提示，比如这样：

```
#如果某个查询需要使用文件排序的方式执行查询，就会在执行计划的`Extra`列中显示`Using filesort`提示
mysql> EXPLAIN SELECT * FROM s1 ORDER BY common_field LIMIT 10;
```

```
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra          |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------+
|  1 | SIMPLE      | s1    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 9688 |   100.00 | Using filesort |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------+
1 row in set, 1 warning (0.00 sec)
```

需要注意的是，如果查询中需要使用`filesort`的方式进行排序的记录非常多，那么这个过程是很耗费性能的，我们最好想办法`将使用文件排序的执行方式改为使用索引进行排序`。

- `Using temporary`

在许多查询的执行过程中，MySQL可能会借助临时表来完成一些功能，比如`去重、排序`之类的，比如我们在执行许多包含`DISTINCT`、`GROUP BY`、`UNION`等子句的查询过程中，如果不能有效利用索引来完成查询，MySQL很有可能寻求通过建立内部的临时表来执行查询。如果查询中使用到了内部的临时表，在执行计划的`Extra`列将会显示`Using temporary`提示

```
mysql> EXPLAIN SELECT DISTINCT common_field FROM s1;
```

```
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-----------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra           |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-----------------+
|  1 | SIMPLE      | s1    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 9688 |   100.00 | Using temporary |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-----------------+
1 row in set, 1 warning (0.00 sec)
```

再比如：

```
mysql> EXPLAIN SELECT common_field, COUNT(*) AS amount FROM s1 GROUP BY common_field;
```

```
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-----------------+
| id | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra           |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-----------------+
|  1 | SIMPLE      | s1    | NULL       | ALL  | NULL          | NULL | NULL    | NULL | 9688 |   100.00 | Using temporary |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-----------------+
1 row in set, 1 warning (0.00 sec)
```

执行计划中出现`Using temporary`并不是一个好的征兆，因为建立与维护临时表要付出很大成本的，所以我们`最好能使用索引来替代掉使用临时表`。比如：扫描指定的索引idx_key1即可

```
mysql> EXPLAIN SELECT key1, COUNT(*) AS amount FROM s1 GROUP BY key1;
```

```
+----+-------------+-------+------------+-------+---------------+----------+---------+------+------+----------+-------------+
| id | select_type | table | partitions | type  | possible_keys | key      | key_len | ref  | rows | filtered | Extra       |
+----+-------------+-------+------------+-------+---------------+----------+---------+------+------+----------+-------------+
|  1 | SIMPLE      | s1    | NULL       | index | idx_key1      | idx_key1 | 303     | NULL | 9688 |   100.00 | Using index |
+----+-------------+-------+------------+-------+---------------+----------+---------+------+------+----------+-------------+
1 row in set, 1 warning (0.00 sec)
```

从`Extra`的`Using index`的提示里我们可以看出，上述查询只需要扫描idx_key1 索引就可以搞定了，不再需要临时表了。

- 其他
  其它特殊情况这里省略。

### 12. 小结

- EXPLAIN不考虑各种Cache
- EXPLAIN不能显示MySQL在执行查询时所作的优化工作
- EXPLAIN不会告诉你关于触发器、存储过程的信息或用户自定义函数对查询的影响情况
- 部分统计信息是估算的，并非精确值

# 7. EXPLAIN的进一步使用

## 7.1 EXPLAIN四种输出格式

这里谈谈EXPLAIN的输出格式。EXPLAIN可以输出四种格式： `传统格式` ， `JSON格式` ， `TREE格式` 以及 `可视化输出` 。用户可以根据需要选择适用于自己的格式。

### 1. 传统格式

传统格式简单明了，输出是一个表格形式，概要说明查询计划。

```
mysql> EXPLAIN SELECT s1.key1, s2.key1 FROM s1 LEFT JOIN s2 ON s1.key1 =
s2.key1 WHERE s2.common_field IS NOT NULL;
```

```
+----+-------------+-------+------------+------+---------------+----------+---------+-------------------+------+----------+-------------+
| id | select_type | table | partitions | type | possible_keys | key      | key_len | ref               | rows | filtered | Extra       |
+----+-------------+-------+------------+------+---------------+----------+---------+-------------------+------+----------+-------------+
|  1 | SIMPLE      | s2    | NULL       | ALL  | idx_key1      | NULL     | NULL    | NULL              | 9954 |    90.00 | Using where |
|  1 | SIMPLE      | s1    | NULL       | ref  | idx_key1      | idx_key1 | 303     | xiaohaizi.s2.key1 |    1 |   100.00 | Using index |
+----+-------------+-------+------------+------+---------------+----------+---------+-------------------+------+----------+-------------+
2 rows in set, 1 warning (0.00 sec)
```

