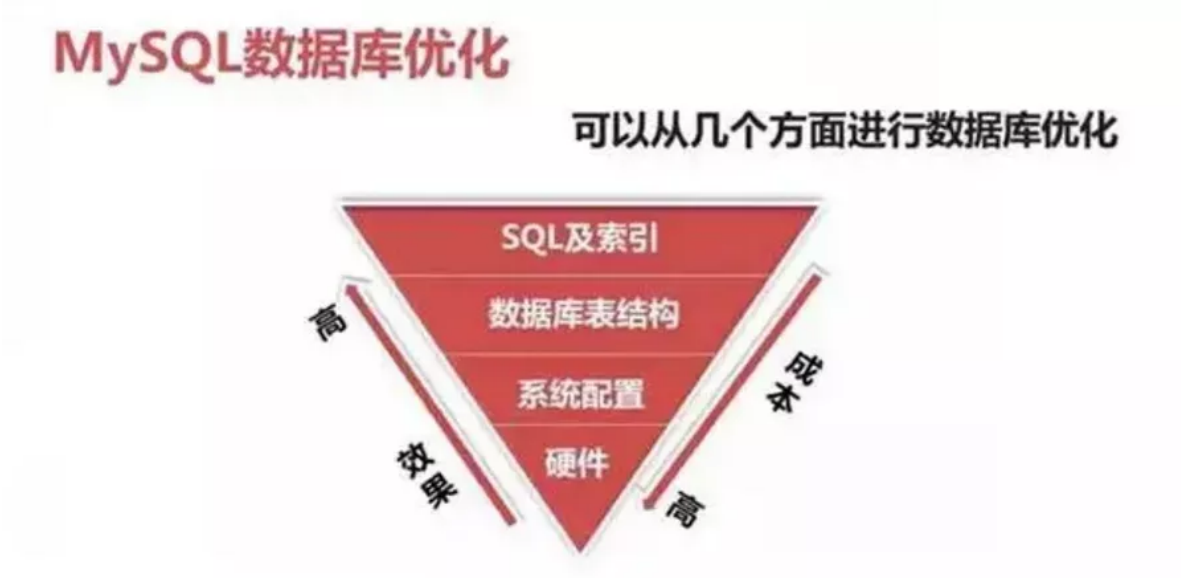
SQL优化技巧

```
@author: _陈哈哈 51CTO技术栈
@date: 2020-08-11
```

SQL 优化已经成为衡量程序猿优秀与否的硬性指标，甚至在各大厂招聘岗位职能上都有明码标注，如果是你，在这个问题上能吊打面试官还是会被吊打呢？

有朋友疑问到，SQL 优化真的有这么重要么？如下图所示，SQL 优化在提升系统性能中是：成本最低和优化效果最明显的途径。

如果你的团队在 SQL 优化这方面搞得很优秀，对你们整个大型系统可用性方面无疑是一个质的跨越，真的能让你们老板省下不止几沓子钱。



优化成本：硬件>系统配置>数据库表结构>SQL 及索引。

优化效果：硬件<系统配置<数据库表结构<SQL 及索引。

```
String result = "嗯，不错，";

if ("SQL优化经验足") {
    if ("熟悉事务锁") {
        if ("并发场景处理666") {
            if ("会打王者荣耀") {
                result += "明天入职"
            }
        }
    }
} else {
    result += "先回去等消息吧";
}

Logger.info("面试官：" + result );
```

别看了，上面这是一道送命题。

好了我们言归正传，首先，对于MySQL层优化我一般遵从五个原则：

- **减少数据访问**：设置合理的字段类型，启用压缩，通过索引访问等减少磁盘 IO。
- **返回更少的数据**：只返回需要的字段和数据分页处理，减少磁盘 IO 及网络 IO。
- **减少交互次数**：批量 DML 操作，函数存储等减少数据连接次数。
- **减少服务器 CPU 开销**：尽量减少数据库排序操作以及全表查询，减少 CPU 内存占用。
- **利用更多资源**：使用表分区，可以增加并行操作，更大限度利用 CPU 资源。

