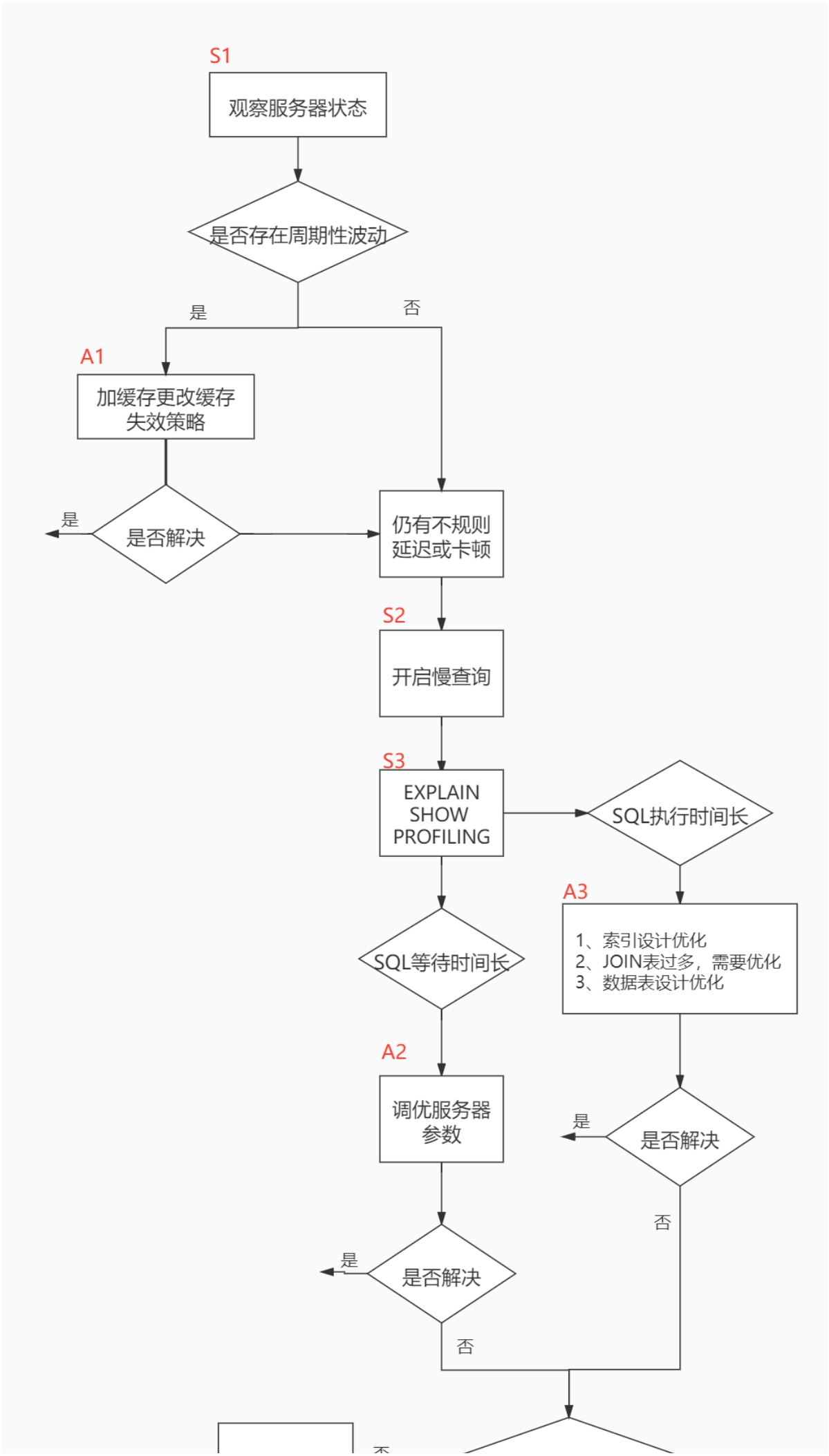
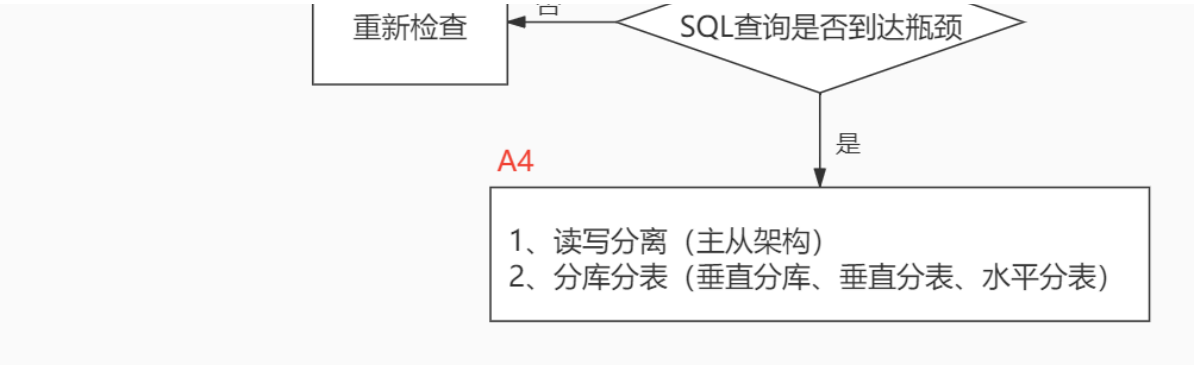
Part 9 性能分析工具的使用

