华中科技大学

数据库系统原理实践报告

专　　业：	计算机科学与技术
班　　级：	本硕博 2301
学　　号：	U202315752
姓　　名：	陈宇航
指导教师：	杨茂林

分数	
教师签名	

2025 年 6 月 5 日

教师评分页

子目标	子目标评分
1	
2	
3	
4	
5	
6	

总分	

目　录

1 课程任务概述

“数据库系统原理实践”是配合“数据库系统原理”课程独立开设的实践课，注重理论与实践相结合。本课程以 MySQL 为例，系统性地设计了一系列的实训任务，内容涉及以下几个部分：

1）数据库、表、索引、视图、约束、存储过程、函数、触发器、游标等数据对象的管理与编程；

2）数据查询，数据插入、删除与修改等数据处理相关任务；

3）数据库的安全性控制，完整性控制，恢复机制，并发控制机制等系统内核的实验；

4）数据库的设计与实现；

5）数据库应用系统的开发(JAVA 篇)。

课程依托头歌实践教学平台，实践课程 url 见相关课堂教师发布链接及其邀请码。实验环境为 Linux 操作系统下的 MySQL 8.0.28（主要为 8.028 版本，部分关卡使用 8.022 版本，使用中基本无差别）。在数据库应用开发环节，使用 JAVA 1.8。

2 任务实施过程与分析

本次实践课程在头歌平台进行，实践任务均在平台上提交代码，所有完成的任务、关卡均通过了自动测评。本次实践最终完成了课程平台中的实训 1 到实训 16 的所有任务，下面将重点针对其中的各个实训中重点的任务阐述其完成过程中的具体工作。

2.1 数据库、表与完整性约束的定义(Create)

本章要求了解数据库、表与完整性约束的定义方法，了解数据库语言的基本语法。

本关希望同学们理解和使用 SQL 语句包括：创建唯一性约束(UNIQUE 约束),以保证字段取值的唯一性；创建默认值约束(Default 约束),为字段指定默认值；创建 CHECK 约束，创建外码约束（参照完整性约束）；建一个表，并为表指定主码；创建数据库，具体代码如下

```
CREATE DATABASE ISDB;
CREATE TABLE Crew (CID VARCHAR(5) PRIMARY KEY, Role TEXT, Line TEXT);
CREATE TABLE Evnts (EID INT PRIMARY KEY, EDesc TEXT, Odds DECIMAL(3,2));
CREATE TABLE Logs (LID INT PRIMARY KEY, Msg TEXT, Tone VARCHAR(5));
CREATE TABLE Plnts (PID VARCHAR(5) PRIMARY KEY, TDEfc DECIMAL(4,1), Life BOOLEAN);
CREATE TABLE Qotes (QID INT PRIMARY KEY, Words TEXT, Spkr VARCHAR(5));
ALTER TABLE Evnts ADD CONSTRAINT UQ_ED UNIQUE (EDesc);
ALTER TABLE Logs ADD CONSTRAINT DF_TN DEFAULT 'Hope' FOR Tone;
ALTER TABLE Plnts ADD CONSTRAINT CK_TDE CHECK (TDEfc >= 0);
ALTER TABLE Qotes ADD CONSTRAINT FK_Spk FOREIGN KEY (Spkr) REFERENCES Crew (CID);
```

图 2.1 任务 2.1 代码

2.2 表结构与完整性约束的修改（ALTER）

本章学习 alter table 语句的大部分功能和修改表结构的方法（如添加列、约束，删除列、约束，修改列）等基础知识，了解表的完整性约束。

具体涉及到修改表名，添加与删除字段，修改字段，添加删除与修改约束

```
ALTER TABLE Evnts RENAME TO MOMNT;
ALTER TABLE Crew ADD Stat CHAR(3);
ALTER TABLE Crew DROP COLUMN Line;
ALTER TABLE Logs MODIFY COLUMN Msg VARCHAR(5);
ALTER TABLE Plnts ADD CONSTRAINT CK_LF CHECK (Life IN (0,1));
ALTER TABLE MOMNT DROP CONSTRAINT UQ_ED;
ALTER TABLE Logs DROP CONSTRAINT DF_TN;
ALTER TABLE Logs ADD CONSTRAINT DF_N DEFAULT 'N' FOR Tone;
```

图 2.2 任务 2.2 代码

2.3 数据查询（Select）之一

本章采用某银行的一个金融场景应用的模拟数据库，提供了六个表，分别为：client（客户表）、bank_card（银行卡）、finances_product（理财资产表）、insurance（保险表）、fund（基金表）、property（资产表）。 本实训中大多采用 select 语句来对表进行查询满足要求的各项数据，设计了 子查询、数据库函数等知识点。

此部分实验已完成全部关卡，将针对第 2~3 关，第 6 关以及第 8~19 关详细展开分析。

2.3.2 邮箱为 null 的客户

本关卡已完成，因较为简单故略过分析，需要注意的是判断是否为 null 不能用 = ，要用 is null。

2.3.3 既买了保险又买了基金的客户

本关卡已完成。

查询既买了保险又买了基金的客户名称、邮箱和电话，结果按照 c_id 排序，仅利用 select、from、where exists、order by 即可实现，代码见图 2.1。

```
SELECT c_name, c_mail, c_phone FROM client a
WHERE EXISTS (SELECT * FROM property WHERE pro_c_id = a.c_id AND pro_type = 2)
AND EXISTS (SELECT * FROM property WHERE pro_c_id = a.c_id AND pro_type = 3)
ORDER BY c_id;
```

图 2.3 任务 2.3.3 代码

2.3.6 商品收益的众数

在 SQL 中增加 HAVING 子句原因是，WHERE 关键字无法与聚合函数一

起使用。HAVING 子句可以让我们筛选分组后的各组数据。本关需要用到 count 函数计算出现次数，用 having 语句实现的“众数”的要求，代码见图 2.2。

```
SELECT column1, aggregate_function(column2)
FROM table_name
GROUP BY column1
HAVING condition;
```

图 2.4 任务 2.3.6 代码

2.3.8 持有两张信用卡的用

查询持有两张（含）以上信用卡的用户的相关信息。本关同样使用嵌套查询，先在子查询中筛选出含有两张及其以上信用卡的用户的 id，再进行选取以及排序，代码见图 2.3。

```
SELECT c_name, c_id_card, c_phone FROM client
WHERE c_id IN (SELECT b_c_id FROM bank_card
WHERE b_type = '信用卡' GROUP BY b_c_id HAVING COUNT(*) >= 2)
ORDER BY c_id;
```

图 2.5 任务 2.3.8 代码

2.3.9 购买了货币型基金的客户信息

本关需要使用两次嵌套查询，先 找出 f_type = ‘货币型’的 f_id，再从 property 表中找出 pro_type = 3（意为基金） 且 pro_pif_id 在第一个集合中的 pro_c_id，最后从 client 表中找到 c_id 在第二个 集合中的用户。代码见图 2.4。

```
select c_name,c_phone,c_mail from client
where c_id in ( select pro_c_id from property
    where pro_pif_id in( select f_id from fund where f_type = '货币型' ) and pro_type = 3)
order by c_id;
```

图 2.6 任务 2.3.9 代码

2.3.10 投资总收益前三名的客户

这一关涉及到子查询已经聚类函数 sum 以及用 limit 的输出前三个元组，代码见图 2.5。

```
select c.c_name,c_id_card,ch.sums as total_income
from client c,(select pro_c_id,sum(pro_income) as sums from property where pro_status =
'可用' group by pro_c_id ) as ch
where c_id = ch.pro_c_id
order by ch.sums desc limit 3;
```

图 2.7 任务 2.3.10 代码

2.3.11 黄姓客户持卡数量

这一关要学会用左外连接(left join),以及通配符%的使用，代码见图 2.6。

```
select c_id,c_name, count(b_c_id) number_of_cards
from client as c left join  bank_card as b
on c.c_id = b.b_c_id
where c_name like "黄%" group by c_id order by number_of_cards desc,c_id;
```