### 2. JSON格式

第1种格式中介绍的 `EXPLAIN` 语句输出中缺少了一个衡量执行计划好坏的重要属性-成本。 而JSON格式是四种格式里面输出信息最详尽的格式，里面包含了执行的成本信息。

- JSON格式：在EXPLAIN单词和真正的查询语句中间加上 `FORMAT=JSON` 。

  ```
  EXPLAIN FORMAT = JSON SELECT ...
  ```

传统格式与json格式的各个字段存在如下表所示的对应关系(mysql5.7官方文档)。

| Column | JSON Name | Meaning |
| --- | --- | --- |
| `id` | select_id | The SELECT identifier |
| select_type | None | The SELECT type |
| table | table_name | The table for the output row |
| partitions | partitions | The matching partitions |
| `type` | access_type | The join type |

| Column | JSON Name | Meaning |
| --- | --- | --- |
| possible_keys | possible_key | The possible indexes to choose |
| key | key | The index actually chosen |
| `key_len` | key_length | The length of the chosen key |
| `ref` | ref | The columns compared to the index |
| rows | rows | Estimate of rows to be examined |
| filtered | filtered | Percentage of rows filtered by table condition |
| `Extra` | None | Additional information |

结果如下，可以看到 json 格式的信息量会更加丰富。尤其是成本信息，是用于衡量一个执行计划的好坏的重要指标

```
mysql> EXPLAIN FORMAT=JSON SELECT * FROM s1 INNER JOIN s2 ON s1.key1 = s2.key2 WHERE s1.common_field ='a'\G;
*************************** 1. row ***************************
EXPLAIN: {
  "query_block": {
    "select_id": 1,#整个查询语句只有1个SELECT关键字，该关键字对应的id号为1
    "cost_info": {
      "query_cost": "1360.07" #整个查询的执行成本预计为3197.16
    },
    "nested_loop": [#几个表之间采用嵌套循环连接算法执行
        #以下是参与嵌套循环连接算法的各个表的信息
      {
        "table": {
          "table_name": "s1",#s1表是驱动表
          "access_type": "ALL",#访问方法为ALL，意味着使用全表扫描访问
          "possible_keys": [#可能使用的索引
            "idx_key1"
          ],
          "rows_examined_per_scan": 9895,#查询一次s1表大致需要扫描9688条记录
          "rows_produced_per_join": 989,#驱动表s1的扇出是968
          "filtered": "10.00",#condition filtering代表的百分比
          "cost_info": {
            "read_cost": "914.80",
            "eval_cost": "98.95",
            "prefix_cost": "1013.75",#单次查询s1表总共的成本
            "data_read_per_join": "1M"#读取的数据量
          },
          "used_columns": [#执行查询中涉及到的列
            "id",
            "key1",
            "key2",
            "key3",
            "key_part1",
            "key_part2",
            "key_part3",
            "common_field"
          ],
```

```
        #对s1表访问时针对单表查询的条件
          "attached_condition": "((`atguigudb1`.`s1`.`common_field` = 'a') and (`atguigudb1`.`s1`.`key1` is not null))"
        }
      },
      {
        "table": {
          "table_name": "s2",#s2表是被驱动表
          "access_type": "eq_ref",
          "possible_keys": [#可能使用的索引
            "idx_key2"
          ],
          "key": "idx_key2",#实际使用的索引
          "used_key_parts": [#用到的索引列
            "key2"
          ],
          "key_length": "5",#key_len
          "ref": [      #与key2列进行等值匹配的对象
            "atguigudb1.s1.key1"#被驱动表s2的扇出是968（由于后边没有多余的表进行连接，所以这个值也没啥用）
          ],
          "rows_examined_per_scan": 1,#查询一次s2表大致需要扫描1条记录
          "rows_produced_per_join": 989,
          "filtered": "100.00",#conditionfiltering代表的百分比
          "index_condition": "(cast(`atguigudb1`.`s1`.`key1` as double) = cast(`atguigudb1`.`s2`.`key2` as double))",
          "cost_info": {
            "read_cost": "247.38",
            "eval_cost": "98.95",
            "prefix_cost": "1360.08",#单次查询s1、多次查询s2表息共的成本
            "data_read_per_join": "1M"#读取的数据量
          },
          "used_columns": [#执行查询中及到的列
            "id",
            "key1",
            "key2",
            "key3",
            "key_part1",
            "key_part2",
            "key_part3",
            "common_field"
          ]
        }
      }
    ]
  }
}
1 row in set, 2 warnings (0.00 sec)
```

我们使用 `#` 后边跟随注释的形式为大家解释了 `EXPLAIN FORMAT=JSON` 语句的输出内容，但是大家可能有疑问 `"cost_info"` 里边的成本看着怪怪的，它们是怎么计算出来的？先看 `s1` 表的 `"cost_info"` 部分：

```
"cost_info": {
    "read_cost": "1840.84",
    "eval_cost": "193.76",
    "prefix_cost": "2034.60",
    "data_read_per_join": "1M"
}
```