总结到 SQL 优化中，就如下三点：

- **最大化利用索引。**
- **尽可能避免全表扫描。**
- **减少无效数据的查询。**

理解 SQL 优化原理，首先要搞清楚 SQL 执行顺序。

SELECT 语句，语法顺序如下：

```
1. SELECT
2. DISTINCT <select_list>
3. FROM <left_table>
4. <join_type> JOIN <right_table>
5. ON <join_condition>
6. WHERE <where_condition>
7. GROUP BY <group_by_list>
8. HAVING <having_condition>
9. ORDER BY <order_by_condition>
10.LIMIT <limit_number>
```

SELECT 语句，执行顺序如下：

```
FROM
<表名> # 选取表，将多个表数据通过笛卡尔积变成一个表。
ON
<筛选条件> # 对笛卡尔积的虚表进行筛选
JOIN <join, left join, right join...>
<join表> # 指定join，用于添加数据到on之后的虚表中，例如left join会将左表的剩余数据添加到虚表中
WHERE
<where条件> # 对上述虚表进行筛选
GROUP BY
<分组条件> # 分组
<SUM()等聚合函数> # 用于having子句进行判断，在书写上这类聚合函数是写在having判断里面的
HAVING
<分组筛选> # 对分组后的结果进行聚合筛选
SELECT
<返回数据列表> # 返回的单列必须在group by子句中，聚合函数除外
DISTINCT
# 数据除重
ORDER BY
<排序条件> # 排序
LIMIT
<行数限制>
```

以下 SQL 优化策略适用于数据量较大的场景下，如果数据量较小，没必要以此为准，以免画蛇添足。

避免不走索引的场景

①尽量避免在字段开头模糊查询，会导致数据库引擎放弃索引进行全表扫描

如下：

```
SELECT * FROM t WHERE username LIKE '%陈%'
```

优化方式：尽量在字段后面使用模糊查询。

如下：

```
SELECT * FROM t WHERE username LIKE '陈%'
```

如果需求是要在前面使用模糊查询：

- 使用 MySQL 内置函数 INSTR（str，substr）来匹配，作用类似于 Java 中的 indexOf()，查询字符串出现的角标位置。
- 使用 FullText 全文索引，用 match against 检索。
- 数据量较大的情况，建议引用 ElasticSearch、Solr，亿级数据量检索速度秒级。
- 当表数据量较少（几千条儿那种），别整花里胡哨的，直接用 like '%xx%'。

②尽量避免使用 in 和 not in，会导致引擎走全表扫描

如下：

```
SELECT * FROM t WHERE id IN (2,3)
```

优化方式：如果是连续数值，可以用 between 代替。

如下：

```
SELECT * FROM t WHERE id BETWEEN 2 AND 3
```

如果是子查询，可以用 exists 代替。

如下：

```
-- 不走索引
select * from A where A.id in (select id from B);
-- 走索引
select * from A where exists (select * from B where B.id = A.id);
```

③尽量避免使用 or，会导致数据库引擎放弃索引进行全表扫描

如下：

```
SELECT * FROM t WHERE id = 1 OR id = 3
```

优化方式：可以用 union 代替 or。

如下：

```
SELECT * FROM t WHERE id = 1
   UNION
SELECT * FROM t WHERE id = 3
```

④尽量避免进行 null 值的判断，会导致数据库引擎放弃索引进行全表扫描

如下：

```
SELECT * FROM t WHERE score IS NULL
```

优化方式：可以给字段添加默认值 0，对 0 值进行判断。

如下：

```
SELECT * FROM t WHERE score = 0
```

⑤尽量避免在 where 条件中等号的左侧进行表达式、函数操作，会导致数据库引擎放弃索引进行全表扫描

可以将表达式、函数操作移动到等号右侧，如下：

```
-- 全表扫描
SELECT * FROM T WHERE score/10 = 9
-- 走索引
SELECT * FROM T WHERE score = 10*9
```