1. 数据库服务器的优化步骤

数据库调优问题的思考流程如下图，包括观察（Show status）和行动（Action）两个部分。S代表观察（会使用相应的分析工具），A代表行动（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
是否解决
是
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
是否解决
是
否
是否解决
是
否
否



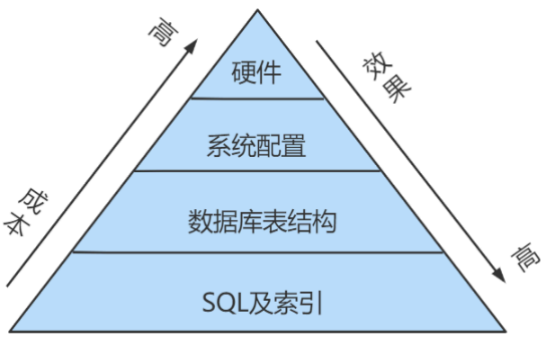
我们可以通过观察了解数据库整体的运行状态，通过性能分析工具可以让我们了解执行慢的SQL都有哪些，查看具体的SQL执行计划，甚至是SQL执行中的每一步的成本代价，这样才能定位问题所在，找到了问题，再采取相应的行动。

详细解释一下这张图：

- **S1**：需要观察服务器的状态是否存在周期性的波动，比如双十一、促销活动等。我们可以通过A1这一步骤解决。
  - **A1**：加缓存，或者更改缓存失效策略。
- 若没有解决则需要进一步分析查询延迟和卡顿的原因。
- **S2**：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。
- **S3**：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`SHOWPROFILING`。

小结：



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。
    - 默认十秒以上
- 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。

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`。再比如，如下的指令可以查看相关的指令情况：

```
show status like 'Innodb_rows_%';
```

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`。

> 在缓冲池中没有查询的相关页时，查询一条记录仅涉及到一页都会花很长时间。当缓冲池中有相关页时，涉及到多个页的时间也可能很短。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：它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，从页加载的角度来看，我们可以得到以下两点结论:

> 1. 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。
> 2. 批量决定效率。如果我们从磁盘中对单一页进行随机读，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。
>
> 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。

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sql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

4.1.1 开启slow_query_log

```
mysql > show variables like '%slow_query_log%';
mysql > set global slow_query_log='ON';
```

```
mysql> show variables like '%slow_query_log%';
+---------------------+-------------------------------------+
| Variable_name       | Value                               |
+---------------------+-------------------------------------+
| slow_query_log      | ON                                  |
| slow_query_log_file | /var/lib/mysql/atguigu02-slow.log |
+---------------------+-------------------------------------+
2 rows in set (0.00 sec)
```

4.1.2 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 (0.01 sec)
```

```
# 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。
# 不加global只是当前窗口有效。
mysql> set global long_query_time = 1 ;
mysql> show global variables like ' %long-query_time% ';

# 即更改global 也更改了session变量
mysql> set long_query_time = 1;
mysql> show variables like '%long_query_time%';
```

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/my-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%';
```

> **补充说明:**
>
> 除了上述变量，控制慢查询日志的还有一个系统变量： 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是慢查询的条件。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。
>
> ```
> mysql> show variables like 'min%';
> ```

> 你也可以根据需要，通过修改“my.ini"文件，来修改"min_examined_row_limit”的值。

4.3 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`。

查看mysqldumpslow的帮助信息

```
mysqldumpslow --help
```

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**

举例：

```
# 我们想要按照查询时间排序，查看前五条 SQL 语句
mysqldumpslow -s t -t 5 /var/lib/mysql/atguigu01-slow.log
```

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
```

4.4 关闭慢查询日志

除了调优需要开，正常还是不要开了

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

配置文件中

```
[mysqld]
# slow_query_log一项置为OFF或注释或删除
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
```

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。

```
# 停止MySQL慢查询日志功能
SET GLOBAL slow_query_log=off;

# 重启MySQL服务
SET GLOBAL slow_query_log=on;

#使用SHOW语句查询慢查询日志功能信息
SHOW VARIABLES LIKE '%slow%';
#以及
SHOW VARIABLES LIKE '%long_query_time%';
```

4.5 删除慢查询日志

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生成慢查询日志文件。

```
# 不使用这个命令，没办法自己创建
mysqladmin -uroot -p flush-logs slow

## 这个命令可以重置其他日志 例如undo日志
```

> 提示
>
> 慢查询日志都是使用mysqladmin flush-logs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

```
# 查看功能是否开启
mysql> show variables like 'profiling';

# 通过设置 profiling='ON' 来开启 show profile
mysql> set profiling = 'ON';

# 执行相关的查询语句。接着看下当前会话都有哪些 profiles
mysql> show profiles;

# 查看最近一次查询的开销
mysql> show profile;

# 查看指定的Query ID的开销
mysql> show profile cpu,block io for query 2;
```

> 如果是executing比较长就可能是代码哪里没写好，使用explain 继续查询问题

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。

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**

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 `explain 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。

6.2 基本语法

EXPLAIN 或 DESCRIBE语句的语法形式如下：

