**SQL优化的35个小技巧**

**前言**

**1. 避免使用 SELECT \*，使用具体字段**

**反例：**

SELECT \* FROM employee;

**正例：**

SELECT id, name, age FROM employee;

使用具体字段可以节省资源、减少网络开销，且能避免回表查询。

**2. 避免在 WHERE 子句中使用 OR**

**反例：**

SELECT \* FROM user WHERE userid=1 OR age=18;

**正例：**

-- 使用 UNION ALL SELECT \* FROM user WHERE userid=1 UNION ALL SELECT \* FROM user WHERE age=18;

**原因**：OR 会导致索引失效并引发全表扫描。

**3. 使用 LIMIT 避免不必要的数据返回**

**反例：**

SELECT id, order\_date FROM order\_tab WHERE user\_id=666 ORDER BY create\_date DESC;

**正例：**

SELECT id, order\_date FROM order\_tab WHERE user\_id=666 ORDER BY create\_date DESC LIMIT 1;

LIMIT 提升查询效率，避免多余的数据返回。

**4. 使用数值类型代替字符串**

**例子**：性别字段建议用数值（如0代表女生，1代表男生）而非字符串（如"WOMEN"、"MAN"）。

**原因**：数值类型占用存储空间小、比较速度更快。

**5. 批量操作（插入、删除、查询）**

**反例：**

for(User u : list) { INSERT INTO user(name, age) VALUES(#name#, #age#); }

**正例：**

INSERT INTO user(name, age) VALUES <foreach collection="list" item="item" index="index" separator=","> (#{item.name}, #{item.age}) </foreach>

**原因**：批量插入性能更优。

**6. 使用 UNION ALL 替换 UNION（无重复记录时）**

**反例：**

SELECT \* FROM user WHERE userid=1 UNION SELECT \* FROM user WHERE age=10;

**正例：**

SELECT \* FROM user WHERE userid=1 UNION ALL SELECT \* FROM user WHERE age=10;

**原因**：UNION 会排序和合并，UNION ALL 则省去这一步。

**7. 尽可能使用 NOT NULL 定义字段**

**原因**：NOT NULL

* 可以防止出现空指针问题。
* NULL值存储也需要额外的空间的，它也会导致比较运算更为复杂，使优化器难以优化SQL。
* NULL值有可能会导致索引失效

**8. 避免在索引列上使用内置函数**

**反例：**

SELECT userId, loginTime FROM loginuser WHERE DATE\_ADD(loginTime, INTERVAL 7 DAY) >= NOW();

**正例：**

SELECT userId, loginTime FROM loginuser WHERE loginTime >= DATE\_ADD(NOW(), INTERVAL -7 DAY);

**原因**：索引列上使用函数会导致索引失效。

**9. 避免在 WHERE 子句中对字段进行表达式操作**

**反例：**

SELECT \* FROM user WHERE age - 1 = 10;

**正例：**

SELECT \* FROM user WHERE age = 11;

**10. 在 GROUP BY 前进行条件过滤**

**反例：**

SELECT user\_id, SUM(amount) AS total\_amount FROM orders GROUP BY user\_id HAVING city = '北京';

**正例：**

SELECT user\_id, SUM(amount) AS total\_amount FROM orders WHERE city = '北京' GROUP BY user\_id;

**11. 优化 LIKE 语句**

**反例：**

SELECT userId, name FROM user WHERE userId LIKE '%123';

**正例：**

SELECT userId, name FROM user WHERE userId LIKE '123%';

**原因**：% 放在前面会导致索引失效。

**12. 使用小表驱动大表**

**小表先执行以减少扫描量，如使用 EXISTS 或 IN 进行过滤。**

假设我们有个客户表和一个订单表。其中订单表有10万记录，客户表只有1000行记录。现在要查询下单过的客户信息，可以这样写：

SELECT \* FROM customers c WHERE EXISTS (     SELECT 1 FROM orders o WHERE o.customer\_id = c.id );