- `read_cost` 是由下边这两部分组成的：
  - `IO` 成本
  - 检测 `rows × (1 - filter)` 条记录的 `CPU` 成本

> 小贴士： rows和filter都是我们前边介绍执行计划的输出列，在JSON格式的执行计划中，rows
> 相当于rows_examined_per_scan，filtered名称不变。

- `eval_cost` 是这样计算的：

  检测 `rows × filter` 条记录的成本。
- `prefix_cost` 就是单独查询 `s1` 表的成本，也就是：`read_cost + eval_cost`
- `data_read_per_join` 表示在此次查询中需要读取的数据量。

对于 `s2` 表的 `"const_info"`

```
"cost_info": {
    "read_cost": "968.80",
    "eval_cost": "193.76",
    "prefix_cost": "3197.16",
    "data_read_per_join": "1M"
}
```

由于 `s2` 表是被驱动表，所以可能被读取多次，这里的 `read_cost` 和 `eval_cost` 是访问多次 `s2` 表后累加起来的值，大家主要关注里边儿的 `prefix_cost` 的值代表的是整个连接查询预计的成本，也就是单次查询 `s1` 表和多次查询 `s2` 表后的成本的和，也就是：

```
968.80 + 193.76 + 2034.60 = 3197.16
```

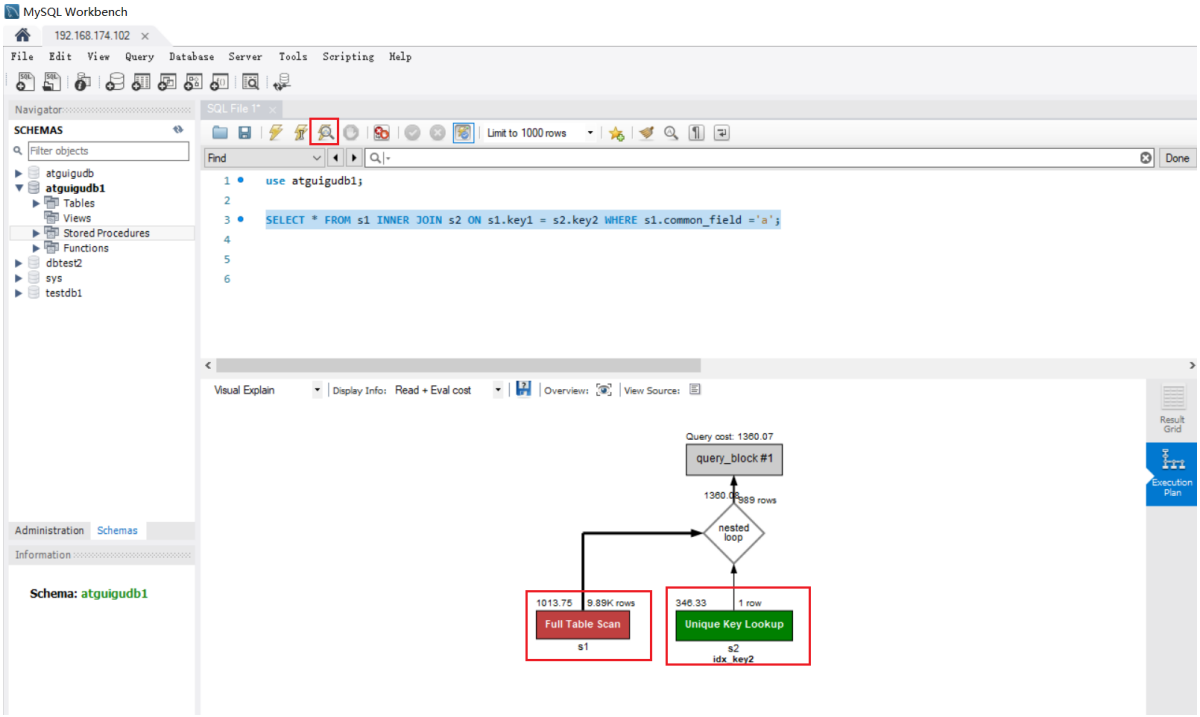
### 3. TREE格式

TREE格式是8.0.16版本之后引入的新格式，主要根据查询的 `各个部分之间的关系` 和 `各部分的执行顺序` 来描述如何查询。

```
mysql> EXPLAIN FORMAT=tree SELECT * FROM s1 INNER JOIN s2 ON s1.key1 =
s2.key2 WHERE s1.common_field = 'a'\G
*************************** 1. row ***************************
EXPLAIN: -> Nested loop inner join (cost=1360.08 rows=990)
    -> Filter: ((s1.common_field = 'a') and (s1.key1 is not null))
(cost=1013.75
    rows=990)
        -> Table scan on s1 (cost=1013.75 rows=9895)
    -> Single-row index lookup on s2 using idx_key2 (key2=s1.key1), with
index condition: (cast(s1.key1 as double) = cast(s2.key2 as double))
(cost=0.25 rows=1)
1 row in set, 1 warning (0.00 sec)
```

