第 3 章 高级数据管理

3.1 实验介绍

本章将实践 openGauss 数据库的高级数据管理功能，涉及各类数据库对象，包括：视图、索引、存储过程、触发器、事务管理、权限管理等。视图可用于简化查询和保护数据，索引用于提高查询效率，存储过程用于封装复杂业务逻辑，触发器可以自动化数据操作以维护完整性，事务管理确保数据库系统的 ACID 特性，权限管理确保数据库系统的安全性。在本章中，将通过实验操作，掌握这些 openGauss 数据库对象的使用，以验证这几类数据库对象所实现的数据库原理。

3.2 实验目的

1. 掌握 openGauss 数据库中视图的使用方法。
2. 掌握 openGauss 数据库中索引的使用方法。
3. 掌握 openGauss 数据库中存储过程的使用方法。
4. 掌握 openGauss 数据库中触发器的使用方法。
5. 掌握 openGauss 数据库中的事务管理机制。
6. 掌握 openGauss 数据库中的权限机制。

3.3 实验原理

3.3.1 视图

视图是由存储在数据库中的查询定义的虚拟表（virtual table）。视图所对应的查询称为视图定义，它规定了如何从一个或几个基本表（base table）中导出视图。数据库中只存放视图的定义，不存放视图对应的数据，数据仍存放在导出视图的基本表中。修改基本表中的数据，相应视图中的数据也随之改变。视图就像一个窗口，透过它可以看到数据库中自己感兴趣的数据。

在 SQL 中，定义视图是设计数据库外模式的基本手段。视图能够为数据库系统提供以下优点：

(1) 数据的逻辑独立性。

前面介绍过数据库的外模式和模式之间形成了数据的逻辑独立性。当数据库的模式结构发生变化时，只需调整外模式到模式的映射关系，而无需改变外模式的定义。这样做的目的是保证用户的应用程序不必重新编写。视图作为虚拟表，它的定义实际上就是外模式到模式的映射。有了视图，可以在不改变基本表结构的前提下，仅通过修改视图的定义就能够更新这种映射关系，从而实现数据的逻辑独立性。

(2) 用户操作的简化。

在实际的数据库系统中，全部基本表的结构和联系往往是复杂的。通常不同部门的用户

只对数据库中的一部分数据感兴趣。视图机制正好适应了用户的这种需要。为不同的用户定义各自的视图，使用户可以将注意力集中在所关心的数据上，用户所做的只是对虚拟表的查询，而这个虚拟表是怎样得到的，用户无需了解。

(3) 数据的安全保护。

视图机制还能够为数据提供一定的安全保护功能。只给用户访问视图的权限，对用户保密的基本表不为其定义视图。这样用户只能透过视图访问到所需的那部分数据，而不会影响到其他数据。

1. 定义视图

SQL 用 CREATE VIEW 语句在基本表上定义视图，其语法格式为：

```
CREATE VIEW <视图名> [(<列名 1>, <列名 2>, …)]
AS <子查询>
[WITH CHECK OPTION];
```

其中，<子查询>可以是任意复杂的 SELECT 语句，它定义了如何从基本表中获取视图数据。每次使用视图时，<子查询>都会执行。既然视图也是一种表（虚拟表），视图中的行是无序的，因此在定义视图的<子查询>中不允许出现 ORDER BY 子句。

WITH CHECK OPTION 选项与视图的更新有关，如果使用该选项，则对视图进行 INSERT、UPDATE 和 DELETE 操作时，要确保插入、修改或删除的行满足<子查询>中的条件表达式。如果不满足条件，则拒绝此更新操作。这样做的目的是保证更新操作的效果能够反映到视图中。

2. 删除视图

使用 DROP VIEW 语句可以删除数据库中已有的视图，语法格式为：

```
DROP VIEW <视图名> [ CASCADE ]
```

如果该视图上还导出了其他视图，则使用 CASCADE 级联删除选项，可将该视图和由它导出的所有视图一起删除。

删除某个基本表后，由该表导出的视图并没有被删除，但由于依赖的基本表已不存在，导出的所有视图均无法使用了。要删除这些视图，仍需要使用 DROP VIEW 语句。

3. 修改视图

由于视图是不实际存储数据的虚拟表，因此对视图的更新最终要转换为对基本表的更新。SQL-92 标准规定只允许更新由单张基本表通过选择和投影操作定义的视图，且视图定义中不允许使用聚合函数。这样的视图称为**可更新视图**（updatable view）。在可更新视图上的更新操作总可以通过更新相应的基本表来实现。

4. 物化视图

在有些场景下，视图定义是一个复杂且耗时的查询，这是可以使用物化视图将视图定义中的查询结果保持起来，后面查询视图时不需要进行再次执行视图定义查询。也就是说，物化视图既具有传统视图定义的优势，又不同于传统视图的虚拟表，是实际存储的数据表，能实现高效率数据访问。物化视图一般用在数据仓库及商业智能应用场景中。定义物化视图的语法格式为：

```
CREATE MATERIALIZED VIEW <物化视图名> [(<列名 1>, <列名 2>, …)]
AS <子查询>;
```

所依赖的表数据发生更新时，物化视图不会自动更新。需要使用专门的语句进行物化视图的刷新，其语法格式为：

```
REFRESH MATERIALIZED VIEW [物化视图名];
```

删除物化视图的语法格式为：

```
DROP MATERIALIZED VIEW [物化视图名];
```

3.3.2 索引

索引是数据库提供的一种存储结构，其作用是通过将已有数据组织存储为一定的结构，用以加速按条件对表中行记录的查找。如果一本书的目录或索引一样，为读者提供根据内容查找所在页位置的途径。当然，构建索引需要一定的存储空间和时间开销，索引维护也需要在数据更新时花费额外的代价。如有根据数据库的查询和更新需求，权衡决定索引的构建策略。

1. 创建索引

使用 CREATE INDEX 语句创建索引，其语法格式为：

```
CREATE INDEX <索引名>
ON <表名> (<列 1>, <列 2>, ...);
```

CREATE INDEX 语句可以在一个表的一列或多列上创建索引。默认情况下，CREATE INDEX 语句会使用 B+树数据结构来实现索引。

2. 索引类型

openGauss 提供的索引类型主要包括：

- B+树索引
  也称 btree 索引。btree 索引基于 B+树数据结构，其将数据记录维护为平衡搜索树。btree 索引可在对数时间复杂度内实现数据查找和更新的。btree 索引是最常用的一种索引类型。
- 哈希索引
  也称 hash 索引。哈希索引基于哈希函数实现，可以支持等值比较（即=运算）查找的快速执行。
- GIN 索引
  全称是 Generalized Inverted 索引，即通用倒排索引。GIN 索引一般用于加速多值列上的查找，比如，存储数组、JSON 或范围类型数据的列。
- GiST 索引
  全称是 Generalized Search Tree 索引，即通用搜索树索引。GiST 索引适用于几何和地理等多维数据类型和集合数据类型。
  可使用如下语法指定索引类型：

```
CREATE INDEX <索引名>
ON <表名> USING <索引类型> (<列 1>, <列 2>, ...);
```

其中<索引类型>取值为：btree、hash、gin 或 gist。

3.3.3 存储过程

存储过程是保存在数据库内部的函数，通过使用存储过程实现较为复杂的业务逻辑和流程。编写存储过程的代码通常要使用 SQL 的过程式编程扩展语言 PL/SQL。openGauss 继承自 PostgreSQL，其存储过程的编写语言也是 PL/pgSQL。

存储过程使执行商业规则的代码可以从应用程序中移动到数据库后端。从而，涉及业务逻辑和数据的代码只需在数据库中存储一次，就能够被多个程序使用。用户可以对存储过程

进行反复调用，从而减少 SQL 语句的重复编写数量，节省数据库服务器与客户端的通信开销，达到提高工作效率的目的。

1. 定义存储过程

定义存储过程的语法格式为：

```
CREATE [OR REPLACE] PROCEDURE <存储过程名> [ ( <参数列表> ) ]
  { IS | AS }
  BRGIN
    <存储过程代码体>
  END
```

语法说明：

- OR REPLACE

  如果存在同名的存储过程，则替换数据库中原有的存储过程。

- <参数列表>的格式为：<参数项 1>, <参数项 2>, ..., <参数项 n>

  每个参数项的格式为：

  [<参数模式>] <参数名> <数据类型> [= <缺省值>]

  <参数模式>为：

  IN：输入参数。在调用存储过程时需要从外部传入该参数到存储过程中使用。

  OUT：输出参数。表示该参数可在存储过程内部被改变，并可传出到存储过程外部。

  INOUT：输入输出参数。兼具 IN 参数和 OUT 参数的性质，可用于传入和传出值。

  若不指定参数模式，默认为 IN。

调用存储过程的语法格式为：

```
CALL <存储过程名> ( <参数表达式> );
```

删除存储过程的语法格式为：

```
DROP PROCEDURE <存储过程名>;
```

还可以使用CREATE FUNCTION语句定义函数，在openGauss中函数也是一种存储过程，其具有返回值。

```
CREATE FUNCTION <函数名> (<参数列表>)
  RETURNS <返回值类型>
AS
$$
DECLARE
 <变量声明>
BEGIN
    <代码体>
END;
$$ LANGUAGE plpgsql;
```

其中，RETURNS 指定返回值类型；AS 后面本应是一个字符串，用两个$$括起来的代码段不必考虑符号转义等，起到增加代码可读性作用。LANGUAGE plpgsql 表示编写函数的语言是 PL/pgSQL，即 openGauss 继承自 PostgreSQL 的 SQL 过程编程语言扩展。

2. 控制语句

(1) 条件语句

条件语句的主要作用是判断是否满足给定条件，根据判断结果执行相应的操作。

- IF-THEN 语句

```
IF <条件> THEN
```

```
  <语句块>
END IF;
```

- IF-THEN-ELSE 语句

```
IF <条件> THEN
  <语句块>
ELSE
  <语句块>
END IF;
```

- IF-THEN-ELSIF 语句

```
IF <条件 1> THEN
  <语句块>
ELSIF <条件 2>
  <语句块>
ELSIF <条件 3>
  <语句块>
...
ELSE
  <语句块>
END IF;
```

