案例驱动式 SQL 教程：从零基础到进阶实战

1. 基础篇：掌握 SQL 核心查询

1.1. 第 1 章：电商用户行为分析 (SELECT, WHERE, GROUP BY 基础)

本章旨在通过一个电商用户行为分析的案例，引导零基础的 SQL 学习者掌握最基础也是最核心的 SQL 查询语句。我们将从一个真实的业务场景出发，逐步构建数据表，插入模拟数据，并通过一系列递进的任务，让读者在实践中学习和理解 SELECT、WHERE、GROUP BY 等关键字的用法。本章的学习目标是让读者能够独立完成简单的数据查询和初步的数据汇总分析，为后续更复杂的 SQL 学习打下坚实的基础。我们将采用“一句话概念 + 一段代码 + 一行注释 + 一个结果截图/表格”的讲解风格，力求清晰明了，避免冗长的理论阐述，让学习过程更加高效和有趣。

业务场景： 分析用户在电商平台的行为，提升用户体验和销售转化率。电商平台每天都会产生海量的用户行为数据，例如用户的浏览记录、加购行为、下单购买等。通过对这些行为数据的分析，可以了解用户的兴趣偏好、购物习惯，从而优化产品推荐、改进页面设计、制定更精准的营销策略。例如，我们可以分析哪些产品最受欢迎，哪些用户是高频消费者，用户在购买前通常会浏览哪些页面等。这些分析结果对于电商平台的运营和决策至关重要。

数据准备：

首先，我们需要创建一个名为 user_behavior 的表来存储用户行为数据。这个表将包含用户 ID、行为类型、产品 ID 以及行为发生的时间戳。以下是创建表的 SQL 语句：

```
CREATE TABLE user_behavior (
    user_id INTEGER,            -- 用户 ID，整数类型
    behavior_type TEXT,         -- 行为类型，文本类型（如'browse', 'purchase'）
    product_id INTEGER,         -- 产品 ID，整数类型
    timestamp DATETIME          -- 时间戳，日期时间类型
);
```

接下来，我们向 user_behavior 表中插入一些模拟数据，以便后续进行查询和分析。这些数据模拟了不同用户在不同时间对不同产品的浏览和购买行为。

```sql
INSERT INTO user_behavior (user_id, behavior_type, product_id, timestamp) VALUE
(1, 'browse', 101, '2025-07-31 10:00:00'),    -- 用户 1 浏览产品 101
(1, 'browse', 102, '2025-07-31 10:05:00'),    -- 用户 1 浏览产品 102
(1, 'purchase', 101, '2025-07-31 10:10:00'), -- 用户 1 购买产品 101
(2, 'browse', 101, '2025-07-31 10:15:00'),    -- 用户 2 浏览产品 101
(2, 'purchase', 102, '2025-07-31 10:20:00'), -- 用户 2 购买产品 102
(3, 'browse', 101, '2025-07-31 10:25:00'),    -- 用户 3 浏览产品 101
(3, 'browse', 102, '2025-07-31 10:30:00'),    -- 用户 3 浏览产品 102
(3, 'browse', 103, '2025-07-31 10:35:00');    -- 用户 3 浏览产品 103
```

执行上述插入语句后，user_behavior 表将包含 8 条记录，涵盖了 3 个用户对 3 种产品的浏览和购买行为。这些数据将作为我们后续 SQL 查询的基础。

任务拆解：

为了系统地学习基础 SQL 查询，我们将本章的学习目标分解为以下几个递进的任务：

1. **找出浏览了特定产品（例如产品 ID 为 101）的用户：** 这个任务将帮助我们学习如何使用 SELECT 语句选择特定的列，以及如何使用 WHERE 子句根据条件过滤数据。同时，我们会引入 DISTINCT 关键字来去除重复的用户 ID。
2. **统计每个用户的行为类型数量：** 这个任务将引导我们学习 GROUP BY 子句的使用，以便对数据进行分组汇总。我们还将使用 COUNT()聚合函数来计算每个用户每种行为类型的发生次数。
3. **找出同时浏览了多个特定产品（例如产品 ID 为 101 和 102）的用户：** 这个任务将在前两个任务的基础上，进一步巩固 GROUP BY 和 HAVING 子句的用法，并引入 IN 操作符来指定多个条件。我们将学习如何通过分组

和条件筛选来找出符合复杂行为模式的用户。

通过完成这些任务，读者将能够掌握 SQL 中最基础也是最常用的查询技能，为后续更复杂的分析打下坚实的基础。

逐步 SQL 代码与结果示例：

- **任务 1：找出浏览了产品 ID 为 101 的用户。**

在这个任务中，我们的目标是找出所有浏览过产品ID为101的用户。首先，我们需要从 user_behavior 表中选择 user_id 列。然后，我们需要使用 WHERE 子句来筛选出行为类型为'browse'且产品ID为101的记录。由于一个用户可能多次浏览同一个产品，为了避免结果中出现重复的用户 ID，我们需要使用 DISTINCT 关键字。

```
-- 查询浏览了产品 ID 为 101 的用户，并去除重复用户 ID
SELECT DISTINCT user_id
FROM user_behavior
WHERE product_id = 101 AND behavior_type = 'browse';
```

执行上述 SQL 语句后，我们期望得到的结果是浏览过产品 101 的用户 ID 列表。根据我们插入的模拟数据，用户 1、用户 2 和用户 3 都浏览过产品 101。

结果示例：

user_id
1
2
3

这个结果清晰地展示了哪些用户对产品 101 表现出了浏览行为。在实际业务中，这类信息可以用于后续的用户画像构建、精准营销等。

- **任务 2：统计每个用户的行为类型数量。**

这个任务的目的是分析每个用户在不同行为类型上的活跃度。我们需要按照 user_id 和 behavior_type 进行分组，然后使用 COUNT(*)函数统计每个分组中的记录数量。GROUP BY 子句用于将数据按照指定的列进行分组，而 COUNT(*)则用于计算每个分组中的行数。

```
-- 统计每个用户每种行为类型的发生次数
SELECT user_id, behavior_type, COUNT(*) AS count
FROM user_behavior
GROUP BY user_id, behavior_type;
```

执行上述SQL语句后，我们将得到一个包含三列的结果集：user_id、

behavior_type 和 count。count 列显示了相应用户的相应行为类型发生的次数。

结果示例：

user_id	behavior_type	count
1	browse	2
1	purchase	1
2	browse	1
2	purchase	1
3	browse	3

从结果中可以看出，用户 1 有 2 次浏览行为和 1 次购买行为，用户 2 有 1 次浏览行为和 1 次购买行为，而用户 3 则有 3 次浏览行为。这种分析有助于了解用户的活跃程度和购买转化情况。

- **任务 3：找出同时浏览了产品 ID 为 101 和 102 的用户。**

这个任务比前两个任务稍微复杂一些，它要求我们找出那些既浏览过产品 101 又浏览过产品 102 的用户。我们可以先筛选出所有浏览行为，并且产品 ID 是 101 或 102 的记录。然后，我们按照 user_id 进行分组。关键点在于，如果一个用户同时浏览了这两个产品，那么在分组后，这个用户对应

的不同product_id的数量应该大于等于2。我们可以使用HAVING子句来对分组后的结果进行过滤，COUNT(DISTINCT product_id)可以统计每个用户浏览的不同产品的数量。

```
-- 找出同时浏览了产品 ID 为 101 和 102 的用户
SELECT user_id
FROM user_behavior
WHERE behavior_type = 'browse' AND product_id IN (101, 102) -- 筛选浏览行为且产
GROUP BY user_id
HAVING COUNT(DISTINCT product_id) >= 2; -- 筛选浏览过至少两种不同产品的用户
```

执行上述 SQL 语句后，我们将得到一个包含符合条件的用户 ID 列表。根据我们的模拟数据，用户 1 和用户 3 都浏览了产品 101 和产品 102。

结果示例：

user_id
1
3

这个查询结果可以帮助我们识别出对多个相关产品感兴趣的用户，这对于交叉推荐、捆绑销售等营销策略的制定非常有价值。

本章总结：

本章通过一个电商用户行为分析的案例，系统地介绍了 SQL 中最基础也是最核心的查询语句。我们首先学习了如何使用 SELECT 语句来选择数据表中的特定列，以及如何使用 WHERE 子句来根据指定的条件过滤数据行。通过 DISTINCT 关键字，我们学会了如何去除查询结果中的重复记录，确保数据的唯一性。接着，我们深入探讨了 GROUP BY 子句的用法，它能够将数据按照一个或多个列进行分组，这对于后续的聚合统计至关重要。结合 COUNT()聚合函数，我们能够轻松统计每个分组中的记录数量。最后，我们学习了 HAVING 子句，它用于对分组后的结果集进行过滤，这与 WHERE 子句在分组前过滤数据行形成了对比。通过完成三个递进的任务，读者应该已经初步掌握了这些基础 SQL 语句的用法，并理解了它们在数据分析中的实际应用。

小练习：

为了巩固本章所学知识，请尝试完成以下两个练习：

1. **统计每个产品被浏览的次数：** 编写 SQL 查询，统计 user_behavior 表中每个产品被用户浏览的总次数。结果应包含 product_id 和相应的浏览次数 browse_count，并按浏览次数降序排列。
2. **找出只浏览过一个产品的用户：** 编写 SQL 查询，找出那些在 user_behavior 表中只浏览过一种不同产品的用户。结果应包含这些用户的 user_id。

常见错误清单：

在学习 SQL 的过程中，初学者常会遇到一些错误。以下是一些在本章内容中可能出现的常见错误及其提示：

- **忘记使用 DISTINCT 导致重复数据：** 当需要统计唯一值时，例如找出浏览过某个产品的不同用户，如果忘记使用 DISTINCT，则可能将同一用户的多次浏览都计算在内，导致结果不准确。
- **GROUP BY 子句使用不当：** 在使用 GROUP BY 时，SELECT 子句中出现的非聚合列（如 user_id, behavior_type）必须包含在 GROUP BY 子句中，否则会导致语法错误或结果不符合预期。
- **混淆 WHERE 和 HAVING 的用法：** WHERE 子句用于在分组前对数据行进行过滤，而 HAVING 子句用于在分组后对分组结果进行过滤。如果将分组后的过滤条件错误地放在 WHERE 子句中，或者将分组前的过滤条件错误地放在 HAVING 子句中，都会导致查询结果错误。
- **聚合函数与非聚合列混合使用不当：** 在 SELECT 子句中，如果使用了聚合函数（如 COUNT(), SUM(), AVG()等），那么所有未包含在聚合函数中的列，都必须出现在 GROUP BY 子句中，否则数据库无法确定如何显示这些非聚合列的值。
- **SQL 语句的书写顺序和执行顺序：** SQL 语句的书写顺序（如 SELECT, FROM, WHERE, GROUP BY, HAVING, ORDER BY）与数据库实际执行这些子句的顺序是不同的。理解执行顺序（FROM -> WHERE -> GROUP BY -> HAVING -> SELECT -> ORDER BY）有助于更好地编写和理解复杂的 SQL 查询。

通过注意这些常见错误，并多加练习，读者可以更快地掌握 SQL 基础查询的精髓。

1.2. 第 2 章：金融销售业绩分析 (聚合函数与排序)

本章将聚焦于金融行业中销售业绩的分析，通过一个具体的案例，带领读者学习 SQL 中常用的聚合函数（如 SUM、AVG、COUNT）以及如何对查询结果进行排序（ORDER BY）和限制返回行数（LIMIT）。在金融产品销售业务中，经常需要对销售数据进行汇总统计，例如计算每个销售员的

总销售额、平均销售额，或者找出销售额最高的产品等。掌握这些 SQL 技能，能够帮助数据分析师快速准确地从大量销售数据中提取有价值的业务洞察，为业绩评估、奖金发放、销售策略调整等提供数据支持。我们将继续采用案例驱动的教学方式，通过实际操作来加深理解。

业务场景： 分析金融产品销售员的业绩，为绩效评估和策略调整提供依据。在银行、证券公司等金融机构中，销售团队负责向客户推荐和销售各类金融产品，如基金、保险、理财产品等。为了评估销售人员的表现，激励团队士气，并优化产品组合和销售策略，管理层需要定期对销售业绩进行分析。这包括但不限于：统计每个销售员的销售额、利润贡献，分析不同产品的销售情况，识别高绩效销售员和滞销产品，以及追踪销售趋势等。通过对这些数据的深入分析，企业可以更科学地进行资源分配和业务决策。

数据准备：

为了进行销售业绩分析，我们需要创建一个名为 sales 的表来存储销售数据。这个表将包含销售员 ID、产品 ID、销售金额以及销售日期等字段。以下是创建表的 SQL 语句：

```
CREATE TABLE sales (
    salesman_id INTEGER,        -- 销售员 ID，整数类型
    product_id INTEGER,         -- 产品 ID，整数类型
    amount DECIMAL(10,2),       -- 销售金额，十进制数，保留两位小数
    sale_date DATE              -- 销售日期，日期类型
);
```

接下来，我们向 sales 表中插入一些模拟的销售数据，这些数据代表了不同销售员在不同日期销售不同产品的记录。

```
INSERT INTO sales (salesman_id, product_id, amount, sale_date) VALUES
(1, 1001, 1000.00, '2025-07-31'),   -- 销售员 1 销售产品 1001，金额 1000
(1, 1002, 1500.00, '2025-07-31'),   -- 销售员 1 销售产品 1002，金额 1500
(2, 1001, 800.00, '2025-07-31'),    -- 销售员 2 销售产品 1001，金额 800
(2, 1003, 1200.00, '2025-07-31'),   -- 销售员 2 销售产品 1003，金额 1200
(3, 1002, 2000.00, '2025-07-31'),   -- 销售员 3 销售产品 1002，金额 2000
(3, 1003, 1800.00, '2025-07-31'),   -- 销售员 3 销售产品 1003，金额 1800
(4, 1001, 900.00, '2025-07-31');    -- 销售员 4 销售产品 1001，金额 900
```

执行上述插入语句后，sales 表将包含 7 条销售记录，涉及 4 位销售员和 3 种金融产品。这些数据将作为我们本章进行销售业绩分析的依据。

任务拆解：

为了系统地学习聚合函数和排序，我们将本章的学习目标分解为以下几个递进的任务：

1. **统计每个销售员的总销售额：** 这个任务将帮助我们学习如何使用 SUM()聚合函数来计算总和，并结合 GROUP BY 子句对数据进行分组。我们还将使用ORDER BY子句对结果进行排序，以便更直观地查看销售业绩。

2. **找出销售额排名前三的销售员：** 在任务 1 的基础上，这个任务将引入 LIMIT 子句，用于限制查询结果返回的行数。这对于快速找出 TOP N 的记录非常有用，例如销售额最高的几位销售员。

3. **统计每个产品的销售次数和总销售额：** 这个任务将综合运用COUNT()和 SUM()聚合函数，并按照产品进行分组。这有助于分析不同产品的受欢迎程度和贡献度。

通过完成这些任务，读者将能够熟练掌握 SQL 中常用的聚合函数和排序技巧，从而能够进行更复杂的销售数据分析。

逐步 SQL 代码与结果示例：

- **任务 1：统计每个销售员的总销售额。**

在这个任务中，我们的目标是计算每位销售员的销售总额。首先，我们需要从 sales 表中选择 salesman_id。然后，我们需要使用 SUM(amount)来计算每个销售员的销售总金额，并为这个汇总结果起一个别名，例如 total_amount。为了实现按销售员分组计算，我们需要使用 GROUP BY salesman_id。最后，为了更清晰地展示业绩，我们可以使用 ORDER BY total_amount DESC 将结果按照总销售额降序排列。

```
-- 统计每个销售员的总销售额，并按总销售额降序排列
SELECT salesman_id, SUM(amount) AS total_amount
FROM sales
GROUP BY salesman_id
ORDER BY total_amount DESC;
```