EXISTS 会逐行扫描 customers 表（即小表），对每一行 c.id，在 orders 表（大表）中检查是否有 customer\_id = c.id 的记录。当然，也可以使用in实现：

SELECT \* FROM customers WHERE id IN (     SELECT customer\_id FROM orders );

in 查询会先执行内部查询部分 SELECT customer\_id FROM orders，获得 orders 表（大表）中的所有 customer\_id，然后在 customers 表（小表）中查找匹配的 id。因为orders表的数据量比较大，因此这里用exists效果会相对更好一点。

**13. IN 查询的元素不宜太多**

如果使用了in，即使后面的条件加了索引，还是要注意in后面的元素不要过多哈。 in元素一般建议不要超过200个，如果超过了，建议分组，每次200一组进行哈。

**反例：**

select user\_id,name from user where user\_id in (1,2,3...1000000);

如果我们对in的条件不做任何限制的话，该查询语句一次性可能会查询出非常多的数据，很容易导致接口超时。尤其有时候，我们是用的子查询，in后面的子查询，你都不知道数量有多少那种，更容易采坑.

如下这种子查询：

select \* from user where user\_id in (select author\_id from artilce where type = 1);

正例是，分批进行，比如每批200个：

select user\_id,name from user where user\_id in (1,2,3...200);

**14. 优化 LIMIT 分页**

**避免深分页，使用“标签记录法”或“延迟关联法”提升性能。**

我们日常做分页需求时，一般会用 limit 实现，但是当偏移量特别大的时候，查询效率就变得低下，也就是出现深分页问题。

**反例：**

select id,name,balance from account where create\_time> '2020-09-19' limit 100000,10;

我们可以通过减少回表次数来优化。一般有标签记录法和延迟关联法。

标签记录法就是标记一下上次查询到哪一条了，下次再来查的时候，从该条开始往下扫描。就好像看书一样，上次看到哪里了，你就折叠一下或者夹个书签，下次来看的时候，直接就翻到啦。

假设上一次记录到100000，则SQL可以修改为：

select  id,name,balance FROM account where id > 100000 limit 10;

这样的话，后面无论翻多少页，性能都会不错的，因为命中了id索引。但是这种方式有局限性：需要一种类似连续自增的字段。延迟关联法延迟关联法，就是把条件转移到主键索引树，然后减少回表。

如下：

select  acct1.id,acct1.name,acct1.balance FROM account acct1 INNER JOIN (SELECT a.id FROM account a WHERE a.create\_time > '2020-09-19' limit 100000, 10) AS acct2 on acct1.id= acct2.id;

优化思路就是，先通过idx\_create\_time二级索引树查询到满足条件的主键ID，再与原表通过主键ID内连接，这样后面直接走了主键索引了，同时也减少了回表。

**15. 优先使用连接查询而非子查询**

因为使用子查询，可能会创建临时表。

**反例：**

SELECT \* FROM customers WHERE id IN (SELECT customer\_id FROM orders);

IN 子查询会在 orders 表中查询所有 customer\_id，并生成一个临时结果集。

**正例：**

SELECT DISTINCT c.\* FROM customers c JOIN orders o ON c.id = o.customer\_id;

通过 JOIN 直接将 customers 和 orders 表关联，符合条件的记录一次性筛选完成。      MySQL 优化器通常可以利用索引来加速 JOIN，避免了临时表的创建，查询效果就更佳

**16. Inner join 、left join、right join，优先使用Inner join，如果是left join，左边表结果尽量小**

如需 LEFT JOIN，左表数据结果尽量小。

* Inner join 内连接，在两张表进行连接查询时，只保留两张表中完全匹配的结果集
* left join 在两张表进行连接查询时，会返回左表所有的行，即使在右表中没有匹配的记录。
* right join 在两张表进行连接查询时，会返回右表所有的行，即使在左表中没有匹配的记录。