(2) 循环语句

- LOOP 语句

```
LOOP
  <语句>
END LOOP;
```

上面 LOOP 语句会无限次执行<语句>。一般在 LOOP 语句中要有 IF 语句，来根据<条件>执行 EXIT 语句，以停止循环。

```
LOOP
  <语句>
  IF <条件> THEN
    EXIT;
  END IF;
END LOOP;
```

- WHILE-LOOP 语句

```
WHILE <条件> LOOP
  <语句>
END LOOP;
```

只要条件表达式为真，WHILE 语句就会不停的在一系列语句上进行循环，在每次进入循环体的时候进行条件判断。

- FOR-LOOP 语句

使用 FOR-LOOP 语句，在一个范围的整数上进行循环。

```
FOR <计数变量> IN [REVERSE] <下限>..<上限> [BY <步长>] LOOP
  <语句>
END LOOP;
```

例如：

```
FOR counter IN 1..5 LOOP
    RAISE NOTICE 'counter: %', counter;
END LOOP;
```

使用 FOR-LOOP 语句，在表查询的结果集上进行循环。

```
FOR <记录变量> IN <查询语句> LOOP
  <语句>
END LOOP;
```

例如：

```
DECLARE
   row_var RECORD;
BEGIN
    FOR row_var IN SELECT * FROM student LOOP
        RAISE NOTICE '学号:%, 姓名: %', row_var.sno, row_var.sname;
    END LOOP;
END;
/
```

3. 异常处理

当存储过程执行过程中发生错误时，会退出该存储过程执行，并将所在的事务进行回滚。可以在 BEGIN-END 语句块中用 EXCEPTION 语句捕获错误并从中恢复。语法格式为：

```
[DECLARE
   <变量声明>]
BEGIN
   <语句>
EXCEPTION
   WHEN <条件> THEN
      <处理语句>
   [WHEN <条件> THEN
      <处理语句>
...]
END;
```

使用 RAISE 语句可以输出消息或报告错误。其语法格式为：

RAISE [<级别>] <格式字符串>[, <变量 1>, <变量 2>, ...];

<级别>包括 6 种：DEBUG、LOG、NOTICE、INFO、WARNING 和 EXCEPTION。不指定<级别>，默认值是 EXCEPTION。发出 EXCEPTION 级别的消息会引发报错并中止当前事务的执行。

4. 游标

如果希望对查询结果集中的每行数据进行访问并执行处理操作，就要用到游标的概念了。游标就像一个指针，可以对查询结果集数据表的进行遍历，自上而下获取每一行的数据。游标的使用分为 4 个步骤：

- 声明游标：定义一个游标；
- 打开游标：为游标赋初始值；
- 使用游标：获取游标指向的行数据并移动游标到下一行；
- 关闭游标：游标使用结束后，关闭游标。

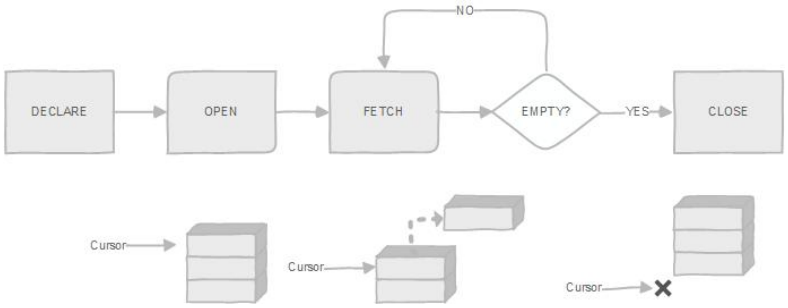

图 3.1 游标使用流程示意图

(1) 声明游标

声明游标的语法格式为：

DECLARE <游标名> CURSOR FOR <查询语句>;

打开游标后，游标将指向<查询语句>结果集的第一行。

也可以在声明游标时不指定查询，在后面打开游标时再指明。这种情况要使用 REFCURSOR 关键字：

DECLARE <游标名> REFCURSOR;

(2) 打开游标

对于声明时已指定查询的游标，可以直接打开，其语法格式为：

OPEN <游标名>;

对于声明时未指定查询的 REFCURSOR 游标，则打开时要指定查询语句，其语法格式为：

OPEN <游标名> FOR <查询语句>;

(3) 使用游标

打开游标之后，使用 FETCH 语句获取行数据，其语法格式为：

FETCH NEXT FROM <游标名> INTO <变量>;

NEXT 表示游标移动方向为下一行，是最常用的游标移动方向。可以指定的游标移动方向有：

- NEXT：下一行；
- PROR：上一行；
- FIRST：第一行；
- LAST：最后一行。

通常在 LOOP 循环结构中使用游标来处理其获取的每行数据，基本形式为：

```
LOOP
 -- 获取下一行
 FETCH NEXT FROM <游标名> INTO <变量>;
-- 如果游标遍历结束，退出循环
EXIT WHEN NOT FOUND;
-- 处理游标获取的行数据
...
END LOOP;
```

(4) 关闭游标

使用完游标后，需关闭游标，其格式为：

CLOSE <游标名>;

3.3.4 触发器

触发器是数据表上某事件发生时自动触发执行的存储过程。事件通常是指表上的数据更新操作，包括：INSERT、UPDATE 和 DELETE。

1. 创建触发器

触发器实际上是与某数据表关联的用户自定义函数。创建触发器时，首先要定义触发器函数，然后将该函数与定义在表上的触发器关联起来。

使用 CREATE FUNCTION 语句创建触发器函数，语法格式如下：

```
CREATE OR REPLACE FUNCTION <触发器函数名>()
  RETURNS TRIGGER
AS $$
BEGIN
  -- 触发器执行的具体语句
END;
$$ LANGUAGE PLPGSQL;
```

在触发器函数声明中，RETURNS TRIGGER 指明该函数用于绑定到触发器。在触发器函数中，可以使用 OLD 来获取事件发生前的行记录，使用 NEW 来获取事件发生后的行记录。

使用 CREATE TRIGGER 语句创建触发器并绑定触发器函数，语法格式如下：

```
CREATE TRIGGER <触发器名> {BEFORE | AFTER | INSTEAD OF} <事件>
  ON <表名>
  [FOR [EACH] { ROW | STATEMENT }]
  [ WHEN ( <条件> ) ]
  EXECUTE PROCEDURE <触发器函数>
```

选项说明：

- 触发器执行时间：BEFORE 表示触发器函数在事件发生之前执行，AFTER 表示触发器函数在事件发生之后执行；INSTEAD OF 表示用触发器函数直接替代事件。
- <事件>：取值包括 INSERT、UPDATE、DELETE，也可以用 OR 连接多个触发事件。
- 触发器的执行频率：FOR EACH ROW 是指触发器对于受事件影响的每一行被触发一次；FOR EACH STATEMENT 是指触发器对于事件的 SQL 语句只触发一次。

2. 修改触发器

使用 ALTER TRIGGER 语句修改触发器，语法格式为：

```
ALTER TRIGGER <触发器名> ON <表名> RENAME TO <新触发器名>;
```

3. 删除触发器

使用 DROP TRIGGER 语句修改触发器，语法格式为：

```
DROP TRIGGER <触发器名> ON <表名> [ CASCADE | RESTRICT ];
```

3.3.5 事务管理

事务是用户定义的一个数据库操作序列，这些操作要么全做要么全不做，是一个不可分割的工作单元。openGauss 数据库支持的事务控制命令有启动、设置、提交、回滚事务。openGauss 数据库支持的事务隔离级别有“读已提交”和“可重复读”。一个典型的事务例子是银行转账，转账操作必须确保转入和转出账户的一致性，也就是转出账户减少多少金额，转入账户就要增加多少金额。

事务要满足 ACID 四个特性：

- 原子性（Atomicity）：事务中的操作要么全做要么全部做；
- 一致性（Consistency）：数据更新遵守数据库中定义的规则；
- 隔离性（Isolation）：确保并发事务之间具有一定的隔离性；
- 持久性（Durability）：确保已提交事务持久地保存在数据库中。

1. 开启事务

实际上，执行任何单条 SQL 语句时，openGauss 数据库均自动开启一个只包含该单条语句的事务。开启事务的语法格式为：

```
BEGIN [ WORK | TRANSACTION ]
 [ { ISOLATION LEVEL { READ COMMITTED | REPEATABLE READ } | { READ WRITE | READ ONLY } }
 ];
```

参数说明：

- WORK | TRANSACTION：开启事务的关键字，两个都可以；
- ISOLATION LEVEL：事务隔离级别，取值包括：
  - READ COMMITTED：读已提交
  - REPEATABLE READ：可重复读
- READ WRITE | READ ONLY：指定事务访问模式，READ WRITE 为“读/写”，READ ONLY 为“只读”。

也可以将 BEGIN 关键字换作 START 关键字，作用一样。

2. 事务隔离级别

openGauss 数据库目前支持两种事务隔离级别：READ COMMITTED（读已提交）和 REPEATABLE READ（可重复读）。

- READ COMMITTED：读已提交。事务只能读到已提交的数据而不会读到未提交的数据。该值是默认值。设置为该隔离级别时，本事务中的 SELECT 语句会读取到并发执行的其他事务中已提交的数据。也就是说，SELECT 查询会查看到在查询开始运行的瞬间该数据库的一个快照。SELECT 能查看到其自身所在事务中先前更新的执行结果，即使先前更新尚未提交。在同一个事务里两个相邻的 SELECT 命令可能会查看到不同的快照，这是因为其它并发执行的事务会在第一个 SELECT 执行期间提交。
- REPEATABLE READ：可重复读。事务只能读到其开始之前已提交的数据，不能读到其开始之前未提交的数据以及事务执行期间其他并发事务提交的数据；但是，事务中的查询能查看到本事务中先前更新的执行结果，即使先前更新尚未提交。可重复读事务中的查询看到的是事务开始时的快照，不是该事务内部当前查询开始时的快照。也就是说，事务中的 SELECT 命令总是查看到同样的提交数据，查看不到本事务开始之后其他并发事务提交的数据。

