一、Explain 工具介绍

使用Explain可以查看sql的性能瓶颈信息，并根据结果进行sql的相关优化。在select 语句前加上explain关键字，执行的时候并不会真正执行sql语句，而是返回sql查询语句对应的执行计划信息。

当然如果select语句的from后面有一个子查询的话，就会执行子查询了并把结果放到一个临时表中。

有三张表：

```
-- 演员表
CREATE TABLE actor (
id INT ( 11 ) NOT NULL,
name VARCHAR ( 45 ) DEFAULT NULL,
update_time datetime DEFAULT NULL,
PRIMARY KEY ( id )
) ENGINE = INNODB DEFAULT CHARSET = utf8;
INSERT INTO actor (id, name, update_time) VALUES (1,'a','2017-12-22 15:27:18'), (2,'b','20 17-12-22
15:27:18'), (3,'c','2017-12-22 15:27:18');
-- 电影表
CREATE TABLE film (
id INT ( 11 ) NOT NULL AUTO_INCREMENT,
name VARCHAR ( 10 ) DEFAULT NULL,
PRIMARY KEY ( id ),
KEY idx_name ( name )
) ENGINE = INNODB DEFAULT CHARSET = utf8;
INSERT INTO film (id, name) VALUES (3,'film0'),(1,'film1'),(2,'film2');
-- 演员和电影中间表
CREATE TABLE film_actor (
id INT ( 11 ) NOT NULL,
film_id INT ( 11 ) NOT NULL,
actor_id INT ( 11 ) NOT NULL,
remark VARCHAR ( 255 ) DEFAULT NULL,
PRIMARY KEY ( id ),
KEY idx_film_actor_id ( film_id, actor_id )
) ENGINE = INNODB DEFAULT CHARSET = utf8;
INSERT INTO film_actor (id, film_id, actor_id) VALUES (1,1,1),(2,1,2),(3,2,1);
```

执行explain select * from actor;

结果：

根据返回的信息可以分析sql的性能瓶颈从而进行优化。

下面分析其中每个字段对应的含义。

1.id

代表sql中查询语句的序列号，序列号越大则执行的优先级越高，序号一样谁在前谁先执行。id为null则最后执行。

2.select_type

查询类型，表示当前被分析的sql语句的查询的复杂度。这个字段有多个值。

SIMPLE：表示简单查询。

explain select * from actor;

PRIMARY：表示复杂查询中的最外层的select查询语句。

SUBQUERY：表是子查询语句 跟在select 关键字后面的select查询语句；

explain select (select 1 from film where id =1) from actor;

derived： 派生查询，跟在一个select查询语句的from关键字后面的select查询语句 例如：

set session optimizer_switch='derived_merge=off'; -- 关闭mysql5.7新特性对衍生表的合并优化

explain select (select 1 from actor where id =1) from (SELECT * from film where id=1) ac;

3.table

表示当前访问的表的名称。

当from中有子查询时，table字段显示的是<derivedN> N为derived的id的值。

4.partitions

返回的是数据分区的信息，不常用 这里不做分析。

5.type

这个字段决定mysql如何查找表中的数据，查找数据记录的大概范围。这个字段的所有值表示的从最优到最差依次为：

system > const > eq_ref > ref > range > index > all;

一般来说我们优化到range就可以了 最好到ref。

null：type字段的值如果为null，那么表示当前的查询语句不需要访问表，只需要从索引树中就可以获取我们需要的数据；

一般如果是主键索引的话 ，查询主键字段或者唯一索引的话 查询主键字段 type字段的值就为null。

explain select id from actor where id =1;

system/const:用户主键索引或者唯一索引查询时，只能匹配1条数据 一般可以对sql查询语句优化成一个常量，那么type一般就是system或者const，system是const的一个特例。

explain select * from (select * from film where id = 1) tmp;

eq_ref:在进行连接查询时，例如left join 时，如果是使用主键索引或者唯一索引连接查询 ，结果返回一条数据，则type的值为一般为eq_ref。

explain SELECT * from film_actor left join film on film.id = film_actor.film_id;