执行上述 SQL 语句后，我们将得到一个包含两列的结果集：salesman_id

和 total_amount。total_amount 列显示了对应销售员的销售总额。

结果示例：

salesman_id	total_amount
3	3800.00
1	2500.00
2	2000.00
4	900.00

从结果中可以看出，销售员 3 的总销售额最高，达到了 3800.00。这种分析对于评估销售员的绩效非常直接有效。

- **任务 2：找出销售额排名前三的销售员。**

这个任务在任务 1 的基础上，进一步要求我们只找出销售额最高的三位销售员。我们可以复用任务 1 的查询语句，并在其末尾添加 LIMIT 3 子句。LIMIT 子句用于限制查询结果返回的记录数量。

```sql
-- 找出销售额排名前三的销售员
SELECT salesman_id, SUM(amount) AS total_amount
FROM sales
GROUP BY salesman_id
ORDER BY total_amount DESC
LIMIT 3;
```

执行上述 SQL 语句后，我们将得到一个只包含前三名销售员及其总销售额的结果集。

结果示例：

salesman_id	total_amount
3	3800.00
1	2500.00
2	2000.00

这个查询结果对于快速识别高绩效销售员，或者进行奖励表彰非常有用。

在实际业务中，LIMIT 子句经常与 ORDER BY 子句配合使用，以实现 TOP N 查询。

- **任务 3：统计每个产品的销售次数和总销售额。**

这个任务的目的是分析每个金融产品的销售情况。我们需要按照

product_id 进行分组。对于每个产品，我们需要计算两个指标：销售次数和总销售额。销售次数可以使用 COUNT(*)来计算，总销售额可以使用 SUM(amount)来计算。我们同样可以使用 ORDER BY 子句对结果进行排序，例如按照销售次数降序排列。

```
-- 统计每个产品的销售次数和总销售额，并按销售次数降序排列
SELECT product_id, COUNT(*) AS sales_count, SUM(amount) AS total_amount
FROM sales
GROUP BY product_id
ORDER BY sales_count DESC;
```

执行上述 SQL 语句后，我们将得到一个包含三列的结果集：product_id、sales_count 和 total_amount。

结果示例：

product_id	sales_count	total_amount
1001	3	2700.00
1002	2	3500.00
1003	2	3000.00

从结果中可以看出，产品1001的销售次数最多，达到了3次；而产品1002的总销售额最高，达到了3500.00。这类分析有助于了解哪些产品更受欢迎，哪些产品贡献了更多的收入，从而指导产品推广和库存管理。

本章总结：

本章通过一个金融销售业绩分析的案例，重点介绍了SQL中聚合函数（如SUM、COUNT）的使用，以及如何通过GROUP BY子句对数据进行分组汇总。我们还学习了如何使用ORDER BY子句对查询结果进行排序，以及使用LIMIT子句限制返回的记录数量。这些技能是进行数据分析和报表制作的基础。通过统计每个销售员的总销售额、找出销售额排名前三的销售员以及分析每个产品的销售情况，我们展示了SQL在处理实际业务数据时的强大能力。掌握这些核心的聚合和排序操作，能够帮助学习者从大量数据中快速提取关键信息，为业务决策提供支持。

小练习：

为了巩固本章所学知识，请尝试完成以下两个练习：

1. **计算每个销售员的平均销售额：** 编写SQL查询，计算sales表中每位销售员的平均单笔销售金额。结果应包含salesman_id和相应的平均销售额avg_amount。
2. **找出销售产品1001的销售员及其销售额：** 编写SQL查询，找出sales表中所有销售过产品ID为1001的销售员，并列出他们的销售员ID和对应的总销售额。

常见错误清单：

在本章学习过程中，初学者可能会遇到以下一些常见错误：

- **忘记使用 GROUP BY 子句：** 当 SELECT 语句中同时包含聚合函数（如 SUM(), COUNT()）和非聚合列时，必须使用 GROUP BY 子句来指定非聚合列的分组依据。如果忘记使用 GROUP BY，数据库将无法确定如何对非聚合列进行分组，通常会导致语法错误或返回不符合预期的单一汇总结果。

- **在 GROUP BY 子句中使用了聚合函数：** GROUP BY 子句后面应该跟的是列名，用于指定分组的依据，而不是聚合函数。聚合函数应该出现在 SELECT 子句中。

- **ORDER BY 子句使用不当：** 在 ORDER BY 子句中，可以指定一个或多个列进行排序，并可以为每个列指定升序（ASC，默认）或降序（DESC）。如果排序的列名写错或者在 SELECT 子句中不存在（且不是聚合函数的参数），则会导致错误。

- **LIMIT 子句的位置错误：** LIMIT 子句通常是 SQL 查询语句的最后一部分（除了 OFFSET）。如果将其放在 GROUP BY 或 ORDER BY 之前，会导致语法错误。

- **混淆 COUNT(column_name)和 COUNT(*)：**
COUNT(column_name)会计算指定列中非 NULL 值的数量，而 COUNT(*)会计算表中的总行数，包括所有 NULL 值。在选择使用时需要根据具体需求决定。

- **对聚合结果进行错误过滤：** 如果需要根据聚合函数的结果进行过滤，应该使用 HAVING 子句，而不是 WHERE 子句。WHERE 子句在数据分组前进行过滤，而 HAVING 子句在数据分组后进行过滤。

通过理解并避免这些常见错误，可以更有效地编写 SQL 查询，并获得准确的分析结果。

1.3. 第 3 章：图书馆库存管理（常用函数与数据汇总）

本章将围绕教育行业中常见的图书馆库存管理场景展开，通过具体的案例分析，向读者介绍 SQL 中一些常用函数（如字符串函数、日期函数）以及

更复杂的数据汇总技巧。图书馆需要有效地管理其藏书，包括跟踪图书的基本信息、库存数量、借阅情况等。通过 SQL 查询，管理员可以方便地获取各类统计信息，例如不同类别图书的数量、最受欢迎的图书、读者的借阅偏好等，从而为图书采购、排架、剔旧等决策提供数据支持。我们将通过一系列实际任务，帮助读者掌握如何运用 SQL 来处理和分析图书馆的库存数据。

业务场景： 管理图书馆书籍的库存，分析借阅情况，为采购和调配提供决策支持。图书馆的核心任务是管理和提供图书资源。为了高效运营，图书馆需要实时掌握各类图书的库存状态，包括图书的总数量、可借数量、已借出数量等。同时，分析图书的借阅情况，例如哪些图书最受欢迎、哪些类别的图书借阅频率高、不同读者群体的借阅偏好等，对于优化馆藏结构、制定采购计划、提高图书利用率至关重要。通过 SQL 对图书馆数据库进行查询和分析，可以自动化许多繁琐的统计工作，并提供有价值的洞察。

数据准备：

为了进行图书馆库存管理分析，我们需要创建一个名为 library_books 的表来存储图书信息。这个表将包含图书 ID、书名、类别、馆藏数量以及借阅次数等字段。以下是创建表的 SQL 语句：

```
CREATE TABLE library_books (
    book_id INTEGER PRIMARY KEY, -- 图书 ID，主键，整数类型
    title TEXT,                  -- 书名，文本类型
    category TEXT,               -- 类别，文本类型（如'计算机', '数学'）
    count INTEGER,               -- 馆藏数量，整数类型
    borrow_times INTEGER         -- 借阅次数，整数类型
);
```

接下来，我们向 library_books 表中插入一些模拟的图书数据，这些数据代表了图书馆中不同图书的基本信息和借阅情况。

```
INSERT INTO library_books (book_id, title, category, count, borrow_times) VALUES
(1, 'Python 编程', '计算机', 5, 20),      -- 《Python 编程》，计算机类，馆藏 5 本，借阅
(2, '数据结构', '计算机', 3, 15),        -- 《数据结构》，计算机类，馆藏 3 本，借阅 15 次
(3, '高等数学', '数学', 8, 10),          -- 《高等数学》，数学类，馆藏 8 本，借阅 10 次
(4, '线性代数', '数学', 4, 5),           -- 《线性代数》，数学类，馆藏 4 本，借阅 5 次
(5, 'SQL 入门', '数据库', 2, 25),        -- 《SQL 入门》，数据库类，馆藏 2 本，借阅 25
(6, 'Java 编程', '计算机', 6, 18),       -- 《Java 编程》，计算机类，馆藏 6 本，借阅 18
(7, '机器学习', '人工智能', 3, 12),      -- 《机器学习》，人工智能类，馆藏 3 本，借阅 12
(8, '统计学', '数学', 5, 8),             -- 《统计学》，数学类，馆藏 5 本，借阅 8 次
(9, '数据库原理', '数据库', 4, 15),      -- 《数据库原理》，数据库类，馆藏 4 本，借阅 15
(10, '算法导论', '计算机', 2, 20);       -- 《算法导论》，计算机类，馆藏 2 本，借阅 20 次
```

执行上述插入语句后，library_books 表将包含 10 条图书记录，涵盖了计算机、数学、数据库和人工智能等不同类别。这些数据将作为我们本章进行图书馆库存管理分析的依据。

任务拆解：

为了系统地学习常用函数和数据汇总技巧，我们将本章的学习目标分解为以下几个递进的任务：

1. **统计每个类别的书籍数量：** 这个任务将帮助我们巩固 GROUP BY 和 COUNT()的用法，并应用于图书类别的统计分析。
2. **找出借阅次数最多的前 5 本书：** 这个任务将结合 ORDER BY 和 LIMIT 子句，用于找出最受欢迎的图书。
3. **计算各类别书籍的平均借阅次数：** 这个任务将引入 AVG()聚合函数，用于计算每个类别图书的平均借阅热度。

通过完成这些任务，读者将能够更熟练地运用 SQL 进行数据汇总和统计分析，并掌握一些常用的数据处理技巧。

逐步 SQL 代码与结果示例：

- **任务 1：统计每个类别的书籍数量。**

在这个任务中，我们的目标是计算图书馆中每个图书类别的藏书数量。首先，我们需要从 library_books 表中选择 category 列。然后，我们需要使用 COUNT(*)函数来计算每个类别下的图书种类数量，并为这个汇总结果起一个别名，例如 book_count。为了实现按图书类别分组计算，我们需要使用 GROUP BY category。

```sql
-- 统计每个类别的书籍种类数量
SELECT category, COUNT(*) AS book_count
FROM library_books
GROUP BY category;
```

执行上述SQL语句后，我们将得到一个包含两列的结果集：category和book_count。book_count列显示了对应类别的图书种类数量。

结果示例：

category	book_count
人工智能	1
数据库	2
数学	3
计算机	4

从结果中可以看出，计算机类图书的种类最多，有4种；而人工智能类图书的种类最少，只有1种。这类统计有助于图书馆了解馆藏结构。

- **任务2：找出借阅次数最多的前5本书。**

这个任务的目的是找出图书馆中最受欢迎的5本图书。我们需要从

library_books表中选择title和borrow_times列。然后，我们需要使用 ORDER BY borrow_times DESC 将结果按照借阅次数降序排列。最后，我们使用 LIMIT 5 子句来限制只返回前 5 条记录。

```
-- 找出借阅次数最多的前 5 本书
SELECT title, borrow_times
FROM library_books
ORDER BY borrow_times DESC
LIMIT 5;
```

执行上述 SQL 语句后，我们将得到一个包含两列的结果集：title 和 borrow_times，列出了借阅次数最多的 5 本书及其借阅次数。

结果示例：

title	borrow_times
SQL 入门	25
Python 编程	20

算法导论	20
数据结构	15
数据库原理	15

从结果中可以看出，《SQL 入门》这本书的借阅次数最高，达到了 25 次。

这类分析有助于图书馆了解读者的阅读偏好，并为图书采购和推荐提供依据。

- **任务 3：计算各类别书籍的平均借阅次数。**

这个任务的目的是分析不同类别图书的平均受欢迎程度。我们需要按照 category 进行分组。对于每个类别，我们需要计算其所有图书的平均借阅次数。这可以使用 AVG(borrow_times)函数来实现。

```
-- 计算各类别书籍的平均借阅次数
SELECT category, AVG(borrow_times) AS avg_borrow
FROM library_books
GROUP BY category;
```

执行上述 SQL 语句后，我们将得到一个包含两列的结果集：category 和 avg_borrow。avg_borrow 列显示了对应类别图书的平均借阅次数。

结果示例：

category	avg_borrow
人工智能	12.0
数据库	20.0
数学	7.666666666666667
计算机	18.25

从结果中可以看出，数据库类图书的平均借阅次数最高，为 20.0 次；而数学类图书的平均借阅次数相对较低。这类分析有助于图书馆评估不同类别图书的利用效率。

本章总结：

本章通过一个图书馆库存管理的案例，进一步巩固了 SQL 中聚合函数（如 COUNT、AVG）和分组（GROUP BY）的用法，并介绍了如何使用 ORDER BY 和 LIMIT 进行结果排序和限制。我们学习了如何统计不同类别图书的数量，找出最受欢迎的图书，以及计算各类别图书的平均借阅次数。这些操作都是图书馆日常管理和决策中非常实用的技能。通过本章的学习，读者应该能够更灵活地运用 SQL 来处理和分析具有层级或分类结构的数据，并从中提取有价值的业务洞察。

小练习：

为了巩固本章所学知识，请尝试完成以下练习：

1. **统计每个类别书籍的总借阅次数：** 编写 SQL 查询，统计 library_books 表中每个类别图书的总借阅次数。结果应包含 category 和相应的总借阅次数 total_borrow_times，并按总借阅次数降序排列。

2. **找出馆藏数量少于 3 本的图书：** 编写 SQL 查询，找出 library_books 表中馆藏数量（count 列）小于 3 本的图书的书名和馆藏数量。

常见错误清单：

在本章学习过程中，初学者可能会遇到以下一些常见错误：

- **在 SELECT 子句中错误地使用非聚合列：** 当使用 GROUP BY 子句时，SELECT 子句中的非聚合列必须是 GROUP BY 子句中包含的列，或者是聚合函数的参数。如果选择了既未分组也未聚合的列，数据库将无法确定如何显示该列的值，通常会导致错误。
- **混淆聚合函数的作用范围：** 聚合函数（如 AVG()）是对每个分组内的数据进行计算的。如果忘记了 GROUP BY 子句，那么聚合函数将对整个表的数据进行计算（如果查询中只有聚合函数），或者会导致错误（如果查询中同时有聚合函数和非聚合列且没有 GROUP BY）。
- **对文本数据排序时的预期不符：** 对文本列（如 title）使用 ORDER BY 时，排序通常是基于字符的字典顺序。如果期望按照其他规则排序（例如书名长度），则需要使用特定的函数或技巧。
- **LIMIT 子句与 OFFSET 的配合使用：** LIMIT 子句可以配合 OFFSET 子句使用，用于实现分页查询。例如，LIMIT 5 OFFSET 10 表示跳过前 10 条记录，返回接下来的 5 条记录。理解 OFFSET 的用法对于处理大量数据的分页显示非常重要。
- **处理 NULL 值时的注意事项：** 聚合函数（如 COUNT()、SUM()、AVG()）通常会忽略 NULL 值。例如，COUNT(column_name)只计算非

NULL值的数量。如果需要将NULL值视为0或其他特定值参与计算，可以使用 COALESCE()函数或 CASE 表达式进行处理。

通过注意这些常见错误，并多加练习，读者可以更深入地理解和掌握 SQL 的数据汇总和分析功能。

2. 进阶篇：玩转多表操作与高级查询

2.1. 第 4 章：医院患者就诊分析 (多表关联 JOIN)

本章将进入 SQL 学习的进阶阶段，重点介绍多表关联查询（JOIN）的应用。我们将通过一个医院患者就诊分析的案例，详细讲解不同类型的 JOIN 操作（如 INNER JOIN, LEFT JOIN, RIGHT JOIN, FULL JOIN）及其使用场景。在医院信息系统中，患者的基本信息和就诊记录通常存储在不同的表中，通过关联查询，我们可以将分散的信息整合起来，进行更全面的分析，例如统计每个患者的就诊次数、分析特定疾病的就诊情况等。掌握 JOIN 操作是进行复杂数据分析和报表制作的关键技能。

业务场景： 分析医院患者的就诊记录，了解疾病分布和患者就诊行为。医院每天都会接待大量患者，并记录他们的基本信息以及每次就诊的详细情况，包括诊断结果、就诊日期、费用等。通过对这些数据的关联分析，医院管理者可以了解不同疾病的发病率、患者的就医频率、不同科室的接诊压力等，从而优化医疗资源配置、改进医疗服务流程、提升患者满意度。例如，可以分析哪些疾病的患者复诊率高，哪些年龄段的患者更容易患某种疾病，以及不同性别患者在疾病谱上的差异等。

