

前提知识

Mysq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提供他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)

当客户端向 MySQL请求一条 Query，命令解析器模块完成请求分类，区别出是 SELECT并转发给 MySQL Query Optimizer时，MySQL Query Optimizer首先会对整条 Query进行优化，处理掉一些常量表达式的预算直接换算成常量值。并对 Query中的查询条件进行简化和转换，如去掉一些无用或显而易见的条件、结构调整等。然后分析 Query中的Hint信息(如果有)，看显示Hint信息是否可以完全确定该 Query的执行计划。如果没有Hint或Hint信息还不足以完全确定执行计划，则会读取所涉及对象的统计信息，根据 Query进行写相应的计算分析，然后再得出最后的执行计划。

MySQL常见性能瓶颈

CPU：CPU在饱和的时候一般发生在数据装入内存或从磁盘上读取数据时候

IO：磁盘I/O瓶颈发生在装入数据远大于内存容量的时候

服务器硬件的性能瓶颈：top、free、iostat 和 vmstat来查看系统的性能状态

Explain-执行计划

用 EXPLAIN关键字可以模拟优化器执行SQL查询语句，从而知道 MySQL是如何处理你的sql语句的。分析你的查询语句或是表结构的性能瓶颈！

用法：explain + SQL

explain能干啥？

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

执行计划包含的信息

```
Database changed
mysql> select * from tbl_emp;
```

id	name	deptId
1	z3	1
2	z4	1
3	z5	1
4	z3	1
5	w5	2
6	w6	2
7	s7	3
8	s8	4
9	s9	51

9 rows in set (0.00 sec)

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

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	tbl_emp	NULL	ALL	NULL	NULL	NULL	NULL	9	100.00	NULL

1 row in set, 1 warning (0.00 sec)

下面解释一下这些字段是干啥的：

1、id

select查询的序列号，包含一组数字，表示查询中执行select子句或操作表的顺序。有三种情况：

1、id相同，执行顺序由上至下

```
mysql> explain select * from table_01, table_02, table_03;
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref
1	SIMPLE	table_01	NULL	index	NULL	PRIMARY	4	NULL
1	SIMPLE	table_02	NULL	index	NULL	PRIMARY	4	NULL
1	SIMPLE	table_03	NULL	index	NULL	PRIMARY	4	NULL

3 rows in set, 1 warning (0.00 sec)

2、id不同，如果是子查询，id的序号会递增，id值越大优先级越高，越先被执行

```
mysql> explain select * from table_01 where id=(select id from table_02 where id=(select id from table_03));
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	PRIMARY	table_01	NULL	const	PRIMARY	PRIMARY	4	const	1	100.00	Using inc
2	SUBQUERY	table_02	NULL	const	PRIMARY	PRIMARY	4	const	1	100.00	Using inc
3	SUBQUERY	table_03	NULL	index	NULL	PRIMARY	4	NULL	1	100.00	Using inc

3 rows in set, 1 warning (0.00 sec)

PRIMARY就是主查询，id越大越先被执行，则毫无疑问在这样的子查询中，肯定最先要查询的表就是table_03，所以这是MySQL自己约定的顺序，就和运算符优先级一样！SUBQUERY代表最外层查询。

3、id相同不同，同时存在

```
mysql> explain select t2.* from (select t3.id from t3 where t3.name='')s1, t2 where s1.id=t2.id;
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	PRIMARY	<derived2>	system	NULL	NULL	NULL	NULL	1	
1	PRIMARY	t2	const	PRIMARY	PRIMARY	4	const	1	
2	DERIVED	t3	ref	name	name	63		1	Using wh

3 rows in set (0.00 sec)

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，衍生=DERIVED，所以这里就体现了explain的第一个用处，那就是它能看出表的读取和加载顺序！

2、select_type

SIMPLE、PRIMARY、SUBQUERY、DERIVED、UNION、UNION RESULT、

查询的类型，主要是用于区别普通查询、联合查询、子查询等的复杂查询

SIMPLE：简单的 select 查询，查询中不包含子查询或者 UNION

PRIMARY：查询中若包含任何复杂的子部分，最外层查询则被标记为 PRIMARY

SUBQUERY：在 SELECT 或 WHERE 列表中包含了子查询

DERIVED：在 FROM 列表中包含的子查询被标记为 DERIVED(衍生)，MySQL 会递归执行这些子查询，把结果放在临时表里。

UNION：若第二个 SELECT 出现在 UNION 之后，则被标记为 UNION；若 UNION 包含在 FROM 子句的子查询中，外层 SELECT 将被标记为: DERIVED

UNION RESULT：从 UNION 表获取结果的 SELECT