分析下这个sql，首先我们需要查询的是film_actor中间表 且这个表是与film表进行主键关联的，索引film_actor表中的film_id字段在film表中只有一个唯一值，所以：

那么，反过来在看一下

explain SELECT * from film left join film_actor on film_actor.film_id = film.id;

film表和film_actor中间表关联查询，根据film电影表中的主键id和film_actor表中的film_id字段进行关联的。电影表中的主键id在film_actor中并不是唯一的。所以：

对于film 需要确定查询id 从索引树中就可以获取值 所以是index。对于film_actor就是全表扫描了。

ref: 相比较eq_ref,不使用主键索引或者唯一索引，使用的是普通索引或者唯一索引的部分前缀，索引与一个值进行比较后可能获取到多个符合条件的行，不在是唯一的行了。

简单查询，name是普通索引

explain select * from film where name = 'film1';

复杂查询，film_actor 有联合索引 idx_film_actor_id('film_id','actor_id') 这里使用了联合索引的左前缀 film_id

explain select fa.film_id from film f left join film_actor fa on fa.film_id = f.id;

range:通常使用范围查找，例如between，in，<,>,>=等使用索引进行范围检索。

explain select * from film where id >2;

index:扫描索引树就能获取到的数据，一般是扫描二级索引，并且不会从根节点扫描，一般直接扫描二级索引的叶子节点，速度比较慢。因为二级索引叶子节点不保存表中其他字段数据 只保存主键，所以二级索引还是比较小的，扫描速度相比All还是很快的。这里用到了覆盖索引，什么是覆盖索引：可以直接遍历索引树就能获取数据叫做覆盖索引。这里遍历name索引树就可以获取到主键id的值就是覆盖索引。

explain select id from film ;

ALL:这是一种效率最低的type，需要扫描主键索引树的叶子节点，获取数据是表中其他列的数据，即全表扫描。

和index有什么区别呢？

拿film电影表举例：添加一个remark影评字段，film表结构如下：

```
CREATE TABLE film (
id int(11) NOT NULL AUTO_INCREMENT,
name varchar(10) DEFAULT NULL,
remark varchar(255) DEFAULT NULL,
PRIMARY KEY (id),
KEY idx_name (name)
) ENGINE=InnoDB AUTO_INCREMENT=4 DEFAULT CHARSET=utf8;
```

表中建了两个索引：id 主键索引 idx_name(name) 二级索引。

那么：

explain select id,name from film ;

上述sql查询id，name两个字段，分析mysql索引数据结构，以及mysql优化后一般扫描二级索引，索引会扫描idx_name索引树的叶子节点，那么根据B+Tree树的结构，叶子节点保存的是name字段的索引值和 data数据（主键id）。而正好我们只需要查询id和name两个字段，我们查询的字段被索引（二级索引）给覆盖了 这就是覆盖索引，因此 type的类型就是index。

再来：

explain select remark from film ;

比较上一个sql，这个sql只查询了一个字段：remark，经过上面分析，这个字段是不在idx_name索引树的叶子节点上的，所以mysql不会在扫描idx_name索引树了，直接扫描主键索引的叶子节点，即进行全表扫描，这个时候type类型为ALL。

6.possible_keys

这个字段显示的是sql在查询时可能使用到的索引，但是不一定真的使用，只是一种可能。

如果在进行explain分析sql时，发现这一列有值，但是key列为null，因为mysql觉得可能会使用索引，但是又因为表中的数据很少，使用索引反而没有全表扫描效率高，那么mysql就不会使用索引查找，这种情况是可能发生的。