3. 设置事务参数

```
{ SET [ LOCAL ] TRANSACTION | SET SESSION CHARACTERISTICS AS TRANSACTION }
  { ISOLATION LEVEL { READ COMMITTED | REPEATABLE READ } | { READ WRITE | READ ONLY } };
```

参数说明：

- LOCAL：表示该设置只在当前事务中有效；
- SESSION：表示该设置对当前会话起作用。

设置隔离级别和事务访问模式与 BEGIN TRANSACTION 语句中的参数选项一样。

3. 提交事务

使用 COMMIT 或者 END 语句提交事务，语法格式为：

{ COMMIT | END } [WORK | TRANSACTION] ;

事务提交完成后，表示事务成功完成，其效果已经反映到数据库中了。

4. 回滚事务

回滚是在事务运行的过程中发生了某种故障，不能继续执行。回滚操作会将事务中对数据库的所有已完成操作全部撤销。使用 ROLLBACK 语句回滚事务，语法格式为：

ROLLBACK [WORK | TRANSACTION];

3.3.6 权限管理

由于数据库中存储着大量重要数据，为不同权限的用户提供数据共享服务，这就要求数据库具备完善的安全防御机制，以抵抗来自内部和外部的安全攻击，保障数据不丢失、不泄露及不被篡改。openGauss 数据库具备完善的安全防御体系，保障数据库在应用中的安全性。完善的权限管理机制可以有效阻断用户的不安全操作。

1. 数据库权限层级

在 openGauss 中，数据库对象依照层级逻辑结构组织，每个数据库实例下可有多个数据库（database），每个数据库下可有多个模式（schema），每个模式下可有多个数据库对象，包括表、视图、索引、存储过程、触发器等，形成如图 3.2 所示的数据库对象层级逻辑结构。

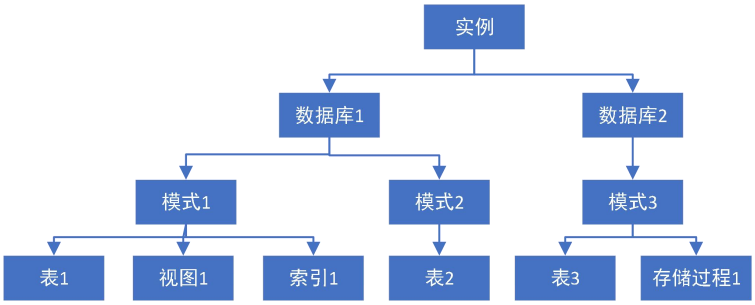

图 3.2 数据库对象层级逻辑结构

对于每个层级，数据库都有该层级的权限控制机制，包括：数据库权限、模式权限、对象级权限、表空间权限等。例如，当用户要查看一张表的数据，那么该用户需要具备登录数据库的权限（LOGIN）、表所在数据库的连接权限（CONNECT）、表所在模式的使用权限（USAGE）和表本身的查看权限（SELECT）。

2. 权限与角色

数据库对象创建后，进行对象创建的用户就是该对象的所有者。数据库安装后的默认情况下，数据库系统管理员具有与对象所有者相同的权限。也就是说对象创建后，默认只有对象所有者或者系统管理员可以查询、修改和销毁对象。openGauss 支持的主要权限及说明如表 3.1 所示。

表 3.1 openGauss 数据库支持的权限及说明

权限名称	权限说明
SELECT	允许对指定的表、视图、序列执行 SELECT 命令，UPDATE 或 DELETE 时也需要对应字段上的 SELECT 权限。
INSERT	允许对指定的表执行 INSERT 命令。
UPDATE	允许对声明的表中任意字段执行 UPDATE 命令。通常，UPDATE 命令也需要 SELECT 权限来查询出哪些行需要更新。

DELETE	允许执行 DELETE 命令删除指定表中的数据。通常，DELETE 命令也需要 SELECT 权限来查询出哪些行需要删除。
TRUNCATE	允许执行 TRUNCATE 命令删除指定表中的所有记录。
REFERENCES	创建一个外键约束，必须拥有参考表和被参考表的 REFERENCES 权限。
CREATE	o 对于数据库，允许在数据库里创建新的模式。 o 对于模式，允许在模式中创建新的对象。如果要重命名一个对象，用户除了必须是该对象的所有者外，还必须拥有该对象所在模式的 CREATE 权限。 o 对于表空间，允许在表空间中创建表，允许在创建数据库和模式的时候把该表空间指定为缺省表空间。
CONNECT	允许用户连接到指定的数据库。
EXECUTE	允许使用指定的函数，以及利用这些函数实现的操作符
USAGE	o 对于过程语言，允许用户在创建函数的时候指定过程语言。 o 对于模式，USAGE 允许访问包含在指定模式中的对象，若没有该权限，则只能看到这些对象的名称。 o 对于序列，USAGE 允许使用 nextval 函数。
ALTER	允许用户修改指定对象的属性，但不包括修改对象的所有者和修改对象所在的模式。
DROP	允许用户删除指定的对象。
COMMENT	允许用户定义或修改指定对象的注释。
INDEX	允许用户在指定表上创建索引，并管理指定表上的索引。
VACUUM	允许用户对指定的表执行 ANALYZE 和 VACUUM 操作。
ALL PRIVILEGES	一次性给指定用户/角色赋予所有可赋予的权限。只有系统管理员有权执行 GRANT ALL PRIVILEGES。

在 openGauss 数据库中，用户（USER）和角色（ROLE）是基本相同的概念，区别在于创建用户时还会默认具有 LOGIN 权限用于登录，而创建角色时不会自动带有 LOGIN 权限。一般情况下，通过用户来连接、访问数据库以及执行 SQL，通过角色来组织和管理权限。通过将不同的权限打包成角色授予用户，使得用户获得该角色中的所有权限。同时通过改变角色的权限，该角色所包含的所有成员的权限也会被自动修改。

在 openGauss 数据库中，权限分为两种：系统权限和对象权限。

- 系统权限

系统权限是指系统规定用户使用数据库的权限，比如登录数据库、创建数据库、创建用户/角色、创建安全策略等。系统权限又称用户属性，具有特定属性的用户会获得指定属性所对应的权限。

表 3.2 openGauss **数据库支持的系统权限及说明**

系统权限名称	系统权限说明
SYSADMIN	系统管理员。具有与对象所有者相同的权限。
MONADMIN	监控管理员。具有查看系统模式 DBE_PERF 下监控视图和函数并进行授权管理的权限。
OPRADMIN	运维管理员。具有使用备份工具执行数据库备份和恢复的权限。
POLADMIN	安全策略管理员。具有创建资源标签、创建动态数据脱敏策略和统一审计策略的权限。

AUDITADMIN	审计管理员。具有查看和删除审计日志的权限。
CREATEDB	允许用户创建数据库。
USEFT	允许用户创建外表。
CREATEROLE	安全管理员。具有创建、修改、删除用户或角色的权限。
INHERIT	允许用户继承所在组的角色的权限。
LOGIN	允许用户登录数据库。
REPLICATION	允许用户执行流复制相关操作。

- 对象权限

对象权限是指在数据库、模式、表、视图、函数等数据库对象上执行操作的权限。不同对象类型与不同权限相关联，比如数据库的连接权限，表的查看、插入、更新等权限，函数的执行权限等。

表 3.3 openGauss **数据库支持的对象权限及说明**

对象	权限名称	对象权限说明
TABLESPACE	CREATE	允许用户在指定的表空间中创建表。
	ALTER	允许用户修改指定表空间的属性。
	DROP	允许用户删除指定的表空间。
	COMMENT	允许用户对指定的表空间定义或修改注释。
DATABASE	CONNECT	允许用户连接到指定的数据库。
	TEMP	允许用户在指定的数据库中创建临时表。
	CREATE	允许用户在指定的数据库中创建模式。
	ALTER	允许用户修改指定数据库的属性。
	DROP	允许用户删除指定的数据库。
	COMMENT	允许用户对指定的数据库定义或修改注释。
SCHEMA	CREATE	允许用户在指定的模式中创建对象。
	USAGE	允许用户访问指定模式中的对象。
	ALTER	允许用户修改指定模式的属性。
	DROP	允许用户删除指定的模式。
	COMMENT	允许用户对指定的模式定义或修改注释。
FUNCTION	EXECUTE	允许用户执行指定的函数。
	ALTER	允许用户修改指定函数的属性。
	DROP	允许用户删除指定的函数。
	COMMENT	允许用户对指定的函数定义或修改注释。
TABLE	INSERT	允许用户对指定的表执行 INSERT 语句插入数据。
	DELETE	允许用户对指定的表执行 DELETE 语句删除数据。
	UPDATE	允许用户对指定的表执行 UPDATE 语句更新数据。
	SELECT	允许用户对指定的表执行 SELECT 语句查询数据。
	TRUNCATE	允许用户对指定的表执行 TRUNCATE 语句删除所有记录。
	REFERENCES	允许用户对指定的表创建一个外键约束。
	TRIGGER	允许用户在指定的表上创建触发器。
	ALTER	允许用户修改指定表的属性。
	DROP	允许用户删除指定的表。
	INDEX	允许用户在指定表上创建索引并管理指定表上的案引。

	VACUUM	允许用户对指定的表执行 ANALYZE 和 VACUUM 操作。
	COMMENT	允许用户对指定的表定义或修改注释。

openGauss 提供了一个隐式定义的拥有所有角色的组 PUBLIC（也是关键字），所有创建的角色将默认拥有 PUBLIC 组中的权限。也就是说，通过 CREATE ROLE 创建的角色将拥有 PUBLIC 组中的权限。PUBLIC 默认拥有的权限包括：

- 数据库的 CONNECT 权限和 CREATE TEMP TABLE 权限
- 函数的 EXECUTE 权限
- 语言和数据类型的 USAGE 权限

3. 权限授予与回收

授予与回收权限分别使用 SQL 中的 GRANT 和 REVOKE 语句。

- 授予权限

对象所有者为使其他用户能够使用对象，必须向用户或包含该用户的角色授予必要的权限，通过 GRANT 语句将对象的权限授予其他用户。

GRANT 语句的语法形式为：

```
GRANT <权限> ON <数据库对象> TO <角色> [WITH GRANT OPTION];
```

如果声明了 WITH GRANT OPTION，则被授权的用户也可以将该权限赋予其他用户，否则就不能将该权限再授权给其他用户。

- 回收权限

要撤销已经授予的权限，可以使用 REVOKE。对象所有者的权限（例如 ALTER、DROP、COMMENT、INDEX、VACUUM、GRANT 和 REVOKE）是隐式拥有的，即只要拥有对象就可以执行对象所有者的这些隐式权限。对象所有者可以撤销自己的普通权限，例如，使表对自己以及其他人只读，系统管理员用户除外。