### 4. 可视化输出

可视化输出，可以通过MySQL Workbench可视化查看MySQL的执行计划。通过点击Workbench的放大镜图标，即可生成可视化的查询计划。




上图按从左到右的连接顺序显示表。红色框表示 `全表扫描` ，而绿色框表示使用 `索引查找` 。对于每个表，显示使用的索引。还要注意的是，每个表格的框上方是每个表访问所发现的行数的估计值以及访问该表的成本。

## 7.2SHOW WARNINGS的使用

在我们使用 `EXPLAIN` 语句查看了某个查询的执行计划后，紧接着还可以使用 `SHOW WARNINGS` 语句查看与这个查询的执行计划有关的一些扩展信息。

可以显示数据库真正执行的 SQL ，因为有时候MySQL执行引擎会对我们的SQL进行优化~

1. 先使用 `Explain` ，我们写的 sql 按道理是使用 s1 作为驱动表，s2作为被驱动表

```
mysql> EXPLAIN SELECT s1.key1, s2.key1 FROM s1 LEFT JOIN s2 ON s1.key1 =
s2.key1 WHERE s2.common_field IS NOT NULL;
```

但是 执行结果把 s2 作为了驱动表，s1 作为了被驱动表

```
+----+-------------+-------+------------+------+---------------+----------+---------+-------------------+------+----------+-------------+
| id | select_type | table | partitions | type | possible_keys | key      | key_len | ref               | rows | filtered | Extra       |
+----+-------------+-------+------------+------+---------------+----------+---------+-------------------+------+----------+-------------+
|  1 | SIMPLE      | s2    | NULL       | ALL  | idx_key1      | NULL     | NULL    | NULL              | 9954 |    90.00 | Using where |
|  1 | SIMPLE      | s1    | NULL       | ref  | idx_key1      | idx_key1 | 303     | xiaohaizi.s2.key1 |    1 |   100.00 | Using index |
+----+-------------+-------+------------+------+---------------+----------+---------+-------------------+------+----------+-------------+
2 rows in set, 1 warning (0.00 sec)
```

紧接着使用`SHOW WARNINGS`，原来执行引擎将`LEFT JOIN`优化成了`INNER JOIN`

```
mysql> SHOW WARNINGS\G
*************************** 1. row ***************************
  Level: Note
   Code: 1003
Message: /* select#1 */ select `atguigu`.`s1`.`key1` AS
`key1`,`atguigu`.`s2`.`key1` AS `key1` from `atguigu`.`s1` join
`atguigu`.`s2` where ((`atguigu`.`s1`.`key1` = `atguigu`.`s2`.`key1`) and
(`atguigu`.`s2`.`common_field` is not null))
1 row in set (0.00 sec)
```

大家可以看到`SHOW WARNINGS`展示出来的信息有三个字段，分别是`Level`、`Code`、Message。我们最常见的就是Code为1003的信息，当Code值为1003时，`Message`字段展示的信息类似于查询优化器将我们的查询语句重写后的语句。比如我们上边的查询本来是一个左（外）连接查询，但是有一个`s2.common_field IS NOT NULL`的条件，这就会导致查询优化器把左（外）连接查询优化为内连接查询，从`SHOW WARNINGS`的`Message`字段也可以看出来，原本的LEFTJOIN已经变成了JOIN。

上面 message 中显示的是数据库优化、重写后真正执行的查询语句。果然它帮我们做了优化

再举一个例子：下面是一个 子查询SQL，应该对应着两个不同的id~

```
EXPLAIN SELECT * FROM s1 WHERE key1 IN (SELECT key2 FROM s2 WHERE
common_field = 'a');
```

但是真正执行后，对应着竟然是相同的id

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 IN (SELECT key2 FROM s2 WHERE common_field = 'a');
+----+-------------+-------+------------+--------+---------------+----------+---------+-------------------+------+----------+-----------------------------------------+
| id | select_type | table | partitions | type   | possible_keys | key      | key_len | ref               | rows | filtered | Extra                                   |
+----+-------------+-------+------------+--------+---------------+----------+---------+-------------------+------+----------+-----------------------------------------+
|  1 | SIMPLE      | s1    | NULL       | ALL    | idx_key1      | NULL     | NULL    | NULL              | 9895 |   100.00 | Using where                             |
|  1 | SIMPLE      | s2    | NULL       | eq_ref | idx_key2      | idx_key2 | 5       | atguigudb1.s1.key1 |    1 |    10.00 | Using index condition; Using where |
+----+-------------+-------+------------+--------+---------------+----------+---------+-------------------+------+----------+-----------------------------------------+
2 rows in set, 2 warnings (0.00 sec)
```

我们使用`SHOW WARNINGS\G;`进行分析，发现执行引擎将其优化成了**多表连接查询**的方式