```
Database changed
mysql> explain select * from tbl_emp a left join tbl_dept b on a.deptId = b.id
-> union
-> select * from tbl_emp a right join tbl_dept b on a.deptId = b.id;
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered
1	PRIMARY	a	NULL	ALL	NULL	NULL	NULL	NULL	9	100.00
1	PRIMARY	b	NULL	ALL	PRIMARY	NULL	NULL	NULL	5	100.00
2	UNION	b	NULL	ALL	NULL	NULL	NULL	NULL	5	100.00
2	UNION	a	NULL	ALL	fk_dept_id	NULL	NULL	NULL	9	100.00
NULL	UNION RESULT	<union1,2>	NULL	ALL	NULL	NULL	NULL	NULL	NULL	NULL

5 rows in set, 1 warning (0.01 sec)

所以这样就看到了数据读取操作的操作类型

3、table

这个基本不用说，显示这一行的数据是关于哪张表的

4、type

type 显示的是访问类型，是较为重要的一个指标，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

一般来说，得保证查询至少达到 **range** 级别，最好能达到 **ref**

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

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	tbl_emp	NULL	ALL	NULL	NULL	NULL	NULL	9	100.00	NULL

1 row in set, 1 warning (0.00 sec)

这样查询的类型就是 ALL，如果数据达到百万级别的一个全表扫描，那么性能肯定会下降的！

常见的 type 就这些：system > const > eq_ref > ref > range > index > ALL

- system 表只有一行记录(等于系统表)，这是 const 类型的特例，平时不会出现，这个也可以忽略不计

- const 表示通过索引一次就找到了，const用于比较primary key或者unique索引。因为只匹配一行数据，所以很快。如将主键至于where列表中，MySQL就能将该查询转换为一个常量，比如where id = 1就被当成是常量

```
mysql> explain select * from table_01 where id = 1;
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref
1	SIMPLE	table_01	NULL	const	PRIMARY	PRIMARY	4	const

1 row in set, 1 warning (0.00 sec)

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

```
mysql> explain select * from table_01, table_02 where table_01.id = table_02.id;
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows
1	SIMPLE	table_02	NULL	index	PRIMARY	PRIMARY	4	NULL	1
1	SIMPLE	table_01	NULL	eq_ref	PRIMARY	PRIMARY	4	day02.table_02.id	1

2 rows in set, 1 warning (0.00 sec)

- ref 非唯一性索引扫描，返回匹配某个单独值的所有行本质上也是一种索引访问，它返回所有匹配某个单独值的行。然而，能会找到多个符合条找和扫描的混合，与eq_ref对比，其实可以发现ref是对非唯一索引进行扫描，其实也就是对数据表的非主键字段建立索引，然后通过这个索引进行扫描，结果可想而知肯定是非唯一性的！
- range 只检索给定范围的行，使用一个索引来选择行。key列显示使用了哪个索引，一般就是在你的 where语句中出现 between、<、>、in等的查询，这种范围扫描索引比全表扫描要好，因为它只需要开始于索引的某一点，而结束于另一点，不用扫描全部索引。

```
mysql> explain select * from table_01 where id between 0 and 50;
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	table_01	NULL	range	PRIMARY	PRIMARY	4	NULL	2	100.00	Using where

1 row in set, 1 warning (0.01 sec)

```
mysql> explain select * from table_01 where id in (1, 20);
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	table_01	NULL	range	PRIMARY	PRIMARY	4	NULL	2	100.00	Using where

1 row in set, 1 warning (0.00 sec)

- index Full Index scan，index与ALL区别为index类型只遍历索引树。这通常比ALL快，因为索引文件通常比数据文件小(也就是说虽然叫all和index都是读全表,但 index是从索引中读取的，而all是从硬盘中读的)

```
mysql> explain select * from table_01 where id in (1, 20);
```

id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	SIMPLE	table_01	NULL	range	PRIMARY	PRIMARY	4	NULL	2	100.00	Using where

1 row in set, 1 warning (0.00 sec)

- all FullTable scan，将遍历全表以找到匹配的行

5、possible_keys

显示可能应用在这张表中的索引，一个或多个。

查询涉及的字段上若存在索引，则该索引将被列出，但不一定被查询实际使用

6、key

实际使用的索引。如果为null则没有使用索引

查询中若使用了覆盖索引，则索引和查询的select字段重叠。

所以possible_keys和key实际上是告诉了使用者，MySQL理论上会用到哪些索引，实际上会用到哪些索引。

```
mysql> create index idx_col1_col2 on t2(col1, col2);
Query OK, 1001 rows affected (0.17 sec)
Records: 1001 Duplicates:0 Warnings:0
```

```
mysql> explain select col1, col2 from t2;
```

select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
SIMPLE	t2	NULL	index	NULL	idx_col1_col2	4	NULL	2	100.00	Using index