图 2.8 任务 2.3.11 代码

2.3.12 客户理财、保险与基金投资总额

本关要求计算用户的投资总金额，由于资产分为三类（理财、保险、 基金），故要用三个 select 语句去分别计算这三类资产的金额。之后，通过 union all 指令将不同的 select 结果连接（注意连接的 select 结果必须拥有相同数量的列，列也必须拥有相似的数据类型，同时，每条 select 语句中的列的顺序必须相同），得到了资产金额表后，将其与 client 表连接，用 sum 函数计算总金额即可， 但是计算时还要注意使用 ifnull 函数将空值转换成 0 值。

注意这里需要用到的是 union all，因为 union 不会去除重复的元组。代码如下所示。

```sql
select c_name,c_id_card,ifnull(sum(profit),0) as total_amount
from client
left join (
    select pro_c_id,pro_quantity*p_amount as profit
    from property,finances_product
    where p_id = pro_pif_id and pro_type = 1  -- 连接条件
    union all -- 组合一下
    select pro_c_id,pro_quantity*i_amount as profit
    from property,insurance
    where i_id = pro_pif_id and pro_type = 2
    union all
    select pro_c_id,pro_quantity*f_amount as profit
    from property,fund
    where f_id = pro_pif_id and pro_type = 3
) f on c_id = pro_c_id    -- 没有这个f不行
group by c_id
order by total_amount desc;
```

图 2.9 任务 2.3.12 代码

2.3.13 客户总资产

使用外连接 left outer join 和 ifnull 函数实现在两表连接时将无记录项置为 0 的操作，并多次使用外连接和子查询统计多表项的数据和，最后在 select 语句中对多项数据进行加减操作运算以计算得出客户总资产。具体的代码不展示。

有一个细节值得注意，在 bank_card 表中信用卡余额即为透支金额，我这里使用了一个 if 的判别语句，见图 2.7，如果是储蓄卡就统计 b_balance,否则统计其相反数，其实也可以是用两次查询实现，分别判断是储蓄卡还是信用卡。

```sql
if(
    b_type = "储蓄卡",
    b_balance,
    - b_balance
)
```

图 2.10 任务 2.3.13 中 if 用法

2.3.14 第 N 高问题

本关只需使用一次嵌 套循环即可完成，使用 limit N, M 指令，可以从第 N 条记录开始，返回 M 条记 录，故在将 insurance 表中数据排序后很容易就能找到第 4 高的产品。代码见图 2.8。

```sql
SELECT DISTINCT i_id, i_amount
FROM insurance
WHERE i_amount = (
    SELECT DISTINCT i_amount FROM insurance ORDER BY i_amount DESC LIMIT 3, 1
);
```

图 2.11 任务 2.3.14 代码

2.3.15 基金收益两种方式排名

本关要求分别实现全局名次不连续的排名和连续的排名，即考察 rank() over 和 dense_rank() over 的区别，前者为名次不连续的排名，后者为连续的排名，代码见图 2.9。

```sql
-- (1) 基金总收益排名(名次不连续)
select pro_c_id, sum(pro_income) as total_revenue,
rank() over(order by sum(pro_income) desc) as `rank`
from property where pro_type = 3 group by pro_c_id
order by sum(pro_income) desc, pro_c_id;
-- (2) 基金总收益排名(名次连续)
select pro_c_id, sum(pro_income) as total_revenue,
dense_rank() over(order by sum(pro_income) desc) as `rank`
from property where pro_type = 3 group by pro_c_id
order by sum(pro_income) desc, pro_c_id;
```

图 2.12 任务 2.3.15 代码

2.3.16 持有完全相同基金组合的用户

查询所有持有相同基金组合的用户对。本关的思路为：先将两个 property 表连接（命名为 first 和 second），连接条件为两表中的 pro_type 均等于 3 且两个 表的 pro_pif_id 相等，这样可以将有相同基金的两个人连到一行里面，然后使用 select 语句查询 A、B 用户，按照 first 中的 c_id 和 second 中的 c_id 分组，使用 having 语句保证选出的行中，基金数与 A 和 B 持有的基金数均相等，即可保证 A 和 B 持有的基金完全相同。

```sql
select first.pro_c_id as c_id1, second.pro_c_id as c_id2
from property first inner join property second on(
first.pro_c_id<second.pro_c_id
and first.pro_type='3' and second.pro_type='3' and
first.pro_pif_id=second.pro_pif_id
) # 将有相同基金的两个人连到一个元组中
group by c_id1,c_id2
# having 语句保证了选出的两个人持有的基金数相同
having count(*)=(
select count(*) from property
where pro_c_id=first.pro_c_id and pro_type='3' # 保证选出的元组，持有
基金数与第一个人持有的基金数相同)
and count(*)=(
select count(*) from property
where pro_c_id=second.pro_c_id and pro_type='3 '# 保证选出的元组，持
有基金数与第二个人持有的基金数相同);
```

图 2.13 任务 2.3.16 代码

2.3.17 购买基金的高峰期

首先进行 select 生成派生表，对每天的交易量进行计算(pro_quantity * f_amount)，并使用 sum 函数求和，标注为 total_amount，同时可根据 datediff 函数及相关计算求出该天是 2021 年第几个工作日（命名为 workday）。 之后外部的一层查询将交易量未达到一百万的记录排除，使用 row_number() 函数对按照 workday 排序后的数据生成从 1 开始的行号，命名为 rownum。我们 注意到，在连续的若干交易日期中，如果交易量均超过一百万，那么 workday 和 rownum 均是连续的。因此通过计算 workday 和 rownum 的差值并按此进行分组，使用 count 函数计算每组的行数，结果大于等于 3 的那些组，就是购买基金 的高峰期。

```
select
    daycnt.pro_purchase_time as pro_purchase_time,
    daycnt.amount as total_amount
from (
        select
            *,
            count(*) over(
                partition by orderday.workday - orderday.rownum
            ) cnt
        from (
                select
                    *,
                    row_number() over(
                        order by workday
                    ) rownum
                from (
                        select
                            pro_purchase_time,
                            sum(pro_quantity * f_amount) as amount,
                            datediff(
                                pro_purchase_time,
                                "2021-12-31"
                            ) - 2 * week(pro_purchase_time) as workday
                        from
                            property,
                            fund
                        where
                            pro_pif_id = f_id
                            and pro_type = 3
                            and pro_purchase_time like "2022-02-%"
                        group by
                            pro_purchase_time
                    ) amount_by_day
                where
                    amount_by_day.amount > 1000000
            ) orderday
    ) daycnt
where daycnt.cnt >= 3;
```

图 2.14 任务 2.3.17 代码

2.3.18 至少有一张信用卡余额超过 5000 元的客户信用卡总余额

通过嵌套查询找出有一张信用卡余额超过 5000 的客户编号，再进一步筛选，然后用 sum 函数求出符合的客户的总额度，代码见图 2.10。图 2.10 任务 2.3.18 代码

```
select b_c_id, sum(b_balance) as credit_card_amount from bank_card
where b_type = "信用卡" and b_c_id in
( select b_c_id from bank_card where b_type = "信用卡" and b_balance > 5000)
group by b_c_id order by b_c_id;
```

图 2.15 任务 2.3.18 代码

2.3.19 以日历表格式显示每日基金购买总金额

以日历表格式列出 2022 年 2 余额每周每日基金购买总金额。本关需要用一 次嵌套循环，先将 property 表与 fund 表连接，连接条件为 pro_pif_id = f_id，从中挑出 2022 年 2 月的数据，按照购买时间 pro_purchase_time 分组，使用 sum 计算每一交易日期的总交易金额（定义为 amount），并用 week 函数计算出该日期是 2 月的第几周，用 weekday 函数算出是星期几(定义为‘id’)。之后用外层的 select 算出所有周一至周五 9 的基金购买总金额，外层这里计算的关键代码见图 2.11。

```
sum(if(`id` = 0, amount, null)) Monday,
sum(if(`id` = 1, amount, null)) Tuesday,
sum(if(`id` = 2, amount, null)) Wednesday,
sum(if(`id` = 3, amount, null)) Thursday,
sum(if(`id` = 4, amount, null)) Friday
```