REVOKE 语句的语法形式为：

```
REVOKE <权限> ON <数据库对象> FROM <角色> [CASCADE];
```

如果用户 A 将权限授予用户 B，用户 B 又将该权限授予用户 C，默认情况下用户 A 使用 REVOKE 语句回收用户 B 的权限会失败，使用 CASCADE 选项的 REVOKE 语句可以使用户 A 回收用户 B 的权限，并且级联回收用户 B 授予出去的该权限。

- 授权图

可以使用授权图来表示数据库中用户之间的授权关系。授权图的节点表示某用户拥有某权限且是否为该权限的源头所有者或是否能继续授权其他用户，节点信息的一般格式为：

A P [**|*]

其中，A 为用户，P 为权限，**表示权限的源头所有者，*表示可继续授权（即 WITH GRANT OPTION）。每个授权 GRANT 操作都将在授权图中添加一条对应的边。

例如，用户 omm 是 student 表的所有者，其将该表的 SELECT 权限授予 myrole 用户并使用了 WITH GRANT OPTION；myrole 用户又将该权限授予了 newrole 用户并没有使用 WITH GRANT OPTION。此时形成的授权图如图 3.3 所示。

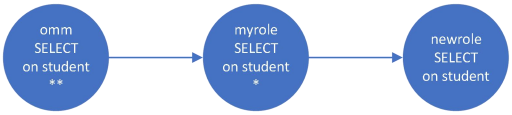

图 3.3 授权图示例

此时，omm 用户执行 REVOKE 语言回收授予 myrole 用户的权限，数据库系统会拒绝执

行，原因是 myrole 用户授予 newrole 用户的权限依赖于要删除的权限；omm 用户使用带有 CASCADE 选项的 REVOKE 语句回收授予 myrole 用户的权限，同时 myrole 用于授予 newrole 用户的权限也会被级联删除，这时授权图会只有“omm SELECT on student **”节点。

- 用户权限集合

根据 openGauss 数据库的权限管理机制，一个用户实际拥有的权限是以下 5 类权限的并集：

- CREATE ROLE 和 ALTER ROLE 语句直接赋予的系统权限
- 创建对象的所有者所默认拥有的权限
- GRANT 语句直接赋予的对象权限
- 继承自所属角色的权限
- 继承自 PUBLIC 组的权限

3.4 实验步骤

3.4.1 使用视图

1. 定义视图

【查询 1】建立 CS 系学生的视图。

```
CREATE VIEW student_cs
AS
    SELECT sno, sname, sgender, sbirth, sdept
    FROM student
    WHERE sdept = 'CS';
```

该视图没有指定列名，则视图各列的名称与查询结果各列名一致。openGauss 处理 CREATE VIEW 语句，只是将视图定义存入数据库，不执行子查询。只有当使用视图时，才按照视图定义从表中查询数据。

对定义好的视图进行查询与查询表完全相同。例如，下面 SELECT 语句列出该视图的全部数据。

```
SELECT * FROM student_cs;
  sno   | sname | sgender |       sbirth        | sdept
-------+-------+---------+---------------------+-------
 22001 | 张三  | 男      | 2002-05-19 00:00:00 | CS
 22002 | 李四  | 女      | 2002-01-09 00:00:00 | CS
(2 rows)
```

【查询 2】建立 CS 系学生的视图，且要求修改和插入数据时保证该视图只有 CS 系的学生。

```
CREATE VIEW student_cs_2
    AS
        SELECT sno, sname, sgender, sbirth, sdept
        FROM student
        WHERE sdept = 'CS'
WITH CHECK OPTION;
```

在定义视图 student_cs_2 时，加上了 WITH CHECK OPTION 子句，则插入和修改数据时必须满足查询条件 sdept='CS'，否则，数据库会拒绝该更新操作。

【查询 3】建立 CS 系选修了数据库系统课程的学生的视图。

```
CREATE VIEW student_cs_3(sno, sname, score)
AS
    SELECT student.sno, sname, score
    FROM student INNER JOIN sc ON student.sno = sc.sno
                 INNER JOIN course on sc.cno = course.cno
    WHERE sdept = 'CS' AND cname = '数据库系统';
```

视图 student_cs_3 建立在三张表连接查询之上。查看该视图：

```
dbsc=# SELECT * FROM student_cs_3;
  sno   | sname | score
-------+-------+-------
```

```
 22001 | 张三  | 85
 22002 | 李四  | 90
(2 rows)
```

可以在一个视图的基础上再建立视图。

【查询 4】建立 CS 系选修了数据库系统课程且成绩在 90 分以上的学生的视图。

```
CREATE VIEW student_cs_4
AS
    SELECT sno, sname, score
    FROM student_cs_3
    WHERE score >= 90;
```

查询该视图：

```
dbsc=# SELECT * FROM student_cs_4;
  sno  | sname | score
-------+-------+-------
 22002 | 李四  | 90
(1 row)
```

设计数据库时，为了减少数据冗余，表中一般只存储基本数据列，由基本数据列经过计算得到的数据列一般是不存储的。但有时某些应用需要集中使用这种经过计算得到的数据列。在这种情况下，定义视图会简化查询编写。

【查询 5】定义一个反映学生年龄的视图。

```
CREATE VIEW student_age(sno, sname, sage)
AS
    SELECT sno, sname, EXTRACT('YEAR' FROM AGE(CURRENT_DATE, sbirth))
    FROM student;
```

sage 列是 student 表的 sbirth 列经过计算得到的虚拟列，因为其不对应于基本表中的任何属性列。查询该视图：

```
dbsc=# SELECT * FROM student_age;
  sno  | sname | sage
-------+-------+------
 22001 | 张三  |   21
 22002 | 李四  |   22
 22003 | 王五  |   22
 22004 | 赵六  |   22
(4 rows)
```

可以使用 gsql 元命令\dv 列出当前数据库中定义的视图信息：

```
dbsc=# \dv
                      List of relations
 Schema |     Name     | Type | Owner |         Storage
--------+--------------+------+-------+-------------------------
 public | student_age  | view | omm   |
 public | student_cs   | view | omm   |
 public | student_cs_2 | view | omm   | {check_option=cascaded}
 public | student_cs_3 | view | omm   |
 public | student_cs_4 | view | omm   |
```

```
(5 rows)
```

2. 查询视图。

视图定义后，用户就可以像对表一样对视图进行查询了。

【查询 6】在视图 student_age 中查找年龄小于 22 岁的学生。

```
SELECT sno, sname, sage
FROM student_age
WHERE sage < 22;
```

一个查询可以同时涉及到视图和表。例如，下面查询对视图和表进行了连接查询：

【查询 7】查询 CS 系中选修了 2 号课程的学生。

```
SELECT s.sno, sname
FROM student_cs s INNER JOIN sc ON s.sno = sc.sno
WHERE cno = 2;
  sno  | sname
-------+-------
 22001 | 张三
 22002 | 李四
(2 rows)
```

3. 修改视图。

视图是不实际存储数据的虚拟表，视图的更新最终要转换为对表的更新。按照 SQL 标准规定，只允许更新由单张表通过选择和投影操作定义的视图，且视图定义中不能使用聚合函数，这类视图称为可更新视图。在可更新视图上的更新操作总可以通过更新背后的表来实现。

【查询 8】向 CS 系学生视图 student_cs 中插入一条新记录。

（学号：22005，姓名：孙七，性别：男，出生日期：2001-11-01，系别：CE）

```
INSERT INTO student_cs VALUES ('22005', '孙七', '男', '2001-11-01', 'CE');
```

再次查询 student_cs 视图，发现结果没有变化；查询 student 表，发现多了该行数据。

```
SELECT * FROM student;
  sno  | sname | sgender |       sbirth        | sdept
-------+-------+---------+---------------------+-------
 22001 | 张三  | 男      | 2002-05-19 00:00:00 | CS
 22002 | 李四  | 女      | 2002-01-09 00:00:00 | CS
 22003 | 王五  | 女      | 2001-12-08 00:00:00 | CE
 22004 | 赵六  | 男      | 2001-08-30 00:00:00 | IS
 22005 | 孙七  | 男      | 2001-11-01 00:00:00 | CE
(5 rows)
```

学生孙七并非 CS 系，但却能够通过 student_cs 视图插入数据到下面的 student 表中，这种情况往往会给用户带来混淆。可以在建立视图时通过 WITH CHECK OPTION 子句加以限制。

向视图 student_cs_2 中插入学生孙七的记录，该视图带有 WITH CHECK OPTION 子句：

```
dbsc=# INSERT INTO student_cs_2 VALUES ('22005', '孙七', '男', '2001-11-01', 'CE');
ERROR:  new row violates WITH CHECK OPTION for view "student_cs_2"
DETAIL:  Failing row contains (22005, 孙七, 男 , 2001-11-01 00:00:00, CE).
```

会提示要插入的行违反视图 student_cs_2 的 WITH CHECK OPTION，插入失败。
通过可更新视图也可以更改和删除已有的记录。
【查询 9】通过视图 student_cs 将学号为 22001 的学生姓名改为“张小明”。

```
UPDATE student_cs
SET sname = '张小明'
WHERE sno = '22001';
```

实际是将视图更新操作转换为等价的表更新操作:

```
UPDATE student
SET sname = '张小明'
WHERE sno = '22001' AND sdept='CS';
```

【查询 10】通过视图 student_cs 将学号为 22002 的学生记录删除。

```
DELETE
FROM student_cs
WHERE sno = '22002';
```

实际是将视图删除操作转换为等价的表删除操作:

```
DELETE
FROM student
WHERE sno = '22002' AND sdept = 'CS';
```

