23 关于 SQL 查询语句，有什么好的建议吗？

更新时间：2020-05-11 09:27:46

> 人的影响短暂而微弱，书的影响则广泛而深远。——普希金

对于 SQL 查询来说，查询的方式、列值的顺序、索引的定义等等都有可能会影响到查询效率。当然，我们同样不能忽视数据表的定义，毕竟数据表是查询的源头。这一节里，我会结合工作经验对 SQL 语句的编写、数据表的定义给出建议，旨在提高你的 SQL 编写能力，提升工作效率。

1. 建表时需要考虑的优化策略

创建数据表是迈入 SQL 查询的第一步，它的重要性自然不言而喻。除了对业务需求的分析之外，理解 MySQL 的特性也是必备的技能。下面，我将结合我在开发、学习中遇到的问题、经验来谈一谈关于建表的优化策略。

1.1 ENUM 也许会比 CHAR/VARCHAR 更好

MySQL 中的 ENUM 类型是一个热点话题，且大多数时候它都是被排斥的对象。其主要原因是更改 ENUM 类型的字段，代价是十分昂贵的。例如，我们把 ENUM(‘a’, ‘b’, ‘c’) 修改为 ENUM(‘a’, ‘b’, ‘d’)，MySQL 就需要重构整个数据表，并且在数据池中查找无效值 c。可想而知，如果枚举值可能会出现变化，ENUM 类型肯定是不合适的。

但是，仍然不可否认，ENUM 类型的速度是非常快的，因为 MySQL 实际上会使用 TINYINT 保存数据，字符串只是它的外衣。所以，当你能够确定枚举值是固定、不会发生变化的，ENUM 类型会比 CHAR/VARCHAR 好很多。

1.2 保证列值是 NOT NULL 的

首先来说，为什么很多人会把列值设置为允许 NULL 呢 ？主要有两点原因：

- 数据表列默认就是 NULL 的，对于初级或者经验不足的同学没有显示指定 NOT NULL

- NULL 列在插入时不需要指定数据，也不需要判断，比较方便简约

但是，为什么又需要保证列值是 NOT NULL 的呢？来看一看 MySQL 官方文档怎么说吧：

> NULL columns require additional space in the row to record whether their values are NULL. For MyISAM tables, each NULL column takes one bit extra, rounded up to the nearest byte.

它的意思是说：NULL 列需要额外的存储空间，且 MySQL 内部需要做特殊的处理。MySQL 难以优化 NULL 列的查询，它会使索引、索引统计更加复杂。特别是对于 MyISAM 存储引擎的表，它可能会导致固定大小的索引变成可变大小的索引。

1.3 IP 地址存成 UNSIGNED INT

相信你一定知道，IP 地址可以与整型互转，且 MySQL 也提供了这样的函数。如下所示：

```
mysql> SELECT INET_ATON('10.0.5.9');
+----------------------+
| INET_ATON('10.0.5.9') |
+----------------------+
|            167773449 |
+----------------------+
1 row in set (0.01 sec)

mysql> SELECT INET_NTOA(167773449);
+----------------------+
| INET_NTOA(167773449) |
+----------------------+
| 10.0.5.9             |
+----------------------+
1 row in set (0.00 sec)
```

既然这样，就一定不要去使用 CHAR 或者 VARCHAR 去存储 IP 地址，因为这至少需要15个字节。如果用 INT（UNSIGNED），则只需要4个字节，且这是一个定长的（IP 地址根据网段的不同，不一定是定长的）字段，查询上也更具有优势。

1.4 选择适当的索引字段，避免过度索引

索引字段的选择一定要去结合业务需求，找到或者预测会作为查询条件且比较频繁的字段，并在这些字段上建立索引、联合索引等等。但是，不应该在 NULL 列或者“大的”数据列上创建索引。

虽然索引能加速查询过程，但是，是不是说索引越多越好呢？当然不是：每一个索引都会占据额外的存储空间，且会降低 INSERT 和 UPDATE 的效率，因为每一次数据更新都可能导致索引的重建。所以，索引不要创建太多，不能过度，通常单表不应该超过5个。否则，就应该思考下这张表设计的合理性。

1.5 比较少见的固定长度表

对于 MySQL 来说，如果一张表的所有列长度都是固定的，即表中不存在 VARCHAR、TEXT 这样的列。那么，这个表就被称为是 Static 或 Fixed-Length 表。这样的表肯定是比较少见的，MySQL 的存储引擎也会针对它做一些优化。

