19 | 基础篇总结：如何理解查询优化、通配符以及存储过程?

到这一篇的时候，意味着 SQL 专栏的基础部分正式更新完毕。在文章更新的时候，谢谢大家积极地评论和提问，让专栏增色不少。我总结了一些基础篇的常见问题，希望能对你有所帮助。答疑篇主要包括了 DBMS、查询优化、存储过程、事务处理等一些问题。

关于各种 DBMS 的介绍

答疑 1

文章中有句话不太理解，“列式数据库是将数据按照列存储到数据库中，这样做的好处是可以大量降低系统的 I/O”，可以解释一些“降低系统 I/O”是什么意思吗?

解答

行式存储是把一行的数据都串起来进行存储，然后再存储下一行。同样，列式存储是把一列的数据都串起来进行存储，然后再存储下一列。这样做的话，相邻数据的数据类型都是一样的，更容易压缩，压缩之后就自然降低了 I/O。

我们还需要从数据处理的需求出发，去理解行式存储和列式存储。数据处理可以分为 OLTP（联机事务处理）和 OLAP（联机分析处理）两大类。

OLTP 一般用于处理客户的事务和进行查询，需要随时对数据表中的记录进行增删改查，对实时性要求高。

OLAP 一般用于市场的数据分析，通常数据量大，需要进行复杂的分析操作，可以对大量历史数据进行汇总和分析，对实时性要求不高。

那么对于 OLTP 来说，由于随时需要对数据记录进行增删改查，更适合采用行式存储，因为一行数据的写入会同时修改多个列。传统的 RDBMS 都属于行式存储，比如 Oracle、SQL Server 和 MySQL 等。

对于 OLAP 来说，由于需要对大量历史数据进行汇总和分析，则适合采用列式存储，这样的话汇总数据会非常快，但是对于插入（INSERT）和更新（UPDATE）会比较麻烦，相比于行式存储性能会差不少。

所以说列式存储适合大批量数据查询，可以降低 I/O，但如果对实时性要求高，则更适合行式存储。

关于查询优化

答疑 1

在 MySQL 中统计数据表的行数，可以使用三种方式：`SELECT COUNT(*)`、`SELECT COUNT(1)`和`SELECT COUNT(具体字段)`，使用这三者之间的查询效率是怎样的？之前看到说是：`SELECT COUNT(*)` > `SELECT COUNT(1)` > `SELECT COUNT(具体字段)`。

解答

在 MySQL InnoDB 存储引擎中，`COUNT(*)`和`COUNT(1)`都是对所有结果进行`COUNT`。如果有 WHERE 子句，则是对所有符合筛选条件的数据行进行统计；如果没有 WHERE 子句，则是对数据表的数据行数进行统计。

因此`COUNT(*)`和`COUNT(1)`本质上并没有区别，执行的复杂度都是`O(N)`，也就是采用全表扫描，进行循环 + 计数的方式进行统计。

如果是 MySQL MyISAM 存储引擎，统计数据表的行数只需要`O(1)`的复杂度，这是因为每张 MyISAM 的数据表都有一个 meta 信息存储了`row_count`值，而一致性则由表级锁来保证。因为 InnoDB 支持事务，采用行级锁和 MVCC 机制，所以无法像 MyISAM 一样，只维护一个`row_count`变量，因此需要采用扫描全表，进行循环 + 计数的方式来完成统计。

需要注意的是，在实际执行中，`COUNT(*)`和`COUNT(1)`的执行时间可能略有差别，不过你还是可以把它俩的执行效率看成是相等的。

另外在 InnoDB 引擎中，如果采用`COUNT(*)`和`COUNT(1)`来统计数据行数，要尽量采用二级索引。因为主键采用的索引是聚簇索引，聚簇索引包含的信息多，明显会大于二级索引（非聚簇索引）。对于`COUNT(*)`和`COUNT(1)`来说，它们不需要查找具体的行，只是统计行数，系统会自动采用占用空间更小的二级索引来进行统计。

然而如果想要查找具体的行，那么采用主键索引的效率更高。如果有多个二级索引，会使用 key_len 小的二级索引进行扫描。当没有二级索引的时候，才会采用主键索引来进行统计。