数据准备：

为了进行医院患者就诊分析，我们需要至少两个表：一个存储患者基本信息的表（patients），另一个存储患者就诊记录的表（visits）。这两个表可以通过患者 ID 进行关联。

首先，创建 patients 表：

```
CREATE TABLE patients (
    patient_id INTEGER PRIMARY KEY, -- 患者 ID，主键，整数类型
    name TEXT,                      -- 患者姓名，文本类型
    age INTEGER,                    -- 患者年龄，整数类型
    gender TEXT                     -- 患者性别，文本类型（如'男', '女'）
);
```

然后，创建 visits 表：

```
CREATE TABLE visits (
    visit_id INTEGER PRIMARY KEY,     -- 就诊记录 ID，主键，整数类型
    patient_id INTEGER,               -- 患者 ID，外键，关联 patients 表
    diagnosis TEXT,                   -- 诊断结果，文本类型
    visit_date DATE,                  -- 就诊日期，日期类型
    fee DECIMAL(10, 2),               -- 就诊费用，十进制数，保留两位小数
    FOREIGN KEY (patient_id) REFERENCES patients(patient_id) -- 定义外键关系
);
```

接下来，我们向这两个表中插入一些模拟数据。

向 patients 表插入数据：

```
INSERT INTO patients (patient_id, name, age, gender) VALUES
(1, '张三', 45, '男'),
(2, '李四', 32, '女'),
(3, '王五', 50, '男'),
(4, '赵六', 28, '女'),
(5, '钱七', 63, '男');
```

向 visits 表插入数据：

```
INSERT INTO visits (visit_id, patient_id, diagnosis, visit_date, fee) VALUES
(1, 1, '感冒', '2025-07-01', 200.00),
(2, 1, '高血压', '2025-07-15', 300.00),
(3, 2, '感冒', '2025-07-10', 150.00),
(4, 3, '糖尿病', '2025-07-20', 500.00),
(5, 3, '高血压', '2025-07-25', 300.00),
(6, 4, '感冒', '2025-07-30', 150.00),
(7, 5, '心脏病', '2025-07-05', 800.00),
(8, 5, '高血压', '2025-07-28', 300.00);
```

执行上述插入语句后，patients 表将包含 5 条患者记录，visits 表将包含 8 条就诊记录。这些数据将作为我们本章进行医院患者就诊分析的依据。

任务拆解：

为了系统地学习多表关联查询，我们将本章的学习目标分解为以下几个递进的任务：

1. **获取每个患者的就诊次数：** 这个任务将帮助我们学习如何使用 LEFT JOIN 来确保即使某些患者没有就诊记录，也能在结果中显示，并结合 GROUP BY 和 COUNT()进行统计。
2. **找到诊断为特定疾病（例如高血压）的患者及其就诊次数：** 这个任务将在关联查

询的基础上，加入 WHERE 子句进行条件过滤，并进一步巩固分组统计的用法。

3. **统计不同年龄段患者就诊次数分布：** 这个任务将引入 CASE 表达式对年龄进行分段，并结合 JOIN 和 GROUP BY 进行更复杂的分组统计。

通过完成这些任务，读者将能够理解并掌握 SQL 中多表关联查询的核心概念和常用技巧。

逐步 SQL 代码与结果示例：

- **任务 1：获取每个患者的就诊次数。**

在这个任务中，我们的目标是列出所有患者及其对应的就诊次数，即使某些患者可能没有就诊记录。因此，我们需要使用 patients 表作为主表，并使用 LEFT JOIN 来连接 visits 表。LEFT JOIN 会返回左表（patients）的所有记录，即使在右表（visits）中没有匹配的记录，此时右表的字段将以 NULL 值填充。然后，我们按照 patient_id 进行分组，并使用 COUNT(visits.visit_id)来统计每个患者的就诊次数。注意，这里使用 COUNT(visits.visit_id)而不是 COUNT(*)，是为了避免将没有就诊记录的患者计为 1 次（因为 COUNT(*)会计算分组内的所有行，包括 NULL 值）。

```
-- 获取每个患者的就诊次数，包括就诊次数为 0 的患者
SELECT p.patient_id, p.name, COUNT(v.visit_id) AS visit_count
FROM patients p
LEFT JOIN visits v ON p.patient_id = v.patient_id
GROUP BY p.patient_id
ORDER BY visit_count DESC;
```

执行上述 SQL 语句后，我们将得到一个包含三列的结果集：patient_id、name 和

visit_count。

结果示例：

patient_id	name	visit_count
1	张三	2
3	王五	2
5	钱七	2
2	李四	1
4	赵六	1

从结果中可以看出，患者张三、王五、钱七各有 2 次就诊记录，而李四和赵六各有 1 次就诊记录。如果存在没有就诊记录的患者，他们的 visit_count 将会是 0。

- **任务 2：找到诊断为高血压的患者及其就诊次数。**

这个任务的目的是找出所有被诊断为高血压的患者，并统计他们因高血压就诊的次数。我们需要连接 patients 表和 visits 表，并使用 WHERE 子句来筛选 diagnosis 为'高血压'的记录。然后，我们按照患者 ID 进行分组，并统计就诊次数。

```sql
-- 找到诊断为高血压的患者及其因高血压就诊的次数
SELECT p.patient_id, p.name, COUNT(v.visit_id) AS hypertension_visits
FROM patients p
JOIN visits v ON p.patient_id = v.patient_id
WHERE v.diagnosis = '高血压'
GROUP BY p.patient_id
ORDER BY hypertension_visits DESC;
```

执行上述SQL语句后，我们将得到一个包含三列的结果集：patient_id、name和hypertension_visits，列出了所有患有高血压的患者及其因高血压就诊的次数。

结果示例：

patient_id	name	hypertension_visits
1	张三	1
3	王五	1
5	钱七	1

从结果中可以看出，患者张三、王五、钱七都被诊断过高血压，并且各有 1 次因高血压就诊的记录。这个查询可以帮助医院识别出特定疾病的患者群体。

- **任务 3：统计不同年龄段患者就诊次数分布。**

这个任务要求我们分析不同年龄段患者的就诊情况。我们需要先将患者的年龄划分为不同的年龄段（例如，30岁以下，30-50岁，50岁以上），这可以通过CASE表达式来实现。然后，我们需要连接 patients 表和 visits 表（使用 LEFT JOIN 以确保统计到所有患者，即使他们没有就诊记录）。接着，我们按照年龄段进行分组，并统计每个年

龄段的患者人数以及总就诊次数。这里总就诊次数的计算需要对每个患者进行汇总，然后再对年龄段进行汇总，可以使用子查询或者窗口函数，但为了简化，我们先统计每个年龄段的患者人数和这些患者的总就诊次数。

```
-- 统计不同年龄段患者就诊次数分布
SELECT
    CASE
        WHEN age < 30 THEN 'Under 30'
        WHEN age BETWEEN 30 AND 50 THEN '30-50'
        ELSE 'Over 50'
    END AS age_group,
    COUNT(DISTINCT p.patient_id) AS patient_count, -- 统计不同患者的数量
    COUNT(v.visit_id) AS total_visits -- 统计总就诊次数
FROM patients p
LEFT JOIN visits v ON p.patient_id = v.patient_id
GROUP BY age_group
ORDER BY age_group;
```

执行上述 SQL 语句后，我们将得到一个包含三列的结果集：age_group、patient_count 和 total_visits。

结果示例：

age_group	patient_count	total_visits
30-50	3	5
Over 50	1	2

Under 30	1	1

结果显示，30-50岁年龄段的患者就诊次数最多，共5次。这个分析有助于医院了解不同年龄段人群的就诊需求，为制定针对性的健康管理服务提供依据。

本章总结：

本章通过一个医院患者就诊分析的案例，系统介绍了 SQL 中多表关联查询（JOIN）的核心概念和应用。我们重点学习了 LEFT JOIN 和 INNER JOIN 的使用场景和区别：LEFT JOIN 用于保留左表所有记录，即使右表无匹配；INNER JOIN 则只返回两个表中都有匹配的记录。通过统计每个患者的就诊次数、分析特定疾病的就诊情况以及按年龄段统计就诊分布，我们实践了 JOIN 操作与 GROUP BY、聚合函数（如 COUNT）、WHERE 子句以及 CASE 表达式的结合使用。掌握这些多表查询技巧，能够帮助我们从更广阔的视角分析复杂数据，提取更深层次的业务洞察。

小练习：

为了巩固本章所学知识，请尝试完成以下练习：

1. **统计每种疾病的就诊总费用：** 编写 SQL 查询，统计 visits 表中每种诊断（diagnosis）的总费用（fee）。结果应包含 diagnosis 和 total_fee，并按总费用降序排列。
2. **找出没有就诊记录的患者：** 修改任务 1 的 SQL 查询，使其只返回那些在 visits 表中没有对应就诊记录的患者 ID 和姓名。
3. **分析不同性别患者的平均就诊费用：** 编写 SQL 查询，连接 patients 表和 visits 表，统计不同性别（gender）患者的平均每次就诊费用。

常见错误清单：

在本章学习过程中，初学者可能会遇到以下一些常见错误：

- **混淆 JOIN 类型导致数据丢失或冗余：** 错误地选择 INNER JOIN 而实际需要 LEFT JOIN，可能导致结果丢失左表中部分记录。反之，则可能引入不必要的 NULL 值或冗余数据。

- **遗漏连接条件或连接条件错误：** 在多表 JOIN 时，如果忘记写 ON 子句或者连接条件写错，可能会导致笛卡尔积，产生大量无意义的组合数据，严重影响查询性能和结果准确性。

- **在 JOIN 后的 WHERE 子句中错误地过滤 NULL 值：** 当使用 LEFT JOIN 时，右表中未匹配的字段会显示为 NULL。如果在 WHERE 子句中直接对这些字段进行过滤（例如 WHERE right_table.column = 'value'），可能会无意中将左表中未匹配的行也过滤掉，使得 LEFT JOIN 的效果等同于 INNER JOIN。正确的做法是将右表的过滤条件也放在 ON 子句中，或者使用 IS NULL 进行判断。

- **GROUP BY 子句使用不当：** 在进行多表 JOIN 后进行分组统计时，SELECT 子句中的非聚合列必须包含在 GROUP BY 子句中，否则会导致错误或不确定的结果。

- **对 CASE 表达式的结果进行错误处理：** 当使用 CASE 表达式生成新的分组列时，确保在 GROUP BY 子句中正确地引用这个新列（或其完整表达式）。

通过注意这些常见错误，并多加练习，读者可以更深入地理解和掌握 SQL 的多表关联查询。

2.2. 第 5 章：电商促销活动效果分析 (子查询应用)

本章重点介绍 SQL 中子查询的应用，通过一个电商平台促销活动效果分析的案例，帮助学习者掌握如何使用子查询来解决复杂的业务问题。业务场景设定为电商平台需要评估近期举办的各项促销活动对订单量和销售额的影响，以便优化未来的营销策略和资源分配。为此，我们创建了两张数据表：promotions 表和 orders 表。promotions 表用于存储促销活动的基本信息，包含 promo_id（促销 ID，主键）、promo_name（促销名称）、start_date（开始日期）、end_date（结束日期）和 discount（折扣力度，例如 0.8 表示打八折）五个字段。orders 表则用于记录用户的订单信息，包含 order_id（订单 ID，主键）、user_id（用户 ID）、amount（订单金额）、order_date（订单日期）和 promo_id（参与的促销 ID，可为空，表示未参与促销）五个字段。通过 CREATE TABLE 语句创建这两张表后，我们向 promotions 表插入了 3 条促销活动数据（如“夏季折扣”、“国庆特惠”、“黑五促销”），并向 orders 表插入了 9 条订单记录，其中部分订单关联了促销活动，部分订单则没有。

数据准备：

首先，创建 promotions 表并插入模拟数据：

```
CREATE TABLE promotions (
    promo_id INTEGER PRIMARY KEY,     -- 促销 ID，主键
    promo_name TEXT,                  -- 促销名称
    start_date DATE,                  -- 开始日期
    end_date DATE,                    -- 结束日期
    discount DECIMAL(3,2)             -- 折扣力度
);

INSERT INTO promotions (promo_id, promo_name, start_date, end_date, discount) V
(1, '夏季折扣', '2025-06-01', '2025-06-30', 0.9),
(2, '国庆特惠', '2025-10-01', '2025-10-07', 0.8),
(3, '黑五促销', '2025-11-28', '2025-11-29', 0.7);
```

然后，创建 orders 表并插入模拟数据：

```sql
CREATE TABLE orders (
    order_id INTEGER PRIMARY KEY,     -- 订单 ID，主键
    user_id INTEGER,                  -- 用户 ID
    amount DECIMAL(10,2),             -- 订单金额
    order_date DATE,                  -- 订单日期
    promo_id INTEGER,                 -- 促销 ID，外键，关联 promotions 表
    FOREIGN KEY (promo_id) REFERENCES promotions(promo_id)
);

INSERT INTO orders (order_id, user_id, amount, order_date, promo_id) VALUES
(1, 101, 200.00, '2025-06-15', 1),
(2, 102, 150.00, '2025-06-20', 1),
(3, 103, 300.00, '2025-06-25', 1),
(4, 101, 400.00, '2025-10-02', 2),
(5, 104, 350.00, '2025-10-03', 2),
(6, 102, 500.00, '2025-11-28', 3),
(7, 105, 100.00, '2025-11-29', 3),
(8, 106, 250.00, '2025-07-10', NULL), -- 未参与促销
(9, 107, 200.00, '2025-08-20', NULL); -- 未参与促销
```

执行上述插入语句后，promotions 表将包含 3 条促销活动记录，orders 表将包含 9 条订单记录。这些数据将作为我们本章进行电商促销活动效果分析的依据。

任务拆解：

为了系统地学习子查询的应用，我们将本章的学习目标分解为以下几个递进的任务：

1. **统计每个促销活动的订单数量和总销售额：** 这个任务将帮助我们学习如何使用 LEFT JOIN 将促销活动表与订单表关联，并进行分组统计，同时巩固聚合函数的使用。
2. **比较有促销活动的订单与无促销活动的订单的平均金额：** 这个任务将引入 CASE 表达式对订单进行分类，并结合 GROUP BY 进行分组比较，同时也会初步涉及子查询的思想。

3. **找出订单金额高于其参与促销活动平均订单金额的订单：** 这个任务将明确使用子查询来计算每个促销活动的平均订单金额，并在主查询中进行比较，从而展示子查询在复杂条件筛选中的应用。

通过完成这些任务，读者将能够理解并掌握 SQL 中子查询的基本用法和常见场景。

逐步 SQL 代码与结果示例：

- **任务 1：统计每个促销活动的订单数量和总销售额。**

这个任务的目的是分析各个促销活动的效果，包括吸引的订单量和产生的总销售额。

我们需要将 promotions 表与 orders 表进行 LEFT JOIN，以确保即使某个促销活动没有产生任何订单，也能在结果中显示。然后，按照促销活动进行分组，并使用 COUNT()和 SUM()聚合函数进行统计。

```
-- 统计每个促销活动的订单数量和总销售额
SELECT p.promo_id, p.promo_name,
        COUNT(o.order_id) AS order_count, -- 统计订单数量
        SUM(o.amount) AS total_sales        -- 统计总销售额
FROM promotions p
LEFT JOIN orders o ON p.promo_id = o.promo_id
GROUP BY p.promo_id
ORDER BY total_sales DESC;
```

执行上述 SQL 语句后，我们将得到一个包含四列的结果集：promo_id、promo_name、order_count 和 total_sales。

结果示例：

promo_id	promo_name	order_count	total_sales
1	夏季折扣	3	650.00
2	国庆特惠	2	750.00
3	黑五促销	2	600.00

从结果中可以看出，“夏季折扣”活动吸引了最多的订单（3 笔），而“国庆特惠”活动的总销售额最高（750.00）。这个分析直接反映了各促销活动的市场吸引力和创收能力。

