第 7 章 表的创建与系统表

7.1 实验介绍

本实验通过探索 openGauss 数据库服务器中 CREATE TABLE 语句执行所对应的相关源代码，引出 openGauss 的系统表机制。在 openGauss 数据库系统中，每创建一张表都会自动在系统表中存储相关信息。本实验以 pg_class、pg_attribute 和 pg_type 这三个重要的系统表为牵引，通过源代码解析深入了解关系表的定义以及属性列的详细信息。通过查询系统表，理解与表定义相关的元数据组织结构；通过浏览重要结构体与系统表源代码，了解代码层面的细节内容。

7.2 实验目的

1. 掌握 CREATE TABLE 语句执行与三个系统表中元数据的关联性。
2. 掌握三个系统表的关键属性、表间关联与查询操作。
3. 了解与本实验相关的重要结构体与系统表源代码。
4. 理解通过添加代码输出 CREATE TABLE 相关系统的原理。

7.3 实验原理

7.3.1 系统表

系统表（system catalog）是数据库管理系统中内置的特殊类型的数据表，用于存储数据库元数据（即关于数据库本身的数据），包括数据库中的表、索引、视图、存储过程等（这些统称为数据库对象）的描述信息。在 openGauss 中，系统表的主要作用是存储数据库中对象的元数据，并且与某些系统函数配合使用以实现对数据库的管理和维护。

每个系统表都有一个定义的格式，其中包括一些字段，每个字段代表一项元数据信息。例如，pg_class 系统表包含一个 relname 字段，用于存储关系表的名称；pg_attribute 系统表包含一个 attname 字段，用于存储属性列的名称。通过查询系统表，可以获取数据库中对象的详细信息，进而实现对数据库的管理。

系统表是由系统进程维护的，并且在数据库启动时自动加载。系统表是隐藏的，因此不能直接查询它们，但是可以通过调用系统函数来访问系统表中的信息。 openGauss 系统表的实现继承了 PostgreSQL 的部分约定。

7.3.2 系统表的关联

openGauss 系统表之间存在着复杂的关联关系，所有系统表协作支持着数据库系统的运作。例如，pg_class 系统表保存了数据库中的每个表的元数据，包括表名、表的 OID（对象

标识符）等；pg_attribute 系统表保存了表中的每个列的元数据，包括列名、数据类型等。这些系统表之间存在着对应关系，例如表的 OID 在 pg_class 中，而该表的列信息在 pg_attribute 中。pg_class 和 pg_attribute 两个系统表之间存在关联关系（用主外键表示）。

除此之外，openGauss 还有一些其他的系统表，如 pg_type、pg_constraint 等，它们与 pg_class、pg_attribute 等系统表同样也存在着复杂的关联关系。通过研究这些关联关系，可以更好地理解 openGauss 数据库的内部机制。

同时，系统表与数据库内其他数据库对象（如用户定义的表）相互关联，并在数据库运行过程中被用于维护数据库的一致性。例如，当创建一个新表时，系统表中的相关记录将更新，以描述这个新表。反之，当删除一个表时，系统表中的相关记录将被删除。因此，系统表与数据库内其他数据库对象的关联性对数据库的正常运行至关重要。

7.3.3 相关结构体

在 openGauss 中，与 CREATE TABLE 和系统表相关的主要结构体有以下几种：

- FormData_pg_class：定义了表级元数据的数据结构，存储了表名、表的 OID、存储类型等信息。对应于系统表 pg_class 的记录。
- FormData_pg_attribute：定义了列级元数据的数据结构，存储了列名、数据类型、列的 OID 等信息。对应于系统表 pg_attribute 的记录。
- FormData_pg_type：用于描述一个数据类型的信息，例如类型名称、大小、对齐方式等。对应于系统表 pg_type 的记录。
- TupleDesc：用于描述关系的表的元数据，存储了关于关系的所有列的信息。
- RelationData：存储了关于关系的所有元数据，包括表的 TupleDesc，表的 FormData_pg_class 结构体，关于系统表的关系 OID 等信息。
- HeapTupleData：表示数据库中的元组的结构体，是存储数据的主要结构。

7.4 实验步骤

7.4.1 使用 CREATE TABLE 创建关系模式

在 SQL 中，使用 CREATE TABLE 语句创建关系模式（数据表）。语法如下：

```
CREATE TABLE table_name (
   column1 datatype constraint,
   column2 datatype constraint,
   ...
   column_n datatype constraint
);
```

其中，

- table_name 是数据表的名称，
- column1，column2，...，column_n 是数据表中的列名，
- datatype 是列的数据类型（例如 INT，VARCHAR，DATE 等），
- constraint 是列的约束（例如 NOT NULL，UNIQUE 等）。