```
EXPLAIN SELECT select_options
# 或者 两个是一样的
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;
```

6.3 EXPLAIN各列作用

`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**（重要）
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

6.3.1 table表名

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） 。用到多少个表，就会有多少条记录。

```
#查询的每一行记录都对应着一个单表
explain select count(*) from s1;

#s1:驱动表  s2:被驱动表
EXPLAIN SELECT * FROM s1 INNER JOIN s2;
# 驱动表和被驱动表是 优化器决定的，他认为哪个比较好久用哪个
```

6.3.2 id

正常来说一个select 一个id，也有例外的可能，查询优化器做了优化。

举例

```
mysql> 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)
```

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

```
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)
```

优化例子

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

```
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)
```

优化后

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

```
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)
```

Union去重

原本想的1个select 一个 id , 预计两个。

```
# union 去重
EXPLAIN SELECT * FROM s1 UNION SELECT * FROM s2;
```

```
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)
```

```
# union all 不去重  所以不需要放在临时表里面
mysql> EXPLAIN SELECT * FROM s1 UNION ALL SELECT * FROM s2;
```

```
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)
```

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

6.3.3 select_type

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)

- `SIMPLE`

```
# 查询语句中不包含UNION或者子查询的查询都算作是SIMPLE类型
EXPLAIN SELECT * FROM s1;

# 连接查询也算是SIMPLE类型
EXPLAIN SELECT * FROM s1 INNER JOIN s2;
```

- `PRIMARY`与`UNION`与`UNION RESULT`
  - `UNION RESULT`

    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`，例子上边有。

- `UNION`或者`UNION ALL`或者子查询的大查询
  - 最左边的那个查询的`select_type`值就是`PRIMARY`
  - 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`
  - `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`

- `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，并且该子查询是不相关子查询，并且查询优化器决定采用将该子查询物化的方案来执行该子查询时，该子查询的第个`SELECT`关键字代表的那个查询的`select_type`就是`SUBQUERY`，比如下边这个查询

  > semi-join：一张表在另一张表找到匹配的记录之后，半连接（semi-jion）返回第一张表中的记录。

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

```
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)
```

- `DEPENDENT 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`。注意的是，select_type为`DEPENDENT SUBQUERY`的查询可能会被执行多次。

```
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 (0.00 sec)
```

- `DEPENDENT UNION`

  在包含`UNION`或者`UNION ALL`的大查询中，如果各个小查询都依赖于外层查询的话，那除了最左边的那个小查询之外，其余的小查询的`select_type`的值就是`DEPENDENT UNION`。

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

```
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 (0.04 sec)
```

- `DERIVED`

  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`。

  > 派生表是在外部查询的FROM子句中定义的，只要外部查询一结束，派生表也就不存在了。和AS有关。

```
EXPLAIN SELECT *  FROM (SELECT key1, COUNT(*) AS c FROM s1 GROUP BY key1)
AS derived_s1 WHERE c > 1;
```

```
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)
```

- `MATERIALIZED`

  当查询优化器在执行包含子查询的语句时，选择将子查询物化之后与外层查询进行连接查询时，该子查询对应的`select_type`属性就是`MATERIALIZED`。

```
EXPLAIN SELECT * FROM s1 WHERE key1 IN (SELECT key1 FROM s2); #子查询被转为了物化表
```

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 IN (SELECT key1 FROM s2); #子查询被转为了物化表
+----+--------------+--------------+------------+--------+---------------------+---------------------+
| id | select_type  | table        | partitions | type   | possible_keys       | key                 |
+----+--------------+--------------+------------+--------+---------------------+---------------------+
|  1 | SIMPLE       | s1           | NULL       | ALL    | idx_key1            | NULL                |
|  1 | SIMPLE       | <subquery2>  | NULL       | eq_ref | <auto_distinct_key> | <auto_distinct_key> |
|  2 | MATERIALIZED | s2           | NULL       | index  | idx_key1            | idx_key1            |
+----+--------------+--------------+------------+--------+---------------------+---------------------+
3 rows in set, 1 warning (0.00 sec)
```

- `UNCACHEABLE SUBQUERY`
- `UNCACHEABLE UNION`

6.3.4 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。

6.3.5 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。

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`。

- `system`

  当表中`只有一条记录`并且该表使用的存储引擎的统计数据是精确的，比如MyISAM、Memory，那么对该表的访问方法就是`system`。这里如果是 innodb 会变成ALL， 因为innodb系统不会存条数字段。。MyISAM会存储这么一个字段