- **任务 2：比较有促销活动的订单与无促销活动的订单的平均金额。**

这个任务的目的是评估促销活动对订单金额的影响。我们需要将订单分为“参与促销”和“未参与促销”两类，然后分别计算它们的平均订单金额。这可以通过在 orders 表内部根据 promo_id 是否为空进行分组，并使用 AVG()函数来实现。

```
-- 比较有促销活动的订单与无促销活动的订单的平均金额
SELECT CASE WHEN promo_id IS NOT NULL THEN 'With Promo'
            ELSE 'Without Promo'
       END AS order_type, -- 订单类型：有促销 / 无促销
       AVG(amount) AS average_amount -- 平均订单金额
FROM orders
GROUP BY order_type;
```

执行上述 SQL 语句后，我们将得到一个包含两列的结果集：order_type 和 average_amount。

结果示例：

order_type	average_amount
With Promo	371.43
Without Promo	225.00

结果显示，参与促销的订单平均金额（约 371.43）显著高于未参与促销的订单平均金额（225.00）。这个分析有助于评估促销活动对提升客单价的效果。

- **任务 3：找出订单金额高于其参与促销活动平均订单金额的订单。**

这个任务要求我们找出那些“表现突出”的订单，即其金额超过了该订单所参与的促销活动的平均订单金额。我们需要先计算出每个促销活动的平均订单金额，然后将这个平均值与每笔订单的实际金额进行比较。这需要使用子查询来获取每个促销活动的平均金额。

```
-- 找出订单金额高于其参与促销活动平均订单金额的订单
SELECT o.order_id, o.user_id, o.amount, o.promo_id, p.promo_name
FROM orders o
JOIN promotions p ON o.promo_id = p.promo_id
WHERE o.amount > (
    SELECT AVG(o_inner.amount) -- 子查询：计算当前订单所属促销活动的平均订单金
    FROM orders o_inner
    WHERE o_inner.promo_id = o.promo_id -- 关联子查询，与外部查询的订单促销 ID
);
```

执行上述 SQL 语句后，我们将得到一个包含符合条件的订单信息的结果集。

结果示例：

order_id	user_id	amount	promo_id	promo_name
3	103	300.00	1	夏季折扣
4	101	400.00	2	国庆特惠
6	102	500.00	3	黑五促销

从结果中可以看出，订单 ID 为 3、4、6 的订单，其金额分别高于它们各自参与的“夏季折扣”、“国庆特惠”和“黑五促销”活动的平均订单金额。这个查询的关键在于子查询 (SELECT AVG(o_inner.amount) FROM orders o_inner WHERE o_inner.promo_id = o.promo_id)。这是一个**关联子查询**，它会为外部查询中的每一行（即每一笔订单）执行一次，计算该订单所属促销活动的平均订单金额。然后，外部查询的 WHERE 子句将这个平均值与订单的实际金额进行比较。

本章总结：

本章通过一个电商促销活动效果分析的案例，重点介绍了 SQL 中子查询的应用。我们学习了如何使用 LEFT JOIN 进行多表关联和分组统计，如何使用 CASE 表达式进行数据分类，以及如何编写和使用子查询（特别是关联子查询）来解决复杂的业务问题。子查询允许我们在一个查询内部嵌套另一个查询，从而能够基于更复杂的逻辑进行数据筛选和计算。通过统计促销活动的订单量和销售额、比较不同类别订单的平均金额，以及找出高于平均水平的订单，我们展示了子查询在实际数据分析中的强大功能和灵活性。掌握子查询对于处理多层次、依赖性的数据关系至关重要。

小练习：

为了巩固本章所学知识，请尝试完成以下练习：

1. **找出参与了所有促销活动的用户（如果存在这样的业务逻辑）：** 这个任务可能需要使用多重嵌套子查询或者 NOT EXISTS 子句。

2. **统计每个用户参与促销活动的次数以及他们的总消费金额：** 结合 GROUP BY 和子查询（如果需要计算复杂的促销相关金额）。

3. **找出那些订单金额高于所有订单平均金额的促销订单：** 思考这个任务与任务 3 的区别，以及子查询应该如何编写。

常见错误清单：

在本章学习过程中，初学者可能会遇到以下一些常见错误：

- **子查询返回多行导致错误：** 如果子查询预期返回单个值（例如，在比较运算符 =、>、< 之后），但实际返回了多行，数据库会报错。确保子查询的逻辑正确，或者在可能返回多行时使用 IN、ANY、ALL 等操作符。
- **关联子查询性能问题：** 关联子查询会为外部查询的每一行执行一次，如果数据量很大，可能会导致性能问题。需要仔细评估并考虑是否有更优的写法，例如使用 JOIN。
- **子查询中的列引用不明确：** 在子查询中引用外部查询的列时，要确保列名是明确的，或者使用表别名进行限定，以避免歧义。
- **忽略 NULL 值处理：** 在子查询中，如果涉及到可能为 NULL 的列，需要注意 NULL 值对比较和聚合操作的影响。例如，NULL 与任何值的比较结果都是 UNKNOWN。
- **子查询嵌套过深导致可读性差：** 虽然 SQL 支持多层嵌套子查询，但过度嵌套会使 SQL 语句难以理解和维护。在复杂情况下，可以考虑使用公共表表达式（CTE）来简化。

通过理解并避免这些常见错误，可以更有效地利用子查询进行复杂的数据分析。

2.3. 第 6 章：金融投资账户分析 (窗口函数入门)

本章将引导学习者初步接触 SQL 中强大的窗口函数（Window Functions），通过一个金融投资账户资金变动分析的案例，展示窗口函数在处理诸如累计求和、排名、分区计算等复杂分析任务时的独特优势。业务场景设定为一家金融机构需要监控其客户投资账户的资金流动情况，计算每笔交易后的账户实时余额，找出特定时间点（如月末）的账户余额，以及统计特定时间段内（如 7 月份）账户的总存款和总取款金额，

以便进行风险控制和客户服务。为此，我们创建了一张名为 investment_accounts 的表，用于记录账户的交易明细。该表包含 account_id（交易 ID，主键）、user_id（用户 ID）、transaction_date（交易日期）、amount（交易金额，正数表示存款，负数表示取款）和 type（交易类型，如'deposit'或'withdraw'）五个字段。通过 CREATE TABLE 语句建表后，我们向表中插入了 9 条模拟的交易记录，涉及 3 个不同的用户（ID 为 1001 至 1003）。

数据准备：

首先，创建 investment_accounts 表并插入模拟数据：

```
CREATE TABLE investment_accounts (
    account_id INTEGER PRIMARY KEY, -- 交易 ID，主键
    user_id INTEGER,                -- 用户 ID
    transaction_date DATE,          -- 交易日期
    amount DECIMAL(10,2),           -- 交易金额（正为存款，负为取款）
    type TEXT                       -- 交易类型：'deposit' 或 'withdraw'
);

INSERT INTO investment_accounts (account_id, user_id, transaction_date, amount, t
(1, 1001, '2025-07-01', 10000.00, 'deposit'),
(2, 1001, '2025-07-05', -2000.00, 'withdraw'),
(3, 1001, '2025-07-15', 5000.00, 'deposit'),
(4, 1002, '2025-07-10', 8000.00, 'deposit'),
(5, 1002, '2025-07-20', -3000.00, 'withdraw'),
(6, 1002, '2025-07-25', 10000.00, 'deposit'),
(7, 1003, '2025-07-12', 15000.00, 'deposit'),
(8, 1003, '2025-07-18', -5000.00, 'withdraw'),
(9, 1003, '2025-07-28', -2000.00, 'withdraw');
```

执行上述插入语句后，investment_accounts 表将包含 9 条交易记录，涉及 3 个用户。这些数据将作为我们本章进行金融投资账户分析的依据。

任务拆解：

为了系统地学习窗口函数的基本用法，我们将本章的学习目标分解为以下几个递进的任务：

1. **计算每个投资账户在每笔交易发生后的实时账户余额：** 这个任务将帮助我们学习如何使用 SUM()作为窗口函数，配合 OVER (PARTITION BY ... ORDER BY ...)子句来实现累计计算。
2. **找出每个投资账户在最近一次交易发生后的账户余额：** 这个任务将在任务 1 的基础上，引入 ROW_NUMBER()窗口函数，用于获取每个分区内的特定排名记录。
3. **计算每个投资账户在特定月份（例如 7 月份）的总存款金额和总取款金额：** 这个任务将巩固聚合函数与 GROUP BY 的用法，并可与窗口函数进行对比。

通过完成这些任务，读者将对 SQL 窗口函数的基本概念和强大功能有一个初步的认识。

逐步 SQL 代码与结果示例：

- **任务 1：计算每个投资账户在每笔交易发生后的实时账户余额。**

这个任务的目的是追踪每个用户账户在每笔交易发生后的资金变动情况，并计算出实时的账户余额。我们可以使用 SUM(amount)作为窗口函数，并按照 user_id 进行分区，在每个分区内按照 transaction_date 进行排序，从而实现累加求和。

```sql
-- 计算每个投资账户在每笔交易发生后的实时账户余额
SELECT account_id, user_id, transaction_date, amount, type,
       SUM(amount) OVER (PARTITION BY user_id ORDER BY transaction_date) AS
FROM investment_accounts
ORDER BY user_id, transaction_date;
```

执行上述 SQL 语句后，我们将得到一个包含六列的结果集，其中 balance 列显示了每笔交易后的实时余额。

结果示例：

account_id	user_id	transaction_date	amount	type	balance
1	1001	2025-07-01	10000.00	deposit	10000.00
2	1001	2025-07-05	-2000.00	withdraw	8000.00
3	1001	2025-07-15	5000.00	deposit	13000.00
4	1002	2025-07-10	8000.00	deposit	8000.00
5	1002	2025-07-20	-3000.00	withdraw	5000.00
6	1002	2025-07-25	10000.00	deposit	15000.00
7	1003	2025-07-12	15000.00	deposit	15000.00
8	1003	2025-07-18	-5000.00	withdraw	10000.0

					0
9	1003	2025-07-28	-2000.00	withdraw	8000.00

从结果中可以看出，用户 1001 在 7 月 1 日存入 10000 元后，余额为 10000 元；7 月 5 日取出 2000 元后，余额变为 8000 元；7 月 15 日再次存入 5000 元后，余额变为 13000 元。这种分析对于实时监控账户状态、预警异常交易至关重要。

- **任务 2：找出每个投资账户在最近一次交易发生后的账户余额。**

这个任务要求我们获取每个用户账户在分析时间点（即数据集中最近一次交易）的最终余额。我们可以先使用 ROW_NUMBER()窗口函数为每个用户的交易记录按时间降序编号，然后筛选出编号为 1 的记录，即为最近一次交易。结合任务 1 中计算的实时余额，即可得到最终余额。

```
-- 找出每个投资账户在最近一次交易发生后的账户余额
WITH AccountBalances AS (
    SELECT account_id, user_id, transaction_date, amount, type,
           SUM(amount) OVER (PARTITION BY user_id ORDER BY transaction_date
           ROW_NUMBER() OVER (PARTITION BY user_id ORDER BY transaction_da
    FROM investment_accounts
)
SELECT user_id, transaction_date, amount, balance
FROM AccountBalances
WHERE rk = 1
ORDER BY user_id;
```

执行上述 SQL 语句后，我们将得到一个包含每个用户最近一次交易及其对应余额的结果集。

结果示例：

user_id	transaction_date	amount	balance
1001	2025-07-15	5000.00	13000.00
1002	2025-07-25	10000.00	15000.00
1003	2025-07-28	-2000.00	8000.00

这个结果直接反映了每个账户在分析时的最新状态。用户 1001 的最终余额为 13000 元，用户 1002 的最终余额为 15000 元，用户 1003 的最终余额为 8000 元。

- **任务 3：计算每个投资账户在特定月份（例如 7 月份）的总存款金额和总取款金额。**

这个任务的目的是分析每个账户在特定时期内的资金流入和流出情况。我们可以使用条件聚合函数 SUM(CASE WHEN ... THEN ... ELSE ... END)结合 GROUP BY 来实现。虽然这个任务不直接依赖窗口函数，但它是对聚合操作的巩固，并可与窗口函数进行对比。

```sql
-- 计算每个投资账户在特定月份（例如 7 月份）的总存款金额和总取款金额
SELECT user_id,
       SUM(CASE WHEN type = 'deposit' THEN amount ELSE 0 END) AS total_depo
       SUM(CASE WHEN type = 'withdraw' THEN amount ELSE 0 END) AS total_wit
FROM investment_accounts
WHERE strftime('%Y-%m', transaction_date) = '2025-07' -- SQLite 日期格式化函数
GROUP BY user_id;
```

注意：不同的数据库系统日期格式化函数可能不同。上述 SQL 语句适用于 SQLite。对于 MySQL，可以使用 DATE_FORMAT(transaction_date, '%Y-%m') = '2025-07'。

执行上述 SQL 语句后，我们将得到一个包含每个用户 7 月份总存款和总取款金额的结果集。

结果示例：

user_id	total_deposit	total_withdraw
1001	15000.00	-2000.00
1002	18000.00	-3000.00
1003	15000.00	-7000.00

从结果中可以看出，用户 1001 在 7 月份总共存入 15000 元，取出 2000 元。这个分析有助于了解账户在特定时期内的资金流入流出情况，评估用户的投资活跃度和风险偏好。

本章总结：

本章通过一个金融投资账户资金变动分析的案例，初步介绍了 SQL 窗口函数的基本概念和应用。我们学习了如何使用 SUM() OVER (PARTITION BY ... ORDER BY ...)来计算累加值（如实时账户余额），以及如何使用 ROW_NUMBER() OVER (PARTITION BY ... ORDER BY ...)来为分区内的行分配序号，从而获取特定排名的记录（如最近一次交易）。同时，我们也复习了条件聚合 SUM(CASE WHEN ...)的用法。窗口函数为处理复杂的分析任务（如排名、移动平均、累积计算等）提供了强大的支持，能够在不减少结果集行数的情况下，为每一行返回基于其“窗口”内其他行的计算结果。掌握窗口函数是提升 SQL 数据分析能力的关键一步。

小练习：

为了巩固本章所学知识，请尝试完成以下练习：

1. **计算每个账户在7月份的期初余额和期末余额：** 期初余额是指7月1日之前的最后一笔交易后的余额（如果存在），期末余额是指7月31日（或数据集中最后一天）的余额。这可能需要结合使用窗口函数和子查询。
2. **找出在7月份内有过任何取款操作的账户及其取款总金额：** 使用窗口函数或聚合函数实现。
3. **计算每个账户每笔存款交易后，账户余额相较于前一次交易的增长百分比（如果前一次交易存在）：** 这可能需要使用 LAG()窗口函数。

常见错误清单：

在本章学习过程中，初学者可能会遇到以下一些常见错误：

- **窗口函数 OVER 子句的语法错误：** 例如括号不匹配、关键字写错（如 PARTITION 写成 PARTITIONING）、ORDER BY 子句在未指定 PARTITION BY 时使用不当等。
- **ROWS BETWEEN 子句使用不当：** 在定义窗口框架时，如果未正确理解 ROWS BETWEEN 的含义，可能导致累加范围不符合预期。例如，默认情况下，如果指定了 ORDER BY，窗口框架是 RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW，这意味着对于具有相同排序值的行，它们都会被视为 CURRENT ROW。
- **混淆窗口函数与聚合函数：** 窗口函数不会减少结果集的行数，而标准的聚合函数

（不带 OVER 子句）会。如果期望的结果是每行都有计算结果，应该使用窗口函数。

- **在 WHERE 或 GROUP BY 子句中直接使用窗口函数：** 窗口函数是在 SELECT 子句（以及 ORDER BY 子句）中计算的，不能直接在 WHERE 或 GROUP BY 子句中使用窗口函数的结果。如果需要对窗口函数的结果进行筛选或分组，通常需要将窗口函数查询作为子查询或 CTE。

- **忽略 NULL 值在窗口函数中的影响：** 与聚合函数类似，某些窗口函数（如 LAG(), LEAD()）在处理 NULL 值时需要特别注意，可能需要使用 COALESCE()或 IFNULL()等函数进行处理。