```
mysql> SHOW WARNINGS\G;
*************************** 1. row ***************************
  Level: Warning
   Code: 1739
Message: Cannot use ref access on index 'idx_key1' due to type or collation
conversion on field 'key1'
*************************** 2. row ***************************
  Level: Note
   Code: 1003
Message: /* select#1 */ select `atguigudb1`.`s1`.`id` AS
`id`,`atguigudb1`.`s1`.`key1` AS `key1`,`atguigudb1`.`s1`.`key2` AS
`key2`,`atguigudb1`.`s1`.`key3` AS `key3`,`atguigudb1`.`s1`.`key_part1` AS
`key_part1`,`atguigudb1`.`s1`.`key_part2` AS
`key_part2`,`atguigudb1`.`s1`.`key_part3` AS
`key_part3`,`atguigudb1`.`s1`.`common_field` AS `common_field`
from `atguigudb1`.`s2`
```

```
join `atguigudb1`.`s1`
where ((`atguigudb1`.`s2`.`common_field` = 'a')
and (cast(`atguigudb1`.`s1`.`key1` as double) =
cast(`atguigudb1`.`s2`.`key2` as double)))
2 rows in set (0.00 sec)
```

# 8. 分析优化器执行计划：trace

`OPTIMIZE_TRACE`是 mysql 5.6 中引入的一个跟踪工具，它可以跟踪优化器做出的各种决策，比如访问表的方法，各种开销计算，各种转换，结果会被记录到`information_schema.optimizer_trace`中。

此功能默认关闭。开启trace，并设置格式为JSON，同时设置trace最大能够使用的内存大小，避免解析过程中因为默认内存过小而不能够完整展示。命令如下：

```
SET optimizer_trace="enabled=on",end_markers_in_json=on;
set optimizer_trace_max_mem_size=1000000;
```

开启后，可分析如下语句：

- SELECT
- INSERT
- REPLACE
- UPDATE
- DELETE
- EXPLAIN
- SET
- DECLARE
- CASE
- IF
- RETURN
- CALL

测试：执行如下SQL语句

```
select * from student where id < 10;
```

最后， 查询 information_schema.optimizer_trace 就可以知道MySQL是如何执行SQL的：

```
select * from information_schema.optimizer_trace\G
```

```
***************************1.row * ***************************
//第1部分：查询语句
QUERY: select * from student where id < 10
//第2部分：QUERY字段对应语句的跟踪信息
TRACE: {
    "steps": [{
        "join_preparation": { //预备工作
            "select#": 1,
            "steps": [{
```

```
                "expanded_query": "/* select#1 */ select `student`.`id` AS `id`,`student`.`stuno` AS `stuno`,`student`.`name` AS `name`,`student`.`age` AS `age`,`student`.`classId` AS `classId` from `student` where (`student`.`id` < 10)"
            }]
            /* steps */
        }
        /* join_preparation */
    },
    {
        "join_optimization": { //进行优化
            "select#": 1,
            "steps": [{
                "condition_processing": { //条件处理
                    "condition": "WHERE",
                    "original_condition": "(`student`.`id` < 10)",
                    "steps": [{
                        "transformation": "equality_propagation",
                        "resulting_condition": "(`student`.`id` < 10)"
                    },
                    {
                        "transformation": "constant_propagation",
                        "resulting_condition": "(`student`.`id` < 10)"
                    },
                    {
                        "transformation": "trivial_condition_removal",
                        "resulting_condition": "(`student`.`id` < 10)"
                    }]
                    /* steps */
                }
                /* condition_processing */
            },
            {
                "substitute_generated_columns": { //替换生成的列
                }
                /* substitute_generated_columns */
            },
            {
                "table_dependencies": [ //表的依赖关系
                {
                    "table": "`student`",
                    "row_may_be_null": false,
                    "map_bit": 0,
                    "depends_on_map_bits": []
                    /* depends_on_map_bits */
                }]
                /* table_dependencies */
            },
            {
                "ref_optimizer_key_uses": [ //使用键
                ]
                /* ref_optimizer_key_uses */
            },
            {
                "rows_estimation": [ //行判断
                {
```