4. 删除视图。
要删除整个视图，需要使用 DROP VIEW 语句。
【查询 11】删除 student_cs_2 视图，同时删除所有依赖该视图的视图。

```
DROP VIEW student_cs_2 CASCADE;
```

使用级联删除选项 CASCADE，将删除所有依赖该视图的视图。
5. 物化视图。
【查询 12】建立 CS 系选修了数据库系统课程的学生的物化视图。
该视图子查询与【查询 3】中的 student_cs_3 视图相同，但这里要求建立物化视图。

```
CREATE MATERIALIZED VIEW student_cs_3m(sno, sname, score)
AS
    SELECT student.sno, sname, score
    FROM student INNER JOIN sc ON student.sno = sc.sno
                 INNER JOIN course on sc.cno = course.cno
    WHERE sdept = 'CS' AND cname = '数据库系统';
```

查询该物化视图:

```
dbsc=# SELECT * FROM student_cs_3m;
  sno  | sname | score
-------+-------+-------
 22001 | 张三  | 85
 22002 | 李四  | 90
(2 rows)
```

通过更新物化视图所依赖的 student 表中的数据，将学生张三的姓名改为张小明:

```
dbsc=# UPDATE student
dbsc-# SET sname = '张小明'
dbsc-# WHERE sname = '张三';
UPDATE 1
```

再次查询物化视图：

```
dbsc=# SELECT * FROM student_cs_3m;
  sno   | sname | score
-------+-------+-------
 22001 | 张三  | 85
 22002 | 李四  | 90
(2 rows)
```

发现物化视图的查询结果没有变化，学生张三的姓名没有发生改变。底层表数据发生更新后，出于效率考虑，物化视图并不会自动更新。需要使用如下语句显式地进行物化视图刷新：

【查询 13】刷新物化视图 student_cs_3m。

```
REFRESH MATERIALIZED VIEW student_cs_3m;
```

再次查询物化视图：

```
dbsc=# SELECT * FROM student_cs_3m;
  sno   | sname  | score
-------+--------+-------
 22001 | 张小明 | 85
 22002 | 李四   | 90
(2 rows)
```

发现查询结果更新了。

删除物化视图，使用 DROP MATERIALIZED VIEW 语句。

【查询 14】删除物化视图 student_cs_3m。

```
DROP MATERIALIZED VIEW student_cs_3m;
```

3.4.2 使用索引

1. 准备工作

   创建函数 gen_hanzi 用于生成随机汉字：

```
CREATE OR REPLACE FUNCTION gen_hanzi(INT) RETURNS TEXT AS $$
DECLARE
res TEXT;
BEGIN
    IF $1 >=1 THEN
        SELECT string_agg(chr(19968+(random()*20901)::int), '') INTO res
        FROM generate_series(1,$1);
        RETURN res;
    END IF;
    RETURN NULL;
END;
$$ LANGUAGE plpgsql STRICT;
```

   为确保实验确定性，设置随机数种子，使得每次生成的随机数序列一样。

```
SELECT setseed(0);
```

2. 装载数据

为了能够体现出索引加速查询的效果，需要表中数据量达到足够规模。使用如下 INSERT 语句，向 student 表插入 70000 行数据：

```
INSERT INTO student VALUES (
    to_char(generate_series(30000, 99999)),
    gen_hanzi(3),
    '男',
    '2001-09-01'::date + floor((random() * 365))::int,
    'CS');
```

每列数值生成方式为：

- sno：30000 到 99999
- sname：3 个随机汉字
- sgender：固定值'男'
- sbirth：日期'2001-09-01'随机增加 0~364 天
- sdept：固定值 'CS'

查询 student 表行数，确认成功装载数据。

```
dbsc=# SELECT COUNT(*) FROM student;
 count
-------
 70004
(1 row)
```

可见，student 表中原有 4 行记录，又插入了 70000 行记录。

3. 创建索引

【查询 15】在 student 表的 sname 列上创建索引，展示创建索引前后按 sname 列查询的时间变化。

使用\timing 元命令设置时间显示：

```
dbsc=# \timing on
Timing is on.
```

查询 student 表中 sname 值为“蔽芃漉”的记录：

```
dbsc=# SELECT * FROM student WHERE sname = '蔽芃漉';
  sno  | sname  | sgender |       sbirth        | sdept
-------+--------+---------+---------------------+-------
 56789 | 蔽芃漉 | 男      | 2002-07-29 00:00:00 | CS
(1 row)


Time: 15.329 ms
```

sname 值为随机生成的三个汉字，“蔽芃漉”是 sno=56789 这行记录的 sname 值。该查询在示例主机上执行时间为 15.329 毫秒。

使用 EXPLAIN 语句查看查询执行计划：

```
dbsc=# EXPLAIN SELECT * FROM student WHERE sname = '蔽芃漉';
                        QUERY PLAN
-----------------------------------------------------------
 Seq Scan on student  (cost=0.00..1464.05 rows=1 width=31)
   Filter: ((sname)::text = '蔽芃漉'::text)
(2 rows)
```

“Seq Scan”表示顺序扫描。该查询执行计划显示数据库需要执行 student 表上的顺序扫描操作，即对表中数据从头开始进行行扫描。这种执行计划对于大规模表数据来说效率并不高，因为在最坏情况下，需要检查表中所有行记录才能返回结果。

下面在 sname 列上创建索引：

```
CREATE INDEX sname_idx ON student(sname);
```

通过查询系统表 pg_indexes，查看表 student 上建立的全部索引：

```
dbsc=# SELECT indexname, indexdef
dbsc-# FROM pg_indexes
dbsc-# WHERE tablename = 'student';
 indexname  |                                  indexdef
------------+-------------------------------------------------------------------------------
 pk_student | CREATE UNIQUE INDEX pk_student ON student USING btree (sno) TABLESPACE pg_default
 sname_idx  | CREATE INDEX sname_idx ON student USING btree (sname) TABLESPACE pg_default
(2 rows)
```

输出显示 sname_idx 是刚建立的索引，“USING btree”表示默认情况下该索引使用了 btree 结构实现；pk_student 是数据库在创建 student 表时，为主键 sno 自动建立的索引。

也可以使用 gsql 元命令\d 查看一个表上建立的索引：

```
dbsc=# \d student
               Table "public.student"
 Column  |             Type              | Modifiers
---------+-------------------------------+-----------
 sno     | character(5)                  | not null
 sname   | character varying(10)         | not null
 sgender | character(4)                  | not null
 sbirth  | timestamp(0) without time zone | not null
 sdept   | character(2)                  | not null
Indexes:
    "pk_student" PRIMARY KEY, btree (sno) TABLESPACE pg_default
    "sname_idx" btree (sname) TABLESPACE pg_default
Referenced by:
    TABLE "sc" CONSTRAINT "fk_student" FOREIGN KEY (sno) REFERENCES student(sno)
```

4. 使用索引

再次执行上面查询：

```
dbsc=# SELECT * FROM student WHERE sname = '蔽芃漉';
  sno  | sname  | sgender |       sbirth        | sdept
-------+--------+---------+---------------------+-------
 56789 | 蔽芃漉 | 男      | 2002-07-29 00:00:00 | CS
(1 row)


Time: 0.880 ms
```

查询执行时间为 0.880 毫秒，是创建索引之前查询耗时的 5.741%。

再次查看查询执行计划：

```
dbsc=# EXPLAIN SELECT * FROM student WHERE sname = '蔽芃漉';
```

```
                                QUERY PLAN
------------------------------------------------------------------------
 Index Scan using sname_idx on student  (cost=0.00..8.27 rows=1 width=31)
   Index Cond: ((sname)::text = '蔽芟澹'::text)
(2 rows)
```

这时查询执行计划改为了使用“Index Scan”，即在 student 表的 sname_idx 索引上的索引查找操作。验证了使用索引可以大幅减少查询时间，提升查询效率。

5. 删除索引

删除索引所用 DROP INDEX 语句，需要指定索引名称：

【查询 16】删除索引 sname_idx。

```
DROP INDEX sname_idx;
```

6. 多列索引

多列索引是建立在一个表中两个及以上列上的索引结构，其可加速以多个列为条件的查询。

【查询 17】在 student 表的 sname 和 sbirth 列上创建多列索引。

查看以下查询的执行计划：

```
dbsc=# EXPLAIN SELECT * FROM student WHERE sname = '蔽芟澹' AND sbirth = '2002-07-29';
                                                 QUERY PLAN
-------------------------------------------------------------------------------------
----------------------------
 Seq Scan on student  (cost=0.00..1639.06 rows=1 width=31)
   Filter:  (((sname)::text  =  ' 蔽 芟 澹 '::text)  AND  (sbirth  =  '2002-07-29
00:00:00'::timestamp without time zone))
(2 rows)
```

可见，因为前面 sname 上的索引已删除，该查询只能使用顺序扫描。

```
dbsc=# SELECT * FROM student WHERE sname = '蔽芟澹' AND sbirth = '2002-07-29';
  sno  | sname  | sgender |       sbirth        | sdept
-------+--------+---------+---------------------+-------
 56789 | 蔽芟澹 | 男      | 2002-07-29 00:00:00 | CS
(1 row)


Time: 15.787 ms
```

实际查询耗时为 15.787 毫秒。

在 student 表的 sname 和 sbirth 列上创建多列索引：

```
CREATE INDEX sname_sbirth_idx ON student(sname, sbirth);
```

再次查看查询执行计划：

```
dbsc=# EXPLAIN SELECT * FROM student WHERE sname = '蔽芟澹' AND sbirth = '2002-07-29';
                                                    QUERY PLAN
-------------------------------------------------------------------------------------
--------------------------------
 Index Scan using sname_sbirth_idx on student  (cost=0.00..8.27 rows=1 width=31)
   Index  Cond:  (((sname)::text  =  ' 蔽 芟 澹 '::text)  AND  (sbirth  =  '2002-07-29
00:00:00'::timestamp without time zone))
(2 rows)
```

可见，查询条件中包括 sname 和 sbirth 两列，恰好使用上刚刚创建的 sname_sbirth_idx 索引。

再次执行该查询：

```
dbsc=# SELECT * FROM student WHERE sname = '蔽芃漉' AND sbirth = '2002-07-29';
  sno   | sname  | sgender |       sbirth        | sdept
-------+--------+---------+---------------------+-------
 56789 | 蔽芃漉 | 男      | 2002-07-29 00:00:00 | CS
(1 row)


Time: 0.727 ms
```