CREATE TABLE 语句实际上创建的是一个关系模式，对于关系表中的实例数据来讲，

CREATE TABLE 中提供的模式信息属于元数据。CREATE TABLE 语句创建的元数据主要有：

- 定义数据表的名称：使用 table_name 来定义数据表的名称，例如：CREATE TABLE users。
- 定义数据表的列：使用 column1，column2，…，column_n 定义数据表的列。每个列都有一个数据类型（例如 INT, VARCHAR, DATE 等）和一个约束（例如 NOT NULL, UNIQUE 等）。例如：CREATE TABLE users (id INT PRIMARY KEY, name VARCHAR(255) NOT NULL)。
- 创建数据表：执行 CREATE TABLE 语句后，将在数据库中创建一个名为 users 的数据表，具有两个列：id 和 name。

下面 CREATE TABLE 语句创建了一个含有 5 个属性的 users 表，用于存储某系统中的用户信息。

```
CREATE TABLE users
(
    u_id varchar(20),
    u_passwd varchar(20),
    u_name varchar(10),
    u_idnum varchar(20),
    u_regtime timestamp
);
```

该 users 表的属性说明如下：

- u_id：用户 id，用于系统登录账户名
- u_passwd：密码，用于系统登录密码
- u_name：真实姓名
- u_idnum：证件号码
- u_regtime：注册时间

1. 启动 openGauss 数据库服务器。

```
[dblab@eduog openGauss-server-v3.0.0]$ gs_ctl start -D $GAUSSHOME/data -Z single_node
-l logfile
```

2. 执行 gsql 客户端连接数据库。

```
[dblab@eduog openGauss-server-v3.0.0]$ gsql postgres -r
```

3. 执行以上 CREATE TABLE 语句，创建 users 表。

```
openGauss=# CREATE TABLE users
openGauss-# (
openGauss(# u_id varchar(20),
openGauss(# u_passwd varchar(20),
openGauss(# u_name varchar(10),
openGauss(# u_idnum varchar(20),
openGauss(# u_regtime timestamp
openGauss(# );
CREATE TABLE
```

4. 执行 DROP TABLE 语句，删除 users 表。

```
openGauss=# DROP TABLE users;
DROP TABLE
```

5. openGauss 支持行列混合存储，行存储是指将表按行存储到硬盘分区上，列存储是

指将表按列存储到硬盘分区上。默认情况下使用 CREATE TABLE 语句创建的表为行存表，若要创建列存表，可用 WITH (ORIENTATION = COLUMN)进行指定（当 ORIENTATION=ROW 或不指定 ORIENTATION 时则为创建行存表）。下面给出创建 users 列存表的示例：

```
CREATE TABLE users
(
    u_id varchar(20),
    u_passwd varchar(20),
    u_name varchar(10),
    u_idnum varchar(20),
    u_regtime timestamp
)
WITH (ORIENTATION = COLUMN);
```

7.4.2 查询系统表

在关系数据库管理系统中，各种数据库对象（如表、视图、约束、存储过程等）的元数据通常也是以关系表结构进行存储与维护，这类表称为系统表（system catalog）。通过对系统表的查询，可以了解当前数据库内部维护的各类数据库对象的情况。在通常情况下，用户不会直接在系统表上进行更改操作，这类操作由某些 SQL 语句在 DBMS 服务器内部执行。

openGauss 的系统表一部分继承自 PostgreSQL。在 openGauss 中执行 CREATE TABLE 语句，会涉及到向若干系统表插入数据。这里介绍 pg_class、pg_attribute 和 pg_type 这 3 个系统表。

1. 系统表 pg_class

   pg_class 是 openGauss 数据库系统中的一个重要的系统表，存储了所有数据库对象的元数据信息，包括表，视图，索引，序列和其他对象。它包含以下重要字段：

   - oid：数据库对象的唯一标识符。内部使用。
   - relname：对象的名称。
   - relkind：对象的类型，可以是表，视图，索引，序列等。(值的解释：普通表 r、索引 i、序列 S、TOAST 表 t、视图 v、物化视图 m、组合类型 c、外部表 f、划分表 p、划分索引 I)。
   - relnamespace：对象所属的模式的 ID。
   - reltablespace：对象所在的表空间的 ID。
   - relpages：对象占用的页数。
   - reltuples：对象中的元组数。(表的行数)。
   - relnatts：表的列数。在系统表 pg_attribute 中要有这些列。