Static 表最大的特点就是它的每一行数据记录都是固定长度的，所以，计算下一行数据的偏移量是非常简单的。由此，读取行记录的速度也是更快的。但是，它也会有一定的副作用：不论你用不用，MySQL 总是会分配那么多空间。

1.6 99% 的表都应该是数字主键

你几乎（报表可以不定义）需要给每张表都定义主键，因为 MySQL 的存储引擎在很多情况下都需要依赖主键，例如：存储、分区等等。且主键可以唯一的确定一行记录，也减轻了数据查询的负担。

同时，这个主键应该是数字类型（int 或 bigint），而不是其他的类型，且给上 AUTO_INCREMENT 自增标识。这是因为：

- 对于自增主键，每次插入新的记录，记录就会顺序添加到当前索引节点的后续位置
- 对于非自增主键，每次插入的主键值近似随机，因此每次新纪录都要被插到现有索引页的中间某个位置，此时 MySQL 不得不为了将新记录插到合适位置而移动数据（造成节点的分裂和重组）

1.7 使用更小的数据类型

毫无疑问，MySQL 的磁盘存储是造成 IO 瓶颈的“罪魁祸首”。所以，选择足够用且更小的数据类型是非常有必要的。它会带来两个主要的优点：

- 数据记录占用的存储空间更少
- 可以将更多的数据读入内存，提升检索性能

一个比较经典的案例是地域字典表，它用于存储省和市。此时，主键就不需要使用 INT 或 BIGINT 了，而是应该选择 SMALLINT。

1.8 字符集编码需要统一

对于使用 MySQL 经验不足的同学来说，大概率会遇到中文乱码的问题：明明大家存储的都是中文，在别人的电脑上能正确显示，但是，在我的机器上却显示乱码。这其实就是字符集编码的问题了。

这里，我给出一条通用建议：操作系统、服务器、客户端、库、表等等，所有你能想到的会与 MySQL 产生交集的地方，都设置成一样的字符集编码，例如：UTF-8。

2. 查询时需要考虑的优化策略

关于 SQL 查询，我们一定见识过相同功能但是写法不同的 SQL 语句，查询性能会差距很大。对于经验不足或者犯错次数过少的同学来说，写出性能低下的 SQL 查询再正常不过了。虽然能够完成查询需求，但是严重的情况下会影响到整个系统的性能。接下来，我也将结合我在开发、学习中遇到的问题、经验来谈一谈关于查询的优化策略。

2.1 尽量避免在 SQL 中出现函数

MySQL 提供了很多功能强大的函数，例如：数学运算、时间计算、字符串拼接等等。但是，我并不建议在 SQL 查询中对列值进行函数计算。主要是基于两点原因：

- 函数计算会导致 SQL 执行速度变慢，增加访问延迟
- 函数计算会消耗更多的 MySQL 资源（CPU 和内存），导致其他客户端饥饿

另外，一个 MySQL 服务可能会被很多产品线使用，更不应该去“过度消费”共享资源。如果确实需要对列值做处理计算，把它放到代码中去，我们也可以更加灵活的控制处理过程。

2.2 慎用 IN 和 NOT IN

即使你在某一列上加了索引，如果使用不恰当的方式查询，仍然会导致全表扫描。先看看下面两个查询语句：

```
SELECT id, name, salary FROM worker WHERE salary IN (1, 2, 3);
SELECT id, name, salary FROM worker WHERE salary NOT IN (1, 2, 3);
```

由于 IN 和 NOT IN 中定义的值是不确定的，所以，MySQL 不会对上述的两个查询使用索引。这里可以做的优化是，对于连续的数值，我们可以使用 BETWEEN 来代替 IN。由于 BETWEEN 定义的是一个连续的区间，所以，可以使用到索引。如下所示：

```
SELECT id, name, salary FROM worker WHERE salary BETWEEN 1 AND 3;
```

2.3 禁用 ORDER BY RAND()

“随机”是很常见的需求，连 MySQL 也预料到了这一点，所以，它提供了 `ORDER BY RAND()`。可以看看下面的两个例子：

```
mysql> SELECT * FROM worker LIMIT 3;
+----+------+------+--------+----------+
| id | type | name | salary | version |
+----+------+------+--------+----------+
|  1 | A    | tom  |   1800 |       0 |
|  2 | B    | jack |   2100 |       0 |
|  3 | C    | pony |   NULL |       0 |
+----+------+------+--------+----------+
3 rows in set (0.01 sec)

mysql> SELECT * FROM worker ORDER BY RAND() LIMIT 3;
+----+------+--------+--------+----------+
| id | type | name   | salary | version |
+----+------+--------+--------+----------+
| 10 | C    | jarvis |   1800 |       0 |
|  6 | C    | tack   |   1200 |       0 |
|  4 | B    | tony   |   3600 |       0 |
+----+------+--------+--------+----------+
3 rows in set (0.02 sec)
```