这就是覆盖索引，也就是如果你要查询的字段顺序正好与索引建立的顺序相等，那么查询类型那直接变为index查询，而不是ref类型！

7、key_len

表示索引中使用的字节数，可通过该列计算查询中使用的索引的长度。在不损失精确性的情况下，长度越短越好

key_len显示的值为索引最大可能长度，并非实际使用长度，即key_len是根据表定义计算而得，不是通过表内检索出的

```
mysql> desc t1;
```

Field	Type	Null	Key	Default	Extra
ID	int(11)	NO	PRI	NULL	auto_increment
col1	char(4)	YES	MUL	NULL	
col2	char(4)	YES		NULL	

```
3 rows in set (0.00 sec)
```

```
mysql> explain select * from t1 where col1 = 'ab';
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	t1	ref	idx_col1_col2	idx_col1_col2	13	const	143	

```
1 row in set (0.00 sec)
```

```
mysql> explain select * from t1 where col1 = 'ab' and col2 = 'ac';
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	SIMPLE	t1	ref	idx_col1_col2	idx_col1_col2	26	const,const	1	

这个其实比较容易理解，精度越高，需要的索引字节数也变长了，很显然，更高精度的查询付出的代价就是key_len变长了！

8、ref

显示索引那一列被使用了，如果可能的话，是一个常数。那些列或常量被用于查找索引列上的值，比如：

```
mysql> explain select t2.* from (select t3.id from t3 where t3.name='')s1, t2 where s1.id=t2.id;
```

id	select_type	table	type	possible_keys	key	key_len	ref	rows	Extra
1	PRIMARY	<derived2>	system	NULL	NULL	NULL	NULL	1	
1	PRIMARY	t2	const	PRIMARY	PRIMARY	4	const	1	
2	DERIVED	t3	ref	name	name	63		1	Using wh

```
3 rows in set (0.00 sec)
```

```
mysql> explain select * from t1, t2 where t1.col1 = t2.col1 and t1.col2 = 'ac';
+-----+-----+-----+-----+-----+-----+-----+-----+
| id | ... | table | type | possible_keys | key | key_len | ref | rows |
+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | ... | t2 | ALL | NULL | NULL | NULL | NULL | 640 |
| 1 | ... | t1 | ref | idx_col1_col2 | idx_col1_col2 | 26 | shared.t2.col1,const | 82 |
+-----+-----+-----+-----+-----+-----+-----+-----+
2 rows in set (0.01 sec)
```

由key_len可知t1表的idx_col1_col2被充分使用，col1匹配t2表的col1，col2匹配了一个常量，即 'ac'

所以通过这里我们已经知道哪些索引可以使用，哪些索引被实际使用！

9、rows

显示索引那一列被使用了，如果可能的话，是一个常数。那些列或常量被用于查找索引列上的值，越少越好！

```
mysql> explain select * from t1, t2 where t1.id = t2.id and t2.col1 = 'ac';
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows | Extra |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | t2 | ALL | PRIMARY | NULL | NULL | NULL | 640 | Using where |
| 1 | SIMPLE | t1 | eq_ref | PRIMARY | PRIMARY | 4 | shared.t2.ID | 1 |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
2 rows in set (0.00 sec)
```

mysql> create index idx_col1_col2 on t2(col1,col2);
Query OK, 1001 rows affected (0.17 sec)
Records: 1001 Duplicates: 0 Warnings: 0

```
mysql> explain select * from t1, t2 where t1.id = t2.id and t2.col1 = 'ac';
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| id | select_type | table | type | possible_keys | key | key_len | ref | rows |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
| 1 | SIMPLE | t2 | ref | PRIMARY,idx_col1_col2 | idx_col1_col2 | 195 | const | 142 |
| 1 | SIMPLE | t1 | eq_ref | PRIMARY | PRIMARY | 4 | shared.t2.ID | 1 |
+-----+-----+-----+-----+-----+-----+-----+-----+-----+-----+
2 rows in set (0.00 sec)
```

通过上面的例子进行分析，在没建立索引前，扫描了641行，先加载t2，再加载t1，查询t2表的时候是ALL，也就是逐行扫描，并且本来可以使用主键索引，但是依旧没什么意义，所以还是以逐行扫描的方式进行查询，这样会有640行结果。接下来对t2表建立复合索引，所以可供选择的索引就有主键索引、自己新建的复合索引。很明显，新建复合索引之后呢，查询的时候不再采用逐行扫描的方式，而是选择了复合索引，行数降到了142行，总共143行就可以搞定这个查询问题！

10、Extra

额外的，扩展的。包含不适合在其他列中显示但十分重要的额外信息！

① Using filesort

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

29课时-04:16