   在 gsql 中，执行下面 SQL 语句，显示 pg_class 系统表的部分列的数据，如下所示，可以看到，刚刚建立的 users 表对应的元数据位于 pg_class 系统表的第 3 行。

```
openGauss=# SELECT oid, relname, reltuples, relkind, relnatts FROM pg_class;
 oid  |             relname            |     reltuples  |   r        |         30
------+--------------------------------+----------------+------------+----------------
 1247 |  pg_type                       |          719   |   r        |          4
 9730 |  gs_clientlobal_keys_args      |            0   |   r        |          6
 24628|  users                         |            0   |   i        |          1
```

```
6124 |  pg_subscription_oid_index      |           0   |   i        |          2
6125 |  pg_subscription_subname_index  |           0   |   v        |          1
11925|  pgxc_prepared_xacts            |           0   |   v        |         18
11933|  pg_shadow                      |           0    |   v        |         27
11929|  pg_roles                       |           0    |   v        |         19
11941|  pg_user                        |           0    |   v        |          3
11937|  pg_group                       |           0    |   v        |          4
11945|  pg_rules                       |           0    |   v        |         26
1260 |  pg_authid                      |           0    |   r        |          6
11949|  gs_labels                      |           0    |   v        |          7
11969|  pg_rispolicies                 |           0    |   v        |          7
11953|  gs_auditing_access             |           0    |   v        |          4
11973|  pg_views                       |           0    |   v        |          7
11957|  gs_auditing_privilege          |           0    |   v        |          7
11977|  pg_tables                      |           0    |   v        |         10
11961|  gs_auditig                     |           0    |   v        |          7
11981|  gs_matviews                    |           0    |   v        |          6
11965|  gs_masking                     |           0    |   v        |          6
11985|  pg_indexes                     |           0    |   v        |          5
11989|  pg_gtt_relastats               |           0    |   v        |          8
11993|  pggtt_attached_pids            |           0    |   v        |          4
```

2. 系统表 pg_attribute

   pg_attribute 是 openGauss 数据库系统中的另一个重要的系统表，存储了数据库表中列的元数据信息。数据库中每个表的每一列在 pg_attribute 表中都有对应的一行。它包含以下重要字段：

   - attrelid：列所属的表的 oid，引用 pg_class.oid。
   - attname：列的名称。
   - atttypid：列的数据类型的 oid，引用 pg_type.oid。
   - attnum：列的编号，用户定义的列从 1 开始编号，系统列为某个负数。
   - attlen：列数据类型的长度。
   - attnotnull：是否需要非空约束。
   - atttypemod：数据类型的特殊修饰符。

   在 gsql 中，执行下面的 SQL 语句，显示 pg_attribute 系统表的部分列的数据，如下所示：

```
openGauss=# SELECT attrelid, attname, atttypid FROM pg_attribute;
 attrelid |               attname               | atttypid
----------+-------------------------------------+------------------
 7815     | pkgnamespace                        |       26
 7815     | pkgowner                            |       26
 7815     | pkgname                             |       19
 7815     | pkgspecsrc                          |       25
 7815     | pkgbodydeclsrc                      |       25
 7815     | pkgbodyinitsrc                      |       25
 7815     | pkgacl                              |     1034
```

```
7815     | pkgsecdef                      |       16
7815     | ctid                           |       27
7815     | oid                            |       26
7815     | xmin                           |       28
7815     | cmin                           |       29
7815     | xmax                           |       28
7815     | cmax                           |       29
7815     | tableoid                       |       26
7815     | xc_node_id                     |       23
1255     | proname                        |       19
1255     | pronamespace                   |       26
1255     | proowner                       |       26
1255     | prolang                        |       26
1255     | procost                        |      700
1255     | prorows                        |      700
1255     | provariadic                    |       26
1255     | protransform                   |       24
--More--
```

3. 查看 users 表的各列信息

   执行下面 SQL 语句，通过 pg_attribute 表和 pg_class 表连接操作，查看 users 表的各个属性列的几个元数据信息（在 pg_attribute 表中的部分列）。如下所示:

```
openGauss=# SELECT attrelid, attname, atttypid, attnum FROM pg_attribute, pg_class WHERE
pg_attribute.attrelid = pg_class.oid AND pg_class.relname='users';
 attrelid  |   attname    |  atttypid  |  attnum
-----------+--------------+------------+-----------------
     24628 |  xc_node_id  |         23 |      -8
     24628 |  tableoid    |         26 |      -7
     24628 |  cmax        |         29 |      -6
     24628 |  xmax        |         28 |      -5
     24628 |  cmin        |         29 |      -4
     24628 |  xmin        |         28 |      -3
     24628 |  ctid        |         27 |      -1
     24628 |  u_id        |       1043 |       1
     24628 |  u_passwd    |       1043 |       2
     24628 |  u_name      |       1043 |       3
     24628 |  u_idnum     |       1043 |       4
     24628 |  u_regtime   |       1114 |       5
(12 rows)
```

   可以看到该查询所列出的列，除了 users 表中用户定义的列之外，还有 7 个系统列（system column），这些列中 tableoid 是该行所属的表的 oid，xc_node_id 与集群存储有关，其他列均与事务管理有关。系统列的具体介绍可参见 PostgreSQL 文档: https://www.postgresql.org/docs/current/ddl-system-columns.html

4. 系统表 pg_type

   pg_type 是 openGauss 数据库系统中的一个重要的系统表，存储了数据库中所有数

据类型的元数据信息。在 CREATE TABLE 语句时，系统会自动在 pg_type 创建新建表对应的组合类型，表示新建表的一行的结构类型。pg_type 包含以下重要字段：

- oid：数据类型的唯一标识符，类型的对象 id。
- typname：数据类型的名称。
- typnamespace：数据类型所在的模式的 ID。
- typtype：数据类型的种类，可以是基本数据类型，自定义数据类型等（值的解释：基础类型 b、组合类型（表的行类型）c、域类型 d、枚举类型 e。
- typlen：数据类型的长度。定长类型为该类型数值所占字节数，变长类型为-1。
- typrelid：组合类型对应的表的 oid，引用 pg_class.oid。
- typbyval：数据类型是否按值传递。