通过理解并避免这些常见错误，可以更有效地利用窗口函数进行复杂的数据分析。

3. 难点&盲点篇：攻克性能瓶颈与复杂逻辑

3.1. 第 7 章：电商业务 SQL 性能优化 (索引与执行计划)

7.1. 业务场景：优化电商平台的订单查询性能

在电商业务中，订单数据的查询频率极高，涉及用户查看订单、商家处理订单、运营分析订单等多个环节。随着订单量的不断增长，原始的订单查询 SQL 语句可能会变得越来越慢，直接影响用户体验和运营效率。本章将围绕一个电商平台的订单查询场景，探讨如何通过 SQL 性能优化手段，提升查询效率。我们将模拟一个订单表，并逐步分析如何优化针对该表的常见查询操作。假设我们有一个名为 orders 的表，包含订单 ID、用户 ID、订单金额、下单时间、订单状态等字段。常见的查询需求包括：根据用户 ID 查询订单、根据订单状态和下单时间范围查询订单、统计不同用户的订单总金额等。这些查询如果未经过优化，在数据量较大时，可能会非常耗时。因此，本章的目标是学习如何识别查询瓶颈，并通过创建合适的索引、理解执行计划等方式，显著提升这些查询的性能。

7.2. 数据准备：创建订单表并插入模拟数据

为了进行性能优化实践，我们首先需要创建订单表并插入足够的模拟数据。以下 SQL 语句用于创建 orders 表，并插入 100 万条模拟订单数据。这些数据将用于后续的性能测试和优化对比。

```sql
-- 创建订单表
CREATE TABLE orders (
    order_id INT AUTO_INCREMENT PRIMARY KEY, -- 订单 ID，主键
    user_id INT NOT NULL,                    -- 用户 ID
    amount DECIMAL(10, 2) NOT NULL,          -- 订单金额
    order_time DATETIME NOT NULL,            -- 下单时间
    status VARCHAR(20) NOT NULL,             -- 订单状态（如：'待支付', '已支付',
    INDEX idx_user_id (user_id),             -- 用户 ID 索引
    INDEX idx_order_time (order_time),       -- 下单时间索引
    INDEX idx_status (status)                -- 订单状态索引
) ENGINE=InnoDB DEFAULT CHARSET=utf8mb4;

-- 插入 100 万条模拟订单数据
DELIMITER //
CREATE PROCEDURE InsertOrders()
BEGIN
    DECLARE i INT DEFAULT 0;
    DECLARE random_user_id INT;
    DECLARE random_amount DECIMAL(10,2);
    DECLARE random_order_time DATETIME;
    DECLARE random_status VARCHAR(20);
    DECLARE status_index INT;

    WHILE i < 1000000 DO
        SET random_user_id = FLOOR(1 + RAND() * 100000); -- 假设有 10 万用户
        SET random_amount = ROUND(RAND() * 1000, 2);     -- 订单金额 0-1000
        SET random_order_time = DATE_ADD('2023-01-01 00:00:00', INTERVAL FLC
        SET status_index = FLOOR(1 + RAND() * 5);        -- 5 种订单状态
        CASE status_index
            WHEN 1 THEN SET random_status = '待支付';
            WHEN 2 THEN SET random_status = '已支付';
            WHEN 3 THEN SET random_status = '已发货';
            WHEN 4 THEN SET random_status = '已完成';
```

这段 SQL 代码首先创建了一个名为 orders 的表，包含了订单的基本信息字段，并为 user_id、order_time 和 status 字段创建了索引，这是后续性能优化的基础。然后，通过一个存储过程 InsertOrders 向表中插入了 100 万条模拟订单数据。数据插入过程中，用户 ID、订单金额、下单时间和订单状态均为随机生成，以模拟真实场景。订单金额在 0 到 1000 之间随机，下单时间在 2023 年内随机，订单状态从'待支付'、'已支付'、'已发货'、'已完成'、'已取消'中随机选择。通过执行这个存储过程，我们可以得到一个具有一定数据量的订单表，用于后续的性能测试和优化实践。在实际应用中，数据量可能远大于此，但 100 万条数据足以演示性能问题及优化效果。

7.3. 任务拆解：识别瓶颈、优化查询、验证效果

本章的核心任务是优化电商平台的订单查询性能。我们将通过以下几个递进的问题来逐步完成这个任务：

1. **如何识别查询瓶颈？**
    a. 问题：当发现某个订单查询 SQL 执行缓慢时，如何定位性能瓶颈？
    b. 方法：学习使用 EXPLAIN 命令分析 SQL 语句的执行计划，理解执行计划中的关键信息，如访问类型（type）、可能使用的索引（possible_keys）、实际使用的索引（key）、扫描行数（rows）等，从而判断查询是否有效利用了索引，是否存在全表扫描等低效操作。
    c. 目标：能够通过执行计划初步判断查询慢的原因。
2. **如何针对单字段查询进行优化？**
    a. 问题：对于常见的单字段查询，例如根据 user_id 查询订单，如何通过索引优化提升查询速度？
    b. 方法：分析针对 user_id 的查询语句，确认现有索引是否被有效使用。如果索引未被使用或使用不当，考虑调整查询语句或索引策略。学习如何选择合适的索引类型。
    c. 目标：掌握单字段查询的索引优化方法，并理解索引对查询性能的影响。
3. **如何优化多条件组合查询？**
    a. 问题：对于包含多个查询条件的 SQL 语句，例如根据 status 和 order_time 范围查询订单，如何设计索引以提升查询效率？

b. 方法：分析多条件查询的特点，学习创建复合索引（Composite Index）来优化这类查询。理解复合索引的列顺序对查询性能的影响，以及最左前缀原则。

c. 目标：掌握复合索引的创建和使用，提升多条件查询的性能。

4. **如何优化排序和分组操作？**

a. 问题：当查询中包含ORDER BY或GROUP BY子句时，如何避免文件排序（filesort）等低效操作？

b. 方法：分析 ORDER BY 和 GROUP BY 操作对索引的依赖，学习如何通过创建合适的索引来优化排序和分组性能。理解索引在排序和分组中的作用机制。

c. 目标：掌握利用索引优化排序和分组操作的方法。

5. **如何验证优化效果并进行权衡？**

a. 问题：在实施了索引优化等措施后，如何量化评估优化效果？索引是否越多越好？

b. 方法：通过对比优化前后的查询执行时间、执行计划的变化来验证优化效果。同时，理解索引的维护成本，以及不当使用索引可能带来的负面影响，如插入、更新、删除操作的性能下降。

c. 目标：能够评估优化效果，并理解索引使用的权衡。

通过解决以上问题，读者将能够系统地学习和实践 SQL 性能优化的核心方法，并将其应用于电商订单查询等实际业务场景中。

7.4. SQL 优化实战：从执行计划到索引策略

在本节中，我们将通过具体的SQL查询示例，结合EXPLAIN命令，逐步分析查询性能瓶颈，并探讨如何通过调整索引策略来优化查询。我们将使用上一节创建的包含 100 万条模拟订单数据的 orders 表。

任务 1：识别查询瓶颈 - 使用 EXPLAIN 分析执行计划

假设我们有一个查询需求：查找用户 ID 为 12345 的所有订单。

```
-- 示例查询 1：根据用户 ID 查询订单
SELECT * FROM orders WHERE user_id = 12345;
```

在没有任何优化的情况下，如果 user_id 字段上没有索引，数据库可能需要进行全表扫描来找到符合条件的订单，这在数据量大的情况下会非常慢。我们可以使用 EXPLAIN 命令来查看该查询的执行计划：

```
EXPLAIN SELECT * FROM orders WHERE user_id = 12345;
```

EXPLAIN 的结果可能如下（具体输出取决于数据库版本和优化器）：

id	select_type	table	partitions	type	possible_keys	key	key_len	ref
1	SIMPLE	orders	NULL	ref	idx_user_id	idx_user_id	4	const

结果解读：

- type: ref: 表示查询使用了非唯一性索引扫描。这是一个相对较好的访问类型，说明索引被用上了。
- possible_keys: idx_user_id: 表示查询可能使用的索引是 idx_user_id。
- key: idx_user_id: 表示查询实际使用的索引是 idx_user_id。这与我们在建表时为 user_id 字段创建的索引一致。
- rows: 10: 表示 MySQL 估计需要扫描大约 10 行数据来找到结果。这个值越小越好。
- Extra: Using where: 表示在存储引擎层检索行后，还需要在服务器层进行过滤。

在这个例子中，由于 user_id 字段上已经存在索引 idx_user_id，查询能够有效地利用该索引。如果 possible_keys 为 NULL 或者 key 为 NULL，则说明查询没有使用到索引，可能进行了全表扫描，这时就需要考虑创建或优化索引。

任务 2：优化单字段查询 - 确认索引有效性

对于单字段查询，如果该字段上没有索引，或者索引未被有效使用，查询性能通常会较差。我们已经看到 user_id 上的索引被有效使用了。现在，假设我们需要根据 order_time 查询某个时间点之后的订单：

```
-- 示例查询 2：根据下单时间范围查询订单
SELECT * FROM orders WHERE order_time > '2023-06-01 00:00:00';
```

查看该查询的执行计划：

```
EXPLAIN SELECT * FROM orders WHERE order_time > '2023-06-01 00:00:00';
```

执行计划可能如下：

id	select_type	table	partitions	type	possible_keys	key	key_len	r
1	SIMPLE	orders	NULL	range	idx_order_time	idx_order_time	5	N L

结果解读：

- type: range: 表示查询使用了索引范围扫描。这通常发生在对索引列进行范围查询（如>、<、BETWEEN）时。
- key: idx_order_time: 表示实际使用了 order_time 字段上的索引 idx_order_time。
- rows: 500000: 估计扫描的行数。由于是范围查询，且时间范围可能覆盖大量数据，这个值可能比较大。

在这个例子中，order_time 字段上的索引也被有效使用了。如果这个查询仍然较慢，可能是因为满足条件的数据量本身就很大，即使使用了索引，也需要从磁盘读取大量数据。此时，可以考虑是否可以通过更精确的时间范围或者其他条件来缩小结果集。

任务 3：优化多条件组合查询 - 使用复合索引

当查询条件涉及多个字段时，单字段索引可能无法达到最佳效果。例如，我们需要查询某个用户在特定时间之后已支付的订单：

```
-- 示例查询 3：根据用户 ID、订单状态和下单时间范围查询订单
SELECT * FROM orders
WHERE user_id = 12345
  AND status = '已支付'
  AND order_time > '2023-06-01 00:00:00';
```

首先，我们查看当前索引情况下的执行计划：

```
EXPLAIN SELECT * FROM orders
WHERE user_id = 12345
  AND status = '已支付'
  AND order_time > '2023-06-01 00:00:00';
```

执行计划可能显示使用了 idx_user_id，但对 status 和 order_time 的过滤可能在 Using where 阶段进行，这意味着索引并没有完全覆盖所有查询条件。为了优化这类查询，我们可以创建一个复合索引。一个常见的策略是根据过滤性（Cardinality）高的字段优先，并结合最左前缀原则。假设 user_id 的过滤性较高，其次是 status，然后是

order_time，我们可以创建如下复合索引：

```
CREATE INDEX idx_user_status_time ON orders (user_id, status, order_time);
```

创建索引后，再次查看执行计划：

```
EXPLAIN SELECT * FROM orders
WHERE user_id = 12345
  AND status = '已支付'
  AND order_time > '2023-06-01 00:00:00';
```

此时，执行计划可能会显示使用了新创建的复合索引 idx_user_status_time，并且 type 可能是 ref 或 range，rows 的数量也会显著减少，因为索引能够更精确地定位到所需数据。

任务 4：优化排序和分组操作 - 利用索引避免文件排序

当查询中包含 ORDER BY 或 GROUP BY 时，如果排序或分组的字段没有索引，或者索引使用不当，数据库可能需要进行文件排序（filesort），这是一个相对耗时的操作。例如，我们需要查询某个用户的所有订单，并按下单时间降序排列：

```
-- 示例查询 4：根据用户 ID 查询订单并按下单时间降序排列
SELECT * FROM orders WHERE user_id = 12345 ORDER BY order_time DESC;
```

如果只有 user_id 上的单列索引 idx_user_id，执行计划可能会显示 Extra: Using filesort。为了优化这个查询，我们可以创建一个包含 user_id 和 order_time 的复合索引，并且注意 ORDER BY 的顺序：

```
CREATE INDEX idx_user_id_order_time_desc ON orders (user_id, order_time DESC);
```

在某些数据库系统中（如 MySQL 8.0+），支持降序索引。创建此索引后，数据库可以直接利用索引的有序性来避免文件排序。再次执行 EXPLAIN，Extra 中应该不再有 Using filesort。

类似地，对于 GROUP BY 操作，如果分组字段没有索引，也可能导致临时表和文件排序。例如，统计每个用户的订单数量：

```
-- 示例查询 5：统计每个用户的订单数量
SELECT user_id, COUNT(*) AS order_count FROM orders GROUP BY user_id;
```

如果 user_id 上有索引（如 idx_user_id），这个查询通常可以利用索引进行分组，避免全表扫描和文件排序。执行计划中的 type 可能是 index 或 range，Extra 可能是 Using index。

任务 5：验证优化效果并进行权衡

在进行了上述索引优化后，我们需要验证优化效果。最直接的方法是对比优化前后查询的执行时间。可以使用数据库提供的性能分析工具，或者简单地记录查询开始和结束的时间。

例如，在执行示例查询 3 之前和创建复合索引 idx_user_status_time 之后，分别执行查询并记录时间。通常，在数据量较大的情况下，正确的索引可以带来数量级的性能提升。

然而，索引并非越多越好。每个索引都会占用额外的存储空间，并且在数据插入、更新和删除时，索引也需要维护，这会带来额外的开销。因此，在创建索引时需要权衡利弊：

- **选择高选择性的列作为索引**：选择性高的列（即列中不同值的数量与总行数的比例较高）作为索引，过滤效果更好。
- **避免在频繁更新的列上创建过多索引**：频繁更新的列会导致索引频繁重建，影响写性能。
- **考虑复合索引的列顺序**：遵循最左前缀原则，将查询中最常使用且选择性高的列放在复合索引的左边。
- **定期审查和优化索引**：随着业务发展，查询模式可能发生变化，需要定期审查现有索引的有效性，删除不再使用或效果不佳的索引。

通过实际测试和权衡，可以找到最适合当前业务场景的索引策略。

7.5. 本章总结：性能优化核心要点

本章通过电商订单查询的场景，系统介绍了 SQL 性能优化的核心方法和实践步骤。我们首先学习了如何通过 EXPLAIN 命令分析 SQL 语句的执行计划，识别查询瓶颈，例如全表扫描、未使用索引等问题。接着，针对单字段查询、多条件组合查询、以及包含排序和分组的查询，探讨了如何设计和应用索引进行优化。特别是对于复合索引的创建和使用，以及其在提升多条件查询和避免文件排序方面的作用，进行了详细的讲解。最后，我们强调了验证优化效果的重要性，并讨论了索引使用的权衡，提醒读者索引并非越多越好，需要根据实际业务需求和数据特点进行合理设计。

核心知识点回顾：

1. **理解执行计划**：EXPLAIN 是性能分析的起点，通过解读其输出信息（如 type, key, rows, Extra），可以判断查询的执行效率。
2. **索引是核心**：合理使用索引是提升查询性能最有效的手段之一。索引可以大大减少数据库需要扫描的数据量。
3. **单字段索引与复合索引**：根据查询条件选择合适的索引类型。复合索引在处理多条