⑥当数据量大时，避免使用 where 1=1 的条件

通常为了方便拼装查询条件，我们会默认使用该条件，数据库引擎会放弃索引进行全表扫描。

如下：

```
SELECT username, age, sex FROM T WHERE 1=1
```

优化方式：用代码拼装 SQL 时进行判断，没 where 条件就去掉 where，有 where 条件就加 and。

⑦查询条件不能用 <> 或者 !=

使用索引列作为条件进行查询时，需要避免使用<>或者!=等判断条件。

如确实业务需要，使用到不等于符号，需要在重新评估索引建立，避免在此字段上建立索引，改由查询条件中其他索引字段代替。

⑧where 条件仅包含复合索引非前置列

如下：复合（联合）索引包含 key_part1，key_part2，key_part3 三列，但 SQL 语句没有包含索引前置列"key_part1"，按照 MySQL 联合索引的最左匹配原则，不会走联合索引。

```
select col1 from table where key_part2=1 and key_part3=2
```

⑨隐式类型转换造成不使用索引

如下 SQL 语句由于索引对列类型为 varchar，但给定的值为数值，涉及隐式类型转换，造成不能正确走索引。

```
select col1 from table where col_varchar=123;
```

⑩order by 条件要与 where 中条件一致，否则 order by 不会利用索引进行排序

如下：

```
-- 不走age索引
SELECT * FROM t order by age;

-- 走age索引
SELECT * FROM t where age > 0 order by age;
```

对于上面的语句，数据库的处理顺序是：

- **第一步：**根据 where 条件和统计信息生成执行计划，得到数据。
- **第二步：**将得到的数据排序。当执行处理数据（order by）时，数据库会先查看第一步的执行计划，看 order by 的字段是否在执行计划中利用了索引。如果是，则可以利用索引顺序而直接取得已经排好序的数据。如果不是，则重新进行排序操作。
- **第三步：**返回排序后的数据。

当 order by 中的字段出现在 where 条件中时，才会利用索引而不再二次排序，更准确的说，order by 中的字段在执行计划中利用了索引时，不用排序操作。

这个结论不仅对 order by 有效，对其他需要排序的操作也有效。比如 group by 、union 、distinct 等。

⑪正确使用 hint 优化语句

MySQL 中可以使用 hint 指定优化器在执行时选择或忽略特定的索引。

一般而言，处于版本变更带来的表结构索引变化，更建议避免使用 hint，而是通过 Analyze table 多收集统计信息。

但在特定场合下，指定 hint 可以排除其他索引干扰而指定更优的执行计划：

- USE INDEX 在你查询语句中表名的后面，添加 USE INDEX 来提供希望 MySQL 去参考的索引列表，就可以让 MySQL 不再考虑其他可用的索引。

  例子: SELECT col1 FROM table USE INDEX (mod_time, name)...
- IGNORE INDEX 如果只是单纯的想让 MySQL 忽略一个或者多个索引，可以使用 IGNORE INDEX 作为 Hint。

  例子: SELECT col1 FROM table IGNORE INDEX (priority) ...
- FORCE INDEX 为强制 MySQL 使用一个特定的索引，可在查询中使用FORCE INDEX 作为 Hint。

  例子: SELECT col1 FROM table FORCE INDEX (mod_time) ...

在查询的时候，数据库系统会自动分析查询语句，并选择一个最合适的索引。但是很多时候，数据库系统的查询优化器并不一定总是能使用最优索引。

如果我们知道如何选择索引，可以使用 FORCE INDEX 强制查询使用指定的索引。

例如：

```
SELECT * FROM students FORCE INDEX (idx_class_id) WHERE class_id = 1 ORDER BY id
DESC;
```

SELECT 语句其他优化

①避免出现 select *

首先，select * 操作在任何类型数据库中都不是一个好的 SQL 编写习惯。

使用 select * 取出全部列，会让优化器无法完成索引覆盖扫描这类优化，会影响优化器对执行计划的选择，也会增加网络带宽消耗，更会带来额外的 I/O，内存和 CPU 消耗。