在 gsql 中，执行下面 SQL 语句，显示 pg_type 系统表的部分列的数据，如下所示：

```
openGauss=# SELECT oid, typname, typlen, typtype, typrelid FROM pg_type;
  oid |                              typname |  typlen | typtype | typrelid
------+--------------------------------------+---------+---------+----------
   16 | bool                                 |       1 | b       |        0
   17 | bytea                                |      -1 | b       |        0
   18 | char                                 |       1 | b       |        0
   19 | name                                 |      64 | b       |        0
   20 | int8                                 |       8 | b       |        0
   21 | int2                                 |       2 | b       |        0
 5545 | int1                                 |       1 | b       |        0
   22 | int2vector                           |      -1 | b       |        0
   23 | int4                                 |       4 | b       |        0
   24 | regproc                              |       4 | b       |        0
   25 | text                                 |      -1 | b       |        0
   26 | oid                                  |       4 | b       |        0
   27 | tid                                  |       6 | b       |        0
   28 | xid                                  |       8 | b       |        0
   31 | xid32                                |       4 | b       |        0
   29 | cid                                  |       4 | b       |        0
   30 | oidvector                            |      -1 | b       |
   32 | oidvector_extend                     |      -1 | b       |        0
   33 | int2vector_extend                    |      -1 | b       |        0
   34 | int16                                |      16 | b       |        0
   86 | raw                                  |      -1 | b       |        0
   87 | _raw                                 |      -1 | b       |        0
   88 | blob                                 |      -1 | b       |        0
 3201 | _blob                                |      -1 | b       |        0
```

5. 查看 users 表的各列的类型信息

执行下面 SQL 语句，通过 pg_attribute、pg_class 和 pg_type 表连接操作，查看 users 表的各个属性列的元数据信息及其对应的类型元数据信息（在 pg_attribute 和 pg_type 表中的部分列），如下所示。

```
openGauss=# SELECT attrelid, attname, atttypid, typname, typlen, typtype, typrelid FROM
pg_attribute, pg_class, pg_type WHERE pg_attribute.attrelid = pg_class.oid AND
```

```
pg_attribute.atttypid = pg_type.oid AND pg_class.relname='users';
 attrelid |    attname   | atttypid |   typname   | typlen | typtype | typrelid
----------+--------------+----------+-------------+--------+---------+--------------
    24628 |  xc_node_id  |       23 |   int4      |      4 |       b |      0
    24628 |  tableoid    |       26 |   oid       |      4 |       b |      0
    24628 |  cmax        |       29 |   cid       |      4 |       b |      0
    24628 |  xmax        |       28 |   xid       |      8 |       b |      0
    24628 |  cmin        |       29 |   cid       |      4 |       b |      0
    24628 |  xmin        |       28 |   xid       |      8 |       b |      0
    24628 |  ctid        |       27 |   tid       |      6 |       b |      0
    24628 |  u_id        |     1043 |  varchar    |     -1 |       b |      0
    24628 |  u_passwd    |     1043 |  varchar    |     -1 |       b |      0
    24628 |  u_name      |     1043 |  varchar    |     -1 |       b |      0
    24628 |  u_idnum     |     1043 |  varchar    |     -1 |       b |      0
    24628 |  u_regtime   |     1114 |  timestamp  |      8 |       b |      0
(13 rows)
```

6. 查看 users 表对应的组合类型（行类型）的类型信息

   执行下面 SQL 语句，通过 pg_class 和 pg_type 表连接操作，查看 users 表对应的组合类型（行类型）的类型信息元数据（在 pg_type 表中的部分列）。如下所示：

```
openGauss=# SELECT typname, typlen, typtype, typrelid FROM pg_class, pg_type WHERE
pg_class.oid = pg_type.typrelid AND pg_class.relname='users';
 typname | typlen | typtype | typrelid
---------+--------+---------+-------------
 users   |     -1 | c       |    24628
(1 row)
```

7. 查看 pg_class 和 pg_type 的关系

   pg_class 表中的 reltype 是表在 pg_type 表中对应行的 oid，pg_type 表中的 typrelid 是行类型在 pg_class 表中对应行的 oid，这两个属性都表示 pg_class 和 pg_type 之间的一对一关系。可以通过以下两个查询进行验证。从 pg_class 表中查询 users 表的 oid 和 reltype，如下所示：

```
openGauss=# SELECT oid, reltype FROM pg_class WHERE relname='users';
    oid | reltype
--------+---------
 24628  |   24630
(1 row)
```

   从 pg_type 表中查询 users 表的 oid 和 typrelid，如**错误!未找到引用源。**所示：

```
openGauss=# select oid, typrelid from pg_type where typname='users';
   oid | typrelid
-------+-----------
 24630 |  24628
(1 row)
```