件查询时尤为有效，但需要注意列的顺序和最左前缀原则。

4. **优化排序和分组**：通过创建包含排序或分组字段的索引，可以避免耗时的文件排序操作，提升 ORDER BY 和 GROUP BY 的性能。

5. **权衡索引的利弊**：索引会带来存储和维护开销，需要在高查询性能和高写操作效率之间进行权衡。

通过本章的学习，读者应能掌握 SQL 性能优化的基本思路和方法，并能够将其应用于实际工作中，解决常见的查询性能问题。

7.6. 小练习：动手优化查询

为了巩固本章所学知识，请尝试完成以下练习：

1. **分析执行计划**：
    a. 针对 orders 表，编写一个查询：查找订单状态为'已发货'且订单金额大于 500 的订单，并按订单金额降序排列。
    b. 使用 EXPLAIN 分析该查询的执行计划，观察是否使用了索引，是否存在文件排序等问题。
2. **优化单字段查询**：
    a. 假设经常需要根据 status 字段查询订单，但该字段的选择性不高（即有很多重复值）。思考在这种情况下，仅为 status 字段创建索引是否总是有效的？为什么？
3. **设计复合索引**：
    a. 针对练习 1 中的查询（查找订单状态为'已发货'且订单金额大于 500 的订单，并按订单金额降序排列），设计一个你认为合适的复合索引，并解释为什么这么设计。
    b. 创建你设计的索引，然后再次使用 EXPLAIN 分析查询，观察执行计划的变化。
4. **优化排序操作**：
    a. 编写一个查询：统计每个订单状态下的订单总数，并按订单总数降序排列。
    b. 分析该查询是否需要文件排序，如果需要，如何通过索引优化？
5. **思考索引的代价**：
    a. 如果在 orders 表的 amount（订单金额）字段上创建一个索引，会对哪些操作产生影响（例如：根据 amount 查询、插入新订单、更新订单金额）？请分别说明。

通过动手实践这些练习，你将更深入地理解 SQL 性能优化的各个方面。

7.7. 常见错误清单：性能优化中的坑

在进行 SQL 性能优化时，初学者常会遇到一些误区或犯一些常见的错误。了解这些“坑”有助于避免不必要的麻烦：

1. **盲目创建索引**：认为索引越多越好，不考虑索引的维护成本和实际查询需求。结果导致写操作变慢，存储空间浪费。
    a. **避坑指南**：根据实际查询频率和过滤效果来创建索引。定期审查并删除无用或低效的索引。
2. **忽略执行计划**：不习惯使用 EXPLAIN 分析查询，仅凭感觉判断查询效率。
    a. **避坑指南**：养成查看执行计划的习惯，它是理解查询如何执行的关键。
3. **复合索引列顺序不当**：创建复合索引时，没有考虑最左前缀原则，或者没有将高选择性的列放在前面，导致索引无法被有效利用。
    a. **避坑指南**：仔细设计复合索引的列顺序，确保其能够覆盖常见的查询模式。
4. **在选择性低的列上创建索引**：例如，在性别、状态等只有少数几个取值的列上创建独立索引，效果通常不佳。
    a. **避坑指南**：这类列更适合作为复合索引的一部分，或者与其他高选择性列结合使用。
5. **过度依赖工具自动优化**：完全依赖数据库的查询优化器，而不主动思考如何编写高效的 SQL 和设计合理的索引。
    a. **避坑指南**：理解优化器的工作原理，但也要主动参与优化过程，编写更优的 SQL 语句。
6. **忽略数据类型隐式转换**：在查询条件中，如果字段类型与传入值类型不匹配，可能导致索引失效。例如，WHERE char_column = 123（char_column 是字符类型，123 是数字）。
    a. **避坑指南**：确保查询条件中的数据类型与字段定义的类型一致。
7. **使用 SELECT ***：查询所有列，即使有些列并不需要。这会增加 I/O 开销和网络传输量。
    a. **避坑指南**：只选择需要的列。
8. **频繁提交小事务**：在循环中执行数据库操作并频繁提交，而不是批量处理，会增加事务开销。
    a. **避坑指南**：尽量使用批量操作，减少事务提交次数。

9. **不关注连接池配置**：在高并发应用中，数据库连接的创建和销毁开销很大。不合理的连接池配置会导致性能瓶颈。
    a. **避坑指南**：合理配置连接池参数，如最大连接数、最小连接数、超时时间等。
10. **忘记更新统计信息**：数据库优化器依赖统计信息来生成执行计划。如果数据分布发生较大变化，而统计信息没有及时更新，可能导致优化器选择次优的执行计划。
    a. **避坑指南**：定期更新表的统计信息（具体命令取决于数据库系统）。

避免这些常见错误，结合本章学习的优化方法，将有助于你更有效地提升 SQL 查询性能。

3.2. 第 8 章：医院科室就诊分析 (复杂逻辑实现)

业务场景：分析医院各科室的就诊情况，为医疗资源调配和科室管理提供数据支持。

8.1. 数据准备

为了进行医院科室就诊情况的分析，我们需要准备相关的数据表。本章案例将涉及三个核心表：departments（科室表）、patients（患者表）和 visits（就诊记录表）。这些表通过外键关联，共同构成了描述患者就诊行为的数据模型。

8.1.1. 表结构

- **departments（科室表）**：该表存储医院各个科室的基本信息。
    - dept_id (INTEGER, PRIMARY KEY): 科室的唯一标识符，作为主键。
    - dept_name (TEXT): 科室的名称，如“内科”、“外科”等。

这个表相对简单，主要用于维护科室的元数据，并作为其他表（如就诊记录表）的参照。

- **patients（患者表）**：该表存储患者的基本信息。
    - patient_id (INTEGER, PRIMARY KEY): 患者的唯一标识符，作为主键。
    - name (TEXT): 患者的姓名。

在实际系统中，患者表可能包含更多信息，如性别、年龄、联系方式等，但为简化案例，此处仅保留核心字段。

- **visits（就诊记录表）**：该表是核心的事实表，记录了每一次患者就诊的详细信息。
  - visit_id (INTEGER, PRIMARY KEY): 就诊记录的唯一标识符，作为主键。
  - patient_id (INTEGER): 就诊患者的 ID，外键关联到 patients 表的 patient_id 字段。
  - dept_id (INTEGER): 就诊科室的 ID，外键关联到 departments 表的 dept_id 字段。
  - visit_date (DATE): 就诊发生的日期。
  - diagnosis (TEXT): 初步诊断结果或主要症状描述。

这张表通过 patient_id 和 dept_id 将患者信息、科室信息和就诊行为关联起来，是进行分析的主要数据来源。

以下是创建这些表的 SQL 语句：

```
CREATE TABLE departments (
    dept_id INTEGER PRIMARY KEY,
    dept_name TEXT
);

CREATE TABLE patients (
    patient_id INTEGER PRIMARY KEY,
    name TEXT
);

CREATE TABLE visits (
    visit_id INTEGER PRIMARY KEY,
    patient_id INTEGER,
    dept_id INTEGER,
    visit_date DATE,
    diagnosis TEXT,
    FOREIGN KEY (patient_id) REFERENCES patients(patient_id),
    FOREIGN KEY (dept_id) REFERENCES departments(dept_id)
);
```

8.1.2. 模拟数据

为了演示后续的 SQL 查询和分析，我们向上述三个表中插入了模拟数据。具体来说，我们插入了：

- **科室表（departments）**：5 个科室，包括内科、外科、妇产科、儿科、眼科。
- **患者表（patients）**：5 位患者，分别为张三、李四、王五、赵六、钱七。
- **就诊记录表（visits）**：共 10 条就诊记录，这些记录分布在不同的日期，涉及不同的患者和科室，并包含了简化的诊断信息。例如，张三于 2025 年 7 月 1 日在内科就诊，诊断为感冒；李四于 2025 年 7 月 2 日在内科就诊，诊断为发烧等。

以下是插入模拟数据的 SQL 语句：

```sql
INSERT INTO departments (dept_id, dept_name) VALUES
(1, '内科'),
(2, '外科'),
(3, '妇产科'),
(4, '儿科'),
(5, '眼科');

INSERT INTO patients (patient_id, name) VALUES
(1, '张三'),
(2, '李四'),
(3, '王五'),
(4, '赵六'),
(5, '钱七');

INSERT INTO visits (visit_id, patient_id, dept_id, visit_date, diagnosis) VALUES
(1, 1, 1, '2025-07-01', '感冒'),
(2, 2, 1, '2025-07-02', '发烧'),
(3, 3, 1, '2025-07-08', '肺炎'),
(4, 4, 2, '2025-07-10', '骨折'),
(5, 5, 2, '2025-07-12', '阑尾炎'),
(6, 1, 3, '2025-07-15', '产检'),
(7, 2, 4, '2025-07-18', '感冒'),
(8, 3, 4, '2025-07-20', '发烧'),
(9, 4, 5, '2025-07-22', '近视'),
(10, 5, 5, '2025-07-25', '结膜炎');
```

这些模拟数据虽然规模较小，但足以支撑本章所要演示的多表关联查询和分组统计等复杂逻辑的实现。通过操作这些贴近真实业务场景的数据，学习者可以更好地理解 SQL 在解决实际问题中的应用。

8.2. 任务拆解

为了全面分析医院各科室的就诊情况，我们将业务需求拆解为以下几个具体的 SQL 查询任务。这些任务由浅入深，逐步引导学习者运用所学的 SQL 知识解决实际问题。

8.2.1. 任务 1：统计各科室的就诊人数

业务需求：了解各个科室在一定时期内的接诊工作量，以便评估科室的运营压力、合理分配医疗资源（如医护人员、设备等）。这对于医院管理者进行科室绩效考核和资源规划至关重要。

SQL 代码：

```
SELECT
    d.dept_id,              -- 选择科室 ID
    d.dept_name,            -- 选择科室名称
    COUNT(v.visit_id) AS visit_count   -- 统计就诊记录数量，并命名为 visit_count
FROM
    departments d           -- 主表为科室表
LEFT JOIN
    visits v                -- 左连接就诊记录表
ON
    d.dept_id = v.dept_id -- 连接条件为科室 ID 相等
GROUP BY
    d.dept_id               -- 按科室 ID 进行分组
ORDER BY
    visit_count DESC;     -- 按就诊人数降序排列
```

执行结果：

dept_id	dept_name	visit_count
1	内科	3
2	外科	2
4	儿科	2
5	眼科	2
3	妇产科	1

结果解释：从查询结果可以看出，在模拟数据中，内科的就诊人次最多，达到了 3 次。其次是外科、儿科和眼科，各有 2 次就诊。妇产科的就诊人次最少，为 1 次。这个结果清晰地反映了各个科室在特定时间段内的工作负荷。需要注意的是，这里使用了 LEFT JOIN，这意味着即使某个科室在 visits 表中没有对应的就诊记录（即就诊人数为 0），它仍然会出现在结果集中，并且 visit_count 为 0。如果使用 INNER JOIN，则只会返回那些至少有一条就诊记录的科室。

8.2.2. 任务 2：找出访问内科的患者姓名和就诊日期

业务需求：获取特定科室（本例为内科）的就诊患者详细信息，包括患者姓名和具体的就诊日期。这类查询常用于追踪特定疾病的患者流向、进行患者回访或分析特定科室的患者构成。

SQL 代码：

```sql
SELECT
    p.name AS patient_name, -- 选择患者姓名，并命名为 patient_name
    v.visit_date            -- 选择就诊日期
FROM
    patients p              -- 主表为患者表
JOIN
    visits v                -- 内连接就诊记录表
ON
    p.patient_id = v.patient_id -- 连接条件为患者 ID 相等
JOIN
    departments d           -- 内连接科室表
ON
    v.dept_id = d.dept_id   -- 连接条件为科室 ID 相等
WHERE
    d.dept_name = '内科';    -- 筛选条件为科室名称为'内科'
```

执行结果：

patient_name	visit_date
张三	2025-07-01
李四	2025-07-02
王五	2025-07-08

结果解释：查询结果显示，在模拟数据中，共有三位患者在内科就诊过。分别是张三（就诊日期 2025-07-01 ）、李四（就诊日期 2025-07-02 ）和王五（就诊日期

2025-07-08 ）。这个查询涉及到了三张表的连接：patients、visits 和 departments。通过两次 JOIN 操作，我们将患者信息、就诊信息和科室信息关联起来，然后通过 WHERE 子句筛选出特定科室（内科）的就诊记录。这种多表关联查询是数据库应用中非常常见的操作。

8.2.3. 任务 3：分析各科室的疾病种类数量

业务需求：统计各个科室所接诊的不同疾病种类的数量。这有助于了解各个科室的业务范围、疾病谱的多样性，以及评估科室的专业特长和应对复杂病例的能力。

SQL 代码：

```
SELECT
    d.dept_name,                                -- 选择科室名称
    COUNT(DISTINCT v.diagnosis) AS disease_count -- 统计不同诊断的数量，并命名
FROM
    departments d                               -- 主表为科室表
JOIN
    visits v                                    -- 内连接就诊记录表
ON
    d.dept_id = v.dept_id                       -- 连接条件为科室 ID 相等
GROUP BY
    d.dept_name                                 -- 按科室名称进行分组
ORDER BY
    disease_count DESC;                         -- 按疾病种类数量降序排列
```

执行结果：

dept_name	disease_count
内科	3
眼科	2
外科	2
儿科	2
妇产科	1

结果解释：查询结果显示，内科接诊的疾病种类最多，达到了 3 种（根据模拟数据，可能是感冒、发烧、肺炎）。眼科、外科和儿科各接诊了 2 种不同的疾病。妇产科接诊的疾病种类最少，为 1 种。这个查询的关键在于使用了 COUNT(DISTINCT v.diagnosis)。DISTINCT 关键字确保了在统计数量时，每种诊断只被计算一次，即使同一种诊断在同一个科室出现了多次。这对于准确评估科室处理的疾病多样性非常重要。如果省略 DISTINCT，则统计的是所有诊断记录的总数，而不是不同诊断的种类数。

8.3. 本章总结

本章通过一个医院科室就诊分析的案例，深入探讨了如何运用 SQL 进行多表关联查询和复杂的分组统计。我们从数据准备入手，构建了科室、患者和就诊记录三张核心数据表，并插入了模拟数据以支持后续的 SQL 操作。随后，我们将复杂的业务需求拆解为三个具体的 SQL 任务：统计各科室的就诊人数、找出访问特定科室的患者信息、以及分析各科室的疾病种类数量。每个任务都通过清晰的 SQL 代码、注释、执行结果和结果解释进行了详细阐述，帮助学习者理解每一步操作的意图和效果。

一分钟总结：

- **业务场景**：分析医院各科室的就诊数据，支持资源调配和管理决策。
- **核心技能**：多表连接（JOIN）、分组聚合（GROUP BY，COUNT，SUM）、条件筛

选 (WHERE)。

- **关键函数**：COUNT(DISTINCT column) 用于统计不重复值的数量。
- **表关系**：理解主键、外键以及表之间的关联关系是进行多表查询的基础。
- **数据解读**：能够根据 SQL 查询结果，解读其业务含义，并从中发现问题或趋势。

练习：

1. **统计每位患者就诊的科室数量**：编写 SQL 查询，列出每位患者的姓名以及他们曾经就诊过的不同科室的数量。这将涉及到对 patients 表和 visits 表的连接，并按患者分组后使用 COUNT(DISTINCT dept_id)。
2. **找出同时就诊于内科和外科的患者**：这个任务相对复杂，可能需要使用子查询或者多次连接。一种思路是先分别找出就诊于内科的患者 ID 集合和就诊于外科的患者 ID 集合，然后取这两个集合的交集。另一种思路是使用自连接或者窗口函数。
3. **分析各科室每月就诊人数的变化趋势**：在 visits 表中增加一个 visit_month 字段（或者从 visit_date 中提取月份），然后统计每个科室在每个月的就诊总人数，并按科室和月份排序。这将涉及到按多个字段进行分组 (GROUP BY dept_id, visit_month)。

常见错误：

- **LEFT JOIN 与 INNER JOIN 的误用**：在任务 1 中，如果错误地使用了 INNER JOIN 来连接 departments 和 visits 表，那么那些没有就诊记录的科室将不会出现在结果集中，导致统计信息不完整。理解不同类型连接（INNER JOIN, LEFT JOIN, RIGHT JOIN, FULL JOIN）的语义和区别至关重要。
- **COUNT(DISTINCT column) 使用不当**：在任务 3 中，如果遗漏了 DISTINCT 关键字，写成 COUNT(v.diagnosis)，那么统计的将是每个科室所有诊断记录的总数，而不是不同诊断的种类数。这会导致对科室疾病多样性的错误评估。
- **连接条件错误或遗漏**：在进行多表连接时，必须确保连接条件 (ON 子句) 是正确的，并且能够准确地关联相关的行。如果连接条件写错或者遗漏，可能会导致笛卡尔积（Cartesian product），产生大量无意义的中间结果，严重消耗系统资源并返回错误数据。
- **分组字段选择不当**：在使用 GROUP BY 子句时，SELECT 子句中非聚合列（如 dept_name）必须出现在 GROUP BY 子句中，否则会导致语法错误或返回不明确的结果。确保理解 GROUP BY 的工作原理。