建议提出业务实际需要的列数，将指定列名以取代 select *。具体详情见《为什么大家都说SELECT * 效率低》

②避免出现不确定结果的函数

特定针对主从复制这类业务场景。由于原理上从库复制的是主库执行的语句，使用如 now()、rand()、sysdate()、current_user() 等不确定结果的函数很容易导致主库与从库相应的数据不一致。

另外不确定值的函数，产生的 SQL 语句无法利用 query cache。

③多表关联查询时，小表在前，大表在后

在 MySQL 中，执行 from 后的表关联查询是从左往右执行的（Oracle 相反），第一张表会涉及到全表扫描。

所以将小表放在前面，先扫小表，扫描快效率较高，在扫描后面的大表，或许只扫描大表的前 100 行就符合返回条件并 return 了。

例如：表 1 有 50 条数据，表 2 有 30 亿条数据；如果全表扫描表 2，你品，那就先去吃个饭再说吧是吧。

④使用表的别名

当在 SQL 语句中连接多个表时，请使用表的别名并把别名前缀于每个列名上。这样就可以减少解析的时间并减少哪些友列名歧义引起的语法错误。

⑤用 where 字句替换 HAVING 字句

避免使用 HAVING 字句，因为 HAVING 只会在检索出所有记录之后才对结果集进行过滤，而 where 则是在聚合前刷选记录，如果能通过 where 字句限制记录的数目，那就能减少这方面的开销。

HAVING 中的条件一般用于聚合函数的过滤，除此之外，应该将条件写在 where 字句中。

where 和 having 的区别：where 后面不能使用组函数。

⑥调整 Where 字句中的连接顺序

MySQL 采用从左往右，自上而下的顺序解析 where 子句。根据这个原理，应将过滤数据多的条件往前放，最快速度缩小结果集。

增删改 DML 语句优化

①大批量插入数据

如果同时执行大量的插入，建议使用多个值的 INSERT 语句（方法二）。这比使用分开 INSERT 语句快（方法一），一般情况下批量插入效率有几倍的差别。

方法一：

```
insert into T values(1,2);

insert into T values(1,3);

insert into T values(1,4);
```

方法二：

```
Insert into T values(1,2),(1,3),(1,4);
```

选择后一种方法的原因有三：

- 减少 SQL 语句解析的操作，MySQL 没有类似 Oracle 的 share pool，采用方法二，只需要解析一次就能进行数据的插入操作。
- 在特定场景可以减少对 DB 连接次数。
- SQL 语句较短，可以减少网络传输的 IO。

②适当使用 commit

适当使用 commit 可以释放事务占用的资源而减少消耗，commit 后能释放的资源如下：

- 事务占用的 undo 数据块。
- 事务在 redo log 中记录的数据块。
- 释放事务施加的，减少锁争用影响性能。特别是在需要使用 delete 删除大量数据的时候，必须分解删除量并定期 commit。

③避免重复查询更新的数据

针对业务中经常出现的更新行同时又希望获得改行信息的需求，MySQL 并不支持 PostgreSQL 那样的 UPDATE RETURNING 语法，在 MySQL 中可以通过变量实现。

例如，更新一行记录的时间戳，同时希望查询当前记录中存放的时间戳是什么？

简单方法实现：

```
Update t1 set time=now() where col1=1;

Select time from t1 where id =1;
```

使用变量，可以重写为以下方式：

```
Update t1 set time=now () where col1=1 and @now: = now ();

Select @now;
```

前后二者都需要两次网络来回，但使用变量避免了再次访问数据表，特别是当 t1 表数据量较大时，后者比前者快很多。

④查询优先还是更新（insert、update、delete）优先

MySQL 还允许改变语句调度的优先级，它可以使来自多个客户端的查询更好地协作，这样单个客户端就不会由于锁定而等待很长时间。改变优先级还可以确保特定类型的查询被处理得更快。

我们首先应该确定应用的类型，判断应用是以查询为主还是以更新为主的，是确保查询效率还是确保更新的效率，决定是查询优先还是更新优先。

下面我们提到的改变调度策略的方法主要是针对只存在表锁的存储引擎，比如 MyISAM 、MEMROY、MERGE，对于 Innodb 存储引擎，语句的执行是由获得行锁的顺序决定的。