- `const`

  当我们根据主键或者唯一二级索引列与常数进行等值匹配时，对**单表**的访问方法就是`const`。

- `eq_ref`

  在连接查询时，如果**被驱动表**是通过主键或者唯一二级索引列等值匹配的方式进行访问的（如果该主键或者唯一二级索引是联合索引的话，所有的索引列都必须进行等值比较），则对该被驱动表的访问方法就是`eq_ref`。

```
mysql>  EXPLAIN SELECT * FROM s1 INNER JOIN s2 ON s1.id = s2.id;
+----+------------+------+--------+---------+---------+------------------+------+
| id | select_type| table| type   | key     | key_len | ref              | rows |
+----+------------+------+--------+---------+---------+------------------+------+
|  1 | SIMPLE     | s1   | ALL    | NULL    | NULL    | NULL             | 9895 |
|  1 | SIMPLE     | s2   | eq_ref | PRIMARY | 4       | atguigudb1.s1.id |    1 |
+----+------------+------+--------+---------+---------+------------------+------+
2 rows in set, 1 warning (0.00 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`

  当通过普通的二级索引列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` 。

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

> tips: 类型相同才可以走索引
>
> ```
> EXPLAIN SELECT * FROM s1 WHERE key2 = 10066;
> # 这个是不会走索引的 因为key2 是字符串
> # 类型不一样，mysql会加函数，进行隐式转换，一旦加上函数，就不会走索引了。
> ```

- `fulltext`

  全文索引

- `ref_or_null`

  当对普通二级索引进行等值匹配查询，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` 。

- `index_merge`

  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。

```
mysql>  EXPLAIN SELECT * FROM s1 WHERE key1 = 'a' OR key3 = 'a';
+---+-------+------------+-----------------+--------+-----------------------------+
| id| table | type       | key             | key_len| Extra                       |
+---+-------+------------+-----------------+--------+-----------------------------+
|  1| s1    | index_merge| idx_key1,idx_key3 | 303,303| Using union(idx_key1,idx_key3)|
+---+-------+------------+-----------------+--------+-----------------------------+
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。

- `unique_subquery`

  `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` 。

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

- `index_subquery`

  和unique_subquery类似，只是子查询使用的是非唯一索引。

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

- `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');
#同上
EXPLAIN SELECT * FROM s1 WHERE key1 > 'a' AND key1 < 'b';
```

- `index`

  当我们可以使用索引覆盖，但需要扫描全部的索引记录时，该表的访问方法就是`index`。

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

> 索引覆盖，`INDEX idx_key_part(key_part1, key_part2, key_part3)` 这3个构成一个复合索引
>
> key_part3 在复合索引里面，，查询的字段也在索引里面，干脆就直接遍历索引查出数据
>
> 思考： 好处，索引存的数据少，数据少页就少，这样可以减少io。

- `ALL`

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

  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

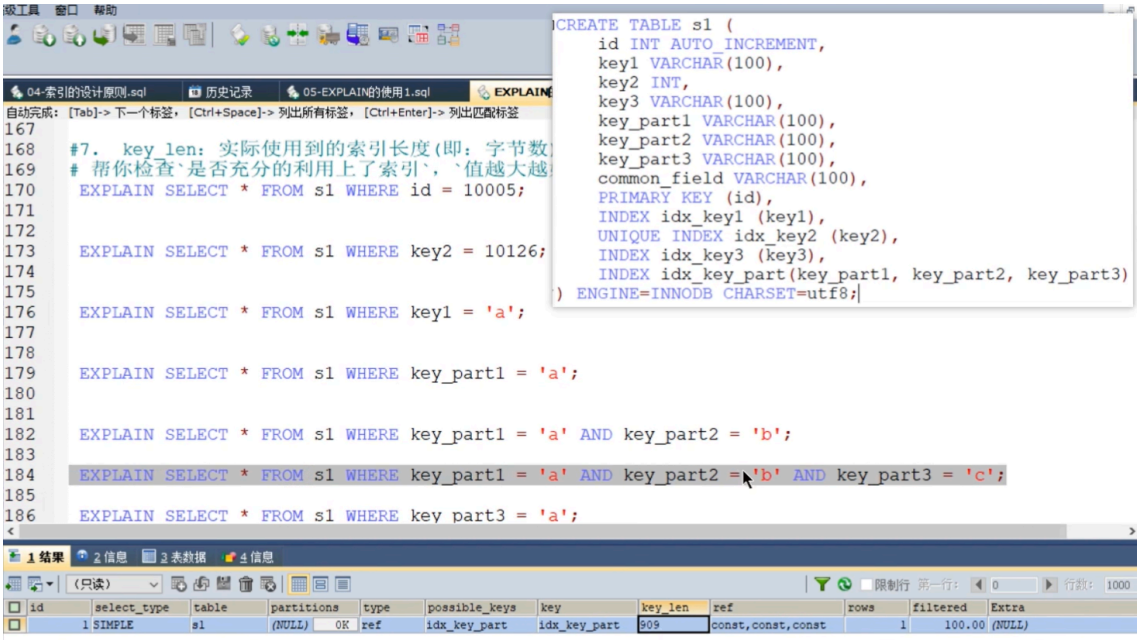
6.3.6 possible_keys和key

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，经过查询优化器计算使用不同索引的成本后，决定实际用的为key。`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。

> 索引只能用一个。所以他要选一个出来用。查看上面`index_merge` or 的话 会走索引合并。

6.3.7 key_len ☆

- key_len：实际使用到的索引长度(即：字节数)
- key_len越小 索引效果越好 这是前面学到的只是，短一点效率更高
- **但是在联合索引里面，帮助检查是否充分利用了索引，命中一次key_len加一次长度。越长代表精度越高，效果越好**