3.3. 第 9 章：金融投资组合风险评估 (高级分析与窗口函数进阶)

业务场景：评估不同金融投资组合的风险水平，为资产配置、风险控制和投资决策提供数据支持。

9.1. 数据准备

为了进行金融投资组合的风险评估，我们需要构建一个包含投资组合信息、证券（如股票、债券）信息以及各组合持仓情况的数据模型。本章案例将使用三个核心数据表：portfolios（投资组合表）、securities（证券表）和 holdings（持仓表）。

9.1.1. 表结构

- **portfolios（投资组合表）**：该表存储不同投资组合的基本信息和总体风险偏好。
    - portfolio_id (INTEGER, PRIMARY KEY): 投资组合的唯一标识符，作为主键。
    - name (TEXT): 投资组合的名称，例如“高风险组合”、“中等风险组合”等。
    - value (DECIMAL(15, 2)): 投资组合的总市值或初始价值。这是一个数值型字段，能够存储较大金额并保留两位小数。
    - risk_level (TEXT): 投资组合预设的风险等级，如“高”、“中”、“低”。这个字段可以用于快速筛选符合特定风险偏好的组合。

这张表是评估的起点，定义了我们要分析的投资对象。

- **securities（证券表）**：该表存储市场上可供投资的各类证券的详细信息，特别是其风险特征。
    - security_id (INTEGER, PRIMARY KEY): 证券的唯一标识符，作为主键。
    - ticker (TEXT): 证券的交易代码，如股票代码 "AAPL"。
    - name (TEXT): 证券的正式名称，如公司全称 "苹果公司"。
    - price (DECIMAL(10, 2)): 证券的当前市场价格。
    - risk_factor (DECIMAL(5, 2)): 衡量该证券风险程度的系数。这个系数可以是基于历史波动率、Beta 值或其他风险评估模型计算得出的。数值越大，代表风险越高。

这张表提供了评估单个证券风险以及计算组合整体风险所需的关键参数。

- **holdings（持仓表）**：该表是连接投资组合和证券的桥梁，记录了每个投资组合持有各种证券的数量。
    - portfolio_id (INTEGER): 投资组合的 ID，外键关联到 portfolios 表的 portfolio_id 字段。
    - security_id (INTEGER): 证券的 ID，外键关联到 securities 表的 security_id 字段。
    - shares (INTEGER): 该投资组合持有该证券的股数或份额。
    - PRIMARY KEY (portfolio_id, security_id): 将 portfolio_id 和 security_id 联合作为主键，确保每个组合对每种证券的持仓记录是唯一的。

这张表是进行风险聚合计算的核心，通过它可以知道每个组合都投了哪些证券，以及投了多少。

以下是创建这些表的 SQL 语句：

```
CREATE TABLE portfolios (
    portfolio_id INTEGER PRIMARY KEY,
    name TEXT,
    value DECIMAL(15, 2),
    risk_level TEXT
);

CREATE TABLE securities (
    security_id INTEGER PRIMARY KEY,
    ticker TEXT,
    name TEXT,
    price DECIMAL(10, 2),
    risk_factor DECIMAL(5, 2)
);

CREATE TABLE holdings (
    portfolio_id INTEGER,
    security_id INTEGER,
    shares INTEGER,
    PRIMARY KEY (portfolio_id, security_id),
    FOREIGN KEY (portfolio_id) REFERENCES portfolios(portfolio_id),
    FOREIGN KEY (security_id) REFERENCES securities(security_id)
);
```

9.1.2. 模拟数据

为了演示后续的 SQL 查询和风险评估，我们向上述三个表中插入了模拟数据。具体来说：

- **投资组合表 (portfolios)**：插入了 3 个投资组合，分别是“高风险组合”（总价值 1,000,000，风险等级高）、“中等风险组合”（总价值 500,000，风险等级中）和“低风险组合”（总价值 300,000，风险等级低）。

- **证券表 (securities)**：插入了 5 种证券，包括苹果（AAPL)、微软（MSFT)、特斯拉(TSLA)、谷歌（GOOGL）和一种债券（BND)。每种证券都有其当前价格和一个模拟的风险系数。例如，AAPL 价格 150，风险系数 0.15；BND 价格 50，风险系数 0.05。

- **持仓表 (holdings)**：记录了这 3 个投资组合对这 5 种证券的持仓情况。例如，“高风险组合”持有 1000 股 AAPL、500 股 MSFT 和 200 股 TSLA。“中等风险组合”持有 800 股 MSFT、100 股 GOOGL 和 1000 股 BND。“低风险组合”持有 3000 股 BND、500 股 AAPL 和 100 股 GOOGL。

以下是插入模拟数据的 SQL 语句：

```sql
INSERT INTO portfolios (portfolio_id, name, value, risk_level) VALUES
(1, '高风险组合', 1000000.00, '高'),
(2, '中等风险组合', 500000.00, '中'),
(3, '低风险组合', 300000.00, '低');

INSERT INTO securities (security_id, ticker, name, price, risk_factor) VALUES
(1, 'AAPL', '苹果', 150.00, 0.15),
(2, 'MSFT', '微软', 300.00, 0.10),
(3, 'TSLA', '特斯拉', 700.00, 0.25),
(4, 'GOOGL', '谷歌', 2500.00, 0.20),
(5, 'BND', '债券', 50.00, 0.05);

INSERT INTO holdings (portfolio_id, security_id, shares) VALUES
(1, 1, 1000), -- 高风险组合持有 AAPL
(1, 2, 500),  -- 高风险组合持有 MSFT
(1, 3, 200),  -- 高风险组合持有 TSLA
(2, 2, 800),  -- 中等风险组合持有 MSFT
(2, 4, 100),  -- 中等风险组合持有 GOOGL
(2, 5, 1000), -- 中等风险组合持有 BND
(3, 5, 3000), -- 低风险组合持有 BND
(3, 1, 500),  -- 低风险组合持有 AAPL
(3, 4, 100);  -- 低风险组合持有 GOOGL
```

这些模拟数据构建了一个简单的投资场景，足以支持本章所要演示的风险评估计算，包括单个证券市值的计算、组合整体风险的评估以及识别组合内主要风险贡献者。

9.2. 任务拆解

为了全面评估金融投资组合的风险，我们将业务需求拆解为以下几个具体的 SQL 查询任务。这些任务将综合运用多表连接、聚合函数以及窗口函数等高级 SQL 特性。

9.2.1. 任务 1：计算每个投资组合中各证券的市场价值

业务需求：了解每个投资组合中，各种持仓证券的当前市场价值。这是进行后续风险评估和资产配置分析的基础。通过计算单个证券在组合中的市值，可以了解其对组合总价值的贡献程度，并识别出组合中的重仓股。

SQL 代码：

```
SELECT
    p.portfolio_id,              -- 选择投资组合 ID
    p.name AS portfolio_name, -- 选择投资组合名称，并重命名
    s.ticker,                    -- 选择证券代码
    s.name AS security_name, -- 选择证券名称，并重命名
    h.shares * s.price AS market_value -- 计算市场价值：持仓股数乘以证券价格
FROM
    portfolios p                 -- 主表为投资组合表
JOIN
    holdings h                   -- 内连接持仓表
ON
    p.portfolio_id = h.portfolio_id -- 连接条件为投资组合 ID 相等
JOIN
    securities s                 -- 内连接证券表
ON
    h.security_id = s.security_id -- 连接条件为证券 ID 相等
ORDER BY
    p.portfolio_id, s.ticker; -- 按投资组合 ID 和证券代码排序
```

执行结果：

portfolio_id	portfolio_name	ticker	security_name	market_value
1	高风险组合	AAPL	苹果	150000.00
1	高风险组合	MSFT	微软	150000.00
1	高风险组合	TSLA	特斯拉	140000.00
2	中等风险组合	MSFT	微软	240000.00
2	中等风险组合	GOOGL	谷歌	250000.00
2	中等风险组合	BND	债券	50000.00
3	低风险组合	BND	债券	150000.00
3	低风险组合	AAPL	苹果	75000.00
3	低风险组合	GOOGL	谷歌	25000.00

结果解释：查询结果清晰地展示了每个投资组合中各项持仓证券的当前市场价值。例如，在“高风险组合”中，苹果公司 (AAPL) 的持仓市值为 150,000.00，微软 (MSFT) 的持仓市值也为 150,000.00，特斯拉 (TSLA) 的持仓市值为 140,000.00。这表明该组合在苹果和微软上的投资额较大。通过这个查询，我们可以初步了解各个组合的资产分布情况，为后续的风险分析打下基础。这个查询主要涉及三张表的连接 (portfolios, holdings, securities) 和简单的数值计算 (shares * price)。

9.2.2. 任务 2：评估每个组合的整体风险

业务需求：量化每个投资组合的整体风险水平。这通常通过将组合内各证券的风险（由其市场价值和风险系数决定）进行加总或采用更复杂的模型来实现。在本案例中，我们简化处理，将各证券的“风险贡献”（市值乘以风险系数）进行加总，作为组合的整体风险度量。

SQL 代码：

```sql
SELECT
    p.portfolio_id,                                    -- 选择投资组合 ID
    p.name AS portfolio_name,                          -- 选择投资组合名称
    SUM(h.shares * s.price * s.risk_factor) AS total_risk -- 计算总风险：Σ(股数 * 价格
FROM
    portfolios p                                       -- 主表为投资组合表
JOIN
    holdings h                                         -- 内连接持仓表
ON
    p.portfolio_id = h.portfolio_id                    -- 连接条件为投资组合 ID 相等
JOIN
    securities s                                       -- 内连接证券表
ON
    h.security_id = s.security_id                      -- 连接条件为证券 ID 相等
GROUP BY
    p.portfolio_id                                     -- 按投资组合 ID 进行分组
ORDER BY
    total_risk DESC;                                   -- 按总风险降序排列
```

执行结果：

portfolio_id	portfolio_name	total_risk
2	中等风险组合	78500.00

1	高风险组合	69500.00
3	低风险组合	14250.00

结果解释：查询结果显示，根据我们简化的风险计算模型，“中等风险组合”的总风险值最高，达到了 78,500.00。其次是“高风险组合”，总风险值为 69,500.00。“低风险组合”的总风险值最低，为 14,250.00。这个结果可能有些反直觉，因为“高风险组合”的名称暗示其风险应该最高。这说明了几个问题：首先，组合的名称可能并不完全反映其实际风险；其次，风险的计算方法对结果影响很大；最后，组合的配置（即持有哪些证券以及各自的权重）是决定其整体风险的关键因素。在本例中，“中等风险组合”可能配置了更多高市值且风险系数相对较高的证券，或者其整体规模较大，导致其计算出的总风险较高。这个查询的核心是使用 SUM() 聚合函数对每个组合内所有证券的风险贡献进行累加。

9.2.3. 任务 3：找出每个组合中风险贡献最大的证券

业务需求：识别出在每个投资组合中，哪些证券对整体风险的贡献最大。这对于风险管理者来说非常重要，因为集中度过高的风险可能意味着潜在的巨大损失。通过找出这些“关键风险点”，可以针对性地进行调整，例如减持这些高风险证券或进行对冲操作。

SQL 代码：

```sql
WITH SecurityRisk AS (
    SELECT
        p.portfolio_id,
        p.name AS portfolio_name,
        s.ticker,
        s.name AS security_name,
        s.risk_factor,
        h.shares,
        h.shares * s.price * s.risk_factor AS security_risk, -- 计算单个证券的风险贡献
        RANK() OVER (PARTITION BY p.portfolio_id ORDER BY h.shares * s.price * s
    FROM
        portfolios p
    JOIN
        holdings h
    ON
        p.portfolio_id = h.portfolio_id
    JOIN
        securities s
    ON
        h.security_id = s.security_id
)
SELECT portfolio_id, portfolio_name, ticker, security_name, risk_factor, shares, secur
FROM SecurityRisk
WHERE risk_rank = 1 -- 筛选出排名为 1 的证券（即风险贡献最大的）
ORDER BY portfolio_id, security_risk DESC;
```

执行结果：

portfolio_id	portfolio_name	ticker	security_name	risk_factor	shares	security_risk

1	高风险组合	MSFT	微软	0.10	500	15000.00
1	高风险组合	AAPL	苹果	0.15	1000	15000.00
2	中等风险组合	GOOGL	谷歌	0.20	100	50000.00
3	低风险组合	BND	债券	0.05	3000	7500.00

结果解释：查询结果清晰地指出了每个投资组合中风险贡献最大的证券。在“高风险组合”中，微软 (MSFT) 和苹果 (AAPL) 的风险贡献并列最高，均为 15,000.00。这表明该组合的风险相对分散在这两只股票上，或者它们的风险贡献确实非常接近。在“中等风险组合”中，谷歌 (GOOGL) 是风险贡献最大的证券，其风险贡献高达 50,000.00，远高于组合内其他证券。在“低风险组合”中，债券 (BND) 是风险贡献最大的，但其绝对风险贡献值（7,500.00）相对较低，符合其低风险特征。这个查询的关键在于使用了窗口函数 RANK()。RANK() OVER (PARTITION BY p.portfolio_id ORDER BY h.shares * s.price * s.risk_factor DESC) 为每个投资组合 (PARTITION BY p.portfolio_id) 内的证券按照其风险贡献 (h.shares * s.price * s.risk_factor) 进行降序排名。然后，外层的 WHERE risk_rank = 1 子句筛选出每个组合中风险贡献排名第一的证券。这里使用了公共表表达式 (CTE) SecurityRisk 来先计算风险贡献和排名，然后在主查询中进行筛选，这是一种更通用的写法，尤其适用于不支持 QUALIFY 子句的数据库系统。

9.3. 总结

本章通过一个金融投资组合风险评估的案例，全面展示了 SQL 在处理复杂数据分析和计算方面的强大能力。我们从构建数据模型开始，创建了投资组合、证券和持仓三张核心表，并填充了模拟数据。随后，我们将风险评估的需求分解为三个逐步深入的 SQL 任务：计算组合内各证券的市值、评估组合的整体风险、以及识别组合中风险贡献最大的证券。每个任务都通过详细的 SQL 代码、清晰的注释、具体的执行结果和深入的结果解释进行了阐述，帮助学习者掌握多表连接、聚合函数（如 SUM）以及窗口函数（如 RANK）的综合运用。

一分钟总结：

- **业务场景**：对金融投资组合进行风险评估，识别风险来源。
- **核心技能**：多表连接（JOIN）、聚合函数（SUM）、窗口函数（RANK）、复杂计算逻辑。
- **窗口函数进阶**：RANK() OVER (PARTITION BY ... ORDER BY ...) 用于在分组内进行排名。
- **风险评估模型**：本章采用了简化的风险计算模型（市值乘以风险系数），实际应用中可能更为复杂。
- **数据驱动决策**：通过 SQL 分析，可以为投资决策和风险管理提供量化依据。

练习：

1. **计算每个组合的夏普比率（需要额外数据支持）**：夏普比率是衡量投资组合风险调整后收益的常用指标，计算公式为（组合平均收益率 - 无风险利率）/ 组合收益率的标准差。尝试设计表结构来存储历史收益率数据或无风险利率，并编写 SQL 计算夏普比率。
2. **找出各组合中风险系数最高的证券**：编写 SQL 查询，找出每个投资组合中持有的、自身风险系数（risk_factor）最高的证券。这可能需要结合窗口函数 MAX() 或 RANK()。
3. **分析组合的行业集中度风险**：如果证券表中增加“行业”字段，如何分析某个投资组合在特定行业的投资占比是否过高，从而带来行业集中度风险？