MySQL 的默认的调度策略可用总结如下：

- 写入操作优先于读取操作。
- 对某张数据表的写入操作某一时刻只能发生一次，写入请求按照它们到达的次序来处理。
- 对某张数据表的多个读取操作可以同时地进行。

MySQL 提供了几个语句调节符，允许你修改它的调度策略：

- LOW_PRIORITY 关键字应用于 DELETE、INSERT、LOAD DATA、REPLACE 和 UPDATE。
- HIGH_PRIORITY 关键字应用于 SELECT 和 INSERT 语句。
- DELAYED 关键字应用于 INSERT 和 REPLACE 语句。

如果写入操作是一个 LOW_PRIORITY（低优先级）请求，那么系统就不会认为它的优先级高于读取操作。

在这种情况下，如果写入者在等待的时候，第二个读取者到达了，那么就允许第二个读取者插到写入者之前。

只有在没有其它的读取者的时候，才允许写入者开始操作。这种调度修改可能存在 LOW_PRIORITY 写入操作永远被阻塞的情况。

SELECT 查询的 HIGH_PRIORITY（高优先级）关键字也类似。它允许 SELECT 插入正在等待的写入操作之前，即使在正常情况下写入操作的优先级更高。

另外一种影响是，高优先级的 SELECT 在正常的 SELECT 语句之前执行，因为这些语句会被写入操作阻塞。

如果希望所有支持 LOW_PRIORITY 选项的语句都默认地按照低优先级来处理，那么请使用--low-priority-updates 选项来启动服务器。

通过使用 INSERTHIGH_PRIORITY 来把 INSERT 语句提高到正常的写入优先级，可以消除该选项对单个 INSERT 语句的影响。

查询条件优化

①对于复杂的查询，可以使用中间临时表暂存数据

②优化 group by 语句

默认情况下，MySQL 会对 GROUP BY 分组的所有值进行排序，如 “GROUP BY col1，col2，….;” 查询的方法如同在查询中指定 “ORDER BY col1，col2，…;” 。

如果显式包括一个包含相同的列的 ORDER BY 子句，MySQL 可以毫不减速地对它进行优化，尽管仍然进行排序。

因此，如果查询包括 GROUP BY 但你并不想对分组的值进行排序，你可以指定 ORDER BY NULL 禁止排序。

例如：

```
SELECT col1, col2, COUNT(*) FROM table GROUP BY col1, col2 ORDER BY NULL ;
```

③优化 join 语句

MySQL 中可以通过子查询来使用 SELECT 语句来创建一个单列的查询结果，然后把这个结果作为过滤条件用在另一个查询中。

使用子查询可以一次性的完成很多逻辑上需要多个步骤才能完成的 SQL 操作，同时也可以避免事务或者表锁死，并且写起来也很容易。但是，有些情况下，子查询可以被更有效率的连接（JOIN）..替代。

例子：假设要将所有没有订单记录的用户取出来，可以用下面这个查询完成：

```
SELECT col1 FROM customerinfo WHERE CustomerID NOT in (SELECT CustomerID FROM
salesinfo )
```

如果使用连接（JOIN）..来完成这个查询工作，速度将会有所提升。

尤其是当 salesinfo 表中对 CustomerID 建有索引的话，性能将会更好，查询如下：

```
SELECT col1 FROM customerinfo
   LEFT JOIN salesinfoON customerinfo.CustomerID=salesinfo.CustomerID
      WHERE salesinfo.CustomerID IS NULL
```

连接（JOIN）..之所以更有效率一些，是因为 MySQL 不需要在内存中创建临时表来完成这个逻辑上的需要两个步骤的查询工作。

④优化 union 查询

MySQL 通过创建并填充临时表的方式来执行 union 查询。除非确实要消除重复的行，否则建议使用 union all。

原因在于如果没有 all 这个关键词，MySQL 会给临时表加上 distinct 选项，这会导致对整个临时表的数据做唯一性校验，这样做的消耗相当高。

高效：

```
SELECT COL1, COL2, COL3 FROM TABLE WHERE COL1 = 10

UNION ALL

SELECT COL1, COL2, COL3 FROM TABLE WHERE COL3= 'TEST';
```