查询执行时间缩短为 0.727 毫秒。

在创建多列索引时，列的顺序是有所考虑的，因为多列索引能够支持从左向右的部分列作为条件的查询，其他的列顺序作为查询条件并不支持。

查看以下查询的执行计划：

```
dbsc=# EXPLAIN SELECT * FROM student WHERE sname = '蔽芃漉';
                                  QUERY PLAN
-------------------------------------------------------------------------------
 Index Scan using sname_sbirth_idx on student  (cost=0.00..8.27 rows=1 width=31)
   Index Cond: ((sname)::text = '蔽芃漉'::text)
(2 rows)
```

可以看到，WHERE 后面条件中虽只有 sname，但其是多列索引(sname, sbirth)的从左到右部分列（前缀），因而使用了 sname_sbirth_idx 索引进行了查找。

再看以下查询的执行计划：

```
dbsc=# EXPLAIN SELECT * FROM student WHERE sbirth = '2002-07-29';
                               QUERY PLAN
-------------------------------------------------------------------------
 Seq Scan on student  (cost=0.00..1464.05 rows=191 width=31)
   Filter: (sbirth = '2002-07-29 00:00:00'::timestamp without time zone)
(2 rows)
```

这个查询条件中只有 sbirth 列，且不是多列索引的前缀部分，因此无法利用索引查找，只能采用顺序扫描。

删除该索引：

```
DROP INDEX sname_sbirth_idx;
```

3.4.3 使用存储过程

1. 创建存储过程

   【查询 18】创建一个存储过程，用于向 sc 表中插入一条记录。

```
CREATE PROCEDURE insert_sc (param1 CHAR(5), param2 SMALLINT, param3 SMALLINT = 0)
IS
BEGIN
    INSERT INTO sc VALUES (param1, param2, param3);
END;
/
```

使用 CREATE PROCEDURE 语句创建名为 insert_sc 的存储过程。3 个参数的数据类型与 sc 表的 3 列数据类型相同，指明了 param3 的默认值为 0。注意，最后一行的斜线“/”表示存储过程结束。

【查询 19】创建一个存储过程，用于删除 sc 表中一条指定学号和课号的记录。

```
CREATE PROCEDURE delete_sc (param1 CHAR(5), param2 SMALLINT)
IS
BEGIN
    DELETE FROM sc
    WHERE sc.sno = param1 AND sc.cno = param2;
END;
/
```

2. 调用存储过程

使用 CALL 语句调用存储过程。

【查询 20】调用存储过程 insert_sc，向 sc 表中插入记录：

```
'22003', 1, 86
```

```
CALL insert_sc ('22003', 1, 86);
```

查看 sc 表：

```
dbsc=# select * from sc;
  sno  | cno | score
-------+-----+-------
 22001 |   1 | 96
 22001 |   2 | 85
 22001 |   3 | 88
 22002 |   2 | 90
 22002 |   3 | 80
 22003 |   1 | 86
(6 rows)
```

【查询 21】调用存储过程 delete_sc，将'22003'选修 1 号课程的记录删除。

```
CALL delete_sc ('22003', 1);
```

查看 sc 表：

```
dbsc=# select * from sc;
  sno  | cno | score
-------+-----+-------
 22001 |   1 | 96
 22001 |   2 | 85
 22001 |   3 | 88
 22002 |   2 | 90
 22002 |   3 | 80
(5 rows)
```

3. 输出参数与返回值

存储过程没有返回值。如果想要从存储过程中输出数据，可以使用 OUT 或 INOUT 类型的参数。

【查询 22】创建一个存储过程，根据指定的学号返回学生姓名。

```
CREATE OR REPLACE PROCEDURE get_sname(IN stu_num TEXT, OUT stu_name TEXT)
```

```
AS
BEGIN
   SELECT sname
   INTO stu_name
   FROM student
   WHERE sno = stu_num;
END;
/
```

如果直接调用该存储过程，无论在第 2 个参数传入的什么值，都会打印列名是 stu_name 的一行表结果：

```
dbsc=# CALL get_sname('22001', 'any_value');
 stu_name
----------
 张三
(1 row)
```

可以把 OUT 参数传出来的值保存到变量中。如下面代码段：

```
DECLARE
   var_out TEXT;
BEGIN
   get_sname('22001', var_out);
   RAISE NOTICE '姓名: %', var_out;
END;
/
```

这里，在 DECLARE 部分，定义了变量 var_out；函数 get_sname 调用后，OUT 参数 stu_name 的值传出来赋值给了 var_out；使用 RAISE NOTICE 语句将 var_out 作为日志消息输出。

```
NOTICE:  姓名: 张三
ANONYMOUS BLOCK EXECUTE
```

ANONYMOUS BLOCK EXECUTE 提示执行了匿名代码块，忽略该消息。

如果想像程序设计语言的函数那样带有返回值，需要使用 CREATE FUNCTION 语句，带有返回值的存储过程也称为函数。

【查询 23】创建一个函数，返回指定学号学生的平均成绩。

```
CREATE OR REPLACE FUNCTION get_avg_score(num CHAR(5))
    RETURNS NUMERIC(5, 2)
AS
$$
DECLARE
   avg_score NUMERIC(5, 2);
BEGIN
   SELECT AVG(score)
   INTO avg_score
   FROM sc
   WHERE sc.sno = num;
   RETURN avg_score;
```

```
END;
$$ LANGUAGE plpgsql;
```

这里 CREATE FUNCTION 语句中的 LANGUAGE plpgsql 表示该函数定义使用了 PL/pgSQL 语法；该函数的返回值数据类型为小数 NUMERIC(5, 2)；在函数体的最后一句，RETURN 语句将 SELECT 语句查询计算获得的平均成绩 AVG(score)返回。

调用该函数：

```
dbsc=# CALL get_avg_score('22001');
 get_avg_score
---------------
         89.67
(1 row)
```

可见，该调用返回了学生 22001 的平均成绩为 89.67。

4. 定义变量

在 DECLARE 语句中，除了定义标准数据类型的变量，还可以使用%TYPE 将变量定义为与某个表某列类型相同，使用%ROWTYPE 将变量定义为与某个表的行类型相同。

【查询 24】创建一个存储过程，根据指定的学号，输出学生的全部信息。

```
CREATE OR REPLACE PROCEDURE get_student(stu_num TEXT)
AS
DECLARE
    stu_name student.sname%TYPE;
    stu_row student%ROWTYPE;
BEGIN
   SELECT sname
   INTO stu_name
   FROM student
   WHERE sno = stu_num;
   RAISE NOTICE '学号: %, 姓名: %', stu_num, stu_name;
   SELECT *
   INTO stu_row
   FROM student
   WHERE sno = stu_num;
   RAISE NOTICE '学号: %, 姓名: %, 性别: %, 生日: %, 系别: %',
        stu_row.sno, stu_row.sname, stu_row.sgender, stu_row.sbirth, stu_row.sdept;
END;
/
```

在该存储过程中，定义变量 stu_name 的类型为 student 表的 sname 列类型；定义变量 stu_row 的类型为 student 表的行类型。

5. 条件判断

如果在存储过程中遇到需要进行条件判断的情况，使用 IF 语句实现。

【查询 25】创建一个函数，返回指定学号的学生是否选修了至少一门选课，是返回 1，否返回 0。

```
CREATE OR REPLACE FUNCTION is_in_sc(stu_num TEXT)
RETURNS BIT
AS $$
```

```
DECLARE ret BIT;
BEGIN
    IF EXISTS (SELECT * FROM sc WHERE sno = stu_num) THEN
        ret := 1;
    ELSE
        ret := 0;
    END IF;
    RETURN ret;
END;
$$ LANGUAGE plpgsql;
```

该函数使用了 IF…THEN…ELSE 语句来实现条件判断；返回值为 BIT 类型，即一位二进制数表示 0 或 1。

调用该函数:

```
dbsc=# CALL is_in_sc('22002');
 is_in_sc
----------
 1
(1 row)

dbsc=# CALL is_in_sc('22003');
 is_in_sc
----------
 0
(1 row)
```

6. 异常处理

【查询 26】使用 RAISE 语句输出所有级别的消息。

```
BEGIN
  RAISE DEBUG 'debug message %', now();
  RAISE LOG 'log message %', now();
  RAISE INFO 'information message %', now() ;
  RAISE NOTICE 'notice message %', now();
  RAISE WARNING 'warning message %', now();
  RAISE EXCEPTION 'exception message %', now();
end;
/
INFO:  information message 2024-03-11 23:07:25.646021+08
NOTICE:  notice message 2024-03-11 23:07:25.646021+08
WARNING:  warning message 2024-03-11 23:07:25.646021+08
ERROR:  exception message 2024-03-11 23:07:25.646021+08
```

执行该语句块，发现 DEBUG 和 LOG 级别的消息没有输出，是因为默认情况下 DEBUG 和 LOG 这两类级别较低的消息不会返回给客户端。

【查询 27】编写一个存储过程，根据学号查找选课记录，对于没有选课的和选修了两门以上课的情况进行异常处理。

```
CREATE OR REPLACE PROCEDURE handle_exception_test(stu_num TEXT)
```

```
AS
DECLARE
    rec RECORD;
BEGIN
   SELECT *
   INTO rec
   FROM sc
   WHERE sno = stu_num;
EXCEPTION
    WHEN no_data_found THEN
        RAISE EXCEPTION '学号为 % 的学生记录未找到', stu_num;
    WHEN too_many_rows THEN
        RAISE EXCEPTION '学号为 % 的学生选了多门课程', stu_num;
END;
/
```

在该存储过程中，使用 WHEN 语句捕获 no_data_found 和 too_many_rows 两种标准异常，no_data_found 异常是在查询结果为空时产生，too_many_rows 异常是在查询结果多于一行时产生。

调用该存储过程：

```
dbsc=# CALL handle_exception_test('22001');
ERROR:  学号为 22001 的学生选了多门课程
dbsc=# CALL handle_exception_test('22003');
ERROR:  学号为 22003 的学生记录未找到
```