如果该列是NULL，则没有相关的索引。在这种情况下，可以通过检查 where 子句看是否可以创造一个适当的索引来提

高查询性能，然后用 explain 查看效果。

7.key

sql执行中真正用到的索引字段。

8.key_len

用到的索引字段的长度，通过这个字段可以显示具体使用到了索引字段中的哪些列（主要针对联合索引）：计算公式如下

字符串

char(n)：n字节长度

varchar(n)：如果是utf-8，则长度 3n + 2 字节，加的2字节用来存储字符串长度

数值类型

tinyint：1字节

smallint：2字节

int：4字节

bigint：8字节

时间类型

date：3字节

timestamp：4字节

datetime：8字节

如果字段允许为 NULL，需要1字节记录是否为 NULL

索引最大长度是768字节，当字符串过长时，mysql会做一个类似左前缀索引的处理，将前半部分的字符提取出来做索

引。

9.ref

表示那些列或常量被用于查找索引列上的值

10.rows

表示在查询过程中检索了多少列 但是并不一定就是返回这么多列数据。

11.Extra

展示一些额外信息。

二、索引实践

以下实践以employees表为例。一个主键索引 一个联合索引

```
CREATE TABLE employees (
id int(11) NOT NULL AUTO_INCREMENT,
name varchar(24) NOT NULL DEFAULT '' COMMENT '姓名',
age int(11) NOT NULL DEFAULT '0' COMMENT '年龄',
position varchar(20) NOT NULL DEFAULT '' COMMENT '职位',
hire_time timestamp NOT NULL DEFAULT CURRENT_TIMESTAMP COMMENT '入职时间',
PRIMARY KEY (id),
KEY idx_name_age_position (name,age,position) USING BTREE
) ENGINE=InnoDB AUTO_INCREMENT=11 DEFAULT CHARSET=utf8 COMMENT='员工记录表';
```

1.联合索引最左列原则

例1：

```
EXPLAIN SELECT * FROM employees WHERE name= 'LiLei';
```

使用联合索引中的name字段索引。

例2：

```
EXPLAIN SELECT * FROM employees WHERE name= 'LiLei' AND age = 22;
```

使用联合索引中的name和gae字段索引。

例3：

```
EXPLAIN SELECT * FROM employees WHERE name= 'LiLei' AND age = 22 AND position ='manager';
```

使用联合索引中的name age position字段索引。

例4：

```
EXPLAIN SELECT * FROM employees WHERE age = 30 AND position = 'dev';
```

仅仅使用了联合索引中的name字段，因为中间age字段断了，所以position字段索引并未用到。解释一下：

索引是一个有序的数据结构，也就是说使用索引时，需要索引保证有序，那么在联合索引中，是先按照name排序，name相同情况下，在按照age排序，age相同情况下 在按照position排序，因此如果age不确定情况下，position是无序的，所以即使你是用position查询了 也无法走索引的。这就是最左列原则并且中间不能断。

例5：

```
EXPLAIN SELECT * FROM employees WHERE name= 'LiLei' AND age > 22 AND position ='manager';
```

这个使用了联合索引中的name和age字段，没有使用position，为什么？原理其实和上面差不多。分析一波：

首先按照顺序 name->age->position,name已经确定了等于LiLei，那么age就是有序的了，所以检索age>22的就很容易了 因为age有序。但是age值其实是不确定的，age可以是23,24,25...等等，所以在age不确定

情况下 position是无序的 因此是不走position索引字段的。

2.全值匹配

EXPLAIN SELECT * FROM employees WHERE name= 'LiLei' AND age = 22 AND position ='manager';

3.不建议在索引列上做任何操作，否则索引会失效转而全表扫描

-- 查询name的最左变的两个字符为Li的行

EXPLAIN SELECT * FROM employees WHERE LEFT(name,2) = 'Li';

4.尽量使用覆盖索引 不需要再回表查询了 效率较高

5.再试用!=或<>不等于查询时，会导致索引失效。

EXPLAIN SELECT * FROM employees WHERE name != 'LiLei';

6.尽量不要使用 ‘or’，‘in’操作，在某些情况下也会导致索引失效。

第一种情况：当表中只有两条数据 数据量很少的时候

explain SELECT * from employees where name in ('LiLei','abc');