很明显，`ORDER BY RAND()` 打乱了返回数据。但是，为什么这种随机操作不放在代码中去完成呢？MySQL 不得不去执行 `RAND` 函数，且需要对数据记录进行排序，这会使数据库的性能呈指数级下降。

2.4 功能强大的 LIMIT

我们的日常工作中，使用 `LIMIT` 关键字的频率还是非常高的。确实，这是一个非常重要的优化，当我们不需要返回“很多”数据时，加上 `LIMIT` 限定关键字，数据库引擎将会在找到所需的数据记录之后停止检索，而不是继续执行任务。

可是，你知道吗？`LIMIT` 还能避免误删除数据记录：当你能够确定所要删除的数据记录数时，在 `DELETE` 后面加上 `LIMIT` 吧。例如：我想要删除 worker 表中的三条记录，可以这样写：

```
mysql> DELETE FROM worker LIMIT 3;
Query OK, 3 rows affected (0.06 sec)
```

2.5 尽量避免在 WHERE 子句中对字段进行表达式操作

虽然 SQL 支持我们做表达式计算，但是，这将会导致优化器引擎放弃使用索引，转而去进行全表的扫描，严重降低服务器性能。就比如下面这个例子：

```
-- 假设我们给 salary 创建了索引
mysql> SELECT * FROM worker WHERE salary * 2 = 4000;
+----+------+-------+--------+---------+
| id | type | name  | salary | version |
+----+------+-------+--------+---------+
|  8 | B    | clock |   2000 |       0 |
+----+------+-------+--------+---------+
1 row in set (0.01 sec)
```

其实，完全可以把表达式计算“调换”过来，即 `WHERE salary = 4000 / 2`。此时，SQL 优化器将会使用到索引进行检索。

2.6 尽量避免在 WHERE 子句中对字段进行函数操作

其实这和上面的一条建议非常类似，在 WHERE 子句中对字段进行函数操作，即使是所在字段创建了索引，优化器也将“视而不见”，直接进行全表的扫描。举个例子：

```
-- 假设我们给 name 创建了索引
mysql> SELECT * FROM worker WHERE SUBSTRING(name, 1, 2) = 'to';
+----+------+------+--------+---------+
| id | type | name | salary | version |
+----+------+------+--------+---------+
|  4 | B    | tony |   3600 |       0 |
+----+------+------+--------+---------+
1 row in set (0.01 sec)
```

由于对 name 字段加上了 `SUBSTRING` 函数，导致列值的不确定性（这是不能使用索引的原因），所以，SQL 将不会使用索引。不过，我们可以想办法修改下这个查询：

```
-- 由于至少知道 name 的前两个字节是“to”，所以，可以使用索引检索
mysql> SELECT * FROM worker WHERE name LIKE 'to%';
+----+------+------+--------+---------+
| id | type | name | salary | version |
+----+------+------+--------+---------+
|  4 | B    | tony |   3600 |       0 |
+----+------+------+--------+---------+
1 row in set (0.01 sec)
```

2.7 尽量避免在 WHERE 子句中判断不等于

“不等于”在 SQL 中是“非”的语义，这同样是很难使用到索引的，毕竟索引是有序存储来达到快速匹配“是”的语义。所以，如果检索的数据量不大，最好的做法是在代码中进行过滤操作。

最后，来做一个总结：不要在 WHERE 子句中对字段值进行函数、算数运算或其他的表达式计算，这将会使列值处于不确定性的状态，SQL 优化器大概率不能使用到索引去优化查询。

2.8 GROUP BY 考虑禁止排序

我们几乎都知道，`GROUP BY` 的性能是比较低的，特别是对于比较大的数据记录或无法使用索引时（想要使用索引，必须满足 `GROUP BY` 的字段同时存放于同一个索引中，且该索引是一个有序索引），需要通过临时表来完成操作。

但是，`GROUP BY` 还有一个特性很容易被忽略：默认情况下，MySQL 会按照 `GROUP BY` 所指定的字段进行排序，不论你有没有指定 `ORDER BY`。如果我们的查询需要 `GROUP BY`，但是，你又想避免排序带来的性能损耗，可以这样：

```
-- 显示指定 ORDER BY NULL 禁止排序
mysql> SELECT version, COUNT(version) FROM worker GROUP BY version ORDER BY NULL;
+---------+----------------+
| version | COUNT(version) |
+---------+----------------+
|       0 |              7 |
|       1 |              1 |
+---------+----------------+
2 rows in set (0.00 sec)
```