调用 insert_sc 存储过程插入一条选课数据：

```
dbsc=# CALL insert_sc ('22003', 1, 86);
 insert_sc
-----------


(1 row)
```

再调用该存储过程：

```
dbsc=# CALL handle_exception_test('22003');
 handle_exception_test
-----------------------


(1 row)
```

7. 游标与循环

【查询 28】创建一个存储过程，获取大于等于指定成绩的学生学号、姓名、课程名称和成绩记录，并将每条记录逐行输出。

```
CREATE OR REPLACE PROCEDURE get_student_by_score(score SMALLINT)
AS
DECLARE
    -- 定义游标
    cur REFCURSOR;
```

```
    -- 定义 RECORD 类型的变量来存储每行的返回值
    row RECORD;
BEGIN
    -- 打开游标
    OPEN cur FOR
        SELECT s.sno, s.sname, c.cname, sc.score
        FROM sc INNER JOIN student s ON sc.sno = s.sno
                INNER JOIN course c ON sc.cno = c.cno
        WHERE sc.score >= score;
    -- 获取并打印每行返回值
    LOOP
        FETCH NEXT FROM cur INTO row;
        EXIT WHEN NOT FOUND; -- 当找不到时退出
        -- 打印每行的返回值
        RAISE NOTICE '学号: %, 姓名: %, 课程: %, 成绩: %', row.sno, row.sname,
                                                        row.cname, row.score;
    END LOOP;
    -- 关闭游标
    CLOSE cur;
END;
/
```

在 DECLARE 语句中，首先定义游标而不打开它，因而使用 REFCURSOR 类型；定义 RECORD 类型的变量用于保存游标所指表的每行记录，获取数据后 RECORD 类型使用方式与%ROWTYPE 类型相似。

调用该存储过程，获取成绩大于等于 90 的学生选课记录：

```
dbsc=# CALL get_student_by_score(90);
NOTICE:  学号: 22001, 姓名: 张三, 课程: 高等数学, 成绩: 96
NOTICE:  学号: 22001, 姓名: 张三, 课程: 数据库系统, 成绩: 85
NOTICE:  学号: 22001, 姓名: 张三, 课程: 操作系统, 成绩: 88
NOTICE:  学号: 22002, 姓名: 李四, 课程: 数据库系统, 成绩: 90
NOTICE:  学号: 22002, 姓名: 李四, 课程: 操作系统, 成绩: 80
 get_student_by_score
----------------------


(1 row)
```

8. 删除存储过程

【查询 29】删除本节创建的所有存储过程。

```
DROP PROCEDURE insert_sc;
DROP PROCEDURE delete_sc;
DROP PROCEDURE get_sname;
DROP FUNCTION get_avg_score;
DROP PROCEDURE get_student;
DROP FUNCTION is_in_sc;
DROP PROCEDURE handle_exception_test;
```

```
DROP PROCEDURE get_student_by_score;
```

3.4.4 使用触发器

1. 创建触发器

   【查询 30】在 sc 表上创建一个触发器，该触发器的作用是对分数 score 列的更新操作进行审计记录，即对 UPDATE 语句所更新的每一行，均记录用户名、操作时间、学号、课号、score 列的旧值和新值。

   首先，创建 sc_audit 表，用于保存该触发器产生的审计记录：

```
CREATE TABLE sc_audit
(
    log_id      SERIAL,             -- IDENTITY 属性
    login_name  VARCHAR(256),       -- 登录名
    update_date TIMESTAMP,          -- 修改时间
    sno         CHAR(5),            -- 学号
    cno         SMALLINT,           -- 课程号
    score_old   SMALLINT,           -- 成绩的旧值
    score_new   SMALLINT,           -- 成绩的新值
    CONSTRAINT sc_audit_pk PRIMARY KEY(log_id)
);
```

   其中，主键 log_id 为 SERIAL 类型的自增编号列。

   创建触发器函数 sc_update_audit：

```
CREATE OR REPLACE FUNCTION sc_update_audit() RETURNS TRIGGER AS $$
BEGIN
    IF OLD.score <> NEW.score THEN
        INSERT INTO sc_audit(login_name, update_date, sno, cno, score_old, score_new)
        SELECT CURRENT_USER, CURRENT_TIMESTAMP, OLD.sno, OLD.cno, OLD.score, NEW.score;
    END IF;
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;
```

   执行该函数将向 sc_audit 表插入一条记录，用 CURRENT_USER 和 CURRENT_TIMESTAMP 获取当前用户名和系统时间；OLD 代表触发事件发生之前的行记录，NEW 代表触发事件发生之后的行记录；这里，不仅记录学号和课号，还记录了 score 列的旧值和新值。

   创建 sc 表上的触发器 tr_sc_update_audit，绑定触发器函数 sc_update_audit：

```
CREATE TRIGGER tr_sc_update_audit
AFTER UPDATE ON sc
FOR EACH ROW
EXECUTE PROCEDURE sc_update_audit();
```

   AFTER UPDATE 表明该触发器将在 UPDATE 事件发生之后被触发执行；FOR EACH ROW 表明对于 UPDATE 事件影响的每一行均执行一次触发器。

2. 使用触发器

执行 UPDATE 语句，给学号为 22001 的学生的每门课程成绩加 1 分：

```
UPDATE sc SET score = score + 1 WHERE sno = '22001';
```

查看 sc 表：

```
SELECT * FROM sc;
  sno   | cno | score
-------+-----+-------
 22002 |   2 |    90
 22002 |   3 |    80
 22001 |   1 |    97
 22001 |   2 |    86
 22001 |   3 |    89
(5 rows)
```

查看 sc_audit 表：

```
SELECT * FROM sc_audit;
log_id | login_name |        update_date         |  sno  | cno | score_old | score_new
--------+------------+----------------------------+-------+-----+-----------+-----------
      1 | omm        | 2024-03-13 22:05:37.528502 | 22001 |   1 |        96 |        97
      2 | omm        | 2024-03-13 22:05:37.543791 | 22001 |   2 |        85 |        86
      3 | omm        | 2024-03-13 22:05:37.544011 | 22001 |   3 |        88 |        89
(3 rows)
```

执行 UPDATE 语句，给学号为 22001 的学生的每门课程成绩减 1 分，即恢复为每门课程原来的成绩：

```
UPDATE sc SET score = score - 1 WHERE sno = '22001';
```

查看 sc 表：

```
SELECT * FROM sc;
  sno   | cno | score
-------+-----+-------
 22002 |   2 |    90
 22002 |   3 |    80
 22001 |   1 |    96
 22001 |   2 |    85
 22001 |   3 |    88
(5 rows)
```

再查看 sc_audit 表：

```
SELECT * FROM sc_audit;
log_id | login_name |        update_date         |  sno  | cno | score_old | score_new
--------+------------+----------------------------+-------+-----+-----------+-----------
      1 | omm        | 2024-03-13 22:05:37.528502 | 22001 |   1 |        96 |        97
      2 | omm        | 2024-03-13 22:05:37.543791 | 22001 |   2 |        85 |        86
      3 | omm        | 2024-03-13 22:05:37.544011 | 22001 |   3 |        88 |        89
      4 | omm        | 2024-03-13 22:06:45.907253 | 22001 |   1 |        97 |        96
      5 | omm        | 2024-03-13 22:06:45.907394 | 22001 |   2 |        86 |        85
      6 | omm        | 2024-03-13 22:06:45.907481 | 22001 |   3 |        89 |        88
(6 rows)
```

可见，sc 表上 UPDATE 事件所涉及的有 score 列数据更新的每一行均在 sc_audit 表中保存了相应的审计记录。

3. 修改触发器名称

【查询 31】将触发器 tr_sc_update_audit 的名称修改为 tr_score_audit。

```
ALTER TRIGGER tr_sc_update_audit ON sc RENAME TO tr_score_audit;
```

4. 删除触发器及相关数据库对象

【查询 32】将前面建立的触发器及相关的数据库对象删除。

删除触发器 tr_score_audit：

```
DROP TRIGGER tr_score_audit ON sc;
```

删除触发器函数 sc_update_audit：

```
DROP FUNCTION sc_update_audit;
```

删除 sc_audit 表：

```
DROP TABLE sc_audit;
```

3.4.5 使用事务机制

1. 验证事务原子性

打开一个新的 gsql 客户端，连接数据库 dbsc：

```
[omm@eduog ~]$ gsql -d postgres -p 26000 -r
gsql ((openGauss 5.0.1 build 33b035fd) compiled at 2023-12-15 20:28:19 commit 0 last mr  )
Non-SSL connection (SSL connection is recommended when requiring high-security)
Type "help" for help.

openGauss=# \c dbsc
Non-SSL connection (SSL connection is recommended when requiring high-security)
You are now connected to database "dbsc" as user "omm".
dbsc=#
```

开启事务：

```
dbsc=# BEGIN TRANSACTION;
BEGIN
```

向 sc 表中插入一条数据：

```
dbsc=# INSERT INTO sc VALUES ('22003', 1, 86);
INSERT 0 1
```

查看 sc 表：

```
dbsc=# SELECT * FROM sc;
  sno   | cno | score
-------+-----+-------
 22001 |   1 |    96
 22001 |   2 |    85
 22001 |   3 |    88
 22002 |   2 |    90
 22002 |   3 |    80
 22003 |   1 |    86
(6 rows)
```

退出该 gsql 客户端：

```
dbsc=# \q
[omm@eduog ~]$
```

再次打开 gsql 客户端并连接 dbsc 数据库，查看 sc 表：

```
[omm@eduog ~]$ gsql -d postgres -p 26000 -r
gsql ((openGauss 5.0.1 build 33b035fd) compiled at 2023-12-15 20:28:19 commit 0 last mr  )
Non-SSL connection (SSL connection is recommended when requiring high-security)
Type "help" for help.

openGauss=# \c dbsc
Non-SSL connection (SSL connection is recommended when requiring high-security)
You are now connected to database "dbsc" as user "omm".
dbsc=# SELECT * FROM sc;
  sno   | cno | score
-------+-----+-------
 22001 |   1 |    96
 22001 |   2 |    85
 22001 |   3 |    88
 22002 |   2 |    90
 22002 |   3 |    80
(5 rows)
```