7.4.3 浏览重要结构体与系统表源代码

在 openGauss 实验过程中，浏览和阅读 openGauss 源代码的益处包括：

- 加深对系统的理解：通过浏览源代码，可以更好地了解 openGauss 的内部系统结构和工作原理。
- 帮助调试：如果在进行实验时遇到了问题，浏览 openGauss 源代码可以帮助定位和解决问题。
- 提高代码能力：通过阅读以 openGauss 为代表的大型开源系统的代码，可以学习其设计和编码技巧，从而提高自己的代码编写能力。
- 改进代码：如果认为 openGauss 存在某些问题或可以改进，那么浏览和阅读源代码是参与开源社区开发一个很好的机会。
- 修改代码：如果需要对 OpenGauss 进行定制化，那么阅读源代码是必不可少的步骤。

总的来说，浏览阅读 openGauss 源代码是一种很好的学习方式，可以帮助更好地理解系统，提高代码能力，并对系统进行改进。

首先来看本实验涉及到的几个重要的结构体。

1. 结构体 RelationData 及其指针 Relation

   RelationData 结构体是一个重要结构体，用于存储和管理关系的信息。它在数据库内部表示一个关系，例如表、索引、视图等。RelationData 结构体的字段存储关系的元数据信息，如关系名称、关系类型、列信息、索引信息、约束信息等。

   RelationData 结构体的重要字段包括：

   - rd_node: 存储关系的内部标识符（内部编号），用于识别关系。
   - rd_rel: 存储关系的元数据信息，如关系名称、关系类型、列数、约束数等。
   - rd_att: 存储关系的属性信息，如列名、列类型、列长度、默认值等。
   - rd_index: 存储关系的索引信息，用于加速数据查询。
   - rd_rules: 存储关系的规则信息，用于实现触发器、视图等功能。
   - rd_options: 存储关系的选项信息，如关系的存储方式、扩展空间等。

   这些字段的信息可以通过相关的系统函数和系统视图来获取和操作。例如，可以通过 pg_class 系统表来查询关系的信息，通过 pg_attribute 系统表来查询关系的属性信息。

   【源码】src/include/utils/rel.h：

```
/*
 * Here are the contents of a relation cache entry.
 */


typedef struct RelationData {
    RelFileNode rd_node; /* relation physical identifier */
    // 省略若干行
    Form_pg_class rd_rel; /* RELATION tuple */
    TupleDesc rd_att;     /* tuple descriptor */
    Oid rd_id;            /* relation's object id */
    // 省略若干行
} RelationData;
```

Relation 结构被定义为 RelationData 结构体的指针。

【源码】src/include/utils/relcache.h：

```
typedef struct RelationData* Relation;
```

2. 结构体 FormData_pg_class 和及其指针 Form_pg_class

FormData_pg_class 是一个结构体，用于存储和管理关系信息。该结构体是在 pg_class 系统表中定义的，该系统表存储了数据库内的所有关系的元数据信息，如表、视图、索引等。FormData_pg_class 结构体中的字段用于存储关系的元数据信息，如关系的 OID，关系名称，关系类型，关系的状态等。这些信息对于数据库的正常运行和管理起着重要作用。

【源码】src/include/catalog/pg_class.h：

```
CATALOG(pg_class,1259) BKI_BOOTSTRAP BKI_ROWTYPE_OID(83) BKI_SCHEMA_MACRO
{
    NameData relname; /* class name */
    Oid relnamespace; /* OID of namespace containing this class */
    Oid reltype;      /* OID of entry in pg_type for table's
                       * implicit row type */
    // 省略若干行
}
FormData_pg_class;
```

Form_pg_class 被定义为指向 FormData_pg_class 结构体的指针。

```
/* ----------------
 *      Form_pg_class corresponds to a pointer to a tuple with
 *      the format of pg_class relation.
 * ----------------
 */
typedef FormData_pg_class* Form_pg_class;
```

3. 结构体 tupleDesc 及其指针 TupleDesc

tupleDesc 用于存储和管理元组的描述信息。元组是数据库中的一行数据，而 TupleDesc 存储了这一行数据的列信息，包括列名、列类型、列宽度、列是否为空等。这些信息对于数据库系统的正确识别和处理数据是非常重要的。通过使用 tupleDesc 结构体，openGauss 能够更加方便地管理和处理元组信息，提高数据库系统的效率。

tupleDesc 结构体的主要字段包括：

- natts：该元组中列的数量。
- attrs：一个指针数组，用于存储列的信息。
- tdtypeid：元组的类型标识符。
- tdtypmod：元组类型的模式。
- tdhasoid：一个标识符，用于标识元组是否有 OID 字段。

除了上述字段外，tupleDesc 结构体中还有其他一些字段，这些字段在不同的应用场景中都有其重要作用。TupleDesc 被定义为 tupleDesc 结构体的指针。

