Java程序员必知的9个SQL优化技巧

大多数的接口性能问题，很多情况下都是SQL问题，在工作中，我们也会定期对慢SQL进行优化，以提高接口性能。这里总结一下常见的优化方向和策略。

避免使用select *，减少查询字段

不要为了图省事，直接查询全部的字段，尽量查需要的字段，特别是复杂的SQL，能够避免很多不走索引的情况。这也是最基本的方法。

检查执行计划，是否走索引

检查where和order by字段是否有索引，根据表的数据量和现有索引，考虑是否增加索引或者联合索引。 然而，索引并不是越多越好，原因有以下几点：

0. 存储空间：每个索引都会占用额外的存储空间。如果为表中的每一列都创建索引，那么这些索引的存储开销可能会非常大，尤其是在大数据集上。
1. 索引重建增加开销：当数据发生变更（如插入、更新或删除）时，相关的索引也需要进行更新，以确保数据的准确性和查询效率。这意味着更多的索引会导致更慢的写操作。
2. 选择性：选择性是指索引列中不同值的数量与表中记录数的比率。选择性高的列（即列中有很多唯一的值）更适合创建索引。对于选择性低的列（如性别列，其中只有“男”和“女”两个值），创建索引可能不会产生太大的查询性能提升。
3. 过度索引：当表中存在过多的索引时，可能会导致数据库优化器在选择使用哪个索引时变得困难。这可能会导致查询性能下降，因为优化器可能选择了不是最优的索引。

因此，在设计数据库时，需要根据查询需求和数据变更模式来仔细选择需要创建索引的列。通常建议只为经常用于查询条件、排序和连接的列创建索引，并避免为选择性低的列创建索引。

避免使用or连接

假设我们有一个数据表employee，包含以下字段：id, name, age。 原始查询使用OR操作符来筛选满足name为’John’或age为30的员工：

```
SELECT * FROM employee WHERE name = ’John’ OR age = 30;
```

使用UNION操作符来实现同样的筛选：

```
SELECT * FROM employee WHERE name = ’John’
UNION
SELECT * FROM employee WHERE age = 30;
```

UNION操作符先查询满足name为’John’的记录，然后查询满足age为30的记录，并将两个结果集合并起来。这样可以减少查询的数据量，提高查询效率。 需要注意的是，**UNION操作符会去除重复的记录**。**如果想要保留重复的记录，可以使用UNION ALL操作符**，例如： 判断两条记录是否为重复记录的标准是通过比较每个字段的值来确定的。

```
SELECT * FROM employee WHERE name = ’John’
UNION ALL
SELECT * FROM employee WHERE age = 30;
```

在使用UNION代替OR时，还需要注意查询语句的语义是否与原始查询相同。有些情况下，OR可能会产生更准确的结果，因此在使用UNION时需谨慎考虑语义问题。

减少in和not in的使用
==========

说实话，这种情况有点难。实际工作中，使用in的场景很多，但是要尽量避免

in后面的数据范围，范围太大的时候，要考虑分批处理等操作。

对于连续的数值，可以考虑使用between and 代替。

避免使用左模糊查询
==========

在工作中，对于姓名、手机号、名称等内容，经常会遇到模糊查询的场景，但是要尽量避免左模糊，这种SQL无法使用索引。

0. 左模糊查询： 假设我们有一个数据表customer，包含字段name，我们想要查询名字以”J”开头的客户：

```
SELECT * FROM customer WHERE name LIKE 'J%';
```

2. 右模糊查询： 继续使用上述customer表，我们想要查询名字以”n”结尾的客户：

```
SELECT * FROM customer WHERE name LIKE '%n';
```

注意，在某些数据库中，对于右模糊查询，可能需要使用转义符号（如””），以防止通配符被误解。

3. 全模糊查询： 还是使用上述customer表，我们想要查询名字中包含”son”的客户：

```
SELECT * FROM customer WHERE name LIKE '%son%';
```

连接查询join替代子查询
==========

假设我们有两个表：订单表（orders）和客户表（customers）。 订单表包含了订单号（order_id）、客户ID（customer_id）和订单金额（amount），而客户表包含了客户ID（customer_id）和客户姓名（customer_name）。

我们要找出所有订单金额大于1000美元的客户姓名：

```
SELECT customer_name
FROM customers
WHERE customer_id IN (SELECT DISTINCT customer_id FROM orders
WHERE amount > 1000);
```

以上查询首先在订单表中挑选出所有金额大于1000美元的客户ID，然后使用这个子查询的结果来过滤客户表并获取客户姓名。

使用 JOIN 来替代子查询的方式：

```
SELECT DISTINCT c.customer_name
FROM customers c
JOIN orders o ON c.customer_id = o.customer_id
WHERE o.amount > 1000;
```