显示并没有前面插入 sc 表的那行记录。这验证了事务的原子性，表明使用 BEGIN TRANSACTION 开启的事务，还没有提交，客户端就退出，事务自动回滚，所有数据修改操作被撤销，数据库回到事务执行前的状态。也可以在事务中使用 ROLLBACK 语句进行事务回滚。

2. 验证事务隔离级别

   openGauss 数据库默认的事务隔离级别是 READ COMMITTED（读已提交），即一个事务中能够读到其他事务已提交的数据，而不能读到未提交的数据。

   在当前 gsql 客户端开启一个事务，记作事务 T1：

```
dbsc=# BEGIN TRANSACTION;
BEGIN
```

   查询 sc 表：

```
dbsc=# SELECT * FROM sc;
  sno   | cno | score
-------+-----+-------
 22001 |   1 |    96
 22001 |   2 |    85
 22001 |   3 |    88
 22002 |   2 |    90
 22002 |   3 |    80
(5 rows)
```

   再打开另一个 gsql 客户端，连接 dbsc 数据库，开启一个事务，记作事务 T2：

```
[omm@eduog ~]$ gsql -d postgres -p 26000 -r
gsql ((openGauss 5.0.1 build 33b035fd) compiled at 2023-12-15 20:28:19 commit 0 last mr  )
```

```
Non-SSL connection (SSL connection is recommended when requiring high-security)
Type "help" for help.

openGauss=# \c dbsc
Non-SSL connection (SSL connection is recommended when requiring high-security)
You are now connected to database "dbsc" as user "omm".
dbsc=# BEGIN TRANSACTION;
BEGIN
```

在事务 T2 中，将 22002 学生的 3 号课程成绩改为 90，然后查看 sc 表：

```
dbsc=# UPDATE sc SET score = 90 WHERE sno = '22002' AND cno = 3;
UPDATE 1
dbsc=# SELECT * FROM sc;
  sno  | cno | score
-------+-----+-------
 22001 |   1 |    96
 22001 |   2 |    85
 22001 |   3 |    88
 22002 |   2 |    90
 22002 |   3 |    90
(5 rows)
```

此时，在事务 T1 中，查看 sc 表：

```
dbsc=# SELECT * FROM sc;
  sno  | cno | score
-------+-----+-------
 22001 |   1 |    96
 22001 |   2 |    85
 22001 |   3 |    88
 22002 |   2 |    90
 22002 |   3 |    80
(5 rows)
```

看到 22002 学生的 3 号课程成绩仍为 80，没有改为 90。因为，事务 T1 中读不到事务 T2 中尚未提交的数据更新操作。

这时，在事务 T2 中，使用 COMMIT 语句提交事务：

```
dbsc=# COMMIT TRANSACTION;
COMMIT
```

再到事务 T1 中，查看 sc 表：

```
dbsc=# SELECT * FROM sc;
  sno  | cno | score
-------+-----+-------
 22001 |   1 |    96
 22001 |   2 |    85
 22001 |   3 |    88
 22002 |   2 |    90
 22002 |   3 |    90
```

```
(5 rows)
```

发现 22002 学生的 3 号课程的成绩已改为 90。这表明，在事务 T1 中，可以读到事务 T2 中已经提交的数据更新。从而验证了 READ COMMITTED 隔离级别。

此时，事务 T2 已提交结束了。

在事务 T1 中，再将 22002 学生的 3 号课程的成绩改回 80：

```
dbsc=# UPDATE sc SET score = 80 WHERE sno = '22002' AND cno = 3;
UPDATE 1
```

接下来，将验证 REPEATABLE READ（可重复读）事务隔离级别，REPEATABLE READ 是比 REPEATABLE READ 更加严格的隔离级别。

将事务 T1 的隔离级别设置为 REPEATABLE READ：

```
dbsc=# SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
SET
```

在另一 gsql 客户端中开启事务 T3：

```
dbsc=# BEGIN TRANSACTION;
BEGIN
```

在事务 T3 中，向 sc 表中插入一条数据，然后提交事务：

```
dbsc=# INSERT INTO sc VALUES ('22003', 1, 86);
INSERT 0 1
dbsc=# COMMIT;
COMMIT
```

在事务 T1 中，查看 sc 表：

```
dbsc=# SELECT * FROM sc;
  sno  | cno | score
-------+-----+-------
 22001 |   1 |    96
 22001 |   2 |    85
 22001 |   3 |    88
 22002 |   2 |    90
 22002 |   3 |    80
(5 rows)
```

发现 sc 表中数据没有变化。在事务 T1 中，由于隔离级别改为了 REPEATABLE READ，即使事务 T3 中已经提交的更新操作，事务 T1 也无法读取到。也就是说，在隔离级别为 REPEATABLE READ 的事务中，所有的读操作都是读取事务开始前的数据库状态。当然一个事务中的更新操作对于该事务内后续的读取操作是有效可见的，否则就不符合事务内正常的业务逻辑操作了。

3.4.6 使用权限

1. 创建角色

   【查询 33】 创建角色 myrole 并设置登录密码。

   在 omm 用户的客户端，执行以下语句：

```
openGauss=# CREATE ROLE myrole WITH PASSWORD 'DBlab@123';
CREATE ROLE
```

测试创建的角色 myrole 能否登录数据库：

在另一个 SSH 客户端中，gsql 用 myrole 角色连接 dbsc 数据库，执行下面命令：

```
[omm@eduog ~]$ gsql -d dbsc -p 26000 -U myrole -W DBlab@123 -r
gsql: FATAL:  role "myrole" is not permitted to login
```

报告不允许 myrole 角色登录数据库。

【查询 34】给角色 myrole 增加登录权限 LOGIN。

需要修改角色信息，使用 ALTER ROLE 语句。

```
ALTER ROLE myrole WITH LOGIN;
```

再次连接数据库 dbsc：

```
[omm@eduog ~]$ gsql -d dbsc -p 26000 -U myrole -W DBlab@123 -r
gsql ((openGauss 5.0.1 build 33b035fd) compiled at 2023-12-15 20:28:19 commit 0 last mr  )
Non-SSL connection (SSL connection is recommended when requiring high-security)
Type "help" for help.


dbsc=>
```

可以登录进 dbsc 数据库了。

查询 student 表，发现没有查询权限。

```
dbsc=> SELECT * FROM student;
ERROR:  permission denied for relation student
DETAIL:  N/A
```

2. 授予权限

【查询 35】授予角色 myrole 查询 student 表的权限。

在 omm 用户的客户端，执行以下 GRANT 语句：

```
dbsc=# GRANT SELECT ON student TO myrole;
GRANT
```

在另一客户端，myrole 角色执行查询 student 表操作：

```
dbsc=> SELECT * FROM student;
  sno   | sname | sgender |       sbirth        | sdept
--------+-------+---------+---------------------+-------
 22001 | 张三  | 男      | 2002-05-19 00:00:00 | CS
 22002 | 李四  | 女      | 2002-01-09 00:00:00 | CS
 22003 | 王五  | 女      | 2001-12-08 00:00:00 | CE
 22004 | 赵六  | 男      | 2001-08-30 00:00:00 | IS
(4 rows)
```

可见，myrole 角色这时可以查询 student 表了。

【查询 36】授予角色 myrole 查询 student 表的权限并允许将该权限授予他人。

在 omm 用户的客户端，再创建一个新角色 newrole：

```
dbsc=# CREATE ROLE newrole WITH LOGIN PASSWORD 'DBlab@123';
CREATE ROLE
```

在 myrole 用户的客户端，向 newrole 用户授权 student 表的 SELECT 权限：

```
dbsc=> GRANT SELECT ON student TO newrole;
ERROR:  no privileges were granted for "student"
```

出现报错信息，显示授权失败。

在 omm 用户的客户端，执行

```
dbsc=# GRANT SELECT ON student TO myrole WITH GRANT OPTION;
GRANT
```

在 myrole 用户的客户端，再次向 newrole 用户授权 student 表的 SELECT 权限：

```
dbsc=> GRANT SELECT ON student TO newrole;
GRANT
```

此时显示授权成功。因为 omm 用户向 myrole 用户授权时使用了 WITH GRANT OPTION 选项，给予了 myrole 用户再次将该权限授予其他用户的权力。

再打开一个新的客户端，使用 newrole 用户登录数据库，经验证能够查询 student 表，证明 newrole 用户拥有对于 student 表的 SELECT 权限，但由于 myrole 用户向 newrole 用户授权时没有使用 WITH GRANT OPTION 选项，因而 newrole 用户无法再将该权限授予其他用户。

3. 回收权限

【查询 37】回收角色 myrole 查询 student 表的权限。

在 omm 用户的客户端，执行以下语句：

```
dbsc=# REVOKE SELECT ON student FROM myrole;
ERROR:  dependent privileges exist
HINT:  Use CASCADE to revoke them too.
```

因为有 myrole 用户授权给 newrole 用户的权限依赖于此权限，所以数据库拒绝执行回收该权限。提示使用 CASCADE 选项，执行以下语句：

```
dbsc=# REVOKE SELECT ON student FROM myrole CASCADE;
REVOKE
```

表明回收权限执行成功。分别在 myrole 用户和 newrole 的客户端执行 SELECT * FROM student 语句进行验证，发现不仅 myrole 用户没有了该权限，newrole 用户的权限也被级联回收了。

4. 删除角色

【查询 38】删除角色 myrole 和 newrole。

在 omm 用户客户端，执行 DROP ROLE 语句，删除两个角色：

```
dbsc=# DROP ROLE myrole;
DROP ROLE
dbsc=# DROP ROLE newrole;
DROP ROLE
```

3.5 实验结果

【请按照要求完成实验操作】

1. 完成实验步骤 3.4.1 节。

2. 完成实验步骤 3.4.2 节。

3. 完成实验步骤 3.4.3 节。

4. 完成实验步骤 3.4.4 节。

5. 完成实验步骤 3.4.5 节。

6. 完成实验步骤 3.4.6 节。

3.6 实验讨论与总结

【请将实验中遇到的问题描述、解决办法与思考讨论列在下面，并对本实验进行总结。】