【源码】src/include/access/tupdesc.h：

```
/*
 * This struct is passed around within the backend to describe the
 * structure of tuples.  For tuples coming from on-disk relations, the
```

```
 * information is collected from the pg_attribute, pg_attrdef, and
 * pg_constraint catalogs.Transient row types (such as the result of a
 * join query) have anonymous TupleDesc structs that generally omit any
 * constraint info; therefore the structure is designed to let the
 * constraints be omitted efficiently.
 * Note that only user attributes, not system attributes, are mentioned
 * in TupleDesc; with the exception that tdhasoid indicates if OID is
 * present.
 // 省略若干行
 */
typedef struct tupleDesc {
    // 省略若干行
    int natts; /* number of attributes in the tuple */
   bool tdisredistable; /* temp table created for data redistribution by the redis tool
*/
    // 省略若干行
} * TupleDesc;
```

4. 结构体 NameData 及其指针 Name

   NameData 用于存储数据库对象的名称。具体来说，NameData 是一个固定长度的数组，用于存储数据库对象的名称，比如表名、列名、索引名等。Name 指针是一个指向 NameData 结构体的指针，通常使用 Name 指针来引用数据库对象的名称，而不是使用 NameData 结构体。

   【源码】src/include/c.h：

```
 * Representation of a Name: effectively just a C string, but null-padded to
 * exactly NAMEDATALEN bytes.  The use of a struct is historical.
 */
typedef struct nameData {
    char data[NAMEDATALEN];
} NameData;
typedef NameData* Name;
```

   NameStr 宏用于获取存储在 Name 指针指向的内存空间中的字符串。通过使用 NameStr 宏可以方便地将 Name 指针所指向的内存空间中的字符串转换为标准的 C 语言字符串。

```
#define NameStr(name) ((name).data)
```

5. 结构体 FormData_pg_attribute 及其指针 Form_pg_attribute

   FormData_pg_attribute 结构体是表示一个数据表的列的元数据信息。它是 pg_attribute 系统表的存储结构。

   【源码】src/include/catalog/pg_attribute.h：

```
CATALOG(pg_attribute,1249)    BKI_BOOTSTRAP    BKI_WITHOUT_OIDS    BKI_ROWTYPE_OID(75)
```

```
BKI_SCHEMA_MACRO
{
    Oid         attrelid;       /* OID of relation containing this attribute */
    NameData    attname;        /* name of attribute */
```

```
    /*
     * atttypid is the OID of the instance in Catalog Class pg_type that
     * defines the data type of this attribute (e.g. int4).  Information
     * in that instance is redundant with the attlen, attbyval, and
     * attalign attributes of this instance, so they had better match or
     * openGauss will fail.
     */
Oid         atttypid;
// 省略若干行
} FormData_pg_attribute;
```

FormData_pg_attribute 被定义为指向 FormData_pg_attribute 结构体的指针。

```
/* ----------------
 *      Form_pg_attribute corresponds to a pointer to a tuple with
 *      the format of pg_attribute relation.
 * ----------------
 */
typedef FormData_pg_attribute *Form_pg_attribute;
```

6. 结构体 HeapTupleData 及其指针 HeapTuple

HeapTupleData 结构体表示一行数据。它包含了一行数据的元数据信息和实际数据。下面是 HeapTupleData 结构体中的一些重要字段的说明:

- t_len：该行数据的长度（字节数）。
- t_data：该行数据的实际数据。
- t_tableOid：该行数据所在的数据表的 OID。

【源码】src/include/access/htup.h：

```
/*
 * HeapTupleData is an in-memory data structure that points to a tuple.
 *
 * There are several ways in which this data structure is used:
 *
 * * Pointer to a tuple in a disk buffer: t_data points directly into
 * the buffer (which the code had better be holding a pin on, but this
 * is not reflected in HeapTupleData itself).
 *
 * * Pointer to nothing: t_data is NULL.  This is used as a failure
 *   indication in some functions.
 *
 * * Part of a palloc'd tuple: the HeapTupleData itself and the tuple
 *   form a single palloc'd chunk.  t_data points to the memory location
 *   immediately following the HeapTupleData struct (at offset
 *  HEAPTUPLESIZE). This is the output format of heap_form_tuple and
 *  related routines.
 *
 * * Separately allocated tuple: t_data points to a palloc'd chunk that
 *   is not adjacent to the HeapTupleData.  (This case is deprecated
```

```
 *   since it's difficult to tell apart from case #1.  It should be used
 *   only in limited contexts where the code knows that case #1 will
 *   never apply.)
 *
 * * Separately allocated minimal tuple: t_data points
 *   MINIMAL_TUPLE_OFFSET bytes before the start of a MinimalTuple.  As
 *   with the previous case,this can't be told apart from case #1 by
 *   inspection; code setting up or destroying this representation has
 *   to know what it's doing.
 *
 * t_len should always be valid, except in the pointer-to-nothing case.
 * t_self and t_tableOid should be valid if the HeapTupleData points to
 * a disk buffer, or if it represents a copy of a tuple on disk.  They
 * should be explicitly set invalid in manufactured tuples.
 */
typedef struct HeapTupleData {
    uint32 t_len;           /* length of *t_data */
    uint1 tupTableType = HEAP_TUPLE;
    uint1 tupInfo;
    int2   t_bucketId;
    ItemPointerData t_self; /* SelfItemPointer */
    Oid t_tableOid;         /* table the tuple came from */
    TransactionId t_xid_base;
    TransactionId t_multi_base;
#ifdef PGXC
    uint32 t_xc_node_id; /* Data node the tuple came from */
#endif
    HeapTupleHeader t_data; /* -> tuple header and data */
} HeapTupleData;
```