都满足SQL需求的前提下，推荐优先使用Inner join（内连接），如果要使用left join，左边表数据结果尽量小，如果有条件的尽量放到左边处理。

**反例:** select \* from tab1 t1 left join tab2 t2  on t1.size = t2.size where t1.id>2;

**正例：** select \* from (select \* from tab1 where id >2) t1 left join tab2 t2 on t1.size = t2.size;

理由：如果inner join是等值连接，或许返回的行数比较少，所以性能相对会好一点。同理，使用了左连接，左边表数据结果尽量小，条件尽量放到左边处理，意味着返回的行数可能比较少。

**17. 避免 != 或 <> 操作符**

**反例：**

SELECT age, name FROM user WHERE age <> 18;

**正例**：可分为两个查询。

select age,name  from user where age <18; select age,name  from user where age >18;

**使用!=和<>很可能会让索引失效**

**18. 使用联合索引时遵循最左匹配原则**

例如联合索引 (userId, age)，查询 userId 和 age 时优先使用 userId。

表结构：（有一个联合索引idx\_userid\_age，userId在前，age在后）

CREATE TABLE `user` (   `id` int(11) NOT NULL AUTO\_INCREMENT,   `userId` int(11) NOT NULL,   `age` int(11) DEFAULT NULL,   `name` varchar(255) NOT NULL,   PRIMARY KEY (`id`),   KEY `idx\_userid\_age` (`userId`,`age`) USING BTREE ) ENGINE=InnoDB AUTO\_INCREMENT=2 DEFAULT CHARSET=utf8;

反例：

select \* from user where age = 10;

**正例**：//符合最左匹配原则

select \* from user where userid=10 and age =10;

//符合最左匹配原则

select \* from user where userid =10;

理由：当我们创建一个联合索引的时候，如(k1,k2,k3)，相当于创建了（k1）、(k1,k2)和(k1,k2,k3)三个索引，这就是最左匹配原则。联合索引不满足最左原则，索引一般会失效，但是这个还跟Mysql优化器有关的。

**19. 对 WHERE 和 ORDER BY 涉及的列建索引**

**反例：**

SELECT \* FROM user WHERE address = '深圳' ORDER BY age;

**正例：**      覆盖索引能够使得你的SQL语句不需要回表，仅仅访问索引就能够得到所有需要的数据，大大提高了查询效率。

ALTER TABLE user ADD INDEX idx\_address\_age (address, age);

**20. 使用覆盖索引**

**正例：**

SELECT id, name FROM user WHERE userid LIKE '123%';

**21. 删除冗余索引**

**避免重复索引，节省资源。**

**反例：**

KEY `idx\_userId` (`userId`)   KEY `idx\_userId\_age` (`userId`,`age`)

**正例:**

//删除userId索引，因为组合索引（A，B）相当于创建了（A）和（A，B）索引   KEY `idx\_userId\_age` (`userId`,`age`)

理由：重复的索引需要维护，并且优化器在优化查询的时候也需要逐个地进行考虑，这会影响性能的。

**22. 避免超过3个以上的表连接**

* 不要有超过3个以上的表连接连表越多，编译的时间和开销也就越大。
* 把连接表拆开成较小的几个执行，可读性更高。
* 如果一定需要连接很多表才能得到数据，那么意味着糟糕的设计了。

**23. 索引数不宜超过5个**

* 索引不宜太多，一般5个以内。

索引并不是越多越好，索引虽然提高了查询的效率，但是也降低了插入和更新的效率。

* insert或update时有可能会重建索引，所以建索引需要慎重考虑，视具体情况来定。
* 一个表的索引数最好不要超过5个，若太多需要考虑一些索引是否没有存在的必要。

**24. 索引不适合建立在大量重复数据的字段上**

如性别字段，重复数据多时优化器可能放弃索引。

**25. 字符串类型字段在 WHERE 中使用引号**

**反例：**

SELECT \* FROM user WHERE userid = 123;