图 2.16 任务 2.3.19 关键代码

2.4 数据查询(Select)-新增

使用 MySQL 语言实现数据查询。掌握 select 及其相关语句的使用。相较于 2.3 难度提升。

此部分实验已完成全部关卡，由于较为复杂，逐一进行分析。

2.4.1 查询销售总额前三的理财产品

查询 2010 年和 2011 年这两年每年销售总额前 3 名（如果有并列排名，则后 续排名号跳过之前的并列排名个数，如 1、1、3）的相关信息。本关卡需要使用 一次嵌套循环，内部的 select 查询出年份在 2010 和 2011 年中的数据（具体是哪 一年定义为 pyear），按 p_id 和 pyear 分组，用 sum 函数计算出销售总额（p_amount * pro_quantity），定义为 sumamount。外层的 select 使用 rank() over 函数对选出 的记录进行排名（以 pyear 分组，sumamount 为排序依据）。如果不分组，那么结果的排序按照顺序来，关键代码见图 2.12。

```
select pyear,
rank() over(partition by pyear order by sumamount desc) as rk,
p_id, sumamount
```

图 2.17 任务 2.4.1 关键代码

2.4.2 投资积极且偏好理财类产品的客户

查询所有此类客户的编号。本关 使用两个子查询先分别计算用户购买理财产品的数量（cnt1）和购买基金的数量 （cnt2），定义为两个表 t1 和 t2，将两个表按照 pro_c_id 连接，使用外层的 select 筛选出。

```sql
with table_cnt_1 as
(
    select pro_c_id, count(distinct(pro_pif_id)) as cnt_1
    from property
    where pro_type = 1
    group by pro_c_id
),#理财产品
table_cnt_3 as
(
    select pro_c_id, count(distinct(pro_pif_id)) as cnt_3
    from property
    where pro_type = 3
    group by pro_c_id
)#基金
select distinct property.pro_c_id
from property
inner join table_cnt_1 on property.pro_c_id = table_cnt_1.pro_c_id
inner join table_cnt_3 on property.pro_c_id = table_cnt_3.pro_c_id
where pro_type = 1 and cnt_1 > cnt_3
order by pro_c_id;
```

图 2.18 任务 2.4.2 关键代码

2.4.3 查询购买了所有畅销理财产品的客户

本关使用了嵌套了两层 where not exists 的查询，首先生成畅销理财产品的表与被用户购买了的理财产品的表。内层的 where not exists 找出那些没有被购买的畅销理财产品，（如果有）再利用外层的 where not exists 查找没有漏掉购买任何一个畅销理财产品的人（即购买了所有畅销理财 产品的人）。

```sql
select pro_c_id
from
(
   select pro_c_id, count(pro_c_id) as cnt1, cnt2
   from
   (
      select
         pro_c_id,
         pro_pif_id,
         (
            select count(*)
            from
            (
               select pro_pif_id
               from property
               where pro_type = 1
               group by pro_pif_id having count(distinct(pro_c_id)) > 2
            ) as t1
         ) as cnt2
      from property as p
      natural join
      (
         select pro_pif_id
         from property
         where pro_type = 1
         group by pro_pif_id having count(distinct(pro_c_id)) > 2
      ) as t2
      where pro_type = 1
      order by pro_c_id
   ) as t3
   group by pro_c_id
) as t4
where cnt1 = cnt2;
```

图 2.19 任务 2.4.3 关键代码

2.4.4 查找相似的理财产品

首先从 property 表中找到全体持有 14 号理财产品的客户 以及每个客户的持有数量，定义为 t1 表；使用 dense_rank() over 函数将其中的数 据按持有

数量排序，定义为 t2 表；从 t2 表中选出持有 14 号理财产品数量最多的 前三名（但可能不止三个人），定义为 t3 表；t3 表与 property 进行自然连接后， 从中选出前三名持有的理财产品（除 14 号以外），定义为 t4 表；t4 表与 property 再进行自然连接，数据按 pro_pif_id 分组，按 pro_pif_id 排序，选出 pro_pif_id 及该编号的理财产品的购买客户总人数 cc，定义为 t5 表；从 t5 表中查找 pro_pif_id，cc，以及用 dense_rank() over 函数生成的排名。

```
SELECT pro_pif_id, COUNT(*) as cc, dense_rank() over(order by COUNT(*) DESC) AS prank
FROM property
WHERE
    pro_type = 1 AND
    pro_pif_id in
    (SELECT DISTINCT pro_pif_id
    FROM property
    WHERE
        pro_type = 1 AND
        pro_pif_id <> 14 AND
        pro_c_id in
        (SELECT pro_c_id
        FROM
            (SELECT pro_c_id, dense_rank() over(order by pro_quantity) AS rk
            FROM property
            WHERE pro_type = 1 AND
                pro_pif_id = 14) fin_rk
        WHERE fin_rk.rk <= 3))
GROUP BY pro_pif_id
ORDER BY cc DESC, pro_pif_id ASC;
```

图 2.20 任务 2.4.4 关键代码

2.4.5 查询任意两个客户的相同理财产品数

本关使用两个 property（命名为 p1 和 p2）进行自然连 接，按 p1.pro_c_id，p2.pro_c_id 进行分组，使用 count 函数计算两人持有的相同 理财产品的数量，使用 having count 满足“至少 2 种”的要求。

```
select p1.pro_c_id, p2.pro_c_id, count(*) as total_count
from property as p1 inner join property as p2
on p1.pro_pif_id = p2.pro_pif_id
where p1.pro_type = 1 and p2.pro_type = 1
    and p1.pro_c_id != p2.pro_c_id#去除本身
group by p1.pro_c_id, p2.pro_c_id
having count(*) >= 2
order by p1.pro_c_id;
```

图 2.21 任务 2.4.5 关键代码

2.4.6 查找相似的理财客户

本关卡已完成。

本关卡需要明确“相似的理财客户”的定义（比较复杂故不再赘述），先将 property 表定义为 t1，将从 property 中 select 出的 pro_type = 1 的用户 id 和业务 id 定义为 t2 表，t1 和 t2 表进行连接后，表的每一行就有了（A 客户，B 客户， 他们共同拥有的产品 id）这样的数据（但注意此时 A 客户和 B 客

户可能是同一 个人）之后从连接的表中 select 出不同的 A 客户作为 pac，不同的 B 客户作为 pbc， 用 count 计算他们共同持有的理财产品数作为 common，将这些内容定义为 t3 表； 使用 rank() over 函数按照 pac 分组，按 common 降序，pbc 升序的方式进行排名， 作为 t4 表；从 t4 表中选出排名值小于 3 的相似客户即可。

```
SELECT pac, pbc, common, crank
FROM
    (SELECT pac, pbc, common, rank() over(partition by pac order by common,pbc) AS crank
    FROM
        (SELECT pac, pbc, COUNT(*) AS common
        FROM
            (SELECT DISTINCT p1.pro_c_id AS pac, p2.pro_c_id AS pbc, p2.pro_pif_id
            FROM property p1, property p2
            WHERE p1.pro_c_id <> p2.pro_c_id AND
                p2.pro_type = 1 AND
                p2.pro_pif_id IN
                (SELECT p3.pro_pif_id
                FROM property p3
                WHERE p3.pro_c_id = p1.pro_c_id AND
                    p3.pro_type = 1)) common_c
        GROUP BY pac,pbc) common_b) common_a
WHERE crank <= 2;
```