这里我总结一下：

1. 一般情况下，三者执行的效率为 COUNT(*)= COUNT(1)> COUNT(字段)。我们尽量使用COUNT(*)，当然如果你要统计的是某个字段的非空数据行数，则另当别论，毕竟比较执行效率的前提是结果一样才可以。
2. 如果要统计COUNT(*)，尽量在数据表上建立二级索引，系统会自动采用key_len小的二级索引进行扫描，这样当我们使用SELECT COUNT(*)的时候效率就会提升，有时候可以提升几倍甚至更高。

答疑 2

在 MySQL 中，LIMIT关键词是最后执行的，如果可以确定只有一条结果，那么就起不到查询优化的效果了吧，因为LIMIT是对最后的结果集过滤，如果结果集本来就只有一条，那就没有什么用了。

解答

如果你可以确定结果集只有一条，那么加上LIMIT 1的时候，当找到一条结果的时候就不会继续扫描了，这样会加快查询速度。这里指的查询优化针对的是会扫描全表的 SQL 语句，如果数据表已经对字段建立了唯一索引，那么可以通过索引进行查询，不会全表扫描的话，就不需要加上LIMIT 1了。

关于通配符的解释

关于查询语句中通配符的使用理解，我举了一个查询英雄名除了第一个字以外，包含“太”字的英雄都有谁的例子，使用的 SQL 语句是：

复制代码

```
SQL> SELECT name FROM heros WHERE name LIKE '_% 太 %'
```

（_）匹配任意一个字符，（%） 匹配大于等于 0 个任意字符。

所以通配符'_%太%'说明在第一个字符之后需要有“太”字，这里就不能匹配上“太乙真人”，但是可以匹配上“东皇太一”。如果数据表中有“太乙真人太太”，那么结果集中也可以匹配到。

另外，单独的`LIKE '%'`无法查出 NULL 值，比如：`SELECT * FROM heros WHERE role_assist LIKE '%'`。

答疑 4

可以理解在 WHERE 条件字段上加索引，但是为什么在 ORDER BY 字段上还要加索引呢？这个时候已经通过 WHERE 条件过滤得到了数据，已经不需要再筛选过滤数据了，只需要根据字段排序就好了。

解答

在 MySQL 中，支持两种排序方式，分别是 FileSort 和 Index 排序。在 Index 排序中，索引可以保证数据的有序性，不需要再进行排序，效率更高。而 FileSort 排序则一般在内存中进行排序，占用 CPU 较多。如果待排结果较大，会产生临时文件 I/O 到磁盘进行排序的情况，效率较低。

所以使用 ORDER BY 子句时，应该尽量使用 Index 排序，避免使用 FileSort 排序。当然你可以使用 explain 来查看执行计划，看下优化器是否采用索引进行排序。

优化建议：

1. SQL 中，可以在 WHERE 子句和 ORDER BY 子句中使用索引，目的是在 WHERE 子句中避免全表扫描，在 ORDER BY 子句避免使用 FileSort 排序。当然，某些情况下全表扫描，或者 FileSort 排序不一定比索引慢。但总的来说，我们还是要避免，以提高查询效率。一般情况下，优化器会帮我们进行更好的选择，当然我们也需要建立合理的索引。
2. 尽量使用 Index 完成 ORDER BY 排序。如果 WHERE 和 ORDER BY 后面是相同的列就使用单索引列；如果不同就使用联合索引。
3. 无法使用 Index 时，需要对 FileSort 方式进行调优。

答疑 5

ORDER BY 是对分的组排序还是对分组中的记录排序呢？

解答

ORDER BY 就是对记录进行排序。如果你在 ORDER BY 前面用到了 GROUP BY，实际上这是一种分组的聚合方式，已经把一组的数据聚合成为了一条记录，再进行排序的时候，相当于对分的组进行了排序。

答疑 6

请问下关于 SELECT 语句内部的执行步骤。

解答

一条完整的 SELECT 语句内部的执行顺序是这样的：

1. FROM 子句组装数据（包括通过 ON 进行连接）；
2. WHERE 子句进行条件筛选；
3. GROUP BY 分组；
4. 使用聚集函数进行计算；
5. HAVING 筛选分组；
6. 计算所有的表达式；
7. SELECT 的字段；
8. ORDER BY 排序；
9. LIMIT 筛选。

答疑 7

不太理解哪种情况下应该使用 EXISTS，哪种情况应该用 IN。选择的标准是看能否使用表的索引吗？

解答

索引是个前提，其实选择与否还是要看表的大小。你可以将选择的标准理解为小表驱动大表。在这种方式下效率是最高的。

比如下面这样：

复制代码

```
SELECT * FROM A WHERE cc IN (SELECT cc FROM B)
SELECT * FROM A WHERE EXISTS (SELECT cc FROM B WHERE B.cc=A.cc)
```