```
                    "table": "`student`",
                    "range_analysis": {
                        "table_scan": {
                            "rows": 3973767,
                            "cost": 408558
                        }
                        /* table_scan */
                        ,
                        //扫描表
                        "potential_range_indexes": [ //潜在的范围索引
                        {
                            "index": "PRIMARY",
                            "usable": true,
                            "key_parts": ["id"]
                            /* key_parts */
                        }]
                        /* potential_range_indexes */
                        ,
                        "setup_range_conditions": [ //设置范围条件
                        ]
                        /* setup_range_conditions */
                        ,
                        "group_index_range": {
                            "chosen": false,
                            "cause": "not_group_by_or_distinct"
                        }
                        /* group_index_range */
                        ,
                        "skip_scan_range": {
                            "potential_skip_scan_indexes": [{
                                "index": "PRIMARY",
                                "usable": false,
                                "cause": "query_references_nonkey_column"
                            }]
                            /* potential_skip_scan_indexes */
                        }
                        /* skip_scan_range */
                        ,
                        "analyzing_range_alternatives": { //分析范围选项
                            "range_scan_alternatives": [{
                                "index": "PRIMARY",
                                "ranges": ["id < 10"]
                                /* ranges */
                                ,
                                "index_dives_for_eq_ranges": true,
                                "rowid_ordered": true,
                                "using_mrr": false,
                                "index_only": false,
                                "rows": 9,
                                "cost": 1.91986,
                                "chosen": true
                            }]
                            /* range_scan_alternatives */
                            ,
                            "analyzing_roworder_intersect": {
                                "usable": false,
```

```
                                "cause": "too_few_roworder_scans"
                            }
                            /* analyzing_roworder_intersect */
                        }
                        /* analyzing_range_alternatives */
                        ,
                        "chosen_range_access_summary": { //选择范围访问摘要
                            "range_access_plan": {
                                "type": "range_scan",
                                "index": "PRIMARY",
                                "rows": 9,
                                "ranges": ["id < 10"]
                                /* ranges */
                            }
                            /* range_access_plan */
                            ,
                            "rows_for_plan": 9,
                            "cost_for_plan": 1.91986,
                            "chosen": true
                        }
                        /* chosen_range_access_summary */
                    }
                    /* range_analysis */
                }]
                /* rows_estimation */
            },
            {
                "considered_execution_plans": [ //考虑执行计划
                {
                    "plan_prefix": []
                    /* plan_prefix */
                    ,
                    "table": "`student`",
                    "best_access_path": { //最佳访问路径
                        "considered_access_paths": [{
                            "rows_to_scan": 9,
                            "access_type": "range",
                            "range_details": {
                                "used_index": "PRIMARY"
                            }
                            /* range_details */
                            ,
                            "resulting_rows": 9,
                            "cost": 2.81986,
                            "chosen": true
                        }]
                        /* considered_access_paths */
                    }
                    /* best_access_path */
                    ,
                    "condition_filtering_pct": 100,
                    //行过滤百分比
                    "rows_for_plan": 9,
                    "cost_for_plan": 2.81986,
                    "chosen": true
                }]
```

```
                /* considered_execution_plans */
            },
            {
                "attaching_conditions_to_tables": { //将条件附加到表上
                    "original_condition": "(`student`.`id` < 10)",
                    "attached_conditions_computation": []
                    /* attached_conditions_computation */
                    ,
                    "attached_conditions_summary": [ //附加条件概要
                    {
                        "table": "`student`",
                        "attached": "(`student`.`id` < 10)"
                    }]
                    /* attached_conditions_summary */
                }
                /* attaching_conditions_to_tables */
            },
            {
                "finalizing_table_conditions": [{
                    "table": "`student`",
                    "original_table_condition": "(`student`.`id` < 10)",
                    "final_table_condition ": "(`student`.`id` < 10)"
                }]
                /* finalizing_table_conditions */
            },
            {
                "refine_plan": [ //精简计划
                {
                    "table": "`student`"
                }]
                /* refine_plan */
            }]
            /* steps */
        }
        /* join_optimization */
    },
    {
        "join_execution": { //执行
            "select#": 1,
            "steps": []
            /* steps */
        }
        /* join_execution */
    }]
    /* steps */
}
//第3部分：跟踪信息过长时，被截断的跟踪信息的字节数。
MISSING_BYTES_BEYOND_MAX_MEM_SIZE: 0 //丢失的超出最大容量的字节
//第4部分：执行跟踪语句的用户是否有查看对象的权限。当不具有权限时，该列信息为1且TRACE字段为空，一般在
调用带有SQL SECURITY DEFINER的视图或者是存储过程的情况下，会出现此问题。
INSUFFICIENT_PRIVILEGES: 0 //缺失权限
1 row in set(0.00 sec)
```

# 9. MySQL监控分析视图-sys schema

关于MySQL的性能监控和问题诊断，我们一般都从performance_schema中去获取想要的数据，在MySQL5.7.7版本中新增`sys schema`，它将performance_schema和information_schema中的数据以更容易理解的方式总结归纳为"视图"，其目的就是为了`降低查询performance_schema的复杂度`，让DBA能够快速的定位问题。下面看看这些库中都有哪些监控表和视图，掌握了这些，在我们开发和运维的过程中就起到了事半功倍的效果。

## 9.1 Sys schema视图摘要

**1. 主机相关：**以host_summary开头，主要汇总了IO延迟的信息。

**2. Innodb相关：**以innodb开头，汇总了innodb buffer信息和事务等待innodb锁的信息。

**3. I/o相关：**以io开头，汇总了等待I/O、I/O使用量情况。

**4. 内存使用情况：**以memory开头，从主机、线程、事件等角度展示内存的使用情况

**5. 连接与会话信息：**processlist和session相关视图，总结了会话相关信息。

**6. 表相关：**以schema_table开头的视图，展示了表的统计信息。

**7. 索引信息：**统计了索引的使用情况，包含冗余索引和未使用的索引情况。

**8. 语句相关：**以statement开头，包含执行全表扫描、使用临时表、排序等的语句信息。

**9. 用户相关：**以user开头的视图，统计了用户使用的文件I/O、执行语句统计信息。

**10. 等待事件相关信息：**以wait开头，展示等待事件的延迟情况。

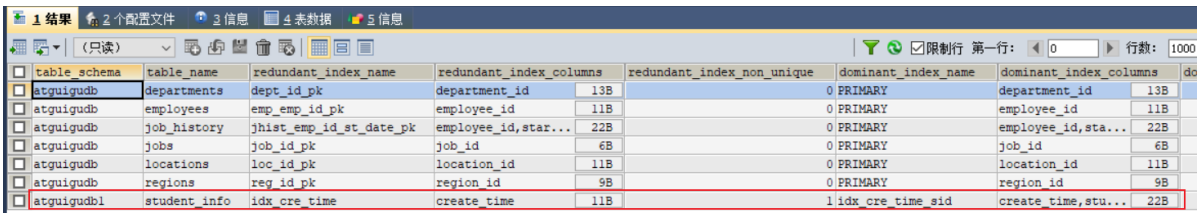
## 9.2 Sys schema视图使用场景

**索引情况**