GETSTRUCT 宏对于给定的 HeapTuple 指针，返回用户数据的地址。

```
/*
 * GETSTRUCT - given a HeapTuple pointer, return address of the user data
 */
#define GETSTRUCT(TUP) ((char*)((TUP)->t_data) + (TUP)->t_data->t_hoff)
```

HeapTuple 是 HeapTupleData 结构体的指针，表示一行数据。

【源码】src/include/utils/relcache.h：

```
typedef HeapTupleData* HeapTuple;
```

7. 结构体 FormData_pg_type 及其指针 Form_pg_type

FormData_pg_type 结构体定义了数据类型的元数据信息。它是 pg_type 系统表的存储结构。

【源码】src/include/catalog/pg_type.h：

```
CATALOG(pg_type,1247) BKI_BOOTSTRAP BKI_ROWTYPE_OID(71) BKI_SCHEMA_MACRO
{
NameData    typname;        /* type name */
```

```
// 省略若干行
/*
     * For a fixed-size type, typlen is the number of bytes we use to
     * represent a value of this type, e.g. 4 for an int4.  But for a
     * variable-length type, typlen is negative.  We use -1 to indicate
     * a "varlena" type (one that has a length word), -2 to indicate a
     * null-terminated C string.
     */
    int2        typlen;
// 省略若干行
} FormData_pg_type;
```

FormData_pg_type 被定义为指向 FormData_pg_type 结构体的指针。

```
/* ----------------
 *      Form_pg_type corresponds to a pointer to a row with
 *      the format of pg_type relation.
 * ----------------
 */
typedef FormData_pg_type *Form_pg_type;
```

7.4.4 输出 CREATE TABLE 相关信息

下面将在文件 heap.cpp 的函数 heap_create_with_catalog 的末尾添加代码输出 CREATE TABLE 的相关信息。函数 heap_create_with_catalog 用于在数据库中创建一个新表。该函数在数据库的系统表（例如 pg_class）中创建一个新的关系记录，并将该关系的元数据（例如列信息）存储在相应的系统表（例如 pg_attribute）中。该函数需要一系列参数，以指定创建的表的名称、模式、列信息等。这些参数包括：表名称、表模式、表列信息（包括列名、数据类型、约束等。在创建表时，heap_create_with_catalog 函数还可以用于创建相关的索引，并将表的元数据存储到数据库的系统表中。该函数返回新创建的表的元数据，该元数据可以用于后续的查询、更新和删除操作。

1. 本实验中，要添加代码的位置是**错误!未找到引用源。**节所添加代码（已注释掉）的后面，函数 heap_create_with_catalog 末尾最后一行语句“return relid;”之前。
   - 文件：src/common/backend/catalog/heap.cpp
   - 函数：heap_create_with_catalog

   添加代码：

```
/* [ DBLAB ================================================= */
Form_pg_class pg_class = new_rel_desc->rd_rel;
TupleDesc t_desc = new_rel_desc->rd_att;
const int buf_size = 512;
char info[buf_size];
int idx = sprintf_s(info, buf_size, "\n"
            "relation's object id : %u\n"
            "pg_class->relname : %s\n"
            "pg_attribute->attname : \n",
```

```
            new_rel_desc->rd_id,
            pg_class->relname.data);
for (int i = 0; i < t_desc->natts; i++) {
    if (idx + 1 == buf_size)
        break;
    Form_pg_attribute pg_attribute = t_desc->attrs[i];
    char* attname = pg_attribute->attname.data;
    Oid oid_type = pg_attribute->atttypid;
    HeapTuple tup = SearchSysCache1(TYPEOID, ObjectIdGetDatum(oid_type));
    Form_pg_type type_tup = (Form_pg_type) GETSTRUCT(tup);
    char* typname = type_tup->typname.data;
   int typlen = type_tup->typlen;
    int len = sprintf_s(&info[idx], buf_size - idx,
                "attr[%d].attname : %s typname : %s typlen : %d\n",
                i, attname, typname, typlen);
    idx += len;
}
info[idx] = '\0';
ereport(INFO, (0, errmsg("%s", info)));
/* DBLAB ] ================================================ */
```