图 2.22 任务 2.4.6 关键代码

2.5 数据查询(Select)-新增 2

使用 MySQL 语言实现数据查询。掌握 select 及其相关语句的使用。相较于 2.4 难度提升。

此部分实验已完成全部关卡，由于较为复杂，逐一进行分析。

2.5.1 将客户年度从各单位获得的酬劳进行排序

首先，将 wage 薪酬表与 client 客户表通过客户 ID 进行内连接，以获取客户详细信息并确保数据有效性。接着，按客户（包括姓名、身份证号）以及从酬劳时间 w_time 中提取的年份进行分组。在每个分组内，运用条件聚合函数（例如 SUM(CASE WHEN w_type = 1 THEN w_amount ELSE 0 END)）来分别计算该客户在该年度的全职酬劳总额和兼职酬劳总额，同时使用 COALESCE 函数确保若无某种类型收入时其金额显示为 0。最后，将包含客户名称、年份、身份证号、全职及兼职酬劳总额的结果集，按照全职酬劳与兼职酬劳两者之和进行降序排列。

```sql
WITH yearly_sum AS (
    SELECT
        w.w_c_id,
        EXTRACT(YEAR FROM w.w_time)                 AS year,
        SUM(CASE WHEN w.w_type = 1                  -- 1 代表全职
                 THEN w.w_amount ELSE 0 END)        AS full_t_amount,
        SUM(CASE WHEN w.w_type = 2                  -- 2 代表兼职
                 THEN w.w_amount ELSE 0 END)        AS part_t_amount
    FROM wage w
    GROUP BY w.w_c_id, EXTRACT(YEAR FROM w.w_time)
)

/* 2. 只保留在 client 表中确实存在的有效客户，并按照规则排序输出 */
SELECT
    c.c_name,
    y.year,
    c.c_id_card,
    y.full_t_amount,
    y.part_t_amount
FROM       yearly_sum y
JOIN       client     c  ON c.c_id = y.w_c_id        -- 排除“悬浮”客户行
ORDER BY   (y.full_t_amount + y.part_t_amount) DESC ,c.c_id ASC; -- 合计酬劳降序
```

图 2.23 任务 2.5.1 关键代码

2.5.2 在均不计兼职的情况下，统计各单位的薪资总额、月平均薪资、最高薪资、最低薪资、中位薪资

思路是，首先，筛选全职员工薪资记录，关联 client 表确保客户有效性，并计算出每位员工在每个单位每月的实际总薪酬。基于此月度薪酬数据，按单位（w_org）分组，直接计算得出薪资总额（total_amount）、实际月薪中的最高薪资（max_wage）和最低薪资（min_wage），同时统计独立员工数和独立月份数。月平均薪资（average_wage）则由该单位的薪资总额除以其独立员工数与独立月份数的乘积得到，并保留两位小数。中位薪资（mid_wage）的计算相对复杂：先计算出每个员工在该单位内的平均月薪，然后对这些个人平均月薪进行排序，取排序后处于中间位置的一个或两个（若员工数为偶数）数值的平均值，同样保留两位小数。最后，将所有计算得到的统计指标按单位汇总，并按照薪资总额降序排列输出。

```sql
with table_cnt_1 as
(
    select pro_c_id, count(distinct(pro_pif_id)) as cnt_1
    from property
    where pro_type = 1
    group by pro_c_id
),#理财产品
table_cnt_3 as
(
    select pro_c_id, count(distinct(pro_pif_id)) as cnt_3
    from property
    where pro_type = 3
    group by pro_c_id
)#基金
select distinct property.pro_c_id
from property
inner join table_cnt_1 on property.pro_c_id = table_cnt_1.pro_c_id
inner join table_cnt_3 on property.pro_c_id = table_cnt_3.pro_c_id
where pro_type = 1 and cnt_1 > cnt_3
order by pro_c_id;
```

图 2.24 任务 2.5.2 关键代码

2.4.3 获得兼职总酬劳前三名的客户

完成此任务的核心思路与查找前三酬劳单位类似，但目标是客户。首先，从 wage 表中筛选出所有类型为兼职（w_type = 2）的酬劳记录。随后，这些记录必须与 client 客户表通过客户编号（wage.w_c_id = client.c_id）进行 INNER JOIN 内连接，以确保只统计有效客户的酬劳，并能获取客户的姓名和身份证号。接着，按客户进行分组，具体是按 client.c_id（以及连带的 c_name, c_id_card，因为它们对于一个客户是固定的）进行 GROUP BY。在每个客户的分组内，使用 SUM(wage.w_amount) 计算该客户获得的兼职酬劳总额，并将其命名为 total_salary。为了找出酬劳最高的客户，需要将聚合后的结果按 total_salary 进行降序排序（ORDER BY total_salary DESC）。最后，根据题目要求（如示例所示，"前 5 名为 16000，12900，12900，8000，8000 时，也只需要给出 16000，12900，12900 即可"），使用 LIMIT 3 子句严格限制输出结果为排序后的前三条记录，即使存在薪酬并列的情况导致第三个薪酬值有多人获得，也只取到第三行数据为止。输出列为客户姓名（c_name）、身份证号（c_id_card）和计算得到的总酬劳（total_salary）。

```sql
WITH part_wage AS (                    -- ① 只保留兼职记录且排除悬浮元组
    SELECT  c.c_id,
            c.c_name,
            c.c_id_card,
            w.w_amount
    FROM    wage   w
    JOIN    client c ON c.c_id = w.w_c_id
    WHERE   w.w_type = 2               -- 兼职
)

SELECT  c_name,
        c_id_card,
        SUM(w_amount) AS total_salary
FROM    part_wage
GROUP BY c_id, c_name, c_id_card       -- ② 按客户汇总
ORDER BY total_salary DESC             -- ③ 按总酬劳降序
LIMIT 3;                               -- ④ 只要前三
```

图 2.25 任务 2.5.3 关键代码

2.4.4 查找兼职酬劳的前三单位

完成此任务的思路首先是数据筛选与关联：从 wage 表中精确筛选出所有 w_type = 2 的兼职酬劳记录，这是定位目标数据的第一步。紧接着，为了确保数据有效性并排除悬浮元组，必须将这些兼职记录与 client 客户表通过 w_c_id（wage 表）和 c_id（client 表）进行 INNER JOIN 内连接操作，这样可以保证每一条参与计算的酬劳都对应一个真实存在的客户。数据聚合与计算阶段：在获得了有效的兼职酬劳数据后，需要按照酬劳发放单位 w_org 进行分组 (GROUP BY w_org)，以便分别统计各单位的酬劳情况。对于每个单位分组，使用聚合函数 SUM(w_amount)计算其发放的兼职酬劳总额，并将此结果列命名为 total_salary，这是核心的统计步骤。结果排序与限制：计算出各单位的总酬劳后，为了找出前三名，需要对结果集按照 total_salary 进行降序排序 (ORDER BY total_salary DESC)，使得酬劳最高的单位排列在前。最后，鉴于题目明确要求仅输出前三个单位，应用 LIMIT 3 子句来截取排序后的前三条记录。输出结果仅包含 w_org 和计算得到的 total_salary。

```sql
with part_job as(
    select w_org,
    w_amount
    from wage w
    join client c on w.w_c_id = c.c_id
    where w.w_type = 2
)

select w_org,sum(w_amount) as total_salary from part_job
group by w_org
order by total_salary desc
limit 3
```

图 2.26 任务 2.4.4 关键代码

2.5.5 用一条 SQL 语句将 new_wage 表的全部酬劳信息插入到薪资表(wage)。

首先使用窗口函数对 new_wage 表按 c_id_card、w_amount、w_org 和 w_time 字段去重，只保留一条。接着，将去重后的数据与 client 表连接以获取客户 ID (w_c_id)。然后，分别处理全职（w_type=1）和兼职（w_type=2）记录：全职记录直接选取所需字段；兼职记录则需按客户 ID、发放单位（w_org）及月份（DATE_FORMAT）分组，对酬劳（w_amount）求和，酬劳时间（w_time）取当月最早，酬劳说明（w_memo）用 GROUP_CONCAT 按时间顺序合并。最后，将处理后的全职与兼职结果用 UNION ALL 合并，统一设置默认的缴纳税标志为'N'，然后将整合后的数据插入到 wage 表中。