```
#1. 查询冗余索引
select * from sys.schema_redundant_indexes;
#2. 查询未使用过的索引
select * from sys.schema_unused_indexes;
#3. 查询索引的使用情况
select index_name,rows_selected,rows_inserted,rows_updated,rows_deleted
from sys.schema_index_statistics where table_schema='dbname' ;
```

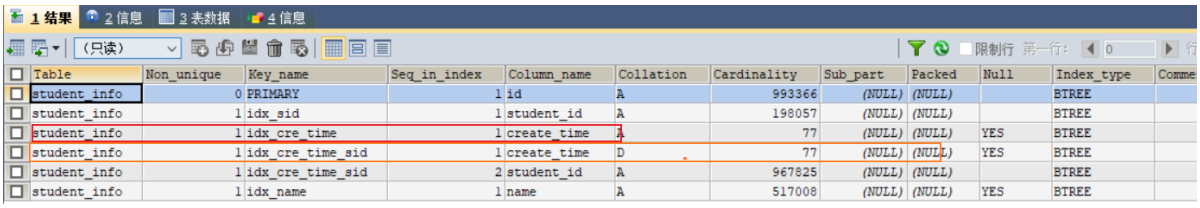
举例：比如我们查看下数据的的冗余索引

```
select * from sys.schema_redundant_indexes;
```




| table_schema | table_name | redundant_index_name | redundant_index_columns | | redundant_index_non_unique | dominant_index_name | dominant_index_columns | |
|---|---|---|---|---|---|---|---|---|
| atguigudb | departments | dept_id_pk | department_id | 13B | 0 | PRIMARY | department_id | 13B |
| atguigudb | employees | emp_emp_id_pk | employee_id | 11B | 0 | PRIMARY | employee_id | 11B |
| atguigudb | job_history | jhist_emp_id_st_date_pk | employee_id,star... | 22B | 0 | PRIMARY | employee_id,sta... | 22B |
| atguigudb | jobs | job_id_pk | job_id | 6B | 0 | PRIMARY | job_id | 6B |
| atguigudb | locations | loc_id_pk | location_id | 11B | 0 | PRIMARY | location_id | 11B |
| atguigudb | regions | reg_id_pk | region_id | 9B | 0 | PRIMARY | region_id | 9B |
| atguigudb1 | student_info | idx_cre_time | create_time | 11B | 1 | idx_cre_time_sid | create_time,stu... | 22B |

我们任意选择一条，比如最后一条，然后查看下student_info的索引情况，看看是否`idx_cre_time`冗余了



| Table | Non_unique | Key_name | Seq_in_index | Column_name | Collation | Cardinality | Sub_part | Packed | Null | Index_type | Comme |
|---|---|---|---|---|---|---|---|---|---|---|---|
| student_info | 0 | PRIMARY | 1 | id | A | 993366 | (NULL) | (NULL) | | BTREE | |
| student_info | 1 | idx_sid | 1 | student_id | A | 198057 | (NULL) | (NULL) | | BTREE | |
| student_info | 1 | idx_cre_time | 1 | create_time | A | 77 | (NULL) | (NULL) | YES | BTREE | |
| student_info | 1 | idx_cre_time_sid | 1 | create_time | D | 77 | (NULL) | (NULL) | YES | BTREE | |
| student_info | 1 | idx_cre_time_sid | 2 | student_id | A | 967825 | (NULL) | (NULL) | | BTREE | |