```
EXPLAIN SELECT * FROM s1 WHERE id = 10005;## 结果key_len =4
EXPLAIN SELECT * FROM s1 WHERE key2 = 10126;## 结果key_len = 5，key2 是int类型 unique 索引。。因为还可能有一个null值，所以 null占一个字段。4+1 = 5
EXPLAIN SELECT * FROM s1 WHERE key1 = 'a';## 结果key_len = 303。`idx_key_part(key_part1, key_part2, key_part3)` 是3个100的字段合起来的。每一个字段可以为空，所以是101*3 = 303
EXPLAIN SELECT * FROM s1 WHERE key_part1 = 'a' AND key_part2 = 'b';## 结果key_len = 606，这里命中了两次联合索引，精度更高，效果更好
```

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)
```

6.3.8 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。比如只是一个常数或者是某个列。

```
mysql> EXPLAIN SELECT * FROM s1 WHERE key1 = 'a';
+----+-------------+------+------+---------------+----------+---------+-------+
| id | select_type | table| type | possible_keys | key      | key_len | ref   |
+----+-------------+------+------+---------------+----------+---------+-------+
|  1 | SIMPLE      | s1   | ref  | idx_key1      | idx_key1 | 303     | const |
+----+-------------+------+------+---------------+----------+---------+-------+
```

类型是type = ref，与const（常量）比较

```
mysql> EXPLAIN SELECT * FROM s1 INNER JOIN s2 ON s1.id = s2.id;
+---+------+--------+---------------+--------+------------------+-----
| id| table| type   | possible_keys | key    | ref              | rows
+---+------+--------+---------------+--------+------------------+-----
|  1| s1   | ALL    | PRIMARY       | NULL   | NULL             | 9895
|  1| s2   | eq_ref | PRIMARY       | PRIMARY| atguigudb1.s1.id |    1
+---+------+--------+---------------+--------+------------------+-----
```

类型是type = eq_ref， 与 atguigudb1.s1.id 比较

6.3.9 rows ☆

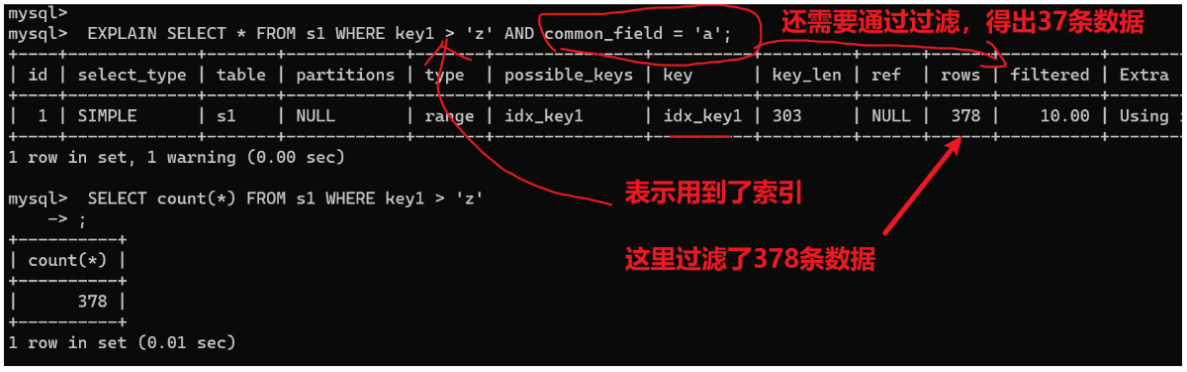
预估的需要读取的记录条数，`值越小越好`，通常与filtered 一起使用。rows 值越小，代表数据越有可能在一个页里面，这样io就会更小。

6.3.10 filtered

filtered 的值指返回结果的行占需要读到的行(rows 列的值)的百分比。

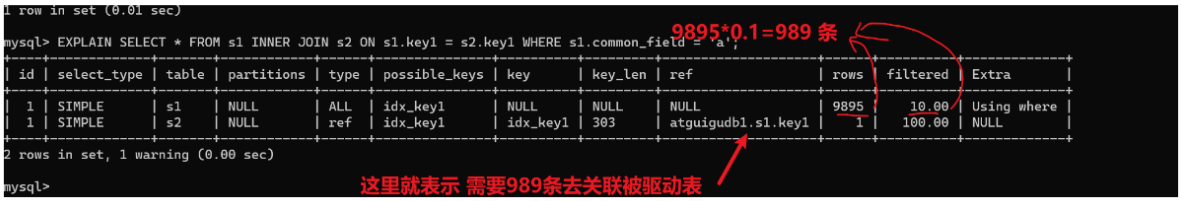
如果使用的是索引执行的单表扫描，那么计算时需要估计出满足除使用到对应索引的搜索条件外的其他搜索条件的记录有多少条。

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



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)

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



6.3.11 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，以下捡重点介绍

- `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;
  ```

  ```
  > EXPLAIN SELECT 1;
  +-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------+
  | select_type | table | partitions | type | possible_keys | key  | key_len | ref  | rows | filtered | Extra          |
  +-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------+
  | SIMPLE      | NULL  | NULL       | NULL | NULL          | NULL | NULL    | NULL | NULL |     NULL | No tables used |
  +-------------+-------+------------+------+---------------+------+---------+------+------+----------+----------------+
  ```

- `Impossible WHERE`

  查询语句的`WHERE`子句永远为`FALSE`时将会提示该额外信息。

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