改造后的查询通过使用 INNER JOIN 将客户表和订单表连接在一起，然后使用 WHERE 子句来过滤出金额大于1000美元的订单。

这种改造不仅使查询更加简洁，而且可能还会提高查询的性能。JOIN 操作通常比子查询的效率更高，特别是在处理大型数据集时。

join的优化
=======

JOIN 是 SQL 查询中的一个操作，用于将两个或多个表连接在一起。JOIN 操作有几种类型，包括 LEFT JOIN、RIGHT JOIN 和 INNER JOIN。要选用正确的关联方式，确保查询内容的正确性。

0. INNER JOIN（内连接）：内连接返回满足连接条件的行，即两个表中相关联的行组合。只有在两个表中都存在匹配的行时，才会返回结果。

```
SELECT *
FROM table1
INNER JOIN table2 ON table1.column = table2.column;
```

2. LEFT JOIN（左连接）：左连接返回左侧表中的所有行，以及右侧表中满足连接条件的行。如果右表中没有匹配的行，则返回 NULL 值。在用left join关联查询时，左边要用小表，右边可以用大表。如果能用inner join的地方，尽量少用left join。

```
SELECT *
FROM table1
LEFT JOIN table2 ON table1.column = table2.column;
```

3. RIGHT JOIN（右连接）：右连接返回右侧表中的所有行，以及左侧表中满足连接条件的行。如果左表中没有匹配的行，则返回 NULL 值。

```
SELECT *
FROM table1
RIGHT JOIN table2 ON table1.column = table2.column;
```

需要注意的是，LEFT JOIN 和 RIGHT JOIN 是对称的，只是左右表的位置不同。INNER JOIN 则是返回共同匹配的行。

这些不同类型的 JOIN 可以灵活地根据查询需求选择使用。INNER JOIN 用于获取两个表中的匹配行，LEFT JOIN 和 RIGHT JOIN 用于获取一个表中的所有行以及另一个表中的匹配行。使用 JOIN 可以将多个表连接在一起，使我们能够根据关联的列获取相关的数据，并更有效地处理复杂的查询需求。但是使用的时候要特别注意，左右表的关联关系，是一对一、一对多还是多对多，对查询的结果影响很大。

group by 字段优化
==========

假设我们要计算每个客户的订单总金额，原始的查询可能如下所示：

```
SELECT customer_id, SUM(amount) AS total_amount
FROM orders
GROUP BY customer_id;
```

在这个查询中，我们使用 GROUP BY 字段 customer_id 对订单进行分组，并使用 SUM 函数计算每个客户的订单总金额。

为了优化这个查询，我们可以考虑以下几种方法：

0. 索引优化：

* 确保在 customer_id 字段上创建索引，以加速 GROUP BY 和 WHERE 子句的执行。
* 如果查询还包含其他需要的字段，可以考虑创建聚簇索引，将相关的字段放在同一个索引中，以减少查询的IO操作。

1. 使用覆盖索引：

* 如果查询中只需要使用 customer_id 和 amount 两个字段，可以创建一个覆盖索引，它包含了这两个字段，减少了查找其他字段的开销。

2. 子查询优化：

* 如果订单表很大，可以先使用子查询将数据限制在一个较小的子集上，然后再进行 GROUP BY 操作。例如，可以先筛选出最近一段时间的订单，然后再对

这些订单进行分组。
3. 条件优化：

* 使用WHERE条件在分组前，就把多余的数据过滤掉了，这样分组时效率就会更高一些。而不是在分组后使用having过滤数据。

深分页limit优化
==========

深分页通常指的是在处理大量数据时，用户需要浏览远离首页的页面，例如第100页、第1000页等。这种场景下，如果简单地一次性加载所有数据并进行分页，会导致性能问题，包括内存消耗、数据库查询效率等。

我们日常使用较多的分页一般是用的PageHelper插件，SQL如下：

```
select id,name from table_name where N个条件 limit 100000,10;
```

它的执行流程：

0. 先去二级索引过滤数据，然后找到主键ID
1. 通过ID回表查询数据，取出需要的列
2. 扫描满足条件的100010，丢弃前面100000条，返回

这里很明显的不足就是只需要拿10条，但是却多回表了100000次。

可采用的策略：主要是使用子查询、关联查询、范围查询和标签记录法这四种方法，当然对于深分页问题，一般都是比较麻烦了，都需要采用标签记录法来改造代码。

标签记录法：就是记录上次查询的最大ID，再请求下一页的时候带上，从上次的下一条数据开始开始，前提是有序的。 主要需要对代码进行改造：

```
```

```
public Page<Item> fetchPageByKey(Long lastKey, int pageSize) {
    // lastKey是上一页最后一项的主键
    // 查询数据库，获取主键大于lastKey的pageSize条记录
    List<Item> items =
itemRepository.findByPrimaryKeyGreaterThan(lastKey, pageSize);
    // 如果没有更多数据，可以设置下一个lastKey为空或特定值（如-1）
    Long nextLastKey = items.isEmpty() ? null : items.get(items.size() -
1).getId();
    return new Page<>(items, nextLastKey);
}

```
原文链接: https://juejin.cn/post/7368377525859008522