| student_info | 1 | idx_name | 1 | name | A | 517008 | (NULL) | (NULL) | YES | BTREE | |

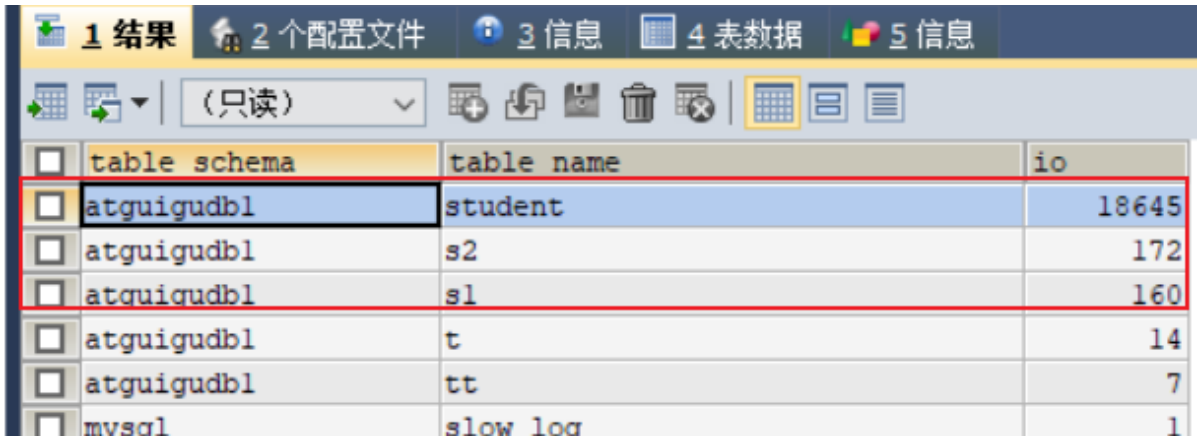
可以 看到 `idx_cre_time` 和 `idx_cre_time_sid` 两个索引中都有 `create_time`。而且联合索引性能要高于单列索引，所以 `idx_cre_time` 完全可以删掉~

**表相关**

```
# 1. 查询表的访问量
select table_schema,table_name,sum(io_read_requests+io_write_requests) as io
from
sys.schema_table_statistics group by table_schema,table_name order by io
desc;
# 2. 查询占用bufferpool较多的表
select object_schema,object_name,allocated,data
from sys.innodb_buffer_stats_by_table order by allocated limit 10;
# 3. 查看表的全表扫描情况
select * from sys.statements_with_full_table_scans where db='dbname';
```

例如：查询表的访问量

```
# 1. 查询表的访问量
select table_schema,table_name,sum(io_read_requests+io_write_requests) as io
from sys.schema_table_statistics group by table_schema,table_name order by io
desc;
```



1 结果 | 2 个配置文件 | 3 信息 | 4 表数据 | 5 信息

（只读）

| table_schema | table_name | io |
|---|---|---|
| atguigudb1 | student | 18645 |
| atguigudb1 | s2 | 172 |
| atguigudb1 | s1 | 160 |
| atguigudb1 | t | 14 |
| atguigudb1 | tt | 7 |
| mysql | slow_log | 1 |

**语句相关**

```
#1. 监控SQL执行的频率
select db,exec_count,query from sys.statement_analysis
order by exec_count desc;
#2. 监控使用了排序的SQL
select db,exec_count,first_seen,last_seen,query
from sys.statements_with_sorting limit 1;
#3. 监控使用了临时表或者磁盘临时表的SQL
select db,exec_count,tmp_tables,tmp_disk_tables,query
from sys.statement_analysis where tmp_tables>0 or tmp_disk_tables >0
order by (tmp_tables+tmp_disk_tables) desc;
```

**IO相关**

```
#1. 查看消耗磁盘IO的文件
select file,avg_read,avg_write,avg_read+avg_write as avg_io
from sys.io_global_by_file_by_bytes order by avg_read limit 10;
```

**Innodb 相关**

```
#1. 行锁阻塞情况
select * from sys.innodb_lock_waits;
```

> **风险提示：**
>
> 通过sys库去查询时，MySQL会`消耗大量资源`去收集相关信息，严重的可能会导致业务请求被阻塞，从而引起故障。建议生产上`不要频繁`的去查询sys或者`performance_ schema`、`information_ schema`来完成监控、巡检等工作。

# 10. 小结

查询时数据库中最频繁的操作，提高查询速度可以有效地提高MySQL数据库的性能。通过对查询语句的分析可以了解查询语句的执行情况，找出查询语句执行的瓶颈，从而优化查询语句！