SQL 性能优化实战案例

一、电商订单查询优化：从 2.4 秒到 0.02 秒的飞跃

业务场景：某电商平台需查询用户特定日期内的订单列表，原始 SQL 执行时间高达 2.4 秒，严重影响用户体验。

原始 SQL：

```
SELECT *
FROM orders
WHERE user_id = 10086
  AND DATE(create_time) = '2023-10-01'
  AND status IN (1,2,3)
ORDER BY create_time DESC
LIMIT 0,10;
```

问题诊断：

1. **全表扫描**：通过 EXPLAIN 发现 type=ALL，扫描行数达 999,876 行。
2. **隐式类型转换**：DATE(create_time)导致索引失效，需改写为范围查询。
3. **回表开销**：SELECT *需多次访问数据行，未使用覆盖索引。

分步优化方案：

1. **索引重构**：

```
ALTER TABLE orders
ADD INDEX idx_user_status_time (user_id, status, create_time);
```

联合索引覆盖 WHERE 条件和排序字段，避免文件排序。

2. **查询逻辑改写**：

```
SELECT id, user_id, amount, create_time
FROM orders
WHERE user_id = 10086
  AND create_time BETWEEN '2023-10-01 00:00:00'
                      AND '2023-10-01 23:59:59'
```

```
  AND status IN (1,2,3)
ORDER BY create_time DESC
LIMIT 0,10;
```

直接使用 BETWEEN 替代 DATE()函数，避免索引失效。

仅选择必要字段，利用覆盖索引减少回表。

3. **执行计划验证：**

```
EXPLAIN
SELECT id, user_id, amount, create_time
FROM orders
WHERE user_id = 10086
  AND create_time BETWEEN '2023-10-01 00:00:00'
                      AND '2023-10-01 23:59:59'
  AND status IN (1,2,3)
ORDER BY create_time DESC
LIMIT 0,10;
```

输出结果：type=ref，扫描行数降至 12 行，Extra=Using index（覆盖索引）。

优化效果：

执行时间：从 2.4 秒降至 0.02 秒，性能提升 120 倍。

资源消耗：扫描行数减少 99.99%，CPU 和内存占用降低 80% 以上。

二、多表 JOIN 优化：三表关联性能提升 20 倍

业务场景：统计购买过某商品的用户信息，涉及 users、orders、order_items 三表关联，原始查询耗时 1.8 秒。

原始 SQL：

```
SELECT DISTINCT u.*
FROM users u
JOIN orders o ON u.id = o.user_id
JOIN order_items oi ON o.id = oi.order_id
WHERE oi.product_id = 123;
```

问题诊断：

1. **索引缺失：**order_items 表的 order_id 和 product_id 未建立联合索引。
2. **大表驱动：**users 表（10 万行）作为驱动表，导致多次扫描大表 orders（100 万行）。

优化方案：

1. **索引优化：**

```
ALTER TABLE order_items
ADD INDEX idx_order_product (order_id, product_id);
```

联合索引覆盖关联条件和筛选字段，加速 JOIN 操作。

2. **查询逻辑调整：**

```
SELECT DISTINCT u.*
FROM users u
JOIN (
  SELECT o.user_id
  FROM orders o
  JOIN order_items oi ON o.id = oi.order_id
  WHERE oi.product_id = 123
) AS filtered_orders
ON u.id = filtered_orders.user_id;
```

子查询先过滤出目标订单，减少 orders 表的扫描量。

以 filtered_orders（小表）驱动 users 表关联，降低循环次数。

3. **执行计划验证：**

```
EXPLAIN
SELECT DISTINCT u.*
FROM users u
JOIN (
  SELECT o.user_id
  FROM orders o
  JOIN order_items oi ON o.id = oi.order_id
```

```
  WHERE oi.product_id = 123
) AS filtered_orders
ON u.id = filtered_orders.user_id;
```

输出结果：type=ref，order_items 扫描行数从 100 万降至 500 行，users 表扫描行数从 10 万降至 500 行。

优化效果：

执行时间：从 1.8 秒降至 0.09 秒，性能提升 20 倍。

锁竞争：因减少大表扫描，锁等待事件减少 70%。

三、子查询优化：IN 子查询转换为 JOIN

业务场景：查询有未支付订单的用户，原始子查询耗时 18 秒。

原始 SQL：

```
SELECT u.user_id, u.username
FROM users u
WHERE u.user_id IN (
  SELECT o.user_id
  FROM orders o
  WHERE o.status = '未支付'
);
```

问题诊断：

1. **全表扫描**：orders 表未对 status 字段建立索引，导致子查询全表扫描。
2. **嵌套循环**：外层 users 表每行触发一次子查询，总扫描量达 100 万次。