当 A 小于 B 时，用 EXISTS。因为 EXISTS 的实现，相当于外表循环，实现的逻辑类似于：

复制代码

```
for i in A
    for j in B
        if j.cc == i.cc then ...
```

当 B 小于 A 时用 IN，因为实现的逻辑类似于：

复制代码

```
for i in B
    for j in A
        if j.cc == i.cc then ...
```

哪个表小就用哪个表来驱动，A 表小就用 EXISTS，B 表小就用 IN。

关于存储过程

答疑 1

在使用存储过程声明变量时，都支持哪些数据类型呢？

解答

不同的 DBMS 对数据类型的定义不同，你需要查询相关的 DBMS 文档。以 MySQL 为例，常见的数据类型可以分成三类，分别是数值类型、字符串类型和日期 / 时间类型。

答疑 2

“IN 参数必须在调用存储过程时指定”的含义是什么？我查询了 MySQL 的存储过程定义，可以不包含 IN 参数。当存储过程的定义语句里有 IN 参数时，存储过程的语句中必须用到这个参数吗？

解答

如果存储过程定义了 IN 参数，就需要在调用的时候传入。当然在定义存储过程的时候，如果不指定参数类型，就默认是 IN 类型的参数。因为 IN 参数在存储过程中是默认值，可以

省略不写。比如下面两种定义方式都是一样的：

复制代码

```
CREATE PROCEDURE `add_num`(IN n INT)
```

复制代码

```
CREATE PROCEDURE `add_num`(n INT)
```

在存储过程中的语句里，不一定要用到 IN 参数，只是在调用的时候需要传入这个。另外 IN 参数在存储过程中进行了修改，也不会进行返回的。如果想要返回参数，需要使用 OUT，或者 INOUT 参数类型。

关于事务处理

答疑 1

如果`INSERT INTO test SELECT '关羽';`之后没有执行 COMMIT，结果应该是空。但是我执行出来的结果是`'关羽'`，为什么 ROLLBACK 没有全部回滚?

代码如下：

复制代码

```
CREATE TABLE test(name varchar(255), PRIMARY KEY (name)) ENGINE=InnoDB;
BEGIN;
INSERT INTO test SELECT '关羽';
BEGIN;
INSERT INTO test SELECT '张飞';
INSERT INTO test SELECT '张飞';
ROLLBACK;
SELECT * FROM test;
```

解答

先解释下连续 BEGIN 的情况。

在 MySQL 中 BEGIN 用于开启事务，如果是连续 BEGIN，当开启了第一个事务，还没有进行 COMMIT 提交时，会直接进行第二个事务的 BEGIN，这时数据库会隐式地 COMMIT 第一个事务，然后再进入到第二个事务。

为什么 ROLLBACK 没有全部回滚呢?

因为 ROLLBACK 是针对当前事务的，在 BEGIN 之后已经开启了第二个事务，当遇到 ROLLBACK 的时候，第二个事务都进行了回滚，也就得到了第一个事务执行之后的结果即“关羽”。

关于事务的 ACID，以及我们使用 COMMIT 和 ROLLBACK 来控制事务的时候，有一个容易出错的地方。

在一个事务的执行过程中可能会失败。遇到失败的时候是进行回滚，还是将事务执行过程中已经成功操作的来进行提交，这个逻辑是需要开发者自己来控制的。

这里开发者可以决定，如果遇到了小错误是直接忽略，提交事务，还是遇到任何错误都进行回滚。如果我们强行进行 COMMIT，数据库会将这个事务中成功的操作进行提交，它会认为你觉得已经是 ACID 了（就是你认为可以做 COMMIT 了，即使遇到了一些小问题也是可以忽略的）。

我在今天的文章里重点解答了一些问题，还有一些未解答的会留在评论里进行回复。最后出一道思考题吧。

请你自己写出下面操作的运行结果（你可以把它作为一道笔试题，自己写出结果，再与实际的运行结果进行比对）：

复制代码

```
DROP TABLE IF EXISTS test;
CREATE TABLE test(name varchar(255), PRIMARY KEY (name)) ENGINE=InnoDB;
BEGIN;
INSERT INTO test SELECT '关羽';
BEGIN;
INSERT INTO test SELECT '张飞';
INSERT INTO test SELECT '张飞';
COMMIT;
SELECT * FROM test;
```