**正例：**

SELECT \* FROM user WHERE userid = '123';

**26. 避免返回过多数据量**

**反例：**

SELECT \* FROM LivingInfo WHERE watchId = userId AND watchTime >= DATE\_SUB(NOW(), INTERVAL 1 YEAR);

**正例：**

-- 分页查询 SELECT \* FROM LivingInfo WHERE watchId = userId AND watchTime >= DATE\_SUB(NOW(), INTERVAL 1 YEAR) LIMIT offset, pageSize;

**理由**：

查询效率：当返回的数据量过大时，查询所需的时间会显著增加，导致数据库性能下降。

通过限制返回的数据量，可以缩短查询时间，提高数据库响应速度。

网络传输：大量数据的传输会占用网络带宽，可能导致网络拥堵和延迟。

减少返回的数据量可以降低网络传输的负担，提高数据传输效率。。

在 SQL 优化方面，除了已经列举的26个技巧，这里再补充9个技巧，使优化点达到35条。这些补充技巧包含一些更加细化的实践，帮助进一步提升 SQL 查询的效率：

**27. 合理利用视图（View）进行复杂查询**

如果一个复杂查询需要频繁使用，可以考虑创建视图，以简化查询结构并提高查询效率。

**正例：**

CREATE VIEW view\_user\_orders AS SELECT u.id, u.name, o.order\_id, o.amount FROM user u JOIN orders o ON u.id = o.user\_id; -- 使用视图查询 SELECT \* FROM view\_user\_orders WHERE amount > 100;

**28. 使用表分区（Partitioning）优化大表性能**

对于数据量较大的表，通过分区可以有效提升查询效率。表分区可以按日期、数值范围等方式进行分割。

**正例：**

CREATE TABLE sales ( sale\_id INT, sale\_date DATE, amount DECIMAL(10,2) ) PARTITION BY RANGE (YEAR(sale\_date)) ( PARTITION p2019 VALUES LESS THAN (2020), PARTITION p2020 VALUES LESS THAN (2021), PARTITION p2021 VALUES LESS THAN (2022) );

**29. 合理使用存储过程（Stored Procedure）来减少多次 SQL 交互**

将多步操作写入存储过程，可以减少客户端与数据库之间的多次交互，提高效率。

**正例：**

CREATE PROCEDURE update\_and\_select(IN user\_id INT) BEGIN UPDATE users SET last\_login = NOW() WHERE id = user\_id; SELECT \* FROM users WHERE id = user\_id; END;

**30. 对频繁变更的数据使用缓存**

对于频繁查询的静态或相对稳定的数据，可考虑将查询结果存放到缓存（如 Redis）中，以减轻数据库的负担。

**31. 使用适当的隔离级别**

在高并发环境中选择适当的事务隔离级别（如 READ COMMITTED），可以避免不必要的锁竞争和阻塞，提升并发效率。

**32. 使用合适的数据类型**

选择合适的数据类型会节省存储空间，提升处理速度。例如：TINYINT（1字节）代替INT（4字节），VARCHAR(50)代替CHAR(50)，存储长度尽可能精确匹配业务需求。

**33. 避免频繁更新索引列**

在高并发写操作的场景中，频繁更新索引字段会导致索引重建，影响性能。如果字段变动频繁且无查询需求，建议避免对该字段建立索引。

**34. 避免在事务中执行非必要的操作**

在事务中应避免执行耗时操作，比如网络请求或复杂计算，以减少锁的持有时间。优先确保事务操作集中在必要的数据变更上。

**35. 使用批量更新或删除**

对于批量更新或删除数据，避免一次性操作大量记录。可以分批次执行，以减少锁定时间，减轻系统压力。

**正例：**

-- 分批删除 DELETE FROM orders WHERE status = 'obsolete' LIMIT 1000;