优化方案：

1. **索引优化：**

```
ALTER TABLE orders
ADD INDEX idx_user_status (user_id, status);
```

2. **查询转换：**

```
SELECT u.user_id, u.username
```

```
FROM users u
JOIN orders o ON u.user_id = o.user_id
WHERE o.status = '未支付'
GROUP BY u.user_id, u.username;
```

使用 JOIN 替代 IN 子查询，利用索引快速过滤。

GROUP BY 消除重复用户（若 user_id 非唯一）。

3. **执行计划验证：**

```
EXPLAIN
SELECT u.user_id, u.username
FROM users u
JOIN orders o ON u.user_id = o.user_id
WHERE o.status = '未支付'
GROUP BY u.user_id, u.username;
```

输出结果：type=ref，orders 表扫描行数从 100 万降至 2 万行，users 表扫描行数从 10 万降至 2 万行。

优化效果：

执行时间：从 18 秒降至 1.8 秒，性能提升 10 倍。

内存消耗：临时表使用量减少 90%，避免内存溢出风险。

四、分页查询优化：从 8.2 秒到 0.03 秒

业务场景：订单列表分页查询，LIMIT 100000, 20 耗时 8.2 秒。

原始 SQL：

```
SELECT *
FROM orders
ORDER BY create_time DESC
LIMIT 100000, 20;
```

问题诊断：

1. **全表扫描**：未建立索引，导致全表扫描 100 万行。
2. **深度分页**：LIMIT 偏移量过大，需遍历前 10 万行数据。

优化方案：

1. **覆盖索引优化：**

```
ALTER TABLE orders
ADD INDEX idx_create_time (create_time DESC) INCLUDE (id, user_id, amount);
```

覆盖索引包含查询字段，避免回表。

2. **游标标记法：**

```
SELECT *
FROM orders
WHERE create_time < (
  SELECT create_time
  FROM orders
  ORDER BY create_time DESC
  LIMIT 100000, 1
)
ORDER BY create_time DESC
LIMIT 20;
```

先查询分页标记点，再按标记点过滤，减少扫描量。

3. **执行计划验证：**

```
EXPLAIN
SELECT *
FROM orders
WHERE create_time < (
  SELECT create_time
  FROM orders
  ORDER BY create_time DESC
  LIMIT 100000, 1
)
ORDER BY create_time DESC
LIMIT 20;
```

输出结果：type=range，扫描行数从 100 万降至 200 行，Extra=Using index。

优化效果：

执行时间：从 8.2 秒降至 0.03 秒，性能提升 273 倍。

I/O 消耗：磁盘读取次数减少 99.98%，适合大数据量分页场景。

五、锁竞争优化：高并发下的库存扣减

业务场景：电商库存扣减接口在高并发下频繁出现锁等待，响应时间超过 5 秒。

原始 SQL：

```
BEGIN;
UPDATE products SET stock = stock - 1 WHERE id = 1;
COMMIT;
```

问题诊断：

1. **行锁范围过大**：未使用索引，导致锁升级为表锁。
2. **事务时间过长**：未及时提交事务，延长锁持有时间。

优化方案：

1. **索引优化**：

```
ALTER TABLE products
ADD INDEX idx_id (id);
```

2. **事务优化**：

```
BEGIN;
UPDATE products SET stock = stock - 1, version = version + 1
WHERE id = 1 AND version = 1;
COMMIT;
```

使用乐观锁（版本号机制），避免锁竞争。

事务内仅包含必要操作，减少锁持有时间。

3. **锁超时设置**：

```
SET innodb_lock_wait_timeout = 5000; -- 锁等待超时 5 秒
```

优化效果：

锁等待次数：从每秒 100 次降至 5 次以下。

响应时间：从 5 秒降至 0.2 秒，吞吐量提升 25 倍。

六、总结与最佳实践

1. 索引设计原则：

覆盖 WHERE、ORDER BY、JOIN 条件，遵循最左前缀原则。

避免在索引列使用函数或类型转换（如 DATE(create_time)）。

2. **查询重构策略：**

用 JOIN 替代子查询，减少嵌套循环。

优先使用覆盖索引，避免回表。

3. **执行计划分析：**

定期使用 EXPLAIN 分析执行计划，重点关注 type、rows、Extra 字段。

警惕 Using filesort（文件排序）和 Using temporary（临时表）。

4. **锁优化技巧：**

为关联字段建立索引，减少锁范围。

使用乐观锁或调整事务隔离级别（如 READ COMMITTED）。

5. **监控与工具：**

启用慢查询日志，定期分析 slow.log。

使用 sysbench 进行压力测试，验证优化效果。

通过系统化应用上述优化策略，可显著提升 SQL 查询性能，同时为高并发、分布式场景奠定基础。