常见错误：

- **RANK 函数位置错误**：窗口函数如 RANK() 应该在 SELECT 子句中使用，并且需要配合 OVER 子句来定义窗口。不能直接在 WHERE 或 GROUP BY 子句中使用窗口函数。
- **QUALIFY 子句使用不当或不被支持**：QUALIFY 子句用于在窗口函数计算后进行筛选，非常方便，但并非所有数据库系统都支持（例如，MySQL 就不支持）。在不支持 QUALIFY 的数据库中，需要将包含窗口函数的查询作为子查询或公共表表达式 (CTE)，然后在外部查询中进行筛选，如本章示例所示。
- **连接条件错误导致数据膨胀或丢失**：在进行多表连接时，务必确保连接条件（ON 子句）的准确性。错误的连接条件可能导致产生笛卡尔积（返回过多错误数据）或丢失本应匹配的数据。

- **对风险计算模型的理解偏差**：本章采用的风险计算模型是一个高度简化的示例。在实际金融风险评估中，会使用更复杂的模型，如 VaR（风险价值）、CVaR（条件风险价值）等，并考虑资产之间的相关性。学习者应理解示例模型的局限性，并认识到实际应用的复杂性。

4. 附录篇：深化理解与拓展知识

4.1. 附录 A：索引详解（原理、类型与优化）

本附录旨在为零基础的 SQL 学习者提供一个关于数据库索引的入门级详解，涵盖索引的基本作用、常见的存储结构（以 B+树为主）、不同类型的索引及其特点、索引的优缺点，以及索引设计的基本原则和常见的索引失效场景。理解索引对于编写高效的 SQL 查询和进行数据库性能优化至关重要。索引的本质是一种帮助 MySQL 高效获取数据的数据结构，可以类比于书籍的目录，通过目录可以快速定位到所需内容，而无需逐页翻阅。在 MySQL 中，无论是 InnoDB 还是 MyISAM 存储引擎，都普遍采用 B+树作为索引的底层数据结构。B+树是一种多路平衡查找树，其特点是所有数据都存储在叶子节点，并且叶子节点之间通过指针连接，形成一个有序链表，这使得 B+树在范围查询和等值查询方面都非常高效，并且树的高度相对较低，减少了磁盘 I/O 次数。

索引的主要优点在于能够显著加快数据检索速度，减少数据库需要扫描的数据量，从而降低磁盘 I/O 成本。此外，唯一索引可以保证数据的唯一性，例如主键索引就是一种特殊的唯一索引。对于包含 ORDER BY 或 GROUP BY 子句的查询，如果相关列上建有索引，数据库可以直接利用索引已经排好序的特性，避免额外的排序操作，提升性能。然而，索引并非没有代价。创建和维护索引本身需要时间和存储空间，特别是对于大表而言，索引会占用额外的磁盘空间。更重要的是，当对表中的数据进行增、删、改（DML）操作时，相关的索引也需要动态更新和维护，这会降低 DML 操作的执行效率。因此，并非索引越多越好，需要根据实际查询需求和数据特点进行权衡。

常见的索引类型包括主键索引（Primary Key）、唯一索引（Unique Key）、普通索引（Index）、联合索引（Composite Index）和覆盖索引（Covering Index）等。主键索引是一种特殊的唯一索引，不允许有空值。唯一索引要求索引列的值必须唯一，但允许有一个空值。普通索引则没有唯一性限制，仅用于加速查询。联合索引是指在多个列上创建的索引，其查询效率通常优于对多个单列索引进行合并。覆盖索引是指一个

索引包含了查询语句所需要的所有字段，这样查询可以直接从索引中获取数据，而无需回表查询数据行，从而显著提升查询性能。在设计索引时，应遵循一些基本原则，例如为经常出现在 WHERE 子句、JOIN 条件、ORDER BY 和 GROUP BY 子句中的列创建索引；选择区分度高（即不同值多）的列作为索引；对于字符串类型的列，可以考虑使用前缀索引以节省空间；避免创建冗余索引等。

尽管索引能够提升查询性能，但在某些情况下索引可能会失效，导致查询效率低下。常见的索引失效场景包括：在索引列上进行计算、函数操作或类型转换；使用 LIKE 查询时以通配符%开头（例如 LIKE '%abc'）；查询条件中使用 OR，且 OR 前后的条件中有一个列没有索引；对于联合索引，查询未遵循最左前缀匹配原则；数据量过小，全表扫描可能比走索引更快；或者 MySQL 优化器判断使用索引的成本高于全表扫描（例如，当查询结果集占表中数据比例过大时）。因此，在编写 SQL 语句和设计索引时，需要充分考虑这些因素，以确保索引能够发挥其应有的作用。通过本附录的学习，希望学习者能够对索引有一个更全面的认识，为后续的 SQL 性能优化打下坚实的基础。

4.2. 附录 B：窗口函数详解 (语法、场景与技巧)

窗口函数（Window Function），也称为开窗函数，是 SQL 中一种极其强大且灵活的特性。它允许用户对一组相关的行执行计算，这组行被称为“窗口”。与标准的聚合函数（如 SUM(), AVG()）不同，窗口函数不会将多行合并成单个输出行；相反，它们为查询结果集中的每一行返回一个值，同时仍然能够访问与该行相关的其他行的数据。这种能力使得窗口函数非常适合处理复杂的分析任务，如排名、累加、移动平均、以及比较当前行与组内其他行的值。

B.1. 窗口函数核心概念与语法

窗口函数的核心在于 OVER()子句，它定义了计算所针对的窗口。OVER()子句可以包含以下三个可选部分：

1. **PARTITION BY**：用于将结果集划分为不同的分区，窗口函数会独立地应用于每个分区。如果省略 PARTITION BY，则整个结果集被视为一个单一分区。
2. **ORDER BY**：用于指定分区内行的排序方式。这对于计算排名、累积和等操作至

关重要。如果省略 ORDER BY，则分区内的行是无序的，某些窗口函数（如排名函数）的行为可能会不符合预期。

3. **frame_clause**（窗口框架）：用于进一步定义窗口的边界，即在分区内，相对于当前行的哪些行应该被包含在计算中。常见的窗口框架定义包括：

    a. ROWS BETWEEN N PRECEDING AND M FOLLOWING：指定从当前行向前 N 行到向后 M 行的范围。

    b. ROWS UNBOUNDED PRECEDING：从分区的第一行到当前行。

    c. ROWS CURRENT ROW：仅当前行。

    d. ROWS UNBOUNDED FOLLOWING：从当前行到分区的最后一行。

    e. RANGE BETWEEN ...：与 ROWS 类似，但基于行的值而非物理位置。

窗口函数的基本语法如下 ：

```
function_name([arguments]) OVER (
    [PARTITION BY partition_expression, ... ]
    [ORDER BY sort_expression [ASC | DESC], ... ]
    [frame_clause]
)
```

窗口函数在 SELECT 子句中使用，并且在逻辑上是在 WHERE、GROUP BY、HAVING 子句之后，ORDER BY 子句之前执行的 。

B.2. 常用窗口函数分类与示例

窗口函数主要可以分为以下几类：

1. **聚合窗口函数（Aggregate Window Functions）**：

这类函数将标准的聚合函数（如 SUM(), AVG(), COUNT(), MAX(), MIN()）与 OVER()子句结合使用，使其能够在窗口上执行聚合计算，而不是对整个结果集或

GROUP BY 分组进行聚合。

a. **SUM()**：计算窗口内指定列的总和。

```
-- 计算每个员工的销售额，以及截至当前日期（按日期排序）的累计销售额
SELECT emp_name, stat_date, sales,
        SUM(sales) OVER (PARTITION BY emp_name ORDER BY stat_date) AS runnir
FROM q1_sales;
```

---

1. **AVG()**：计算窗口内指定列的平均值。

```
-- 计算每个销售员（dealer_id）的平均销售额，以及整体平均销售额
SELECT emp_name, dealer_id, sales,
        AVG(sales) OVER (PARTITION BY dealer_id) AS avg_sales_per_dealer,
        AVG(sales) OVER () AS overall_avg_sales
FROM q1_sales;
```

---

1. **COUNT()**：计算窗口内的行数。

```
-- 计算每个订单状态的出现次数
SELECT DISTINCT status,
        COUNT(*) OVER (PARTITION BY status) AS status_count
FROM orders;
```

---

1. **MAX() / MIN()**：找出窗口内指定列的最大值/最小值。

```
-- 找出每个用户的最大订单金额
SELECT user_id, amount,
        MAX(amount) OVER (PARTITION BY user_id) AS max_order_amount
FROM orders;
```

---

1. **排名窗口函数（Ranking Window Functions）**：

这类函数用于为分区内的行分配排名或序号。

a. **ROW_NUMBER()**：为分区内的每一行分配一个唯一的连续整数序号，从 1 开始。即使存在相同的排序值，行号也不会重复 。

```
-- 为每个部门的员工按薪水降序排名
SELECT emp_name, department, salary,
        ROW_NUMBER() OVER (PARTITION BY department ORDER BY salary DESC) 
FROM employees;
```

---

1. **RANK()**：为分区内的行分配排名。如果存在相同的排序值，则这些行会获得相同的排名，并且下一个排名会跳过相应的数量（例如，1，2，2，4）。

```
-- 为每个班级的学生按成绩降序排名，允许并列
SELECT student_name, class, score,
        RANK() OVER (PARTITION BY class ORDER BY score DESC) AS class_rank
FROM student_scores;
```

---

1. **DENSE_RANK()**：与 RANK()类似，但排名是连续的，即使存在相同的排序

值，下一个排名也不会跳过（例如，1，2，2，3）。

```
-- 为每个班级的学生按成绩降序排名，允许并列且排名连续
SELECT student_name, class, score,
       DENSE_RANK() OVER (PARTITION BY class ORDER BY score DESC) AS dense
FROM student_scores;
```

---

1. **NTILE(n)**：将分区内的行尽可能均匀地分配到 n 个桶（或组）中，并为每一行分配其所属的桶号（从 1 到 n）。

```
-- 将员工按薪水高低分成 4 个等级
SELECT emp_name, salary,
       NTILE(4) OVER (ORDER BY salary DESC) AS salary_quartile
FROM employees;
```

---

1. **PERCENT_RANK()**：计算每行的百分比排名，公式为 (RANK() - 1) / (分区总行数 - 1)，结果在 0 到 1 之间 。

```
-- 计算每个学生成绩在其班级内的百分比排名
SELECT student_name, class, score,
       PERCENT_RANK() OVER (PARTITION BY class ORDER BY score DESC) AS per
FROM student_scores;
```

---

1. **CUME_DIST()**：计算每行的累积分布，即小于等于当前行值的行数占分区总行数的比例，结果在 0 到 1 之间 。

```
-- 计算每个订单金额的累积分布
SELECT order_id, amount,
        CUME_DIST() OVER (ORDER BY amount) AS cumulative_distribution
FROM orders;
```

---

1. **值窗口函数（Value Window Functions）**：

这类函数允许访问窗口中其他行的值。

a. **LAG(column, offset, default)**：返回窗口中当前行之前 offset 行的 column 值。如果不存在该行，则返回 default（如果指定）或 NULL 。

```
-- 获取每个员工当前销售额以及前一天的销售额
SELECT emp_name, stat_date, sales,
        LAG(sales, 1, 0) OVER (PARTITION BY emp_name ORDER BY stat_date) AS pr
FROM q1_sales;
```

---

1. **LEAD(column, offset, default)**：返回窗口中当前行之后 offset 行的 column 值。如果不存在该行，则返回 default（如果指定）或 NULL 。

```
-- 获取每个员工当前销售额以及后一天的销售额
SELECT emp_name, stat_date, sales,
        LEAD(sales, 1, 0) OVER (PARTITION BY emp_name ORDER BY stat_date) AS r
FROM q1_sales;
```

---

1. **FIRST_VALUE(column)**：返回窗口内第一行的 column 值 。

```
-- 获取每个部门薪水最高的员工及其薪水
SELECT DISTINCT department,
       FIRST_VALUE(emp_name) OVER (PARTITION BY department ORDER BY salar
       FIRST_VALUE(salary) OVER (PARTITION BY department ORDER BY salary DES
FROM employees;
```

---

1. **LAST_VALUE(column)**：返回窗口内最后一行的 column 值。**注意**：默认窗口框架是 RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW，这会导致 LAST_VALUE()返回当前行的值。通常需要指定完整的窗口框架，如 ROWS BETWEEN UNBOUNDED PRECEDING AND UNBOUNDED FOLLOWING，才能得到分区内的最后一个值 。

```
-- 获取每个部门薪水最低的员工及其薪水
SELECT DISTINCT department,
       LAST_VALUE(emp_name) OVER (PARTITION BY department ORDER BY salary
                                  ROWS BETWEEN UNBOUNDED PRECEDING AN
       LAST_VALUE(salary) OVER (PARTITION BY department ORDER BY salary ASC
                                ROWS BETWEEN UNBOUNDED PRECEDING AND
FROM employees;
```

---

1. **NTH_VALUE(column, N)**：返回窗口内第 N 行的 column 值。如果不存在第 N 行，则返回 NULL 。

```
-- 获取每个部门薪水第二高的员工
SELECT DISTINCT department,
       NTH_VALUE(emp_name, 2) OVER (PARTITION BY department ORDER BY sal
                                    ROWS BETWEEN UNBOUNDED PRECEDING A
FROM employees;
```

B.3. 窗口函数应用场景与技巧

窗口函数在解决复杂分析问题时非常强大，以下是一些常见的应用场景和技巧：

1. **排名与分页**：
   a. 使用 ROW_NUMBER(), RANK(), DENSE_RANK()可以轻松实现各类排名需求，如销售排名、成绩排名等。
   b. 结合 ROW_NUMBER()可以实现高效的分页查询，尤其是在需要跨多个列进行复杂排序时。
2. **移动平均与累积计算**：
   a. 通过定义合适的窗口框架（如 ROWS BETWEEN N PRECEDING AND CURRENT ROW），可以计算移动平均、移动总和等，常用于时间序列数据分析，如股票价格、销售额的平滑处理。
   b. 使用 SUM()配合 ORDER BY 和 ROWS UNBOUNDED PRECEDING 可以计算累积和。
3. **同比与环比分析**：
   a. 利用 LAG()和 LEAD()函数，可以方便地获取上一期或下一期的数据，从而计算同比增长率、环比增长率等指标 。

```sql
-- 计算每月销售额的环比增长率
WITH monthly_sales AS (
    SELECT DATE_FORMAT(order_time, '%Y-%m') AS month,
           SUM(amount) AS total_sales
    FROM orders
    GROUP BY DATE_FORMAT(order_time, '%Y-%m')
)
SELECT month, total_sales,
       LAG(total_sales, 1) OVER (ORDER BY month) AS prev_month_sales,
       (total_sales - LAG(total_sales, 1) OVER (ORDER BY month)) / LAG(total_sales,
FROM monthly_sales;
```

---

1. **数据去重与选取特定行**：
   a. 使用 ROW_NUMBER()可以为重复数据分配不同的序号，然后通过筛选序号为 1 的行来实现去重。
   b. 结合 NTILE()可以将数据分成若干组，然后选取特定组的数据进行分析。
2. **计算占比与差异**：
   a. 通过将聚合窗口函数（如 SUM()）与 OVER(PARTITION BY ...)结合，可以计算每个分组占总体的百分比。
   b. 使用 LAG()或 LEAD()可以计算当前行与上一行或下一行之间的差异。
3. **处理复杂业务逻辑**：
   a. 窗口函数可以简化许多原本需要复杂子查询或自连接才能实现的逻辑，使 SQL 代码更简洁、易读。

技巧与注意事项：

- **理解窗口框架的默认行为**：如果指定了 ORDER BY 但未指定窗口框架，默认的框架是 RANGE BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW。这可能导致 LAST_VALUE()等函数的行为与预期不符。
- **性能考虑**：虽然窗口功能强大，但复杂的窗口函数（尤其是涉及大量数据排序或大范围窗口框架）可能会对性能产生影响。需要结合执行计划进行分析和优化。

- **分区键的选择**：合理的 PARTITION BY 子句可以显著提高窗口函数的效率，因为它将计算限制在更小的数据集上。
- **与GROUP BY的结合**：窗口函数可以在GROUP BY之后使用，对分组聚合后的结果进行进一步分析。

通过灵活运用不同类型的窗口函数及其组合，可以高效地解决各种复杂的数据分析需求，提升 SQL 的 expressiveness 和解决问题的能力。