使用in查询，没有走索引，进行了全表扫描，为什么？分析一波：

首先 如果使用索引的话，mysql大概会怎么操作？ 应该先在name索引树中定位到name=LiLei这个节点（最少一次I/O），然后定位到name=abc这个节点（一次I/O），然后分别拿到主键id，在去主键索引树上扫描定位（最少又要两次I/O），总共4次I/O。

如果不使用索引，直接全表扫描，那么直接扫描主键索引树的叶子节点 只需要两次I/O即可（因为只有两条数据），所以mysql评估全表扫描效率可能会更高，就不会在走索引了。

第二种情况：当表中数据量很多，例如7条数据

同样的sql查询

explain SELECT * from employees where name in ('LiLei','abc');

结果：走了索引

为什么会出现这种情况？再来分析一波：

首先走索引的话 大概需要4次I/O 上面已经分析过了。

那么不走索引的话 需要全表扫描 最坏的情况需要扫描7次，进行7次I/O,mysql评估一下发现全表扫描的效率可能是低于走索引的，所以就走了索引。

第三种情况：数据还是7条，但是我in查询时条件有8个

explain SELECT * from employees where name in ('LiLei','abc','cde','asc','ssw','2dff','wsa','sda');

看下结果:

为啥又不走索引了呢？经过上面的两波强势分析，这里也很容知道原因，就不过多的赘述了。or查询的情况类似。

7.is null ，is not null一般情况下也无法使用索引

8.是用字符串查询 不见引号 索引也会失效

explain SELECT * from employees where name = 1324;

9.针对范围查找的不走索引的优化

首先看个例子：

-- 先给age加一个独立索引

ALTER TABLE employees ADD INDEX idx_age (age) USING BTREE ;

-- 查询age 在1到2000分为内的数据

explain SELECT * from employees where age >1 and age < 2000 ;

结果：

显然并没有走索引 为什么？再来强势分析一波：

首先，我们脑海中要有一个age的索引树：

我们要找到1-2000的数据，那么在这棵树书上怎么定位？

如果我来定位的话 我会定位一个age=2在树上的位置 在定位一个age=1999在树上的位置，然后从age=2的节点开始取右边的节点，一直取下去 直到age=1999为止，但是我们表总只有7条数据，mysql觉得这样操作还没有全表扫描快，毕竟一共才几条数据全表扫描反而更快些，所以mysql就去全表扫描了。

怎么优化呢？

explain SELECT * from employees where age >1 and age < 1000 ;

explain SELECT * from employees where age >1001 and age < 2000 ;

把一个大的范围拆成多个小的范围 可以利用索引查询。

10.like查询建议使用xxx%方式匹配，%xxx或者%xxx%索引失效

EXPLAIN SELECT * FROM employees WHERE name like '%Lei';

EXPLAIN SELECT * FROM employees WHERE name like '%Lei%';

结果：全表扫描

思考下在索引树上name的排序规则，先按照第一个字符比较然后第二个字符依次向后比较，如果是用%xxx，字符串前面的字符不确定，怎么在树上定位呢？显然没法按照顺序定位，只能一个一个遍历比较 所以不会走索引。%xxx%也一样。

在看下面这个例子

EXPLAIN SELECT * FROM employees WHERE name like 'Lei%'

结果：走了索引

其实Lei%匹配相当于范围查询，只要name的值的前三个字符为Lei符合条件，等价于查找前三个字符=Lei的字符串，这个在索引树上是有序的，当然可以使用索引定位。

总结：

1.使用explain关键字，可以分析出sql的性能瓶颈并加以优化

2.了解explain返回的各字段值代表的意义，结合索引数据结构有助于我们对sql的查询效率的分析和优化

3.列举部分可能不会进行索引检索的情况，例如 !=,<>,is null,like的某些情况，or或者in的某些情况，字符串不加引号等

4.对某些不走索引查询的情况作了一些比较详细的分析