由于代码较多，不再展示。

2.5.6 对身份证号为 420108199702144323 的客户 2023 年的酬劳代扣税

首先据身份证号从 client 表查得客户 ID；接着，计算该客户 2023 年总收入及总应扣税额；最后，更新 wage 表中该客户 2023 年的每条薪资记录，按月薪占年收入比例分摊总税额，并设置相应扣税标志。

```sql
JPDATE wage AS w
JOIN client AS c                    ON c.c_id = w.w_c_id
JOIN (  /*— 先算出该客户 2023 年总收入及应扣税 —*/
      SELECT  c.c_id                                          AS cid,
              SUM(w2.w_amount)                                AS total_salary,
              GREATEST(SUM(w2.w_amount) - 60000, 0) * 0.2     AS total_tax
      FROM    wage   w2
      JOIN    client c ON c.c_id = w2.w_c_id
      WHERE   c.c_id_card = '420108199702144323'
        AND   YEAR(w2.w_time) = 2023
      GROUP BY c.c_id
    ) AS t  ON t.cid = w.w_c_id
/*— 更新 2023 全部记录(包含兼职) —*/
SET w.w_amount = CASE
                   WHEN t.total_tax > 0
                   THEN w.w_amount
                        - t.total_tax * (w.w_amount / t.total_salary)
                   ELSE w.w_amount
                 END,
    w.w_tax    = CASE
                   WHEN t.total_tax > 0 THEN 'Y'
                   ELSE 'N'
                 END
WHERE c.c_id_card = '420108199702144323'
  AND YEAR(w.w_time) = 2023;      /* 仅动 2023 年记录 */
```

图 2.27 任务 2.5.6 关键代码

2.6 数据的插入、修改与删除

使用 MySQL 语言实现表数据的插入、修改与删除。需要掌握 Insert,Update,Delete 及其相关语句的使用。

此部分实验已完成全部关卡，着重分析第 6 关卡。

2.5.6 连接更新

这一关涉及到两个表，我直接 update 后跟两个表，通过 where 条件进行筛选后赋值，代码见

```
UPDATE property,client
SET pro_id_card=c_id_card
WHERE pro_c_id = c_id;
```

图 2.13 任务 2.6.6 代码

2.7 视图

视图（view）是一个虚拟表，其内容由查询定义。同真实的表一样，视图包含一系列带有名称的列和行数据。但是，数据库中只存放了视图的定义，而并没有存放视图中的数据，这些数据存放在原来的表中。使用视图查询时，数据库会从原来的表中取出对应的数据。本章要求学会用 create 创建视图，以及利用视图 查找表中数据。

此部分实验已完成全部关卡，由于较为简单，不展开分析。

2.8 存储过程与事务

存储过程是在大型数据库系统中，一组为了完成特定功能的 SQL 语句集，存储在数据库中，其经过第一次编译后再次调用不需要再次编译，用户通过指定存储过程的名字并给出参数（如果有）来调用存储过程。本章要求学会使用流程控制语句/游标/事务的存储过程。

2.8.1 使用流程控制语句的存储过程

首先使用 delimiter 重定义 mysql 分隔符为$$，使用 create procedure 创建存储过程，在函数体中，先用 declare 定义相关参数，使用流程控制语句以及迭

代的思想设计插入斐波那契数列的功能，对前两项单独判断，第三项及以后的值为其前面两项之和。

由于篇幅限制，本关代码不再给出。

2.8.2 使用游标的存储过程

本关要先设置两个游标，分别用于指向护士和医生，且还要设置一个 NOT FOUND 的 HANDLER(处理例程)用来判断是否到达末尾，在我们的循环中进行排班工作，我们每次先从游标 fetch 一下，接着判断是否达到末尾，如果到达末尾的我们需要把游标关闭又开启，这样我们又能重新开始读取数据，安排完护士以后我们就可以进行医生的排班，我们要注意的是主任不安排在周末，我们要先判断当前是星期几，如果是星期一我们需要判断我们设置的存储变量是否为空，不为空就将存储变量中的主任放入排班，如果存储变量为空，则我们需要进行正常的排班，且如果当前是周末的话，我们就要进行判断是否是主任，如果是主任我们就要把主任放在存储变量里面，然后取出下一个医生进行排班，依次往复操作。

由于篇幅限制，代码不给出。

2.8.3 使用事务的存储过程

这一关要读懂题目的意思，区分储蓄卡和信用卡的状态，理清楚，用三个 update 负责选择更新转账的需求，接着需要用 if 语句首先判断转出卡的余额是否剩余，若没有剩余，返回 0，接着检查接受者以及其卡号是否存在，若不存在，返回 0，剩下的情况就是符合的情况，进行提交。

```
create procedure sp_transfer(
                    IN applicant_id int,
                    IN source_card_id char(30),
                    IN receiver_id int,
                    IN dest_card_id char(30),
                    IN amount numeric(10,2),
                    OUT return_code int)
BEGIN
SET autocommit = OFF;
START TRANSACTION; -- 开启事务
    UPDATE bank_card SET b_balance = b_balance-amount WHERE b_number = source_card_id AND b_c_id =
applicant_id AND b_type = "储蓄卡";
    UPDATE bank_card SET b_balance = b_balance+amount WHERE b_number = dest_card_id AND b_c_id = receiver_id
AND b_type = "储蓄卡";
    UPDATE bank_card SET b_balance = b_balance-amount WHERE b_number = dest_card_id AND b_c_id = receiver_id
AND b_type = "信用卡";
    IF NOT EXISTS(SELECT * FROM bank_card WHERE b_number = source_card_id AND b_c_id = applicant_id AND
b_type = "储蓄卡" AND b_balance > 0) THEN    -- 检查申请的
        SET return_code = 0;
        ROLLBACK;
    ELSEIF NOT EXISTS(SELECT * FROM bank_card WHERE b_number = dest_card_id AND b_c_id = receiver_id) THEN -
- 检查接收到的
        SET return_code = 0;
        ROLLBACK;
    ELSE
        SET return_code = 1;
        COMMIT;
    END IF;
SET autocommit = TRUE;
END$$
```

2.9 触发器

触发器（trigger）是由事件来触发某个操作。这些事件包括 insert 语句、update 语句和 delete 语句。当数据库系统执行这些事件时，就会激活触发器执行相应的操作。本章要求为指定的表编写触发器以实现特定的业务规则。

2.8.1 为投资表 property 实现业务约束规则-根据投资类别分别引用不同表的主码

理解题意后这题变得简单，我们首先需要注意我们的触发器类型是行级触发器，每插入一行就触发一次，接着我们需要对插入的数据的 type 进行 1,2,3 类型分别处理，且我们插入数据的 pro_pif_id 必须是对应表的 id,然后如果不符合条件，就进行异常处理，最后如果我们插入的数据 type 不在 1,2,3 的范围内，我们也要发出 illegal 的异常处理。代码见图 2. 12。

```
delimiter $$
CREATE TRIGGER before_property_inserted BEFORE INSERT ON property
FOR EACH ROW  -- 行级触发器
BEGIN
DECLARE msg VARCHAR(50);
IF new.pro_type = 1 AND NOT EXISTS (SELECT * FROM finances_product WHERE p_id = new.pro_pif_id) THEN
    SET msg = CONCAT("finances product #",new.pro_pif_id," not found!");
    SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = msg;
ELSEIF new.pro_type = 2 AND NOT EXISTS (SELECT * FROM insurance WHERE i_id = new.pro_pif_id) THEN
    SET msg = CONCAT("insurance #",new.pro_pif_id," not found!");
    SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = msg;
ELSEIF new.pro_type = 3 AND NOT EXISTS (SELECT * FROM fund WHERE f_id = new.pro_pif_id) THEN
    SET msg = CONCAT("fund #",new.pro_pif_id," not found!");
    SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = msg;
ELSEIF new.pro_type NOT IN (1,2,3) THEN
    SET msg = CONCAT("type ",new.pro_type," is illegal!");
    SIGNAL SQLSTATE '45000' SET MESSAGE_TEXT = msg;
END IF;
END$$
delimiter ;
```