添加代码的具体位置如图 7.1 所示。

```
3014      /* DBLAB ] ================================================================ */
3015
3016      /* [ DBLAB ================================================================ */
3017      Form_pg_class pg_class = new_rel_desc->rd_rel;
3018      TupleDesc t_desc = new_rel_desc->rd_att;
3019      const int buf_size = 512;
3020      char info[buf_size];
3021      int idx = sprintf_s(info, buf_size, "\n"
3022                  "relation's object id : %u\n"
3023                  "pg_class->relname : %s\n"
3024                  "pg_attribute->attname : \n",
3025                  new_rel_desc->rd_id,
3026                  pg_class->relname.data);
3027      for (int i = 0; i < t_desc->natts; i++) {
3028          if (idx + 1 == buf_size)
3029              break;
3030          Form_pg_attribute pg_attribute = t_desc->attrs[i];
3031          char* attname = pg_attribute->attname.data;
3032          Oid oid_type = pg_attribute->atttypid;
3033          HeapTuple tup = SearchSysCache1(TYPEOID, ObjectIdGetDatum(oid_type));
3034          Form_pg_type type_tup = (Form_pg_type) GETSTRUCT(tup);
3035          char* typname = type_tup->typname.data;
3036          int typlen = type_tup->typlen;
3037          int len = sprintf_s(&info[idx], buf_size - idx,
3038                      "attr[%d].attname : %s typname : %s typlen : %d\n",
3039                      i, attname, typname, typlen);
3040          idx += len;
3041      }
3042      info[idx] = '\0';
3043      ereport(INFO, (0, errmsg("%s", info)));
3044      /* DBLAB ] ================================================================ */
3045
3046      return relid;
3047  }
```

图 7.1 添加代码输出 CREATE TABLE 的相关信息

我们添加的这段代码是在输出一个 CREATE TABLE 语句刚刚新建的关系表的信息。变量 new_rel_desc 的类型是 Relation。

- 首先，定义了一个结构体指针 pg_class，指向 new_rel_desc 的 rd_rel 字段。然后，定义了一个 TupleDesc 类型的指针 t_desc，指向 new_rel_desc 的 rd_att 字段。
- 接着，定义了一个大小为 512 字节的 char 数组 info，并用 sprintf_s 函数向 info 数组中写入了一些信息，其中包括 new_rel_desc 的 rd_id，以及 pg_class 的 relname。
- 接下来，进入了一个 for 循环，循环次数为 t_desc->natts，其中 natts 为 t_desc 指向的 TupleDesc 结构体中存储的该表的列数量。在循环中，首先定义了一个 Form_pg_attribute 类型的指针 pg_attribute，指向 t_desc 的 attrs 数组的第 i 个元素。
- 接下来，定义了一个 char 指针 attname，指向 pg_attribute 的 attname 字段。然后，使用 pg_attribute 的 atttypid 字段作为 TYPEOID 的参数，调用 SearchSysCache1 函数，查询出了一个 HeapTuple 类型的指针 tup。
- 随后，定义了一个 Form_pg_type 类型的指针 type_tup，指向 tup 中的内容。然后，使用 type_tup 的 typname 字段、typlen 字段，以及 i，调用 sprintf_s 函数，向 info 数组写入了一些信息。
- 最后，使用 ereport 函数，输出 info 数组中的内容。

2. 编译添加的代码并安装，启动数据库服务器，gsql 客户端连接数据库服务器，输入 7.4.1 节中的 CREATE TABLE 语句，创建 users 表。gsql 的执行结果如下：

```
openGauss=# CREATE TABLE users
openGauss-# (
openGauss(# u_id varchar(20),
openGauss(# u_passwd varchar(20),
openGauss(# u_name varchar(10),
openGauss(# u_idnum varchar(20),
openGauss(# u_regtime timestamp
openGauss(# );
INFO:
relation's object id : 57354
pg_class->relname : users
pg_attribute->attname :
attr[0].attname : u_id typname : varchar typlen : -1
attr[1].attname : u_passwd typname : varchar typlen : -1
attr[2].attname : u_name typname : varchar typlen : -1
attr[3].attname : u_idnum typname : varchar typlen : -1
attr[4].attname : u_regtime typname : timestamp typlen : 8


CREATE TABLE
```

可以看到，我们添加的代码被成功执行。通过访问 CREATE TABLE 语句新创建的 users 表的 Relation 中的 pg_class 和 pg_attribute，获得并输出了表的信息和表中各属性列的信息，包括：表的对象 id、表的名称、各属性的名称、各属性的类型及类型长度。

3. 将添加代码注释掉。退出 gsql，关闭数据库服务器。重新编译和安装，使 openGauss 中去掉添加代码的影响。

如果在执行 CREATE TABLE 语句时数据库服务器崩溃或 gsql 连接断开，或没有实现预期的输出效果，是由于编写代码过程中引入了错误（bug）。此时，通过使用第 3 章中实践过的单步调试方法对添加代码进行调试（debug），以排除错误。通过“编辑——编译——测试——调试”这样的步骤循环，直到成功执行为止。

7.5 实验结果

【请按照要求完成实验操作】

1. 完成实验步骤 7.4.1，创建 user 行存表和列存表。

2. 完成实验步骤 7.4.2，就其中给出的系统表属性，画出系统表 pg_class、pg_attribute、pg_type 之间的关系的 ER 图。

3. 完成实验步骤 7.4.4，验证添加代码的执行效果，查看输出的表信息和表的属性列信息。

7.6 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。】