  ```
  mysql>  #查询语句的`WHERE`子句永远为`FALSE`时将会提示该额外信息
  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`

  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。

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

  ```
  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 | 9895 |    10.00 | Using where |
  +----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------------+
  1 row in set, 1 warning (0.00 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。

```
mysql> SELECT * FROM s1 WHERE key1 = 'fUhcQU' and  common_field = 'uDHCOnalcF';
+--------+--------+--------+--------+-----------+-----------+-----------+--------------+
| id     | key1   | key2   | key3   | key_part1 | key_part2 | key_part3 | common_field |
+--------+--------+--------+--------+-----------+-----------+-----------+--------------+
| 10012 | fUhcQU | 10336 | eKldJX | RspuelQyWl | pPhqg     | KiKYOaKweg | uDHCOnalcF   |
+--------+--------+--------+--------+-----------+-----------+-----------+--------------+
1 row in set (0.00 sec)

mysql> explain SELECT * FROM s1 WHERE key1 = 'fUhcQU' and  common_field = 'uDHCOnalcF';
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------------+
| 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 |    10.00 | Using where |
+----+-------------+-------+------------+------+---------------+----------+---------+-------+------+----------+-------------+
1 row in set, 1 warning (0.00 sec)
```

- `No matching min/max row`

  当查询列表处有`MIN`或者`MAX`聚合函数，但是并没有符合`WHERE`子句中的搜索条件的记录时，将会提示该额外信息。

```
1  # 数据库不存在 QLjKYOx
2  EXPLAIN SELECT MIN(key1) FROM s1 WHERE key1 = 'QLjKYOx';
```

```
mysql>  EXPLAIN SELECT MIN(key1) FROM s1 WHERE key1 = 'QLjKYOx';
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-------------------------+
| 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)
```

数据库没有QLjKYOx 时

```
1  # 数据库存在 QLjKYO
2  EXPLAIN SELECT MIN(key1) FROM s1 WHERE key1 = 'QLjKYO';
```

```
mysql>  EXPLAIN SELECT MIN(key1) FROM s1 WHERE key1 = 'QLjKYO';
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-----------------------------+
| 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 (0.01 sec)
```

存在

- `Using index`

  当我们的查询列表以及搜索条件中只包含属于某个索引的列，也就是在可以使用覆盖索引的情况下，在`Extra`列将会提示该额外信息。

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

```
mysql>  EXPLAIN SELECT key1,id 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 | 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。

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

```
mysql>  #有些搜索条件中虽然出现了索引列，但却不能使用到索引
mysql>  #看课件理解索引条件下推
mysql>  EXPLAIN SELECT * FROM s1 WHERE key1 > 'z' AND key1 LIKE '%a';
+----+-------------+-------+------------+-------+---------------+----------+---------+------+------+----------+-----------------------+
| 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 |  378 |   100.00 | Using index condition |
+----+-------------+-------+------------+-------+---------------+----------+---------+------+------+----------+-----------------------+
1 row in set, 1 warning (0.00 sec)
```

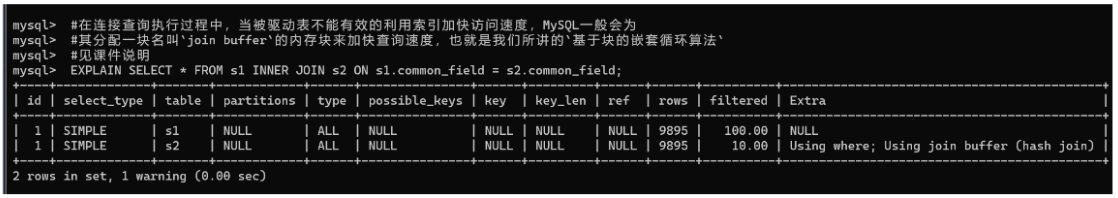
  - 以前版本的Mysql中先根据`key1 > 'z'`这个条件，从二级索引`idx_key1`中获取到对应的二级索引记录。再从得到的二级索引记录回表，检测该记录是否符合`key1 LIKE '%a'`这个条件，将符合条件的记录加入到最后的结果集。
  - Mysql的改进：根据`key1 > 'z'`这个条件，定位符合记录，不回表，直接判断是否满足`key1 LIKE '%a'`这个条件，以满足这两个条件的记录回表。

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`

- `Using join buffer (Block Nested Loop)`

  没有索引的字段进行表关联。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`

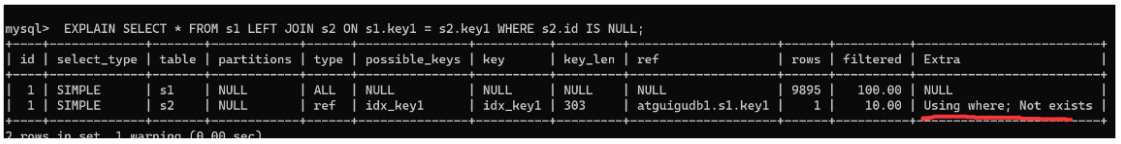
```
mysql> EXPLAIN SELECT * FROM s1 INNER JOIN s2 ON s1.common_field = s2.common_field;
```