图 2.14 任务 2.8.1 代码

2.10 用户自定义函数

本章要求掌握在 SELECT 语句中应用自定义函数。。具体来说就是通过定义特定的模块化函数，满足用户特定的需求，在之后的多次调用中可以发挥很大的作用。此部分实验通过了所有关卡。

2.10.1 创建函数并在语句中使用它

本关卡已完成。

使用 CREATE FUNCTION function_name([para data_type[,...]]) returns data_type begin function_body; return expression; end 语句进行用户函数的自定义。在此部分函数定义的具体代码见图 2.15。

```
create function get_deposit(client_id int)
returns numeric(10,2)
begin
    return ( SELECT SUM(b_balance) FROM bank_card WHERE b_type = '储蓄卡'
        GROUP BY b_c_id HAVING b_c_id = client_id);
end$$
```

图 2.15 任务 2.9.1 的定义函数代码

进一步调用自定义函数的方法，与其他函数的调用方法基本相同，具体代码见图 2.16。

```
select *
from (select c_id_card, c_name, get_deposit(c_id) total_deposit
    from client) tmp
where total_deposit >= 1000000
order by total_deposit desc;
```

图 2.16 任务 2.9.1 的调用函数代码

2.11 安全性控制

考察 MySQL 的安全控制机制、create user 语句的使用和 rant 和 revoke 语句的使用等。具体来说就是通过自主存取控制方法，通过授予创建用户、 特定用户特定的权限和收回特定的权限等，保证数据库安全，防止数据泄露。此 部分实验中通过了所有的关卡。

2.11.1 用户和权限

这一关我们要掌握 create user 语句和 grant 和 revoke 语句，这些语句都有固定的语法，具体的实现代码见图 2.17。

```
#(1) 创建用户tom和jerry，初始密码均为'123456';
create user tom identified by '123456';
create user jerry identified by '123456';
#(2) 授予用户tom查询客户的姓名，邮箱和电话的权限,且tom可转授权限;
grant select (c_mail, c_name, c_phone)
on table client
to tom
with grant option;
#(3) 授予用户jerry修改银行卡余额的权限;
grant update (b_balance)
on table bank_card
to jerry;
#(4) 收回用户Cindy查询银行卡信息的权限。
revoke select on table bank_card from Cindy;
```

图 2.17 任务 2.10.1 代码

2.11.2 用户、角色与权限

这一关创建角色，授予角色一组权限，并将角色代表的权限授予指定的一组用户。这一关目的在于让我们熟悉角色的用法，体会其便利之处，实现代码与上一关有异曲同工之处，由于篇幅限制，代码不给出。

2.12 并发控制与事务的隔离级别

对于并发操作，可能会产生数据不一致性，如丢失修改(lost update)，读脏数据(dirty read)，不可重复读(non-repeatable read)，幻读(phantom read)，选择合适

的事务隔离级别，使得两个事务并发执行时不发生数据不一致的问题。

2.12.1 并发控制与事务的隔离级别

使用 set session transaction isolation level 语句设置事务的隔离等 级。事务隔离级别从低到高分以下四级，读未提交（READ UNCOMITTED）、读 已提交（READ COMMITTED）、可重复读（REPEATABLE READ）、可串行化（SERIALIZABLE）。然后通过 start transaction;开启事务。具体代码见图 2. 18。

```
set session transaction isolation level read uncommitted;
start transaction;
insert into dept(name) values('运维部');
rollback;
```

图 2. 18 任务 2.11.1 代码

2.12.2 读脏

读脏(dirty read)，或者又叫脏读，是指一个事务(t1)读取到另一个事务(t2)修改后的数据，后来事务 t2 又撤销了本次修改(即事务 t2 以 roll back 结束)，数据恢复原值。这样，事务 t1 读到的数据就与数据库里的实际数据不一致，这样的数据被称为“脏”数据，意即不正确的数据。只有一种隔离级别可能会产生读脏，即为读不提交。

其中主要通过 set @n = sleep(时间)进行时间上的控制，以事务 1 为例，其事务的具体代码见图 2. 19。

```
set @n = sleep(1);
select tickets from ticket where flight_no = 'CA8213';
commit;
```

图 2. 19 任务 2.11.2 事务的实现代码

2.12.3 不可重复读

不可重复读(unrepeatable read)，是指一个事务(t1)读取到某数据后，另一个事务(t2)修改了该，事务 t1 并未修改该数据，但当 t1 再次读取该数据时，发现两次读取的结果不一样。不可重复读产生的原因，是事务 t1 的两次读取之间，有另一个事务修改了 t1 读取的数据。有两种级别可以发生不可重复读的现象。具体来说，本实验需要通过更加精细化的时间控制，实现对于两个事务不可重复读现象的重现。具体代码省略。

2.12.4 幻读

幻读是指一个事务(t1)读取到某数据后，另一个事务(t2)作了 insert 或 delete 操作，事务 t1 再次读取该数据时，魔幻般地发现数据变多了或者变少了(记录数量不一致)；而不可重复读限指事务 t2 作了 update 操作，致使 t1 的两次读操作读到的结果(数据的值)不一致。幻读与不可重复读之间的区别就是幻读是发现数据莫名其妙地增加或减少，而不可重复读现象是数据在两次读取时，发现读取的内容不一致的现象。幻读产生的原因，是事务 t1 的两次读取之间，有另一个事务 insert 或 delete 了 t1 读取的数据集。具体代码省略。

2.12.5 主动加锁保证可重复读

MySQL 提供了主动加锁的机制，使得在较低的隔离级别下，通过加锁，以实现更高级别的一致性。SELECT 语句支持 for share 和 for update 短语，分别表示对表加共享(Share)锁和写(write)锁，共享锁也叫读锁，写锁又叫排它锁。具体添加共享锁的代码见图 2. 20。

```
select tickets from ticket where flight_no = 'MU2455' for share;
```

图 2. 20 任务 2.11.5 添加共享锁代码

2.12.6 可串行化

多个事务并发执行是正确的，当且仅当其结果与按某一次序串行地执行这些事务时的结果相同。两个事务 t1,t2 并发执行，如果结果与 t1→t2 串行执行的结果相同，或者与 t2→t1 串行执行的结果相同，都是正确的(可串行化的)。

选择除 serializable(可串行化)以外的任何隔离级别，保证两个事务并发执行的结果是可串行化的。需要通过时间控制，交错执行程序，实现可串行化调度。具体代码省略。

2.13 备份+日志：介质故障与数据库恢复

MySQL 利用备份、日志文件实现恢复，在这一关中我们需要了解并掌握之本的恢复机制和备份与恢复工具。

此部分实验已完成全部关卡。由于篇幅限制，此部分实验不展开分析。

2.14 数据库设计与实现

本章将从实际出发，从按需求建表到设计 E-R 图并进一步转换为关系模型，再到使用建模工具对数据库进行建模，了解一个数据库从概念模型到具体实现的步骤。此外，在工程认证中，制约因素分析与设计和工程师责任及其分析也是必不可少的内容。

此部分完成全部关卡，逐一分析全部关卡。

2.14.1 从概念模型到 MySQL 实现

本关卡已完成。按要求建表即可，要注意不要遗漏约束，本关略过分析。

2.14.2 从需求分析到逻辑模型

本关已完成。

根据题目要求画出 E-R 图如下图 2.21，具体图片存放位置见 https://cdn.jsdelivr.net/gh/niuniu0101/storageofPicture/images/ersolution.jpg。