2.9 避免出现大事务

对“大事务”的定义其实是比较模糊的，我们通常认为运行时间比较长，操作数据比较多的事务就是大事务。大事务主要会造成两个问题：

- 锁定太多的数据，造成大量的阻塞和锁超时，严重影响数据库性能
- 执行时间过长，延迟较大，容易造成主从延迟

所以，更好的做法是将大事务进行拆分，把单个复杂的 SQL 拆分为多个小的 SQL。同时，简单较小的 SQL 更容易利用到 MySQL 的查询缓存。

2.10 合理的分页非常重要

将表中的所有数据（或者根据一定的条件筛选出的数据）一次性返回给前端是很傻的做法，由此产生了分页的思想。通常，分页包含三个部分：总数、上一页和下一页，查询语句可能会像这样：

```
-- 查询 worker 表总记录数
SELECT COUNT(1) FROM worker;
-- 查询分页数据
SELECT * FROM worker ORDER BY id DESC LIMIT 0, 10;
SELECT * FROM worker ORDER BY id DESC LIMIT 10000, 10;
```

这种实现方式当然是可以的，但是，对于分页查询来说，越往后执行的时间会越长。因为对于 `LIMIT M, N` 的写法，MySQL 总是会去扫描 M + N 条数据来得到你想要的数据，慢的原因也就是大量的数据扫描。此时，我们可以换一种方式来实现：

```
SELECT * FROM worker, (SELECT id FROM worker ORDER BY id DESC LIMIT 10000, 10) worker_ WHERE worker.id = worker_.id;
```

先去查询分页中需要数据的主键 id，之后再根据主键 id 去查询你所需要的数据信息，这个查询数据的过程是依赖主键索引完成的，所以，效率提升非常明显。

2.11 小心使用 LIKE

`LIKE` 关键字实现的是模糊查询的功能，同样，你应该少用这个关键字。特别是下面这种情况：

```
-- 注意：% 放在前面
SELECT * FROM worker WHERE name LIKE '%ah';
```

由于无法确定前缀，优化器不能使用索引，导致全表扫描。如果确实需要类似的查询，应该是预估可能的前缀，再对结果集做筛选。

2.12 少用 SELECT ... FOR UPDATE

`FOR UPDATE` 由 InnoDB 存储引擎提供，且必须在事务块（BEGIN/COMMIT）中才能生效。通过 `FOR UPDATE` 语句，MySQL 会对查询结果集中每行数据都添加排他锁，其他线程对该记录的更新与删除操作都会阻塞。用法如下：

```
SELECT * FROM worker WHERE id = 10 FOR UPDATE;
```

但是，需要注意，使用 `FOR UPDATE` 必须确保字段存在索引，否则，行锁将会上升为表锁。很多情况下的死锁都是由 `FOR UPDATE` 造成。所以，除非必要，不要使用 `SELECT ... FOR UPDATE`。

3. 常见问题的思考

很多时候，我们只是听到建议说：“不要这样用”、“这样写效率很低” 等等类似的话。但是，为什么这样不好的答案却鲜有人能说得出来。这里，我将举例说明一些常见的问题，解释说明它们为什么好、又为什么不好，其目的是要知其然也知其所以然。

3.1 SELECT * 会降低查询速度，你知道吗？

关于这个问题，我以 worker(id, type, name, salary, version) 表来举例说明。除了主键（聚簇）索引之外，我还给 name 列加上了一个普通索引。此时，对于 worker 表来说，它包含两棵 B+ 树：

- 聚簇索引：是由 MySQL 根据主键自动生成的，保存的数据是 (id, type, name, salary, version)
- 辅助索引：自行定义的 name 列索引，保存的数据是 (name, id)

如果我们的查询条件（WHERE 子句）可以通过 name 来过滤，查询优化器就会选择辅助索引。且如果我们只是 `SELECT id, name`，则查询会直接返回，也就是使用到了覆盖索引。但是，如果是直接 `SELECT *`，就不得不再去查询一次聚簇索引，速度肯定会慢很多。

由于辅助索引通常都会比较小，所以，MySQL 会把它载入内存中（有可能会是这样）。如果通过覆盖索引满足要求，就不需要再去读磁盘（聚簇索引在磁盘上），性能上肯定是有大幅提升的。而对于我们的业务来说，能使用到一张表所有字段的场景是比较少的，所以，更应该去设置合理的索引，而不要使用 `SELECT *`。