- `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。

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



- `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；
  - 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。

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

- `Zero limit`
- `Using filesort`

  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，比如下边这个查询:

```
EXPLAIN SELECT * FROM s1 ORDER BY key1 LIMIT 10;
```

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。

```
EXPLAIN SELECT * FROM s1 ORDER BY common_field LIMIT 10;
```

- `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。

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

```
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 | 9895 |   100.00 | Using temporary |
+----+-------------+-------+------------+------+---------------+------+---------+------+------+----------+-----------------+
1 row in set, 1 warning (0.00 sec)

mysql>
```

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 | 9895 |   100.00 | Using index |
+----+-------------+-------+------------+-------+---------------+----------+---------+------+------+----------+-------------+
1 row in set, 1 warning (0.00 sec)
```

6.3.12 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

`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。

7.1.1 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。

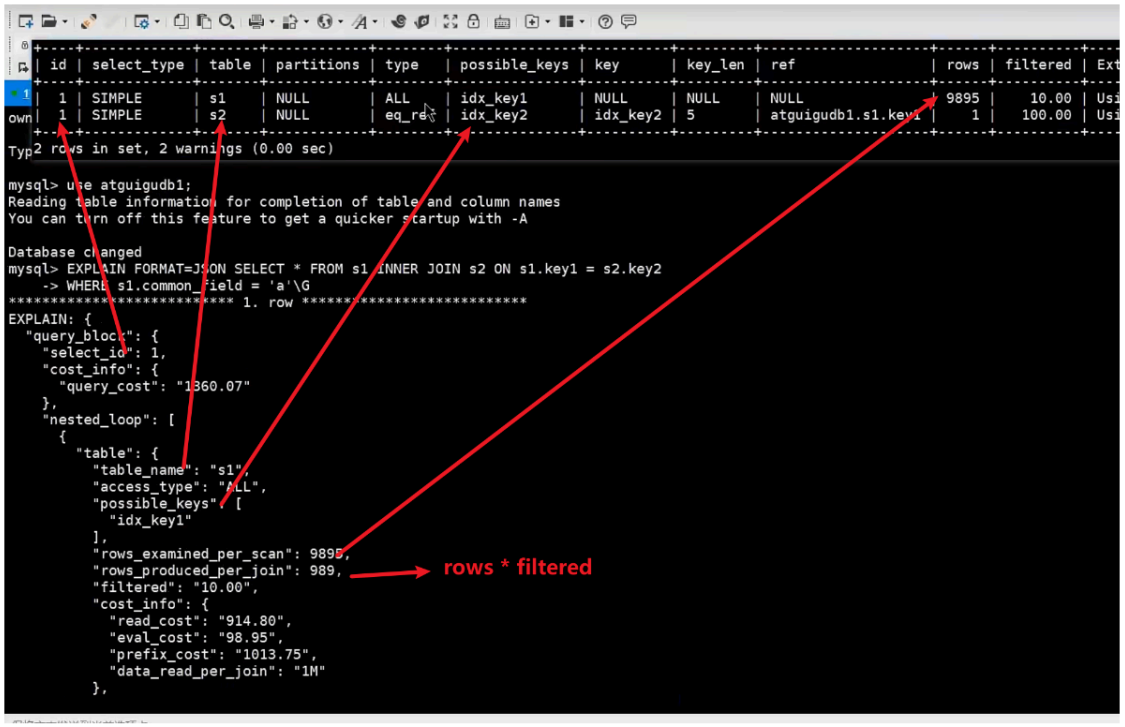
```
mysql> EXPLAIN SELECT s1.key1, s2.key1 FROM s1 LEFT JOIN s2 ON s1.key1 = s2.key1 WHERE
s2.common_field IS NOT NULL;
+----+-------------+-------+------------+------+---------------+-------
| id | select_type | table | partitions | type | possible_keys | key
+----+-------------+-------+------------+------+---------------+-------
|  1 | SIMPLE      | s2    | NULL       | ALL  | idx_key1      | NULL
|  1 | SIMPLE      | s1    | NULL       | ref  | idx_key1      | idx_ke
+----+-------------+-------+------------+------+---------------+-------
2 rows in set, 1 warning (0.00 sec)
```

7.1.2 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的Column与JSON的对应关系:(来源于MySQL 5.7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
possible_keys	possible_keys	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



- `cost_info`

  s1的cost_info部分

```
"cost_info": {
    "read_cost": "914.80",
    "eval_cost": "98.95",
    "prefix_cost": "1013.75",
    "data_read_per_join": "1M"
}
```

  s2的cost_info部分