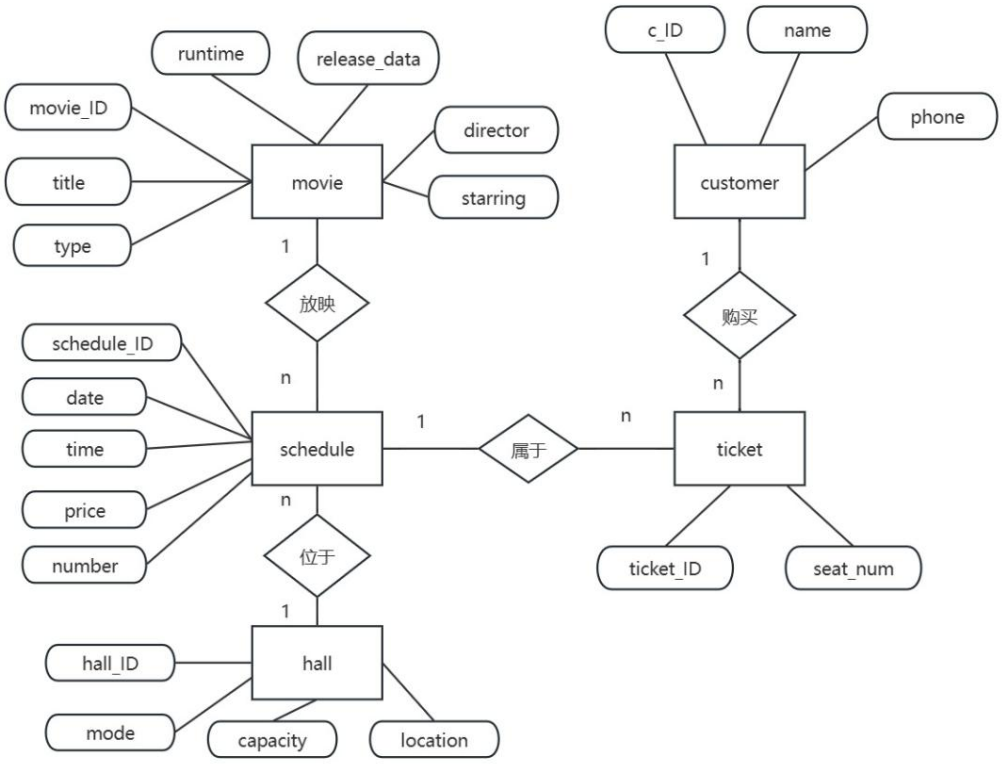

图 2.21 任务 2.13.2E-R 图

以此建立五个关系模式：

movie(movie_ID,title,type,runtime,release_data,director,starring),primary

key:(movie_ID),foreign key:(schedule_ID)

customer(c_ID,name,phone), primary key:(c_ID)

hall(hall_ID,mode,capacity,location),primary key:(hall_ID)

schedule(schedule_ID,date,time,price,number), primary key:(schedule_ID),foreign key:(hall_ID,movie_ID)

ticket(ticket_ID,seat_num), primary key:(ticket_ID),foreign key:(c_ID,schedule_ID)

2.14.3 建模工具的使用

本关卡已完成。受篇幅限制，本关略过分析。

2.14.4 制约因素分析与设计

在实际工程中设计数据库时，除了满足客户需求外，还必须综合考虑社会、健康、安全、法律、文化及环境等制约因素。

社会层面：数据库设计应考虑用户友好性、数据透明性和隐私保护，确保公平性和无歧视性。设计要基于社会责任，满足不同背景用户的需求。

健康层面：确保健康相关数据的准确性和实时性，减少用户心理负担，提供简洁的操作界面和清晰的健康数据反馈，以帮助用户有效管理健康。

安全层面：设计方案必须保障数据安全与信息安全，采用数据加密、访问控制和多因素身份验证等措施，防止数据泄露和非法访问。

法律层面：设计必须合法合规，遵守相关数据保护法规，如 GDPR 和 HIPAA，保障用户权益，不利用数据库进行非法活动。

文化层面：设计应尊重不同文化背景，提供多语言支持和本地化服务，遵守工程职业道德和规范，维护用户权益。

环境层面：设计应优化数据库性能，减少能耗，采用绿色计算技术和云计算支持，考虑可持续发展原则，降低对环境的影响。

2.14.5 工程师责任及其分析

在产品实现过程中，社会、安全、法律及文化等因素对解决方案起到了制约和调整作用。工程师应承担一定的生态伦理责任，准确有效地说明新建工程或新技术可能带来的后果，避免对社会和生态环境的危害；同时，应承担职业伦理责

任，秉持追求真理、客观求实、诚实公平的精神；并承担保护公众安全、健康和福祉的责任，不仅要遵守设计规则和标准，还应考虑产品或技术在市场上的效用。

工程师在设计、实现和维护数据库过程中，需综合考虑前述各种制约因素，设计出既能满足用户需求，又能在其他方面达到最优的数据库。同时，工程师必须遵守法律，谨慎行事，不可投机取巧，不走歪门邪道。尤其在信息技术行业，由于数据库缺陷造成的损失可能是巨大的。

此外，中国的工程师还需肩负技术发展和提升国家综合实力的责任，不断进行创新设计，推动国家整体进步。通过综合考虑社会、安全、法律及文化等因素，工程师可以提供安全、可靠、高效的数据库解决方案，满足用户需求并促进社会和技术的可持续发展。

2.15 数据库应用开发（JAVA 篇）

JDBC（Java DataBase Connectivity，java 数据库连接）是一种用于执行 SQL 语句的 Java API，可以为多种关系数据库提供统一访问，它由一组用 Java 语言编写的类和接口组成。JDBC 提供了一种基准，据此可以构建更高级的工具和接口，使数据库开发人员能够编写数据库应用程序。

此部分实验完成所有的关卡，重点分析第 3 关到第 7 个关卡，由于篇幅限制，省略代码。

2.15.3 添加新客户

编程完成向 client(客户表)插入记录的方法。已知用户的 id、姓名、邮箱、身份证号、电话和登陆密码，将上述信息与已连接的数据库的 Connection 对象传入方法中。可以使用 PreparedStatement 接口进行 SQL 语句的传递。将上述用户的相关信息插入到 SQL 语句中，使用该接口的 executeUpdate()方法，实现对于元组的插入操作。

具体的关键代码见图 2. 22。

```
String sql = "insert into client values(?, ?, ?, ?, ?, ?)";
    try {
        PreparedStatement pps = connection.prepareStatement(sql);
        pps.setInt(1, c_id);
        pps.setString(2, c_name);
        pps.setString(3, c_mail);
        pps.setString(4, c_id_card);
        pps.setString(5, c_phone);
        pps.setString(6, c_password);
        return pps.executeUpdate();
        } catch (SQLException e) {
            e.printStackTrace();
        }
        return 0;
```

图 2.22 任务 2.14.3 关键代码

2.15.4 银行卡销户

编写一个能删除指定客户编号的指定银行卡号的方法。具体方法与插入类似。已知用户的 id 和银行卡号，将上述信息与已连接的数据库的 Connection 对象传入方法中。可以使用 PreparedStatement 接口进行 SQL 语句的传递。将上述用户的相关信息插入到 SQL 语句中，使用该接口的 executeUpdate()方法，实现对于元组的删除操作。

2.15.5 客户修改密码

编写修改客户登录密码的的方法。具体方法也与插入类似。首先通过一系列方法获取需要修改的用户，同时获得其旧密码，进一步对于旧密码进行更新，将用户的邮箱、旧密码和新密码与已连接的数据库的 Connection 对象传入方法中。可以使用 PreparedStatement 接口进行 SQL 语句的传递。将上述用户的相关信息插入到 SQL 语句中，使用该接口的 executeUpdate()方法，实现对于元组的更新操作。

2.15.6 事务与转账操作

使用 JDBC 的事务处理编写一个银行卡转账方法。需要传入已连接数据库的 Connection 对象、转出账号、转入账号和转账金额。首先设置隔离级别为 REPEATABLE READ，具体实现为 connection. SetTransactionIsolation(4)。然后，进行转出账号扣账，转入账号还账，为储蓄卡时入账。然后进行操作合法性的检验，如果操作不合法，例如账号不存在，扣账金额不足等，则进行事务回滚，具

体实现为 connection.rollback()，且置返回值为 false。其他情况下则进行事务提交，具体实现为 connection.commit()，且置返回值为 true。