低效：

```
SELECT COL1, COL2, COL3 FROM TABLE WHERE COL1 = 10

UNION

SELECT COL1, COL2, COL3 FROM TABLE WHERE COL3= 'TEST';
```

⑤拆分复杂 SQL 为多个小 SQL，避免大事务

如下：

- 简单的 SQL 容易使用到 MySQL 的 QUERY CACHE。
- 减少锁表时间特别是使用 MyISAM 存储引擎的表。
- 可以使用多核 CPU。

⑥使用 truncate 代替 delete

当删除全表中记录时，使用 delete 语句的操作会被记录到 undo 块中，删除记录也记录 binlog。

当确认需要删除全表时，会产生很大量的 binlog 并占用大量的 undo 数据块，此时既没有很好的效率也占用了大量的资源。

使用 truncate 替代，不会记录可恢复的信息，数据不能被恢复。也因此使用 truncate 操作有其极少的资源占用与极快的时间。另外，使用 truncate 可以回收表的水位，使自增字段值归零。

⑦使用合理的分页方式以提高分页效率

使用合理的分页方式以提高分页效率 针对展现等分页需求，合适的分页方式能够提高分页的效率。

案例 1：

```
select * from t where thread_id = 10000 and deleted = 0
   order by gmt_create asc limit 0, 15;
```

上述例子通过一次性根据过滤条件取出所有字段进行排序返回。数据访问开销=索引 IO+索引全部记录结果对应的表数据 IO。

因此，该种写法越翻到后面执行效率越差，时间越长，尤其表数据量很大的时候。

适用场景：当中间结果集很小（10000 行以下）或者查询条件复杂（指涉及多个不同查询字段或者多表连接）时适用。

案例 2：

```
select t.* from (select id from t where thread_id = 10000 and deleted = 0
   order by gmt_create asc limit 0, 15) a, t
      where a.id = t.id;
```

上述例子必须满足 t 表主键是 id 列，且有覆盖索引 secondary key：（thread_id, deleted, gmt_create）。

通过先根据过滤条件利用覆盖索引取出主键 id 进行排序，再进行 join 操作取出其他字段。

数据访问开销=索引 IO+索引分页后结果（例子中是 15 行）对应的表数据 IO。因此，该写法每次翻页消耗的资源和时间都基本相同，就像翻第一页一样。

适用场景：当查询和排序字段（即 where 子句和 order by 子句涉及的字段）有对应覆盖索引时，且中间结果集很大的情况时适用。

建表优化

①在表中建立索引，优先考虑 where、order by 使用到的字段。

②尽量使用数字型字段（如性别，男：1 女：2），若只含数值信息的字段尽量不要设计为字符型，这会降低查询和连接的性能，并会增加存储开销。

这是因为引擎在处理查询和连接时会 逐个比较字符串中每一个字符，而对于数字型而言只需要比较一次就够了。

③查询数据量大的表 会造成查询缓慢。主要的原因是扫描行数过多。这个时候可以通过程序，分段分页进行查询，循环遍历，将结果合并处理进行展示。

要查询 100000 到 100050 的数据，如下：

```
SELECT * FROM (SELECT ROW_NUMBER() OVER(ORDER BY ID ASC) AS rowid,*
   FROM infoTab)t WHERE t.rowid > 100000 AND t.rowid <= 100050
```

④用 varchar/nvarchar 代替 char/nchar。

尽可能的使用 varchar/nvarchar 代替 char/nchar ，因为首先变长字段存储空间小，可以节省存储空间，其次对于查询来说，在一个相对较小的字段内搜索效率显然要高些。

不要以为 NULL 不需要空间，比如：char(100) 型，在字段建立时，空间就固定了， 不管是否插入值（NULL 也包含在内），都是占用 100 个字符的空间的，如果是 varchar 这样的变长字段， null 不占用空间。

*作者：*_陈哈哈**

编辑：陶家龙

出处：https://sohu.gg/FGG98i