```
"cost_info": {
    "read_cost": "247.38",
    "eval_cost": "98.95",
    "prefix_cost": "1360.08",
    "data_read_per_join": "1M"
}
```

  - `read_cost` 是由IO 成本和检测 `rows × (1 - filter)` 条记录的 CPU 成本决定的。

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

7.1.3 TREE格式

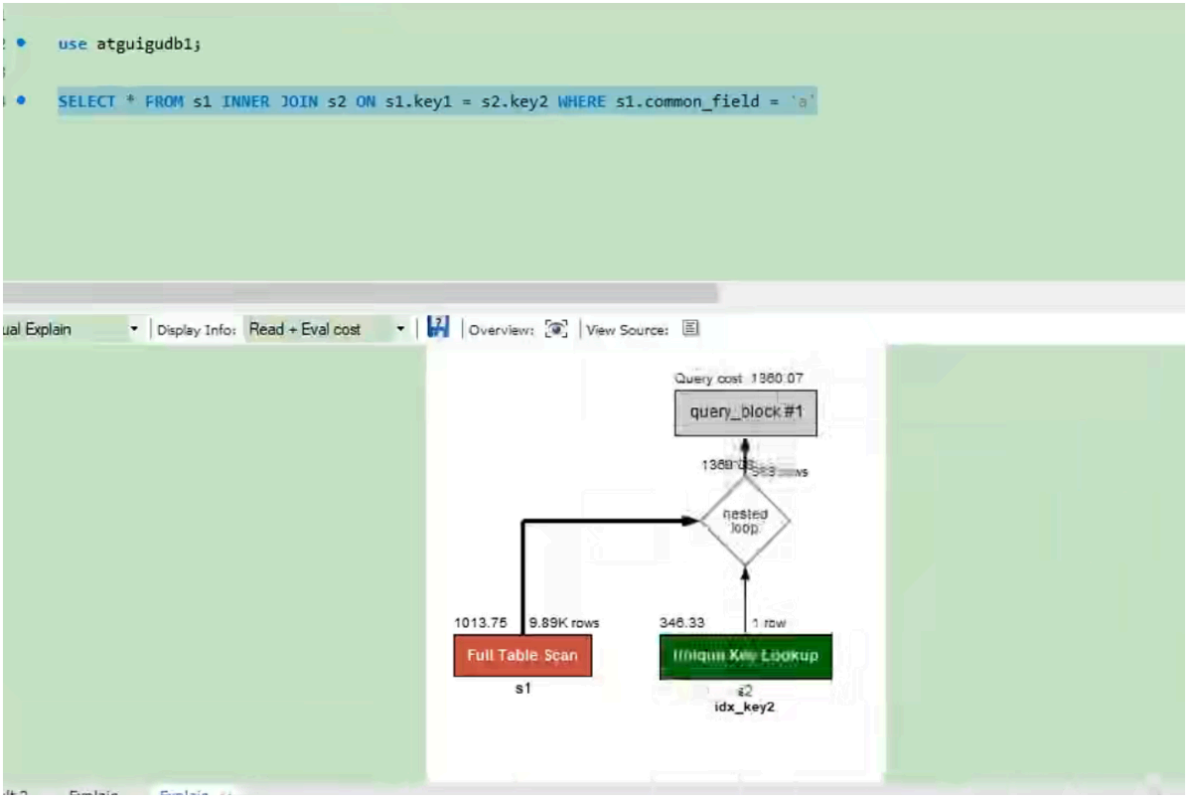
TREE格式是8.0.16版本之后引入的新格式，主要根据查询的 `各个部分之间的关系` 和 `各部分的执行顺序` 来描述如何查询

```
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)
```

7.1.4. 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。

7.2 SHOW WARNINGS的使用

使用完explain 后紧接着使用 `SHOW WARNINGS \G`

```
mysql> SHOW WARNINGS\G
*************************** 1. row ***************************
  Level: Note
   Code: 1003
Message: /* select#1 */ select `atguigudb1`.`s1`.`key1` AS
`key1`,`atguigudb1`.`s2`.`key1` AS `key1` from `atguigudb1`.`s1` join
`atguigudb1`.`s2` where ((`atguigudb1`.`s1`.`key1` =
`atguigudb1`.`s2`.`key1`) and (`atguigudb1`.`s2`.`common_field` is not null))
1 row in set (0.00 sec)
```

> 可以看到查询优化器真正执行的语句
>
> 粘出来并不一定可以运行

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LEFT JOIN已经变成了JOIN(内连接)。

8. 分析优化器执行计划：trace

`OPTIMIZER_TRACE` 是MySQL 5.6引入的一项跟踪功能，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、各种转换等)，并将跟踪结果记录到 `INFORMATION_SCHEMA.OPTIMIZER_TRACE` 表中。

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。

```
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
```

```
mysql> 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": 3945207,
                      "cost": 404306
                    } /* table_scan */,/*表扫描*/
                    "potential_range_indexes": [
                      {
                        "index": "PRIMARY",
                        "usable": true,
                        "key_parts": [
                          "id"
                        ] /* key_parts */
                      }
                    ] /* potential_range_indexes */,
                    "setup_range_conditions": [
                    ] /* 设置条件范围 */,
                    "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,
                        "in_memory": 0.159895,
                        "rows": 9,
                        "cost": 1.79883,
                        "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": 1.79883,
                    "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": 2.69883,
                      "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": 2.69883,
                "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 (0.01 sec)
```

9. MySQL监控分析视图-sys schema

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' ;
```

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';
```

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。