2.15.7 把稀疏表格转为键值对存储

将一个稀疏的表中有保存数据的列值，以键值对(列名，列值)的形式转存到另一个表中，这样可以直接丢失没有值列。具体来说，首先需要查询稀疏表中的所有元组，进行学生学号和各科成绩的提取，然后将其填入键值对表中。 然后，将已连接的数据库的 Connection 对象和键值对传入方法中。可以使用 PreparedStatement 接口进行 SQL 语句的传递。将上述用户的相关信息插入到 SQL 语句中，使用该接口的 executeUpdate()方法，实现对于元组的插入操作。

2.16 存储管理(Storage Manager)

这些任务共同构成了一个小型关系数据库管理系统（RMDB）核心存储与记录管理功能的实现。首先，DiskManager 类作为最底层，直接与操作系统交互，负责将文件抽象为页面序列，管理文件的创建、删除、打开、关闭，以及页面数据的物理读写、页面编号的分配。其次，LRUReplacer 类实现了一种具体的页面替换策略——最近最少使用算法，当缓冲池需要空间时，它能决策出哪个页面帧应该被置换出去。接着，BufferPoolManager 类是内存缓冲区的管理者，它协调 DiskManager 和 LRUReplacer 的工作，负责从磁盘读取页面到内存缓冲帧、将修改过的（脏）页面写回磁盘、提供页面的上锁（Pin）与解锁（Unpin）机制，以及在新页面分配和页面获取请求时，管理缓冲池中的有限帧资源。最后，在这些存储管理基础之上，RmFileHandle 类和 RmScan 类构成了记录管理器的一部分。RmFileHandle 针对单个数据文件，提供记录级的操作接口，如插入、删除、更新和获取记录，它内部管理着数据页面的布局（如页头、记录槽位、位图）和空闲空间。而 RmScan 类则实现了记录的迭代器功能，允许用户顺序遍历文件中的所有有效记录。这四个组件层层递进，从物理 I/O 到内存缓冲，再到逻辑记录操作，共同奠定了数据库数据持久化存储和高效访问的基础，由于代码过多，不再给出，下面是分点描述：

2.16.1 DiskManager 类实现思路

DiskManager 的核心思路是封装底层文件 I/O，提供面向页面的操作。页面读写（read_page/write_page）通过页号计算文件内偏移（page_no * PAGE_SIZE），使用 lseek 定位后调用 read/write。AllocatePage 采用原子自增方式为指定文件分配新页号，更新内部的 fd 与页号计数映射。文件操作如 is_file/is_dir 依赖 stat 系统调用；create_file/open_file 使用 open 系统调用（O_CREAT 或 O_RDWR 模式），并维护 path2fd 和 fd2path 映射表来跟踪已打开文件，防止重复打开并支持双向查询；close_file 关闭 fd 并更新映射；destroy_file 使用 unlink 删除文件，需确保文件已关闭。GetFileSize 等也基于 stat 或映射表。

2.16.2 LRUReplacer 类实现思路

LRUReplacer 的思路是实现最近最少使用页面替换策略。核心数据结构采用双向链表（维护帧的 LRU 顺序，头部为最少使用，尾部为最近使用）和哈希表（映射 frame_id 到链表节点迭代器，实现快速访问）。Victim 操作从链表头部移除并返回最久未使用的帧，并更新哈希表。Pin 操作当帧被固定时，若它在 LRU 链表中（可替换），则将其从中移除。Unpin 操作当帧解除固定且 pin_count 为 0 时，将其添加到链表的 MRU 端（尾部），并更新哈希表。所有公共方法需使用 std::mutex 保证线程安全。

2.16.3 BufferPoolManager 类实现思路

BufferPoolManager 作为内存缓冲池管理器，其思路是协调磁盘与内存间页面数据的高效移动。它维护一个 Page 对象数组（帧池）、一个 page_table（PageId 到 frame_id 的映射）、一个 free_list（空闲帧列表），并使用 DiskManager 和 Replacer。NewPage 时，先尝试从 free_list 获取帧，若无则调用 Replacer 的 Victim 选择牺牲帧；若牺牲帧为脏页，则先写回磁盘。然后通过 DiskManager 分配新磁盘页，更新 page_table，初始化 Page 对象（pin_count=1），并 Pin 住该帧。FetchPage 时，先查 page_table；若命中，增加 pin_count 并 Pin 住；若未命中，同样获取帧（可能牺牲脏页并写回），通过 DiskManager 从磁盘读取页面内容，更新 page_table 和 Page 对象，然后 Pin 住。UnpinPage 减少 pin_count，

若为 0 则调用 Replacer 的 Unpin 使帧可被替换，并处理脏页标记。DeletePage 需在 pin_count 为 0 时进行，从 page_table 移除，帧加入 free_list，并通知 DiskManager 回收磁盘页。FlushPage/FlushAllPages 负责将指定脏页或所有脏页写回磁盘。并发控制对共享数据结构至关重要。

2.16.4 RmFileHandle 和 RmScan 类实现思路

RmFileHandle 的思路是管理单个文件内的记录，它依赖 BufferPoolManager 获取页面。文件通常有文件头（存储元数据如记录大小、空闲页链表头）和数据页。数据页内有页头（含记录位图、空闲槽信息）。insert_record 时，通过文件头的空闲页指针或创建新页找到有空间的页面，在位图中找到空闲槽位存入记录，更新位图和页头/文件头元数据。delete_record 则标记位图，并可能因页面变空闲而更新空闲页链表。get_record 根据 Rid 读取位图和对应槽数据。
update_record 直接修改槽数据。 RmScan 的思路是实现记录迭代。它内部保存当前记录的 Rid 和文件句柄。next()方法从当前 Rid 的下一槽位开始，在当前页利用位图查找下一有效记录;若当前页无更多记录,则获取下一数据页继续查找，直至文件末尾。is_end()判断是否已扫描完毕。rid()返回当前指向的记录 ID。

3 课程总结

通过这次数据库系统原理的实践，我全面完成了从实训 1 到实训 16 的所有任务，深刻体验了数据库的基础操作，包括创建、删除、修改、查询等核心功能。在此基础上，我进一步探索了数据库的安全性控制、并发控制、自定义函数、数据库恢复等关键技术，这些技术对于数据库系统的运行和维护具有重要意义。通过这次实践，我对数据库管理系统的整体框架和特点有了更加深入的了解，同时也加深了对数据库及其上层应用程序的认识。

在具体实训任务中，我掌握了数据定义语言 DDL 的使用，学会了如何定义数据库、表以及完整性约束条件；我掌握了如何利用 SQL 语句修改表结构和约束条件；在数据查询方面，我熟练掌握了基本查询方法以及查询过程中所用到的函数、技巧；同时，我也完成了数据的插入、删除、修改、连接更新等相关任务。

此外，我还学会了创建视图与基于视图的查询，更加理解了模式和外模式之间的映像关系，以及应用程序是如何基于视图这一层次进行查询的。在存储过程与事务方面，我掌握了三种存储过程的具体结构和实现方法，并完成了相关实训任务。同时，我也掌握了触发器和用户自定义函数的设计方法。

在数据库安全性方面，我学会了创建用户、角色以及权限授予，掌握了自主存取方式对于用户权限的设置和对于数据库的保护。在事务并发控制与隔离级别方面，我加深了对读脏、不可重复读、幻读、可串行化等概念的理解，并掌握了数据库事务的加锁操作。

最后，我还完成了使用备份和日志文件实现数据恢复的相关任务，对数据库恢复技术有了更加深入的理解。同时，我也掌握了数据库设计的全过程，包括概念模型设计、关系模式设计、建模实现设计等。通过这次实践，我还学会了如何利用 JDBC 体系实现数据库应用设计，这对于我未来的专业成长具有重要意义。

“纸上得来终觉浅，绝知此事要躬行”，总的来说，这次实践让我获得了巨大的成长。我不仅真正了解了数据库在应用程序中的重要作用，还深入掌握了如何通过数据库的系列操作实现数据的存储、管理和应用。特别是在数据查询方面，我以前从未想过可以使用 SQL 语句完成如此复杂的操作。这次实践让我更加自信地面对未来的专业挑战，并为我未来的职业发展奠定了坚实的基础。