3.2 你知道哪些情况会让查询缓存失效吗？

在解析器解析一个 SQL 查询之前，如果查询缓存是打开的，MySQL 会优先检查这个查询是否会命中查询缓存中的数据。如果查询恰好命中查询缓存，则直接从缓存中拿到数据返回给客户端（当然，还会校验用户权限的问题）。这种情况下，查询不会被解析、生成执行计划及执行。所以，查询性能会大幅提升。

默认情况下，查询缓存是关闭的，它所对应的 MySQL 变量是 query_cache_type（默认值是 OFF）。我们需要修改配置文件以打开查询缓存，修改 my.cnf（需要重启 MySQL 服务）：

```
# 打开查询缓存
query_cache_type = ON
```

验证查询缓存已经打开，可以查看 query_cache_type 变量的值，如下所示。

```
mysql> SHOW VARIABLES LIKE 'query_cache_type';
+------------------+-------+
| Variable_name    | Value |
+------------------+-------+
| query_cache_type | ON    |
+------------------+-------+
1 row in set (0.00 sec)
```

对于我们的线上服务来说，查询缓存大多都是打开的，这是提高查询性能最有效的方法之一。但是，需要注意，某些查询方式会导致 MySQL 不使用查询缓存，总结如下：

- 查询缓存的内容是 `SELECT` 的结果集，缓存会使用完整的 SQL 字符串作为 KEY，且区分大小写和空格。所以，只有完全一致的 SQL 才可能会命中缓存
- Prepared Statements 几乎不会命中查询缓存，即使参数完全一样
- `WHERE` 子句中如果包含任何一个不确定的函数，将永远不会命中缓存。最经典的示例是时间函数，如下的 SQL 将不会被缓存：

```
-- 由于 CURDATE 返回值是不确定的，所以，这条查询结果不会被缓存
SELECT * FROM daily_creative_stat WHERE date <= CURDATE();
```

- 特别大的结果集将不会被缓存，这个阈值由 query_cache_limit 参数控制
- 在分库分表环境下，查询缓存将不起作用
- SQL 中存在自定义函数、触发器时，查询缓存将不起作用

3.3 我可以强制使用索引查询吗？

有些情况下，虽然我们知道可以使用索引，但是 MySQL 的优化器可能会放弃。例如在 `WHERE` 子句中使用参数查询的情况：

```
mysql> SET @sal = 1000;
Query OK, 0 rows affected (0.00 sec)

-- WHERE 子句中使用参数，会导致全表扫描
mysql> SELECT id, name, salary FROM worker WHERE salary >= @sal;
```

但是，我们确切的可以知道，当前的查询语句是可以使用索引的。那么，就可以显示的声明强制索引查询。语法及使用方法如下：

```
-- 语法
FORCE INDEX(索引名称/主键)

-- 强制使用 salary_name_idx 索引（salary_name_idx 需要存在）
SELECT id, name, salary FROM worker FORCE INDEX(salary_name_idx) WHERE salary >= @sal;
```

强制存储引擎使用索引时，首先会检查索引是否可用，如果不可用，还是需要扫描全表来执行查询计划。MySQL 官方也对强制使用索引进行了说明，主要意思就是不需要担心强制索引存在的副作用，原文如下。

> The FORCE INDEX hint acts like USE INDEX (index_list), with the addition that a table scan is assumed to be very expensive. In other words, a table scan is used only if there is no way to use one of the named indexes to find rows in the table.

4. 总结

日常工作中，我们几乎每天都会接触到 MySQL（我几乎每天都会定位、排查慢查询），所以，优化查询的重要性不言而喻。我在这一节里总结的表、语句相关的优化策略，基本上能够“应付”日常的开发工作了。更多的优化方法和经验，就需要你在工作中多去思考和总结了。

5. 问题

对于创建数据表，你有怎样的经验和建议呢 ？

你能总结下，哪些使用方法会导致索引失效吗 ？

你能说一说，你在工作、学习中都使用过哪些优化查询的方法吗 ？

6. 参考资料

《高性能 MySQL（第三版）》

MySQL 官方文档：Miscellaneous Functions

MySQL 官方文档：Static (Fixed-Length) Table Characteristics

MySQL 官方文档：Data Type Storage Requirements

MySQL 官方文档：Optimization

MySQL 官方文档：The InnoDB Storage Engine

}

22 binlog 实现增量数据收集方案的设计

24 大数据量插入遇到瓶颈，我该怎样做性能优化呢？